# Leveraging Lock Contention to Improve OLTP Application Performance

Cong Yan
University of Washington
congy@cs.washington.edu

Alvin Cheung
University of Washington
akcheung@cs.washington.edu

## ABSTRACT

Locking is one of the predominant costs in transaction processing. While much work has focused on designing efficient concurrency control mechanisms, not much has been done on understanding how transaction applications issue queries and leveraging application semantics to improve application performance. This paper presents QURO, a query-aware compiler that automatically reorders queries in transaction code to improve performance. Observing that certain queries within a transaction are more contentious than others as they require locking the same tuples as other concurrently executing transactions, QURO automatically changes the application such that contentious queries are issued as late as possible. We have evaluated QURO on various transaction benchmarks, and our results show that QURO-generated implementations can increase transaction throughput by up to 6.53×, while reduce transaction latency by up to 85%.

## 1. INTRODUCTION

From ticket reservation systems, online shopping, to banking applications, we interact with online transaction processing(OLTP) applications on a daily basis. These applications are often implemented using database transactions, where each transaction consists of multiple read and write queries to the database management system (DBMS) that stores persistent data. One of the goals of OLTP applications is to handle large number of concurrent transactions simultaneously. However, since multiple transactions might access the same tuple stored in the DBMS at the same time, some form of concurrency control must be implemented to ensure that all transactions get a consistent view of the persistent data.

*Two-phase locking* (2PL) [7, 8] is one of the most popular concurrency control mechanisms implemented by many DBMSs. In 2PL, each data element (e.g., a tuple or a partition) stored in the database is associated with a read and a write lock, and a transaction is required to acquire the appropriate lock associated with the given database element before operating on it. For example, while multiple transactions can be holding the read lock on the same data concurrently, only one transaction can hold the write lock. When a transaction cannot acquire a lock, its pauses execution until the lock is released. To avoid deadlocks, (strict) 2PL requires that a transaction not request additional locks once it releases any lock. Thus, as each transaction executes, it goes through an expanding phase where locks are acquired and no lock is released, followed by a shrinking phase where locks are released and no locks are acquired.

Unfortunately, not all locks are created equal. While each transaction typically operates on different elements stored in the DBMS, it is often the case that certain elements are more contentious than others, i.e., they tend to be read from or written to by multiple concurrent transactions. As an illustration, imagine an application where all transactions need to update a single tuple (such as a counter) among other operations. If each transaction starts by first acquiring the write lock on the counter tuple before acquiring locks on other data elements, then essentially all but one of the transactions can make progress while the rest are blocked, even though other transactions could have made further progress if they first acquired locks on other data. As a result, each transaction takes a longer time to execute, and the overall system throughput suffers. This is exacerbated in main-memory databases. Since the transaction no longer need to access the disk, most of transaction running time is spent on executing queries, and the long lock waiting time is likely to become the predominant performance bottleneck.

One way to avoid the above problem is to reorder the queries in each transaction such that operations on the most contentious data elements are performed last. Indeed, as our results show, doing so can significantly improve application performance. Unfortunately, reordering queries in transaction code raises various challenges:

- OLTP applications are typically written in a high-level programming language such as C or Java, and compilers for these languages treat queries as black-box library calls. As such, they are unaware of the fact that these calls are executing queries against the DBMS, let alone ordering them during compilation based on the level of contention.
- DBMS only receives queries from the application as it executes and does not understand how the queries are semantically connected. As such, it is very difficult for the DBMS to reorder queries during application execution, since the application will not issue the next query until the results from the current one have been returned.
- Queries in a transaction are usually structured based on application logic. Reordering them manually will make the code difficult to understand. Furthermore, developers need to preserve the data dependencies among different queries as they reorder them, making the process tedious and error-prone.

In this paper we present QURO, a query-aware compiler that automatically reorders queries within transaction code based on lock contention while preserving program semantics. To do so, QURO first profiles the application to estimate the amount of contention among queries. Given the profile, QURO then formulates the reordering problem as an Integer Linear Programming (ILP) problem, and uses the solution to reorder the queries and produces an application binary by compiling the reordered code using a standard compiler.

This paper makes the following contributions:

- We observe that the order of queries in transaction code can drastically affect performance of OLTP applications, and that current general-purpose compiler frameworks and DBMSs do not take advantage of that aspect to improve application performance.
- We formulate the query reordering problem using ILP, and devise a number of optimizations to make the process scale to real-world transaction code.
- We implemented a prototype of QURO and evaluated it using popular OLTP benchmarks. When evaluated on main-memory DBMS implementations, our results show that the QURO-generated transactions can improve throughput by up to 6.53×, while reducing the average latency of individual transactions by up to 85%.

The rest of this paper is organized as follows. We first illustrate query reordering with an example and give an overview of QURO in Section 2. Then we describe the preprocessing performed by QURO in Section 3 followed by details of the reordering algorithm in Section 4 and profiling in Section 5. We present our experiment results using three OLTP benchmarks in Section 6, discuss related work in Section 7, and then conclude.

## 2. OVERVIEW

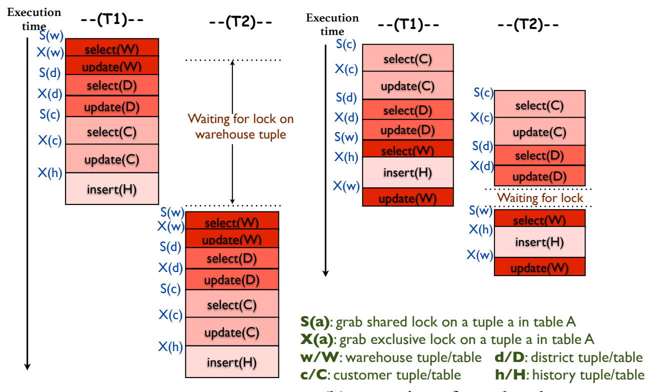
In this section we discuss query reordering in transaction code using an example and describe the architecture of QURO. To motivate, Listing 1 shows an excerpt from an open-source implementation [3] of the payment transaction from the TPC-C benchmark [21], which records a payment received from a customer. In the excerpt, the code first finds the warehouse to be updated with payment on line 1 and subsequently updates it on line 2. Similarly, the district table is read and updated on lines 3 and 4. After that the code updates the customer table. The customer can be selected by customer id, or customer name. If the customer has good credit, only the customer balance will be updated, otherwise the detail of this transaction will be appended to the customer record. Finally it inserts a tuple into the history table recording the change.

```
1  w_name = select("warehouse");
2  update("warehouse", w_name);
3  d_name = select("district");
4  update("district");
5  if (c_id == 0) {
6    c_id = select("customer", c_name);
7  }
8  c_credit = select("customer", c_id);
9  if (c_credit[0] == ('G')) {
10   update("customer", c_id, w_id);
11 } else {
12   c_id = "..." + w_id + c_id + "...";
13   update("customer", c_id);
14 }
15 insert("history", w_name, d_name, ...);
```

Listing 1: Original code fragment from TPC-C payment transaction. Hereselect("t", v) represents a selection query on table t that uses the value of program value v as one of its parameters, likewise for update and insert.

```
1  if (c_id == 0) {
2    c_id = select("customer", c_name);
3  }
4  c_credit = select("customer", c_id);
5  if (c_credit[0] == ('G')) {
6    update("customer", c_id, w_id);
7  }else{
8    c_id = "..." + w_id + c_id + "...";
9    update("customer", c_data);
10 }
11 d_name = select("district");
12 update("district");
13 w_name = select("warehouse");
14 insert("history", w_name, d_name, ...);
15 update("warehouse", w_name);
```

Listing 2: Reordered code fragment from Listing 1


Execution time
--(T1)--
--(T2)--
S(w) X(w) S(d) X(d) S(c) X(c) X(h)
select(W)
update(W)
select(D)
update(D)
select(C)
update(C)
insert(H)
Waiting for lock on warehouse tuple
Waiting for lock
S(a): grab shared lock on a tuple a in table A
X(a): grab exclusive lock on a tuple a in table A
w/W: warehouse tuple/table
d/D: district tuple/table
c/C: customer tuple/table
h/H: history tuple/table

(a) execution of original payment (b) execution of reordered payment

Figure 1: Comparison of execution between original and reordered implementation of the payment transaction. The darker the color, more likely the query is going to access contentious data.

As written, the implementation shown in Listing 1 performs poorly due to high data contention. In a typical TPC-C setup, the warehouse table contains the fewest tuples. Hence as the number of concurrent transactions increases, the chance that multiple transactions will update the same warehouse table tuple (on line 2 in Listing 1) also increases, and this will in turn increase lock contention and the amount of time spent in executing each transaction. This is illustrated pictorially in Figure 1a with two concurrent transactions that try to update the same tuple in the warehouse table. When executed under 2PL, each transaction attempts to acquire the exclusive lock on the same warehouse tuple before trying to update it, and will only release the lock when the transaction commits. In this case T1 acquires the lock, blocking T2 until T1 commits. Thus the total amount of time needed to process the two transactions is close to the sum of these two transactions executed serially.

However, there is another way to implement the same transaction, as shown in Listing 2. Rather than updating the warehouse (i.e., the most contentious) table first, this implementation updates the customer's balance first, then updates the district and warehouse tables afterwards. This implementation has the same semantics as that shown in Listing 1, but with very different performance characteristics, as shown in Figure 1b. By performing the updates on warehouse table at a later time (line 15), transaction T1 delays acquiring the exclusive lock on the warehouse tuple, allowing T2 to proceed with operations on other (less contentious) tuples concurrently with T1. Comparing the two implementations, reordering increases transaction concurrency, and reduces the total amount of time needed to execute the two transactions.

While reordering the implementation shown in Listing 1 to Listing 2 seems trivial, doing so for general transaction code is not an easy task. In particular, we need to ensure that the reordered code does not violate the semantics of the original code in terms of data dependencies. For instance, the query executed on line 15

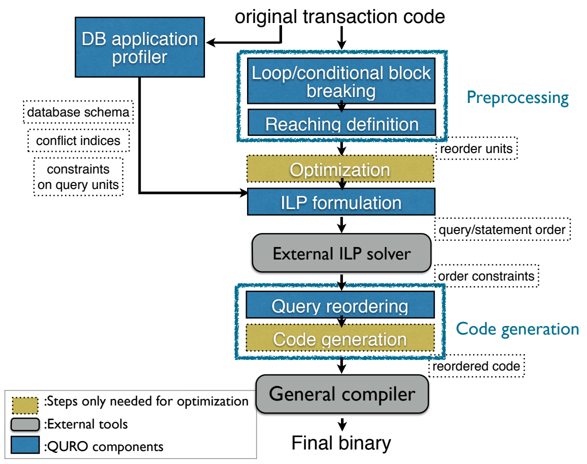


Figure 2: Architecture and workflow of QURO

in Listing 1 can only be executed after the queries on lines 1 and 3, because it uses `w_name` and `d_name`, which are results of those two queries. Besides such data dependencies on program variables (as `w_name` and `d_name` mentioned above), there may also be dependencies on database tuples. For example, the query on line 3 reads a database tuple which the query on line 4 later updates, so the two queries have a data dependency on that tuple. While such dependencies can be inferred manually, doing so for longer and more complex transactions puts excessive burden on developers. Unfortunately, typical compilers do not perform such aggressive transformations, as they treat queries as external function calls.

QURO is designed to optimize transaction code by reordering query statements according to the lock contention each query incurs. To use QURO, the developer first demarcates each transaction function with `BEGIN_TRANSACTION` and `END_TRANSACTION`.[1] Our current prototype is built on Clang and accepts transaction code written in C/C++, and QURO assumes that the transactions use standard APIs to issue queries to the DBMS (e.g., ODBC).

The architecture of QURO is shown in Figure 2. After parsing the input code, QURO first generates an instrumented version to profile the running time of each query and gathers information about query contention. QURO deploys the instrumented version using the same settings as the original application and runs it for a user-specified amount of time. After profiling, QURO assigns a contention index to each query to be used in subsequent reordering steps. QURO also collects information about database schema, which is used to generate order constraints.[2]

After profiling, QURO performs a number of preprocessing steps on the input code. First, it performs reaching definition analysis for each transaction function. Reaching definition analysis is used to infer data dependencies among different program variables. After that, QURO performs loop fission and breaks compound statements (e.g., conditionals with multiple statements in their bodies) into smaller statements that we refer to as *reorder units*. This is to expose more reordering opportunities, to be discussed in Section 3.

QURO next uses the results from profiling and preprocessing to discover order constraints on queries before reordering. Data dependencies among program variables or database tuples may induce order constraints. QURO first uses the results from reaching definition analysis during preprocessing to construct order constraints based on program variables. QURO then analyzes the queries with the database schemas to infer order constraints among database tuples, e.g., if two queries may update the same tuple in the same table, the order of these queries cannot be changed. Reordering is then formulated as an ILP problem based on the ordering constraints, and solving the program returns the optimal way to implement the transaction subject to the data dependencies given the contention indices. While a simple implementation is to encode each reorder unit as a variable in the ILP, solving the ILP might take a substantial amount of time, especially for transactions with many lines of code. In Section 4.4 we propose an optimization to make this tractable and more efficient. After receiving the order of queries from ILP solver, QURO restructures the program, and uses a general-purpose compiler to produce the final binary of the application.

In the next sections we discuss each step involved in the reordering process in detail.

## 3. PREPROCESSING

Before statement reordering, QURO parses the input transaction code into an abstract syntax tree (AST) and performs two preprocessing tasks: breaking the input code into small units to be reordered, and analyzing the data dependencies among program variables. In this section we describe the details of these two steps.

### 3.1 Breaking Loop and Conditional Statements

The purpose of the first task is to enable more queries to be reordered. For loop and conditional statements, it is hard to change the ordering of statements within each block, as each such statement can be nested within others. Disregarding the bodies inside loop and conditional statements and treating the entire statement as one unit limits the number of possible ways that the code can be reordered. In fact, as we will demonstrate in Section 6, breaking loop and conditional statements is essential to improve the performance for many transaction benchmarks.

For loop statements, QURO applies loop fission, a well-studied code transformation technique [22], to split an individual loop nest into multiple ones. The basic idea is to split a loop with two statements $S_1$ and $S_2$ in its body into two individual loops with the same loop bounds if:

1. There is no loop carry dependency. If $S_1$ defines a value that will be used by $S_2$ in later iterations, then $S_1$ and $S_2$ have to reside in the same loop.

2. There is no data dependency between the two statements to be split. If $S_1$ defines a value that is used by $S_2$ in the same iteration, and $S_1$ will rewrite that value in some later iteration, then $S_1$ and $S_2$ cannot be split.

3. The statements do not affect the loop condition. If $S_1$ writes some value that affects the loop condition, then all the statements within the loop cannot be split into separate loops.

We apply the fission algorithm discussed in prior work [23] and handle nested loops by checking the fission condition iteratively.

Listing 3 and 4 show an example of a loop before and after loop fission. Line 2 defines `var1` at every iteration, but since line 3 uses `var1`, so lines 2 and 3 have to reside in the same loop. One the other hand, line 4 defines `var2[i]` at iteration `i`, and line 5 uses `var2[i]`. Since line 4 does not redefine `var2[i]` in other iterations, lines 4 and 5 can be split. Notice in the example that there are no dependencies between lines 2 and 3 (when considered in tandem),

[1]QURO currently assumes that each transaction is implemented within a single function and leave inter-procedural analysis as future work.

[2] QURO assumes the database schema doesn't change when transactions are running.

line 4, and line 5. Since none of these statements affect the loop condition, they can be safely split into individual loops.

```
for(i=0; i<n; i++){
  var1 = select("table1");
  update("table1", var1+1);
  var2[i] = select("table2");
  update("table2", var2[i]+1);
}
```

Listing 3: Loop fission example

```
for(i=0; i<n; i++){
  var1 = select("table1");
  update("table1", var1+1);
}
for(i=0; i<n; i++){
  var2[i] = select("table2");
}
for(i=0; i<n; i++){
  update("table2", var2[i]+1);
}
```

Listing 4: Loop code after fission

Similarly, conditional statements under a single condition can also be split. In general, conditional statements $S_1$ and $S_2$ under the same Boolean condition can be split into multiple ones with the same condition if:

1. Neither statement affects the condition.
2. The condition does not have any side effects, for example, changing the value of a program variable that is used by any other statements in the program.

As an illustration, breaking the conditional block will transform line 9 to line 14 from Listing 1 into Listing 5.

```
if(c_credit[0] == ('G')) {
  update("customer", c_id, w_id);
}
if(!(c_credit[0] == ('G')){
  c_id = "..." + w_id + c_id + "...";
}
if(!(c_credit[0] == ('G')){
  update("customer", c_id);
}
```

Listing 5: Example of breaking conditional statements

## 3.2 Analyzing Reaching Definitions

After breaking loop and conditional statements, QURO analyzes the data dependencies among statements by computing reaching definitions. Formally speaking, a definition of a program variable v by program statement $S_1$ *reaches* a subsequent (in terms of control flow) statement $S_2$ if there is a program path from $S_1$ to $S_2$ without any intervening definition of v. We compute reaching definitions for each program statement using a standard dataflow algorithm [19]. The process is straightforward for most types of program statements. For function calls, however, we distinguish between those that are database calls (e.g., those that issue queries), for which we precisely model the def-use relationships among the function parameters and return value, and other functions, for which we conservatively assume that all parameters with pointer types or parameters that are passed by reference are both defined and used by the function.

## 4. REORDERING STATEMENTS

The preprocessing step normalizes the input code into individual statements that can be rearranged. We call each such statement, including an assignment, a function call statement, or a loop/conditional block that cannot be further decomposed using methods as described in Section 3.1, a *reorder unit*. In this section we discuss how we formulate the statement reordering problem by making use of the information collected during preprocessing.

## 4.1 Generating Order Constraints

As discussed in Section 2, the goal of reordering is to change the structure of the transaction code such that the database queries are issued in an increasing order of lock contention. However, doing so is not always possible because of data dependencies among the issued queries. For instance, the result of one query might be used as a parameter in another query, or the result of one query might be passed to another query via a non-query statement. Furthermore, two queries might update the same tuple, or a query might update a field that is a foreign key to a table involved in another query. In all such cases the two queries cannot be reordered even though one query might be more contentious than the other.

Formally, we need to preserve the data dependencies 1) among the program variables, and 2) among the database tuples, when restructuring queries in transaction code.[3] The reaching definition analysis from preprocessing infers the first type of data dependency, while analyzing the queries using database schema information infers the second type. These data dependencies set constraints on the order of reorder units. In the following we discuss how these constraints are derived.

**Dependencies among program variables:**

1. Read-after-write (RAW): Reorder unit $U_i$ uses a variable that is defined by another unit $U_j$. *Formal constraint*: Reorder unit $U_i$ should appear before $U_j$ in the restructured code.
2. Write-after-read (WAR): $U_j$ uses a variable that is later updated by another unit $U_k$. *Formal constraint*: If both $U_i$ and $U_k$ define the same variable $v$, and $U_j$ uses $v$ defined by $U_i$, then $U_k$ cannot appear between $U_i$ and $U_j$. If no such $U_i$ exists, as in the case of $v$ being a function parameter that is used by $U_j$, then $U_k$ should appear after $U_j$ in the restructured code.
3. Write-after-write (WAW): $v$ is a global variable or a function parameter that is passed by reference, and both $U_i$ and $U_l$ define $v$, with $U_l$ being the last definition in the body of the function. *Formal constraint*: $U_i$ should appear before $U_l$ in the restructured code. If $v$ is a global variable, we assume that program locks are in place to prevent race conditions.

We use the code shown in Listing 1 to illustrate the three kinds of dependency discussed above. For instance, the insertion into the history table on line 15 uses variable `w_name` defined on line 1 and `d_name` defined on line 3. Thus, there is a RAW dependency between line 15 along with line 1 and line 3. Hence, a valid reordering should always place line 15 after lines 1 and 3. Meanwhile, the update on customer table on line 8 uses variable `c_id`, which is possibly defined on line 6 or is passed in from earlier code. Furthermore, line 12 redefines this variable. Thus, there is a WAR dependency between line 12 and line 8, meaning that in a valid ordering line 12 should not appear between line 6 and line 8.

**Dependencies among database tuples:**

[3] QURO currently does not model exception flow. As such, the reordered program might have executed different number of statements when an exception is encountered as compared to the original program.

1. Operations on the same table: queries $Q_i$ and $Q_j$ operate on the same table, and at least one of the queries performs a write (i.e., update, insert, or delete).
2. View: query $Q_i$ operates on table $T_i$ which is a view of table $T_j$, and query $Q_j$ operates on $T_j$. At least one of the queries performs a write.
3. Foreign-key constraints: $Q_i$ performs an insert/delete on table $T_i$, or change in the key field $C_i$ of $T_i$. $Q_j$ operates on column $C_j$ in table $T_j$, where $C_j$ is a foreign key to $T_i$ which includes a column that column $C_i$ in $T_i$ references.
4. Triggers: $Q_i$ performs an insert or a delete on table $T_i$, which triggers a set of pre-defined operations that alter $T_j$. Query $Q_j$ operates on table $T_j$.
   *Formal constraint*: In all of the above cases, the order of $Q_i$ and $Q_j$ after reordering should remain the same as in the original program.

For the code example in Listing 1, the queries on line 1 and line 2 operate on the same table, so they cannot be reordered or else the query on line 1 will read the wrong value.

To discover the dependencies among database tuples, QURO analyzes each database query from the application to find out which tables and columns are involved in each query. Then QURO utilizes the database schema obtained during preprocessing to discover the dependencies listed above.

## 4.2 Formulating the ILP Problem

QURO formulates the reordering problem as an instance of ILP. As mentioned in Section 2, QURO first profiles the application to determine how contentious the queries are among the different concurrent transactions. Profiling gives a conflict index $c$ to each query: the larger the value, the more likely that the query will have data conflict with other transactions. The conflict index for non-query statements is set to zero. Under this setting, the goal of reordering is to rearrange the query statements in ascending conflict index, subject to the order constraints described above.

Concretely, assume that there are $n$ reorder units, $U_1$ to $U_n$, in a transaction, with conflict indices $c_1$ to $c_n$, respectively. We assign a positive integer variable $p_i$ to represent the final position of each of the $n$ reorder units. The order constraints derived from data dependencies can be expressed as the following constraints in the ILP problem:

- $p_i \le n, i \in [1, n]$, such that each unit is assigned a valid position.
- $p_i \ne p_j, i \ne j$, such that each unit has a unique position.
- $p_i < p_j$, if there is a RAW dependency between $U_i$ and $U_j$.
- $(p_k < p_i) \mid (p_k > p_j)$, if there is a WAR dependency between $U_j$, $U_k$ and $U_i$, where $U_i$ redefines a variable that $U_j$ uses.
- $p_k > p_j, k \ne j$, if there is a WAR dependency between $U_j$ and $U_k$, and and there is no intervening variable redefinition.
- $p_l > p_i, i \ne l$, if there is a WAW dependency between $U_l$ and $U_i$, and $U_l$ is the last definition of a global variable or a return value in the transaction.
- $p_i < p_j, i < j$, if $U_i$ contains query $Q_i$, query $Q_j$ is in reorder unit $U_j$, and the order of $Q_i$ and $Q_j$ needs to be preserved due to data dependencies among database tuples referenced by $Q_i$ and $Q_j$.

Given these, the objective of the ILP problem is to maximize:

$$\sum_{i=1}^{n} p_i * c_i$$

Solving the program will give us the value of $p_1, \ldots p_n$, which indicates the position of each reorder unit in the final order. For big $c_i$, $p_i$ will have high value without violating the constraints, indicating that the corresponding statement will appear late in the transaction.

As an example, the code shown in Listing 1 generates the following constraints. Here we assume that there are no view, trigger or foreign-key relationships between any two tables used in the transaction:

$p_i \le 11, i \in \{1, 2, 3, 4, 6, 8, 10, 11, 13, 14, 16\}$

$p_i \ne p_j, i \ne j$

$p_1 < p_{15}$, RAW on variable `w_name`

$p_3 < p_{15}$, RAW on variable `d_name`

$\vdots$

$(p_{12} < p_6) \mid (p_{12} > p_8)$, WAR on variable `c_id`

$(p_{12} < p_6) \mid (p_{12} > p_{10})$, WAR on variable `c_id`

$\vdots$

$p_1 < p_2$, Query order constraint, as both query $Q_1$(in $U_1$) and $Q_2$(in $U_2$) operate on the warehouse table

$p_3 < p_4$, Query order constraint, as both query $Q_3$(in $U_3$) and $Q_4$(in $U_4$) operate on the district table

With the conflict indices for query-related units as shown in Table 1, QURO will give an reordering of the units shown in Listing 2.

Table 1: conflict index for each reorder units in Listing 1

| unit | 1 | 2 | 3 | 4 | 6 | 8 | 10 | 13 | 15 |
|---|---|---|---|---|---|---|---|---|---|
| $c$ | 500 | 510 | 100 | 110 | 50 | 50 | 60 | 60 | 10 |

## 4.3 Removing Unnecessary Dependencies

Solving ILP problems is NP-hard in general. In the following sections we describe two optimizations to reduce the number of constraints and variables in the ILP problem, and we evaluate these techniques in Section 6.9.

First, we describe a technique to reduce the number of ILP constraints. Consider the example shown in Listing 6.

```
v = select("table1");
update("table2", v);
v = select("table3", v);
```

Listing 6: Code example to illustrate renaming

```
v = select("table1");
v_r = v;
v_r = select("table3", v_r);
update("table2", v);
```

Listing 7: Code example after renaming

If the update on line 2 is more contentious than the query on line 3, then QURO's reordering algorithm would place line 3 before line 2. However, RAW and WAR lead to the following constraints:

$$p_1 < p_2;\ p_1 < p_3; \qquad \text{(RAW)}$$
$$(p_3 < p_1) \vee (p_3 > p_2); \qquad \text{(WAR)}$$

meaning that reordering will violate data dependency. However, we can remove the WAR dependency with variable renaming by creating a new variable `v_r`, assigning `v` to it before `v` is used, and replacing subsequent uses of `v` to be `v_r`. This allows us to restructure the code to that shown in Listing 7.

In general, WAR and WAW are *name* dependencies (in contrast to *data* dependencies, as in the case of RAW) that can be removed by renaming. Doing so reduces the number of ILP constraints, and

```
1 v=select("table1");
2 if(cond)
3  v=select("table2");
4 update("table3", v);
```

Listing 8: Renaming example with multiple reaching definitions

```
v=select("table1");
v_r2=v;
if(cond){
 v_r1=select("table2");
 v_r2=v_r1;
}
update("table3", v_r2);
```

Listing 9: Example after renaming

makes more queries able to be reordered. As shown in the example above. However, if the variable v involved in a WAR or WAW dependency satisfies any of the following conditions, then removing WAR and WAW will be more complicated:

- v *is not of a primitive type (e.g., a pointer or class object).* Since renaming requires cloning the original variable, it might be impossible to do so for non-primitive types as they might contain private fields. Besides, cloning objects can slow down the program. Thus, we do not rename non-primitive variables and simply encode any WAW and WAR dependencies involving these variables in the ILP.
- v *is both used and defined in the same reorder unit.* If the same variable is both defined and used in the same reorder unit, such as `f(v)` where v is passed by reference, then replacing v with `v_r` in the statement will pass an uninitialized value to the call. To handle this issue, the value of v should be first copied to `v_r` before the call. This is done by inserting an assign statement before the statement containing the variable to be renamed: `v_r = v; f(v_r)`.
- *Multiple definitions reach the same use of* v. In this case, if any of the definitions is renamed, then all other definitions will need to be renamed as well. We use the example in Listing 8 to illustrate. The definitions of v on lines 1 and 3 both reach the update on line 4. If v needs to be renamed on line 3 due to data dependency violation (not shown in the code), then we create a new variable `v_r2` to hold the two definitions so that both `v_r1` and v reach the use at the update query, as shown in Listing 9.

## 4.4 Shrinking Problem Size

Transactions that implement complex program logic can contain many statements, which generate many reordering units and variables in the ILP problem. This can cause the ILP solver to run for a long time. In this section we describe an optimization to reduce the number of variables required in formulating the ILP problem.

Since our goal is to reorder database query related reorder units, we could formulate the ILP problem by removing all variables associated with non-query related reorder units from the original formulation. This does not work, however, as dropping such variables will mistakenly remove data dependencies among query related reorder units as well. For example, suppose query $Q_3$ contained in $U_3$ uses as parameter the value that is computed by a reorder unit $U_2$ containing no query, and $U_2$ uses a value that is returned by query $Q_1$ in $U_1$. Dropping non-query related variables in the ILP (in this case $p_2$ that is associated with $U_2$) will also remove the constraint between $p_1$ and $p_3$, and that will lead to an incorrect order. The order will be correct, however, if we append extra constraints to the ILP problem(in this case $p_1 < p_3$) to make up for the removal of the non-query related variables. To do so, we take the original set of ILP constraints and compute transitively the relationship between all pairs of query related reorder units. First, we define an auxiliary Boolean variable $x_{ij}$ for $i < j$, where $i, j \in [1, n]$, to indicate that $p_i < p_j$. Then, we rewrite each type of constraints in the original ILP into Boolean clauses using the auxiliary Boolean variables as follows:

- $p_i < p_j \Rightarrow x_{ij} = true$
- $(p_k < p_i) \mid (p_k > p_j) \Rightarrow \begin{cases} (x_{kj} \rightarrow x_{ki}) = true, \text{if } k < i \\ (x_{ik} \rightarrow x_{jk}) = true, \text{if } k > j \end{cases}$
- $p_k > p_j \Leftrightarrow x_{jk} = true$
- $p_l > p_i \Leftrightarrow x_{il} = true$

After that, we combine all rewritten constraints as a conjunction $E$. Clauses in $E$ can include either a single literal, such as $x_{ij}$, or two literals, as in $x_{kl} \rightarrow x_{mn}$:

$$E = x_{ij} \wedge ... \wedge (x_{kl} \rightarrow x_{mn}) \wedge ...$$

Note that any ordering that satisfies all the constraints from the original ILP will set the values of the corresponding auxiliary Boolean variables such that $E$ evaluates to true.

We now use the existing clauses in $E$ to infer new clauses by applying the following inference rules:

- $(x_{ij} \rightarrow x_{kl}) \wedge (x_{kl} \rightarrow x_{uv}) \Rightarrow x_{ij} \rightarrow x_{uv}$
- $x_{ij} \wedge x_{jk} \Rightarrow x_{ik}$
- $x_{ij} \wedge (x_{ij} \rightarrow x_{kl}) \Rightarrow x_{kl}$
- $(x_{ij} \wedge x_{jk}) \Rightarrow x_{ik}$
- $((x_{ij} \wedge x_{kl}) \rightarrow x_{uv}) \wedge (x_{uv} \rightarrow x_{mn}) \Rightarrow (x_{ij} \wedge x_{kl}) \rightarrow x_{mn}$
- $((x_{ij} \wedge x_{kl}) \rightarrow x_{uv}) \wedge (x_{mn} \rightarrow x_{ij}) \Rightarrow (x_{mn} \wedge x_{kl}) \rightarrow x_{uv}$

Applying each inference rule generates a new clause, and the process continues until no new clauses can be generated. All clauses are then collected into a conjunction $E'$ with the form:

$$E' = x_{ij} \wedge ... \wedge (x_{kl} \rightarrow x_{mn}) \wedge ... \wedge ((x_{uv} \wedge x_{wx}) \rightarrow x_{yz}) \wedge ...$$

which encodes all the dependencies across each pair of reorder units.

After this process, we convert all clauses in $E'$ back into our ILP constraints. As we go through each clause in $E'$, we only select those clauses with literals about query related reorder units, i.e., $\{x_{ij} : U_i \text{ and } U_j \text{ contain queries}\}$, and convert them back into ILP constraints with the following rules:

- $x_{ij} = true \Rightarrow p_i < p_j$
- $(x_{ij} \rightarrow x_{kl}) \Rightarrow (p_i < p_j) \vee (p_k > p_l)$
- $((x_{ij} \wedge x_{kl}) \rightarrow x_{uv}) \Rightarrow (p_i > p_j) \vee (p_k > p_l) \vee (p_u < p_v)$

The ILP constraints will now only involve query related units, and solving these constraints will give us the optimal ordering.

We now prove that the iterative inference process described above converges in polynomial time $n$, where $n$ is the number of reorder units. To see why, notice that each application of inference rules introduces a new clause of the form $x_{ij}$, $x_{ij} \rightarrow x_{kl}$, or $x_{ij} \wedge x_{kl} \rightarrow x_{uv}$. If no new clause is generated, the process terminates. Since the number of Boolean literals $x_{ij}$ is bounded by $\mathcal{O}(n^2)$, the number of possible clauses is also polynomial in $n$. Thus, it will take a polynomial number of inference rule applications. Since searching all existing clauses for rule application is done in polynomial time as well, the induction process described above will converge in polynomial time with respect to the number of variables in the original ILP problem.

Given a solution to the optimized ILP, there always exists an ordering of all reorder units such that all constraints are satisfied. We prove this by contradiction. If no such ordering exist, substituting the value of $x_{ij}$ (where $U_i$ and $U_j$ include queries) back to the Boolean expression $E$ will always falsify $E$, or generate values of literals that cannot form a valid order, for instance, $x_{ik} = true, x_{kj} = true$ and $x_{ij} = false$ means that $U_i$ is before $U_k$, $U_k$ is before $U_j$ but $U_j$ is before $U_i$, which results in a contradiction. In the following we will show that non of the situations above will happen.

- **Assignment to literals falsifies $E$.** If the assignment to literals falsifies $E$, there must exist $i, j, k, l \in [1, n]$ such that $x_{ij}$ and $x_{ij} \rightarrow x_{kl}$ are clauses in $E$, $x_{ij} = true$ and $x_{kl} =$

$false$. Based on which units among $U_i, U_j, U_k, U_l$ are units including queries, we enumerate cases as listed below and prove that in any case the above situation always results in a contradiction:

1. $U_i, U_j, U_k, U_l$ are all units including queries. In this case, if $x_{ij} = true$ and $x_{ij} \rightarrow x_{kl}$, the constraint $x_{ij} \rightarrow x_{kl}$ will appear as a constraint in the ILP, and the ILP solver will not give a result in which $x_{kl} = false$.

2. $U_i, U_j$ include queries while $U_k, U_l$ do not. In this case, if $x_{ij} = true$, $x_{ij} \rightarrow x_{kl}$ and $x_{kl} = false$, there must exist query units $U_{k'}, U_{l'}$ such that the result of ILP makes $x_{k'l'} = false$, and

$$x_{kl} \rightarrow x_{k_1l_1}, x_{k_1l_1} \rightarrow x_{k_2l_2}, ..., x_{k_pl_p} \rightarrow x_{k'l'}$$

should appear as clauses in $E$. Only in this case $x_{kl}$ will evaluate to false. However, since $x_{ij} \rightarrow x_{kl}$, the induction rules will generate $x_{ij} \rightarrow x_{k'l'}$ to be a constraint in the ILP. If $x_{ij} = true$, then $x_{k'l'} = true$, and $x_{kl} = false$ will result in a contradiction.

3. $U_k, U_l$ include queries, while $U_i, U_j$ do not. Like the case above, there should exist query units $U_{i'}$ and $U_{j'}$ such that the result of ILP makes $x_{i'j'} = true$, and

$$x_{i'j'} \rightarrow x_{i_1j_1}, x_{i_1j_1} \rightarrow x_{i_2j_2}, ..., x_{i_pj_p} \rightarrow x_{ij}$$

should appear as clauses in $E$. Similarly, the induction process will generate $x_{i'j'} \rightarrow x_{kl}$ as an ILP constraint. If $x_{ij} = true$, then $x_{i'j'} = true$, and $x_{kl} = false$ will result in a contradiction.

4. None of $U_i, U_j, U_k, U_l$ includes queries. In this case, there should be some query units $U_{i'}, U_{j'}, U_{k'}, U_{l'}$ such that

$$x_{kl} \rightarrow x_{k_1l_1}, x_{k_1l_1} \rightarrow x_{k_2l_2}, ..., x_{k_pl_p} \rightarrow x_{k'l'}$$
$$\text{and } x_{i'j'} \rightarrow x_{i_1j_1}, x_{i_1j_1} \rightarrow x_{i_2j_2}, ..., x_{i_qj_q} \rightarrow x_{ij}$$

appear as clauses in $E$, and the result of ILP makes $x_{i'j'} = true$ and $x_{k'l'} = false$. Similar to the analysis in 2 and 3, by applying the induction rules, $x_{i'j'} \rightarrow x_{k'l'}$ will be a constraint in the ILP, which contradicts to $x_{ij} = true, x_{kl} = false$.

5. $U_i, U_j$ include queries and only one of $U_k, U_l$ includes query. Analysis is similar to case 2.

6. $U_k, U_l$ include queries and only one of $U_i, U_j$ includes query. Analysis is similar to case 3.

7. Only one of $U_i, U_j, U_k, U_l$ includes query and the other three not. Analysis is similar to case 4.

- **Assignment generates ordering requirements resulting in a contradiction.** Even if the assignment makes $E = true$, the ordering may also be invalid when it makes the conversion from $x_{ij}$ back to the order of $U_i, U_j$ impossible: there must exist $k$ where $i < k < j$ and the ILP generates a result in which $x_{ik} = true$, $x_{kj} = true$ and $x_{ij} = false$. Similar to the previous analysis, we prove by enumerating cases based on which reorder unit may contain query. If non of $U_i, U_j, U_k$ includes queries, there must exist some query units $U_{i'}, U_{j'}, U_{k'}$ ($i' < j' < k'$) and

$$x_{i'k'} \rightarrow x_{i_1k_1}, x_{i_1k_1} \rightarrow x_{i_2k_2} ..., x_{i_pk_p} \rightarrow x_{ik}$$
$$\text{and } x_{k'j'} \rightarrow x_{k_1j_1}, x_{k_1j_1} \rightarrow x_{k_2j_2}, ..., x_{k_qj_q} \rightarrow x_{kj}$$
$$\text{and } x_{ij} \rightarrow x_{i_1j_1}, x_{i_1j_1} \rightarrow x_{i_2j_2} ..., x_{i_rj_r} \rightarrow x_{i'j'}$$

appear as clauses in $E$, while in the ordering that ILP solver generates $x_{i'k'} = true$, $x_{k'j'} = true$ and $x_{i'j'} = false$. However, since $(x_{ik} \wedge x_{kj}) \rightarrow x_{ij}$ is also a clause in $E$, the induction process will produce ILP constraints $(x_{i'k'} \wedge x_{k'j'}) \rightarrow x_{i'j'}$, and this contradicts to the ILP result that set $x_{i'k'} = true$, $x_{k'j'} = true$ and $x_{i'j'} = false$. If $U_i$ includes a query, set $i' = i$ and perform similar analysis as above, likewise when $U_j$ or $U_k$ includes query.

Thus we have proved that there always exists an ordering of all reorder units such that all constraints are satisfied.

## 4.5 Restructuring Transaction Code

After QURO receives the ordering of queries from the ILP solver, it restructures the input code accordingly. If we rely on the ILP solver to find the order of all reorder units (as discussed in Section 4.2), then generating the final code would be easy. However, if we apply the optimization discussed in Section 4.4 to only solve for query related reorder units, then we need to determine the ordering of all non-query related reorder units as well. We discuss the restructuring process in this section.

The basic idea of restructuring code is to iterate through each query according to its new order as given by the solver, try to place other reorder units that have data dependencies on the query being processed, and roll back upon violating any order constraint. As listed in Algorithm 1, we start with an empty list $U\_list$, which is used to store the list of reordered units. We insert a unit $U$ from the set $U_s$ of all reordered units into the list when all other units producing values that $U$ uses are already in the list. To do so, we define two functions: `Defs`($U_i$) and `Uses`($U_i$). `Defs` returns the set of reorder units that defines variables used by unit $U_i$, and `Uses` returns the set of reorder units that uses values defined by $U_i$. The values to be returned are computed during preprocessing as discussed in Section 3.2. For each query $Q_i$, we first insert all units in `Defs`($Q_i$) into $U\_list$ (line 2 to line 8), followed by $Q_i$ itself (line 9), and `Uses`($Q_i$) (line 10 to line 16). For every reorder unit $U$ that is inserted into $U\_list$, we check if $U$ violates any data dependency constraints using the function `CheckValid`. Checking is done by scanning the clauses in $E'$ as discussed in Section 4.4 to see if the current ordering of units would make $E'$ evaluate to false. If so, the current order violates some data dependency constraint encoded in the ILP problem, and the algorithm attempts to resolve the WAR or WAW violation using variable renaming as described in Section 4.3. If the variable cannot be renamed, then the algorithm backtracks to reprocess all reorder units starting from the first reorder unit that falsifies $E'$. For each query $Q_i$, we keep a reject list ($Rej[Q_i]$) to record all reorderings that have been attempted but failed and led to a rollback. The process continues until a satisfying reordering is found, and Section 4.4 showed that there always exists a valid order.

## 5. PROFILING

As mentioned in Section 2, QURO profiles the transaction code by running an instrumented version of the application to estimate the amount of lock contention for each query. There has been prior work that studies how to estimate locking contention. Johnson et al. [15] use Sun's profiling tools to calculate time breakdown of

**Algorithm 1** Algorithm For Restructuring Transaction Code

```
1: for $Q_i \in Q\_list$ do
2:   for $U_j \in U_s$ and $U_j \notin U\_list$ and $U_j \notin Rej[Q_i]$ and
     Defs($U_j$)$\in U\_list$ and $Q_i \in$ Uses($U_j$) do
3:     if CheckValid($U_j$) then
4:       $U\_list$.insert ($U_j$);
5:     else
6:       break;
7:     end if
8:   end for
9:   $U\_list$.insert ($Q_i$);
10:  for $U_k \in U_s$ and $U_k \notin U\_list$ and $U_k \notin Rej[Q_i]$ and
     Defs($U_k$)$\in U\_list$ and $Q_i \in$ Defs($U_k$) do
11:    if CheckValid($U_k$) then
12:      $U\_list$.insert ($U_k$);
13:    else
14:      break;
15:    end if
16:  end for
17: end for
18:
19: function CheckValid($U_i$)
20: ... // check all clauses in $E'$ (details not shown)
21: if $E'$ evaluates to true then
22:   return 1;
23: else
24:   if Variable v in $U_i$ can be renamed then
25:     Rename v in $U_i$ and Uses($U_i$);
26:   else
27:     $temp$ = clear $U\_list$ to the failing point $U_f$;
28:     reinsert $temp$ into $U\_list$;
29:     for query units $Q_f \in temp$ do
30:       $Rej[Q_f]$.insert($U_f$);
31:     end for
32:   end if
33: end if
34: end function
```

database transactions, analyzing the time spent on useful work and lock waiting to identify contention level in the lock manager. Syncchar [20] runs a representative sample workload to calculate the conflict density and infer lock contention. QURO can use such techniques, but chose a simpler method which examines the running time of each query and computes its standard deviation. In our current prototype, most of the transaction time is spent on lock waiting. If the query accesses contentious data, then the lock waiting time will vary greatly from one execution to another. Hence the larger the deviation, the greater the possibility of data conflict.

To collect query running time, QURO adds instrumentation code before and after each query, and computes the standard deviation after profiling is completed. In the current prototype, we assume that the profiler runs with the same machine settings as the actual deployment of the application.

## 6. EVALUATION

We have implemented a prototype of QURO using Clang [2] to process transaction code written in C/C++, and gurobi [5] as the external ILP solver. In this section we report our experiment results under different settings where we compare the performance of the original implementation and the one generated by QURO.

We first study the performance of transaction code generated by QURO by isolating the effects of disk operations (e.g., fetching and writing committed data back to the disk), which can dominate the amount of time spent in processing each transaction. To do so, we disabled flushing data to the disk at commit time. The machine we use has memory large enough to hold the entire data set of any application used in the evaluation. All of the following experiments were performed on MySQL 5.5 server hosted on a machine with 128 2.8GHz processors and 1056GB memory.

### 6.1 Benchmarks

We used the following OLTP applications for our experiments:

1. The TPC-C benchmark. We used an open source implementation [3] of the benchmark as input source code, and performed experiments by running each type of transactions individually and different mixes of transactions.

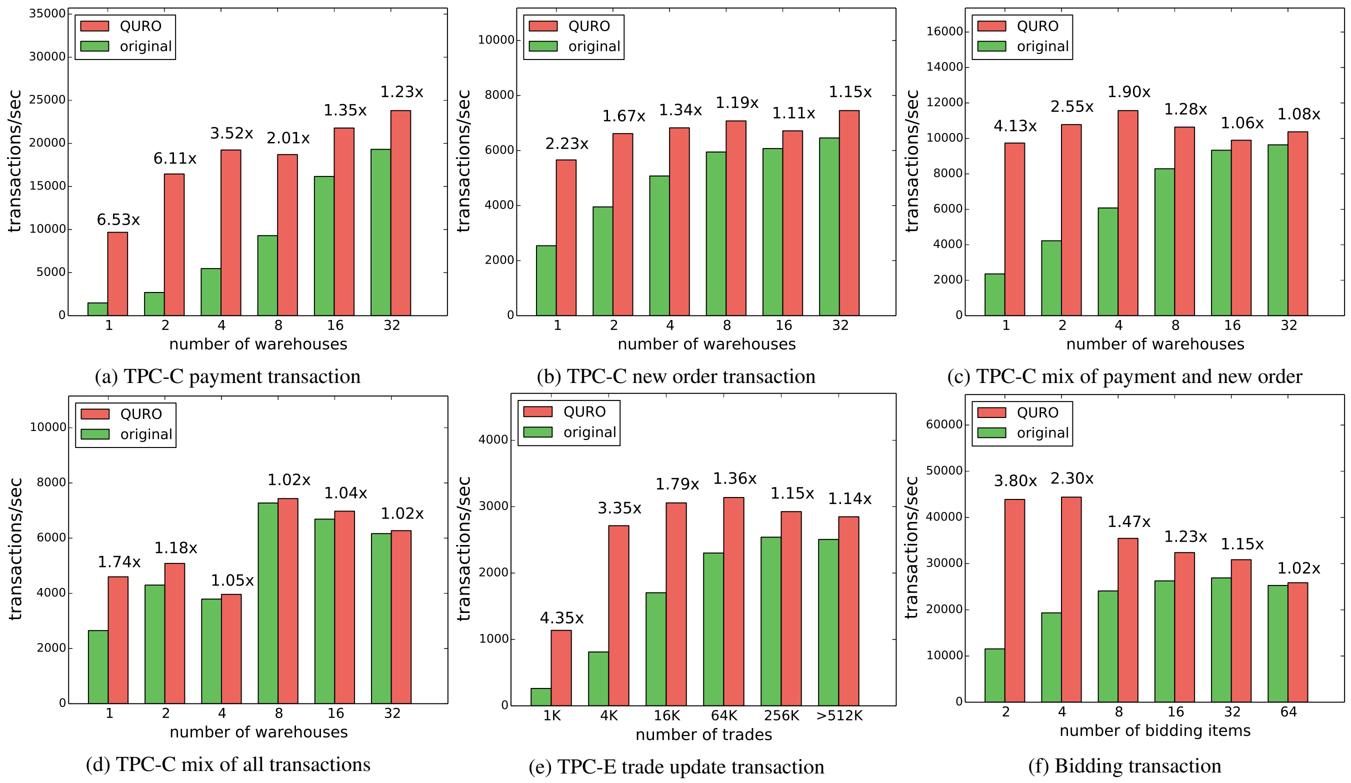
2. The trade related transactions from the TPC-E benchmark. The TPC-E benchmark models a financial brokerage house using three components: customers, brokerage house, and stock exchange. We used an open source implementation [4] as the input. The transactions we evaluated includes trade update, order, result and status transactions.

3. Transaction from the bidding benchmark. We use an open source implementation of this benchmark [1]. This benchmark simulates the activity of users placing bids on items. There is only one type of transaction in this benchmark: it reads the current bid, updates the user and bidding item information accordingly, and inserts a record into the bidding history table.

We ran the applications for 20 minutes in the profiling phase to collect the running time of queries. In each experiment, we omitted the first 5 minutes to allow threads to start and fill up the buffer pool, and the measured the throughput for 15 minute. We ran each application for 3 times and report the average throughput. We also omitted the thinking time on the customer side, and we assume that there are enough users issuing transactions to keep the system saturated.

### 6.2 Varying Data Contention

In the first experiment, we compared the performance of the original code and the reordered code generated by QURO by varying contention rates while fixing the number of concurrently running database threads to 32. Varying data contention is done by changing the data set size a transaction accesses. For the TPC-C benchmark, we change the data size by adjusting the number of warehouses, from 1 to 32. With 1 warehouse, every transaction either reads or writes the same warehouse tuple. With 32 warehouses, concurrent transactions are likely to access different warehouses as they have little data contention. For the TPC-E benchmark, we adjust the number of trades each transaction accesses. As the trade update transaction is designed to emulate the process of making minor corrections to a set of trades, changing the number of trades changes the amount of data the transaction accesses. We varied the number of trades from 1K to the size of entire trade table, 576K. When the number of trades being updated is small, multiple concurrent transactions will likely modify the same trade tuple. In contrast, when transactions randomly access any trade tuple in the trade table, they will likely modify different trades and have little data contention. We did not evaluate the other transactions since no contentious data is accessed in these transactions, so varying



(a) TPC-C payment transaction

(b) TPC-C new order transaction

(c) TPC-C mix of payment and new order

(d) TPC-C mix of all transactions

(e) TPC-E trade update transaction

(f) Bidding transaction

Figure 3: Performance comparison: varying data contention

data size won't significantly change the locking situation, and performance will not be much affected by locking. For the bidding benchmark, we adjusted the number of bidding items. The bidder giving a higher bidding price will change the current price on that bid item. We set the percentage of bidder giving higher bidding price to be 75%, which means that 75% of the transactions will write the item tuple.

Figure 3a shows the results of TPC-C running only the payment transaction (an excerpt is shown in Listing 1). Reordered implementation generated by QURO achieves up to 6.53× speedup as compared to the original implementation. Figure 3b shows the results of TPC-C benchmark running only new order transactions. In this transaction, the flexibility of reordering is restricted by the many data dependencies among program variables. Despite this limitation, QURO still achieves up to 2.23× speedup as a result of reordering. Figure 3c shows the results of the TPC-C benchmark comprising 50% new order and 50% payment transactions. Under high data contention, the speedup of reordering is 4.13×. Figure 3d shows the results of TPC-C with standard mix of five types of transactions according to the specification. Increasing the types of transactions makes more data contentious, as some tables are only read by one type of transaction, hence there are no data contentions on those tables when only that type of transactions are executed. But with transaction mixes, there might be other transaction types that would write to the same table, thus causing contentions. However, with a mix of five types of transactions, reordering still increases the overall throughput by up to 1.74×.

For TPC-E, Figure 3e shows the results of trade update transaction. This benchmark has a while loop that on average runs for 20 iterations. There are multiple read queries within each iteration, but only one update query. QURO discovered the optimal reordering by breaking the loop and putting all the write operations from different iterations towards the end of the transaction, resulting in a speedup of up to 4.35× as compared to the original implementation. Even when there is little data contention, reordering still outperforms the original implementation by 1.14×.

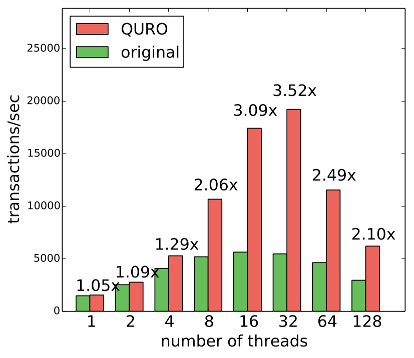
Finally, Figure 3f shows the results of the bidding benchmark. The bidding transaction is short and contains only five queries. In the reordered implementation, the bidding item is read as late as possible, followed by the item update being the last operation in the transaction. This resulted in a speedup of up to 3.80×.

The results show that as the data size decreases, the contention rate increases, which in return increases the chance of improving performance by reordering.

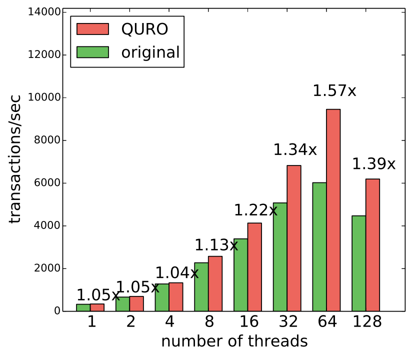
## 6.3 Varying Number of Database Threads

In the next set of experiments, we ran the benchmarks on the same data, but varied the number of database threads from 1 to 128. With the same database size, running more threads increases the amount of contention. We would like to study how the Quro-generated implementations behave as the number of threads increases.

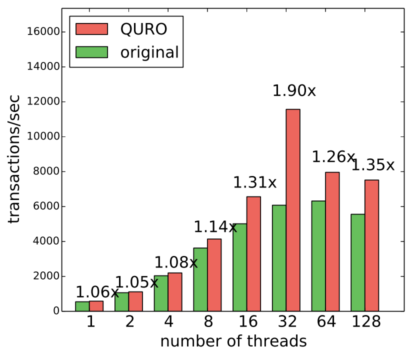
Figure 4a-d shows the results of running transactions from TPC-C. In this experiment we fixed the number of warehouse to be 4. For the payment transaction, QURO speeds up the application by up to 3.52×. For the new order transaction, the amount of speedup due to reordering increases as thread number increases, reaching the maximum of 1.57× when using 64 threads. Stock level and order status transactions are read-only transactions. In these transactions, no query needs to wait on lock when running standalone, and QURO-generated implementation has the same performance as the original implementation. We did not analyze the delivery transaction in TPC-C, since this transaction performs both read and write on a very small set of data. When it is running standalone, most



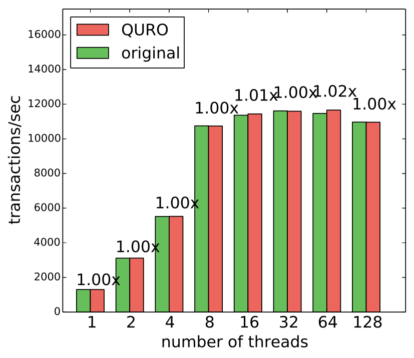
(a) TPC-C payment transaction



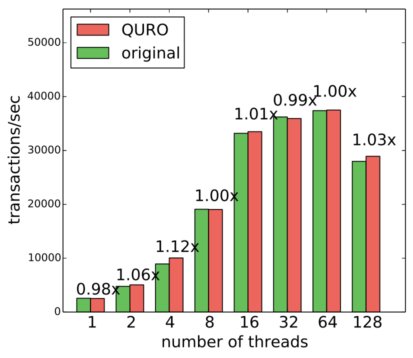
(b) TPC-C new order transaction



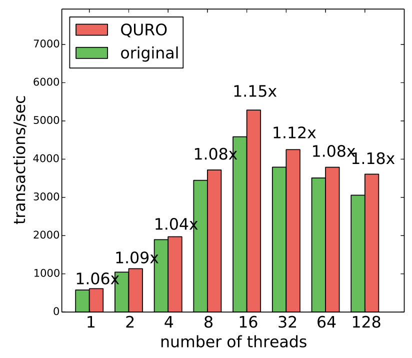
(c) TPC-C mix of payment and new order



(d) TPC-C stock level transaction



(e) TPC-C order status transaction



(f) TPC-C mix of all transactions

Figure 4: Performance comparison: varying number of threads for TPC-C benchmark

transactions will abort.

Figure 5d shows the results of the TPC-E trade update transaction, where we fixed the number of trades to be 4K. The throughput of original implementation falls greatly as the number of threads exceeds 16, while the reordered implementation only decreases slightly, and still speeds up the application by up to 3.35×.

Unlike TPC-C new order and payment transactions which have significantly contentious queries, queries in trade order and trade result transaction access data with small contention. For trade order transaction, 19 out of 21 queries are read queries, and the remaining 3 write queries are insert operations. These write queries are executed after all the read queries in the transaction in the original implementation. QURO only changes the order of the last three insert operations, which does not greatly affect the overall performance. Figure 5a shows that QURO-generated implementation has nearly the same performance as the original implementation. In the best case QURO improves the application throughput by 1.05×, and in worst case decreases throughput by only 2%. Since the input of trade result transaction is dependent on the trade order transaction, we only evaluated a mix of these two types of transactions. Figure 5b shows that reordering increases the throughput by up to 1.57×. For the trade status transaction which is a read-only transaction, QURO-generated implementation has the same performance as the original one as shown in Figure 5c.

Finally, Figure 5e shows the results of the bidding transaction. We fixed the number of bidding items to be 4. As shown in the figure, reordering increases application throughput by up to 2.30×.

We also compared how each benchmark scales as the number of threads increases by evaluating self-relative speedup. The baseline for both implementations is the throughput of single-thread execution with the original implementation, and Figure 6 shows how throughput changes as the number of threads increases. As shown in the figure, the reordered implementation has larger self-relative speedup than the original implementation as the number of threads exceeds 8. When running on 32 threads on the payment transaction, QURO-generated implementation has 12.4× self-relative speedup while 3.8× for the original. By reordering queries, QURO allows applications to scale up to a larger number of concurrent database connections as compared to the original implementation.

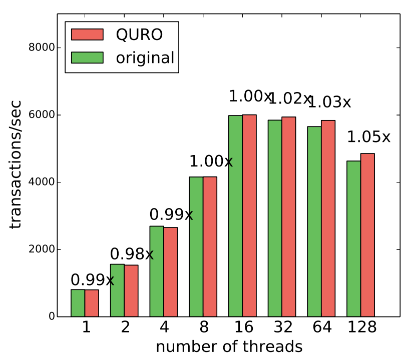
## 6.4 Analyzing Performance Gain

We did another set of experiments to gain insights on the sources of performance gain due to reordering.

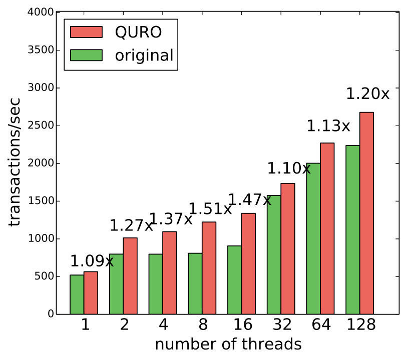
*Query execution time.* We profiled the amount of time spent in executing each query. We show the results for the TPC-C payment transaction in Table 2. The table lists the aggregate time breakdown of 10K transactions, with 32 database threads running transactions on a single warehouse. In this case reordering sped up the benchmark by 6.53×, as shown in Figure 3a.

The single-thread result in the third column indicates the running time of queries without locking overhead as each transaction is executed serially. By comparing the values to the single-thread query time on each row, we can infer amount of time spent in waiting for locks of that query in both implementations.

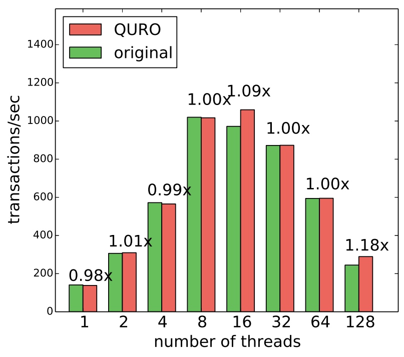
For the original implementation, the results show that most of the execution time was spent on query 2. As shown in Listing 1, this query performs an update on a tuple in the warehouse table, and



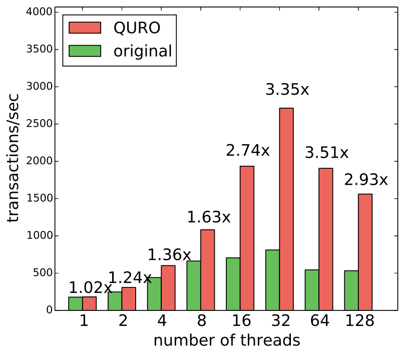
(a) TPC-E trade order transaction



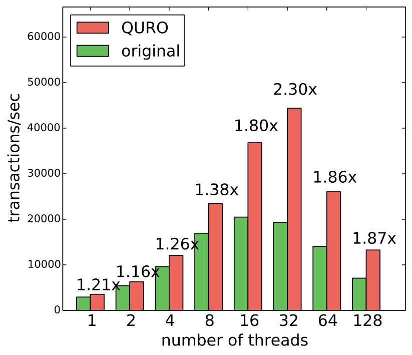
(b) TPC-E mix of trade order and trade result transaction



(c) TPC-E trade status transaction



(d) TPC-E trade update transaction



(e) Bidding transaction

Figure 5: Performance comparison: varying number of threads

| | original | reordered | single-thread | ratio |
|---|---|---|---|---|
| query 1 | 0.97 | 1.09 | 0.72 | 1.12 |
| query 2 | 210.27 | 6.06 | 0.73 | 0.03 |
| query 3 | 0.97 | 1.12 | 0.68 | 1.15 |
| query 4 | 0.80 | 17.18 | 0.62 | 21.48 |
| query 5 | 1.38 | 1.59 | 0.70 | 1.15 |
| query 6 | 1.86 | 1.31 | 0.98 | 0.70 |
| query 7 | 1.09 | 1.52 | 0.81 | 1.39 |
| query 8 | 1.16 | 1.60 | 0.17 | 1.38 |
| query 9 | 0.79 | 0.96 | 0.70 | 1.22 |
| total_latency | 219.29 | 32.43 | 6.11 | 0.15 |

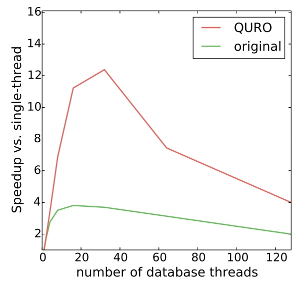
Table 2: Query running time of the payment transaction, where ratio=reordered/original. The reordered implementation reduces latency by 85%.

| | original | reordered | single-thread | ratio |
|---|---|---|---|---|
| query 1 | 0.73 | 0.76 | 1.91 | 1.04 |
| query 2 | 89.77 | 16.16 | 0.93 | 0.18 |
| query 3 | 0.92 | 1.01 | 1.02 | 1.09 |
| query 4 | 0.81 | 0.88 | 0.76 | 1.09 |
| query 5 | 0.63 | 0.69 | 0.71 | 1.08 |
| query 6 | 0.74 | 0.97 | 0.78 | 1.30 |
| query 7 | 0.66 | 0.76 | 0.62 | 1.16 |
| query 8 | 0.69 | 0.80 | 0.78 | 1.15 |
| query 9 | 0.79 | 1.06 | 0.69 | 1.34 |
| query 10 | 0.67 | 0.82 | 0.62 | 1.23 |
| total_latency | 114.99 | 46.66 | 27.01 | 0.41 |

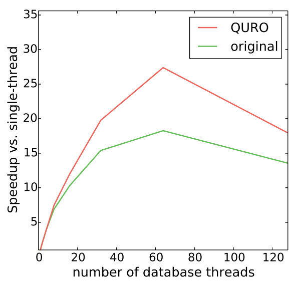
Table 3: Query running time of the new order transaction. The reordered implementation reduces latency by 59%.

every transaction updates the same tuple. By reordering, the running time of this query is significantly shortened: the query time in the original implementation is reduced by 97%. However, the execution time for all other queries increases after reordering. This is due to the increased chances of other queries getting blocked during lock acquisition. In the original implementation, the query accessing the most contentious data effectively serializes all transactions as each thread needs to acquire the most contentious lock in the beginning. As a result, the execution time of all subsequent queries is nearly the same as the time running on a single thread (i.e., without locking overhead). But reordering makes these queries run concurrently as they need to compete for locks, and this increases the running time.

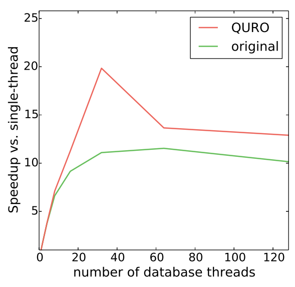
We also profiled the new order transaction. In this transaction, the opportunities for reordering is limited by the data dependencies among queries. In particular, query 2 reads a tuple from the district table and reserves it for update later in the transaction. This query is most contentious one. However, query 2 cannot be reordered to the bottom of transaction since there are many other queries that depend on the result of this query. This limits the amount of query time reduction, as shown in Table 3.



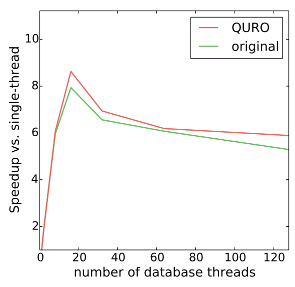
(a) TPC-C payment



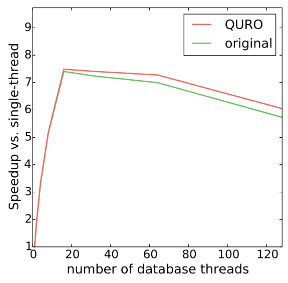
(b) TPC-C new order



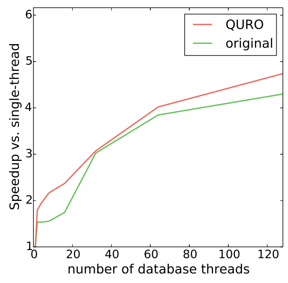
(c) TPC-C mix



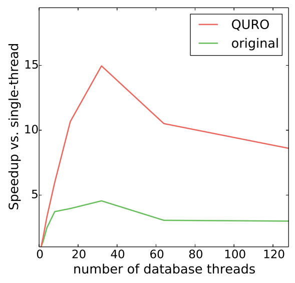
(d) TPC-C all txns



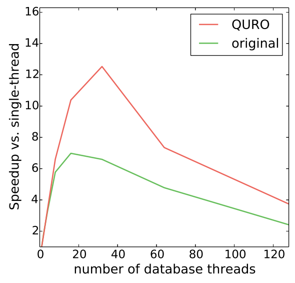
(e) TPC-E trade order



(f) TPC-E trade mix



(g) TPC-E tradeupdate



(h) Bidding

Figure 6: Self-relative speedup comparison

| #of trades | original | reordered | ratio | speedup |
|---|---|---|---|---|
| 1K | 53.89% | 39.12% | 0.73 | 4.35x |
| 4K | 20.08% | 1.78% | 0.09 | 3.35x |
| 16K | 2.39% | 0.35% | 0.15 | 1.79x |
| 64K | 0.55% | 0.08% | 0.15 | 1.36x |
| 256K | 0.14% | 0.02% | 0.14 | 1.15x |
| >512K | 0.00% | 0.00% | 1.00 | 1.14x |

Table 4: Abort rate comparison of the trade update transaction, where ratio=reordered/original. With 4K trades, reordering can reduce the abort rate by up to 91%.

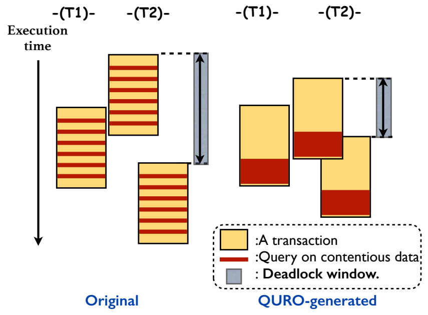


Figure 7: Execution of trade update transactions T1 and T2 before and after reordering. Deadlock window is the time range where if T2 starts within the range, it is likely to deadlock with T1.

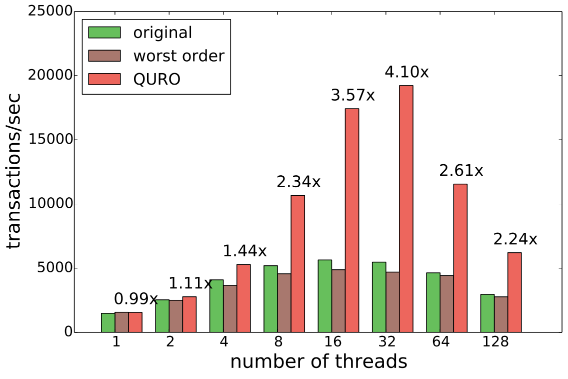
The above analysis indicates a trade-off with reordering. Reordering decreases the time spent on lock waiting on conflicting queries, but increases the time spent on less-conflicting queries. In most cases, the decrease in lock waiting time for contentious data usually outweighs the increase in lock waiting time for queries that access non-contentious data, and reordering reduces query latency despite some queries now takes slightly longer time to execute.

*Abort rate.* For the TPC-E trade update transaction, reordering improves performance by reducing abort rate. On average one trade update transaction randomly samples 20 trades and modifies the trade tuples selected. When the number of trades is small, concurrent transactions are likely to access the same set of trades, but in a different order, which causes deadlock. Reducing the locking time not only makes transaction faster, but also reduces the abort rate as shown in Table 4.

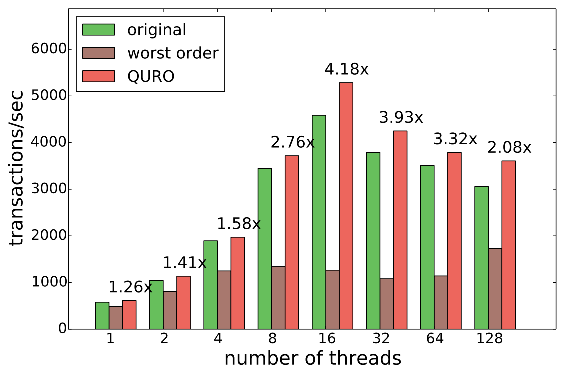
Figure 7 shows how reordering reduces abort rate. In the trade update transaction, only one query modifies the trade table in every loop, while there are other queries performing read on other tables that are not being modified. Assuming transaction T1 starts first, there is a time window within which if T2 starts, then T1 and T2 will likely deadlock each other. We refer this time range as the deadlock window, as shown in Figure 7. After reordering, the deadlock window is shortened, so there is less chance that the transaction would abort, and thus throughput increases.

## 6.5 Worst-Case Implementations

To further validate our observation that the order of queries affects transaction performance, we manually implement a "worst-case" implementation where the most contentious queries are issued *first* within each transaction. We then compared the performance of those implementations against the original and QURO generated implementations. The result for the TPC-C payment transaction is shown in Figure 8a with the throughput ratio of the best and worst case implementation labeled. As expected, the implementation with the most contentious queries executed as early as possible shows the worst performance as the number of threads increases. In contrast to Figure 4a, reordering obtained even larger speedups when compared to the worst-case implementation, with up to 4.10× improvement when 32 threads are used. The results



(a) TPC-C payment



(b) TPC-C standard mix

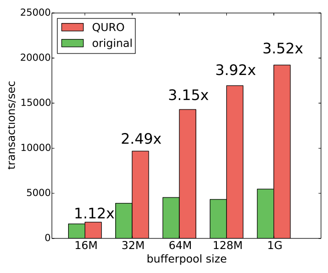
Figure 8: TPC-C worst-case ordering comparison

for the TPC-C standard benchmark are shown in Figure 8b, where QURO-generated implementation can have up to 4.18× improvement in performance comparing to a "worst-case" implementation.

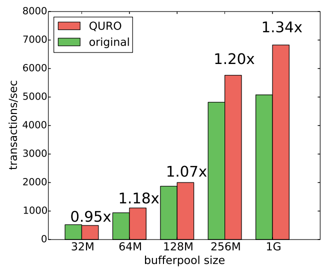
This experiment shows that the order of queries can have great impact on performance. For TPC-C payment transaction, an implementation with a bad ordering of queries can decrease performance by up to 14%. For standard mix of all TPC-C transactions, the decrease can be up to 62%.

While QURO relies on profiling to estimate the lock contention, if the workload changes, transactions need to be re-profiled and reordered accordingly. We leave dynamic profiling and regenerating transaction code as future work.

## 6.6 Disk-based DBMS



(a) TPC-C payment



(b) TPC-C neworder

Figure 9: Performance comparison: varying buffer pool size

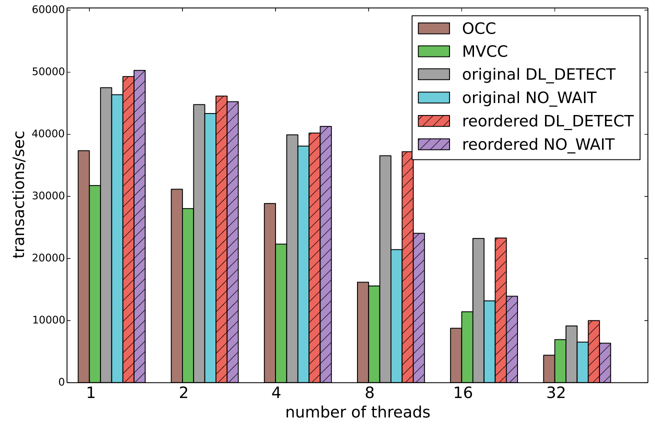
Next we consider the performance of QURO-generated code for disk-based DBMSs. We measured application performance while changing the amount of time spent on disk operations during transaction execution. This is done by varying buffer pool sizes. When the buffer pool is smaller than the database size, dirty data needs to be flushed to disk, making disk operations a significant portion of transaction execution time. Figure 9 shows the performance comparison of the TPC-C transactions. When the buffer pool is small, disk operations dominate the transaction processing time, thus the performance gained by reordering becomes trivial. Increasing buffer pool size decreases the time spent on disk, and as expected that increases throughput as a result of reordering. This experiment shows that even when the database is not completely in-memory, reordering can still improve application throughput.

## 6.7 Performance Comparison with Other Concurrency Control Schemes

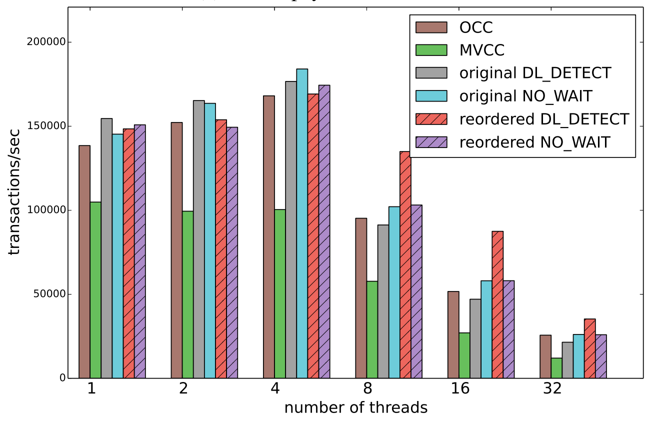
We also run experiments to test the performance of 2PL reordered implementation of TPC-C transactions against other concurrency control schemes like optimistic concurrency control (OCC) and multi-version concurrency control (MVCC). In this experiment, we use the main memory DBMS provided by Yu et al. [24], which supports different schemes including several variants of 2PL, OCC and MVCC. Since this DBMS doesn't use standard query interface, we manually ordered the transaction according to the order generated by QURO (the same order as in previous experiments). We evaluated the throughput of the TPC-C payment, new order, and a mix of these two transactions with original and reordered implementation under two versions of 2PL: 2PL with deadlock detection (DL_DETECT) and 2PL with non-waiting deadlock prevention (NO_WAIT), as well as the original implementation under OCC and MVCC. We set the number of warehouses to be 4 and varied the number of threads from 1 to 64, to test how different schemes scale.

Figure 10a-c show the results of this experiment. Shaded bars show the throughput of reordered implementation while other bars show the original implementation under different concurrency control schemes. As the number of threads increases, data contention increases, and the performance gain of reordered transaction under 2PL comparing to the original transaction under OCC and MVCC becomes larger. As shown in Figure 10c, for a mix of payment and new order transaction with 16 threads, reordered under 2PL with deadlock detection(DL_DETECT) outperforms OCC by 3.04×, and MVCC by 2.74×.

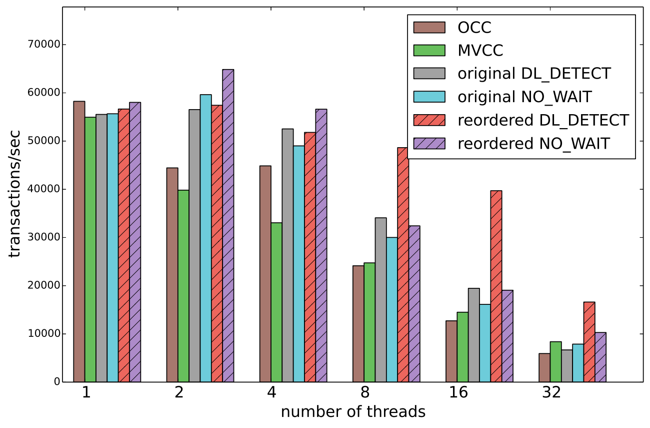
Figure 11 explains why reordered transaction under 2PL outperforms original transaction under OCC and MVCC. Assume two transactions T1 and T2 both read and write on a contentious tuple, and T1 starts first. Under each concurrency control scheme, there are time ranges within which if T2 starts, then it will end up being aborted. We refer this time range as abort window. Since OCC only writes to database at validation phase, any read on the contentious tuple before the validation phase will cause T2 to abort. Figure 11 shows that the original implementation under 2PL and OCC has much longer abort window than MVCC and reordered implementation under 2PL. Since MVCC requires complicated version management and the concurrency control overhead is larger than 2PL, reordered 2PL has higher throughput than MVCC.



(a) TPC-C payment transaction



(b) TPC-C new order transaction



(c) TPC-C mix of payment and new order

Figure 10: Performance comparison of TPCC transactions among original implementation under OCC, MVCC, 2PL-DL_DETECT, 2PL-NO_WAIT, and reordered implementation under 2PL-DL_DETECT, 2PL-NO_WAIT

## 6.8 Performance Comparison with Stored Procedures

In the previous experiments, we ran our database server and client program on the same machine to minimize the effect of network round trips as a result of issuing queries. In this experiment we added another open source implementation that uses stored procedures [3]. Using stored procedures, the client issues a single query to invoke the stored procedure that implements the entire transaction. As a result, the amount of network communication between the client and the database is minimized.

Figure 12a-d show the results of TPC-C transaction in this experiment. When there is little data contention, the stored procedure implementation outperforms the others due to shorter round trip time. However, as the amount of data contention increases, the time spent on locking far exceeds the time spent on the network communication. For the payment transaction, the reordered implementation has higher performance improvement (3.52×) over the original, as compared to the improvement of the stored procedure implementation (1.41×).

## 6.9 ILP Optimization

In the final experiment, we quantified the optimization presented in Section 4.4 in reducing the amount of ILP solving time. For the experiment, we chose two transactions: new order transaction with 40 statements and 9 queries, and trade order transaction with 189 statements and 21 queries. After preprocessing, the two transactions were split into 27 and 121 reordering units respectively. We used QURO to formulate each transaction into an ILP problem, using both the simple formulation discussed in Section 4.2 and the optimized formulation as discussed in Section 4.4. We then used two popular open source ILP solvers, lpsolve [6] and gurobi [5], to solve the generated programs with a timeout of 2 hours. The algorithm for restructuring the transaction code is described in Section 4.5, which is only needed when optimization is used, runs for 1 second in both cases.

| | Transaction 1 | | Transaction 2 | |
|---|---|---|---|---|
| # statements | 40 | | 189 | |
| # queries | 7 | | 25 | |
| # variables | 27 | | 121 | |
| # constraints | 266 | | 3595 | |
| # constraints post-opt | 6 | | 64 | |
| | lpsolve | gurobi | lpsolve | gurobi |
| Original solving time | >2hrs | 1s | >2hrs | >2hrs |
| Optimized solving time | 1s | 1s | >2hrs | 1s |

Table 5: ILP experiment results. The number of variables equals to the number of reordering units. Together with the number of constraints, these two numbers indicate the ILP problem size.

The results are shown in Figure 5. Under the original problem formulation where each reorder unit is represented by a variable in the ILP, both solvers did not finish before timeout for transaction 2. In contrast, the optimized formulation reduces the number of variables by 79% and the number of constraints by 98%. The problem also solves much faster, and hence allowing QURO to process larger transaction code inputs.

## 7. RELATED WORK

Besides lock-based methods, various concurrency control mechanisms have been proposed, such as optimistic concurrency control(OCC) [18] and multi-version concurrency control(MVCC) [7]. Yu et al. [24] studied the performance and scalability of different concurrency control scheme for main-memory DBMS.

There has been work done on improving the efficiency of locking-based concurrency control schemes. Shore-MT [16] applies 2PL and provides many system-level optimization to achieve high scalability on multi-core machines. Jung et al. [17] implemented a lock manager in MySQL and improves scalability by enabling lock allocation and deallocation in bulk. Horikawa [14] adapted a latch-free

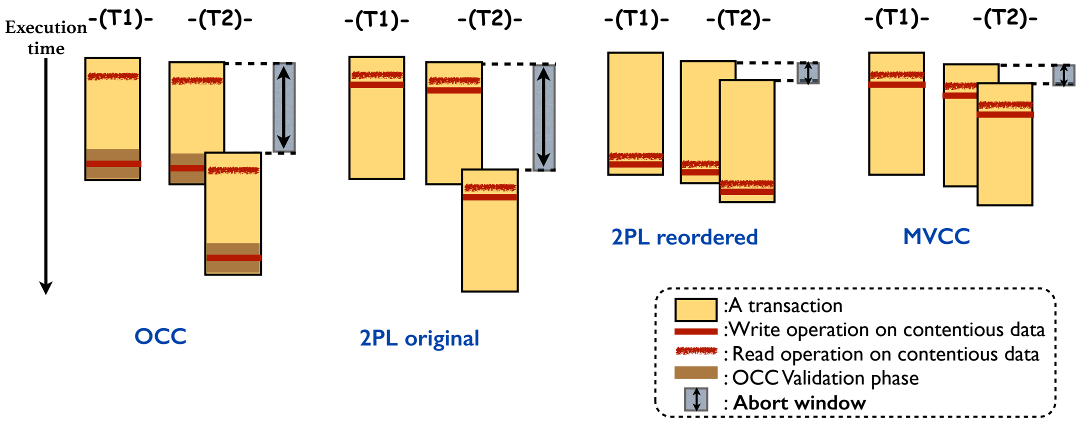
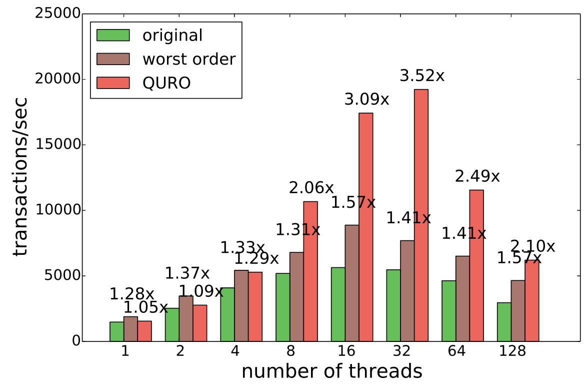
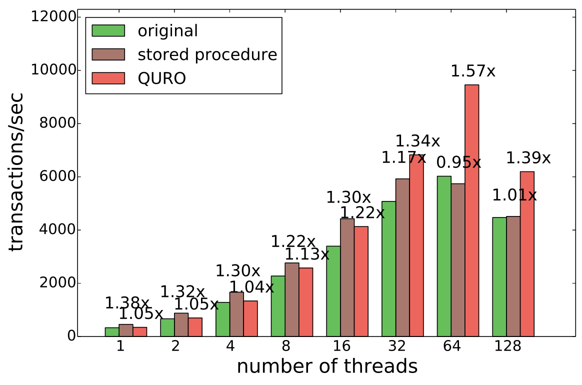


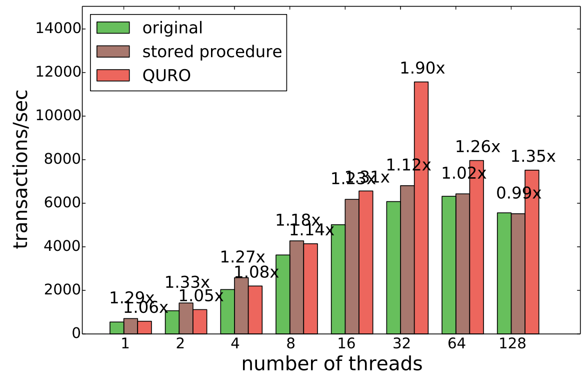
Figure 11: Execution of transaction T1 and T2 under different concurrency control schemes. Abort window is the time range where if T2 starts within the range, it is likely to abort because of data conflict with T1.



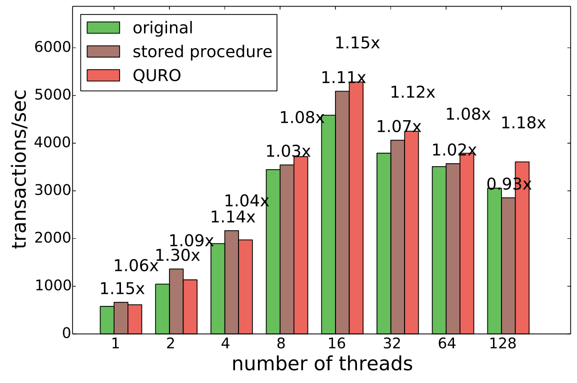
(a) TPC-C payment transaction



(b) TPC-C new order transaction



(c) TPC-C mix of payment and new order



(d) TPC-C mix of all transactions

Figure 12: Stored procedure performance, varying number of threads

data structure in PostgreSQL and implemented a latch-free lock manager. However, none of these systems examine how queries are issued by the application. QURO improves the performance by changing the database application, and our work can also leverage other locking implementations as well such as the ones described.

There has also been work done in looking into improving database performance from the database application's perspective. DBridge [9, 13] is a program analysis and transformation tool that optimize database application performance by query rewriting. Sloth [11] and Pyxis [10] are tools that use program analysis to reduce the network communication between the application and DBMS.

Finally, there is also work done in using database application semantics to design concurrency control protocol. Faleiro et al. [12] show that lazy transaction processing improves cache locality and achieves better load balancing. They used contention footprint to help decide which query to delay execution and reduce data contention. However, this technique only applies to deterministic DBMSs and requires knowing all the queries to be executed before transaction starts. Our work combines the knowledge of concurrency control with program analysis of database applications and is applicable to a wider range of DBMSs.

## 8. CONCLUSION

In this paper, we presented QURO, a new tool for compiling transaction code. QURO improves performance of OLTP applications by leveraging information about query contention to automatically reorder transaction code. QURO formulates the reordering problem as an ILP problem, and uses different optimization techniques to effectively reduce the solving time required. Our experiments show that QURO can find orderings that can reduce latency up to 85% along with an up to 6.53× improvement in throughput as compared to open source implementations.

## 9. ACKNOWLEDGMENT

The authors are grateful for the support of NSF Grant IIS-1546083.

## 10. REFERENCES

[1] The bidding benchmark from silo. `https://github.com/stephentu/silo.git`.

[2] Clang. `http://clang.llvm.org`.

[3] dbt2 benchmark tool. `http://osdldbt.sourceforge.net/#dbt2`.

[4] dbt5 benchmark tool. `http://osdldbt.sourceforge.net/#dbt5`.

[5] Gurobi optimization. `http://www.gurobi.com`.

[6] lpsolve. `http://sourceforge.net/projects/lpsolve`.

[7] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. In *ACM Comput. Surv.*, volume 13, pages 185–221, June 1981.

[8] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. In *IEEE Trans. Softw. Eng.*, volume 5, pages 203–216, May 1979.

[9] M. Chavan, R. Guravannavar, K. Ramachandra, and S. Sudarshan. DBridge: A program rewrite tool for set-oriented query execution. In *Data Engineering (ICDE), 2011 IEEE 27th International Conference on*, pages 1284–1287, 2011.

[10] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. In *Proceedings of the VLDB Endowment*, volume 5, pages 1471–1482, 2012.

[11] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 931–942, 2014.

[12] J. M. Faleiro, A. Thomson, and D. J. Abadi. Lazy evaluation of transactions in database systems. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, pages 15–26, 2014.

[13] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. In *Proceedings of the VLDB Endowment*, volume 1, pages 1107–1123, Aug. 2008.

[14] T. Horikawa. Latch-free data structures for DBMS: Design, implementation, and evaluation. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 409–420, 2013.

[15] R. Johnson, I. Pandis, and A. Ailamaki. Improving OLTP scalability using speculative lock inheritance. In *Proceedings of the VLDB Endowment*, volume 2, pages 479–489, Aug. 2009.

[16] R. Johnson, I. Pandis, N. Hardavellas, A. Ailamaki, and B. Falsafi. Shore-MT: A scalable storage manager for the multicore era. In *Proceedings of the 12th International Conference on Extending Database Technology: Advances in Database Technology*, 2009.

[17] H. Jung, H. Han, A. D. Fekete, G. Heiser, and H. Y. Yeom. A scalable lock manager for multicores. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 73–84, 2013.

[18] H. T. Kung and J. T. Robinson. On optimistic methods for concurrency control. In *ACM Trans. Database Syst.*, 1981.

[19] T. J. Marlowe and B. G. Ryder. Properties of data flow frameworks: A unified model. In *Acta Inf.*, pages 121–163, 1990.

[20] D. Porter and E. Witchel. Understanding transactional memory performance. In *Proceedings of the 2010 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 97–108, March 2010.

[21] Transaction Processing Performance Council. TPC-C Benchmark Revision 5.11. Technical report, 2010.

[22] M. J. Wolfe. *High Performance Compilers for Parallel Computing*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[23] L. T. Yang and M. Guo. *High-performance computing: paradigm and infrastructure*, volume 44. John Wiley & Sons, 2005.

[24] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker. Staring into the abyss: An evaluation of concurrency control with one thousand cores. In *Proceedings of the VLDB Endowment*, volume 8, pages 209–220, Nov. 2014.