CATS: A Computation-Aware Transaction Processing System with Proactive Unlocking

Bolun Zhu, Yu Hua, Ziyin Long, Xue Liu*
Huazhong University of Science and Technology, *McGill University
Corresponding author: Yu Hua (csyhua@hust.edu.cn)

***Abstract*—With the increasing complexity of network applications and high demands for QoS, transaction processing systems have received more attentions due to salient features of simplicity and atomicity. Computation operations play an important role in transaction processing systems. However, conventional QoS-based mechanisms become inefficient due to the limited concurrent support upon computation operations, thus causing high time consumption in the critical path of concurrency control. In order to efficiently offer concurrent computations, we propose CATS, a Computation Aware Transaction processing System, to mitigate performance impacts caused by computation operations. CATS further leverages program semantics to defer the execution of transaction operations in the commit phase to alleviate unnecessary conflicts caused by computations. Extensive evaluation results demonstrate that CATS significantly outperforms state-of-the-art designs, including 2PL and OCC based transaction processing systems on high-contended and computation-intensive workloads. We have released the open-source codes in GitHub for public use.**

I. Introduction

QoS-based transaction processing systems via high-speed networks become the important infrastructure of distributed systems that are able to support many real-world applications, such as cloud computing [1], blockchain [2], [3], data analytics [4] and social networks [5]. In practice, due to the existence of network communication delays, data changes may not be immediately observed by processes. When multiple processes concurrently access data, the data versions may be inconsistent, which exacerbates the QoS in the distributed systems. To mitigate or even avoid such inconsistency, transaction processing systems are widely used in many distributed systems due to their high efficiency and ease of use.

A transaction (txn) is a powerful abstraction to guarantee that a sequence of operations is performed atomically. The operations within a transaction follow the "all-or-nothing" semantic, which are either executed all at once or not. The states of partially executed transactions are not allowed. By using transactions, the correctness of applications can be guaranteed due to the atomic property [6], [7]. Applications only need to annotate their operations as transactions, and put them into a transaction processing system, without the need of handling concurrency control. The correct parallel executions of transactions are guaranteed by the underlying concurrency control mechanism of transaction processing systems. For example, MySQL-InnoDB [8] and Postgres [9] use two phase locking (2PL) to serialize transactions.

979-8-3503-9973-8/23/$31.00 ©2023 IEEE

Traditional transaction processing systems are designed for simplicity, in which each transaction only contains several read and write operations [10]. Most optimized designs focus on read and write operations [11] since computations become a small portion of transaction operations. However, when the complexity of transactional applications (e.g. HTAP [4]) increases, the system performance becomes heavily impacted by intensive computations. For example, the real-time analytical workloads decrease transactional throughput by 89% on TiDB [11], a state-of-the-art HTAP system. The decreased performance comes from the observations that more tasks have to be completed by limited computation resources, and the concurrency control mechanism becomes inefficient.

Conventional concurrency control mechanisms, such as two-phase-locking [12] and optimistic concurrency control (OCC) [13], fail to meet the needs that transactions perform intensive computations on source data. Prior intensive computations, such as data analysis, data compression, and data encryption/decryption, were considered to be read-only and can usually be performed outside the transaction without concurrency control [14], [15]. However, applications making decisions on real-time analysis may be both contended and compute-intensive [4]. In the conventional 2PL, executing these contended and compute-intensive transactions significantly blocks other transactions, since computations lengthen the duration of locks. Moreover, the scenario of OCC becomes worse and generally considered to be unsuitable for high contended workloads due to more aborts and wasted resources [16]. Hence, we need to rethink the design of concurrency control mechanisms especially when for computation-intensive applications.

In order to address this problem and improve QoS, an intuitive approach is to unlock some data before performing time-consuming computations, called *Proactive Unlocking*. For example, before carrying out the partial computational operations in 2PL, some data can be unlocked earlier by moving some unlock operations to a point immediately after the transaction can commit. However, this approach cannot correctly work in case of system crashes. We assume that txn T has already released some locks and still contains some un-executed tasks. Once aborted, T cannot roll back to an uncommitted state since other txns may read T's unlocked data and consider that T has committed (also known as read-uncommitted).Hence, strict 2PL [12] requires that locks need to be held until the end of a transaction.

In this paper, we aim to explore the design space of proactive unlocking and pay special attention to the impact of compute operations within transactions. Our observation is that some operations can not be placed after unlocking. We call these operations "critical" whereas others are called "uncritical". The results of the critical operations are used by lock operations to protect correct addresses. Therefore, the critical operations must precede the lock operations. Moving critical operations behind some unlock operations causes acquiring locks after releasing locks, which violates 2PL.

In order to avoid unnecessary blocking caused by computational operations without violating 2PL, we propose CATS, a Computation Aware Transaction Processing System, which includes three components: (1) To classify critical and uncritical operations, we leverage a critical-operation identification algorithm via the static data dependency analysis. (2) To address the inaccuracy of static analysis, we actively overlook the memory dependency in the dependency analysis and propose a dependency-conflict avoidance mechanism to detect the memory dependency at runtime. (3) To speed up transactions' uncritical operations, we use the computation-aware concurrency control protocol. By moving some unlock operations before uncritical operations, uncritical computational operations can be removed from the critical path of contended transactions.

We have released the open-source codes in GitHub for public use at https://github.com/BolunZhu/CATS.

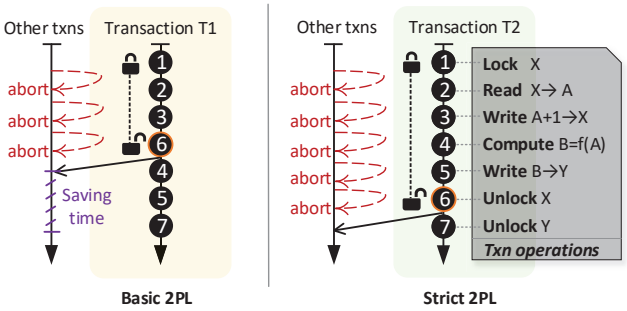


Fig. 1: Benefits of basic 2PL compared with strict 2PL.

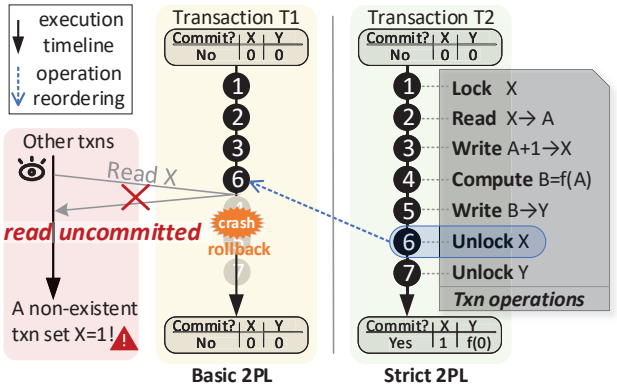


Fig. 2: Violating strict 2PL incurs the read uncommitted issue.

II. Background and Motivation

In this paper, we focus on transaction processing systems and explore how to speed up these systems on highly contentious and computation-intensive workloads. For convenience, we first describe a baseline system using two-phase locking as its concurrency control protocol.

Baseline System: In the baseline system, the application interacts with the system through basic transaction interfaces, including $TX\_begin()$, $TX\_end()$, $TX\_read()$ and $TX\_write()$. The transaction process includes two phases, i.e., the execution phase and the commit phase. In the execution phase, operations in the transaction logic (including read, write, and compute operations) are executed in the program order. During the execution phase, transactions acquire locks before each read and write operation. If the locking is successful, the address of data will be recorded in the transaction's read-write set. Otherwise, the transaction (acquirer) introduces the conflicts with the other transaction (owner). To resolve the conflict, one txn needs to be aborted, according to the deadlock-prevention mechanism [16]. The aborted txn will release all locks of data in the read-write set. If not aborting but completing all operations, this txn is considered ready to commit and will enter the commit phase. During the commit phase, the txn flushes the logs to the persistent media, then writes data back and releases locks.

Two-Phase Locking: Two-Phase locking (2PL) [12] is a widely used concurrency control protocol due to ease of use. 2PL requires transaction execution to be separated into two phases, i.e., the growing phase and the shrinking phase. In the growing phase, each transaction continuously acquires locks as needed. In the shrinking phase, the transaction begins to release the locks, and no locks can be acquired in this phase. We choose 2PL as a baseline concurrency control protocol because 2PL is more suitable for highly contended workloads and exploits the weak assumption that *a transaction is concurrently executed with an unknown set of other transactions*.

Strict 2PL vs. Basic 2PL: According to the requirements for releasing locks, 2PL can be further divided into strict 2PL and basic 2PL. The strict 2PL requires that the lock is not released until the end of the transaction, while the basic 2PL does not. *The advantage of basic 2PL* is the relaxed execution order, which allows earlier unlocking. As shown in Fig 1, T1 immediately unlocks data X after performing the 3rd operation. While its strict 2PL version (T2) holds the lock until the 5th operation, leading to a longer duration of blocking.

The disadvantage of the basic 2PL is that when considering crash and recovery, it incurs the read uncommitted issue. As illustrated in Fig 2, the read uncommitted issue occurs when (1) the system crashes right after transaction T1 unlocking the data X and (2) other transactions have already read the unlocked data X. In this scenario, the modification of X is not committed but observed by other transactions, which breaks the *semantics of transactions*— uncommitted data are not allowed to be observed.

To relax the requirements of strict 2PL and avoid the read uncommitted issue, many different concurrency control

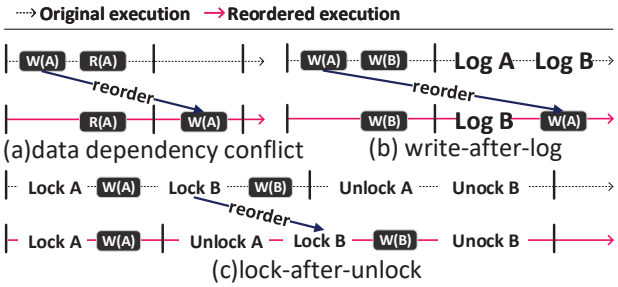


Fig. 3: Typical examples of false reordering.

mechanisms are proposed. Transaction chopping [17] and its improvements [18], [19] permit uncommitted data to be read without concurrent conflicts by performing program analysis of the workload before execution, which relaxes the assumption of 2PL (there is an *unknown* set of concurrent transactions). Instead, Bamboo [20] speculatively releases locks as early as possible and fixes the read uncommitted issue with cascading aborts at runtime. They all need global knowledge of the workload, whether before executions (program analysis in Transaction chopping) or at runtime (dependency graph in Bamboo), which significantly limits their applicability.

Our CATS Approach: In contrast to these approaches, we seek a computation-aware transaction processing system that can release locks as early as possible and avoid the read uncommitted issue. To this end, we rely on ***deferred execution***–delaying some transaction operations to the commit phase when these operations are time-consuming. Prior work [21] proposed a similar idea called *lazy evaluation* in deterministic databases, which requires a pre-defined total order of transactions. DRP [22] uses deferred execution to avoid the rank mismatch problem when using the runtime pipelining technique without global prior knowledge of concurrent transactions. Though the benefits of deferred execution have been discussed in existing works, there remain some unsolved challenges, preventing deferred execution from being deployed in general transaction processing systems.

Challenge 1: Data dependency conflicts. If the deferred execution breaks the original data dependency of the transaction as shown in Fig 3(a), the execution results of transactions will not be the same as the baseline system. Therefore, the data dependency must be maintained during reordering.

Challenge 2: Write-after-log. As shown in Fig 3(b), data writes after logging cannot be recovered after the system crashes, thus violating durability. Because some values to log come from the results of deferred execution, we cannot log them in advance. To address this problem, a computation-aware transaction processing system needs to re-compute the results of the deferred execution.

Challenge 3: Lock-after-unlock. When a deferred execution accesses the data that has not been accessed in the execution phase, the transaction must acquire the lock. However, acquiring locks after releasing some locks violates 2PL. As shown in Fig 3(c), before acquiring the lock B, the writes to A are able to be read by other transactions, leading to the read

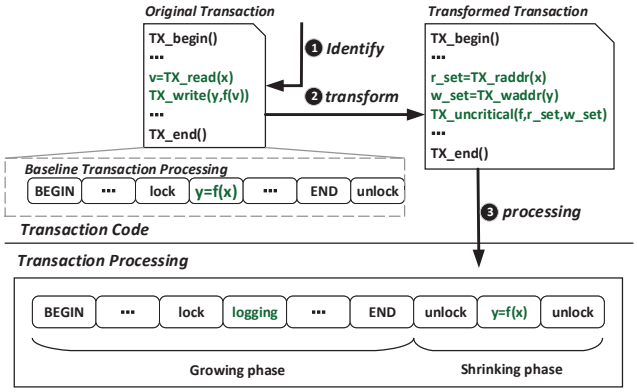


Fig. 4: System Overview.

uncommitted issue. To resolve the read uncommitted issue, the system cannot release the locks before the commit phase.

In summary, to be computation-aware, transaction processing systems need to maintain the original data dependency (C1), re-compute the results of the deferred execution (C2) and acquire the lock only in the execution phase(C3). CATS achieves these requirements by using the following schemes. (1) We detect data dependency conflicts before execution via the critical-operation identification algorithm and dependency-conflict avoidance mechanism at runtime. (2) We pack multiple operations into a function, lock the function's dependent data and record the functions and read-write sets in logs. During recovery, the system can use the functions to re-compute the results and write them to the persisted media. (3) We select only uncritical operations for deferred execution. The accessed data of uncritical operations are deterministic. Hence, we acquire locks in the execution phase and complete uncritical operations in the commit phase.

III. System Design

A. Design Overview

CATS aims to shorten the growing phase in 2PL so that some data can be unlocked earlier. Our approach moves the execution of some instructions from the growing phase to the shrinking phase, e.g. the execution of $y = f(x)$ in Fig 4.

We build CATS as an extension of the transaction processing system that uses two-phase-locking for concurrency control, which we denote as the baseline system. Transactions in the baseline system can perform three types of operations: read, write, and computation, according to their behaviors toward the data protected by the system. Read/write within transactions is called *transaction read/write*.

Initially, we analyze the transaction code used in the baseline system via the critical-operation identification algorithm as described in Section III-B1, which identifies critical and uncritical instructions within the transaction. Critical instructions are executed as normal while the executions of uncritical instructions will be deferred. We explain the details of critical instructions in Section III-B1.

Moreover, the original transaction is transformed to a new version with annotations for uncritical instructions. The execution of annotated instructions will be moved from the growing

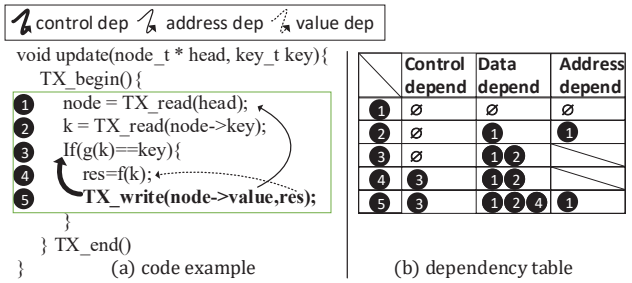


Fig. 5: An example of a dynamic transaction with the second address indexing and conditional update.

phase to the shrinking phase. In the transformation, each continuous sequence of uncritical instructions will be extracted into a new function, which we refer to as an uncritical function. We annotate the function and its read-write set using three newly added interface functions, $TX\_uncritical$, TX_raddr, and TX_waddr. Specifically, $TX\_uncritical$ receives a function and its read-write set as parameters and logs them for the following execution. TX_raddr and TX_waddr are used to pass the addresses to $TX\_uncritical$ as a read-write set.

After that, transactions are processed in the execution phase and the commit phase: in the execution phase, the system executes critical instructions in the program order and skips uncritical functions by using locking and logging; in the commit phase, the current transaction needs to release locks that are not used in any read-write sets of uncritical functions, and then execute the skipped uncritical functions in order. Once an uncritical function completes, we check if it is safe to release locks in the read-write set. Note that only when the transaction has already obtained all locks as needed, which indicates it is ready to commit, can it enter the commit phase. Otherwise, the current transaction needs to be aborted.

B. Identify critical instructions

We first introduce the concept of critical instructions and then present an algorithm to identify critical instructions.

1) Critical Instructions: The concept of critical instructions comes from our observation that: only partial instructions of a transaction are necessary to determine whether the transaction can commit. Specifically, to be committed, a transaction must not conflict with other transactions. To guarantee this, a transaction needs to calculate its read-write set in the execution phase and acquire locks according to the read-write set in the commit phase. To calculate the read-write set of a transaction, traditional approaches execute the whole transaction, while we observe that executing a subset of instructions within the transaction, which we called critical instructions, is enough.

For static transactions whose read-write sets are known in advance, there is no critical instruction. However, for dynamic transactions, some instructions may be critical, such as the second address indexing, whose operand cannot be known before execution.

Traditional transaction processing systems regard all instructions as critical ones and execute the whole transaction to confirm its read-write set since they are unable to identify critical instructions. Only when we can identify critical instructions, can the system release locks earlier.

To efficiently identify critical instructions, we consider the read-write set of the transaction as shown in Fig 5(a). The Transaction in Fig 5(a) conditionally updates a node according to its computation results of $g(k)$ and $f(k)$. $g(k)$ and $f(k)$ are pure functions that only contain arithmetic instructions and never access global data. Before executions, the read set contains $head$ and $node\text{->}key$, but the address of $node\text{->}key$ is unknown, which is the execution result of Line 1 and depends on the states of the system. It is difficult to confirm whether $node\text{->}value$ exists in the write set since TX_write (Line 5) is control dependent on Line 3, let alone what the address of $node\text{->}value$ is.

The lesson we learned from this example is that there are two types of dependencies that determine the read-write set of a transaction. Specifically, the first dependency is the control dependency of read and write in a transaction, which determines whether the accessed address exists in the read-write set. The second one is the data dependency of the addresses of read and write operations in a transaction, which determines the actual address in the read-write set. For simplicity, we refer to the second type of dependency as the address dependency.

While traditional data and control dependencies have been well studied in existing works [23], the address dependency, as a special subset of data dependency, has gained little attention since it is not only related to program semantics, but also concurrency control.

To distinguish address dependency from data dependency, we define two relationships for transaction read and write:
Address dependency: For a transaction read or write i, if *the memory address* comes from a variable that is defined by another instruction j, we call i *is address dependent on* j.
Value dependency: For a transaction read or write i, if *the value* for read or write comes from a variable that is defined by another instruction j, we call i *is value dependent on* j.

As shown in Fig 5(a), the transaction write in Line 5 uses[1] two variables $node\text{->}value$ (defined in Line 1) and res (defined in Line 4). According to our definitions, the Line 5 is address dependent on Line 1 and value dependent on Line 4. The execution of Line 1 (address dependency) determines the write set while line 4 (value dependency) only determines the actual value to write. Therefore, Line 1 is a critical instruction, while Line 4 is not.

The instructions that are address or control dependent by transaction read and write are critical instructions. We refer to these instructions as *directly critical instructions*, to distinguish them from indirectly critical instructions. *Indirectly critical instructions* are those instructions that are data or control dependent by directly critical instructions. Though indirectly critical instructions may not directly determine the read-write set, they are necessary for the correct execution of

[1]The use and $define$ come from compiler community [24]. Use-def chains created by dataflow analysis are widely used for many compiler optimizations, such as constant propagation and common subexpression elimination [25].

directly critical instructions. We further present the definition of critical instructions.

Definition 1: *Instruction i is directly critical if there exists a transaction read or write j satisfying that j is control dependent on i or address dependent on i.*

Definition 2: *Instruction i is indirectly critical if there exists a critical instruction j satisfying that j is control dependent on i or data dependent on i.*

Instruction i is critical if i is a directly critical instruction or indirectly critical instruction.

2) Critical-operation Identification Algorithm:

We now present our algorithm to illustrate the critical instructions based on our previous definitions. As shown in Algorithm 1, we receive S_{all} (the set of instructions in a transaction) as the input, which returns $S_{critical}$ (the set of critical instructions) and $S_{uncritical}$ (the set of uncritical instructions) as the output. Specifically, this algorithm is executed in three steps: First, S_{rw} is obtained by finding all transaction reads and writes from S_{all}; Second, S_{direct} is obtained by filtering instructions from S_{all}, which are address or control dependent on instructions in S_{rw}; Third, $S_{indirect}$ is obtained by selecting the instructions from S_{all} that are data or control dependent on instructions in S_{direct}. The $S_{critical}$ is the combination of S_{direct} and $S_{indirect}$, and the other instructions (other than $S_{critical}$) are $S_{uncritical}$.

Algorithm 1: Identify Critical Instructions

```
   Input: S_all = the set of instructions in a transaction
   Output: S_critical, S_uncritical
   // Step 1: get the set of read or write instructions
1  for instruction i in S_all do
2      if i is a transaction read or write then
3          S_rw.insert(i)
   // Step 2: obtain the set of directly critical instructions
4  for instruction i in S_rw do
5      for variable v that i uses do
6          if v is used as memory address then
7              instruction j = v.define()
                   // i is address dependent on j
8              S_direct.insert(j)
9      for instruction j in S_all do
10         if i is control dependent on j then
11             S_direct.insert(j)
   // Step 3: obtain the set of indirectly critical instructions
12 for instruction i in S_direct do
13     for instruction j in S_all do
14         if i is control dependent or data dependent on j then
15             S_indirect.insert(j)
16 S_critical = S_direct ∪ S_indirect
17 S_uncritical = S_all \ S_indirect
18 return S_critical, S_uncritical
```

The evaluation results in Fig 5 are shown in Table I. $S_{critical} = \{1, 2, 3\}$ indicates that the transaction can determine the read-write set after executing instructions in Lines 1, 2, and 3. We hence unlock the data $head$ and $node\text{->}key$ in this case, after executing critical instructions in Lines 1 to 3.

Our algorithm exploits static data and control dependency tracking. However, in practice, static dependency tracking is inaccurate due to variable addresses, loops, etc. In the static data dependency analysis, different variables might refer to the same address, which is called *memory dependency*. Currently, there is no practical algorithm that can calculate the memory dependency accurately. Thus, the result of our algorithm using inaccurate memory dependency is also inaccurate. Since it is hard to accurately identify critical instructions, we allow some critical instructions to be regarded to be uncritical, and provide a dependency-conflict avoidance mechanism to fix.

TABLE I: Execution results of Algorithm 1.

S_{rw}	S_{direct}	$S_{indirect}$	$S_{critical}$	$S_{uncritical}$
{1,2,5}	{1,3}	{1,2}	{1,2,3}	{4,5}

Specifically, our algorithm overlooks the uses that have multiple definitions in Line 9, thus avoiding some complex data dependency tracks. The cost is that some data dependency conflicts may occur. As some critical instructions may be considered to be uncritical, and their executions will be wrongly deferred, which will be discussed in Section III-D.

C. Transform transactions

Identifying Uncritical Sequences. After identifying critical instructions, the instructions within a transaction are separated into several sequences of critical and uncritical instructions, called critical and uncritical sequences. In principle, each uncritical sequence can be annotated to defer its execution. However, eagerly deferring all uncritical sequences may degrade system performance. This is because not all uncritical sequences can benefit from this design. Since deferring uncritical sequences adds some overheads, such as logging and function calls, only those uncritical sequences whose execution time is longer than that of logging its read-write set can bring performance improvements.

Since statically computing the execution time of a sequence of instructions is hard, our current implementation requires manual efforts to decide which uncritical sequences to be annotated, which has been widely used in the community [20], [26]. In our real implementations, most uncritical sequences are the last writes of a memory address and computations for values. The uncritical sequences that contain expensive computations are more suitable to be annotated.

Annotating Uncritical Sequences. We use function-level annotations as the basic unit of uncritical sequences. This is because adjacent instructions usually have dependencies, and putting them in the same function can reduce annotation overheads. When a sequence of uncritical instructions is selected, it is extracted into a new wrapper function, which receives a read set and a write set as parameters and returns void. Each wrapper function has the same type, which avoids passing different types of function pointers as parameters.

To annotate an uncritical sequence, we need to calculate its read-write set. The read-write set of an uncritical sequence includes all addresses to be accessed. We assume that the read-write sets of all uncritical sequences are correctly annotated for correctness. We further insert all addresses into the read-write set using TX_raddr and TX_waddr. Although the

```
1  void update(node_t* head, key_t key){
2    TX_begin();
3    node_t* node=TX_read(head);
4    key_t   k=TX_read(node->key);
5    if(g(k)==key){
6  -   val_t res=f(k);
7  -   TX_write(node->value,res);
     // f(k) is wrapped into new_f
     // Record new_f as a deffered funtion
8  +   TX_uncritical(new_f,
9  +                 TX_raddr(k),
10 +                 TX_waddr(node->value));
11   }
12   TX_end();
13 }
14 +void new_f(void* r_set, void* w_set){
15 +  key_t   k=TX_get(r_set,0);
16 +  node_t* node->value=tx_get(w_set,0);
17 +  val_t   res=f(k);
18 +  TX_write(node->value,res);
19 + }
```

Fig. 6: Transforming transaction. $f(k)$ and its associated TX_write are annotated as uncritical instructions using $TX\_uncritical$.

actual addresses in the read-write set are unknown before execution, we can annotate the read-write set with high-level language variables. Our annotation functions only require these addresses to be known just before annotating them. After that, we move the uncritical sequence to a wrapper function and pass the function using $TX\_uncritical$.

Fig 6 shows an example of transforming the transaction in Fig 5. After identifying critical instructions, we find Lines 6-7 in the source code are uncritical instructions. Therefore, we extract these two lines into a wrapper function new_f, which receives read and write sets as parameters. We replace these two lines with a $TX\_uncritical$ in their original location and pass the wrapper function new_f and its read-write set as parameters. The use of TX_raddr and TX_waddr is similar to a stack push while TX_get is similar to a stack pop. The system pushes a function pointer and its read-write set in the thread-local space and pops them in the latter execution. The execution of some instructions can be deferred after obtaining all locks, thus allowing some locks to be unlocked earlier.

Some addresses come from local variables in the stack and have limited life cycles. When the system dereferences these local variables' addresses in the commit phase, these local variables may have been freed, which causes program errors. We allow local variables not to be passed by TX_raddr and TX_waddr but never check this special case because passing the reference of local variables as parameters is a common programming bug. How to avoid this bug is beyond the scope of this work.

D. Dependency Conflict Avoidance Mechanism

We discuss data dependency conflicts and introduce our dependency-conflict avoidance mechanism.

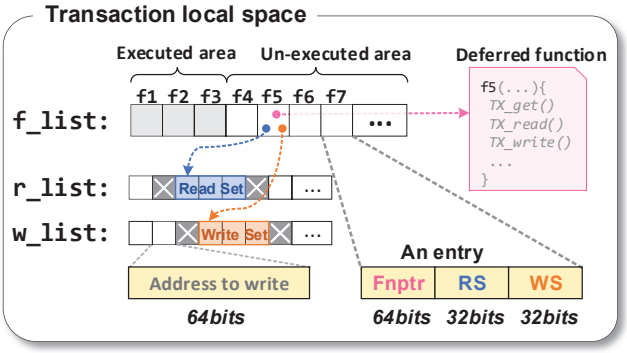


Fig. 7: Data structures to record un-executed functions.

Data Dependency Conflicts. In general, only uncritical instructions are deferred. However, due to the inaccurate dependency tracking, some critical instructions may be regarded as uncritical and wrongly deferred, which results in two types of dependency conflicts:

(i) instruction i depends on j, but j is deferred after i.

(ii) instruction i depends on j, i and j are both deferred, but j is reordered after i.

For the first type, we check the function pointer list before each transaction read and write. If the address to be read or written exists in the read-write set of an annotated function, there is a dependency conflict. Thus, the annotated function needs to be executed before reading or writing to maintain original data dependency. After execution, to avoid an annotated function being executed twice, the list head needs to move to the last unexecuted function pointer.

For the second type of dependency conflict, we store the function pointers of the annotated functions in a list in the FIFO manner. Moreover, the executions of the annotated functions always start from head to tail, thus preventing the reorder among functions.

E. Extending baseline system

Data Structures. As shown in Fig 7, we use three double linked lists, i.e., f_list, r_list, and w_list, to respectively store those annotated functions and their read-write sets. Each node in f_list contains a function pointer, including a r_list pointer and a w_list pointer. The r_list and a w_list pointers point to the beginning of the read and write set. In r_list and w_list, each node contains a 64-bit address. We use $2^{64}-1$ to indicate the end of a function's read-write set. Moreover, each list contains a head pointer and a tail pointer to respectively record the first and last un-executed nodes. When a function is executed, the head pointer will move to the next un-executed node.

To accurately identify the address of the last read/write, we use a hash table cnt_tab to maintain a reference counter. The address x is passed by TX_raddr or TX_waddr, and the counter of x will increase by one. When the annotated function f is executed, the counters of addresses in f's read-write set will decrease by one. Moreover, if f is executed in the commit phase, we will check whether a counter is zero after decreasing the counter. If the counter is zero, f

is the last function that accesses this address, and we then unlock this address. The system can unlock data after its last access without waiting for the whole transaction. In cnt_tab, when two addresses a and b are hashed into the same counter, one may not be unlocked, since our design only unlocks data whose counter is equal to zero. We avoid this issue by using the same hash function in the lock-entry hash table. The data structure described above is allocated in transaction local space without centralized synchronization.

Annotation Functions. We add four annotation functions to the baseline system: $TX\_uncritical$, TX_raddr, TX_waddr and TX_get. $TX\_uncritical$ receives a function pointer of a wrapper function and two pointers. The last two pointers are obtained through the return value of TX_raddr and TX_waddr. When a $TX\_uncritical$ is called, the input parameters will be inserted into the tail of f_list.

TX_raddr and TX_waddr receive a 64bit address as input and return a r_list or w_list pointer. When a TX_raddr is called, the transaction will read and lock the address. If failing, the transaction will abort and roll back. If successful, the system will insert the address into the tail of r_list and increase the corresponding counter in cnt_tab. TX_waddr is similar to TX_raddr except for the need for writing locks.

TX_get is used in the wrapper functions to pop the addresses from r_list and w_list. TX_get receives a r_list or w_list pointer and an index to obtain the index's address in the list.

Transaction Processing. TX_read: Before reading an address, the transaction will acquire the read-lock of the address. If the address has already been write-locked, we check whether this address exists in the write set of an annotated function via scanning the w_list from the tail to the head. Assuming the address is found in the write set of a function, the transaction needs to execute functions in f_list from the head to the function. Only after functions that have the same address in their write set are executed, can the transaction read the address.

TX_write: Before writing an address, the transaction will acquire the write-lock of the address. If the address has been locked, it checks whether this address is in the read or write sets of an annotated function by scanning both w_list and r_list from the tail to the head. If the address exists, the transaction executes the functions in f_list from the first to the last function that accesses the address.

The difference between TX_read and TX_write is that TX_write needs to check both read and write sets. This is because the transaction read only maintains read-after-write dependency while write should maintain write-after-read and write-after-write dependencies.

TX_end: When the transaction is ready to be committed, we first unlock the addresses whose counters in cnt_tab are zero, which indicates the addresses will not be accessed again. We further execute the annotated functions in f_list one by one. After each function is executed, the counters of addresses in the read-write set decrease, and the lock of the address whose counter is equal to zero is released. We clear all local data structures of transactions to prepare for the next transactions.

Logging. Logging is important for durable transactions. A durable transaction can persist its updates if the transaction is committed and recovers these updates from crashes. Without logging, the system possibly loses some committed updates when crashes occur. In general, a durable transaction system needs to guarantee ACID, which means **A**tomicity, **C**onsistency, **I**solation, and **D**urability. For durability, we store data updates to several log files in the persistent media, such as hard disks and SSDs, and recover from crashes by using these files.

On the other hand, for transactions that are not durable, logging is unnecessary, which can be exploited to decrease system overheads and improve performance. Some transaction systems (e.g. software transactional memory [27], [28] and in-memory databases [15], [29]) only guarantee **ACI** without **D**urability. In these systems, no modifications are required compared to baseline systems. Therefore, we only discuss how to adapt our design to durable transaction systems in this section.

In durable transaction systems, logging must be performed before unlocking any data. For baseline systems, logging contains all transaction writes. However, in our design, some transaction writes are represented as a function and dependent data. As a result, these functions and data dependencies need to be persisted to log files as well. Specifically, at the beginning of TX_end, the data structure in Fig 7 needs to be flushed to the log file. After logging these transaction writes and dependencies, the system can commit the transaction as described in TX_end. Each time completing a deferred function, we add a record in the log.

Limitations of Recovery. Deferred execution regards some data as the combination of their dependent data and the function to execute. For recovery, both dependent data and the function need to be persisted. The key challenge is how to persist and recover a function. A function contains a set of instructions and the associated runtime context.

Unfortunately, the runtime context, such as register, stack, and global data, becomes vulnerable to being destroyed during a system restart. To recover functions from crashes, one approach is *checkpointing*, which requires each memory read/write to be persisted, leading to high logging overheads. Another approach leverages the function be *pure*, which means the execution of the function does not depend on the runtime context. Therefore a function pointer is enough for correct recovery. The main drawback of this approach is that most deferred functions are non-pure. The execution of a non-pure function depends on the runtime context. Deferring the execution of a non-pure function possibly leads to a different execution result if some relevant runtime contexts have changed.

CATS leverages the latter approach (pure function) to provide durable transactions. Hence, the durable transactions in CATS are defined below:

- Data dependencies of deferred execution cannot contain

any local variables, since these data are allocated in the stack, which is runtime context.
- Data dependencies of deferred execution cannot contain any data that will be modified by other processes, e.g. CPU clocks.
- Deferred functions cannot contain explicit calls to TX_abort. Otherwise, a committed transaction may call TX_abort, leading to read-uncommitted-data.

IV. Performance Evaluation

A. Configurations

Experimental Setup and Baselines. Our experiments are conducted on a 52-core Intel machine with 2 NUMA nodes. Each NUMA node contains 26 cores (Xeon Gold 6230R 2.20GHz) and 187GB memory.

We implemented CATS on top of RSTM [28], an open-sourced word-based transactional memory system. Although our design is not limited to transactional memory, we choose it as a case in point since ACID databases contain logging and recovery operations that are hard to support recoverable deferred execution as discussed in Section III-E. We collect transaction throughputs by running each workload for at least 60 seconds and calculate the speedup of CATS compared with the baseline by dividing the throughput.

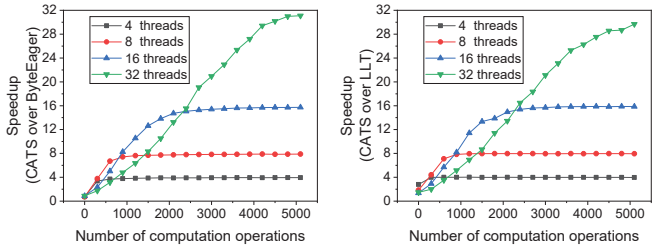
We evaluated our work with two open-sourced implementations of OCC [13] and 2PL [12] in the same configurations:

- **ByteEager** [30]: An implementation of 2PL. We acquire locks before reading and writing the data. The writes are performed in place and an undo log is used to roll back when an abort occurs. It is worth noting that the undo log is allocated in memory and only used for rollback without persistent operations.
- **LLT** [27]: An implementation of OCC. The write operations are buffered in the write log, and the locks of all relevant data are acquired in the commit phase for updates.

B. Workloads and setup

Our design focuses on transactional workloads, which is *computation-intensive* (e.g. complex data analysis) and *highly contentious* (e.g. multiple txns compete for a hotspot). The rationality is that more and more real-world applications require performing complex analysis over recently changed data in an all-or-nothing manner [4]. Ideal transaction systems and their concurrency control protocols are supposed to scale with the computation in transactions, which recently becomes more important.

We comprehensively examined existing real-world transactional workloads that are widely used and found none of them met our requirements. Database workloads contain either low contentions (e.g. TPC-E [31]) or few uncritical operations (e.g. TPC-C [10] and smallbank [32]). For example, the new-order transaction in TPC-C contains a read-modify-write hotspot at the beginning and most of the remaining operations have no data dependency upon the hotspot. We defer the executions of these operations to mitigate concurrency conflicts. However,



(a) Speedup over ByteEager (b) Speedup over LLT

Fig. 8: The throughput speedup of our design over ByteEager and LLT when the number of computation operations increases.

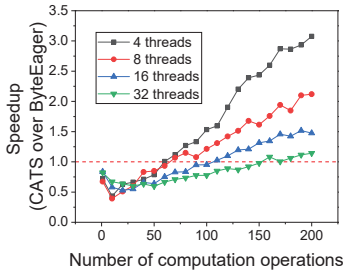


Fig. 9: The throughput speedup of our design over ByteEager when the number of computation operations increases.

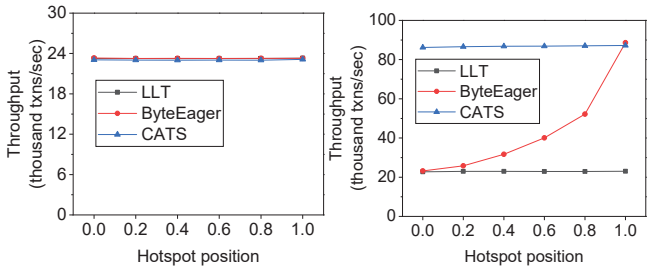
these operations are simple computations and consume several CPU clocks, which is cost-efficient in our design. Moreover, the widely used transactional memory workload STAMP [33], which contains several transactional applications, does not exhibit the characteristics that our design targets: *intruder* and *labyrinth* contain few computation operations. *kmeans* and *genome* contain lots of computation operations but they are outside the transaction. *yada* and *vacation* contain some computation operations within the transaction, but the evaluation results of these operations are immediately used by others, which means these operations are critical and cannot be deferred. Hence, we do not expect speedups on these transactional workloads.

We use several synthetic workloads similar to Bamboo [20] to evaluate our design.

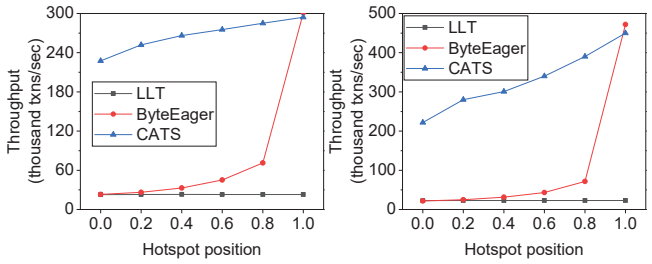
C. Synthetic workloads

Varying the number of computation operations. We design a high contention workload that contains a read-modify-write hotspot at the beginning, a for-loop at the middle, and a write at the end. The write address at the end has a data dependency upon the read at the beginning and is calculated through the for-loop, supporting the rationality for putting these operations in a transaction. We vary the number of computation operations in a transaction by changing the amounts of for-loop iterations. Within each iteration, we perform an arithmetic addition as the computation operation.

Fig 8a shows the throughput speedup of CATS over ByteEager when the number of computation operations changes. With the increase of computational operations, the speedup increases until reaching the theoretical upper limit (i.e. the



(a) 1 thread (b) 4 threads



(c) 16 threads (d) 32 threads

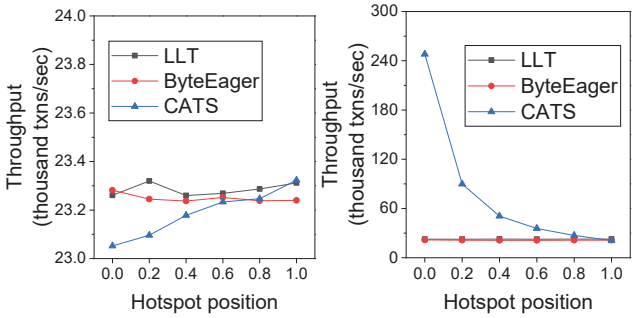
Fig. 10: The throughputs of CATS, ByteEager and LLT as the hotspot position changes.

number of threads). The results demonstrate that CATS can completely mitigate the impact of uncritical computational operations within txns. We report similar results for LLT in Fig 8b. The throughput speedup of CATS over LLT is slightly higher than ByterEager. The main reason is that OCC performs worse than 2PL in this highly contentious workload. Specifically, compared to 2PL, which detects contentions immediately and aborts if failing to lock, OCC does not acquire the lock of the hotspot until completing the expensive computations, which makes OCC easier to abort and waste computation resources.

Fig 9 shows the minimal number of computational operations to benefit from CATS. When the number of computation operations is larger than 60, 70, 110, and 150, our work obtains higher throughput under 4, 8, 16, and 32 threads respectively. These results demonstrate that our work outperforms prior schemes on computation-intensive workloads.

Varying hotspot positions. We put a read-modify-write hotspot between two for-loops and vary the hotspot positions by changing the number of computational operations within the first for-loop. The total number of computation operations within these two for-loops is fixed at 200.

Fig 10 shows the throughput among LLT, ByteEager, and our work using different numbers of threads when the hotspot position changes. Since recording the read-set and write-set for correct deferred execution incurs additional overheads, the throughput of CATS is slightly lower than LLT and ByteEager , as shown in Fig 10a. As the number of threads increases, these overheads are negligible compared to the performance gains from reducing conflicts. As shown in Figures 10b, 10c, and 10d, CATS outperforms both ByteEager and LLT in most cases except for the hotspot position is equal to 1.0. When the hotspot is at the end of the txn, all computational operations become *critical* and cannot be reordered by CATS, leading to the same behavior as ByterEager.



(a) 1 thread (b) 32 threads

Fig. 11: The throughput of CATS, ByteEager and LLT when the percentage of uncritical operations changes.

Varying the percentage of critical operations. To further understand the impact of the proportion of critical operations on our design, we add a read-modify-write hotspot at the beginning based on the second workload. There is a data dependency between the newly inserted hotspot and the old hotspot. According to our design, the computational operations between these two hotspots are *critical*, which cannot be optimized by CATS. We change the percentage of critical operations by fixing the first hotspot at the beginning and moving the second hotspot from the beginning to the end of the txn. The closer the second hotspot is to the end of the txn (i.e. the closer hotspot position is 1.0), the more the percentage of critical operations.

Fig 11 shows the throughputs among LLT, ByteEager and CATS by using different numbers of threads, when the percentage of critical operations changes.

As shown in the Fig 11a, the throughput of CATS increases by using 1 thread when the percentage of critical operations increases, and finally becomes equal to ByterEager and LLT. This is because when the second hotspot position is close to 0.0, most operations are *uncritical* and will be reordered by CATS. Such reordering incurs additional overhead but cannot reduce concurrency conflicts in the single-threaded case, leading to a slightly lower throughput compared to other works.

For the multi-threaded case, the additional overhead can be ignored and concurrency conflicts become the main bottleneck. Fig 11b shows that CATS obtains significant performance improvement compared with LLT and ByteEager in the multi-thread cases when the percentage of uncritical operations is high (the hotspot position is close to 0.0). When all operations are critical, the throughput of CATS is the same as LLT and ByteEager since there are no uncritical operations to reorder.

In summary, in the multi-thread cases, CATS obtains higher throughput than LLT and ByteEager when the percentage of uncritical operations is high. The performance of CATS is similar to LLT and ByteEager when the proportion of uncritical operations is low.

V. Related Work

Deferred execution. Lazy evaluation [21] and early write visibility [34] propose to execute some transactional operations after transactions are committed in deterministic databases. However, these schemes rely on a pre-generated total order to avoid concurrent conflicts which limits their applicability. Inspired by lazy evaluation, DRP [22] introduces deferred execution to avoid the rank mismatch issue inherent with runtime pipelining [19]. Unfortunately, none of these deferred execution approaches can handle transactions with complex logic such as conditional updates and the second address indexing. Instead, CATS can leverage deferred execution to speed up these transactions and fix data dependency conflicts at runtime.

Transaction execution reordering. There are various approaches to speed up traditional transactions by changing the execution orders of transactions. QURO [35] and Chiller [36] focus on reordering queries within transactions and executing the queries with high conflict rate. Both QURO and Chiller decide the order of locking and unlocking in advance, while we dynamically decide it at runtime in general cases. Transaction chopping [17] and improved designs (e.g., IC3 [18] and Runtime Pipelining [19]) require static program analysis before the execution to split transactions into small pieces. Instead of acquiring global knowledge of transactions, Bamboo [20] speculatively releases locks and allows cascading aborts. To cope with cascading aborts, Bamboo maintains a centralized data structure to track dependent transactions. In contrast, CATS only reorders inter-transaction operations and thus requires no global knowledge and no significant modifications to the transaction system.

VI. Conclusion

In order to efficiently mitigate concurrency conflicts, this paper presents CATS, a computation-aware transaction processing system that reorders the operations within transactions. CATS leverages the critical-operation identification algorithm to efficiently partition the operations within a transaction into critical and uncritical ones. These uncritical operations are placed after unlocking to alleviate potential conflicts among transactions. Our experimental results demonstrate that CATS significantly outperforms conventional OCC and 2PL-based schemes for computation-intensive workloads.

Acknowledgment

This work was supported in part by National Natural Science Foundation of China (NSFC) under Grant No. 62125202 and U22B2022.

References

[1] D. Kossmann, T. Kraska, and S. Loesing, "An evaluation of alternative architectures for transaction processing in the cloud," SIGMOD, 2010.
[2] Y. Sokolik and O. Rottenstreich, "Age-aware fairness in blockchain transaction ordering," IWQoS, 2020.
[3] J. Zhang, Y. Cheng, X. Deng, B. Wang, J. Xie, Y. Yang, and M. Zhang, "Preventing spread of spam transactions in blockchain by reputation," IWQoS, 2020.
[4] J. Chen, Y. Ding, Y. Liu, F. Li, L. Zhang, M. Zhang, K. Wei, L. Cao, D. Zou, Y. Liu, L. Zhang, R. Shi, W. Ding, K. Wu, S. Luo, J. Sun, and Y. Liang, "ByteHTAP: bytedance's HTAP system with high data freshness and strong data consistency," VLDB, 2022.
[5] G. J. Chen, J. L. Wiener, S. Iyer, A. Jaiswal, R. Lei, N. Simha, W. Wang, K. Wilfong, T. Williamson, and S. Yilmaz, "Realtime Data Processing at Facebook," SIGMOD, 2016.
[6] J. Gray, "The transaction concept: Virtues and limitations," VLDB, 1981.
[7] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," CSUR, 1983.
[8] ORACLE, "Mysql," http://www.mysql.com, 2022.
[9] The PostgreSQL Global Development Group, "Postgresql," http://www.postgresql.org, 2022.
[10] The Transaction Processing Council. TPC-C Benchmark V5.11. http://www.tpc.org/tpcc/
[11] G. Kang, L. Wang, W. Gao, F. Tang, and J. Zhan, "OLxPBench: Real-time, Semantically Consistent, and Domain-specific are Essential in Benchmarking, Designing, and Implementing HTAP Systems," ICDE, 2022.
[12] K. P. Eswaran, J. N. Gray, R. A. Lorie, and I. L. Traiger, "The notions of consistency and predicate locks in a database system," Commun. ACM, 1976.
[13] H. T. Kung and J. T. Robinson, "On optimistic methods for concurrency control," TODS, 1981.
[14] S. Tu, W. Zheng, E. Kohler, B. Liskov, and S. Madden, "Speedy transactions in multicore in-memory databases," SOSP, 2013.
[15] Z. Wang, H. Qian, J. Li, and H. Chen, "Using restricted transactional memory to build a scalable in-memory database," EuroSys, 2014.
[16] X. Yu, G. Bezerra, A. Pavlo, S. Devadas, and M. Stonebraker, "Staring into the abyss: An evaluation of concurrency control with one thousand cores," VLDB, 2014.
[17] D. Shasha, F. Llirbat, E. Simon, and P. Valduriez, "Transaction chopping: Algorithms and performance studies," TODS, 1995.
[18] Z. Wang, S. Mu, Y. Cui, H. Yi, H. Chen, and J. Li, "Scaling Multicore Databases via Constrained Parallel Execution," SIGMOD, 2016.
[19] C. Xie, C. Su, C. Littley, L. Alvisi, M. Kapritsos, and Y. Wang, "High-performance ACID via modular concurrency control," SOSP, 2015.
[20] Z. Guo, K. Wu, C. Yan, and X. Yu, "Releasing Locks As Early As You Can: Reducing Contention of Hotspots by Violating Two-Phase Locking," SIGMOD, 2021.
[21] J. M. Faleiro, A. Thomson, and D. J. Abadi, "Lazy evaluation of transactions in database systems," SIGMOD, 2014.
[22] S. Mu, S. Angel, and D. Shasha, "Deferred Runtime Pipelining for contentious multicore software transactions," EuroSys, 2019.
[23] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck, "An efficient method of computing static single assignment form," POPL, 1989.
[24] H. D. Pande, W. A. Landi, and B. G. Ryder, "Interprocedural def-use associations for c systems with single level pointers," TSE, 1994.
[25] C. Lattner and V. Adve, "Llvm: A compilation framework for lifelong program analysis & transformation," CGO, 2004.
[26] Y. Wu and K.-L. Tan, "Scalable in-memory transaction processing with htm," USENIX ATC, 2016.
[27] D. Dice, O. Shalev, and N. Shavit, "Transactional locking II," DISC, 2006.
[28] Rochester Synchronization Group. "Rochester Software Transactional Memory." https://www.cs.rochester.edu/research/synchronization/rstm/.
[29] X. Wei, J. Shi, Y. Chen, R. Chen, and H. Chen, "Fast in-memory transaction processing using RDMA and HTM," SOSP, 2015.
[30] P. Felber, C. Fetzer, and T. Riegel, "Dynamic performance tuning of word-based software transactional memory," PPoPP, 2008.
[31] The Transaction Processing Council. TPC-E Benchmark V1.14. http://www.tpc.org/tpce/
[32] The H-Store Team. SmallBank Benchmark. https://hstore.cs.brown.edu/documentation/deployment/benchmarks/smallbank/
[33] C. C. Minh, J. Chung, C. Kozyrakis, and K. Olukotun, "Stamp: Stanford transactional applications for multi-processing," IISWC, 2008.
[34] J. M. Faleiro, D. J. Abadi, and J. M. Hellerstein, "High performance transactions via early write visibility," VLDB, 2017.
[35] C. Yan and A. Cheung, "Leveraging lock contention to improve OLTP application performance," VLDB, 2016.
[36] E. Zamanian, J. Shun, C. Binnig, and T. Kraska, "Chiller: Contention-centric Transaction Execution and Data Partitioning for Modern Networks," SIGMOD, 2020.