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 locking mechanisms, not much has been done on understanding how transaction applications issue queries and leveraging application semantics to improve performance. This paper presents QURO, a query-aware compiler that automatically reorders queries in transaction code to improve performance. Utilizing 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 results show that implementations generated by QURO can increase transaction throughput by up to 6.53×, while reduce transaction latency by 85%.

1. INTRODUCTION

From airline ticket reservation systems, online shopping, to banking applications, we interact with online transaction processing (OLTP) applications on a daily basis. These applications are often organized 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) [8, 9] 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. 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, 2PL requires that a transaction cannot request additional locks once it releases any lock. Thus, every transaction has an expanding phase where locks are acquired and no locks are 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 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. Although using 2PL ensures data consistency across multiple concurrent transactions, its two-phase requirement essentially forces transactions to acquire locks on data elements as it accesses them, and release all the locks when it commits. 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 elements, then essentially all but one of the transactions can proceed while the rest are blocked, even though other transactions could have made further progress if they first acquired locks on other data elements. As a result, each transaction takes a longer time to be processed, and the overall system throughput suffers. This is exacerbated with main-memory DBMS, as locking becomes the predominant bottleneck in transaction processing since transactions no longer need to access the disk [15].

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 automatically 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 levels 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 might break the original flow and make the code difficult to read. In addition, developers need to preserve the data dependencies among different queries (for example, one query's parameters might come from the results returned from a previous query as parameters) 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 query contention. QURO profiles the application to estimate the amount of contention among queries. Given the profile, QURO formulates the reordering problem using Integer Linear Programming (ILP), and uses the solution to restructure the transaction code while preserving the original semantics of the application.

This paper makes the following contributions:

- We observe that reordering queries in transaction code based on contention can drastically affect performance of OLTP applications, and that current general-purpose compiler frameworks and DBMSs do not take advantage of that aspect to optimize 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 reordered transactions generated by QURO can improve throughput up to 6.53×, while reducing the average latency of individual transactions 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

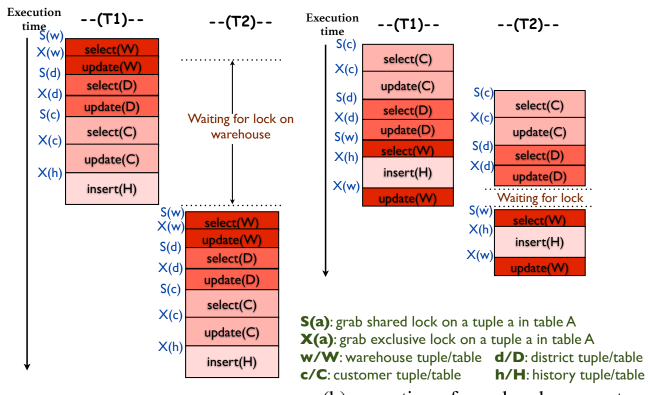
In this section we discuss query reordering in transaction code using an example and describe the architecture of QURO. To motivate, Listing 1 shows an excerpt from an open-source implementation [3] of the payment transaction from the TPC-C benchmark [22], which records a payment received from a customer. In the excerpt, the code first finds the warehouse to be updated with payment on line 1 and subsequently updates it on line 2. The same happens to the district table on line 3 and line 4. After that the code updates the customer record. 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 payment 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. Here `select("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
Execution time
--(T1)--
--(T2)--
S(c) X(c) S(d) X(d) S(w) X(h) X(w)
select(C) update(C) select(D) update(D) select(W) insert(H) update(W)
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 the execution of the 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 the typical TPC-C setup, the warehouse table contains the fewest tuples. Hence when 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 thread contention and the amount of time spent in processing 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, both transactions attempt 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 the two 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 district and warehouse 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 point (line 15), transaction T1 delays acquiring the exclusive lock on the warehouse tuple, allowing T2 to proceed with other operations on other (less contentious) tuples concurrently with T1. Comparing the two transaction implementations, reordering increases transaction concurrency, and reduces the total amount of time incurred in processing 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 in Listing 1 can only be executed after the queries on line 1 and line 3

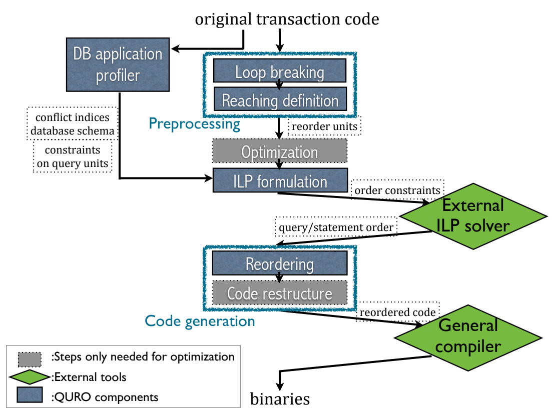


Figure 2: Architecture and workflow of QURO

because it uses `w_name` and `d_name`, which are results of those two queries. Besides the 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 does 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 amount of 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++, and we assume that the transactions use standard APIs to issue queries to the DBMS (e.g., ODBC and MySQL connector API).

The architecture of QURO is shown in Figure 2. After parsing the input code, QURO first generates an instrumented version in order to profile the running time of each query and gather information about query contention. QURO executes the instrumented version using the same deployment as the real 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 needed 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 done in order 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. Both the data dependencies among program variables and database tuples may induce order constraints. QURO first uses the reaching definition analysis from preprocessing to construct order constraints based on program variables. QURO then analyzes the queries with the database schemas to find out order constraints on 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 based on the ordering constraints, and solving the program returns the optimal way to implement the transaction subject to the data dependencies and the amount of contention estimated during profiling. 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 implemented using many lines of code. In Section 4.4 we propose an optimization to make this tractable and efficient. At the end of the process, QURO generates the reordered version of the input code in C++ and compiles it using a general-purpose compiler to produce the final 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 further nested within others. Disregarding the bodies inside loop and conditional statements and treating the entire statement as one unit will limit 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 [23], to split an individual loop nest into multiple ones. The basic idea is that a loop with two statements S_1 and S_2 in its body can be split 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 has 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 loop condition. If S_1 writes some value that affects the loop condition, then all statements within the loop cannot be split into separate loops.

We apply the fission algorithm discussed in prior work [24] and handle nested loops by checking the fission condition iteratively.

Listing 3 and 4 shows and 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, line 4 and 5 can be split. Notice in the example that there are no dependencies between lines 2 and 3 (when considered in tandem), line 4, and line 5. Since none of these statements affect the loop condition, they can be safely split into individual loops.

[1]We currently assume that each transaction is implemented within a single function and leave inter-procedural analysis as future work.
[2]We assume the database schema doesn't change when transactions are running.

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

Listing 4: Loop code after fission

Similarly, the conditional statements under a single condition can also be split. In general, conditional statements S_1 and S_2 under a 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, consider lines 9 to 14 in code example shown in Listing 1. 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 Reaching Definition Analysis

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 [20]. 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.

As an example, for the code fragment shown in Listing 1, line 1 defines the variable `w_name` that is used by line 15 as one of the function call parameters. We refer to line 1 and line 15 as a def-use pair (on variable `w_name`). In general, each statement can appear in multiple def-use pairs as it can define and use multiple variables. Computing reaching definition identifies all such def-use pairs in the input code.

4. REORDERING STATEMENTS

The preprocessing step normalizes the input code into individual statements that can be rearranged. We call each such statement, for instance an assignment, or a loop/conditional block that cannot be further decomposed using methods as described in Section 3.1, a *reorder unit*. In general, reorder unit includes top-level loops, conditionals, assigns, and function call statements.

In this section we discuss how we formulate the statement reordering problem by making use of the information collected during preprocessing.

4.1 Order Constraints Generation

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 the foreign key to a table involved in another query. In both of these 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, 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 of dependency. These data dependencies set constraints on the order of reorder units. In the following we discuss how to derive these constraints.

Dependencies among program variables:

• Read-after-write (RAW): Reorder unit U_i uses a variable that is defined by another unit U_j.
Formal constraint: U_i should appear before U_j in the restructured code.

• 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_i, U_j form a def-use pair, 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.

• Write-after-write (WAW): If 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 line 1 and line 3. Meanwhile,

[3]We currently do 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.

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 order line 12 should not appear between line 6 and line 8.

Dependencies among database tuples:

- 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 (update, insert, or delete).
- 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 query performs a write.
- Foreign-key constraints: Q_i performs an insert/delete on table T_i, or updates column C_i in T_i. Q_j operates on T_j, which includes a column referencing column C_i in T_i.
- Triggers: Q_i performs an insert or a delete on table T_i, which triggers a set of pre-defined operations, including operations on table 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 listed in Listing 1, queries on line 1 and line 2 operate on the same table, and they cannot be reordered or else the query on line 1 will read the wrong value.

In order to find out the dependencies among database tuples, QURO analyze each database query string to find out which tables and columns are involved in each query. Then QURO utilizes the database schema obtained from preprocessing and discover the dependencies listed above.

4.2 Problem Formulation

We formulate the reordering problem as an instance of ILP in QURO. 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 are 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. Then the order constraints derived from data dependencies are expressed as the following constraints in ILP:

- $p_i \le n, i \in [1, n]$, to ensure that each unit is assigned a valid position.
- $p_i \ne p_j, i \ne j$, to ensure that each unit has a unique position.
- $p_i < p_j$, if there is a RAW 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 redefinitions.
- $p_l > p_i, i \ne l$, if there is a WAW dependency between U_l and U_i, and U_l is the last define of a global variable WAW involved.
- $p_i < p_j, i < j$, 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.

Given that, the goal of the ILP 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 order for each reorder unit.

As an example, the code shown in Listing 1 generates the following constraints. Here we assume that there are no view/trigger/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, both operate on warehouse table
$p_3 < p_4$, Query order constraint, both operate on district table

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

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.

4.3 Removing Unnecessary Dependencies

Solving ILP is a NP-hard problem in general. In the following sections we describe two optimizations that reduce the number of constraints and variables in the ILP, and we evaluate these techniques in Section 6.8.

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 our 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, 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, along with enabling more queries to be reordered. As shown in the example above, a typical way to remove WAR and WAW on a variable `v` is to introduce an additional variable `v_r`, assign the

```
v=select("table1");
if(cond)
  v=select("table2");
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

old value of v to v_r, and then change subsequent define of v to v_r instead [21]. 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 primitive type (e.g., 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 significantly to affect performance. 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 solve this, 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 Lines 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 generates many reordering units and variables in the ILP. This can cause the ILP solver to run for unacceptably long time. In this section we describe an optimization to reduce the number of variables required in formulating the ILP.

Since our goal is to reorder database query related reorder units, we could formulate the ILP 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. For example, suppose query U_3 uses as parameter the value that is computed by a non-query statement U_2, and U_2 uses a value that is returned by query 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 reordering. The reordering will be correct, however, if we append extra constraints to the ILP (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 $i < j, i, j \in [1, n]$, to indicate $p_i < p_j$. 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 dependencies across each pair of reorder units.

After this process, we convert all clauses in E' back into our ILP constraint. 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 order to arrange them.

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, then 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.

Given a solution to the optimized ILP, we have also proved that there always exists an ordering of all reorder units (including those non-query reorder units) such that all constraints are satisfied. The proof is included in technical report [6] online due to space limitation.

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 output would be easy. However, if we instead 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 is to iterate through each query according to its new order as given by the solver, and try to place other reorder units that have data dependencies on the query being processed, and 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 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 `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 (line 19). If so, the current order violates some data dependency constraint encoded in the ILP, 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 backtrack to reprocess all reorder units starting from the first reorder unit that falsifies E' (line 26). 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.

Algorithm 1 Algorithm For Restructuring Transaction Code

```
1: for $Q_i \in Q\_list$ do
2:    for $U_j \in Us$ 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 $U_k \in Us$ 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: fuction CheckValid($U_i$)
19: // check all clauses in $E'$ (details not shown)
20: if $E'$ evaluates to true then
21:    return 1;
22: else
23:    if Variable v in $U_i$ can be renamed then
24:       Rename v in $U_i$ and Uses($U_i$)
25:    else
26:       $temp$ = clear $U\_list$ to the failing point $U_f$;
27:       reinsert $temp$ into $Q\_list$ or $Us$;
28:       for query units $Q_f \in temp$ do
29:          $Rej[Q_f]$.insert($U_f$);
30:       end for
31:    end if
32: end if
33: end function
```

5. PROFILING

As mentioned in Section 2, QURO profiles the transaction code by running an instrumented version of application to estimate the amount of contention for each query. In the current prototype, we assume that the profiler will run on the same machine as the application with identical number of database threads.

Unfortunately, estimating the amount of contention among queries is not straightforward. Different DBMSs provide different ways to profile queries, and some do not show the amount of time spent on lock acquisition for a query. However, we can infer contentious rates by examining the running time of each query. The standard deviation of the running time on a query indicates how likely it is going to experience data conflict. If the query accesses contentious data, the dominating portion of execution time is spent on lock waiting, and the waiting time can vary greatly from one execution to another. Hence the larger the deviation, the greater the possibility of data conflict. As such, given the query runtime information, we currently compute the conflict index of a query i to be the standard deviation of query i's execution time during profiling.

To collect query running time, QURO adds instrumentation code before and after each query, and computes standard deviation after profiling is completed. The tool assumes users will run the instrumented version under the same machine setting as real deployment with similar workloads.

6. EVALUATION

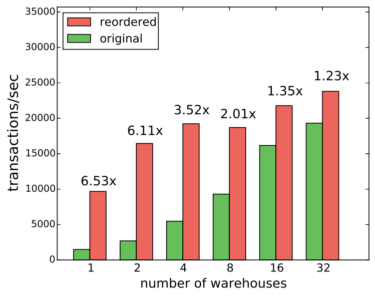
We have implemented a prototype of QURO using Clang [2] to process transaction code written in C++, and gurobi [5] as the external ILP solver. In this section we report our experiment results where we compare the performance of the original implementation and the one generated by QURO.

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. All the following experiments were performed with MySQL 5.5 server hosted on a machine with 128 2.8GHz processors and 1056G memory.

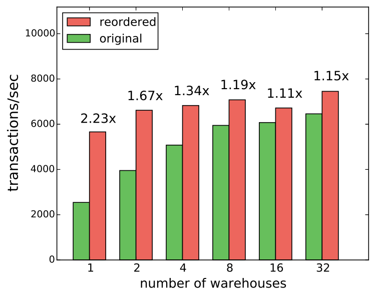
6.1 Benchmarks

We used the following OLTP applications for our experiments:

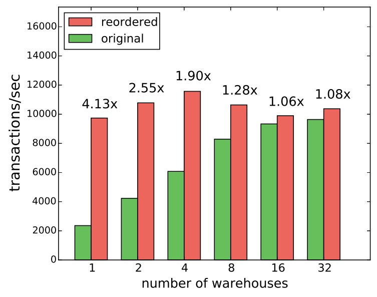
- The TPC-C benchmark. We used an open source implementation [3] of the benchmark as input source code, and performed experiments by running each type of transactions individually and different mixes of transactions. Due to limited space, we only present the results of new order and payment transactions running alone, a mix of payment and new order transaction, and a standard mix of all types according to the specification [22]. The results for other transactions can be found in the technical report [6].
- The trade related transactions from 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. We only present the results of trade update transaction because other transactions don't include queries on contentious



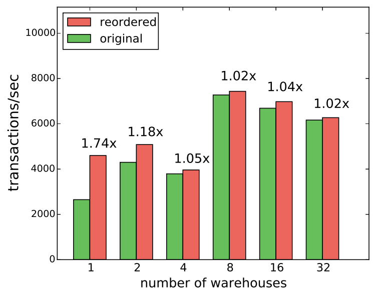
(a) TPC-C payment transaction



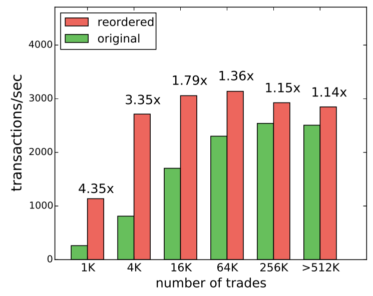
(b) TPC-C new order transaction



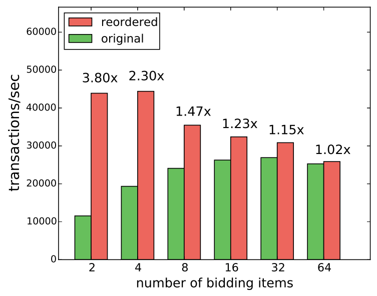
(c) TPC-C mix of payment and new order



(d) TPC-C mix of all transactions



(e) TPC-E trade update transaction



(f) Bidding transaction

Figure 3: Performance comparison: varying data contention

data, where reordering gives limited benefits. The complete results of all types of transaction can be found in the technical report.

• 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. The transaction reads the current bid, updates user and bidding item information accordingly, and inserts a record into the bidding history table.

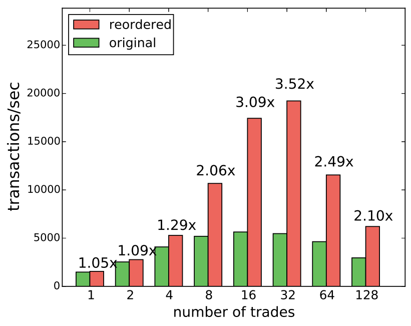
We ran the applications for 20 minutes in the profiling phase to collect the running time of queries. In each experiment, we omitted the first 5 minutes to allow threads to start and fill in the buffer pool, and the measured the throughput for 15-minute. We ran each application for 3 times and report 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

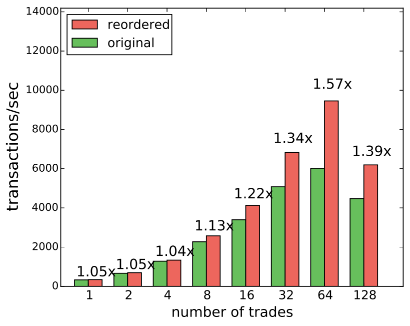
In the first experiment, we compared the performance of the original code and the reordered code generated by QURO by varying contention rates, fixing the number of concurrently running database threads to 32. Varying data contention is done by changing the data set size a transaction accesses. For the TPC-C benchmark, we adjusted the number of warehouses. The size of TPC-C databases are typically measured by the number of warehouses. Many of the other tables scale together with the warehouse table, which is the table with fewest tuples. In this experiment, we varied the number of warehouses from 1 to 32. With 1 warehouse, every transaction either reads or writes a single warehouse tuple. With 32 warehouses, concurrent transactions are likely to access different warehouses and have little data contention. For the TPC-E benchmark, we adjusted the number of trades each transaction can access. As the trade update transaction is designed to emulate the process of making minor corrections to a set of trades, the number of trades being updated determines the size of the working set. 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 to modify the same trade tuple. In contrast, when transactions randomly access any trade tuple in the trade table, they will mostly modify different trades and have little data contention. 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 tuples contentious. However, reordering still increases the throughput by up to 1.74×.

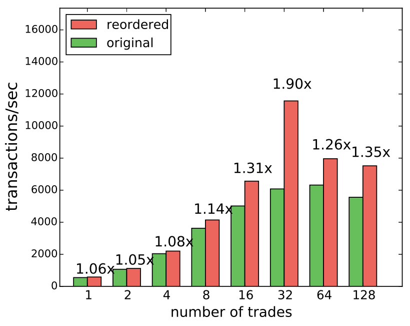
For TPC-E, Figure 3e shows the results of trade update transaction. This benchmark has a while loop which on average runs for 20 iterations. There are multiple read queries in a single iteration, but only one update query. QURO discovered the optimal reordering by putting all the write operations from 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 14%. This might be caused by improved cache locality, since after loop fission, a single table is accessed repeti-



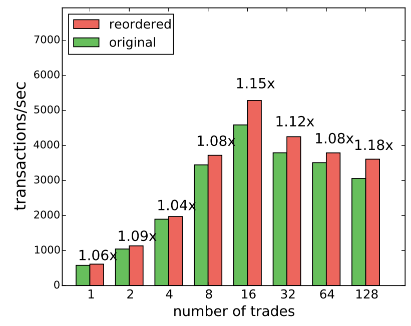
(a) TPC-C payment transaction



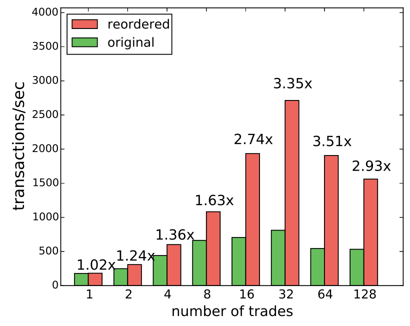
(b) TPC-C new order transaction



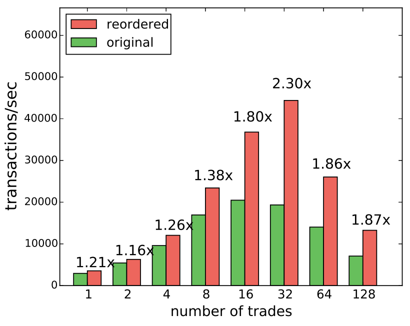
(c) TPC-C mix of payment and new order



(d) TPC-C mix of all transactions



(e) TPC-E trade update transaction



(f) Bidding transaction

Figure 4: Performance comparison: varying number of threads

tively within a single loop, as compared to the original application where multiple tables are accessed in one loop iteration.

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×.

In general, as the data size decreases, the contention rate increases, and the performance gain by reordering becomes larger.

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 working set size, more concurrently running threads indicates a higher level of contention, which hampers the benefits brought by parallelism. The goal of this experiment is to test the performance gain by reordering when the system scales.

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 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.

Figure 4e 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 speeds up the application by up to 3.35×.

Reordering improves performance on the other transactions from the same benchmark, even when queries do not touch contentious tuples. Reordering increases the throughput of the trade order transaction by up to 1.05×, and a mix of trade order and trade result transactions by up to 1.57×. Trade status is a read-only transaction, so reordering has the same throughput as the original implementation. Detailed results and analysis can be found in the technical report [6].

Finally, Figure 4f 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×.

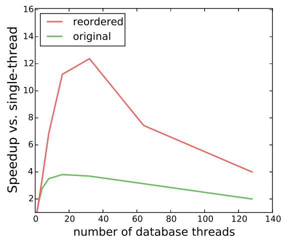
We also compared how each benchmark scales as the number of threads increases by evaluating self-relative speedup. The baseline for both implementations is the single-thread execution throughput of the original implementation, and Figure 5 shows how throughput changes as the number of threads increases. We only present the results of the payment and trade update transaction here while the results for other transactions can be found in the technical report. As shown in the figure, the reordered implementation has much larger self-relative speedup than the original implementation as the number of threads exceeds 8. 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

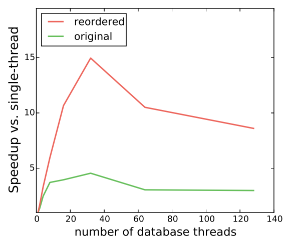
We did another set of experiments to give detailed analysis about the performance to gain insights about reordering.

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 1 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 was executed serially. By comparing the values to the single-thread query time on each row, we can infer the lock waiting time of that



(a) TPC-C payment



(b) TPC-E trade update

Figure 5: Self relative speedup comparison

	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 payment transaction. Ratio=reordered/original. Ratio in the last column indicates the reduce in latency. The reordered implementation reduce the latency by 85%.

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 every transaction updates the same tuple. By reordering, the running time of this query is significantly shortened: the query time in 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 makes the running time longer.

We also profiled the new order transaction. In this transaction, the flexibility of reordering is restricted by the data dependency between queries. In particular, query 2 reads a tuple from district table and reserves for updates. This query is most likely to cause waiting among all queries. 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. The limitation in reordering explains the less-significant shortening of time on that query. The query breakdown of new order transaction is shown in Table 3.

The above analysis indicates a tradeoff with reordering. Reordering decreases the time spent on lock waiting on conflicting queries, but increases the time spent on non-conflicting queries. Since non-conflicting queries are not likely to cause waiting or stalling of transactions, the decrease in waiting time on the lock of contentious data by reordering usually outweighs the increase of locking time in queries that access non-contentious data.

	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 new order transaction. The reordered implementation reduce the latency by 59%.

#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, ratio=reordered/original. With 4K trades, reordering can reduce the abort rate by up to 91%.

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 tuple. When the number of trades is small, concurrent transactions are likely to access the same set of trades, but in 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 6 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 which are not being modified. We define deadlock window of a transaction T1 to be a time range: if another transaction T2 starts within this range, T2 and T1 might have deadlock. 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 wrote a "worst-case" implementations where the most contentious queries are issued *first* within the transaction. We then compared the performance of those implementations against the original and the implementations generated by QURO, with the most contentious queries are issued as late as possible. The results for the TPC-C payment transaction are shown in Figure 7 (results for other benchmarks are presented in the technical report), with the throughput ratio of best and worst case implementation labeled. The results confirm our hypothesis: the implementation with the most contentious queries executed first show 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.

6.6 Varying Buffer Pool Size

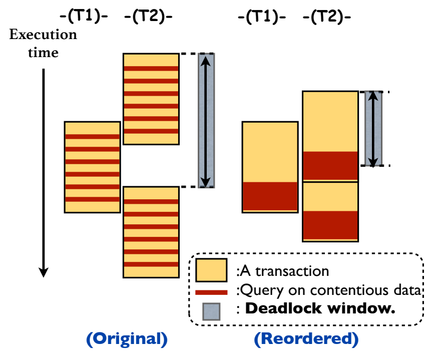


Figure 6: Execution of trade update before and after reorder. Conflicting window is the time range where if T2 starts within the range, T2 is likely to have deadlock with T1.

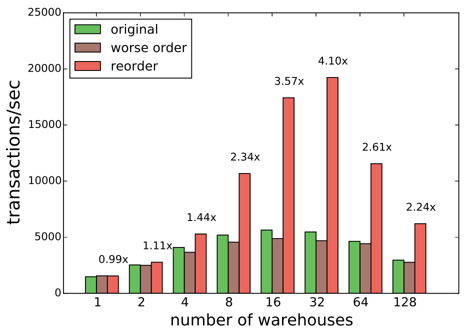


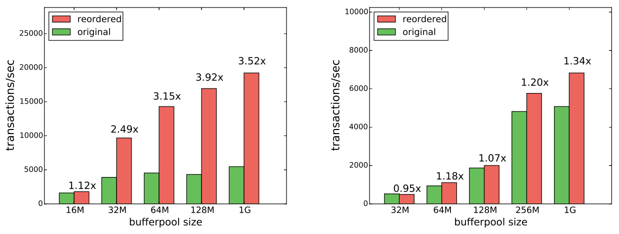
Figure 7: TPC-C worst-case ordering comparison

Next we measured the performance with different buffer pool sizes. We changed the buffer pool size to vary the time spent on disk operations. Figure 8 shows the performance comparison of the TPC-C transactions running at different buffer pool sizes. When the buffer pool is small, disk operation dominates the transaction processing time, and the performance gain of reordering is trivial. Increasing buffer pool size decreases the time spent on disk, and that increases throughput as a result of reordering. Furthermore, it also shows that even when the database is not completely in-memory, reordering can still improve the throughput.

6.7 Performance Comparison with Stored Procedures

In previous experiments, we ran our client program and database server on the same machine to minimize the effect of network round trips as a result of issuing queries. In this experiment we compare the performance of the original, reordered, and an implementation using 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 9 shows the results of TPC-C payment transaction in this experiment. When there is little data contention, the stored procedure implementation outperforms the others due to less 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, and the reordered implementation has significantly higher throughput than the stored procedure implementation. When running with 32 threads, the stored procedure increases the throughput by 1.50× as compared to 6.53× from the reordered implementation.



(a) TPC-C payment

(b) TPC-C neworder

Figure 8: Performance comparison: varying buffer pool size

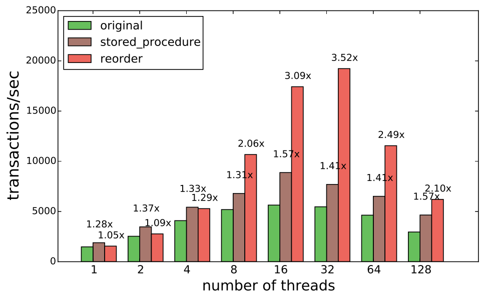


Figure 9: TPC-C performance comparison to stored procedure

6.8 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 generate the ILP for each transaction, 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 [7] and gurobi [5], to solve the generated programs with a time out of 2 hours. The algorithm for restructuring the transaction code is described in Section 4.5, which is only needed with optimization, 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 indicates 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 time out for transaction 2. In contrast, the optimized formulation solves much faster, allowing QURO to process larger transaction code inputs.

7. RELATED WORK

Besides lock-based methods, various concurrency control mechanisms, besides locking, have been proposed, such as optimistic concurrency control [19] and multi-version concurrency control [8]. However, Yu et al. [25] studies the scalability of different concurrency control scheme for main-memory DBMS, and their results show that on write-intensive, highly contentious workload, 2PL has the best performance among all the concurrency control schemes, and scales better to hundreds of cores.

There has been work done on improving the efficiency of locking-based concurrency control schemes. Shore-MT [17] applies 2PL and provides many system-level optimization to achieve high scalability on multi-core machines. Jung et al. [18] implemented a lock manager in MySQL and improves scalability by enabling lock allocation and deallocation in bulk. Horikawa [16] adapted a latch-free 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 [14, 10] is a program analysis and transformation tool that optimize database application performance by query rewriting. Sloth [12] and Pyxis [11] are other 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. [13] show that lazy transaction processing improves cache locality and achieves better load balancing. It uses 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, and uses different optimization techniques to effectively reduce the solving time required. Our experiments show that QURO can find reorderings that can reduce latency up to 85% along with a 6.53× improvement in throughput as compared to open source implementations.

9. REFERENCES

[1] 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] Leveraging lock contention to improve oltp application performance, technical report. `http://people.csail.mit.edu/congy/papers/reorder_techreport.pdf`.

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

[8] P. A. Bernstein and N. Goodman. Concurrency control in distributed database systems. *ACM Comput. Surv.*, 13(2):185–221, June 1981.

[9] P. A. Bernstein, D. W. Shipman, and W. S. Wong. Formal aspects of serializability in database concurrency control. volume 5, pages 203–216, May 1979.

[10] 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*, 2011.

[11] A. Cheung, S. Madden, O. Arden, and A. C. Myers. Automatic partitioning of database applications. *Proc. VLDB Endow.*, 5(11):1471–1482, 2012.

[12] A. Cheung, S. Madden, and A. Solar-Lezama. Sloth: Being lazy is a virtue (when issuing database queries). In *Proc. SIGMOD*, pages 931–942, 2014.

[13] 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*, 2014.

[14] R. Guravannavar and S. Sudarshan. Rewriting procedures for batched bindings. In *Proceedings of the VLDB Endowment*, 2008.

[15] S. Harizopoulos, D. J. Abadi, S. Madden, and M. Stonebraker. Oltp through the looking glass, and what we found there. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data*, 2008.

[16] 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*, 2013.

[17] 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.

[18] 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*, 2013.

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

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

[21] J. E. Smith and A. R. Pleszkun. Implementation of precise interrupts in pipelined processors. In *Proc. of the 12th annual Intl. Symp. on Computer Architecture*, 1985.

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

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

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

[25] 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*, 2014.