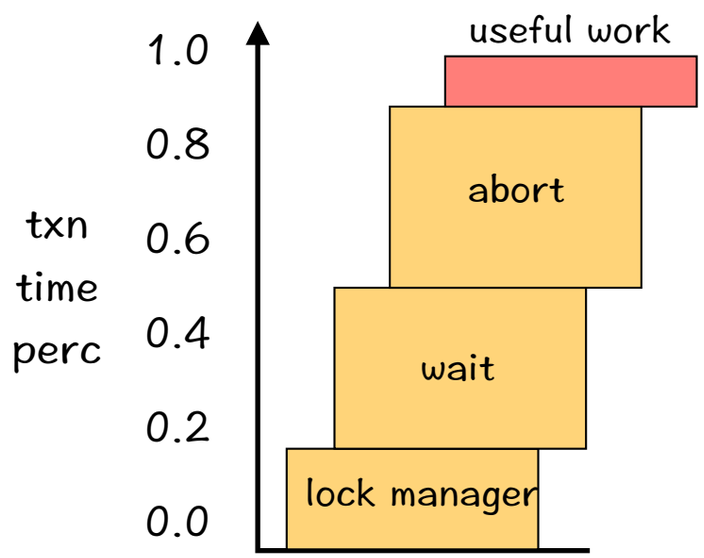
Leveraging Lock Contention to Improve Transaction Applications

Cong Yan Alvin Cheung
University of Washington

Background

- Pessimistic locking-based CC protocols
  - Poor performance under high data contention
    - long lock waiting time and high abort rate

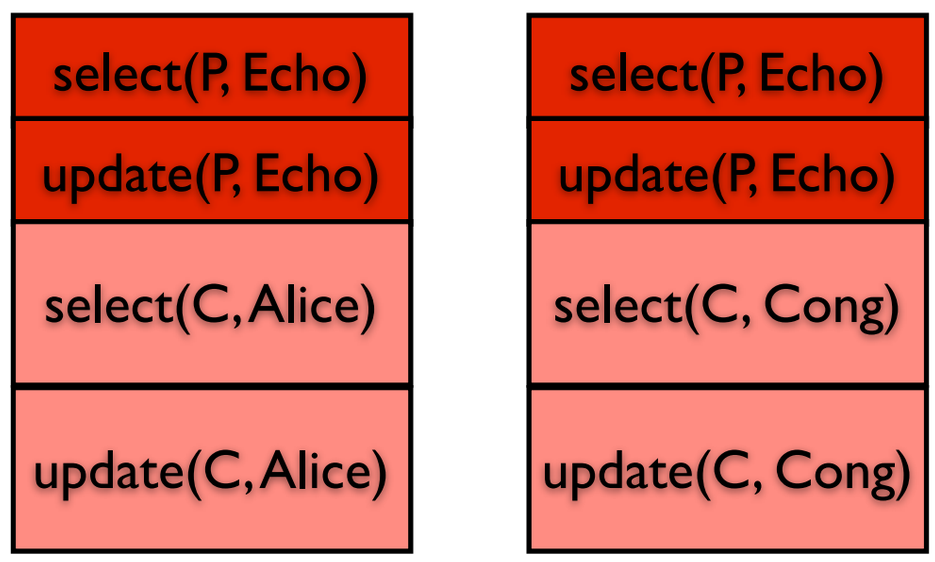


Time breakdown for YCSB transaction under high data contention
Main-memory DBMS, 2PL
("*Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores*", VLDB15)

Two-phase Locking

- Example transaction: online shopping
  - Alice and Cong buying Echo at the same time

```
void transaction(){
  v = select(P, Echo);
  update(P, Echo);
  b = select(C, Alice);
  update(C, Alice);
}
```




```
void transaction(){
  v = select(P, Echo);
  update(P, Echo);
  b = select(C, Cong);
  update(C, Cong);
}
```

P = Product table
C = Customer table

Two-phase Locking

Original txn

Execution time

--(T1)--

--(T2)--

select(P, Echo)

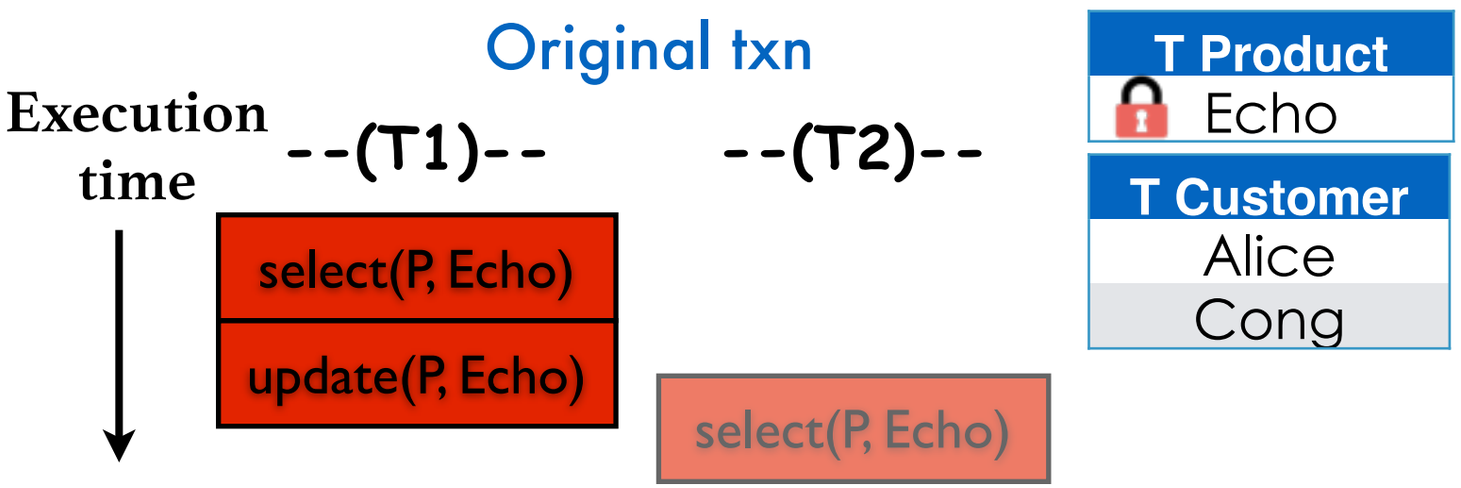
T Product
Echo

T Customer
Alice
Cong

:shared lock

:exclusive lock

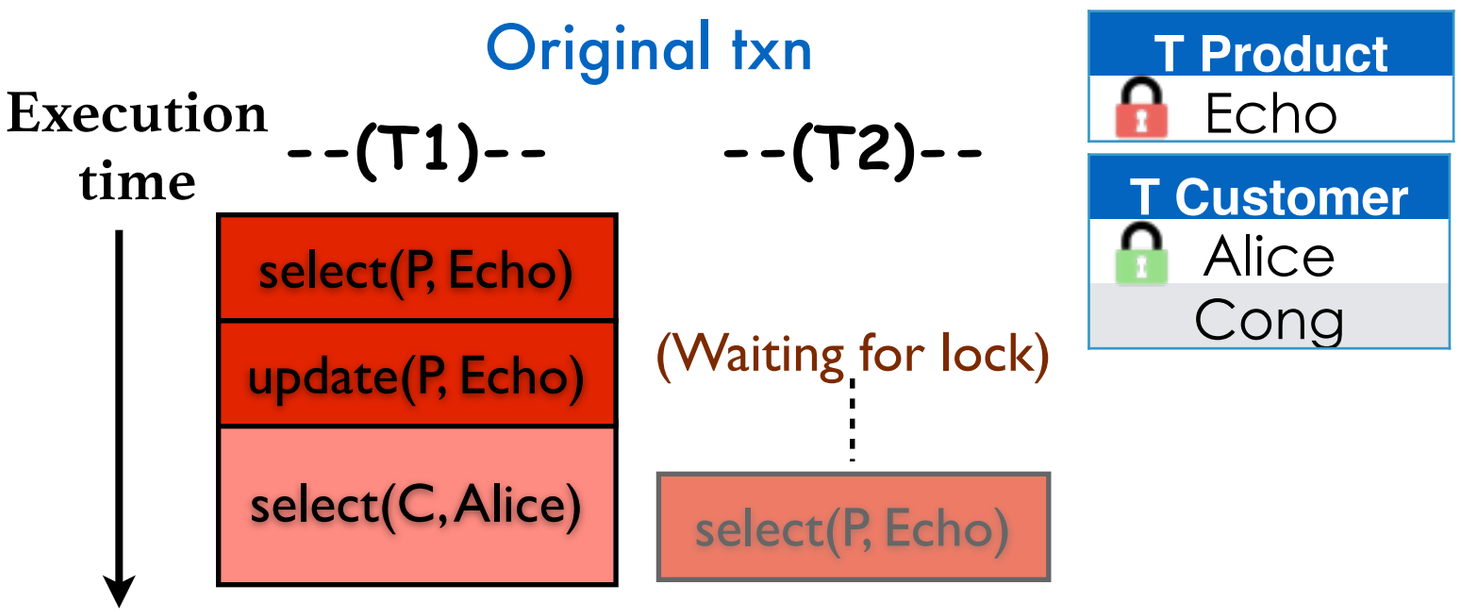
Two-phase Locking



:shared lock

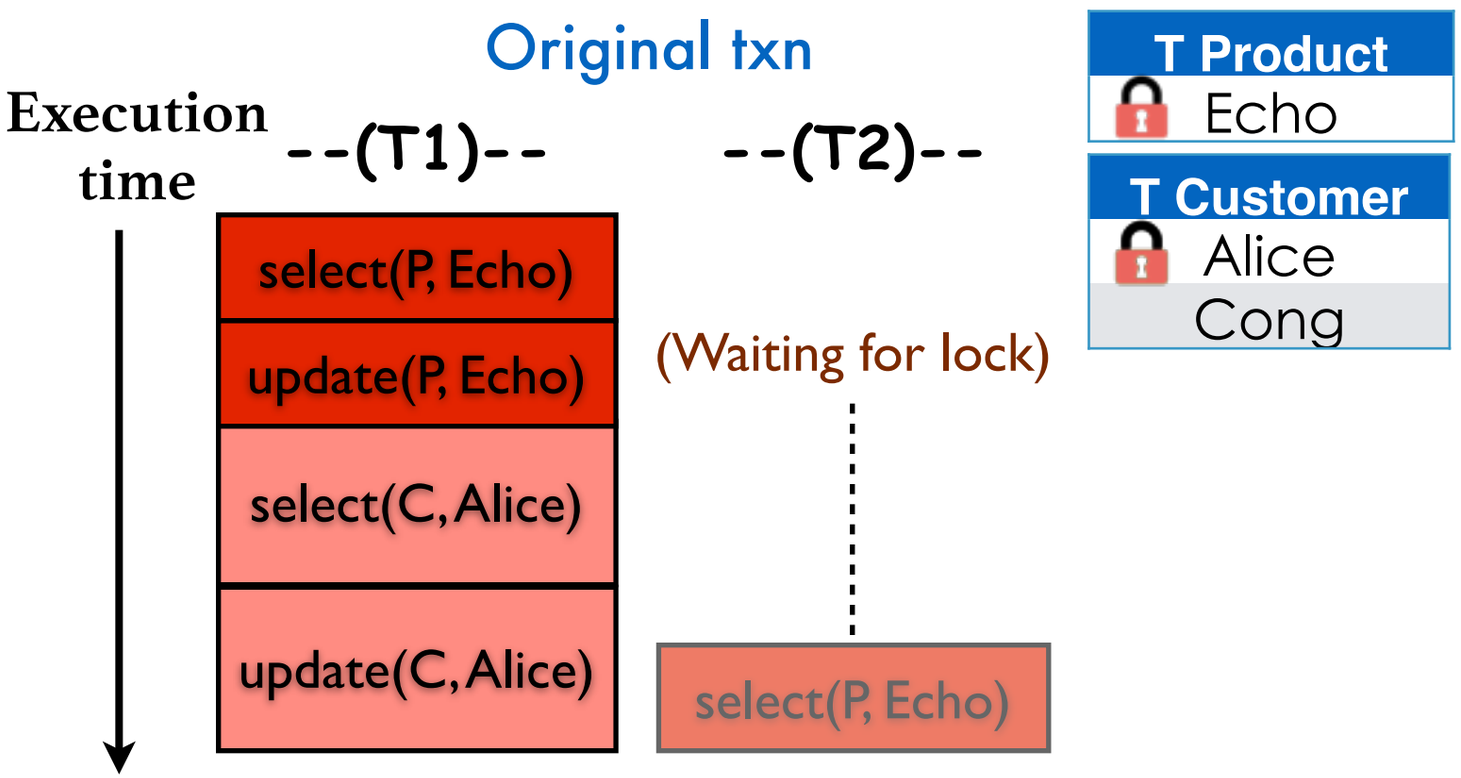
:exclusive lock

Two-phase Locking



:shared lock
:exclusive lock

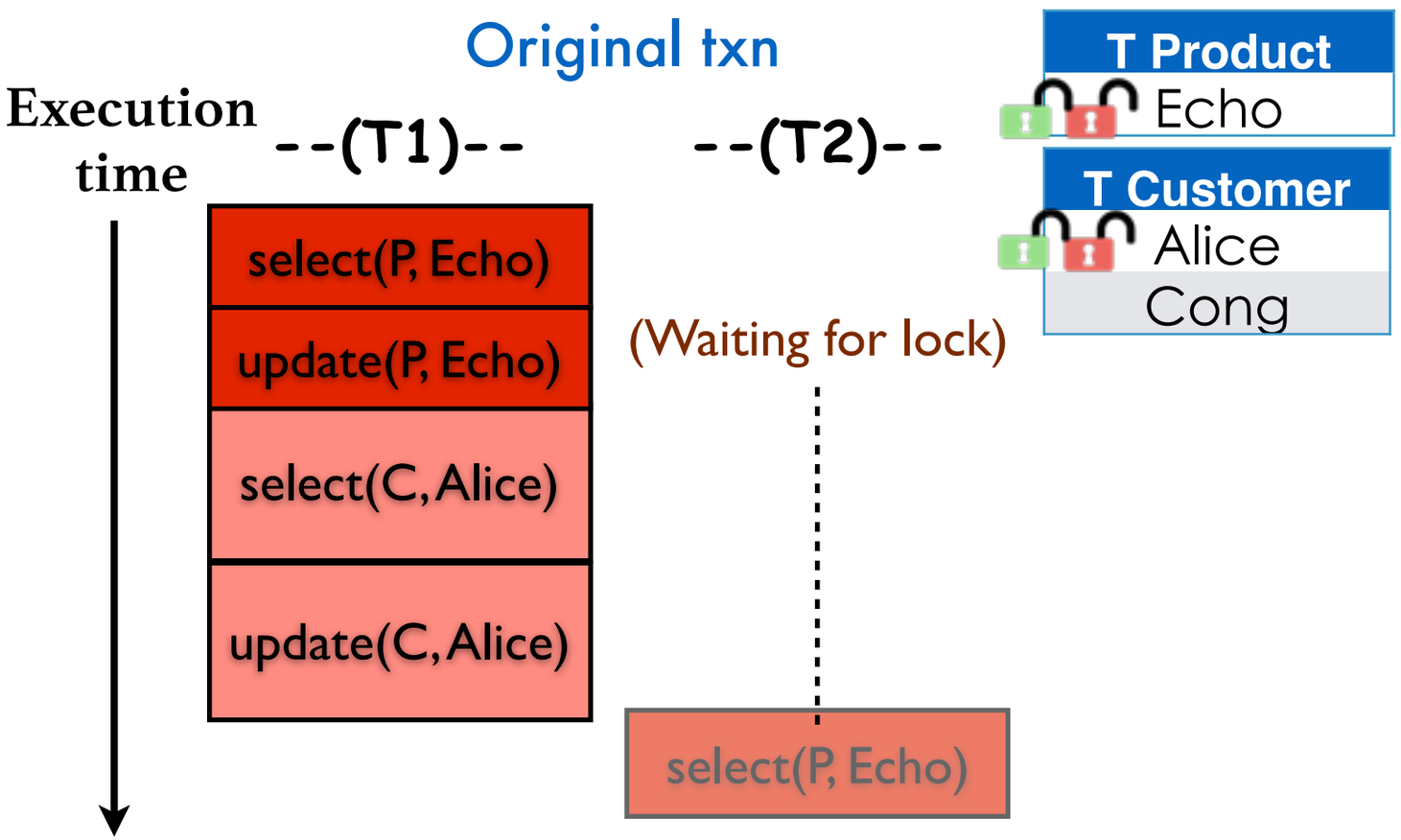
Two-phase Locking



:shared lock

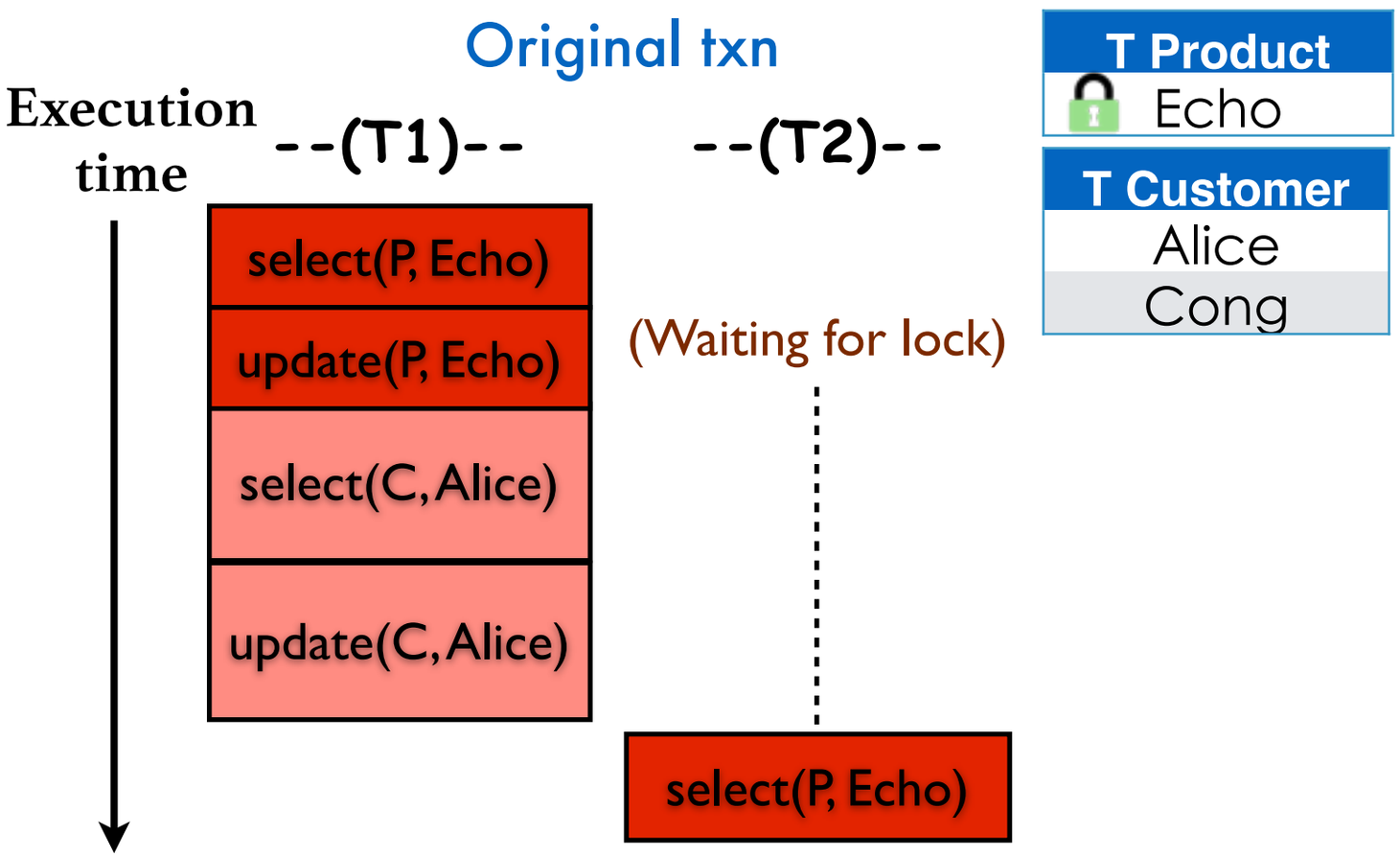
:exclusive lock

Two-phase Locking



:shared lock
:exclusive lock

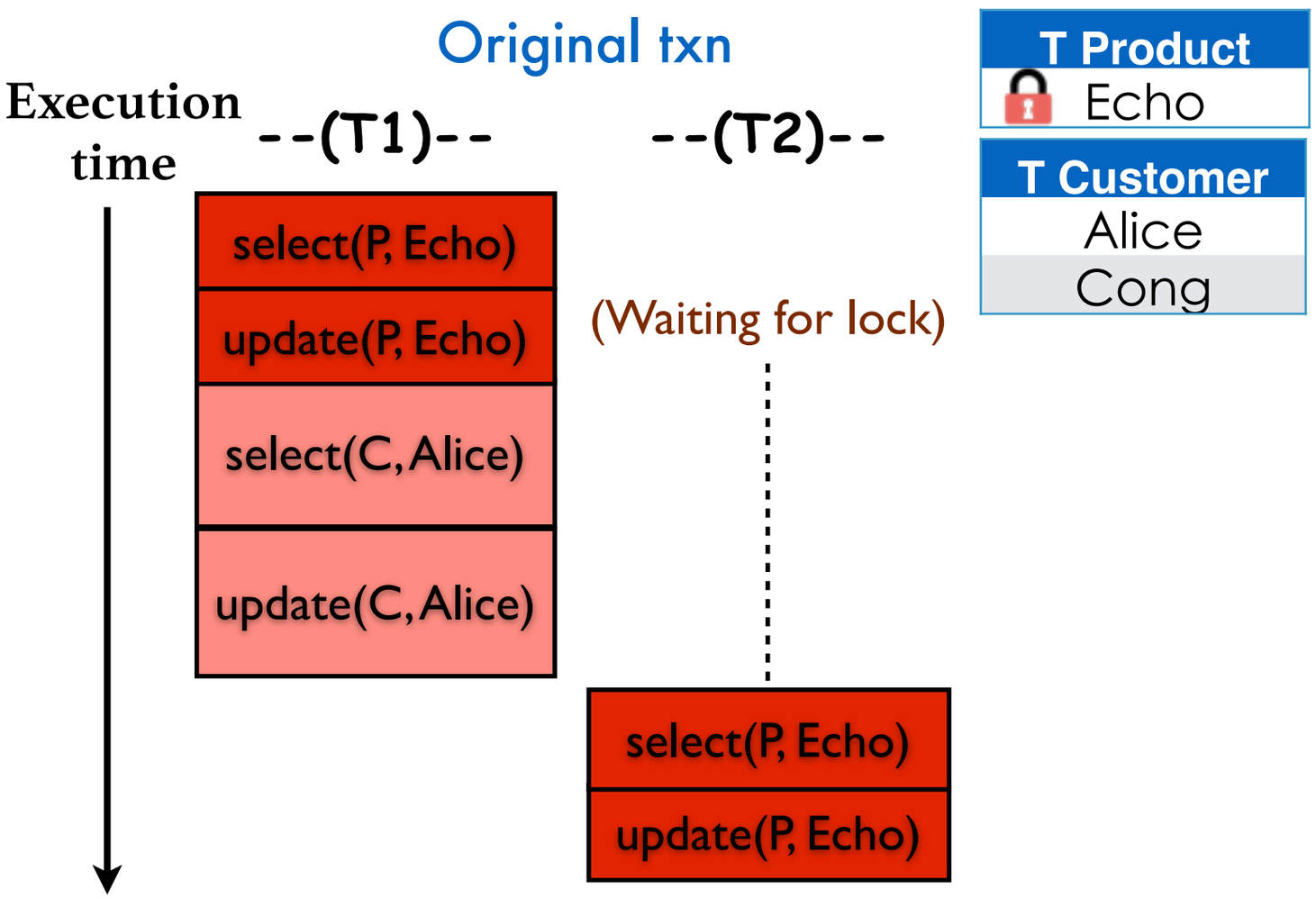
Two-phase Locking



:shared lock

:exclusive lock

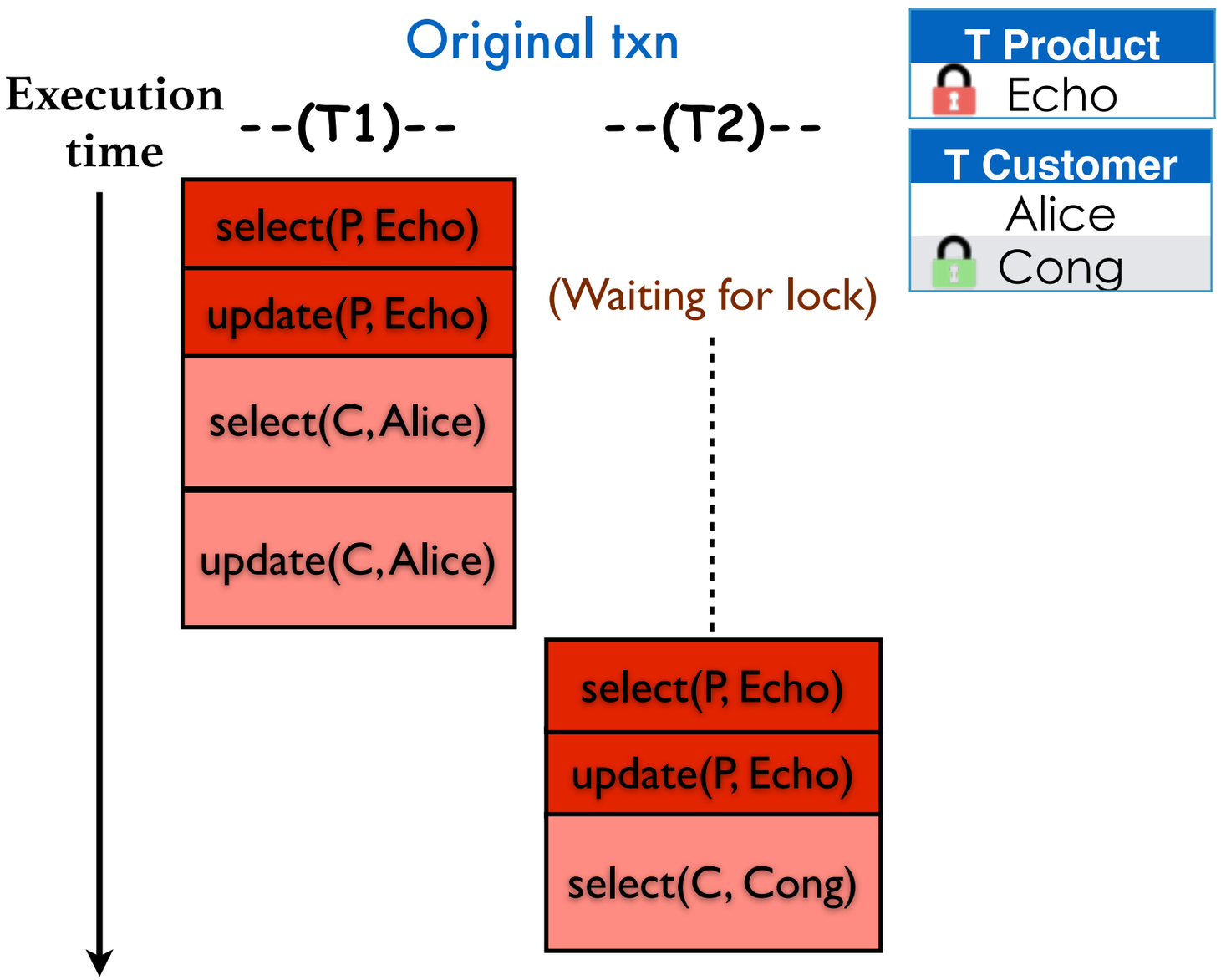
Two-phase Locking



:shared lock

:exclusive lock

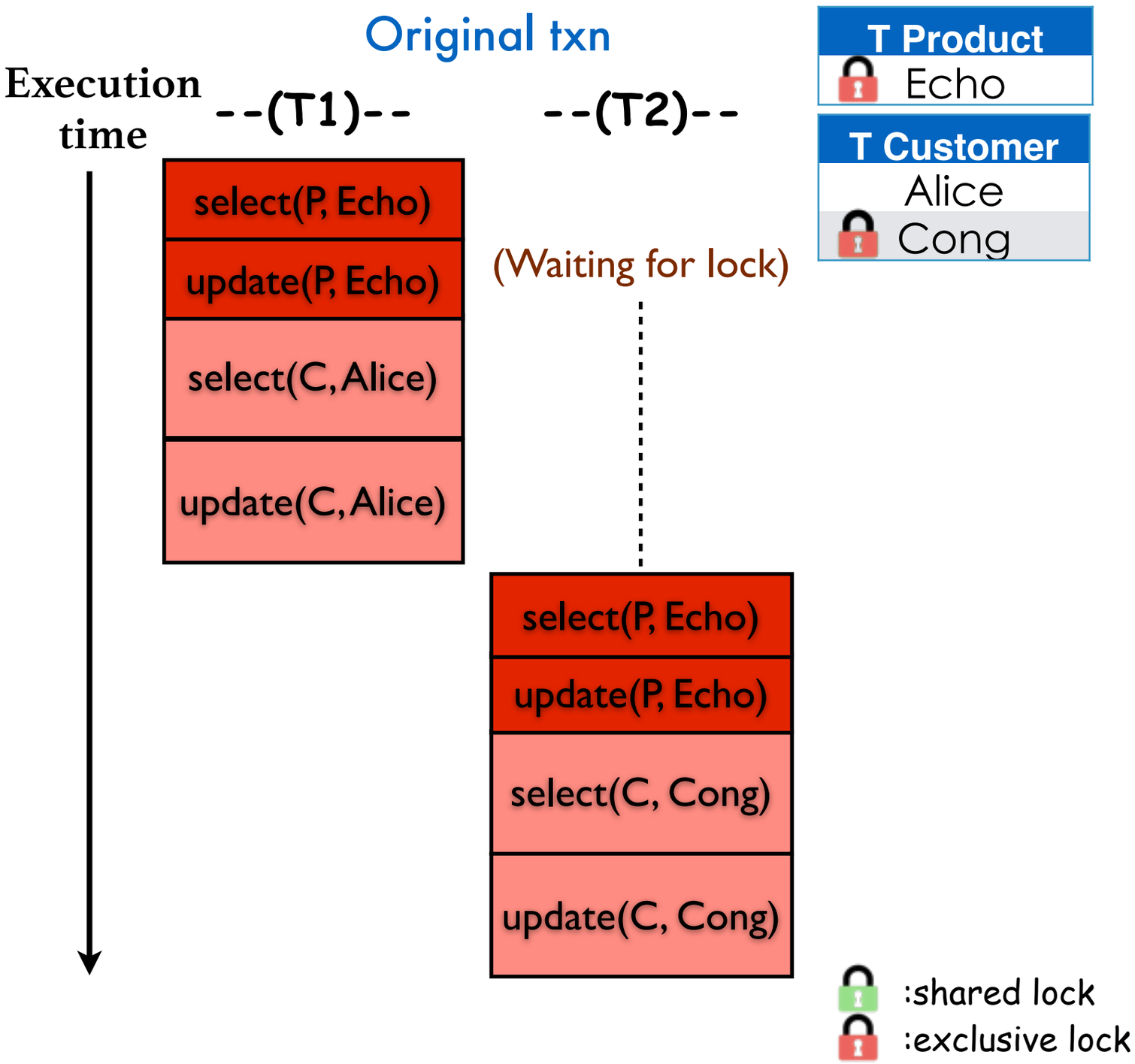
Two-phase Locking



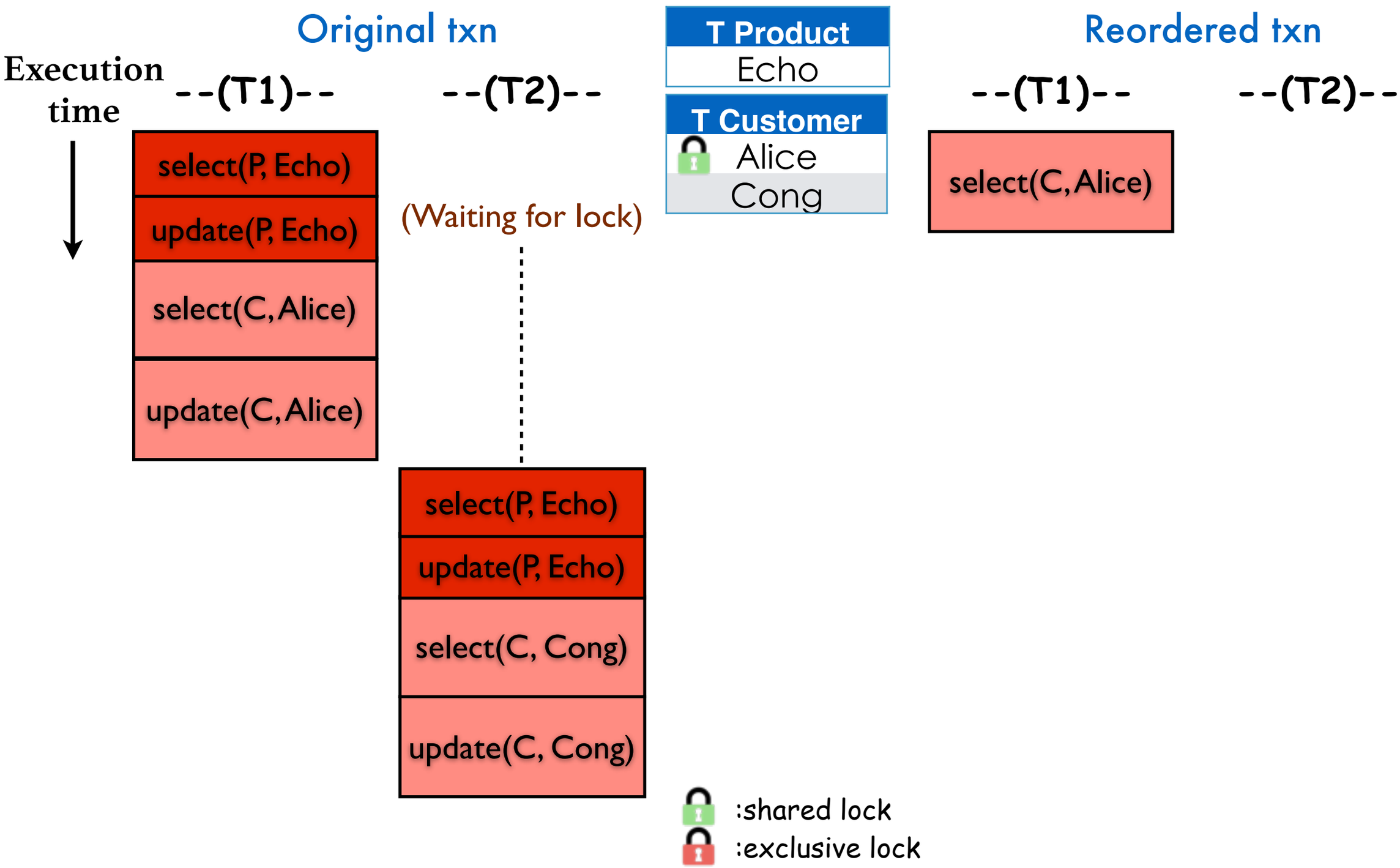
:shared lock

:exclusive lock

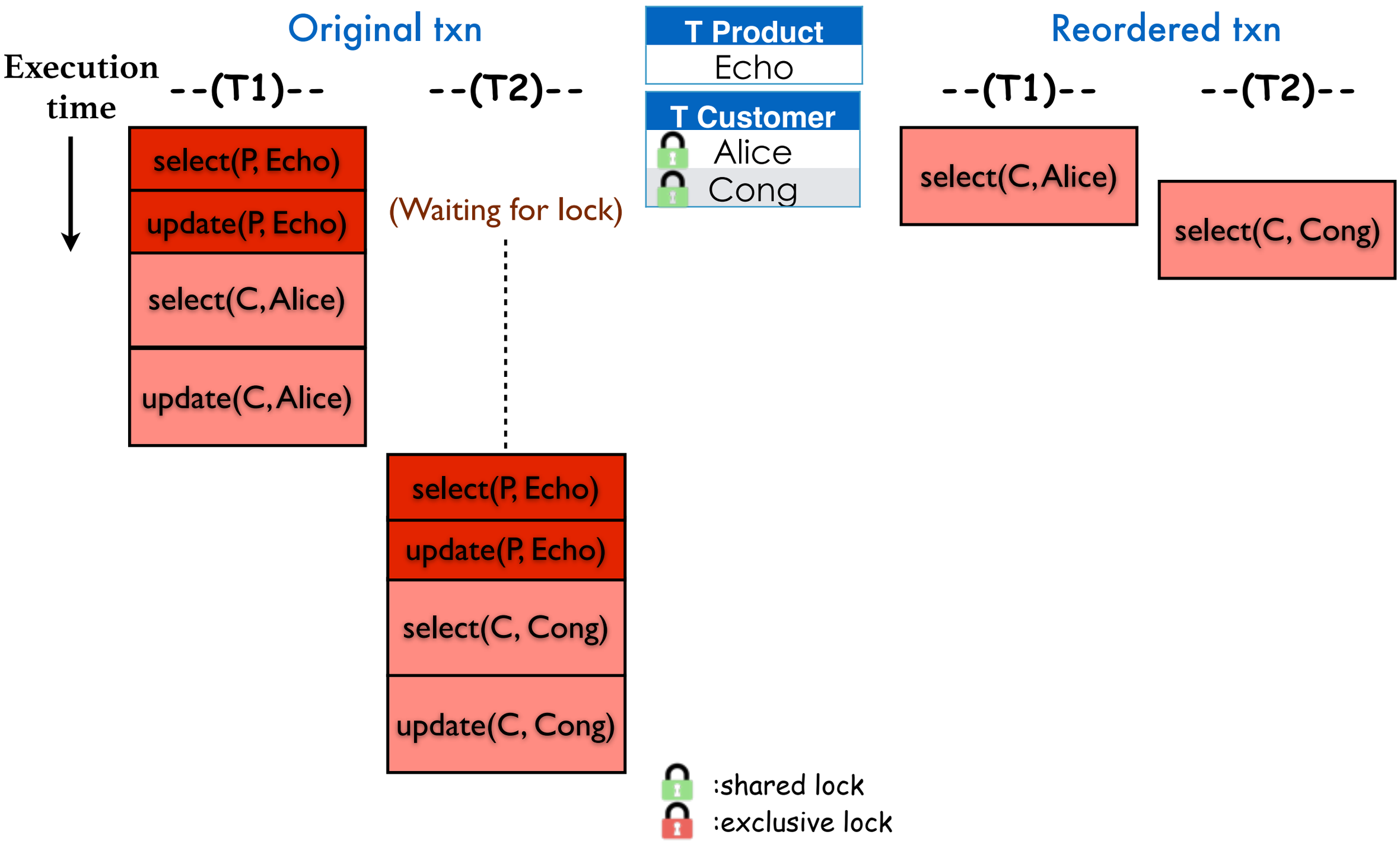
Two-phase Locking



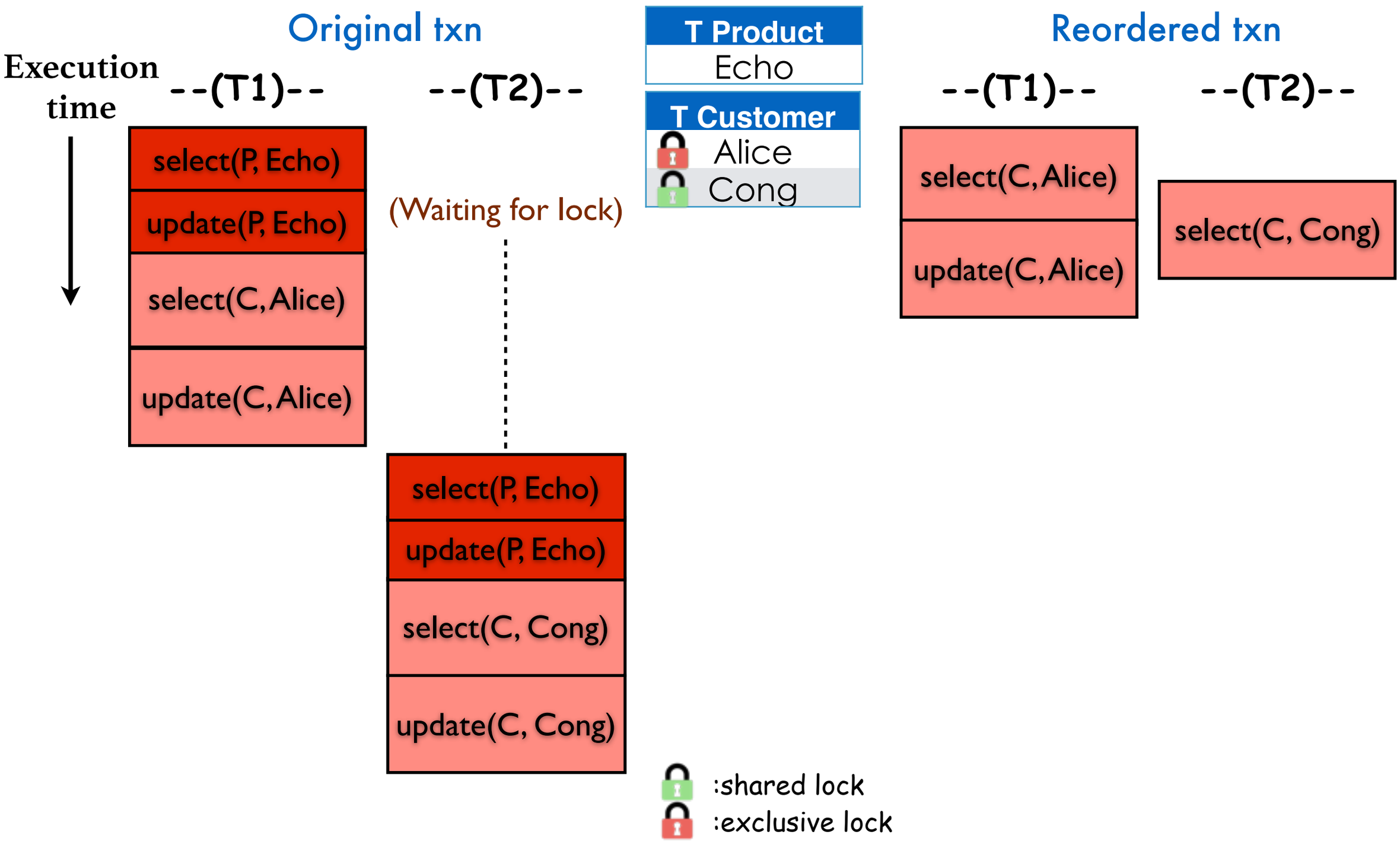
Two-phase Locking



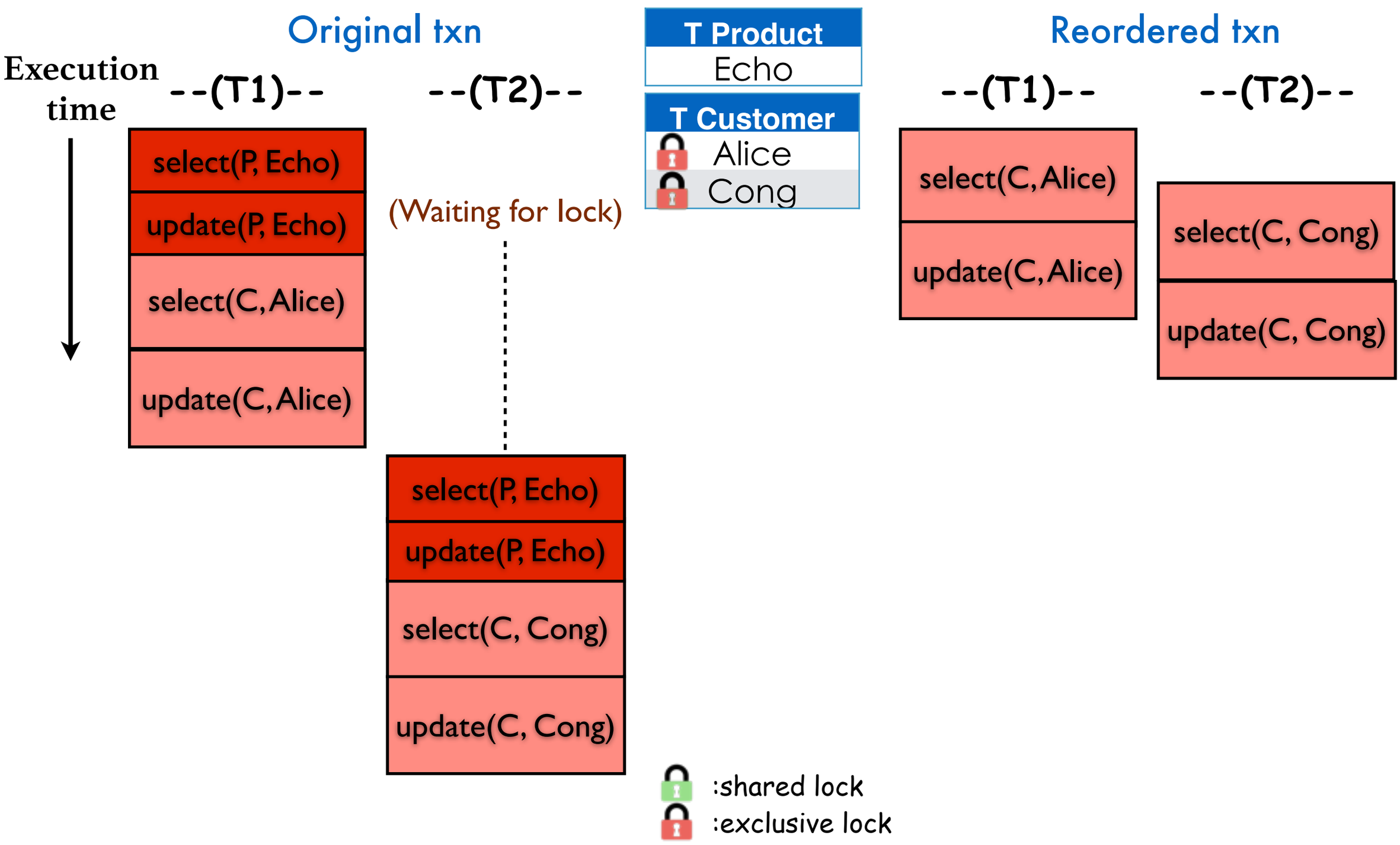
Two-phase Locking



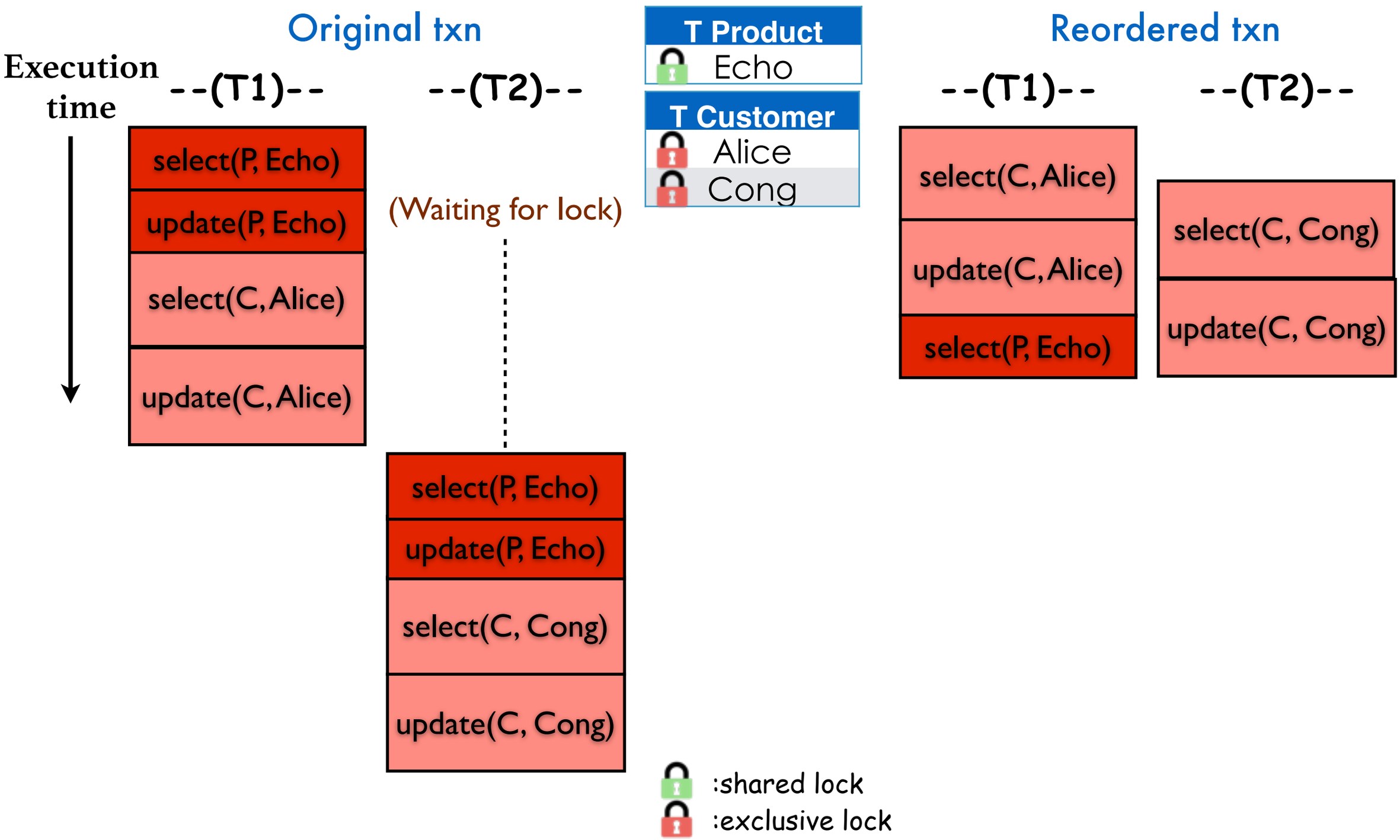
Two-phase Locking



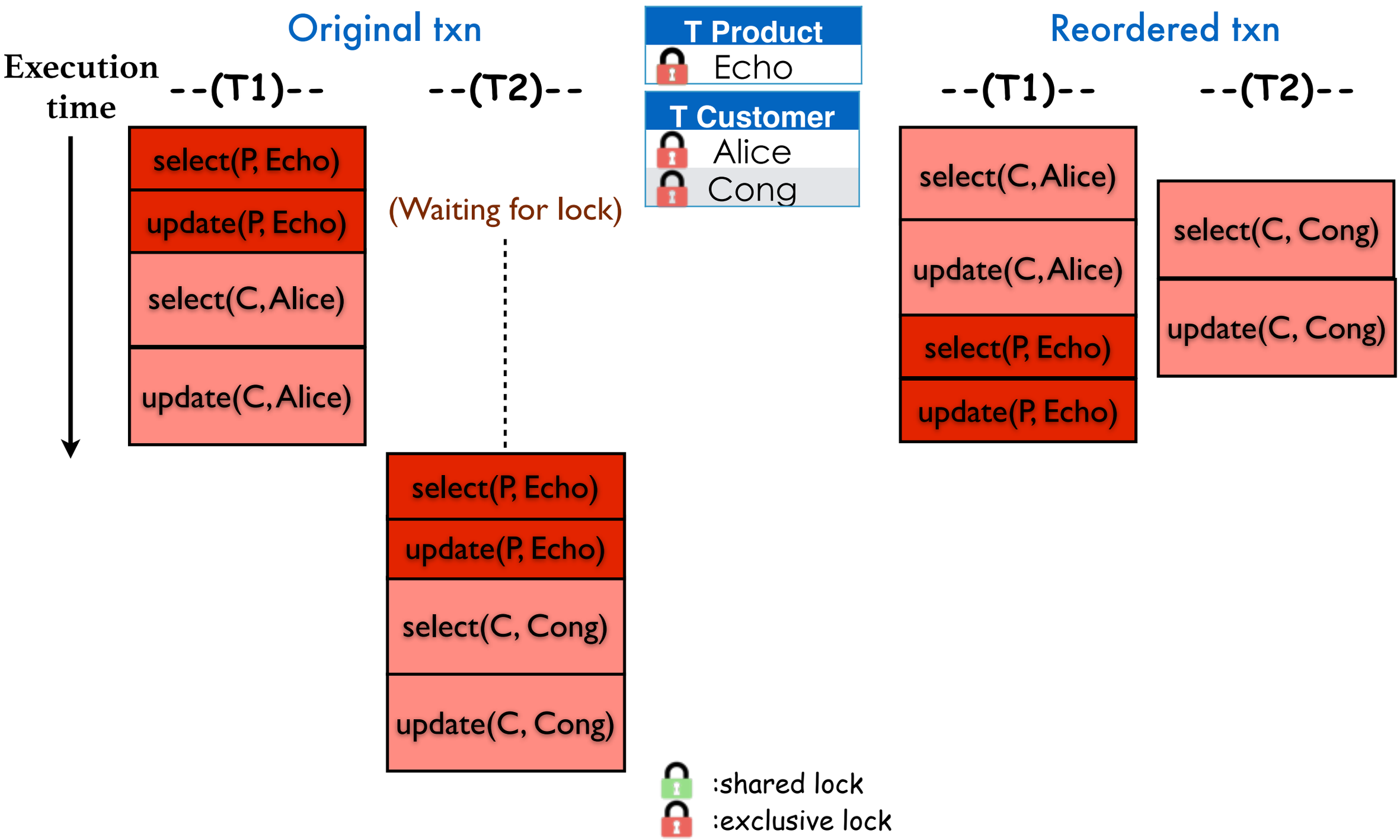
Two-phase Locking



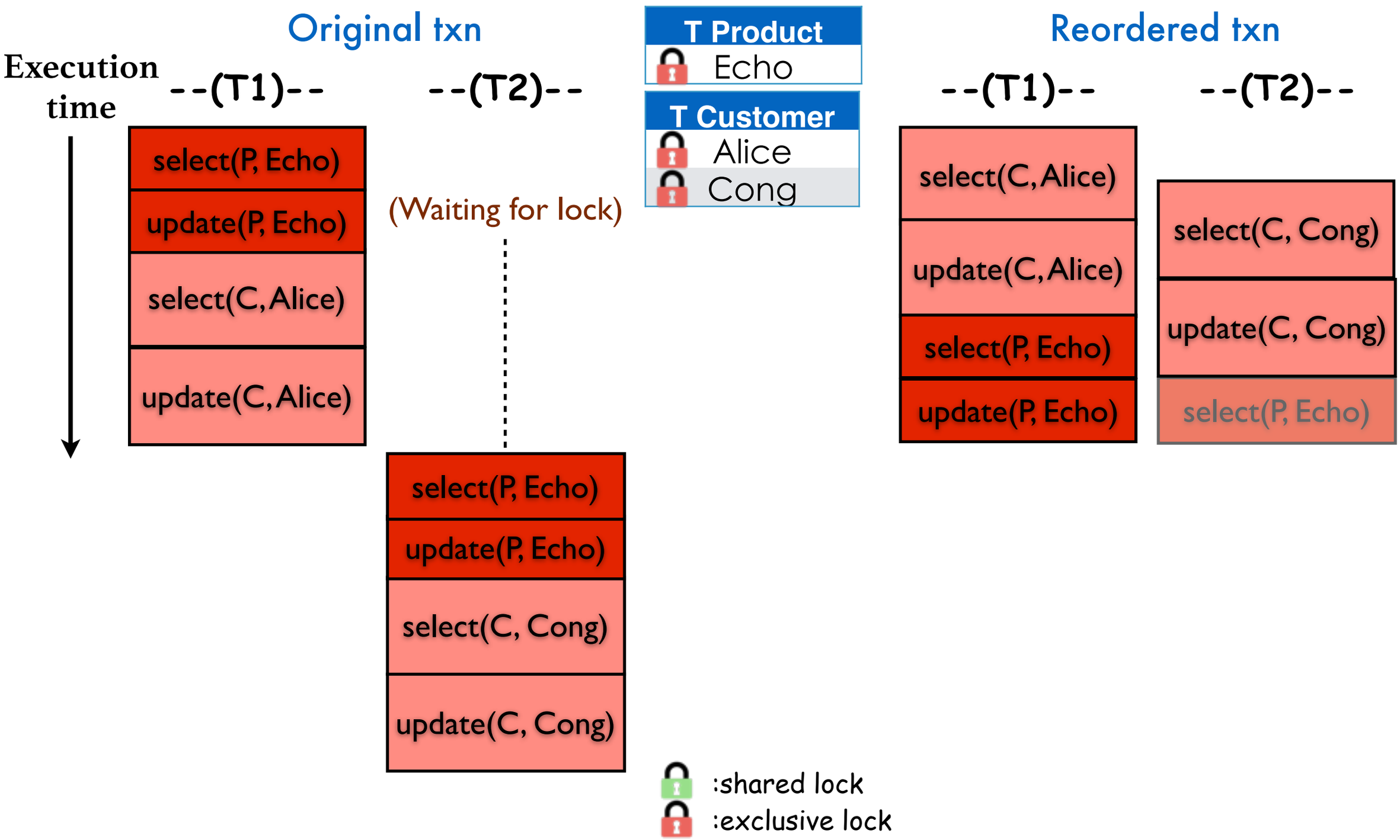
Two-phase Locking



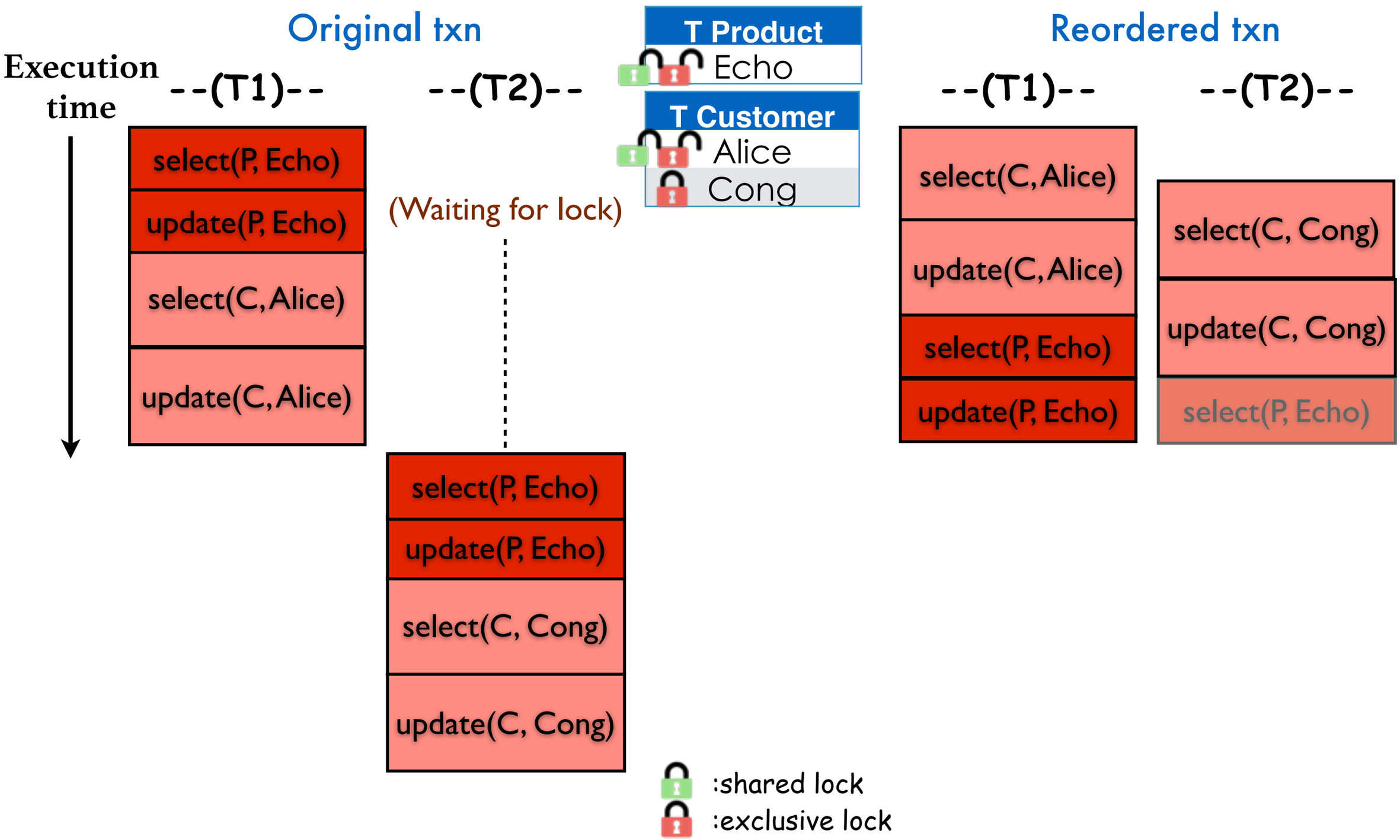
Two-phase Locking



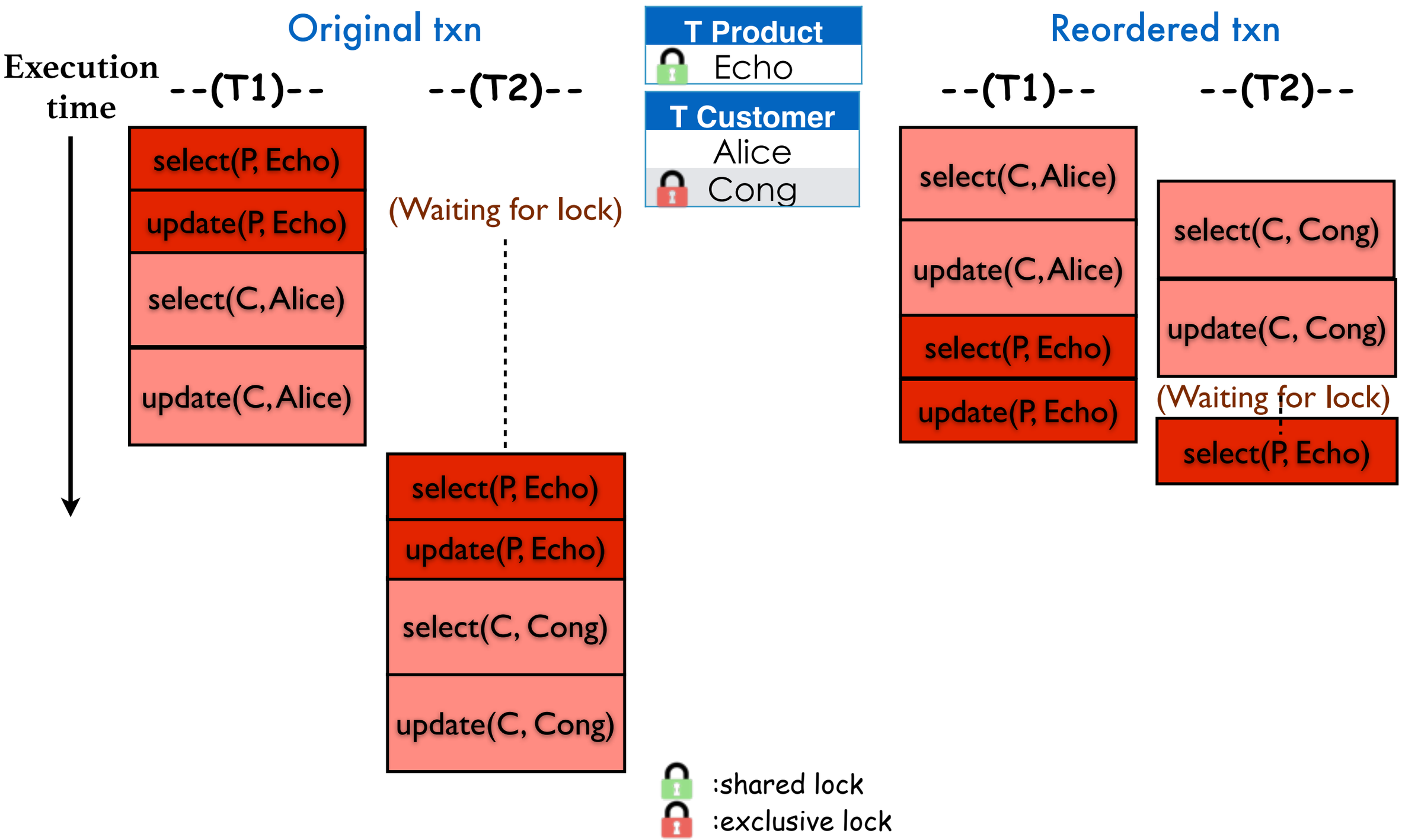
Two-phase Locking



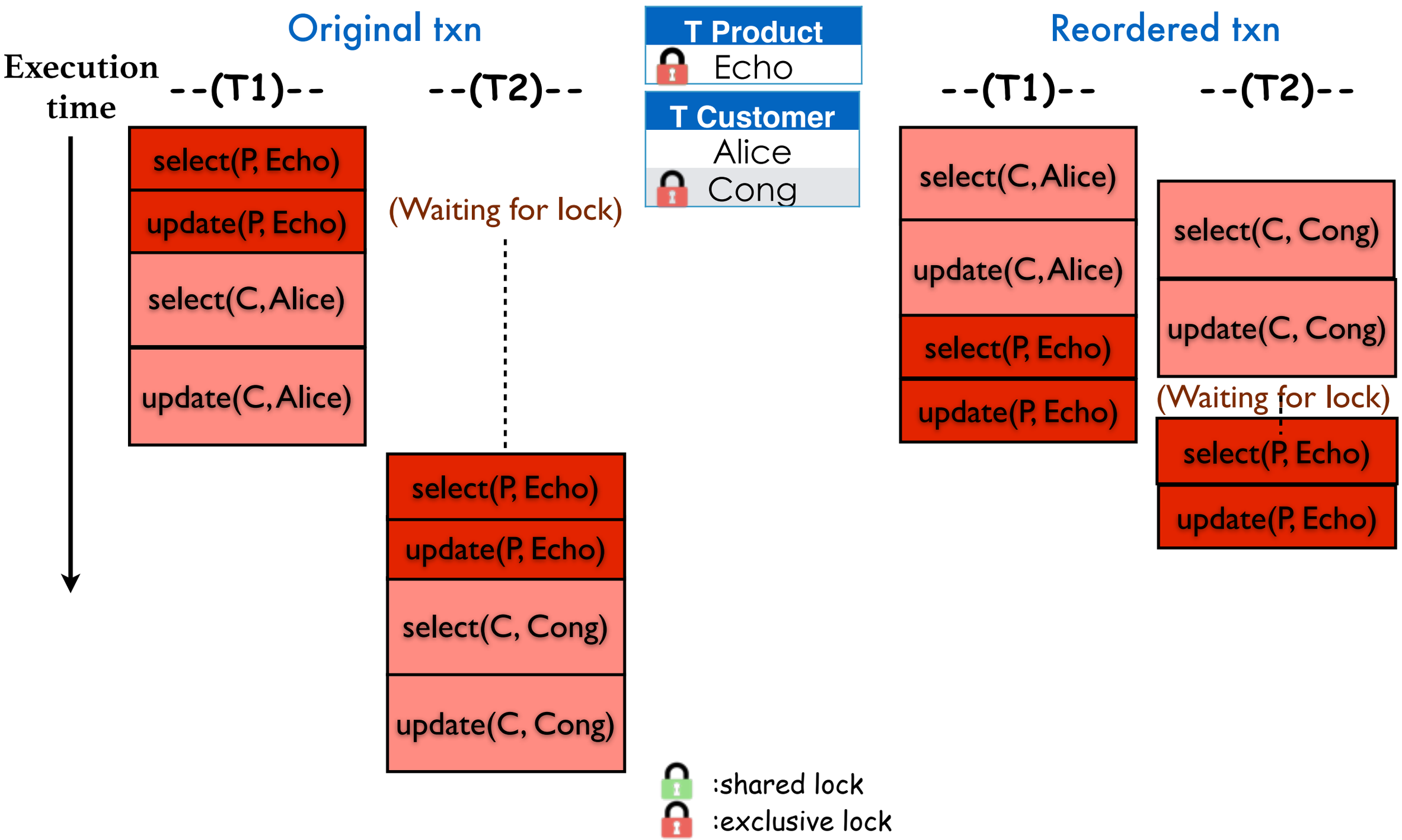
Two-phase Locking



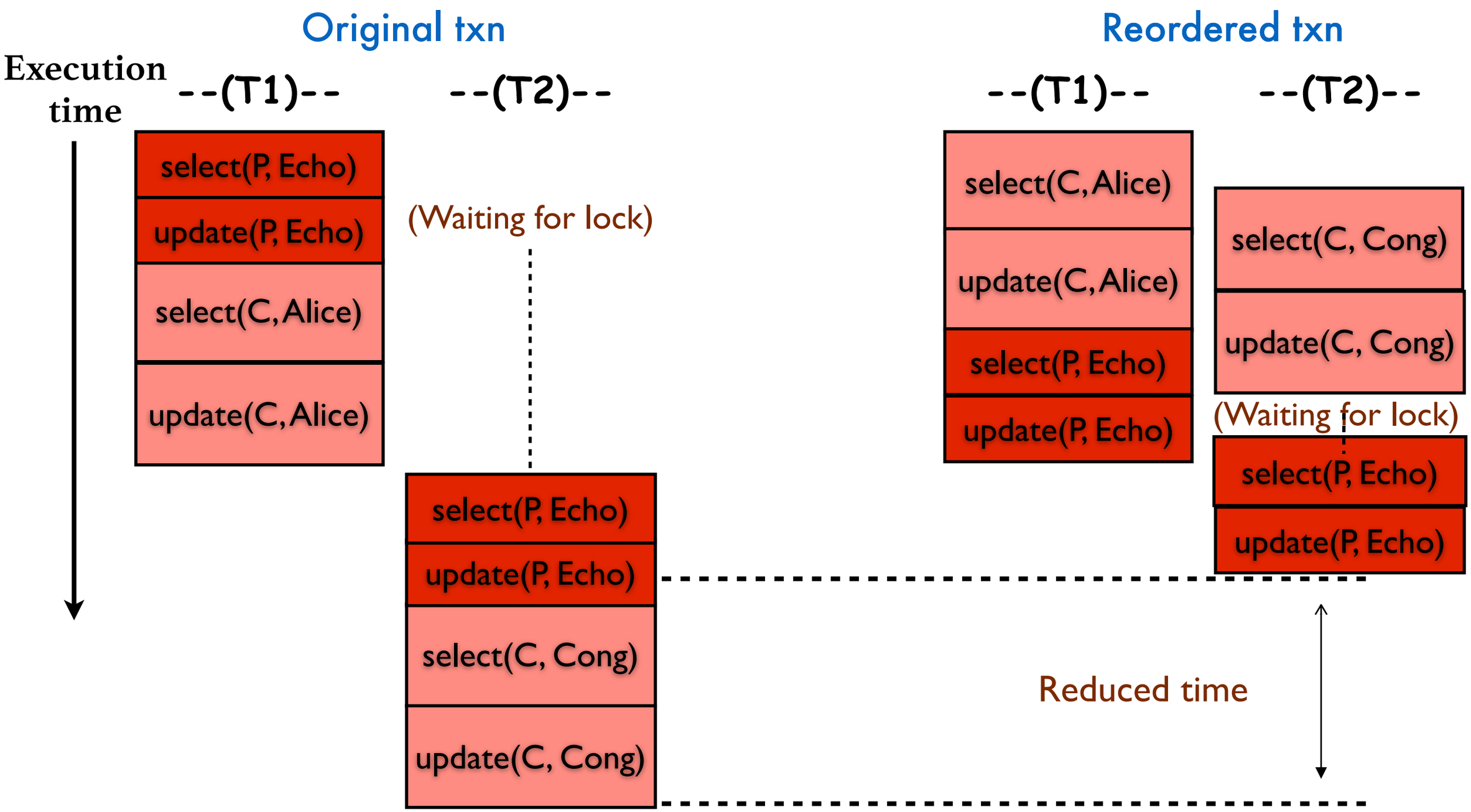
Two-phase Locking



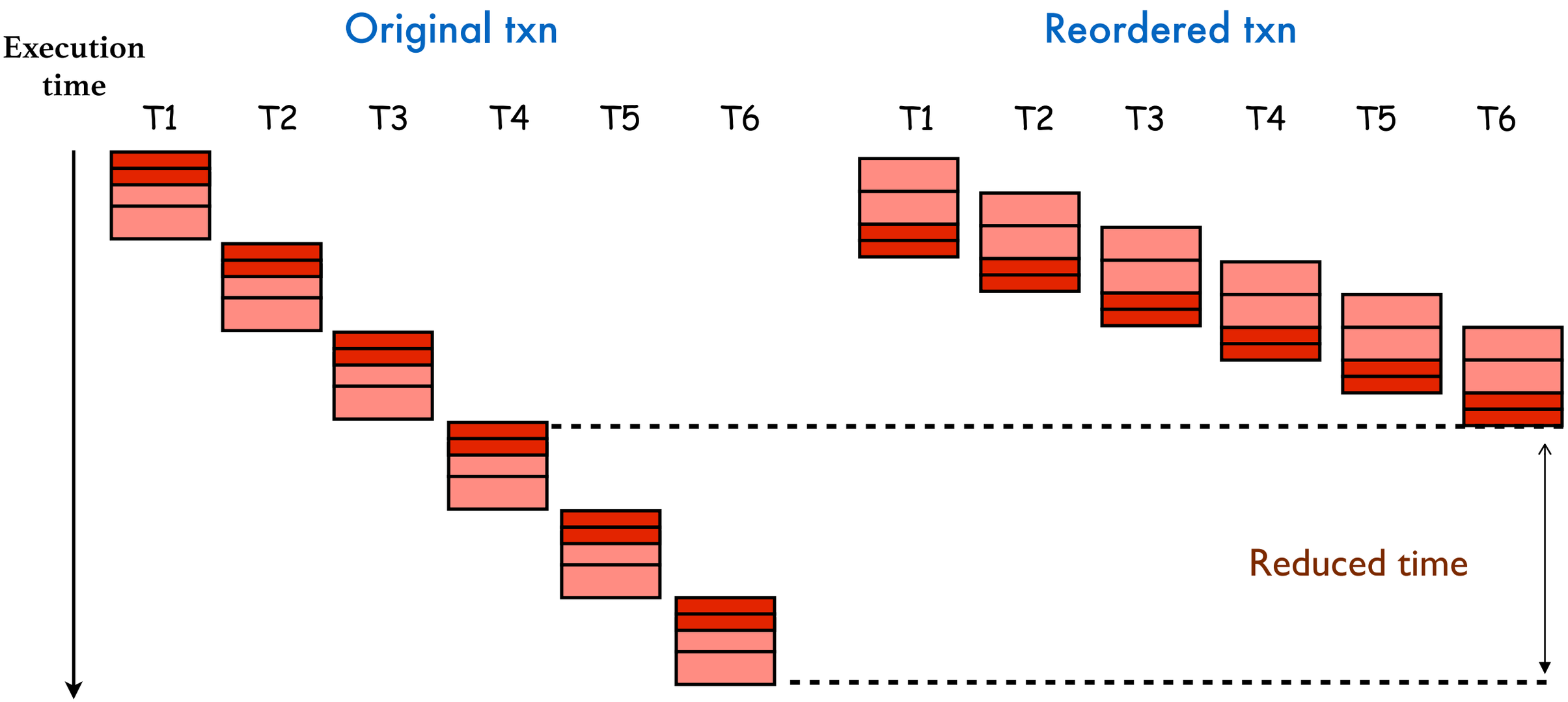
Two-phase Locking



Two-phase Locking

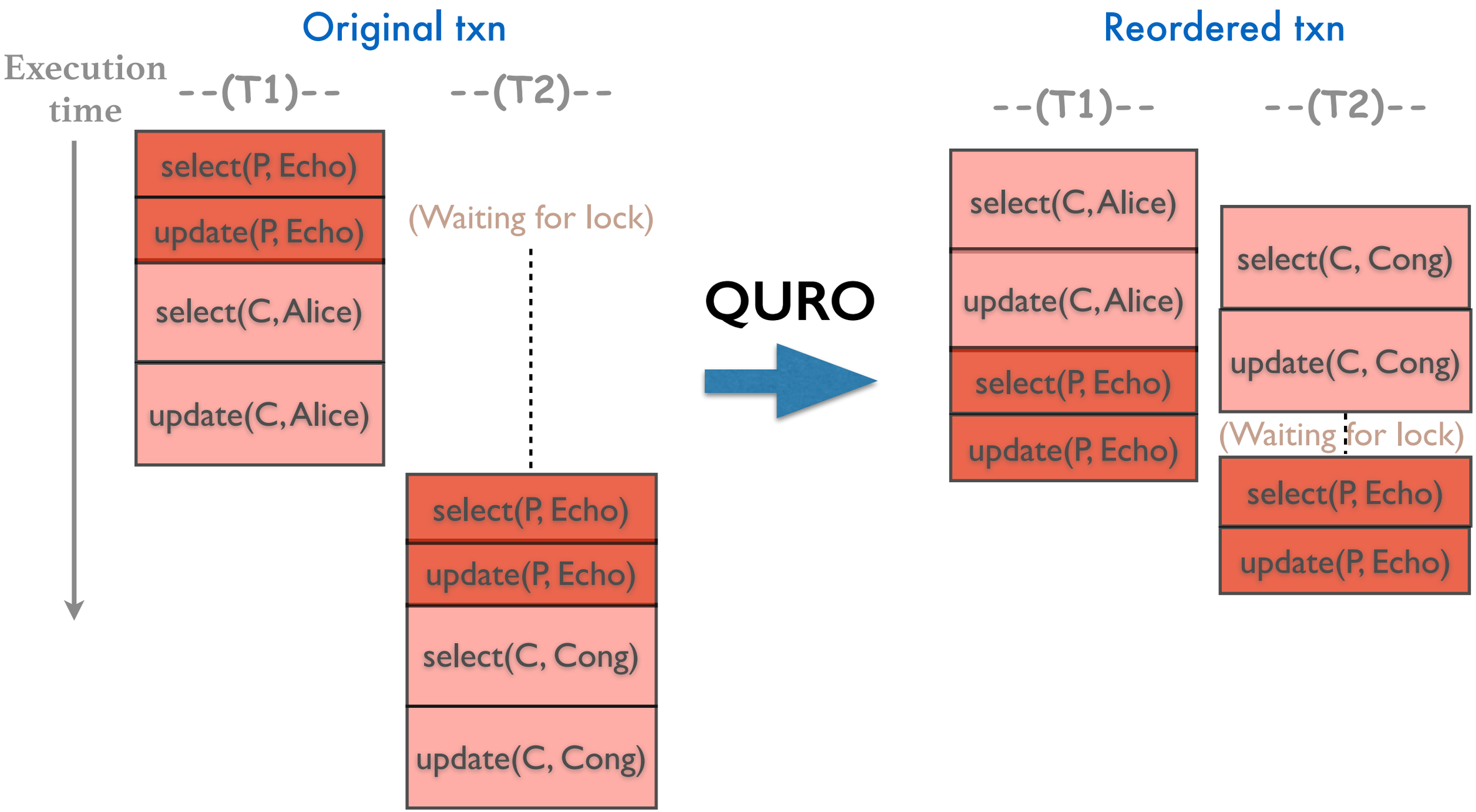


Shortening Lock Waiting



Greater performance improvement when running with more concurrent txns

Two-phase Locking



Overview

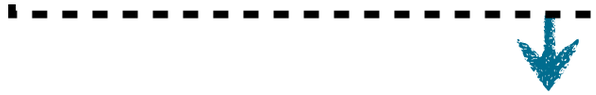
- QURO: compiler for query reordering
  - Architecture
  - Optimizations
- Evaluation
  - QURO Implementation & benchmarks
  - Compare the throughput of QURO-generated txn with:
    - Original impl. / stored procedure / other concurrency control schemes

Overview

- QURO: compiler for query reordering
  - Architecture
  - Optimizations
- Evaluation
  - QURO Implementation & benchmarks
  - Compare the throughput of QURO-generated txn with:
    - Original impl. / stored procedure / other concurrency control schemes

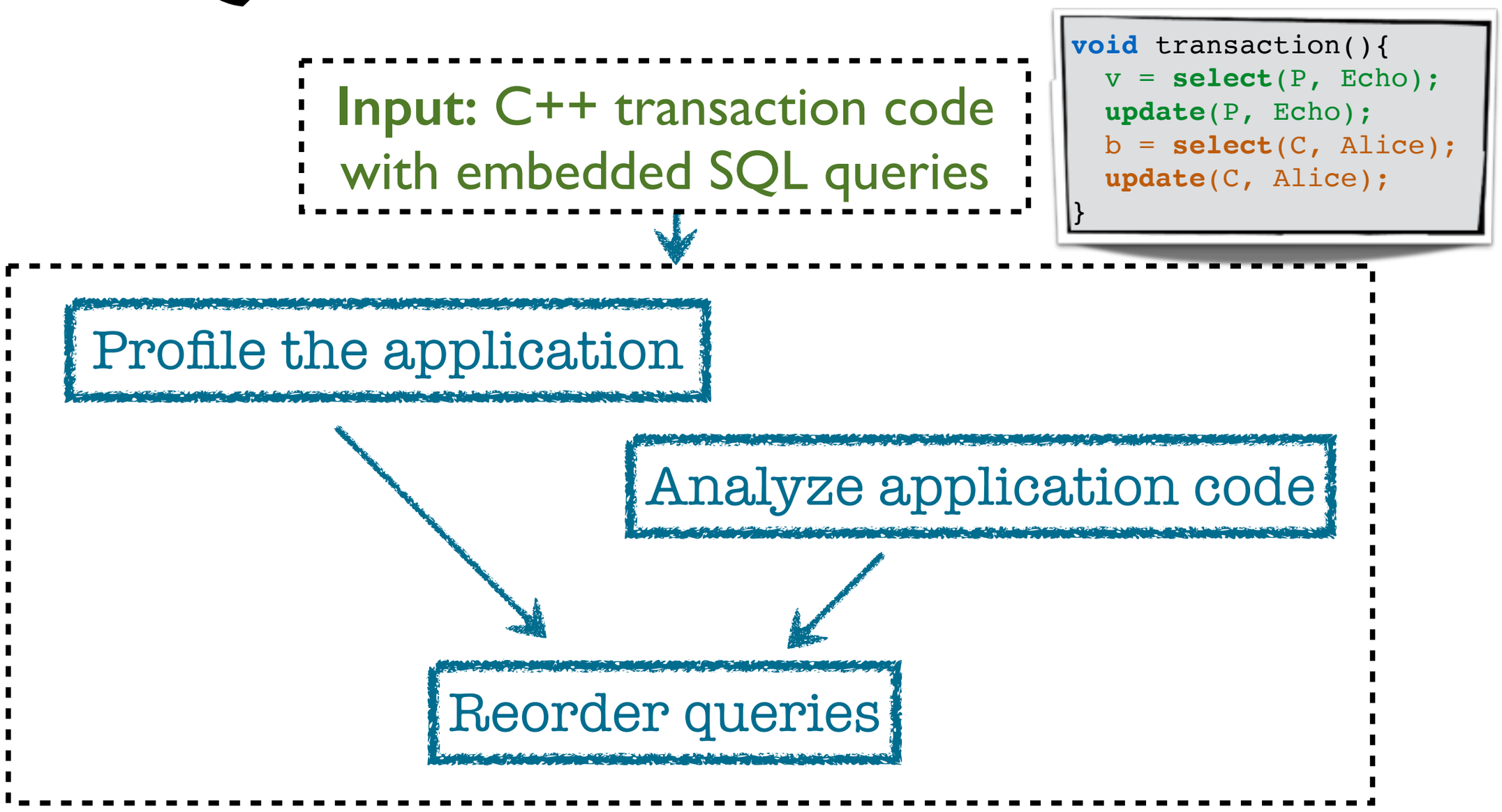
QURO Architecture

Input: C++ transaction code with embedded SQL queries

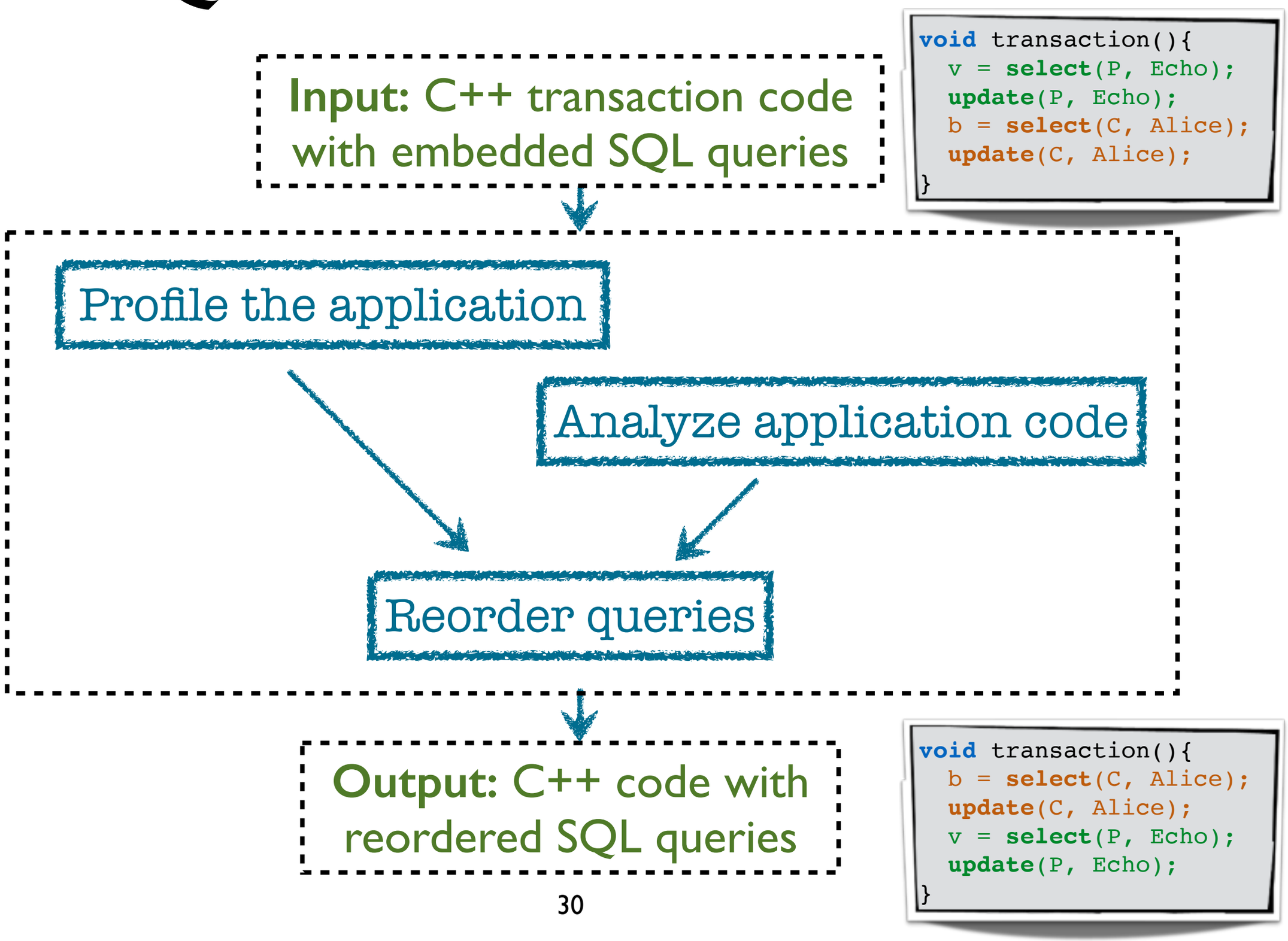


```
void transaction(){
  v = select(P, Echo);
  update(P, Echo);
  b = select(C, Alice);
  update(C, Alice);
}
```

QURO Architecture



QURO Architecture



Preserving Semantics

- Analyze the relationship among queries
  - Data dependency among program variables (RAW, WAR, WAW)

```
1. v1 = select(table1);
2. v2 = select(table2, v1);
3. update(table1, v3);
```

Statement 1 should appear before statement 2

  - Database constraints (R/W on the same table, view, foreign key...)

```
1. v1 = select(table1);
2. v2 = select(table2, v1);
3. update(table1, v3);
```

Statement 1 should appear before statement 3

ILP Formulation

position of statement i contention level of query i	p_i, $(1 \le p_i \le n)$ c_i, (const. Larger c_i, higher contention)
Constraint: data dependencies & database constraints	1. v1 = select(table1); 2. v2 = select(table2, v1); 3. update(table1, v3); $\begin{cases} p_1 < p_2, \text{variable v1} \\ p_1 < p_3, \text{table1} \end{cases}$
Goal: contentious query appear as late as possible	$\text{maximize} \sum_{i=1}^{n} p_i * c_i$
Result: query order	values of $p_1, p_2, \ldots, p_n$

ILP Formulation

- Goal: contentious queries appear as late as possible in transactions

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

```
void transaction(){
   v = select(P, Echo)
   update(P, Echo);
  b = select(C, Alice);
  update(C, Alice);
}
```

ILP Formulation

- Goal: contentious queries appear as late as possible in transactions

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

(Contention index)

$c_1 = 100$
$c_2 = 110$
$c_3 = 1$
$c_4 = 10$

```
void transaction(){
  v = select(P, Echo)
  update(P, Echo);
  b = select(C, Alice);
  update(C, Alice);
}
```

ILP Formulation

- Goal: contentious queries appear as late as possible in transactions

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

(Contention index)

$c_1 = 100$
$c_2 = 110$
$c_3 = 1$
$c_4 = 10$

```
void transaction(){
  v = select(P, Echo)
  update(P, Echo);
  b = select(C, Alice);
  update(C, Alice);
}
```

(Order constraints)

$$\begin{cases} p_1 < p_2 \\ p_3 < p_4 \end{cases}$$

ILP Formulation

- Goal: contentious queries appear as late as possible in transactions

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

(Contention index)

$c_1 = 100$
$c_2 = 110$
$c_3 = 1$
$c_4 = 10$

```
void transaction(){
  v = select(P, Echo)
  update(P, Echo);
  b = select(C, Alice);
  update(C, Alice);
}
```

(Order constraints)

$$\begin{cases} p_1 < p_2 \\ p_3 < p_4 \end{cases}$$

ILP solver

$p_3 = 1$
$p_4 = 2$
$p_1 = 3$
$p_2 = 4$

ILP Formulation

- Goal: contentious queries appear as late as possible in transactions

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

(Contention index)

$c_1 = 100$
$c_2 = 110$
$c_3 = 1$
$c_4 = 10$

```
void transaction(){
  v = select(P, Echo)
  update(P, Echo);
  b = select(C, Alice);
  update(C, Alice);
}
```

(Order constraints)

$$\begin{cases} p_1 < p_2 \\ p_3 < p_4 \end{cases}$$

ILP solver

$p_3 = 1$
$p_4 = 2$
$p_1 = 3$
$p_2 = 4$

```
void transaction(){
  b = select(C, Alice);
  update(C, Alice);
  v = select(P, Echo);
  update(P, Echo);
}
```

How to get contention index?

- Profiling the application
  - Clients repetitively issue txns
  - Calculate the **standard deviation** of query running time
  - Larger the deviation, more likely to access contentious data
  - Same machine settings, same workload (given by user)

Overview

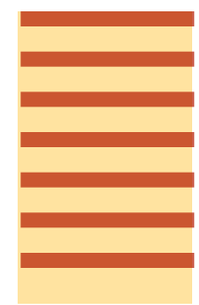
- QURO: compiler for query reordering
  - Architecture
  - Optimizations
- Evaluation
  - QURO Implementation & benchmarks
  - Compare the throughput of QURO-generated txn with:
    - Original impl. / stored procedure / other concurrency control schemes

Improving reorder results

- Allow more flexible reordering
  - Loop fission

```
for (i=1;i<n;i++){
  update(table1, v1);
  update(table2, v2);
}
```

original



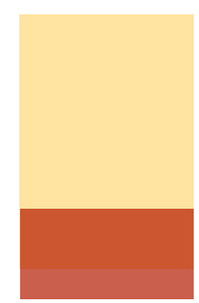
: query on contentious data

```
for (i=1;i<n;i++)
  update(table1, v1);
for (i=1;i<n;i++)
  update(table2, v2);
```

after loop fission

```
for (i=1;i<n;i++)
  update(table2, v2);
for (i=1;i<n;i++)
  update(table1, v1);
```

after reordering



  - Breaking conditional block

Reduce ILP solving time

- Using transitive closure to reduce
  - # variables
  - # constraints

	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

Overview

- QURO: compiler for query reordering
  - Architecture
  - Optimizations
- Evaluation
  - QURO Implementation & benchmarks
  - Compare the throughput of QURO-generated txn with:
    - Original impl. / stored procedure / other concurrency control schemes

Implementation

- QURO implementation:
  - Build with Clang tools
  - Use external ILP solver (gurobi)
- Experiment overview:
  - System throughput (txns/sec), latency
  - Increasing data contention
    - Smaller database
    - More clients

Evaluation

- Benchmarks:
  - TPC-C
    - 5 types of transactions
  - TPC-E
    - trade-related of transaction (update/order/result/status)
  - Bidding
- Compare with:
  - Original implementation
  - Stored procedure implementation
  - Other concurrency control protocols
- Experimental settings:
  - MySQL 5.5
  - 128 2.8GHz processors, 1056G memory

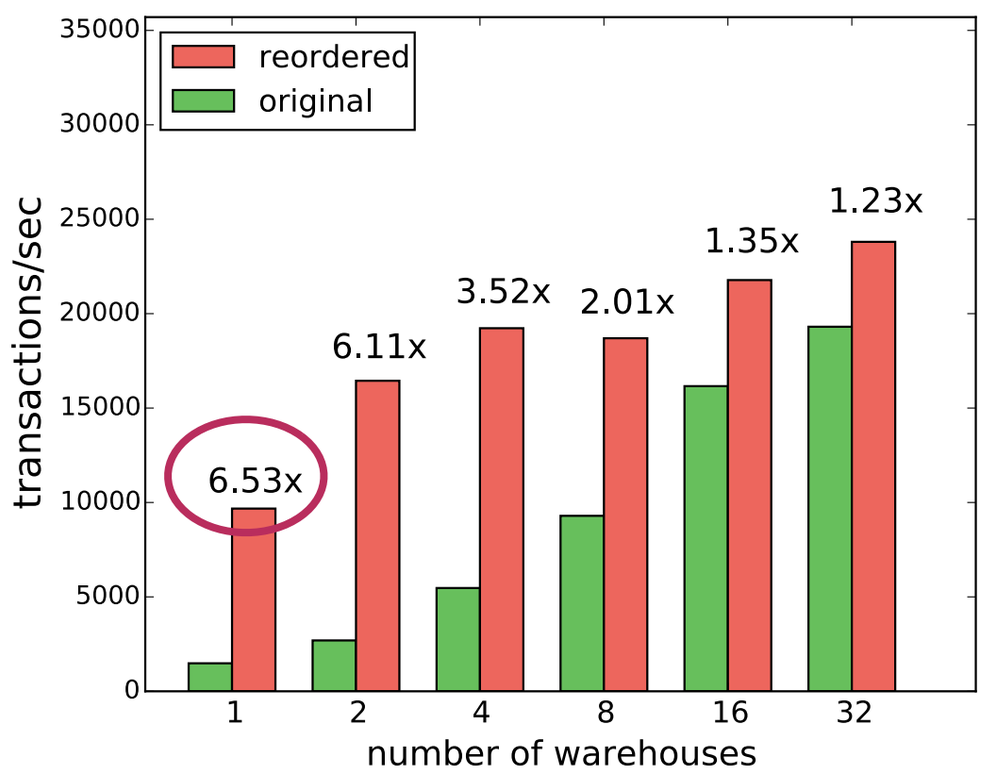
Overview

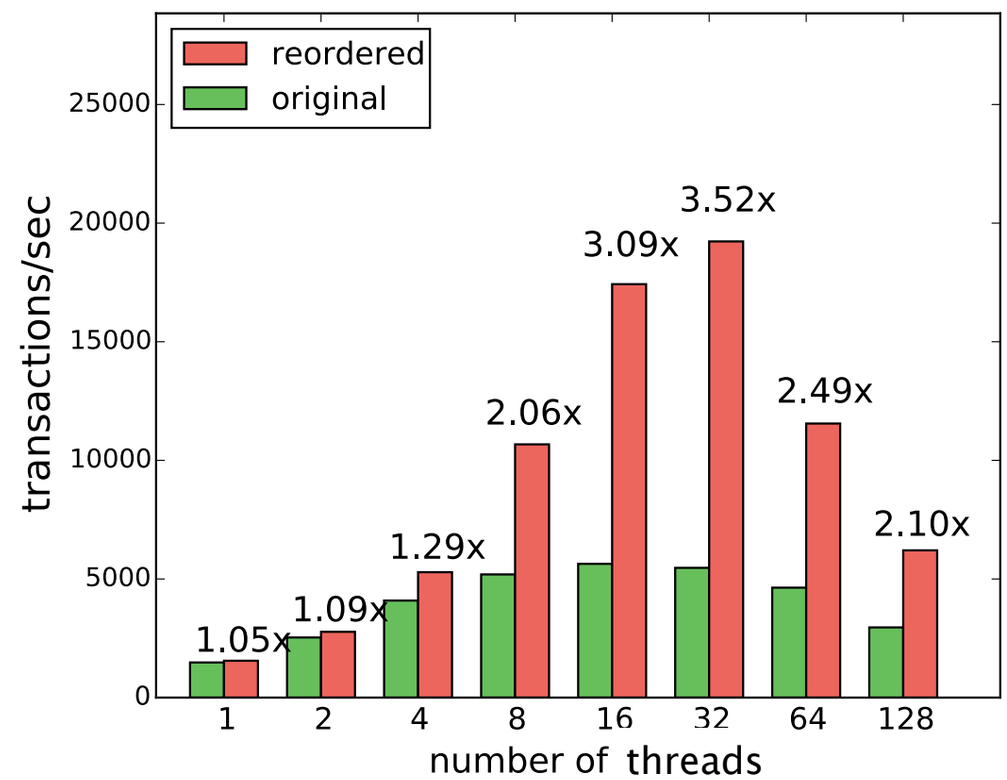
- QURO: compiler for query reordering
  - Architecture
  - Optimizations
- Evaluation
  - QURO Implementation & benchmarks
  - Compare the throughput of QURO-generated txn with:
    - Original impl. / stored procedure / other concurrency control schemes

QURO vs. Original

- Benchmark: TPC-C payment transaction
  - changing data size
  - scaling to more threads

most contentious ← → most contentious





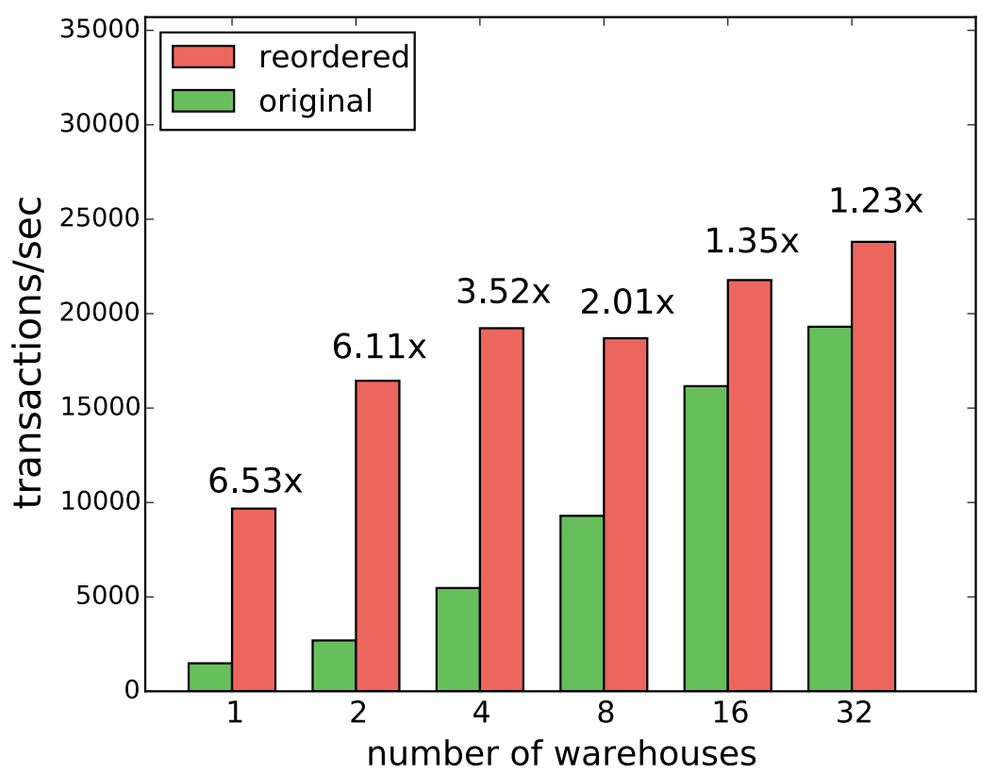
latency: decreases up to 83%

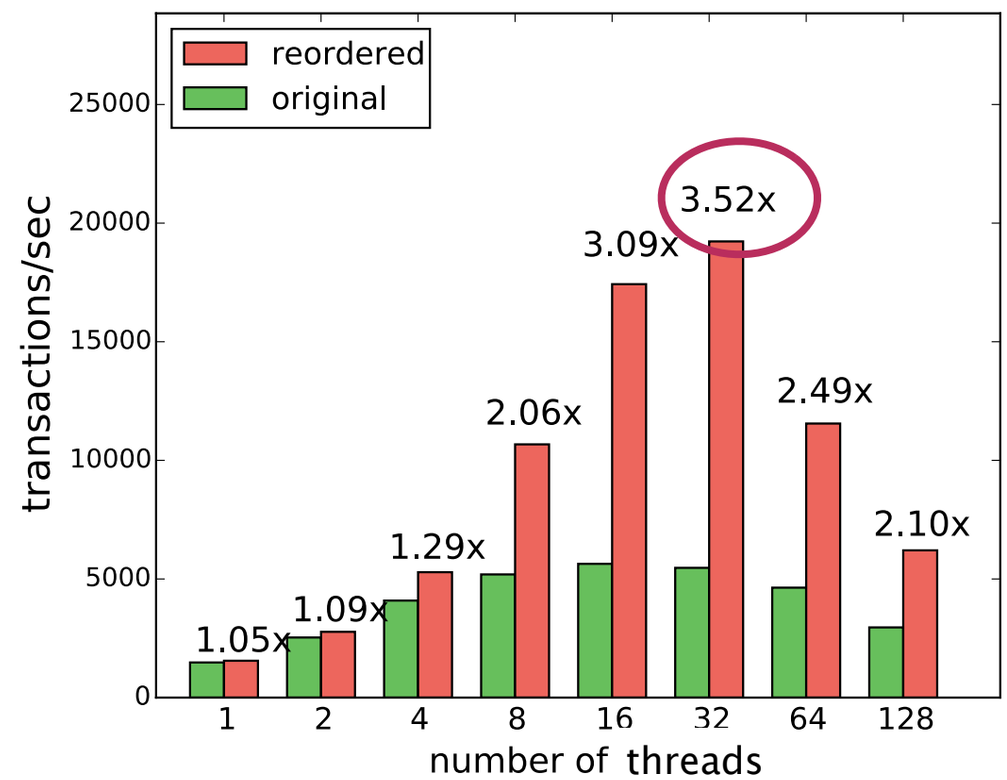
latency: decreases up to 70%

QURO vs. Original

- Benchmark: TPC-C payment transaction
  - changing data size
  - scaling to more threads

most contentious ← → most contentious





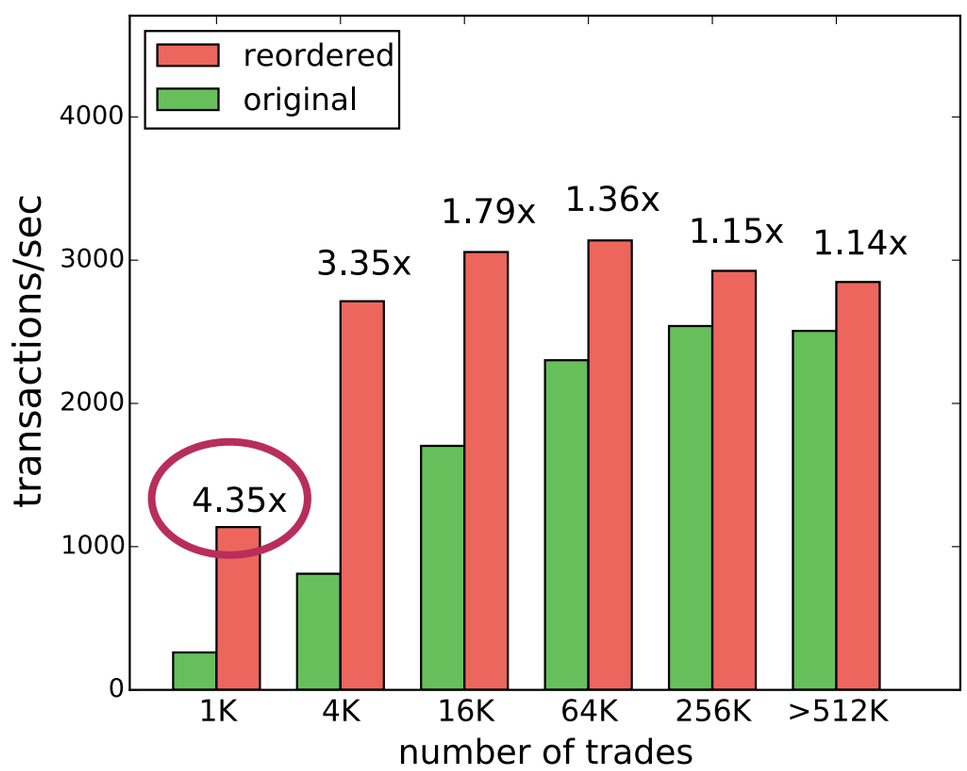
latency: decreases up to 83%

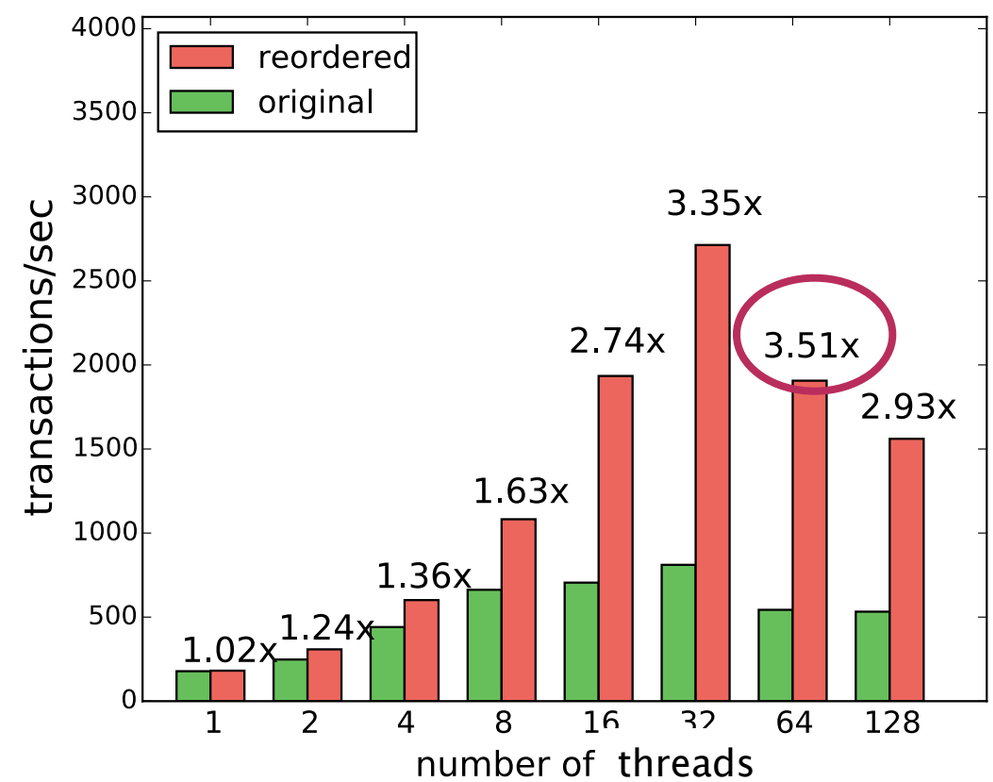
latency: decreases up to 70%

QURO vs. Original

- Benchmark: TPC-E trade update transaction
  - changing data size
  - scaling to more threads

most contentious ← → most contentious



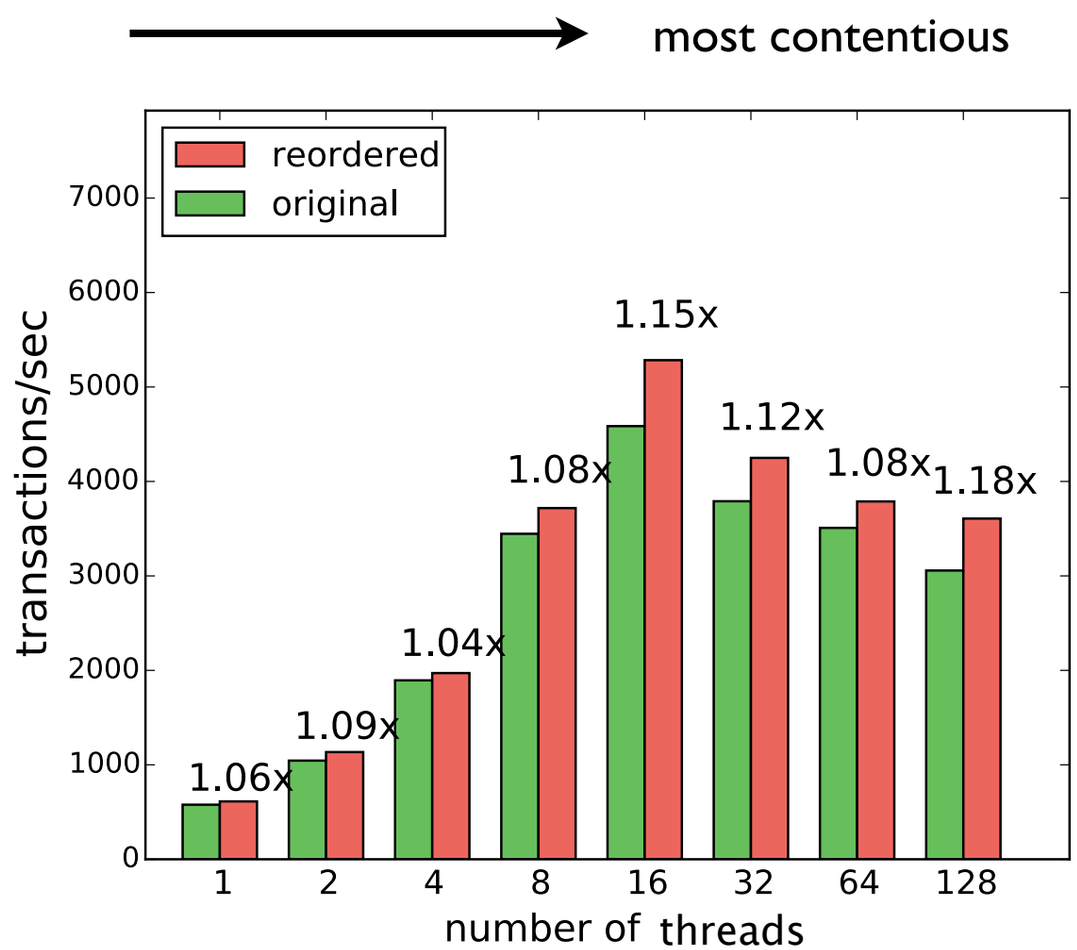


latency: decreases up to 75%

latency: decreases up to 66%

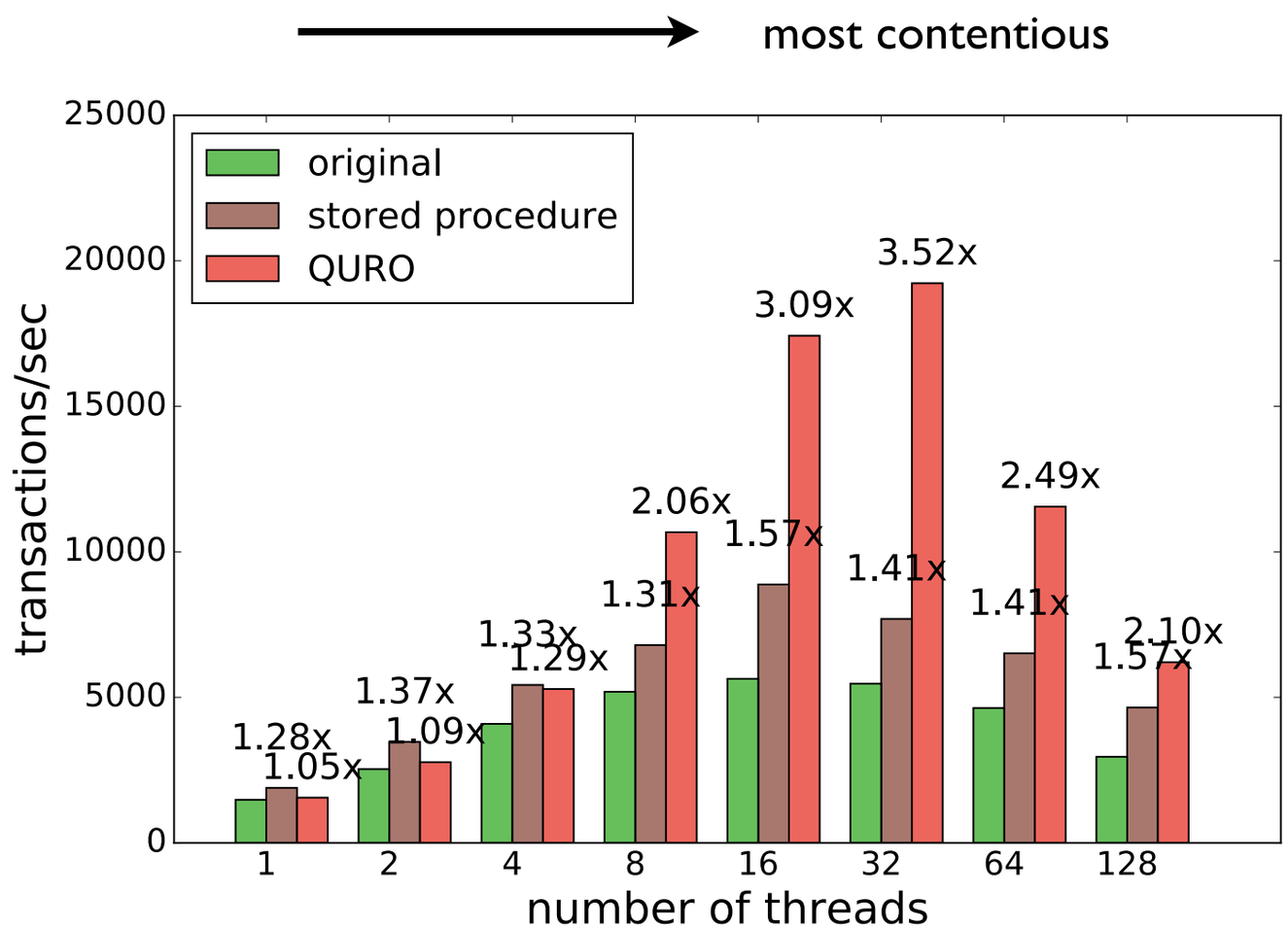
QURO vs. Original

- Mix of different transactions
  - TPC-C standard mix: 5 types of transactions



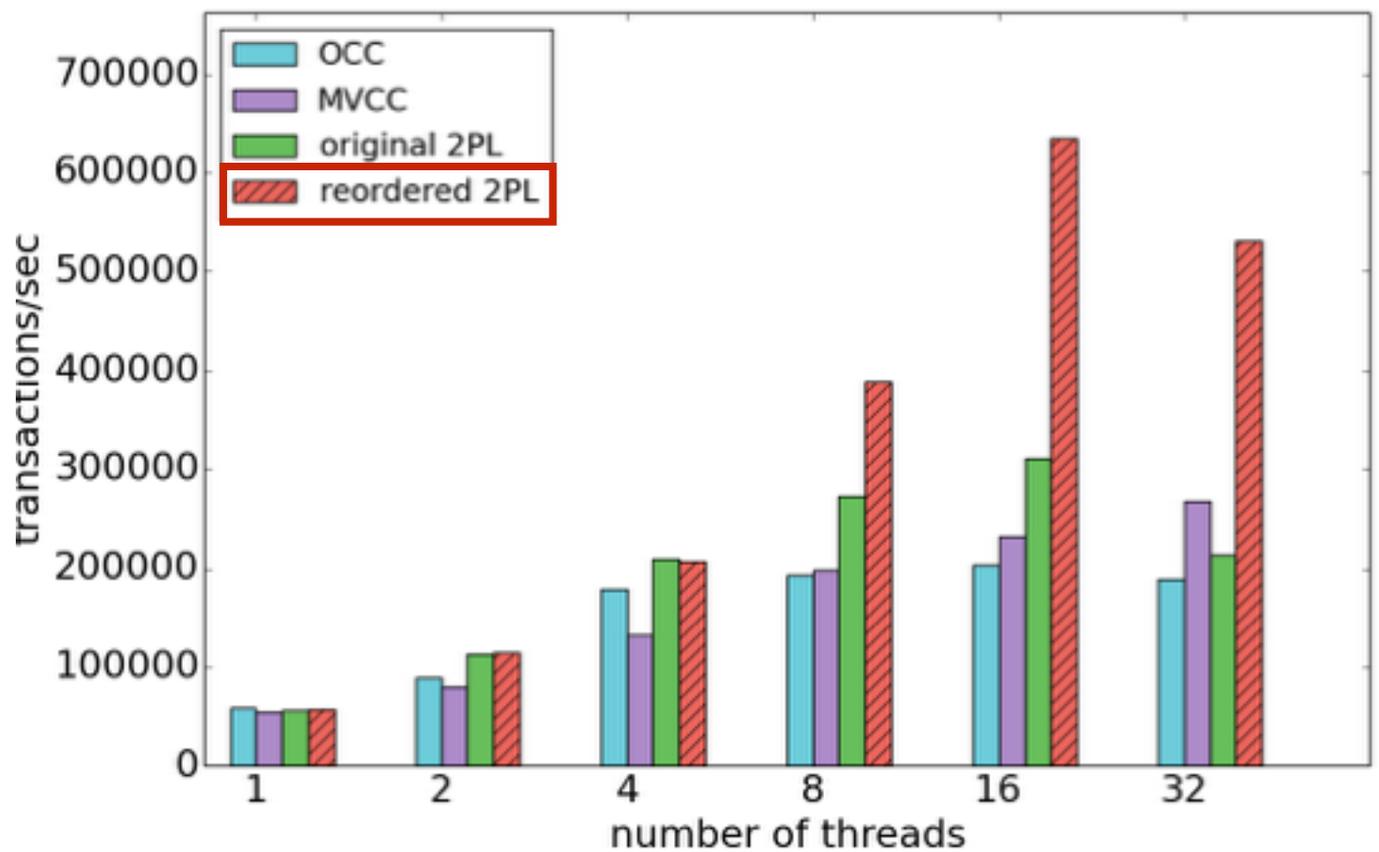
QURO vs. Stored-procedure

- TPC-C payment transaction



QURO vs. Other CC Schemes

- TPC-C a mix of payment and new order transactions



Technical report: http://homes.cs.washington.edu/~congy/reorder_techreport.pdf

Conclusion

- The order of queries has large impact on transaction performance.
- QURO leverages information about query contention, and automatically reorders the queries.
- Reordered code generated by QURO can improve throughput up to 6.53x, and can be applied to a wide range of applications.
- QURO also outperforms stored-proc and other concurrency control schemes under high data contention.

Project website: http://db.cs.washington.edu/projects/quro/

Poster session: tomorrow (Sept. 6th) @ Maple, 11:15am

backup 1

- Analysis
  - Trade-off

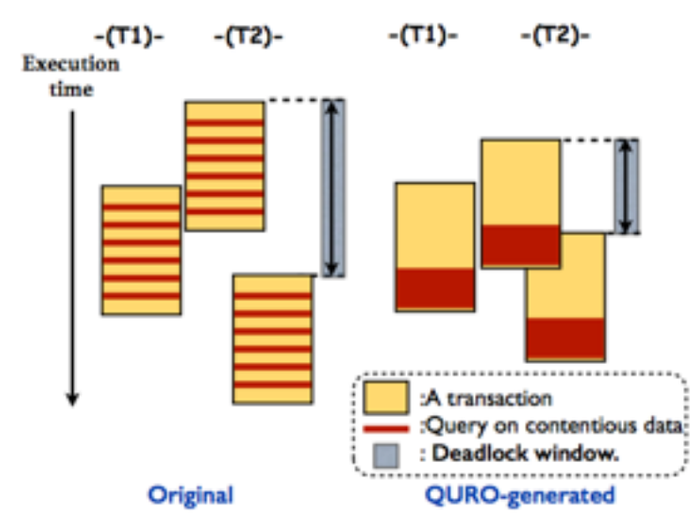
	original	reordered	single-thread	ratio
query 1	0.73	0.76	1.91	1.04
query 2	89.77	16.16	0.93	0.18
query 3	0.92	1.01	1.02	1.09
query 4	0.81	0.88	0.76	1.09
query 5	0.63	0.69	0.71	1.08
query 6	0.74	0.97	0.78	1.30
query 7	0.66	0.76	0.62	1.16
query 8	0.69	0.80	0.78	1.15
query 9	0.79	1.06	0.69	1.34
query 10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

↑ : query time increase (after reorder)
↓ : query time decrease

- Loop breaking

#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

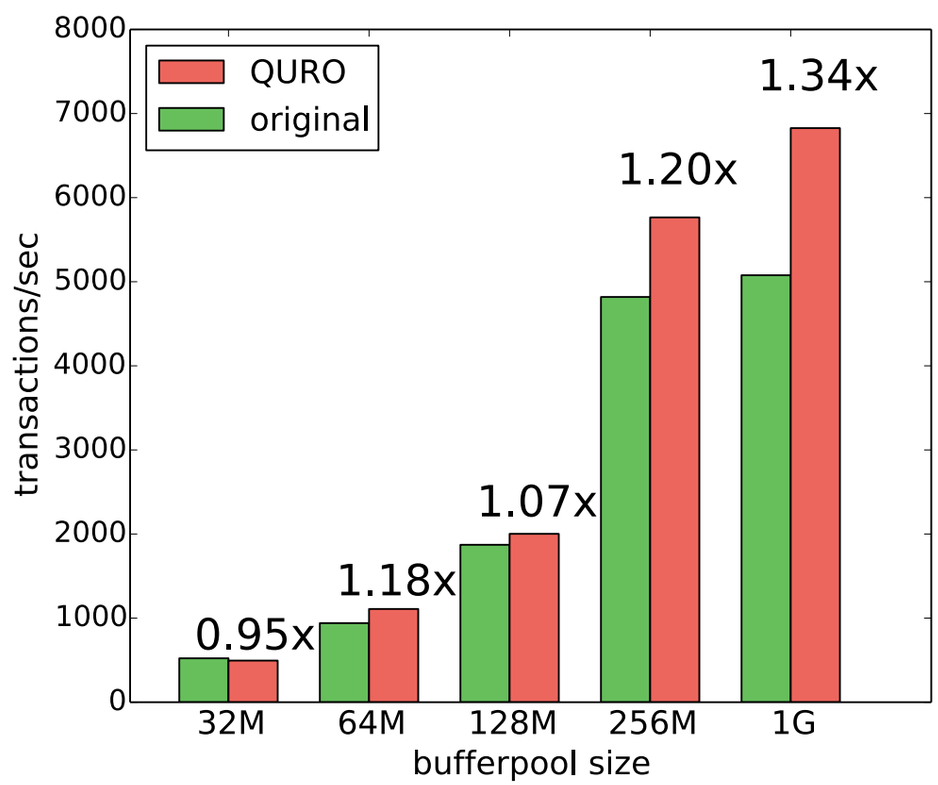
most contentious



QURO: Smaller Bufferpool

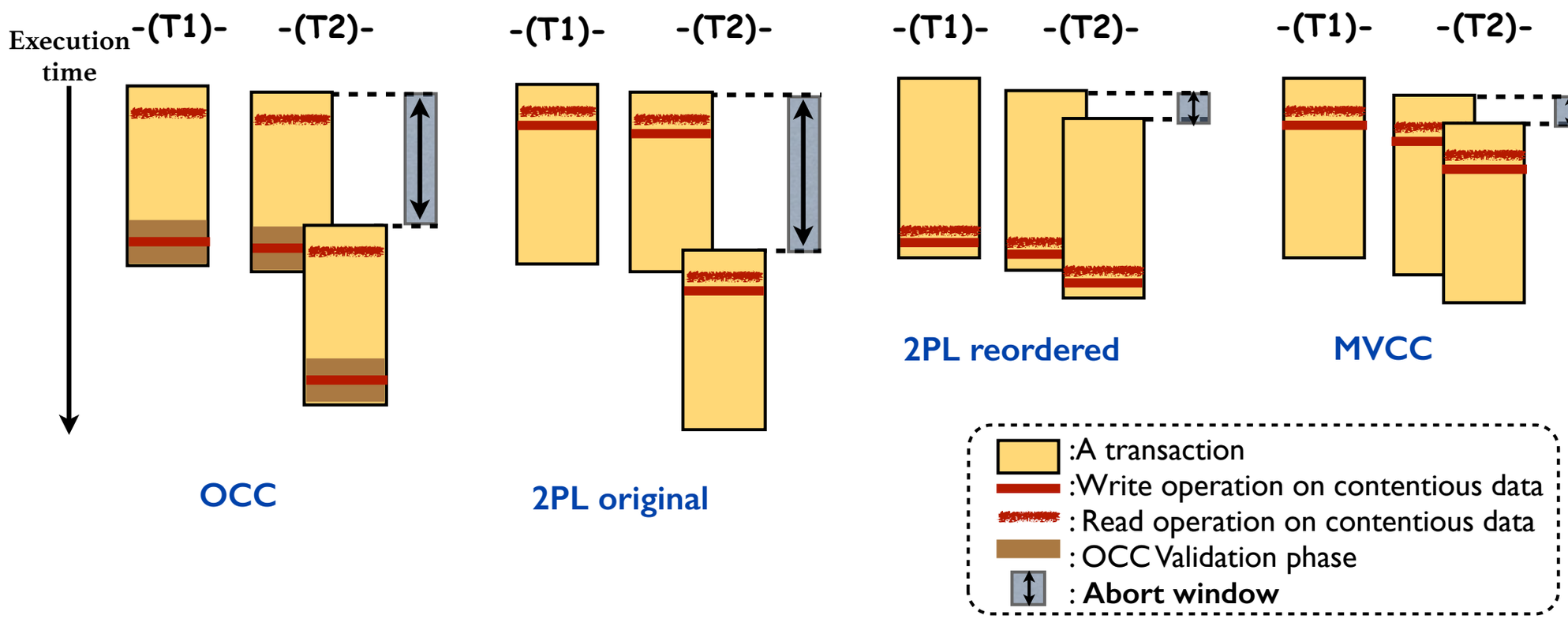
- Smaller bufferpool → increase buffer management time
  - TPC-C new order transaction

smaller bufferpool ←



backup 2

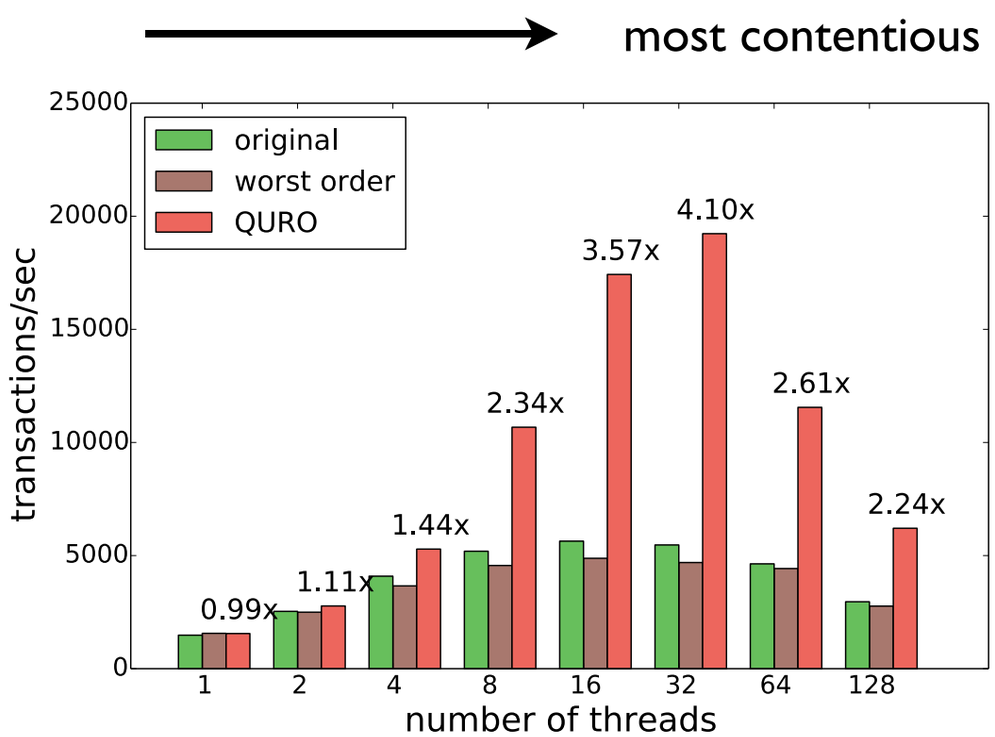
- Comparing with other concurrency control schemes



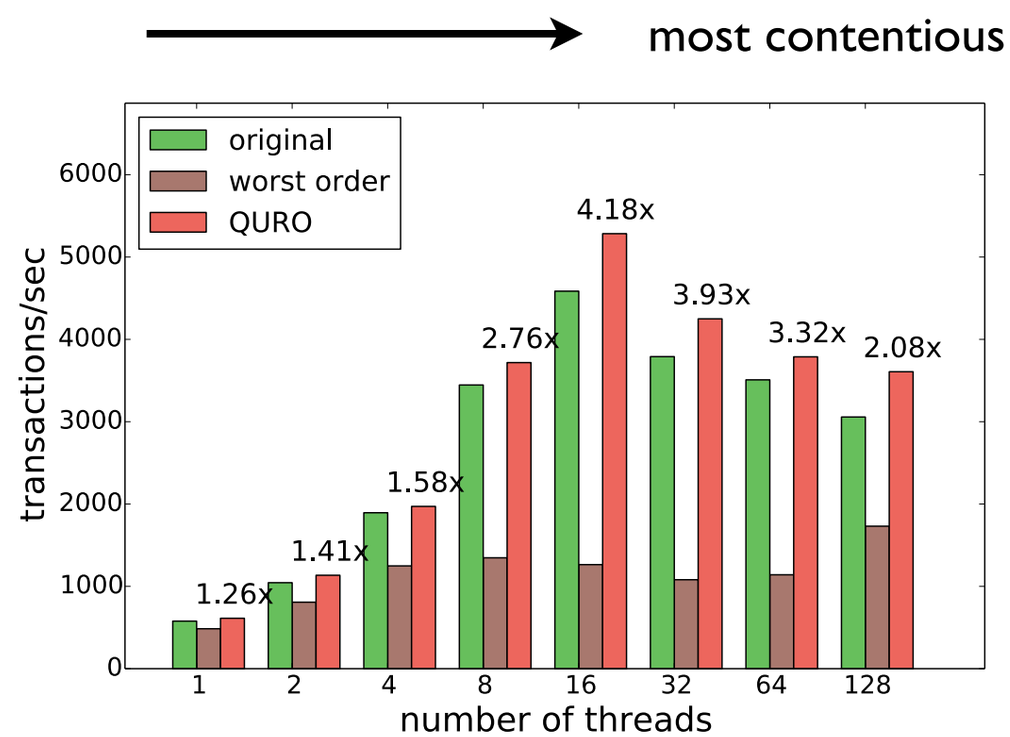
Execution of transactions T1 and T2 under different concurrency control schemes. Both T1 and T2 read and write the same tuple.

backup 3

- Comparing with worst-case ordering



TPCC payment transaction



TPCC standard mix