Improving Optimistic Concurrency Control Through Transaction Batching and Operation Reordering

Bailu Ding
Microsoft Research
badin@microsoft.com

Lucja Kot
Cornell University
lucja@cs.cornell.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 a framework to incorporate batching at multiple stages of transaction execution for OLTP systems based on optimistic concurrency control. Storage batching enables reordering of transaction reads and writes at the storage layer, reducing conflicts on the same object. Validator batching enables reordering of transactions before validation, reducing conflicts between transactions. Dependencies between transactions make transaction reordering a non-trivial problem, and we propose several efficient and practical algorithms that can be customized to various transaction precedence policies such as reducing tail latency. We also show how to reorder transactions with a thread-aware policy in multi-threaded OLTP architecture without a centralized validator.

In-depth experiments on a research prototype, an open-source OLTP system, and a production OLTP system show that our techniques increase transaction throughput by up to 2.2x and reduce their tail latency by up to 71% compared with the start-of-the-art systems on workloads with high data contention.

1. INTRODUCTION

Transaction processing is a fundamental aspect of database functionality, and improving OLTP system performance is a key research goal. The throughput of OLTP systems can be increased through *batching* of operations; specifically, some component can buffer a number of operations as they arrive then process them as a group [22, 15, 25]. Batching can improve system performance for several reasons. First, at the networking layer, it increases the communication efficiency by packing messages [16, 22]. Second, it amortizes the cost of system calls by condensing multiple requests into a single one, as in group commit [15, 25]. Third, it reduces the number of requests by discarding duplicate or stale requests, such as writes to the same record [19]. All of those are optimizations based on low-level techniques, and they do not take the semantics of the system into account.

We propose to embrace semantic batching as a core design principle throughout transaction execution for OLTP systems with optimistic concurrency control (OCC) [34]. OCC is a popular concurrency control protocol due to its low overhead in low-contention settings [2, 6, 8, 9, 10, 13, 17, 42, 43, 36]. However, it has been shown that OCC wastes resources when conflicts are frequent [3]. We show how semantic batching (and associated reordering of transactions) can reduce the number of conflicts, improve throughput and latency, and allow us to use OCC with higher-contention workloads.

Figure 1 shows a prototypical architecture of a modern, loosely coupled OCC-based transaction processing system with a separation of compute and storage such as Centiman [16]. The system consists of three components: processors, storage, and one or more validators. 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, where the validator checks if there are conflicts with a previously committed transaction. 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.

OCC presents unique opportunities for batching because the final serialization order of transactions is only decided at commit time in the validator. There are three opportunities to apply semantic batching. The first is the processor in the transactions' read phase, where transaction requests can be batched before execution. Recent work in the context of locking-based protocols batch transactions and serialize them before execution to reduce overhead [18, 40, 48]; these techniques can be adapted and applied in OCC as well.

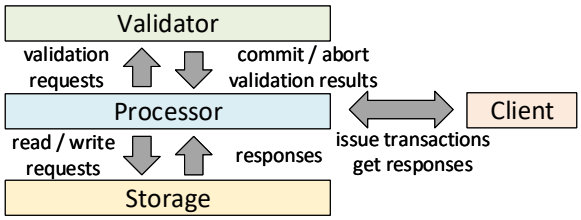


Figure 1: OCC system architecture

The second possible place is the validator. The validator can batch validation requests and then select a validation order that reduces the number of conflicts and aborts. Assume again two transactions t and t' where t reads x, and t' writes x after t's read. Without batching, if t' arrives at the validator before t and commits, t will fail. With batching, if t and t' are in the same validation batch, we can serialize t before t' (assuming no other dependencies between the transactions), and we can 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 joint batches as they arrive. If a batch contains read and write requests for the same object, the system can apply all the writes first in their serialization order and then process the reads. Prioritizing writes over reads is always optimal as this reduces the number of aborts later in the validator 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. We explore the benefits of transaction batching and reordering in optimistic concurrency control with backward validation, focusing on storage and validator batching.

• Our first contribution is to show how to integrate batching and reordering throughout the lifetime of a transaction to enhance OCC-based protocols. 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 semantic batching. We introduce intra-batch storage reordering and intra-batch validator reordering, and we show that selecting the optimal transaction ordering at the validator batch is NP-hard. (Section 3).

• Our second contribution is two practical classes of greedy algorithms for validator reordering that balance abort rates and reordering overheads. We also extend these algorithms to weighted versions that can incorporate policies such as transaction priorities (Section 4).

• In a detailed experimental study of in a prototype system, as well as on top of a state-of-the-art OLTP system and a commercial database system, we show that batching and reordering always increases transaction throughput and, surprisingly, also reduces transaction latency, especially the tail latency. For workloads with high data contention, batching and reordering can improve the throughput by up to 2.2x and reduce the tail latency by up to 71% compared with the state-of-the-art OLTP systems (Section 5).We further show how to reduce the amortized overhead of reordering through parallelism (Appendix A and B).

2. BACKGROUND

We use the term optimistic concurrency control (OCC) to refer to the (traditional backward) validation based OCC protocol introduced by Kung [34]. 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 $WS(i) \cap RS(j) = \emptyset$ and $WS(i) \cap WS(j) = \emptyset$

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 [34], but there has been much progress on OCC over the last decades to scale OCC-based OLTP systems that decouple the two phrases, resulting in out-of-order write requests. We handle such 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]

The versioned datastore also provides an easy way to determine whether $T(j)$ and $T(i)$ overlap temporally; 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 [16]. For the ease of explanation, we assume the validation phase is sequential, however all aspects of validation can be easily parallelized as discussed in Appendix A.

[1]We assume full serializability here and in the remainder of this paper, although our ideas can easily be extended to snapshot isolation using multi-version storage.

3. OVERVIEW

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 reordering opportunities: Storage batching and validator batching.

We first make a conceptual distinction between two types of transaction aborts: intra-batch and inter-batch aborts. Assume transaction T abort due to its conflict with T'. If T and T' are in the same batch, we call the resulting abort of T' an *intra-batch abort*; otherwise, we call it an *inter-batch abort*. In this work, we focus on managing intra-batch aborts by strategically reordering the batched requests at storage and validator. Reducing inter-batch aborts is complementary to our effort.

Our approach is agnostic to isolation levels as long as the reordering respects the corresponding definitions of conflicts for write-read, write-write, and read-write dependencies. In the remainder of this paper, we describe how to batch and reorder for serializability, but our techniques can be adapted accordingly to other isolation levels, such as snapshot isolation.

3.1 Reordering at the Storage

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

We implement this idea as follows. We buffer a number of read and write requests from transactions into batches. As a batch of requests arrives at the storage layer, 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.[2] Next, we handle all the read requests for the same object. This strategy works very well 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.

3.2 Reordering at the Validator

In validator batching, we buffer transaction validation requests at the validator as they arrive. Once a batch has been collected, the validator can reduce intra-batch aborts by selectively aborting transactions while 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 transaction aborts, i.e., we maximize the number of transactions in each batch that commit. Alternatively, 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.

We define the problem of intra-batch validator reordering (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 previously 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 subset $B' \subseteq B$ of transactions to abort such that there is a serialization order $\prec$ for the the remaining transactions $B \setminus B'$, and all of the transactions in $B \setminus B'$ commit when validated in this order. We then process 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 $\prec$. Note that there is always a trivial solution to any IBVR instance, namely to abort all transactions but one. This solution is 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. A simple objective function is the size of B' (the fewer transactions to abort the better), and more complex functions can take transaction priorities or the tail latency of transactions into account. We call this objective function a *policy* P.

How do we compute B'? We observe that every batch B of viable transactions has an associated *dependency graph* G, a directed graph whose nodes are the transactions in B and whose edges are read-write dependencies. 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 a topological sort.

If G is not acyclic, we can model this problem as an instance of the feedback vertex set problem. 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. 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. Unfortunately, it is NP-hard and APX-hard [31, 32], and it is still an open problem whether there exists any constant-factor approximation. We propose several practical algorithms for finding a FVS in the next section.

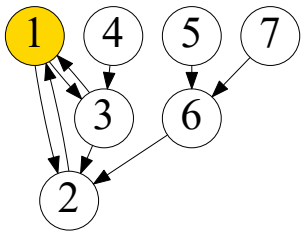


Figure 2: An example of a directed graph; node 1 forms a feedback vertex set.

[2]Recall that we assume full serializability; this may be different for snapshot isolation.

```
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 ← SelectVertexByPolicy(S, P)
14     S' ← GetGraphAfterVertexRemoval(S, v)
15     return v ∪ GreedySccGraph(S', P)
```

Algorithm 1: SCC-based greedy algorithm

```
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 ← ∅
3      G ← trim(G)
4      while G ≠ ∅ do
5          if G.size < k then
6              V ← V ∪ GreedySortGraph(G, P, 1)
7              break
8          end
9          Q ← QuickSelectVertexByPolicy(G, P, k)
10         for i = 1; i ≤ k; ++i do
11             V ← V ∪ Q[i]
12             G ←
                   GetGraphAfterVertexRemoval(G, Q[i])
13         end
14         G ← trim(G)
15     end
16     return V
```

Algorithm 2: Sort-based greedy algorithm

4. VALIDATOR BATCHING

All our IBVR algorithms begin by constructing the dependency graph G. 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. We implement this by creating a hash table from the write sets and probing it with the read sets. Since a read in t can potentially conflict with all the other transactions in the batch, the time complexity to probe the hash table for a single read is $O(|B|)$, where $|B|$ is the size of the batch. The complexity of building G is thus $O(|B|^2 + |R| + |W|)$, 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 the nodes that have no incoming edges and/or no outgoing edges since such nodes cannot participate in any cycles.

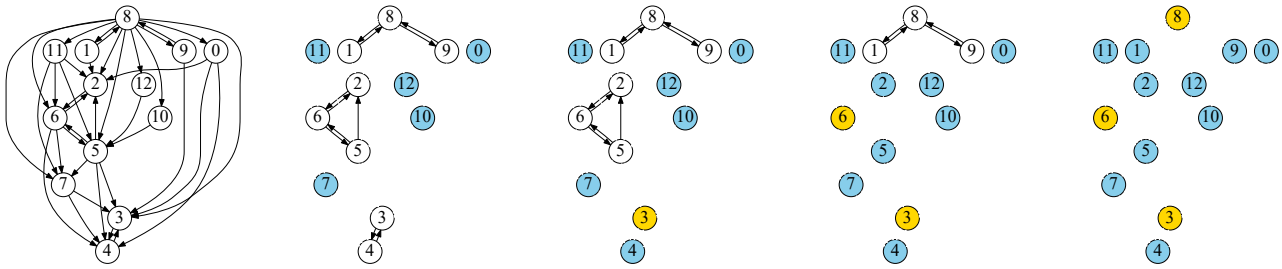
4.1 Algorithms

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 [46].

Nodes in SCCs of size one cannot belong to any cycle. For a SCC that contains more than one node, we choose a vertex to remove according to a *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 13). It removes v from S and recursively calls *GreedySccGraph* on the remaining graph S' (line 14 - 15). Finally, it returns the union of all the FVSs obtained in processing S (line 15). When the top-level procedure $GreedySccGraph(G, P)$ has processed all the SCCs of G, it returns the union of the FVSs obtained (line 8). Trimming and updating the graph after removing a node take $O(|V| + |E|)$ per graph in total, since each node / edge can be removed only once. In the worst case, e.g., in a fully connected graph, we may only remove one node per iteration. Since SCC takes $O(|V| + |E|)$ per iteration, the time complexity of this algorithm is $O(|V|(|V| + |E|))$. The policy P is at the heart of the algorithm, and it ranks vertices which are likely to be included in a desirable FVS highly. We discuss possible policies in Section 4.2.

An example. Figure 3 shows an instance of the SCC-based greedy algorithm that aims at minimizing the size of FVS, with a policy P selecting the node with the largest product of its in-degree and out-degree, i.e., `prod-degree`. 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.

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



(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. Blue nodes are trimmed during the algorithm, and yellow nodes form the FVS.

according to a policy P, and includes the k top-ranked 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 start by trimming 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 and select top ranked k nodes into a queue Q using P with the Quickselect algorithm [29], and include the k nodes in V (line 9-13). After removing the selected nodes from G, we trim the remaining graph again (line 14). We repeat this procedure until the graph is empty (line 4). As with the previous algorithm, updating the graph after removing a node and *trim* takes $O(|V|+|E|)$ in total. Updating the weights of the remaining nodes takes $O(|V|)$ per iteration. The Quickselect algorithm [29] has an amortized time complexity of $O(|V|)$. In the worst case, it will take $O(|V|/k)$ iterations to terminate. So the overall time complexity is $O(|V|^2/k+|V|+|E|)$. This algorithm has smaller time complexity than the SCC-based greedy algorithm, and we can increase k to further trade accuracy for shorter runtime. As we will see in Section 5, it produces results of comparable accuracy to the SCC-based algorithm in practice.

Example (Continued). 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.

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. 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.

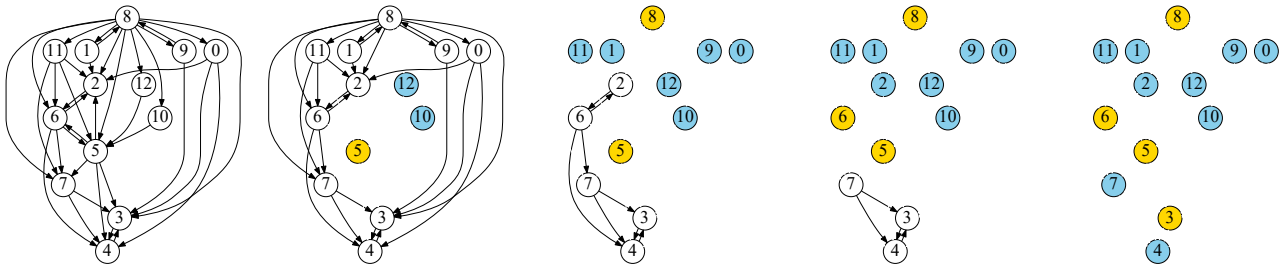
4.2 Policies

Policies are of utmost 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 discuss three kinds of policies for different performance objectives and system architectures.

Minimize the number of aborts. 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, based on 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 [14]. 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.

Minimize tail latency. More sophisticated policies are possible if the system is optimizing for 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 much less likely to enter the FVS and have a higher chance of committing. Alternatively, we can also devise policies that combine the 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. In business applications, the monetary value of different transactions varies. Optimizing for maximal monetary value, we can design policies that favor more valuable transactions. For example, we can customize the policy to always add a transaction with the lowest value to the FVS until the resulting dependency graph is acyclic.

Reduce inter-thread conflicts. Many recent OCC-based OLTP systems use a decentralized architecture [38, 49, 55, 33], where there is no centralized validation. Each transaction is scheduled to a dedicated *thread* and processed by this thread synchronously for its entire lifetime. Since a transaction is executed synchronously, it can only conflict with transactions executed by other threads.



(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. Blue nodes are trimmed during the algorithm, and yellow nodes form the FVS.

In such an architecture, while batching operations across transactions does not apply, we can assign transactions to threads to reduce inter-thread conflicts. Intuitively, for a batch of incoming transactions, we schedule conflicting transactions to execute on the same thread where they are processed serially. Because conflicting transactions access the same objects, reordering also improves caching within a thread.

We extend our sort-based algorithm to a *thread-aware policy* which assigns transactions thread by thread. We batch a number of transactions and assign the same number of transactions to each thread.[3] At each thread, we compute the weight of a transaction first based on the number of conflicts to the set of transactions T in the batch that have already been assigned to this thread, then the number of object accesses shared between this transaction and T, and finally its number of conflicts to the unassigned transactions in this batch. For each thread, our sort-based algorithm will first pick the transaction with the most conflicts against the remaining transactions, and then it iteratively updates the weights of the transactions left to pick the next one with the highest weight. This policy greedily puts conflicting transactions on the same thread to minimize conflicts across threads; as a side effect, it tries to assign transactions accessing the same objects to the same thread, resulting in better caching. As we will see later in our experiments (Section 5.6), this simple policy significantly improves throughput and reduces tail latency.

5. EVALUATION

In our evaluation, we wanted to understand the effect of batching and reordering, the performance of our validator reordering algorithms and policies, and the impact of parallelization at the validator. We wanted to answer the following questions:

1. How well do our validator reordering algorithms from Section 4.1 perform? How do these algorithms affect the end-to-end system performance? (Section 5.2)

2. How does storage and validator batching and reordering affect system performance? (Sections 5.3 and 5.4)

3. How do different policies from Section 4.2 impact alternative performance goal? (Section 5.5)

4. How do we incorporate our techniques on top of a state-of-the-art OLTP system and a commercial database, and what is the resulting performance? (Sections 5.6 and 5.7)

Table 1 summarizes the settings we evaluated and describes the tradeoff encapsulated in each parameter. We also describe parallelization of our algorithms in detail in Appendix A and show the resulting benefits in Appendix B.

5.1 Experimental Setup

We first describe the experimental setup for our research prototype where we isolate the impact of different parameters in our algorithms. Section 5.6 and 5.7 describe our experiments with an open-source OLTP system and a commercial database system; both of these sections will describe their own experimental setup respectively.

Our research prototype system architecture has four components: Transaction clients, processors, storage, and validator. The components communicate through consumer-producer queues. The transaction client continuously produces new transactions until the system reaches the maximum permitted concurrency level. The processor acts as a transaction coordinator and *multiplexes* multiple transactions in parallel. It receives transaction requests from transaction clients, sends read/write requests to the storage, sends validation requests to the validator, and replies to the transaction client. It also restarts aborted transactions so that it only communicates commit decisions to the transaction client.

The processor is *non-blocking*: It continuously processes requests from its consumer-producer queue without waiting for the response of the request. For example, when it receives a transaction request from a transaction client, it sends read requests on behalf of this transaction to the storage's consumer-producer queue, and then it continues to process the next request in its queue without waiting for the response from the storage. This asynchronous processing allows the processor to efficiently multiplex many transactions in parallel to improve its throughput.

The storage continuously processes read and write requests. When storage batching is enabled, the storage buffers requests into batches, and uses the strategy described in Section 3.1 to first execute all the write requests and then all the read requests in the batch.

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

[3]More sophisticated assignments are possible, but we leave them for future work.

Table 1: Summary of settings and trade-offs

Setting	Trade-off
Batch size	Larger batch sizes give more flexibility of reordering but can increase transaction latency.
FVS algorithms	The sort-based algorithm is cheaper but includes more transactions in the FVS.
Policy: Minimize aborts	Reorder transactions to reduce the number of aborts.
Policy: Minimize tail latency	Reorder transactions to reduce tail latency at the cost of slightly increased aborts.
Policy: Reduce inter-thread conflicts	Reorder transactions to achieve thread locality for decentralized system architecture.

in-memory hash table, until these writes are overwritten by later updates. We have decoupled the validator into three components as described in Section 3.2. 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 all the transactions in the batch against the current state of the validator cache, reorders the transactions, and sends ordered transactions to the validation workers. A validation worker processes batches of transactions one by one. It validates transactions against the current validator cache and applies updates from committed transactions to the cache based on the transaction serialization order. When batching is enabled, the validator collects the requests into a batch as they arrive, and runs one of the algorithms from Section 4.1 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. By default, the validator uses the sort-based greedy algorithm with the `prod-degree` policy and multi factor value 2.

We have parallelized the transaction generation, the storage, and the transaction reordering at the validator. By default, two transaction clients 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 3.1. In the validator, we first introduced pipeline parallelism by processing the three subcomponents concurrently. Since we observed that the transaction reordering consumes much more time than the other components, we use multi-threading and four transaction reordering workers by default.

We implemented our prototype 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, implemented 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. We choose the concurrency level and batch size empirically to properly load the system.

The baseline configuration (*base*) represents the system running with both storage and validator batching turned off. We optimize the code path for transactions without batching to avoid the overhead from batching and reordering, including skipping packing transactions into batches as well as the reordering workers in validator. We further add a batch mode (*batch*) to separately measure the effects of batching and reordering, where requests are batched at both storage and validator, but no reordering is performed. The batch mode has improved performance over *base*, because it benefits 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. The standard deviation was not significant in any of the experiments, so we omit the error bars for clarity of presentation. We report throughput (the number of committed transactions per second), average latency, and percentile latency to show tail latencies.

5.2 Validator Reordering Algorithms

We first investigate the performance of the feedback vertex set algorithms from Section 4.1 for their accuracy and running time. We run the algorithms on graphs constructed as described in Section 4, 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), as well as 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 constructed from a batch of transactions, *random_3* runs *random* 3 times and returns the smallest FVS, mitigating the effect of bad random choices.

Figure 5 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.

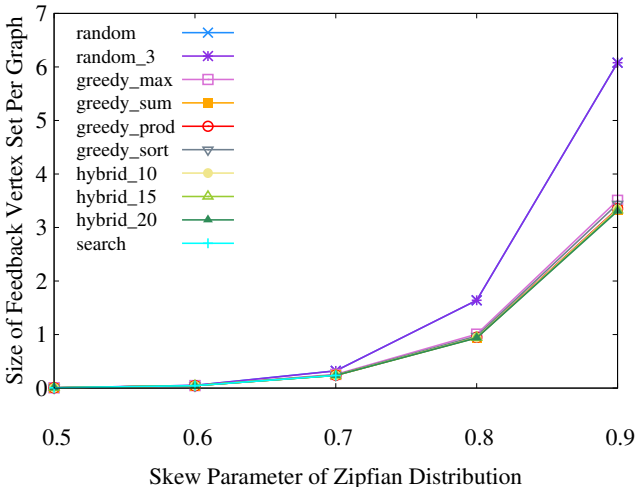


Figure 5: Size of FVS per graph

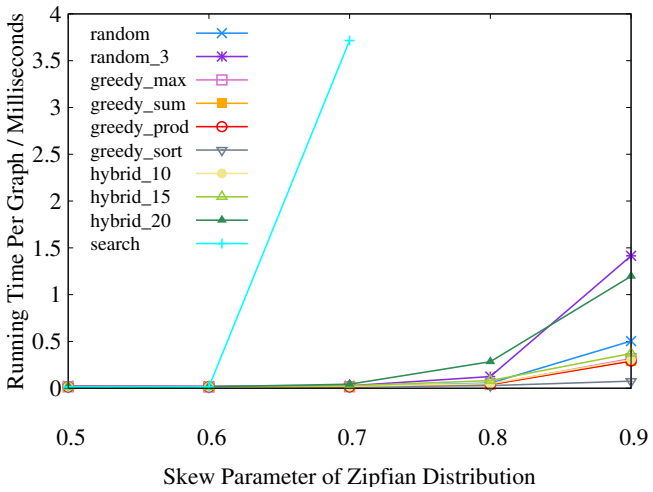


Figure 6: Running time of finding FVS

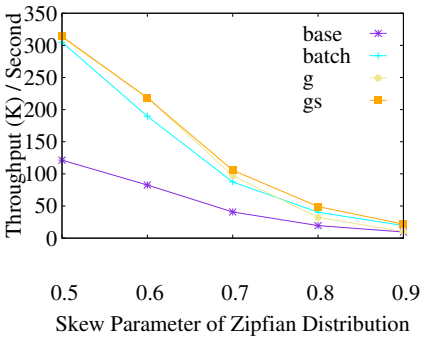


Figure 7: Throughput with SCC-based and sort-based greedy algorithms

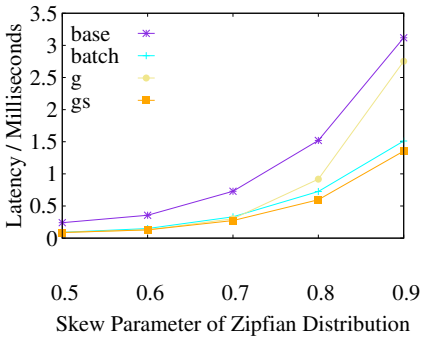


Figure 8: Average latency for greedy algorithms

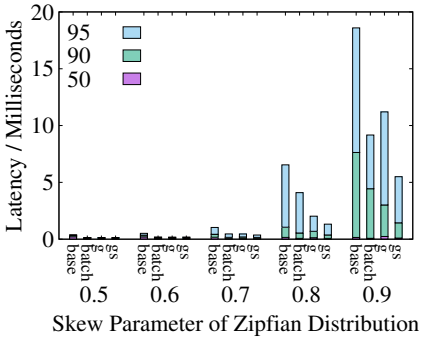


Figure 9: Percentile latency for greedy algorithms

Figure 6 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 iterations to find FVS. 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 the running time of these algorithms by 74%.

We compare the end-to-end performance of the best SCC-based algorithm (*greedy_prod*) against the sort-based greedy algorithm. Figure 7 and 8 show the throughput and the average latency of the system with *greedy_prod* (*g*) and *greedy_sort* (*gs*). In both cases, storage batching is enabled. The *base* 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 9 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 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 (*base*), batching without reordering (*batch*), storage only batching with reordering (*sr*), validator only batching with reordering (*vc*), and both storage and validator batching with reordering (*srvc*).

Figures 10, 11, and 12 show the throughput, the average latency, and the percentile latency of different system modes under a variety of data skew parameters. Overall, using batching with reordering at the storage and/or the validator consistently leads to significant improvements in throughput by up to 2.7x. In addition, storage and validator reordering consistently improve the throughput to up to 1.3x as compared to *batch*. Moreover, validator reordering significantly reduces the average latency by up to 67% and

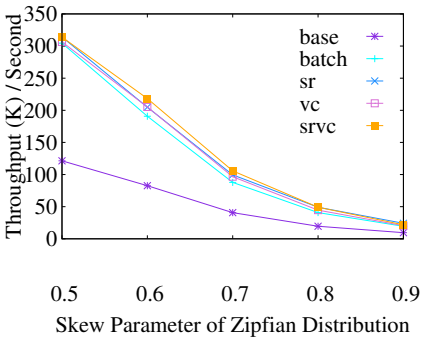


Figure 10: Throughput under workloads of Zipfian distribution

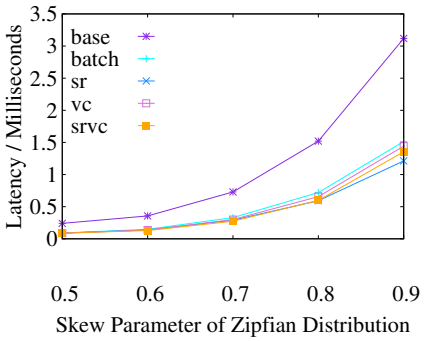


Figure 11: Average latency under workloads of Zipfian distribution

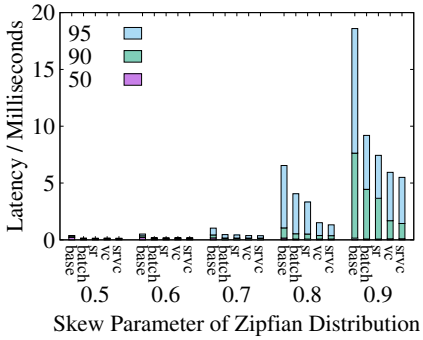


Figure 12: Percentile latency under workloads of Zipfian distribution

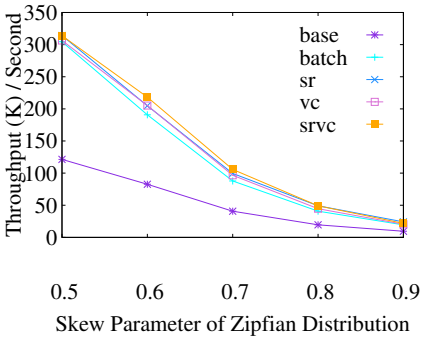


Figure 13: Throughput with Small Bank benchmark

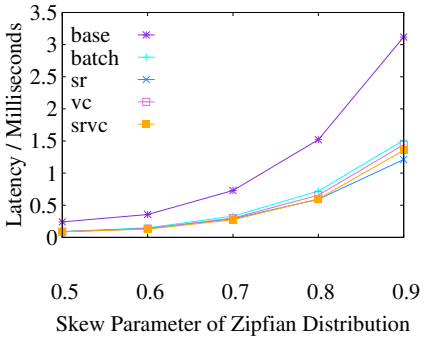


Figure 14: Average latency with Small Bank benchmark

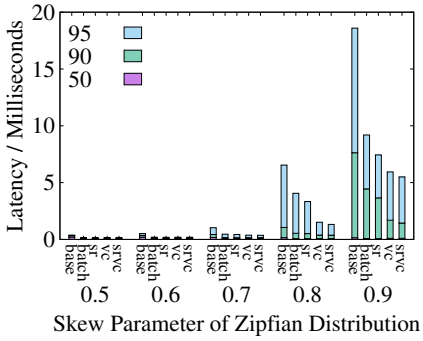


Figure 15: Percentile latency with Small Bank benchmark

the tail latencies by up to 82% as compared to *base*, and by up to 25% and 70% as compared to *batch*.

When the data contention is extremely high (i.e., skew factor 0.9), the number of intra-batch conflicts that cannot be resolved by validator reordering increases. Validator reordering is slower due to denser graphs, while bringing less benefit. Thus, the best throughput in this case is achieved by using storage only batching with reordering (*sr*).

5.4 Small Bank Benchmark

The Small Bank benchmark [4] 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 funds from one customer to another, 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 13, 14, 15 show the throughput, the average latency, and the percentile latency of transactions. Overall, batching and reordering improved the throughput by up to 3.1x, reduced average latency by up to 68%, and tail latency by up to 62%. Storage and validator reordering can always improve the throughput and reduce the latency on top of batching, confirming our findings in Section 5.3.

5.5 Reducing Tail Latency

In this experiment, we explore validator reordering with more sophisticated policies as discussed in Section 4.2. Our baselines are the `prod-degree` policy that maximizes the number of commits (*mc*) as well as no batching (*base*) and batching without reordering (*batch*). Our first tail-latency aware policy (*rct*) favors transactions that have already been aborted and restarted. When choosing a node to include in the FVS, it chooses the node with the smallest number of restarts, breaking ties using `prod-degree`. Our second latency-aware policy (*rdeg*) combines the number of restarts and the incoming/outgoing degrees of a transaction into a weight. 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 picks 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. Figures 16 and 17 show the throughput and the average latency. The impact of tail-latency aware policies on transaction throughput and average latency are negligible as compared to when we maximize the number of commits (*mc*). Figure 18 shows the tail latencies above 90%. While our first latency-aware policy *rct* performs similar to *mc*, the more sophisticated policy *rdeg* consistently performs significantly better than all the others, and it reduces the tail latency by up to 86%.

Thus, with latency-aware policies, we effectively reduce transaction tail latencies without sacrificing either the throughput or the average latency.

5.6 Experiments with Cicada

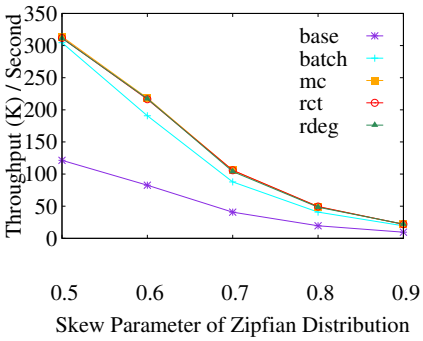


Figure 16: Throughput with tail latency optimized policies

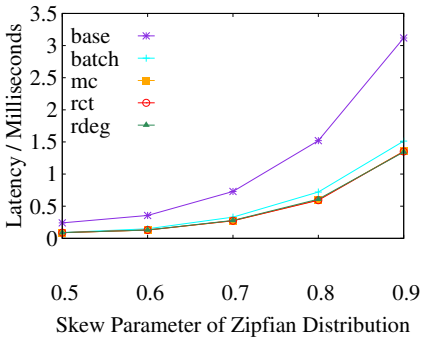


Figure 17: Average latency with tail latency optimized policies

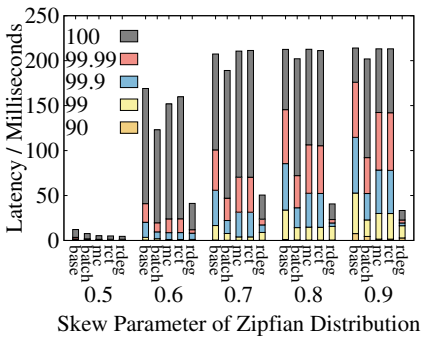


Figure 18: Percentile latency with tail latency optimized policies

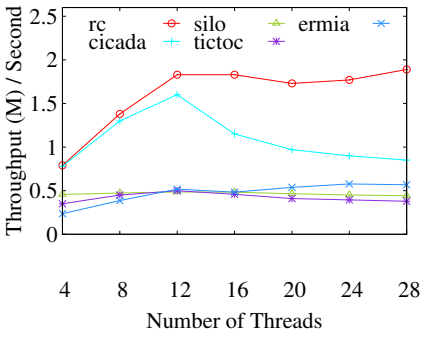


Figure 19: Throughput of YCSB with skew factor 0.99

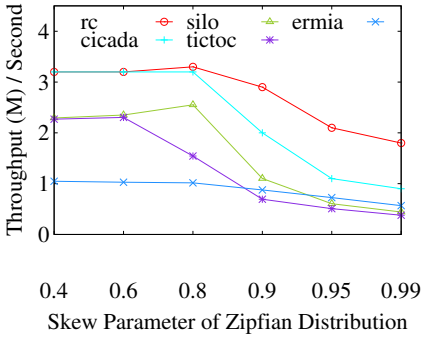


Figure 20: Throughput of YCSB with 28 threads

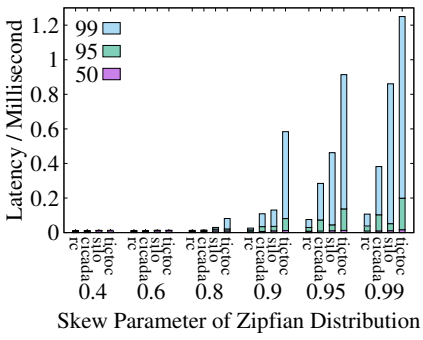


Figure 21: Percentile latency of YCSB with 28 threads

We integrated the idea of transaction batching and reordering into Cicada, an open-source OLTP system [38]. Cicada represents an important class of in-memory OCC-based OLTP system architectures. In Cicada, every thread executes its transactions independently, and there are no centralized threads for transaction validation or storage access. We integrated our techniques with its architecture by batching and reordering the transactions in a preprocessing step using the thread-aware policy as described in Section 4.2.

We compare our strategy (*rc*) with a number of the-state-of-the-art OLTP systems, including Cicada (*cicada*) [38], Silo (*silo*) [49], ERMIA (*ermia*) [33], and TicToc (*tictoc*) [55], using the same YCSB benchmark configuration as in the original Cicada paper [38]. Each transaction consists of 16 requests generated from a Zipfian distribution, with 50% read and 50% RMW. The database consists of 10 million keys, each with 100-byte payload.

We run the experiment on a machine with two Intel Xeon Processor E5-2690 v4 CPUs (each with 14 physical cores) and 256GB of DRAM. We pin threads to different cores and use NUMA-aware memory to allocate hugepages.

Figure 19 shows the throughput of different systems with a write-intensive, highly-skewed workload, varying the number of threads. Figures 20 and 21 show the throughput and tail latency with the maximal number of threads (i.e., 28) under workloads with different degrees of data contention. ERMIA is not shown in Figure 21 since it does not report percentile latencies.

As the degree of data contention increases, either with more threads or a more skewed data distribution, reordering outperforms the other systems with higher throughput and lower tail latency. With 28 threads and skew factor 0.99, reordering improves throughput by up to 2.2x and reduces 99% latency by up to 71% compared with the next best.

In this experiment, we configure our strategy with a batch size of $4 \times num\_threads$ and assign 4 transactions to each thread within a batch after reordering, i.e., a transaction is never reordered across batch boundaries. The reordering has low overhead, which is up to 5x in throughput compared with transaction processing.[4].

In practice, many applications and services use a middle tier, such as web servers, to maintain business logic and to serve as an arrival layer for transactions from clients before submitting them to the data tier. The reordering and batching can happen in the web servers as a preprocessing step before submitting the transactions to the data tier.

5.7 Experiments with DBMS-X

In our final experiment, we implemented the idea of transaction batching and reordering on top of a commercial OLTP system, called DBMS-X. DBMS-X is a high performance OLTP engine using optimistic concurrency control. Upon receiving transactions, it processes transactions concurrently with a first-come-first-served order.

We incorporated transaction batching and reordering to DBMS-X at the client side. We implemented validator reordering for the transactions batched at the middle tier – before submitting them to the database server. Since the

[4]The reordering can be further parallelized as we discuss in Appendix B

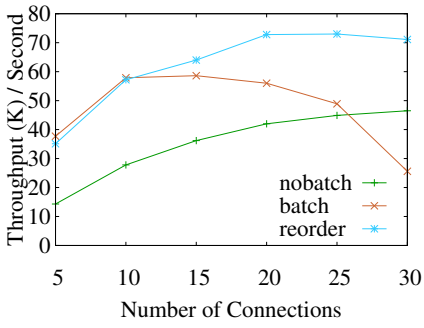


Figure 22: Throughput of DBMS-X with SmallBank benchmark

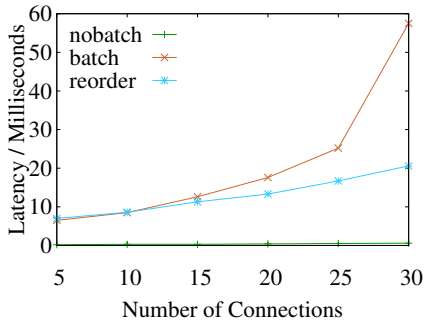


Figure 23: Average latency of DBMS-X with SmallBank benchmark

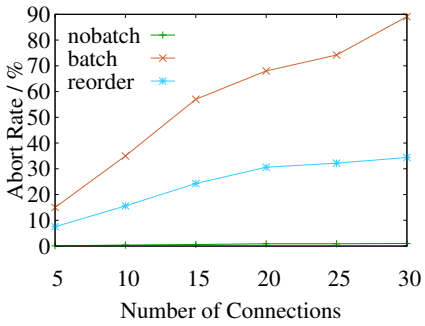


Figure 24: Abort rate of DBMS-X with SmallBank benchmark

transactions haven't started executing and their read timestamps are not yet available, we conservatively assume that all the transactions in a batch read from the same snapshot of the database. We analyze the potential conflicts between the transactions and then reorder them with our policy to minimize the number of aborts. The transactions excluded from the batch, together with future incoming transactions, will be included in the next batch for reordering.

We issue transactions to DBMS-X using JDBC. A connection to DBMS-X can send transaction statements individually or in batch with JDBC calls, where the call returns after all the transactions are processed (either commit or abort). As DBMS-X receives a batch of transaction statements from a connection, it can execute them concurrently. Batching transaction statements reduces the overhead of communication and can increase the concurrency level of DBMS-X.

We use the SmallBank benchmark with a Zipf skew of 0.9 as a high-data-contention scenario. We compare transaction batching and reordering (*BatchReorder*) with two baselines: no batching (*NoBatch*) and batching without reordering (*Batch*). In *NoBatch*, we submit transactions to DBMS-X one at a time. In *Batch* and *BatchReorder*, we batch transactions before sending them to DBMS-X. We choose a batch size of 50, which gives reasonable throughput for this workload.

Figures 22, 24, and 23 show the resulting throughput, latency, and the abort rate when we increase the number of database connections. When we do not batch transactions, the number of concurrent transactions is small and the throughput is low. As a result of this low concurrency level, the chance of conflicts is low. Thus, both abort rate and transaction latency are low.

As we send transactions in batches, the throughput increases dramatically. However, as the load continues to increase, the system runs into data contention. This further leads to resource contention due to restarts. Thus, both abort rate and the latency rise significantly. When we both batch and reorder transactions, the performance improves on all metrics: Peak throughput increases by 1.25x, throughput increases by up to 3.1x, latency is reduced by up to 66%, and the abort rate drops by up to 62%. In addition, the system performance degrades much more gracefully with increasing load.

5.8 Summary

The main practical takeaways from our experiments are as follows:

(1) The simple sort-based FVS greedy algorithm works really well in practice, and it reduces the running time of the SCC-based greedy algorithms by 74%, leading to the best overall system performance.

(2) Batching and reordering increases throughput by up to 2.7x, and reduces average and percentile latency by up to 67% and 82%. Batching by itself already improves throughput significantly, and adding reordering on top of batching consistently improves throughput by up to 1.3x. In addition, reordering significantly reduces average latency and tail latency by up to 25% and 70%. It is always beneficial to use storage reordering. Validator reordering consistently improves the tail latency but can hurt throughput and average latency when data contention is extremely high.

(3) In the Small Bank benchmark, batching and reordering improves the throughput by up to 3.1x, and it reduces the average and percentile latency by up to 68% and 62%.

(4) For alternative reordering policies at the validator, prioritizing transactions with a combination of the degree of a transaction in the dependency graph and its number of restarts reduces tail latency by up to 82%, without sacrificing either the throughput or the average latency.

(5) Integrating our techniques to a high performance OCC-based OLTP system shows that, compared with the-state-of-the-art OLTP systems, the thread-aware reordering policy improves transaction throughput by up to 2.2x and reduces the 99% latency by up to 71% compared with the next best under write-intensive YCSB workload.

(6) Integrating the idea of transaction batching and reordering to a commercial database system shows that, under high data contention, our techniques increase the throughput by up to 2.8x and reduce the average latency by up to 66%. This indicates that there is much room for performance improvements using our techniques even for mature database systems.

6. RELATED WORK

We discussed OCC and its applications in Section 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 either concurrency level and/or data contention are high [20, 5]; this has particular impact on OCC [3]. Hybrid approaches combine OCC and locking to limit the number of transaction restarts [47, 54]. The problem can also be addressed by adjusting the concurrency level adaptively, limiting the

number of arriving transactions and/or using an exponential backoff for aborted transactions [28, 38]. Transaction chopping reduces contention by partitioning transactions into smaller pieces and executing dependent pieces in a chained manner [40, 45, 53]. Follow-up work explores other ways to analyze the access patterns of transactions to expose intermediate transaction state at a fine-grained level [51]. It is also possible to reduce conflicts by executing transactions at heterogeneous isolation levels [52, 53] or using a mix of optimistic and pessimistic concurrency control protocols [50]. While we also address the problem of reducing conflicts under data contention, our batching and reordering techniques are different from and complementary to previous work.

Transaction scheduling. The dynamic timestamp assignment technique assigns each transaction a timestamp interval and flexibly picks the commit timestamp from the interval [7]. A similar technique can be used to optimize read-only transactions in distributed asynchronous OCC [16]. This approach can be extended to dynamically update the timestamp intervals of live transactions while committing a different transaction [11]. Recent work also proposes lazy timestamp assignment to reduce conflicts [55]. Dynamic timestamp assignment is compatible with our batching.

Transaction scheduling has also been studied in real-time databases, where urgent or high value transactions are prioritized [27]. 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 [26, 35, 37]. There are also systems that use locking and preemption [1], as well as hybrid optimistic/pessimistic methods [30, 39]. 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 [48, 40, 18]. These are complementary to our work. Our approach is more flexible as it allows reordering at multiple stages in transaction execution.

Batching. Batching to amortize costs and condense work is a common optimization technique. One application is to pack networking and logging messages [12, 16, 22, 23]. Batching is also widely applied to aggregate application requests to improve performance, including group commits [15, 25], condensing IO requests [15, 19], and Paxos [44]. Since batching is often associated with a throughput/latency tradeoff, there is work on adaptive batching [21, 41]. 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 show how to significantly improve transaction performance in transaction processing systems based on optimistic concurrency control through storage and validator batching and reordering. Besides clean problem formulations and reducing validator reordering to the problem of finding the minimal feedback vertex set (FVS), we propose two new practical greedy algorithms for this problem that are fast and that perform well in practice. We show that our algorithms can integrate different policies into the reordering to optimize for a variety of performance objectives and system architectures, such as low tail latency and multi-threaded decentralized OLTP architectures. We further propose a parallel validator design to reduce the overhead of reordering. Our extensive experimental study both in a prototype system, as well as with a state-of-the-art OLTP system and a commercial database system show that both storage and validator batching consistently improve throughput, and that validator reordering significantly reduces latency profiles. We also demonstrate how we optimize for low tail latency with alternative policies, and how our parallelization further improves throughput and reduces latency.

In future work, we plan to explore more sophisticated batch creation techniques. Since we observe a sweet spot for the batch size in our experiment, we want to systematically investigate adaptive batching that intelligently adjusts the batch size for the best performance given the workload.We are also interested in designing additional policies to apply the idea of batching and reordering to alternative system architectures and concurrency control protocols.

8. REFERENCES

[1] R. Abbott et al. 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.

[3] R. Agrawal et al. Concurrency control performance modeling: alternatives and implications. *TODS*, 1987.

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

[5] R. Appuswamy, A. C. Anadiotis, D. Porobic, M. K. Iman, and A. Ailamaki. Analyzing the impact of system architecture on the scalability of oltp engines for high-contention workloads. *Proc. VLDB Endow.*, 11(2):121–134, Oct. 2017.

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

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

[8] P. A. Bernstein, S. Das, B. Ding, et al. Optimizing optimistic concurrency control for tree-structured, log-structured databases. In *SIGMOD*, 2015.

[9] P. A. Bernstein et al. Hyder - A transactional record manager for shared flash. In *CIDR*, 2011.

[10] P. A. Bernstein, C. W. Reid, et al. Optimistic concurrency control by melding trees. *PVLDB*, 2011.

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

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

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

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

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

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

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

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

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

[20] P. Franaszek and J. Robinson. Limitations of concurrency in transaction processing. *TODS*, 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 et al. Quickly generating billion-record synthetic databases. In *SIGMOD Rec.*, 1994.

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

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

[27] J. Haritsa et al. Value-based scheduling in real-time database systems. *The VLDB Journal*, 1993.

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

[29] C. A. R. Hoare. Algorithm 65: Find. *Commun. ACM*, 4(7):321–322, July 1961.

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

[31] V. Kann. *On the approximability of NP-complete optimization problems*. PhD thesis, 1992.

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

[33] K. Kim, T. Wang, R. Johnson, and I. Pandis. Ermia: Fast memory-optimized database system for heterogeneous workloads. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1675–1687. ACM, 2016.

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

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

[36] P.-Å. Larson et al. High-performance concurrency control mechanisms for main-memory databases. *VLDB*, 2011.

[37] J. Lee et al. Using dynamic adjustment of serialization order for real-time database systems. In *RTSS*, 1993.

[38] H. Lim, M. Kaminsky, and D. G. Andersen. Cicada: Dependably fast multi-core in-memory transactions. In *Proceedings of the 2017 ACM International Conference on Management of Data*, pages 21–35. ACM, 2017.

[39] Y. Lin and S. H. Son. Concurrency control in real-time databases by dynamic adjustment of serialization order. In *RTSS*, 1990.

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

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

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

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

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

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

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

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

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

[49] S. Tu, W. Zheng, et al. Speedy transactions in multicore in-memory databases. In *SOSP*, 2013.

[50] T. Wang and H. Kimura. Mostly-optimistic concurrency control for highly contended dynamic workloads on a thousand cores. *Proceedings of the VLDB Endowment*, 10(2):49–60, 2016.

[51] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li. Scaling multicore databases via constrained parallel execution. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1643–1658. ACM, 2016.

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

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

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

[55] X. Yu, A. Pavlo, D. Sanchez, and S. Devadas. Tictoc: Time traveling optimistic concurrency control. In *Proceedings of the 2016 International Conference on Management of Data*, pages 1629–1642. ACM, 2016.

APPENDIX

A. PARALLELISM

Since batching and reordering occur as part of the transaction execution, it can increase the overall transaction latency, resulting in a higher chance of conflicts. Thus, we introduce parallelism into validation wherever possible to reduce latency. Recall that the validator first prepares a batch of transactions, then reorders them using one of our algorithms from Section 4, and finally validates them and caches the resulting updates of transactions that commit. Each of these steps corresponds to a subcomponent in the validator. Parallelism across these components seems difficult since there are strict sequential dependencies among these components. In this section, we explain how we actually manage to parallelize execution within each component and then even across components.

Figure 25 shows the architecture of our parallel validator. The validator has three components: The *batch preparation component* receives transactions from the processor, packages them into batches, and sends batches to the *transaction reordering component*. The transaction reordering component reorders the transactions using one of our algorithms from Section 4, and then sends a validation request to the *transaction validation component*. The transaction validation component takes a batch of ordered transactions and validates them against the latest validator cache. It also updates the validator cache with the updates from transactions that have passed the validation.

Parallelism Within Each Component. First, we can introduce parallelism into each of the components. In batch preparation, multiple threads can package transactions into batches. We can either assign each processor to send its validation request to a specific batch preparation thread, or we can create a consumer-producer queue to connect the processors 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 batches are not ordered yet in the reordering stage (although transactions within each batch are now ordered), the threads can send the processed batches to the validation component in any order. At the validation component, the batches are processed in order and validated against all previously committed transactions. Within an ordered batch, since the transactions are already serialized by the ordering produced from the reordering component, the only source of conflicts is from all the transactions committed prior to the batch. Thus the validation of all transactions within a batch can be processed in parallel. Since conflicts can happen across batch boundaries, processing a new batch can only start after all the transactions in the previous batch have been validated. The updates from committed transactions in a batch are applied to the validator cache in serialization order at the end of the processing of the batch. Alternatively, we can use partitioned parallelism within a batch: We partition the key space and break a transaction into smaller pieces, one for each piece of the key space, and we assign different threads to validate transaction pieces in different partitions. This is analogous to the design of a partitioned validator in [16].

Parallelism Across Components. We can further increase parallelism by running the three validator components in parallel using pipelined parallelism, with each component working on a different batch in parallel. The batches then shift from component to component through the validator.

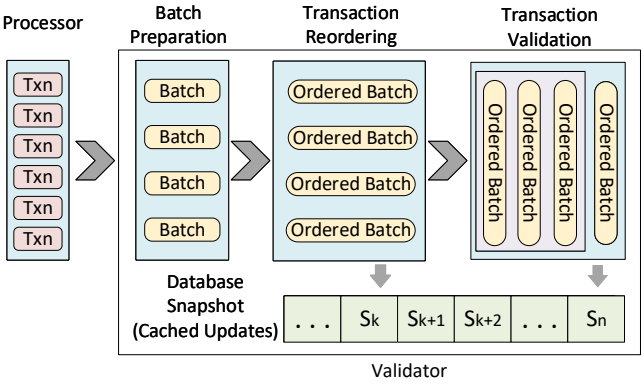


Figure 25: The architecture of parallel validator. It is decoupled into three subcomponents for pipeline parallelism: batch preparation, transaction reordering, and transaction validation. Each subcomponent can be further parallelized.

A Further Refinement: Pre-Validation. As we mentioned in Section 3.2, prior to reordering, we can remove transactions that are not viable, i.e., conflicting with previously committed transactions. While this *pre-validation* adds an additional validation for transactions on top of the final validation which serializes them, it reduces the number of transactions to reorder and the reordering algorithm runs faster. In our experiments, we observed that the reordering component was by a large factor the bottleneck piece, and eliminating non-viable transactions with pre-validation significantly improved performance. Thus, after batch creation, we first pre-validate the batch, reorder the remaining transactions, and then perform a second and final validation against the current database state.

We illustrate this design in Figure 25. The bottom row shows the set of database snapshots, one after each batch of transactions that has been validated. Consider a batch B. With pre-validation, B will get validated against a "stale" database state S. Those transactions within batch B that did not abort during pre-validation were then re-ordered and validated a second time in the transaction validation component against the "correct" database state S'. Since S' now reflects all the updates from transactions that committed while B was in the reordering component, transactions in B can show conflicts during the final validation even though they were once already pre-validated in the reordering component.

B. EXPERIMENTAL EVALUATION: PARALLEL VALIDATOR REORDERING

In this experiment, we study the benefit of introducing parallelism into the validator. Since we have observed that the reordering of FVS is the most time consuming subcomponent in the validator, we increase the number of threads to parallelize batch reordering as described in Appendix A. Figure 26 and 27 show the throughput and the average latency with the number of reordering workers from 1 to 4 ($w1$, $w2$, $w3$, $w4$). The performance improves significantly with more reordering workers when the skew factor is medium

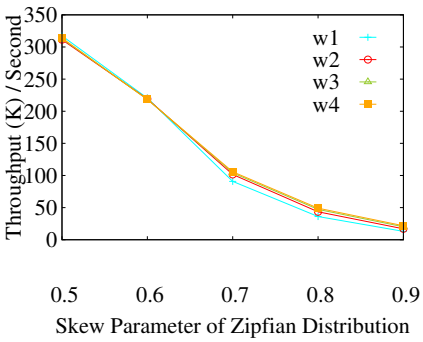


Figure 26: Throughput with different number of reordering workers

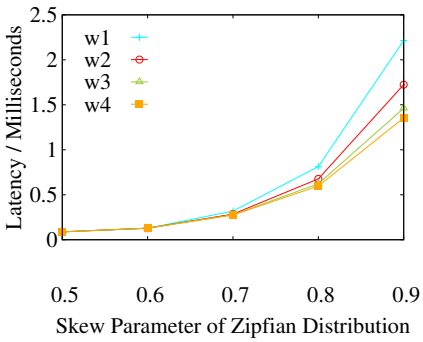


Figure 27: Average latency with different number of reordering workers

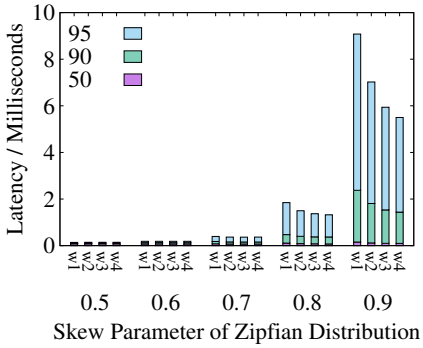


Figure 28: Percentile latency with different number of reordering workers

to high. With 4 reordering workers, the throughput is increased by up to 2.6x, and the average latency is reduced by up to 39%, as compared to the result with 1 reordering worker. Figure 28 shows the percentile transaction latency. With more reordering workers, more transactions are reordered concurrently, and the transaction queuing time at validator is reduced. With 4 reordering workers, the tail latency is reduced by up to 41%. The improvement is not linear since the bottleneck of the system shifts to other components as we increase the capacity of the transaction reordering.