Data Sci. Eng. (2017) 2:301–315
https://doi.org/10.1007/s41019-017-0054-0

Reordering Transaction Execution to Boost High-Frequency Trading Applications

Ningnan Zhou[1,2] · Xuan Zhou[3] · Xiao Zhang[1,2] · Xiaoyong Du[1,2] · Shan Wang[1,2]

Received: 25 August 2017 / Revised: 25 October 2017 / Accepted: 5 November 2017 / Published online: 15 November 2017
© The Author(s) 2017. This article is an open access publication

Abstract High-frequency trading (HFT) has always been welcomed because it benefits not only personal benefits but also the whole social welfare. While the recent advance of portfolio selection in HFT market enables to bring about more profit, it yields much contended OLTP workloads. Featuring exploiting the abundant parallelism, transaction pipeline, the state-of-the-art concurrency control (CC) mechanism, however, suffers from limited concurrency confronted with HFT workloads. Its variants that enable more parallel execution by leveraging fine-grained contention information also take little effect. To solve this problem, we for the first time observe and formulate the source of restricted concurrency as *harmful ordering* of transaction statements. To resolve harmful ordering, we propose PARE, a pipeline-aware reordered execution, to improve application performance by rearranging statements in order of their degrees of contention. In concrete, two mechanisms are devised to ensure the correctness of statement rearrangement and identify the degrees of contention of statements, respectively. We also study the off-line reordering problem. We prove that this problem is NP-hard and present an off-line reordering approach to approximate the optimal reordering strategy. Experiment results show that PARE can improve transaction throughput and reduce transaction latency on HFT applications by up to an order of magnitude than the state-of-the-art CC mechanism.

✉ Xiao Zhang
zhangxiao@ruc.edu.cn

[1] School of Information, Renmin University of China, Beijing, China

[2] MOE Key Laboratory of DEKE, Renmin University of China, Beijing, China

[3] School of Data Science and Engineering, East China Normal University, Shanghai, China

Keywords Concurrency control · Reordered execution · Online reordering · Off-line reordering

1 Introduction

The ever-increasing CPU core counts and memory volume are witnessing a renaissance of concurrency control (CC) mechanisms in exploiting the abundant parallelism [24]. Transaction pipeline, the state-of-the-art CC mechanism, takes advantage over prior CC mechanisms, including two-phase locking (2PL), optimistic concurrency control (OCC) and multi-version concurrency control (MVCC), by allowing more parallel execution among conflicting operations [16, 36]. However, it suffers from long-time delays confronted with high-frequency trading (HFT) applications.

HFT applications have been pervading the worldwide market since the last decade [1], ranging from individual investment, such as mutual fund management, to social welfare, such as pension fund management [8]. The recent advance in portfolio selection in HFT market encourages to compose each trade of both strong and weak investment signals in sake of risk management [35]. Because it is easier to capture strong investment signals, such as new production release, different portfolio selection algorithms tend to receive the same strong investment signals while differ at weak investment signals [12]. For example, Listing 1 describes the transactions produced by two portfolios which reflect the same strong investment signals to buy stocks from security `Alphabet`, `Amazon` and `Twitter` while share no weak investment signal in common. As a result, the HFT workloads interleave much contended operations with rarely contended ones, which reflect strong and weak investment signals, respectively.

```
Transaction T1                      Transaction T2
update("Alphabet",  30);            update("Alphabet",  30);
update("Amazon",    30);            update("Twitter",   30);
update("Cisco",     10);            update("Facebook",  10);
update("Microsoft", 10);            update("Macy's",    10);
update("Tesla",     10);            update("Oracle",    10);
update("Twitter",   30);            update("Amazon",    30);
```

Listing 1: Two transactions generated by two portfolios, where the quantity 30 implies a strong investment signal and 10 a weak investment signal. Here `update("t", v)` represents that the transaction attempts to buy the security `t` with quantity `v`.

Figure 1a illustrates the execution of the two transactions in Listing 1 under the state-of-the-art CC mechanism transaction pipeline. The first conflicting operation, e.g., `update` on `Alphabet`, determines the serializable order of the two transactions, i.e., T_1 happens before T_2. Then, any operation in T_2 must follow this order; otherwise, the operation will be re-executed after a violation is detected. As Fig. 1a illustrates, this is equivalent to delaying the execution of any operation in T_2 till the completion of its corresponding conflicting operation in T_1. On the one hand, this allows T_2 to perform `update(Alphabet)` before T_1 completes. Thus, transaction pipeline indeed outperforms all of 2PL, OCC and MVCC, which do not allow any overlapping execution because otherwise deadlock (2PL) or rollback (OCC and MVCC) will happen. On the other hand, the second update in T_2 on Twitter must be delayed until the completion of T_1. Compared to Fig. 1b, which reorders the execution of T_1 and T_2 according to Listing 2, we can see that the delay in Listing 1 unnecessarily compromises throughput and increases transaction latency.

Because delay is mainly caused by conflicting operations, much work focuses on extracting more fine-grained contention information from transaction semantics to benefit transaction pipeline [13, 36, 41, 42]. However, fine-grained contention information reduces false contention, while Fig. 1a indicates that HFT applications benefit little from this kind of variants of transaction pipeline.

As a remedy, we present PARE, a pipeline-aware reordered execution of transactions, in this paper. To the best of our knowledge, it is the first time that reordering transaction execution is considered to benefit the CC mechanism of transaction pipeline. First, we observe and formulate harmful ordering of statements in transaction code and propose to eliminate harmful ordering by rearranging statements in decreasing order of the degree of contention. To this end, we devise two mechanisms. On the one hand, to preserve serializability after reordering, we devise a reordering block extraction algorithm. On the other hand, to measure the degree of contention, we devise a physical operator-based counter.

```
Transaction T1                      Transaction T2
update("Alphabet",  30);            update("Alphabet",  30);
update("Amazon",    30);            update("Amazon",    30);
update("Twitter",   30);            update("Twitter",   30);
update("Cisco",     10);            update("Facebook",  10);
update("Microsoft", 10);            update("Macy's",    10);
update("Tesla",     10);            update("Oracle",    10);
```

Listing 2: Reorder Transactions from Listing 1.

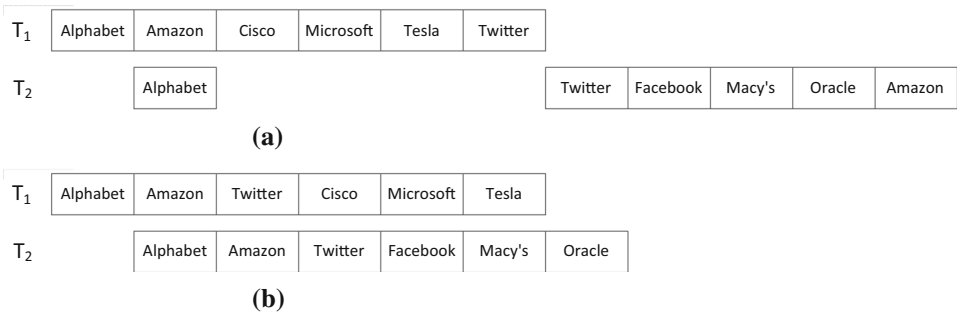

Fig. 1 Comparison of delay between original and reordered transactions. **a** Transaction pipeline on the original transaction. **b** Transaction pipeline on the reordered transaction

When transaction requests are available in advance, we propose an off-line reordering approach to optimize the scheduling of transactions. We prove that this optimization problem is NP-hard and proposes a heuristic approach to approximate the optimal solution. We evaluate the performance and practicality of PARE. The experiment results show that PARE improves transaction throughout and reduces transaction latency by up to an order of magnitude than the state-of-the-art CC mechanisms. In addition, the runtime overhead is limited.

The contributions of this paper are fivefold:

- We show the importance of reordering transaction execution for the state-of-the-art CC mechanism transaction pipeline under HFT applications.
- We observe and formulate harmful ordering under transaction pipeline and propose to rearrange statements in decreasing order of contention to eliminate harmful ordering.
- We propose two mechanisms to ensure the correctness of rearrangement of transaction statements and measure the degree of contention to enforce the elimination of harmful ordering.
- We formulate the off-line reordering problem, prove it is NP-hard and present an approach to approximate the optimal reordering schedule.
- We conduct experiments to demonstrate the effectiveness and practicality of PARE.

2 Preliminary and Related Works

In this section, we first introduce the CC mechanism of transaction pipeline and then review the related work. Readers familiar with transaction pipeline can pass Sect. 2.1.

2.1 Transaction Pipeline

Transaction pipeline is the state-of-the-art CC mechanism, and it exploits much more parallelism than other CC mechanisms. To ensure serializability, prior CC mechanisms including 2PL, OCC and MVCC restrict interleavings among conflicting transactions [16]. In particular, if transaction T_2 reads T_1's write to x, all 2PL, OCC and MVCC produce schedules in which T_2's read always follows T_1's completion. Under 2PL, each transaction holds long-duration locks on records; any locks acquired by a transaction are only released at the end of its execution [15]. This long discipline constraints the execution of conflicting reads and writes; if transaction T_2 reads T_1's write to record x, and T_1 holds a write lock on x until it completes, T_2's read can only be processed after T_1 completes. Under OCC, transactions perform writes in a local buffer and only copy these writes to the active database after validation [23]. Thus, a transaction's writes are only made visible at the very end of the transaction. Under MVCC, each write is assigned with a write timestamp when the transaction commits and each read is associated with a read timestamp when the transaction begins. Any read can only read a record whose write timestamp is less than the read timestamp [27]. Thus, MVCC similarly constrains conflicting transactions.

Applications including HFT applications call for more aggressive concurrency. As a result, transaction pipeline opens a new design choice of operating uncommitted data. Two mechanisms are enforced to ensure the serializability. First, because uncommitted data is read, a commit discipline is enforced such that (1) if transaction T operates on uncommitted data from T' and T' has not committed, T should not commit; and (2) if T' aborts, T should also abort [16, 36]. Second, dependencies that arise as transactions make conflicting data accesses should be tracked and these dependencies should be enforced by constraining a transaction's subsequent data access. Transaction pipeline combines runtime techniques with a static analysis of the transaction workloads. The static analysis is based on prior work on transaction chopping [5, 34]. In particular, it constructs a static conflict graph (SC-graph) in which a transaction is represented as a series of atomic pieces, each making one or more database access. Two pieces of different transactions are connected if both access the same table and one of the accesses is a write. A cycle involving multiple pieces of some transaction (i.e., an SC-cycle) indicates a potential violation of serializability. Transaction pipeline uses its runtime to constrain the execution of the corresponding pieces. For example, the first time T_2 reads T_1's uncommitted write, it is determined that T_2 should happen after T_1. Then, if T_2 is about to write y, which T_1 may also access, T_2's access will be deferred until T_1 terminates or will not access y again.

2.2 Related Works

Our work is built on transaction pipeline, and it leverages transaction code further than the original transaction pipeline.

Variants of transaction pipeline are attracting the most attention in concurrency control research. These methods feature assuming that the workloads are available beforehand. In this way, they are able to chop transactions into pieces such that once each piece of transaction code begins to execute, it will not need to delay until the completion of the whole transaction. In the original transaction pipeline approach named IC3 [36], whether two pieces conflict depends on whether they access the same table and one of them contains write. Fine-grained variants are further proposed by regarding two pieces as conflicting pieces if they perform conflicting accesses at runtime [13, 16, 41]. In particular, TheDB [41] captures the dependencies access operations and partially update the membership of read/

write sets by uncommitted data from its dependent transactions. Both IC3 and TheDB adopt an underlying single version storage. MV3C [13] adopts a multi-version storage. Because the only way to generate a conflict is that a transaction reads some data objects from the database that become stale by the time it tries to commit. In this way, uncommitted write should be applied to these reads. To this end, each read is associated with the blocks of code that depend on them. In this way, the portion of a transaction that should be re-executed in the case that uncommitted data are applied to the read is identified quickly. PWV [16] operates uncommitted data on deterministic database. It decomposes transactions into a set of sub-transactions or pieces such that each piece consists of one or more transaction statements. PWV makes a piece's writes visible as soon as its host transaction guarantees to commit, even if there are still pieces of the same transaction not executed.

Among these methods, IC3 [36] and TheDB [41] assume fixed workloads on simple Get/Put/Scan interfaces and thus cannot be applied to dynamic HFT workloads, which are expressed by SQL. MV3C [13] and PWV [16] annotate transactions by hand, which is also impractical in HFT applications. In contrast, our PARE operates on dynamically generated SQLs automatically and thus can be applied on fast changing transactions issued by HFT applications. In addition, these methods do not consider reordering transaction execution and thus suffer from inferior performance under HFT workloads.

Program analysis is widely adopted to improve database performance from the data application's perspective. Both variants of transaction pipeline and PARE are built upon prior work on statically analyzing transactions to assist runtime concurrency control. Conflict graphs have been long used to analyze conflicting relationships among transaction operations and preserve serializability by predefining orders of transactions [6, 7]. However, early work handles cyclic dependency by stopping all other transactions in a cycle. Further work begins to decompose transactions into pieces [5, 14, 18, 19, 34]. To preserve serializability after chopping, it is first observed that if all pieces of a decomposed transaction commute, it is safe to interleave executions [17]. However, this is a strong condition and many OLTP workloads do not suffice. Further, it is shown that serializability can be preserved if there is no SC-cycle in an SC-graph [5]. This theory is further extended to distributed systems [9].

Program analysis is also exploited to extract semantics to allow non-serializable interleavings of database operations. The demarcation protocol [4] allows asynchronous execution of transactions in a replicated system by maintaining an integrity constraint. The distributed divergence control protocols [31, 40] allow some inconsistency but bounds it to ensure that the produced schedules are within epsilon of serializability [32]. More recently, a continuous consistency model [43] is proposed for replicated services which allows bounded inconsistency and uses anti-entropy algorithms [28, 29] to keep it at an acceptable level. A homeostasis protocol is recently proposed. It features inferring the constraints to be maintained automatically [33]. I-confluence [3] determines whether a conflicting operation will preserve application invariants based on application dependent correctness criterion. There is also extensive research on identifying cases where transactions or operations commute and allowing interleavings which are non-serializable yet correct [2, 22, 25]. Different from these works, PARE prohibits non-serializable execution and thus can be applied without interference of human.

Program analysis is also used to speed up transactions. Sloth [11] and Pyxis [10] are tools that use program analysis to reduce the network communication between the application and DBMS. QURO [42] tracks dependencies among transaction statements to reorder transaction execution for 2PL. DORA [30] also partitions transaction codes to exploit intra-transaction parallelism provided by multi-core server for 2PL. Majority of these techniques do not consider reordering transaction execution and thus cannot reduce delays for transaction pipeline. QURO seems the most related approach to PARE since both of them reorder transaction execution. However, two inappropriate design choices of QURO hinder transaction pipeline from boosting HFT applications: (1) QURO assumes fixed workloads and requires to sample the degrees of contention for each statement in advance. Unfortunately, workload sampling is prohibitive because each transaction is dynamically generated according to the timely business information in HFT market. (2) QURO estimates the degree of contention at statement level and by the delayed time. As discussed in Sect. 3.3, this underestimates the degree of contention and repeatedly obtains inconsistent degree of contention. In contrast, PARE estimates at physical operator level and counts the occurrence frequency of contention.

3 Pipeline-Aware Reordered Execution

In this section, we first deduce the reordering strategy of PARE and then present two mechanisms to enable this strategy.

3.1 Reordering Strategy

We observe that there are two kinds of delays under transaction pipeline, which are the only source of inferior performance. The first kind of delays comes from the first conflicting operation, such as the `update(Alphabet)` of transaction

T_2 in Fig. 1. Because it is undefined to write to the same object simultaneously, such delay is inevitable. Fortunately, this kind of delays lasts for only one operation and thus can be neglected in transaction execution. The second kind of delays can last indefinitely long time, such as the `update(Twitter)` of transaction T_2 in Fig. 1a. We observe that such delay is formed when non-conflicting operations are encompassed by conflicting operations and formulate this observation as harmful ordering of conflict statements.

Definition 1 (*Harmful Ordering of Conflict Statements*) Suppose that two transactions consisting of n and m statements are denoted by $T_1 = [s_1, s_2, \ldots, s_n]$ and $T_2 = [s'_1, s'_2, \ldots, s'_m]$, a harmful ordering of conflict statements exists if there exist $1 \le i < j \le n$ and $1 \le p < q \le m$ such that the following three conditions hold:

1. s_i conflicts with s'_p and s_j conflicts with s'_q
2. $\forall i < x < j, p < y < q$, s_x does not conflict with s_y
3. $j - i \neq q - p$

For example, in Listing 1, where $n = m = 6$, a harmful ordering of conflict statements exists by setting $i = 1, j = 2, p = 1$ and $q = 6$. Condition 1 holds because `update(Alphabet)` and `update(Twitter)` of T_1 and T_2 are two conflicting operations. Condition 2 and condition 3 hold because there is no operations between `update(Alphabet)` and `update(Twitter)` in T_1, while there are other operations between `update(Alphabet)` and `update(Twitter)` in T_2.

Theorem 1 *Any bubble in the transaction pipeline is caused by harmful ordering of conflict statements.*

Proof Assume that there is no harmful ordering, there are three possibilities: (1) condition 1 does not hold. Then, there is no conflicting operations or only one conflicting operation. In the first case, transactions can execute without any delay. In the second case, the transactions delay at the first conflicting operation. (2) condition 1 holds while condition 2 is violated. In this case, $\exists i < x < j, p < y < q$ such that s_x conflicts with s'_y, we have $x - i = y - p$. In this way, if s_i is first executed, when T_2 is about to execute s'_y, s_x must have been completed and thus there is no delay. Otherwise if $x - i \neq y - p$, we can construct another harmful ordering which contradicts with our assumption. (3) Condition 1 and condition 2 hold while condition 3 is violated. In this case, if $s_i(s'_p)$ is first executed, when it is about to execute $s'_q(s_j)$, $s_j(s'_q)$ will be placed exactly before $s'_q(s_j)$ in the transaction pipeline and thus there is no transaction bubble. In summary, if there is no harmful ordering, no bubble will occur in the transaction pipeline. □

Transaction pipelines without bubbles do not induce execution without delay. This is because transactions in different threads do not proceed with the same speed. For example, Fig. 2 illustrates that although there is no bubble in transaction pipeline, because the execution of the update on Twitter tuple consumes a little more time, transactions are delayed. Even though, suppose that the duration difference of each operation is at most δ, a transaction with n operations will be delayed for at most $n\delta$ time. So the average delay for each operation is δ, which is limited. For transaction pipeline with bubble, there is no such guarantee. Thus, in this paper, we aim to eliminate harmful ordering.

To eliminate harmful ordering, it works to rearrange statements in increasing or decreasing order of contention. In this way, the condition 2 in Definition 1 will not hold in overall cases. In this paper, we make an arbitrary choice of reordering in decreasing order of contention and experiments show that these two strategies match each other. Applying this strategy to Listing 1, we obtain the reordered Listing 2 and Fig. 1b illustrates the execution and we can see that there is no delay longer than one operation.

To enforce this strategy, there are two obstacles: (1) reordering some statements may violate program semantics and (2) the degree of contention is fast changing and cannot be obtained in advance. Next, we show how to reorder statements safely and dynamically estimate the degree of contention.

3.2 Reordering Block Extraction

PARE manipulates on *reordering blocks*, which can be rearranged arbitrarily and preserve transaction semantics at the same time. These reordering blocks are extracted from transaction code automatically by leveraging data dependencies.

Definition 2 (*Reordering Block*) Given each transaction in the form of a sequence of statements $S = [s_1, s_2, \ldots, s_n]$, a reordering block B is a subsequence $B = [s_{i_1}, s_{i_2}, \ldots, s_{i_m}]$, where $1 \le i_1 < i_2 < \cdots < i_m \le n$, such that every statement s_{i_j} in block B does not depend on any statement s' in other reordering blocks in terms of data flow. Formally, $\forall s' \in S - B$, $\forall s_{i_j} \in B$, the three types of dependencies should be eliminated:

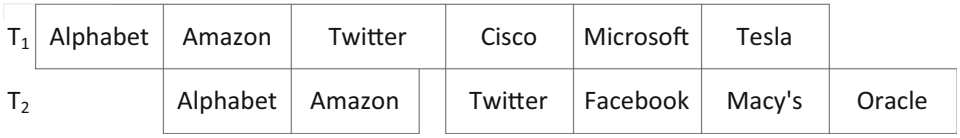

Fig. 2 A delay in transaction execution

1. $\texttt{readSet}(s_{i_j}) \cap \texttt{writeSet}(s') = \emptyset$ such that s_{i_j} does not depend on s' in terms of `read-after-write` dependency.
2. $\texttt{writeSet}(s_{i_j}) \cap \texttt{readSet}(s') = \emptyset$ such that s_{i_j} does not depend on s' in terms of `write-after-read` dependency.
3. $\texttt{writeSet}(s_{i_j}) \cap \texttt{writeSet}(s') = \emptyset$ such that s' and s_{i_j} do not depend on each other in terms of `write-after-write` dependency.

In brief, each reordering block collects the statements that do not have the three types of dependencies with the remaining statements. If we partition the statements of a transaction into disjoint reordering blocks, it ensures serializability after arbitrary rearrangement without extra handling.

Theorem 2 *Given a transaction S and a set of reordering blocks* $\mathcal{B} = \{B\}$, *where* $\forall B, B' \in \mathcal{B},\ B \cap B' = \emptyset$ *and* $\bigcup_{B \in \mathcal{B}} = S$, *rearranging reordering blocks will not compromise serializability under serializable schedule.*

Proof According to the definition of reordering blocks, the reordering blocks do not conflict with each other. Following the theory of Conflicting Serializability [37], these reordering blocks can be rearranged arbitrarily such that the behavior of the transaction will not be affected. □

The three types of dependencies should be tracked on both program statements and SQL statements. To deal with program statements, it is straightforward for us to adopt the standard dataflow algorithm [26, 42]. For simplicity, in this paper, we regard the `if`-statement and `loop`-statement as a whole statement, although sophisticated techniques such as loop fossil can make fine-grained dependency tracking [39, 42]. As for SQL queries, we precisely model the dependency relationships among the query input and output:

1. Simple selection with no sub-queries: the read set encompasses all predicates in the `where` clause. Empty `where` clause leads to a universal read set. The write set is set to empty.
2. Other selection: the read set encompasses all predicates occurred in all `where` clauses. If empty `where` clause occurs in one of the sub-queries, the read set of the query is also a universal set. The write set is set to empty.
3. Simple update with no sub-queries: the read set is set as "simple selection with no sub-queries" does. The write set is the same as the read set.
4. Other update: the read set is set as "other selection" does. The write set is the same as the read set.

According to these rules, Algorithm 1 shows how to transform transaction code into reordering blocks. We iteratively check for each unused statement which statements cannot reorder with it (Line 4–6). All statements that cannot reorder each other will compose a reordering block (Line 11). The concatenation preserves the original execution order for statements in any reordering block (Line 9, where $\oplus$ denotes concatenation). After all statements are assigned with certain reordering blocks, all reordering blocks are found (Line 4,12). For example, for Transaction T_1 in Listing 1, all statements are simple updates and the read set and write set of these statements are disjoint. In this way, each statement creates a reordering block. In other words, all updates can be reordered arbitrarily. In this paper, we focus on reordering selection and update. Dependency relationships of other SQL queries can be defined accordingly.

Algorithm 1: Reordering Block Extraction

Input: A sequence of statements $S = [s_1, s_2, ..., s_n]$
Output: A set of Reordering Blocks $\mathcal{B}$

```
1  B = ∅
2  for each statement s ∈ S do
3  |  compute the readSet and writeSet of s
4  for S ≠ ∅ do
5  |  B = [S[0]]
6  |  for each s' ∈ S, s' ∉ B do
7  |  |  for each s ∈ B do
8  |  |  |  if readSet(s) ∩ writeSet(s') ≠ ∅ or writeSet(s) ∩ readSet(s') ≠ ∅
   |  |  |  or writeSet(s) ∩ writeSet(s') ≠ ∅ then
9  |  |  |  |  B = B ⊕ s'
10 |  S = S − B
11 |  B = B ∪ {B}
12 return B
```

3.3 Contention Estimation

In this section, we first show the design to measure the degree of contention and then show a memory-compact way to recycle counters.

To obtain the degree of contention of a reordering block, we first estimate the degree of contention of each statement thereof and use the maximum as the reordering block's degree of contention. In this way, we will not miss highly contended operations enclosed by reordering blocks with many less contended operations. Existing method uses waiting time and its variation on each statement to estimate the degree of contention [42]. This solution has two shortcomings: (1) the counter set for each unique statement underestimates the real degree of contention for the statement. For HFT application, each statement usually contains different parameters. For example, the SQL query for `update(Alphabet, 30)` is usually instantiated as `update Stock set quantity -= 30, price = p where sid = Alphabet`, where `p` is the latest price provided by user. Thus, many statements updating the same quantity and price fields are counted in different counters. (2) Estimating waiting time and its variance suffers from thrashing. This is because properly reordered highly contended operations will not wait for long time and thus are always categorized as less contended operations. Then, these operations will be placed to the end of the transaction and be considered as heavily contended operations again. To address these two issues, we propose an execution plan-based estimation which collects the occurrence of each object in a time sliding window.

We observe that although conflicting operations may be represented in different SQLs, they must create the same physical operator referring to the same attributes. For example, although the `update(Alphabet, 30)` may update different prices, its corresponding physical operator will have the same type of `update` and indicate the same fields of `quantity` and `price`. The only difference is the parameters for the target values. Because contention only occurs at the operator placed at the bottom level of the execution plan in form of a tree structure, it is sufficient to allocate a counter recording the occurrence frequency for each physical operator at the bottom level of the execution plan. Algorithm 2 details the counter implementation. The time sliding window size *WS* is discretized into time slots with granularity of *g* so that when the time flows, the time slots are recycled to reflect the latest count. Once one physical operator is created at the bottom level of the execution plan, the `Increment` of the associated counter is invoked. Once we extract all reordering blocks, we invoke `Get` for each counter within each reordering block and pick up the maximum count as the degree of contention for each reordering block.

Algorithm 2: Time Sliding Window Counter

Input: Time Sliding Window size WS, Granularity g
Initialize $C[0 : \lceil WS/g \rceil]$ to $[0, 0, ..., 0]$
Function `Increment(`t`)` *// t is the current timestamp*
  $i = t$ % $(\lceil WS/g \rceil + 1)$;
  $C[i] += 1$;
  $i = (t - \lceil WS/g \rceil + 1)$ % $(\lceil WS/g \rceil + 1)$;
  $C[i] = 0$;
  return;
Function `Get()`**:**
  $c = 0$;
  for $i = 0$ *to* $\lceil WS/g \rceil$ **do**
    $c = c + C[i]$;
  return c;

It may waste memory to allocate each physical operator with a counter, especially under HFT workloads where contended objects are fast changing so that the allocated counter may no longer be used in the next time period. To ease this concern, we maintain a counter pool and every time we need to allocate a new counter, we recycle the counter not used in the last time window.

Our counter mechanism has two parameters to set. From the perspective of the precision of the degree of contention, less *g* and moderate WS are welcomed. This is because larger *g* and WS will lead to stale degree of contention and the reordering will not eliminate harmful ordering. Too small *WS* compared to *g* will lost much contention information. From the perspective of memory overhead, large WS is not expected. However, too small *g* is prohibited by hardware. In this paper, experiments show that the setting of WS = 1 s and $g = 100$ ms works well under HFT workloads.

4 Off-line Reordering

So far, we have discussed a runtime reordering approach to reorder transactions in batch. In practice, HFT applications may submit their transaction programs before the stocking market is open and would like to pay more for transactions with priority. In this section, we can reorder transactions off-line so that more important transactions can achieve more concurrency at runtime.

4.1 Problem Formulation

The off-line reordering problem has two motivations. First, it can arrange the transactions so that the same harmful ordering from different transactions does not overlap. In this way, although the harmful ordering still exists, they will not block transactions. For example, Listing 3 shows two types of transactions. We can see that transactions of

both Type I and II can only be executed serially. Rather, suppose that there are four transactions T_1, T_2, T_3 and T_4 and T_1 and T_2 belong to Type I and T_3 and T_4 belong to Type II. If we execute T_1 and T_3 concurrently and after they complete we execute T_3 and T_4 concurrently, no transaction will be blocked.

```
1 Transaction Type I       Transaction Type II
2 update("x");             update("a");
3 update("y=x");           update("b=a");
4 update("x=y+1");         update("a=b+1");
```

Listing 3: An Example of Two Types of Transactions. Transactions belonging to the same transaction type can only be executed serially.

Second, if some transactions are regarded more important, they can be scheduled to complete faster. Listing 4 describes this case. There are three types of transactions. We can see that the first two types of transactions can be executed simultaneously while the last two types of transactions can only be executed serially. Given a fixed time, if we want to maximize the throughput, we would like to run transactions of Type I and II as many as possible and then execute the transactions of Type III. However, if the stock market platform knows that the requestors of the last two types of transactions would like to pay more fees for completing their transactions within one hour, the stock market would like to run these two types of transactions in the limited time.

```
1 Transaction Type I       Type II                 Type III
2 update("x");             update("a");            update("b");
3 update("y");             update("b");            update("a");
```

Listing 4: An Example of Three Types of Transactions.

To formulate these two goals, each schedule can be formulated as a configuration on C timeline of size L. Here, we use C to denote the number of processors and L the length of the longest timeline. Each slot in the $C \times L$ configuration indicates a statement from given transactions. If each transaction statement is filled in the configuration exactly once and it still preserves the semantics of any serial order of the transactions, we have a valid configuration. We formulate a configuration as follows.

Definition 3 A valid configuration is a matrix with L rows and C column such that the slot at the l-th row and c-th column indicates a unique statement from a transaction, and it preserves the semantics of any serial order of the transactions.

The optimal configuration is thus defined as follows:

Definition 4 Given a set of transactions $\mathcal{T}$ and the fee f_t associated with each transaction $t \in \mathcal{T}$, find a configuration c with size of T with respect to $\mathcal{T}$ such that the aggregated fees are maximized:

$$argmax_c \sum_t f_t$$

In this way, off-line reordering aims to find an optimal configuration for the given set of transactions to maximize the profit that the stock market platform can make. This optimization problem is NP-hard.

Theorem 3 *The problem of off-line reordering is NP-hard.*

Proof We first introduce the 0–1 knapsack problem. In 0–1 knapsack problem, there are n items and each item i has size s_i and cost c_i. The decision version of the 0–1 knapsack problem decides whether a subset of the items can be selected with size equal or greater than S with maximum cost M.

Given an instance of the 0–1 knapsack problem, we construct an off-line reordering instance as follows. Each item i corresponds to a transaction t. The transaction t is composed of s_i updates on s_i distinct records that are different from other transactions. The requestor if the transaction would like to pay c_i if the transaction is completed within the time M/C.

The two instances are equivalent. If the 0-1 knapsack problem can pick up a qualified subset of items, the transactions corresponding to these items can be finished within time C at one processor and weight at least S. In this way, we can divide the timeline into C parts evenly. These parts must partition transactions by the boundary of their updates. In this way, each processor can execute the

operations in each part with time M/C. If our problem can find a schedule, we can trivially merge the operations into one sequence and that corresponds to a subset of items.

In this way, we reduce the 0/1 knapsack problem to the off-line reordering problem. Because the 0/1 knapsack problem is NP-hard [20], our off-line reordering problem is also NP-hard. □

4.2 GA-Based Approach

Initially, we propose a genetic algorithm to solve the off-line reordering problem. Genetic algorithm is a well-known meta-heuristic to solve hard optimization problems [38]. Given a computation budget for each iteration, a number of iterative procedures will be executed. The idea is to set an initial individuals and mimic the evolution of individuals to find the most fit individual. In the first iteration, a number of individuals are picked up. In each further iteration, the individuals are evaluated and the best ones will be selected to evolve. The evolution is performed by a crossover procedure. In each crossover, selected individuals exchange some different parts so that the new individual, regarded as their offspring, would like to be evaluated better than its parents. Each iteration takes some budget, and after the budget is exhausted, the remaining individual is returned.

In the GA-based off-line reordering algorithm, each configuration is defined as an individual. The fitness function is its totally delayed time. In the first iteration, we randomly generate m individual. For each individual, the transactions are partitioned into C disjoint sets randomly and evenly. Then, for each partition, the transactions in each of the C sets are randomly sorted into a sequence. In this way, we obtain m valid configurations. In each iteration, all obtained configurations from the last iteration are evaluated. We select k top-fittest ones for the next iteration. Each two configurations of the k configurations are used to crossover to produce an offspring. In particular, a processor is first randomly selected and two operations are randomly selected. If the two operations belong to different reordering units, they will be swapped. Correspondingly, their reordering block and transaction order may be violated. As a remedy, if one operation is in the middle of a reordering block, its previous operations in the reordering block will be put ahead of the new place of the operation. Further, if the transaction order is violated, the other operations from affected transactions will be put ahead accordingly. Note that this may be achieved in an iterative way, because moving other operations ahead may violate transaction order also. We set m to memory capacity and k to ensure their generated offspring can still hold in memory.

However, the GA-based solution quickly got trapped in a local minimum. Experiments in Sect. 5.3 validate this argument.

4.3 SA-Based Approach

As a remedy, we propose a probabilistic algorithm called SOR (simulated annealing-based off-line reordering algorithm) to solve the off-line reordering problem. As shown in Algorithm 3, SOR leverages basic properties of simulated annealing [21] to solve the problem and generates an approximation of the optimal off-line reordering.

In SOR, each valid configuration corresponds to a state of the algorithm. Intuitively, a "good" configuration S is similar to another better configuration S'. Such a heuristic greatly reduces the search space without searching all possible states. In SOR, the energy of a state is measured by its consumed time. Different from other simple heuristic algorithms, SOR makes decisions between accepting or rejecting the neighbor state probabilistically. In concrete, given a configuration, we also define its neighboring state as randomly swapping two operations in a randomly chosen processor and then adjusting the operations in the corresponding reordering blocks and affected transactions. The distribution of the acceptance probability is determined by the annealing schedule. This ensures that SOR is unlikely to be trapped into an undesirable local minimum.

Given transactions and an initial configuration S_0, SOR returns a preferable configuration S, so that the consumed time of the transactions is significantly reduced. In Algorithm 3, the main loop shows the iterative search process based on an annealing schedule proposed in [21]. The temperature function is the core function of the annealing schedule. In this algorithm, the temperature shrinks at a rate of (1 - cooling_rate). Function Neighbor(S) generates another configuration which swaps two random reordering blocks of the configuration of S.

SOR has three important parameters: the initial temperature t_0, the cooling_rate and the maximum number of iterations max. These parameters are set following the best practices of simulated annealing. Parameters t_0 and cooling_rate have been studied in the literature [21] and as suggested, we set t_0 to be slightly larger than the consumed time of the initial S. The parameter max controls the number of iterations executed in the main loop of Algorithm 3. The final temperature at the end of algorithm execution is $t_0 \times (1 - cooling\_rate)^{max}$. In this paper, we set max so that the final temperature is close to 0, i.e., 0.01% of the initial t_0.

Algorithm 3: SOR

Input: A set of transactions and their reordering blocks
Output: A configuration of reordering blocks on C processors

```
set S a configuration that arrange the reordering blocks of each transaction in
round robin way to the C processors
set t0 to the consumed time of S
for k = 1 to max do
    t = Temperature(t, cooling_rate)
    S' = Neighbor(S)
    e = Cost(S)
    e' = Cost(S')
    if (e' < e) or exp((e − e')/t) > random(0, 1) then
        S' = S
return S
```

5 Experiment

In this section, we report experiment results on both workloads of HFT applications and typical OLTP workloads. We demonstrate the effectiveness and practicality of PARE.

5.1 Experimental Setting

In the experiments, we compare PARE against the original transaction pipeline implementation and a well-adopted popular CC mechanism MVCC. The evaluation is performed on contended workloads representing the HFT application and a widely adopted OLTP benchmark TPC-C. The HFT workload runs on a simplified stock exchange market based on a single table: Stock(SecurityName, Price, Time, Quantity). Each item in the stock table represents a security in the market with its name, latest price, latest trading time and the accumulated trading quantity. The HFT workload runs a unique type of transaction parameterized by a vector of security names. The n security names are selected by a portfolio selection algorithm. This design is motivated by the typical OLTP workload TPC-E, except that it allows various degrees of contention. In every f seconds, 5 out of n random securities are selected as strong investment signals. We mimic the dynamic nature and different degrees of contention of HFT applications by adjusting f and n, respectively.

The experiments were carried out on HP workstation equipped with 4 Intel Xeon E7-4830 CPUs (with 32 cores and 64 physical threads in total) and a 256GB RAM. The operating system was 64-bit Ubuntu 12.04. During the experiments, all databases were stored in an in-memory file system (tmpfs) instead of the hard disk, so that we could exclude the influence of I/O and focus on the efficiency of concurrency control.

5.2 Performance Comparison

In this section, we first compare the performance under HFT workloads to demonstrate the effectiveness of PARE. We present the results under three settings of extremely, highly and moderately contended workloads. Then, we compare the performance under typical OLTP workload, TPC-C, to demonstrate the broad scope of application of PARE.

Under extremely contended workloads, every transaction only contains the five conflicting operations ($n = 5$). As Fig. 3a illustrates, we can see that only PARE scales to 32 working threads and other methods crumble. What is

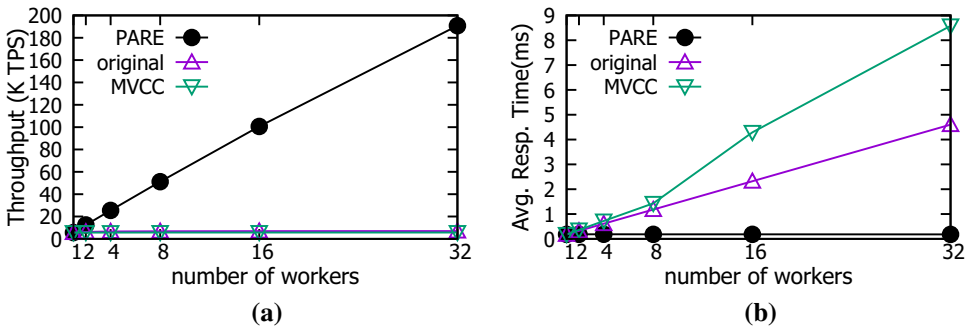

Fig. 3 Performance comparison under extremely contended HFT workloads. **a** Throughput, **b** average response time

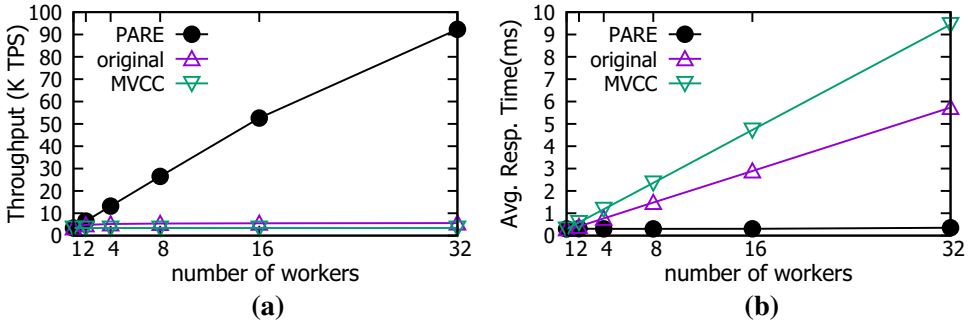

Fig. 4 Performance comparison under highly contended HFT workloads. **a** Throughput, **b** average response time

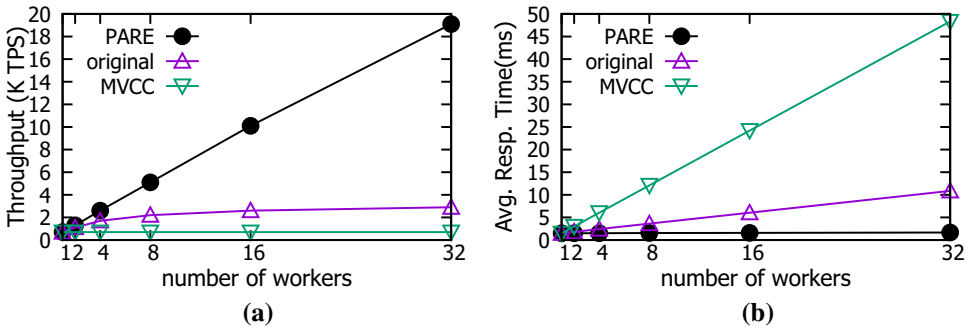

Fig. 5 Performance comparison under moderately contended HFT workloads. **a** Throughput, **b** average response time

more, Fig. 3b shows that PARE scales well without sacrificing transaction latency. On the other hand, the throughput of transaction pipeline increases by 25% from 1 worker to 32 workers and MVCC does not scale at all. This is because transaction pipeline always needs to delay due to harmful ordering. As for MVCC, every transaction will abort if its update operation is not the first to issue after the last transaction commits.

Under highly contended workloads, the five conflicting operations are interleaved with five non-conflicting operations ($n = 10$). As Fig. 4a illustrates, PARE still outperforms the others. It is not surprising to observe PARE achieves only roughly half the throughput of that achieved under extremely contended HFT workloads. This time each transaction contains 10 updates, which doubles the workload than the extremely contended workload. Transaction pipeline this time boosts 66% from 1 worker to 32 workers and From Fig. 4b, we can also see that given less than 8 workers, transaction pipeline scales better than MVCC, compared to Fig. 3b. This is because the execution on non-conflicting operations makes use of the delay.

Under moderately contended workloads, the five conflicting operations are interleaved with forty five non-conflicting operations ($n = 50$). In this way, each transaction is ten times and five times heavier than that under extremely and highly contended workloads. As Fig. 5a illustrates, PARE still outperforms other methods significantly and it can be more obviously observed that transaction pipeline outperforms MVCC. In detail, transaction pipeline speeds up 4.8x from 1 worker to 32 workers. This is also because given more non-conflicting operations, there are more opportunity for non-conflicting operations to use the delayed time that is wasted under previous workloads.

In addition to the HFT workloads, we also compare the performance on the original TPC-C benchmark, the widely adopted OLTP workload. We populate the database with 10 warehouses. In Fig. 6a, we can see that the performance of PARE and the original transaction pipeline matches and outperforms MVCC by up to 50%. Specifically, more workers result in larger performance gap between transaction pipeline and MVCC. This is because transaction under MVCC will abort due to the contended operations, i.e., update district table in new-order transaction and consecutive updates on warehouse and district tables in payment transaction. Rather, transaction pipeline and PARE are able to schedule the later update after the former update once the former update completes. This demonstrates that transaction pipeline indeed exploits more parallelism than MVCC. This also shows that the performance of transaction pipeline is not sensitive to single highly contended operation and two consecutive highly contended operations, which supports our defined "harmful ordering" because one contended operation and two consecutive contended operations do not form harmful ordering. Because there is no harmful ordering in TPC-C,[1] the little performance margin between PARE and original

[1] The stock-level update routine in new-order transaction exhibits a harmful ordering, but the conflicting operations are rarely contended due to the large item table.

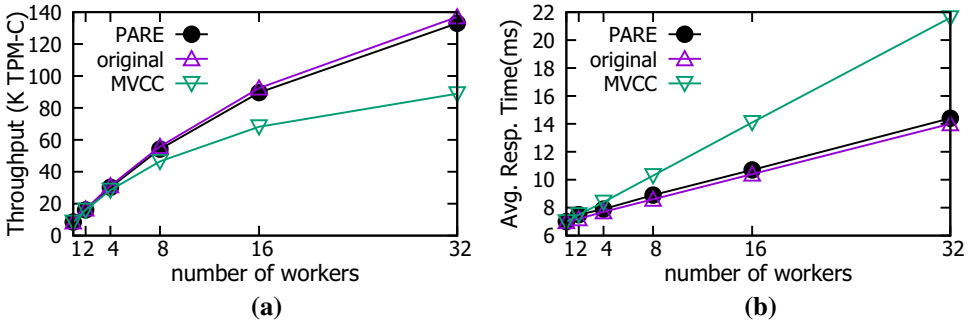

Fig. 6 Performance comparison under TPC-C. **a** Throughput, **b** average response time

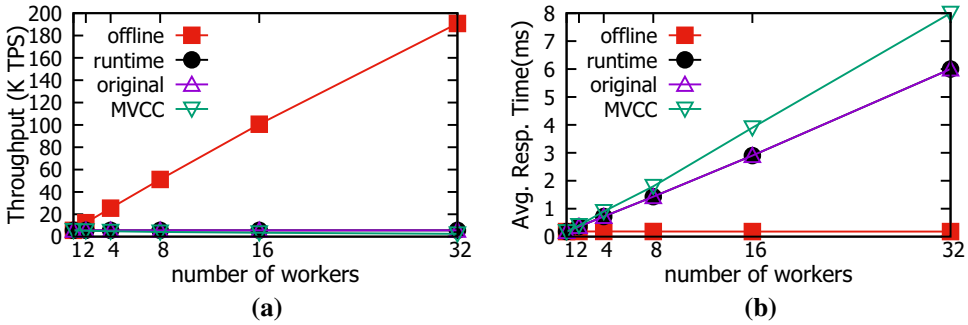

Fig. 7 Performance comparison under HFT workloads with off-line reordering. **a** Throughput, **b** average response time

transaction pipeline comes from the runtime overhead of PARE. We measure that the combination of extracting reordering blocks for each issued transaction and contention detection makes PARE less than 5% slower than the original transaction pipeline. This shows the PARE can also be applied to general workloads where harmful ordering may not exist.

So far, we have evaluated the runtime reordering approach. Then, we evaluate the off-line reordering on the both microbenchmark for HFT applications and TPC-C benchmark where our runtime reordering does not perform significantly faster than non-reordered approach.

Figure 7 shows the results on microbenchmark for HFT applications. In this set of experiments, there are two types of transactions. The first type of transactions updates update 10 tuples $a[0], a[1], \ldots, a[9]$ such that $a[i] = a[i-1] + 1$ $(1 \leq i \leq 9)$. The second type of transactions update the 10 tuples $a[9], a[8], \ldots, a[0]$ such that $a[i] = a[i+1] + 1$ $(8 \geq i \geq 0)$. In this way, there is only one reordering unit so that reordering does not take effect. As we can see, it is thus not surprising that our approach has the similar performance with the original transaction pipeline approach. In contrast, we adopt off-line reordering; the two types of transactions are arranged such that all transactions of the same type are executed simultaneously. In this way, there is no transaction bubble and thus our off-line reordering scheme outperforms existing approaches up to two or three magnitudes.

Figure 8 shows the performance after transactions are arranged by off-line reordering. We can see that our approach significantly outperforms the state-of-the-art approach, i.e., transaction pipeline, and the popular CC mechanism MVCC. This is because under 10 warehouses, transactions, especially the new-order and payment transactions under TPC-C benchmark, suffer from data contention. Rather, by reordering transactions in advance, transactions that access different tuples are executed simultaneously. In this way, transactions are executed as they are serially executed.

In summary, PARE receives huge performance benefit under HFT applications when other concurrency control mechanisms plummet under such workloads. In addition, under rarely contended workloads such as TPC-C, PARE matches the performance of transaction pipeline due to its low runtime overhead.

5.3 Detailed Performance

In this section, we further demonstrate the practicality of PARE by evaluating the overhead of PARE in terms of reordering block extraction, contention identification and memory consumption.

It may be worried that extracting reordering blocks from each issued transaction program is time-consuming, especially when harmful ordering cannot be guaranteed to appear in all workloads. To ease such concern, Tables 1 and 2 profile the amount of time spent on extracting reordering blocks and transaction execution under single thread, respectively. We can see that for both typical OLTP workloads TPC-C and TPC-E, the overhead for the

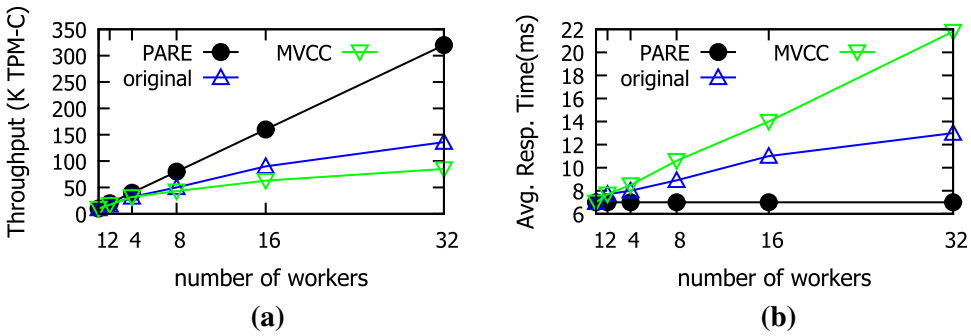

Fig. 8 Performance comparison under TPC-C with off-line reordering on 10 warehouse. **a** Throughput, **b** average response time

Table 1 Execution time (ms) spent on reordering block extraction on TPC-C benchmark

Transaction type	Extraction time (ms)	Execution time (ms)	Ratio (%)
New-order	0.28	7.1	3.9
Payment	0.26	8.4	3.1
Order-status	0.12	7.6	1.6
Delivery	0.07	6.2	1.1
Stock-level	0.02	12	0.2

Table 2 Execution time (ms) spent on reordering block extraction on TPC-E benchmark

Transaction type	Extraction time (ms)	Execution time (ms)	Ratio (%)
Broker-volume	0.01	4.3	0.2
Market-feed	0.21	3.8	**5.5**
Security-detail	0.08	2.6	3.1
Trade-order	0.36	7.4	4.8
Trade-status	0.03	7.7	0.4
Customer-position	0.05	4.9	1.0
Market-watch	0.17	9.6	1.8
Trade-lookup	0.07	9.4	0.7
Trade-result	0.41	8.3	4.9
Trade-update	0.15	13.9	1.1

Bold represents the onlytransaction type Market-Feed on which the extraction time occupies more than 5% of the executiontime

additional dependency analysis is very limited. This is due to our simple but efficient modeling of `readSet` and `writeSet` for SQL queries. Except for the Market-Feed transaction, which occupies 1% in TPC-E workload, the extraction of reordering blocks consumes less than 5% of the execution time. This demonstrates that the overhead of reordering block execution is limited and PARE is practical to typical OLTP workloads.

It is performance-critical for PARE to update the degree of contention of different objects timely. Figure 9a compares the real-time throughput between $g = 100$ ms and $g = 500$ ms with WS fixed to 1 s. It is not surprising to observe a performance trough per second when the highly contended object changes once per second. This is because when stale contended objects and latest contended objects are weighted equally in counters, reordering will mix the current highly contended objects with non-conflicting objects and thus reordering will not take effect. Before the performance trough, it is expected that the performance is not sensitive to the misjudgment of highly contended objects. This is because at this time, statements are somewhat reordered in increasing rather than the desired decreasing order of degree of contention and thus still eliminates harmful ordering. This validates our argument that arbitrary choice of increasing or decreasing order of contention does not matter. Figure 9b compares the throughput among different settings of WS with g fixed to 100 ms. We can see that WS = 200 ms and WS = 2 s perform inferior to WS = 1 s. On the one hand, too short window size loses much contention information and harmful ordering occurs. On the other hand, too large window size covering many contention change will mix highly contended objects with lowly contended objects many times and thus harmful ordering occurs many times.

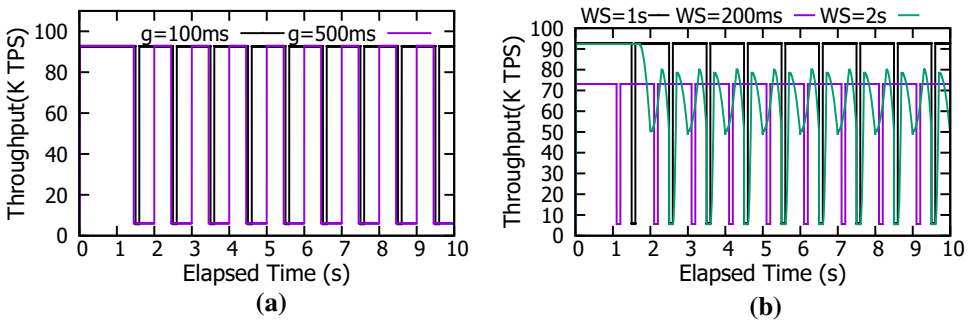

Fig. 9 Effect of counter setting. **a** Effect of granularity. **b** Effect of window size

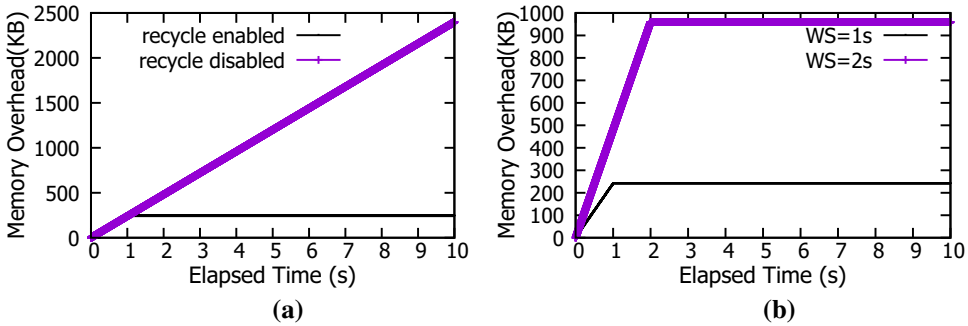

Fig. 10 Overhead of memory consumption. **a** Effect of counter recycling, **b** Effect of window size

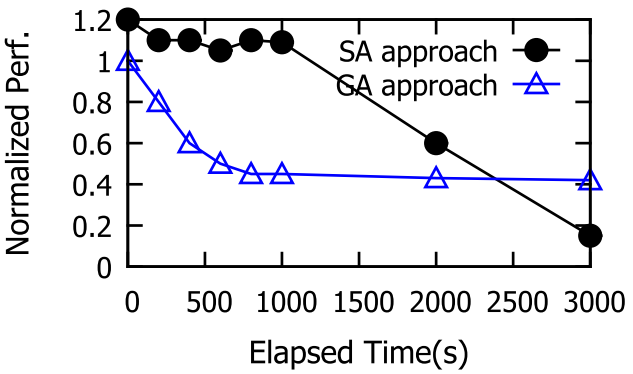

Fig. 11 Performance comparison: GA-based approach versus SA-based approach

Then, we evaluate the memory consumption of PARE. Figure 10a shows that without recycling strategy, the memory overhead will continuously increase because more and more objects will be accessed. Figure 10b shows that less window size can save more memory because it saves the occupied memory of each counter and recycles counters more frequently.

These sets of experiments demonstrate the robustness and practicality of our runtime reordering approach. Next, we evaluate our off-line reordering approach.

Figure 11 validates our choice of SA-based off-line reordering approach. We can see that the GA-based approach gets trapped into local minimum faster than SA-based approach.

6 Conclusion

This paper novelly proposes PARE to reorder transaction execution to improve the performance of the state-of-the-art concurrency control mechanism, transaction pipeline, under HFT applications. We for the first time observe and formulate the harmful ordering, which is brought with transaction pipeline and hidden under other concurrency control mechanisms. We deduce to reorder transaction execution in order of contention and further propose two mechanisms to extract reordering blocks and identify the degree of contention of objects. We also propose and formulate the problem of off-line reordering. We prove this is an NP-hard problem and present an off-line reordering approach to approximate the optimal reordering strategy. Experiments demonstrate that PARE outperforms the state-of-the-art concurrency control mechanism significantly on HFT applications in terms of both transaction throughput and latency, and the overhead is limited on typical OLTP benchmarks.

Acknowledgements This work is supported by Nature Science foundation of China Key Project No. 61432006 and The National Key Research and Development Program of China, No. 2016YFB1000702.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. http://www.investopedia.com/articles/investing/091615/world-high-frequency-algorithmic-trading.asp (2015)
2. Alvaro P, Conway N, Hellerstein JM, Marczak WR (2011) Consistency analysis in bloom: a calm and collected approach. Fifth biennial conference on innovative data systems research. Online proceedings, CIDR 2011. Asilomar, CA, USA, 9–12 January 2011, pp 249–260
3. Bailis P, Fekete A, Franklin MJ, Ghodsi A, Hellerstein JM, Stoica I (2014) Coordination avoidance in database systems. PVLDB 8(3):185–196
4. Barbará-Millá D, Garcia-Molina H (1994) The demarcation protocol: A technique for maintaining constraints in distributed database systems. VLDB J 3(3):325–353
5. Bernstein AJ, Gerstl DS, Lewis PM (1999) Concurrency control for step-decomposed transactions. Inf Syst 24(9):673–698
6. Bernstein PA, Shipman DW (1980) The correctness of concurrency control mechanisms in a system for distributed databases (sdd-1). ACM Trans Database Syst 5(1):52–68
7. Bernstein PA, Shipman DW, Rothnie JB Jr (1980) Concurrency control in a system for distributed databases (sdd-1). ACM Trans Database Syst 5(1):18–51
8. Brandt MW (2010) Chapter 5—portfolio choice problems. In: Handbook of Financial Econometrics: Tools and Techniques vol 1, pp 269–336
9. Cheung A, Arden O, Madden S, Solar-Lezama A, Myers AC (2014) Using program analysis to improve database applications. EEE Data Eng Bull 37:186–213
10. Cheung A, Madden S, Arden O, Myers AC (2012) Automatic partitioning of database applications. PVLDB 5(11):1471–1482
11. Cheung A, Madden S, Solar-Lezama A (2014) Sloth: being lazy is a virtue (when issuing database queries). In: SIGMOD'14, pp 931–942
12. Creamer GG, Freund Y (2013) Automated trading with boosting and expert weighting. Quant Finance 4(10):401–420
13. Dashti M, John SB, Shaikhha A, Koch C (2016) Repairing conflicts among MVCC transactions. CoRR. arXiv:1603.00542
14. Davies CT (1978) Data processing spheres of control. IBM Syst J 17(2):179–198
15. Eswaran KP, Gray JN, Lorie RA, Traiger IL (1976) The notions of consistency and predicate locks in a database system. Commun ACM 19(11):624–633
16. Faleiro JM, Abadi D, Hellerstein JM (2017) High performance transactions via early write visibility. PVLDB 10(5):613–624
17. Garcia-Molina H (1983) Using semantic knowledge for transaction processing in a distributed database. ACM Trans Database Syst 8(2):186–213
18. Garcia-Molina H, Salem K (1987) Sagas. SIGMOD Rec 16(3):249–259
19. Garcia-Molina H, Salem K (1987) Sagas. In: Proceedings of the 1987 ACM SIGMOD international conference on management of data, SIGMOD'87, New York. ACM, pp 249–259
20. Garey MR, Johnson DS (1990) Computers and intractability; a guide to the theory of NP-completeness. W. H. Freeman & Co., New York
21. Kirkpatrick S, Gelatt CD, Vecchi MP (1983) Optimization by simulated annealing. Science 220(4598):671–680
22. Kumar A, Stonebraker M (1988) Semantics based transaction management techniques for replicated data. In: Proceedings of the 1988 ACM SIGMOD international conference on management of data, SIGMOD'88, New York. ACM, pp 117–125
23. Kung HT, Robinson JT (1981) On optimistic methods for concurrency control. ACM Trans Database Syst 6(2):213–226
24. Larson P-A, Blanas S, Diaconu C, Freedman C, Patel JM, Zwilling M (2011) High-performance concurrency control mechanisms for main-memory databases. PVLDB 5(4):298–309
25. Li C, Porto D, Clement A, Gehrke J, Preguiça N, Rodrigues R (2012) Making geo-replicated systems fast as possible, consistent when necessary. In: Proceedings of the 10th USENIX conference on operating systems design and implementation, OSDI'12. USENIX Association, Berkeley, pp 265–278
26. Marlowe TJ, Ryder BG (1990) Properties of data flow frameworks: a unified model. Acta Inf 28(2):121–163
27. Neumann T, Mühlbauer T, Kemper A. Fast serializable multi-version concurrency control for main-memory database systems. SIGMOD'15, pp 677–689
28. Olston C, Loo BT, Widom J (2001) Adaptive precision setting for cached approximate values. In: Proceedings of the 2001 ACM SIGMOD international conference on management of data, SIGMOD'01. ACM, New York, pp 355–366
29. Olston C, Widom J (2000) Offering a precision-performance tradeoff for aggregation queries over replicated data. In: Proceedings of the 26th international conference on very large data bases, VLDB'00. Morgan Kaufmann Publishers Inc, San Francisco, pp 144–155
30. Pandis I, Johnson R, Hardavellas N, Ailamaki A (2010) Data-oriented transaction execution. PVLDB 3(1–2):928–939
31. Pu C, Leff A (1991) Replica control in distributed systems: as asynchronous approach. In: Proceedings of the 1991 ACM SIGMOD international conference on management of data, SIGMOD'91. ACM, New York, pp 377–386
32. Ramamritham K, Pu C (1995) A formal characterization of epsilon serializability. IEEE Trans Knowl Data Eng 7(6):997–1007
33. Roy S, Kot L, Bender G, Ding B, Hojjat H, Koch C, Foster N, Gehrke J (2015) The homeostasis protocol: avoiding transaction coordination through program analysis. In: Proceedings of the 2015 ACM SIGMOD international conference on management of data, SIGMOD'15. ACM, New York, pp 1311–1326
34. Shasha D, Llirbat F, Simon E, Valduriez P (1995) Transaction chopping: algorithms and performance studies. ACM Trans Database Syst 20(3):325–363
35. Shen W, Wang J, Jiang Y-G, Zha H. Portfolio choices with orthogonal bandit learning. In: IJCAI'15, pp 974–980
36. Wang Z, Mu S, Cui Y, Yi H, Chen H, Li J (2016) Scaling multicore databases via constrained parallel execution. In: Proceedings of the 2016 international conference on management of data, SIGMOD'16, pp 1643–1658
37. Weikum G, Vossen G (2001) Transactional information systems: theory, algorithms, and the practice of concurrency control and recovery. Elsevier, Amsterdam
38. Whitley D (1994) A genetic algorithm tutorial. Stat Comput 4(2):65–85
39. Wolfe MJ, Shanklin S, Ortega L (1995) High performance compilers for parallel computing. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA
40. Wu K-L, Yu PS, Pu C (1992) Divergence control for epsilon-serializability. In: Proceedings of the 8th international conference on data engineering. IEEE Computer Society, pp 506–515, Washington
41. Wu Y, Chan C-Y, Tan K-L (2016) Transaction healing: scaling optimistic concurrency control on multicores. In: SIGMOD'16, pp 1689–1704
42. Yan C, Cheung A (2016) Leveraging lock contention to improve oltp application performance. PVLDB 9(5):444–455
43. Yu H, Vahdat A (2000) Design and evaluation of a continuous consistency model for replicated services. In: Proceedings of the 4th conference on symposium on operating system design & implementation, vol 4, OSDI'00. USENIX Association, Berkeley