FPGA-Accelerated Optimistic Concurrency Control for Transactional Memory

Zhaoshi Li
Tsinghua University
lizhaoshi@tsinghua.edu.cn

Leibo Liu*
Tsinghua University
liulb@tsinghua.edu.cn

Yangdong Deng
Tsinghua University
dengyd@tsinghua.edu.cn

Jiawei Wang
Tsinghua University

Zhiwei Liu
Tsinghua University

Shouyi Yin
Tsinghua University

Shaojun Wei
Tsinghua University

ABSTRACT

Transactional Memory (TM) has been considered as a promising alternative to existing synchronization operations, which are often the largest stumbling block to unleashing parallelism of applications. Efficient implementations of TM, however, are challenging due to the tension between lowering performance overhead and avoiding unnecessary aborts.

In this paper, we present Reachability-based Optimistic Concurrency Control for Transactional Memory (ROCoCoTM), a novel scheme which offloads concurrency control (CC) algorithms, the central building blocks of TM systems, to reconfigurable hardware. To reduce the abort rate, an innovative formalization of mainstream CC algorithms is developed to reveal a common restriction that leads to unnecessary aborts. This restriction is resolved by the ROCoCo algorithm with a centralized validation phase, which can be efficiently pipelined in hardware. Thanks to a high-performance offloading engine implemented in reconfigurable hardware, ROCoCo algorithm results in decreased abort rates and reduced performance overhead. The whole system is implemented on Intel's HARP2 platform and evaluated with the STAMP benchmark suite. Experiments show 1.55x and 8.05x geomean speedup over TinySTM and an HTM based on Intel TSX, respectively. Given the fast-growing deployment of commodity CPU-FPGA platforms, ROCoCoTM paves the way for software programmers to exploit heterogeneous computing resources with a high-level transactional abstraction to effectively extract the parallelism in modern applications.

CCS CONCEPTS

• **Hardware → Hardware accelerators**; • **Computing methodologies →** *Parallel programming languages*; • **Computer systems organization →** *Reconfigurable computing*.

*Corresponding author: Leibo Liu (liulb@tsinghua.edu.cn)

MICRO-52, October 12–16, 2019, Columbus, OH, USA
© 2019 Association for Computing Machinery.
ACM ISBN 978-1-4503-6938-1/19/10…$15.00
https://doi.org/10.1145/3352460.3358270

KEYWORDS

FPGA, Hardware Accelerator, Transactional Memory

ACM Reference Format:
Zhaoshi Li, Leibo Liu, Yangdong Deng, Jiawei Wang, Zhiwei Liu, Shouyi Yin, and Shaojun Wei. 2019. FPGA-Accelerated Optimistic Concurrency Control for Transactional Memory. In *The 52nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-52), October 12–16, 2019, Columbus, OH, USA*. ACM, New York, NY, USA, 13 pages. https://doi.org/10.1145/3352460.3358270

1 INTRODUCTION

With the prevalence of parallel computing facilities, the ability to expose the inherent parallelism in applications with a reduced level of programming effort is becoming increasingly critical. Transactional memory (TM) has been considered as a promising solution toward the above objective and attracted extensive attraction in both the programming and architecture communities [21]. TM provides a concise framework to a wide variety of essential problems such as parallelizing programs with unknown dependence [50], achieving fine-grained synchronization without managing locks [63], extracting concurrency from legacy codes [56], and achieving durability in non-volatile memories [32].

From the programming perspective, the basic idea of TM is to devise semantics and language constructs, which can be used by programmers to identify atomic code blocks and thus alleviate the burden of coordinating concurrent access to shared states [1]. From an implementation point of view, TM can be realized as a software runtime (STM) or a hardware architecture (HTM). Both approaches are designed to extract parallelism by proactively orchestrating transactions to maximize parallelism and aborting transactions that would result in unsafe states to enforce correctness. Numerous studies have been dedicated to developing efficient STM and HTM solutions. Among these, both Intel [71] and IBM [28] already released commodity processors incorporating HTM.

However, empirical evaluations of TM on commodity systems only demonstrate limited performance improvement. The speedup is especially unsatisfying on benchmarks with long transactions running on processors with a modest number of cores [13, 14, 70]. In the case of STM, the major performance overhead is caused by the manipulation and inspection of transactional states (i.e. metadata) to resolve conflicts among concurrent transactions [6, 62]. On the other hand, HTM alleviates the overhead by providing architectural support for transactions. For example, Intel TSX [71] is equipped with hardware support for new instructions to identify transactional regions and modified cache coherence protocols for conflict detection. However, HTMs suffer from spurious aborts introduced by architectural limitations like cache capacity [14]. In summary, the tradeoff between reducing performance overhead and avoiding spurious aborts is the central challenge of TM designs.

To make things worse, TM systems are notoriously hard to design, in particular their concurrency control (CC) algorithms for achieving transactional semantics by scheduling interleaved transactions. Previous works investigate the effectiveness of CC algorithms by arguing how *anomalies* (race conditions) and *restrictions* (false alarms) are resolved in a case-by-case manner [3, 41, 59]. Given the complexity of ordering read-write operations among transactions, even a tiny modification on CC algorithm may hamper its correctness.

The prevalence of commodity CPU-FPGA platforms [8, 26, 52] offer new opportunities to resolve the tension between performance overheads and abort rates in TM designs [7, 42]. In this work, we first formalize transactional semantics. We use the formalization to analyse mainstream CC algorithms and identify a new restriction that leads to unnecessary aborts. To relieve this restriction, we propose a novel CC algorithm with centralized validation scheme. This algorithm can be efficiently pipelined on FPGAs, which also reduces the performance overhead.

The major contributions of this paper are as follows.

- We establish an axiom-based transactional semantics on the basis of the order theory to discern ambiguity around design choices of CC algorithms. With this formalization we identify the if-and-only-if conditions of common transactional semantics. Then we show that mainstream CC algorithms are based the sufficient but not necessary condition for correctness and thus lead to over-stretched designs.
- We propose a novel **ROCoCo algorithm** (Reachability-based Optimistic Concurrency Control) for serializability. Experiments show that this algorithm reduces aborts by up to 56.2% and 20.2% when compared with 2PL and TOCC algorithms on synthesized benchmarks.
- Based on the ROCoCo algorithm, we implement a hybrid TM system (**ROCoCoTM**) whose validate phase is offloaded to an out-of-core FPGA. Evaluations with the STAMP benchmark suite [46] on Intel HARP2, a heterogeneous CPU-FPGA platform, show that ROCoCoTM enables 1.55x and 8.05x geomean speedups over TinySTM [17], a STM with TOCC, and an HTM with 2PL based on Intel TSX, respectively.

The rest of this paper is structured as follows. Section 2 introduces backgrounds and related works. Section 3 proposes the axiom-based transactional semantics. Section 4 presents the ROCoCo algorithm. Section 5 shows how ROCoCoTM is implemented on a heterogeneous CPU-FPGA platform. Section 6 reports experimental results on Intel HARP2 platform. Section 7 concludes the paper.

2 BACKGROUNDS

2.1 Transactional Semantics

To simplify the programming model, a *transaction* groups several read and write operations into one code snippet that executes as if it is the only computation in the system. Practical systems execute multiple transactions concurrently for better performance. For example, in figure 1 two transactions t_1 and t_2 access two objects x and y concurrently. The classic definition of transaction asserts *atomicity* and *isolation*. Roughly speaking, atomicity means a transaction either completes (*commits*) as a whole, or *aborts* and undoes any update it has made to the system, while isolation suggests a transaction should not interfere concurrent transactions [20].

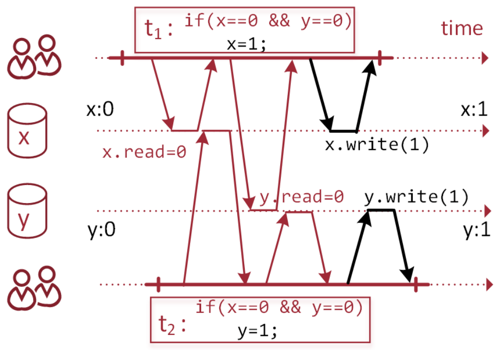


Figure 1: Two threads manipulate two objects $\{x, y\}$ through transactions.

The definition is vague [43] in that the concept of isolation is not accurate. **A common interpretation of isolation** is that "state changes made by other concurrently executing threads after a transaction T begins are not visible to T while T executes" [48]. Then both transactions in figure 1 will commit, i.e. both x and y are set to 1, since both writes are not visible to the other's reads. This result is counter-intuitive because only one transaction can write successfully if these two transactions are executed serially by a single thread.

Transactional **semantics** specify which results of concurrent executions of transactions meet intuitive expectations [20]. TM systems differ substantially in terms of the semantics they enforce. The aforementioned isolation leads to a semantic named **snapshot isolation** (SI) that suffers from the write skew anomaly [16] as shown in figure 1. The majority of TMs claim **serializability** [49], which states that the result of concurrent transactions should be identical to *a* result in which they execute serially. Since serializability leads to more intuitive results than SI by avoiding more race conditions, we say serializability is **stronger** than SI.

2.2 Concurrency Control

A **concurrency control** (CC) algorithm enforces a specific semantic by (a) scheduling concurrent read/write (R/W) operations to avoid violations during execution, and/or (b) optimistically executing transactions and validating their memory footprints before committing transactions. In case of serializability, CC can be classified as **optimistic CC** (OCC) or **pessimistic CC** (PCC) depending on whether approach (b) is used or not [36]. Empirically, PCC causes more unnecessary aborts than OCC [48, 72].

A transactional semantics S is **compositional** if the whole system enforces whenever each object enforces S [69]. It has be proven that SI is compositional, while serializability is not. Since SI's correctness is easy to guarantee by providing multi-version objects, it is provided by almost all databases [35] and some TMs [10, 40, 55]. In contrast, one has to reason how race conditions are resolved case-by-case if targeted semantics is serializability [44, 47, 61].

A key insight from previous works [22, 69] is that CC algorithms based on a non-compositional semantics must either rely on a *centralized scheduler* for all objects, or else place additional *restrictions*. This insight can be exploited to guide the design of CC algorithms for serializability. For example, in PCC algorithms exemplified by 2-phase-locking (2PL) [5, 37, 48], an object that is locked by a transaction's *execution phase* cannot be accessed by another one, until it is released during the *commit phase* of the first transaction. This restriction may degrade 2PL's performance as it forbids concurrent accesses to an object. Under 2PL, t_2 of figure 1 will be either blocked or aborted when it tries to access object x that was locked by t_1.

2.3 Related Works

Previous works on OCC rely on centralization and/or restriction to enforce serializability during the *validation phase*, which is launched between the (optimistic) execution phase and the commit phases.

Centralization. In *Backward OCC* (BOCC) and *Forward OCC* (FOCC) algorithms [25, 36], validations are conducted by broadcasting the updated objects of a validated/committed transaction to all the other transactions. These algorithms are only efficient for broadcast-friendly systems, e.g. HTMs based on snooping protocols [2, 57]. Otherwise, the serialized broadcast will significantly drag down the overall performance. Other endeavours attempt to conduct system-wide validation with a centralized thread [45] or bookkeeping [12, 44, 62], which are prone to become bottleneck.

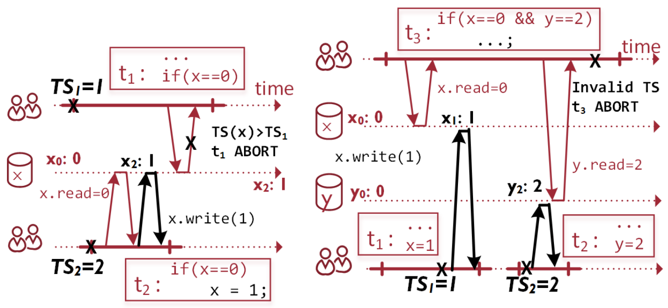
Restrictions. A sufficient condition for serializability is that the ordering among concurrent transactions with regard to their R/W-dependences is acyclic [49, 54]. A sufficient approach to maintaining acyclicity is to abort transactions that have both incoming and outgoing R/W-dependences during validation [4, 29, 53, 54]. These implementations are conservative because they do not check whether dependences of a transaction actually occur within a cycle.

To reduce unnecessary aborts caused by restrictions, recent works on serializability in both TM [17, 53] and database [66, 73] resort to **Timestamped OCC (TOCC)** with a minimized centralization, i.e. a shared monotonic-increasing timestamp. A unique timestamp is associated with each transaction during execution or validation phase, with which a transaction stamps its updated objects at commit phase. A transaction is aborted if it reads some objects with greater timestamps. Then transactions are serialized in the order of timestamps. Nevertheless, TOCC is sufficient but not necessary to serializability, as demonstrated in section 3.1.

We propose a novel CC algorithm, ROCoCo, to balance the performance overhead and abort rate. Our key motivation is that with the advent of heterogeneous architectures, it is profitable to offload the centralized validation phase off OCC to an out-of-core accelerator. FPGAs could serve our purpose by providing massive bit-level parallelism to track the dependencies among transactions.

3 AXIOM-BASED SEMANTICS

Transactional semantics are essential to guarantee the correctness and avoid unnecessary aborts of TM designs. Although previous



(a) If transactions acquire timestamps at the start, t_1 will abort.

(b) If transactions acquire timestamps before committing updates, t_3 will abort.

Figure 2: Two cases for phantom orderings

works have formalized transactional semantics with the order theory [19, 60], our formalization distinguishes the previous ones in the following two points: (1) we apply interval order to analyse the restriction of TOCC algorithms and show that TOCC is sufficient but unnecessary for serializability; (2) we prove that acyclicity is the if-and-only-if conditions of serializability so that over-stretched semantics can be avoided. To motivate the new formalization proposed in this work, we demonstrate a case study on the restriction, so called *phantom ordering*, that is ubiquitous in TOCC.

3.1 Case Study: Phantom Ordering

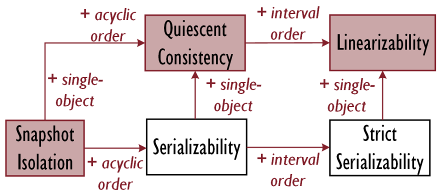
As discussed in section 2.3, a transaction in TOCC acquires a monotonically-increasing timestamp to validate its R/W-conflicts to concurrent transactions. A happen-before relation $\rightarrow_{rw}$ on a set of transactions can be deducted from the R/W-conflicts. Note that $\rightarrow_{rw}$ is a logical order rather than the order in which transactions should happen in real time.

- (Read-after-Write) If a transaction t_1 reads an updated object from t_2, t_2 should happen before t_1, denoted as $t_2 \rightarrow_{rw} t_1$.
- (Write-after-Read) If a transaction t_1 writes an object whose previous version was read by t_2, then $t_2 \rightarrow_{rw} t_1$.
- (Write-after-write) If a transaction t_1 overwrite an object that was written by t_2, then $t_2 \rightarrow_{rw} t_1$.

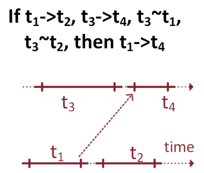
With timestamped objects, TOCC algorithms enforce serializability by aborting transactions whose $\rightarrow_{rw}$ violate the ordering of timestamps. As a result, the equivalent serial execution of transactions can be predicated by their timestamps.

However, a restriction arises if an aborted transaction could have been committed by reordering the timestamps. For example, assuming each transaction acquires a timestamp when it starts [54, 74], in figure 2 (a), t_1 acquires a smaller timestamp ($TS_1 = 1$) than t_2, which causes it to abort since t_1 reads the updated x whose version ($TX(x) = 2$) is updated by t_2. Although t_1 can commit if it is ordered after t_2, a phantom ordering induced by the ordering of timestamp acquisition prevents the reorder.

To overcome the restriction, Lazy Snapshot Algorithm (LSA) [17] postpones the acquisition of timestamps until the validation phase. Then t_1 of figure 2 (a) can commit with a larger timestamp than that of t_2. Yet the phantom ordering persists. As shown in figure 2 (b), even though this trace can be serialized as $t_2 \rightarrow_{rw} t_3 \rightarrow_{rw} t_1$



(a) Semantics and axioms. Shady blocks indicate compositional semantics.



(b) The phantom ordering from interval order

Figure 3: Transactional semantics can be incrementally built up with axioms.

according to R/W dependencies, the timestamps forbid ordering t_2 before t_1, which causes t_3 to abort.

Being aware of these false alarms, some TM designs [10, 29] rely on more complex timestamps. Nevertheless, our formalization will show that the phantom ordering still haunts CC algorithms as long as global ordering primitives [34] are used as criteria for validation.

3.2 Formalization

The gist of the above-mentioned restriction is that the current definition of serializability only states the existence of a serial execution, without telling how to achieve the serial execution. Although monotonically timestamping each transaction is sufficient, its necessity is challenged by the phantom ordering.

To avoid over-stretching serializability, we introduce the axiom-based transactional semantics, where a set of transactions $T = \{t_1, t_2, \ldots\}$ meet a semantic *if and only if* some axioms are followed by their R/W-dependency $\rightarrow_{rw}$. Then a CC algorithm is both sufficient and necessary for a semantic when it only aborts transactions whose R/W-dependencies violate the corresponding axioms.

Nomenclature. The nomenclature of our discussion largely follows that of Herlihy et al. [22, 23]. The order theory generalizes intuitions on a binary relation of a set by specifying **axioms** that the relation must follow. For example, a strictly *partial order* is a relation following irreflexive, asymmetric and transitive order; while a strictly total order, or *linear order*, is a strictly partial order plus complete order (i.e. all pairs of elements are related).

Generally, a set of transactions T and its relation are denoted as a tuple $(T, \rightarrow)$, where $t_1 \rightarrow t_2$ indicates that behaviours of transaction t_1 should be visible to t_2 under that relation. Given $(T, \rightarrow)$, two transactions t_1 and t_2 are **concurrent** if they are not related, written as $t_1 \sim t_2$ [1]. Since arbitrarily scheduling concurrent transactions in a set of R/W-dependent transactions $(T, \rightarrow_{rw})$ will lead to incomprehensible results, transactional semantics are required.

An overview of **axiom-based semantics** is shown in figure 3 (a). Here the arrow between two semantics $S_1 \rightarrow S_2$ indicates that the semantic S_2 strengthens S_1 by following the axiom on the arrow in addition to S_1's axioms. With additional axioms, a semantic is *stronger* than another one in the sense that it is more restrictive and leads to more aborts.

The notion of strengthened semantics in Figure 3 (a) can be captured by extended relations. A relation $\rightarrow$ on transaction set T is said to be **extended** to a stronger relation $\rightarrow_s$ if $\forall t_1, t_2 \in T$, $t_1 \rightarrow t_2$ indicates $t_1 \rightarrow_s t_2$, written as $(T, \rightarrow) \subseteq (T, \rightarrow_s)$. The common interpretation of atomicity and isolation from section 2.1 leads to snapshot isolation in 3 (a), which can be extended to other semantics with axioms.

[1]Not to be confused with NOT visible, written as $t_1 \nrightarrow t_2$.

Serializability. Finding an equivalent serial execution of concurrent transactions T can be viewed as extending $\rightarrow_{rw}$ on T to a linear order. To this end, PCC algorithms like 2PL enforce serializability by solely scheduling R/W operations so that $\rightarrow_{rw}$ is a partial order [2], which may result in too many unnecessary waits and aborts during execution. On the other hand, an OCC algorithm generates $(T, \rightarrow_{rw})$ in execution phase, and finds a set of committed transaction $T_c \subseteq T$ by aborting some transactions in validation phase, such that $(T_c, \rightarrow_{rw})$ can be linearly-extended.

It is well understood that $(T_c, \rightarrow_{rw})$ can be extended to linear order *if* $\rightarrow_{rw}$ is acyclic. Since any finite and acyclic set has at least one minimal elements, a linear order can be constructed by iteratively picking up a minimal element. We can further prove that *acyclicity is both sufficient and necessary to serializability*[3].

As the axiom for serializability, acyclicity is also not compositional. For example, in figure 1 (b), although R/W-dependencies with regard to either object x or y are acyclic individually, their compositional trace is not acyclic.

Strict serializability and interval order. TOCC algorithms associate committed transactions with unique timestamps that correspond to the order of equivalent serial executions. Since the timestamps are acquired between the start and the end of transactions, they also represent the *real-time order* $\rightarrow_{rt}$, where $t_1 \rightarrow_{rt} t_2$ if t_2 starts after the end of t_1 in real time [22]. A serializable $(T, \rightarrow)$ is **strict serializable** if its serial equivalence is compatible with its real-time order [23]. Apparently, TOCC enforces strict serializability. But how does strict serializability limits concurrency and lead to unnecessary abort for serializability?

We will show *TOCC is unnecessary to serializability* by analysing the inevitability of the phantom ordering in strict serializability, no matter how the timestamp is designed. If the timestamps can be acquired anytime between the start and the end of transactions, each transaction can be viewed as an interval on the real axis. The left-to-right precedence relation of these intervals on a real axis, i.e. the real-time order $\rightarrow_{rt}$ of transactions on the real time, is characterized as **interval order** in order theory. By definition [18], a partial-ordered set is an interval order if it has no subset isomorphic to a pair of two-element linearly ordered sets shown in figure 3 (b), i.e., for every two related transactions committed by TOCC ($t_1 \rightarrow t_2$ and $t_3 \rightarrow t_4$ in figure 3 (b)), there always exists a phantom ordering ($t_1 \rightarrow t_4$) even though t_1 and t_4 are unrelated in R/W-dependency.

Note that strict serializability is also not a compositional semantic, since acyclicity, the non-compositional axiom, cannot be

[2]According to the order-extension principle, a relation can be extended to a linear order *if but not only if* it is a partial order.

[3]**Proof**. (Acyclicity $\Rightarrow$ serializability) It can be proved by constructing the linear order, i.e. order of serial execution, from acyclic order with topological sorting.

(Acyclicity $\Leftarrow$ serializability) Assuming $\rightarrow_{rw}$ is cyclic, let $\rhd$ be the transitive closure of $\rightarrow_{rw}$. Then there exists distinct t_1, $t_2 \in T$, such that $t_1 \rhd t_2$ and $t_1 \lhd t_2$. As $\rhd$ is the smallest transitive relation that contains $\rightarrow_{rw}$, any linear order (which is also transitive) that contains $\rightarrow_{rw}$ must contain $\rhd$. Then the linear order extended from $\{T, \rightarrow_{rw}\}$ is not asymmetric when t_1 and t_2 are considered, which counters the definition of linear order. Therefore $\rightarrow_{rw}$ should be acyclic.

inferred from interval order. But if each transaction only operates on a single object, an interval order on these transactions is sufficient for acyclicity[4] , which leads to the compositional semantic named linearizability[5], as shown in Figure 3 (a).

4 ROCOCO ALGORITHM

ROCoCo algorithm avoids restriction of TOCC by validating acyclicity without using timestamps. Although detecting cycles in the R/W dependency graph has been extensively studied in transaction processing and race detection, these approaches either incur huge computation complexity which can only be applied to offline analysis [41], or adapt Kahn's topological sorting [33] algorithm that is equivalent to TOCC [67]. Without using timestamps, ROCoCo can detect and abort a transaction that cause cycles in $O(1)$, and validate $(T, \rightarrow_{rw})$ in $O(|T|)$, making it suitable for online validation of OCC as discussed in section 4.1. The centralized validation mechanism of ROCoCo can be efficiently pipelined on FPGA to further reduce overhead as discussed in section 4.2.

4.1 ROCoCo

ROCoCo algorithm is inspired by the Warshall algorithm [68] to compute the transitive closure, or reachability, of a graph. Intuitively, for k acyclic transactions T_k, a cycle is formed between T_k and a new transaction t_{k+1} if there exists two paths in the R/W-dependency graph, such that t_{k+1} can reach a transaction $t_i \in T_k$ through one path, and t_i can reach t_{k+1} through another path. ROCoCo ameliorates Warshall algorithm by iteratively constructing a bitwise matrix to record reachability of transactions during OCC.

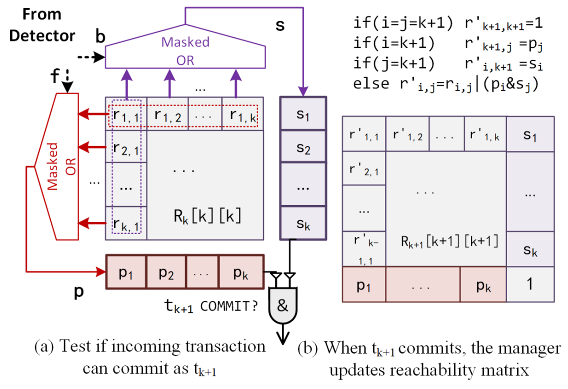
The essence of validating acyclicity is to probe the extended linear order from the incomplete and intransitive relation $\rightarrow_{rw}$. To this end, **transitive closure** is a powerful tool in that it describes the smallest (in the subset sense) transitive binary relation that contains $\rightarrow_{rw}$. In what follows, terminologies from order theory and graph theory, such as element (i.e. transaction) and vertex, related pair (i.e. R/W-dependency) and edge, acyclic relation and *directed acyclic graph* (DAG), precede and "can reach", succeed and "be reachable from", are used interchangeably.

Formally, the transitive closure $(T, \triangleright)$ of any finite binary relation $(T, \rightarrow)$ can be constructed iteratively as follows. Initially $\triangleright_0 = \rightarrow$. Then $\triangleright_k$ is the transitive relation induced by k relations from $\rightarrow$, i.e. $t_0 \triangleright_k t_{k+1}$ if $\exists t_1, ..., t_k \in T$ such that $t_0 \rightarrow t_1 \rightarrow ... \rightarrow t_k \rightarrow t_{k+1}$ (t_0 reaches t_{k+1} by stepping k edges in the corresponding graph). We say t_0 can **reach** t_{k+1}, or t_{k+1} **is reachable from** t_0, if $t_0 \triangleright t_{k+1}$. Finally, $\triangleright = \bigcup_{i \leq |T|, i \in \mathbb{N}} \triangleright_i$, where $|T|$ is the size of T.

Warshall's algorithm [68] for transitive closure simplifies this procedure by observing the fact that an element t_i can reach a distinct t_j, *iff.* $t_i \rightarrow t_j$, or there exists a distinct t_k such that $t_i \rightarrow t_k$ and $t_k \triangleright t_j$. Then the transitive closure can be constructed by iterating vertices in the DAG. However, the best-case performance of Warshall's algorithm is $O(|T|^3)$. This cost is still unacceptable for

[4]A relations of single-object transactions is irreflexive and asymmetric. Irreflexivity and interval order indicates transitivity (considering t_2 and t_3 are the same transaction in figure 3 (b)). Then the relation is partial order, and thus acyclic.

[5]The term "linearizability" is heavily overloaded in the TM community. The linearizability in our paper follows the definition of Herlihy et al. [23], i.e. it "can be viewed as a special case of strict serializability where transactions are restricted to consist of a single operation applied to a single object".



```
if(i=j=k+1) r'_{k+1,k+1}=1
if(i=k+1)   r'_{k+1,j} =p_j
if(j=k+1)   r'_{i,k+1} =s_i
else r'_{i,j}=r_{i,j}|(p_i&s_j)
```

(a) Test if incoming transaction can commit as t_{k+1} (b) When t_{k+1} commits, the manager updates reachability matrix

Figure 4: ROCoCo algorithm for serializability

on-line validation of TM. Moreover, this algorithm cannot handle cyclic graph no element can abort during its procedure.

To address this problem, we further exploit the fact: an element t_i is reachable from a distinct t_j *iff.* $t_i \leftarrow t_j$, or there exists a distinct t_k such that $t_i \leftarrow t_k$ and $t_k \triangleleft t_j$. Note this fact is the *dual* of above-mentioned fact in Warshall's algorithm. We use *forward* to represent the direction of relations (i.e. $\rightarrow$) in **Warshall's fact**, and *backward* to represent the direction (i.e. $\leftarrow$) in the **dual fact**. With these facts, we propose the ROCoCo algorithm that can detect and abort transactions that cause cycles in $O(1)$, and calculate the transitive closure of $(T, \rightarrow_{rw})$ in $O(|T|)$.

We start describing ROCoCo with conflict detection of $(T, \rightarrow_{rw})$. Similar to adjacent vectors for edges of a DAG, *vectors of forwardly and backwardly related pairs*, $\mathbf{f}$ and $\mathbf{b}$, between the transaction t and previous k committed transactions T_k, are generated from $(T, \rightarrow_{rw})$. Here $\mathbf{f}[i]$ indicates $t \rightarrow_{rw} t_i$. And $\mathbf{b}[i]$ indicates $t \leftarrow_{rw} t_i$

A manager validates the transaction t by inspecting $(\mathbf{f}, \mathbf{b})$, and discards t if cycles are detected, as shown in figure 4 (a). We exploit the fact for cycle detection: a cycle is formed if there exists a distinct transaction $t_i \in T_k$ such that $t \triangleleft t_i$ and $t \triangleright t_i$. To detect cycles within $O(1)$, the transitive closure of T_k is stored as an intermediate result. All committed transactions T_k before t form a DAG, whose transitive closure is maintained by the manager as a reachability matrix $\mathbf{R}_k$, where r_{ij} indicates $t_i \triangleright t_j$, as shown in Figure 4. To begin with , when t_1 commits, $\mathbf{R}_1 = [1]$ since a vertex can always reach itself. When validating t, according to the Warshall's fact, a *proceeding vector* $\mathbf{p}$ where $\mathbf{p}[i]$ indicates $t \triangleright t_i$, can be calculated as $\mathbf{p}[i] = \mathbf{f}[i] \vee (\bigvee_{j \in \{1,...,k\}}(\mathbf{f}[j] \wedge r[i][j]))$. Dually, a *succeeding vector* $\mathbf{s}$ where $\mathbf{s}[i] = \mathbf{b}[i] \vee (\bigvee_{j \in \{1,...,k\}}(\mathbf{b}[j] \wedge r[j][i]))$ can be calculated. If $\vee$ and $\wedge$ are viewed as $+$ and $\times$ on boolean algebra, then $\mathbf{s} = \mathbf{b} + \mathbf{R}_k \times \mathbf{b}$ and $\mathbf{p} = \mathbf{f} + \mathbf{R}_k^T \times \mathbf{f}$. According to the fact of cycle detection, if $\mathbf{p} \wedge \mathbf{s} \neq 0$, a cycle is detected.

If no cycle is detected, t can commit as t_{k+1}. Then $\mathbf{R}_k$ must be updated to the transitive closure $\mathbf{R}_{k+1}$ of $\{T_k, t_{k+1}\}$ for future validations, as shown in figure 4 (b). For this purpose, $\mathbf{p}$ and $\mathbf{s}$ are appended to $\mathbf{R}_k$ as the $(k+1)^{th}$ row and column, respectively. And the previous $\mathbf{R}_k[i][j], (i, j \leq k)$ should be updated accordingly. Considering the transitivity of $(\{T_k, t_{k+1}\}, \triangleright)$, we can see $t_i \triangleright t_j$ if $(t_i \triangleright t_{k+1}) \wedge (t_j \triangleleft t_{k+1})$, which is equivalent to $(\mathbf{p}[i] \wedge \mathbf{s}[j])$. Since

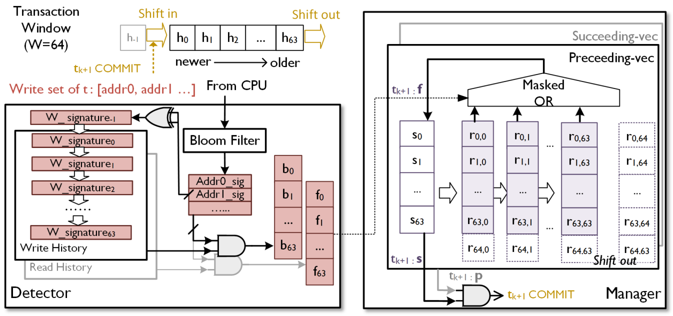


Figure 5: Pipelined ROCoCo on FPGA with sliding window of transactions ($W = 64$)

all bit manipulations in figure 4 are parallel, ROCoCo can validate $(T, \to_{rw})$ with $|T|$ iterations of the above-mentioned procedure.

A deficiency in ROCoCo is that it greedily commits a transaction if it does not cause cycles with regard to previous transactions, without considering future transactions. There exists cases in which committing a transaction may cause more future transactions to abort. Optimizations on ROCoCO are possible if the validation phase has a global view.

ROCoCo allows more transactions to commit compared to previous algorithms based on topological sorting. Although Kahn's topological sorting has running time linear to the number of transactions plus the number of relation pairs, it suffers the phantom ordering since it presumes a linear order on a DAG during its traversal, which contradicts the intuition that the transitive closure of a DAG can be disjoint sub-graphs if the DAG is disjoint.

4.2 ROCoCo on FPGA

ROCoCo algorithm exploits bit-level parallelism to provide $O(1)$ validation for each transaction. However, commodity CPUs are inefficient in bit manipulation. In particular, computation in Figure 4 (a) involves transposition of bit matrices $\mathbf{R}_k$, which is time-consuming in RAM-based architectures. Thus, ROCoCo is impractical on commodity CPUs with SIMD instructions.

FPGAs arise as the Dues Ex Machina to our bit-intensive algorithm. Commercial FPGAs provide massive number of LUTs, registers, BRAMs and DSPs to exploit various forms of parallelism, ranging from bit-level to task-level parallelism [65]. Moreover, emerging CPU-FPGA platforms are equipped with low-latency interconnections [11] to accommodate fine-grained CPU-FPGA interactions. These heterogeneous systems match the requirements of ROCoCo.

When implementing the ROCoCo algorithm on FPGA, a problem appears as there is no bound on the number of transactions, even though $|T|$ is finite. Since hardware resource has to be bounded, we employ the *sliding window of transactions* in our implementation, a technique adapted from task window [31, 39, 64]. Our insight for this technique is that transactions relying on distant snapshots are prone to abort. Thus, we only provide serializability for a sliding window of W transactions with ROCoCo algorithm on FPGA. When transaction t_{k+1} has committed, transactions that neglect updates of t_{k-W} abort. In this way, the FPGA only need to keep records for W transactions. To ensure long transactions can eventually commit, irrevocability may be required. In our evaluation, $W = 64$ is chosen as we spawn at most 28 threads in current ROCoCoTM on Intel HARP2.

Figure 5 shows a pipelined ROCoCo implementation. Before a transaction is allowed to commit as t_{k+1}, its bloom-filter signature and succeeding/proceeding vector are stored as bookkeeping h_{-1}. The oldest bookkeeping h_{63} for t_{k-63} is discarded when t_{k+1} commits, as shown in the top left corner of Figure 5. A conflict detector, as shown in the left part of Figure 5, detects conflicts between incoming transaction t's read/write addresses and historical signatures. The design choice for the signature will be detailed in section 4.2. The reachability matrix is implemented as 2D-registers in the manager. When t_{k+1} commits, the 2D-registers shift and update accordingly.

Although multiple transactions are concurrently validated on FPGA, the atomicity of validation phase can still be guaranteed. When a transaction is allowed to commit, the following transactions will observe this effect immediately (broadcast of the signal "t_{k+1} commit" in Figure 5) and act accordingly. Speculative conflict detection to previously undecided transactions in the detector is needed, in order to ensure reactions to commits are finished in one clock cycle. In this way, each transaction commits atomically, while a non-blocking pipeline is maintained.

5 IMPLEMENTATION OF ROCOCOTM

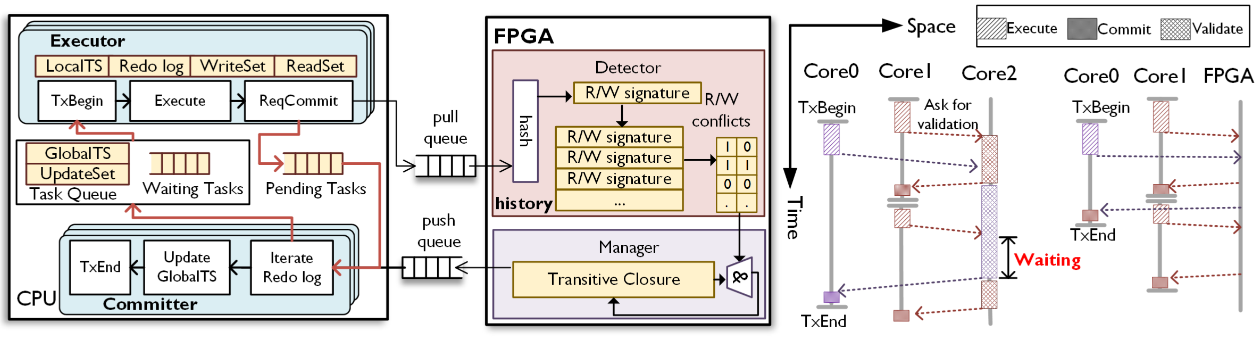
In this section, we present ROCoCoTM, a TM based on the ROCoCo algorithm. ROCoCoTM can be deployed on tightly coupled CPU-FPGA architectures. An overview of ROCoCoTM is presented in section 5.1, which highlights important design choices involved in offloading the validation phase to FPGA. Design choices for the bloom-filter signature and the algorithm on CPU-side are elaborated in section 5.2 and section 5.3, respectively.

5.1 ROCoCoTM Overview

ROCoCoTM features a hierarchical *meta-pipeline* in which each stage can be further composed of pipelines [51]. Figure 6 (a) and (b) show how transactions flow through the architectural meta-pipeline across CPU and FPGA. Transactions are first executed (**Executor** in Figure 6 (a)) and then committed (**Committer**) on CPU, while conflicts are detected (**Detector** in Figure 6 (b)) and managed (**Manager**) by a pipeline on FPGA. These stages of transactional execution are cascaded via two asynchronous message queues (the pull/push queue in Figure 6) between CPU and FPGA to form a meta-pipeline so that communication latency may be amortized by overlapped transactions.

An out-of-core FPGA is deployed for the detection of conflicts and the respective validation as discussed in section 4.2. Both Detector and Manager are fully-pipelined without back pressure on the pull queue to avoid stalls of transactional execution on CPU-side. Compared to the exclusive validation on a dedicated thread (figure 6 (c)) in a previous centralized validation scheme [45], pipelined validation on FPGA can significantly reduce the amortized validation overhead per transaction, as shown in figure 6 (d).

To decouple validation from other steps of transactional processing, ROCoCoTM adopts a lazy version management strategy and a hybrid conflict detection scheme, i.e. eager conflict detection



(a) ROCoCoTM on CPU (b) ROCoCo on FPGA (c) Exclusive Validation (d) Pipelined Validation

Figure 6: Transactions flow through the architectural meta-pipeline across CPU and FPGA in ROCoCoTM. The timing diagram for validating transactions with an exclusive core and a hardware pipelined module are compared.

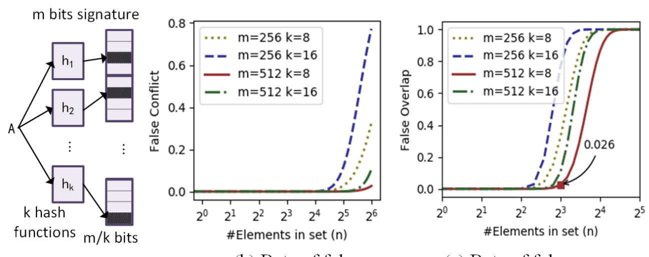
by CPU and lazy detection by Detector on FPGA purely based on global metadata. A thread can serve as either an executor or a committer based on the status of the transaction. During execution, a transaction bookkeeps its read/write addresses and tentative writes in private R/W-set and redo-log, respectively. It sends R/W-set to FPGA and waits for verdict when the execution finishes. If approved, the committer iterates the redo-log and updates the actual locations. To ensure that the read-set of each transaction remains consistent during execution, a transaction intersects its read-set signature against write-set signatures of committed transactions and may abort when read-after-write conflicts are eagerly detected. In this way, fast paths for detecting true conflicts between executing transactions and committed transactions is constructed without any atomic operation. These transactions aborts fast without incurring the out-of-core latency. ROCoCoTM prefers global metadata (bloom-filter signatures) to per-location metadata (locks or timestamps per object) so that FPGA could detect conflicts without incurring the latency of accessing memory locations. The detailed algorithm on CPU-side will be discussed section 5.3.

The current implementation of ROCoCoTM supports serializability among 64 transactions in the sliding window on FPGA. For other pairs of transactions, strict serializability is guaranteed by eager conflict detection and early abort on CPU. Since most transactions that violate timestamped order re-arrange transactions within 64 transactions incurred by 28 concurrent threads, ROCoCoTM's semantic allow more transactions to commit than TOCC.

As for the forward progress, the deadlock-freedom is guaranteed by commit-time locking such that no shared resources are locked by any transaction before committing. However, ROCoCoTM cannot guarantee livelock-freedom in two cases: 1) when the number of threads T exceeds the size of sliding window W, a long-running transaction may be continuously aborted due to the overflow of the sliding window; 2) when $T <= W$, a skewed workload with extreme long-running transactions flooded by an overwhelming number of tiny transactions may suffer from livelock. Under current configuration ($T = 28, W = 64$), no livelock is observed.

5.2 Parallel Bloom-Filter Signatures

Parallel (partitioned) bloom filters [58] have been widely used in HTMs to compute signatures representing an unbounded set of elements. The problem of such an approach is the introduction of false positivity. A signature is computed by using k parallel filters, each of which hashes an element to one of m/k bits, as shown in Figure 7 (a). It supports element insertion, membership query, set union, and set intersection operations with bit-wise operators [9].



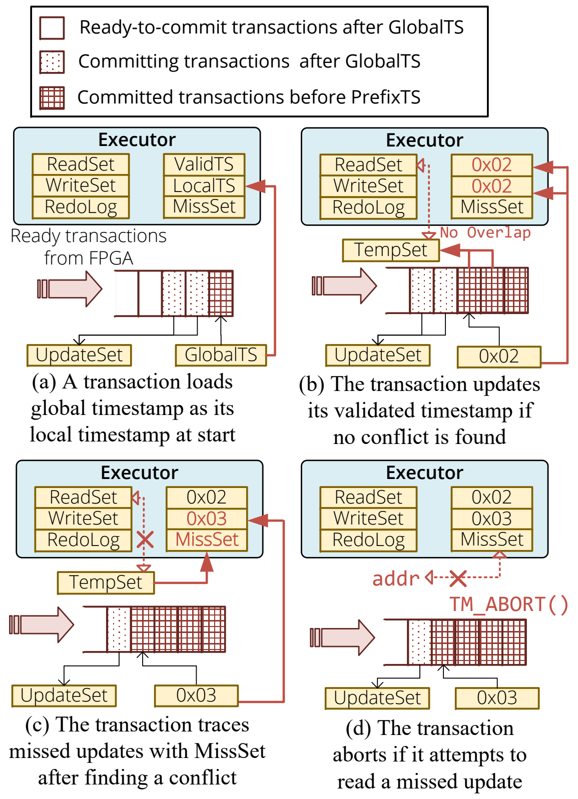
(a) A parallel bloom filter (b) Rate of false conflict for Query (c) Rate of false overlap for Intersection

Figure 7: False positivity of query and set intersection of bloom-filter signatures under different parameters

For ROCoCoTM, parallel bloom-filter signatures would be implemented with both hardwired logic on FPGA and AVX instructions on CPU. To adapt to different architectures, we choose the approximated universal hashing with the multiply-shift scheme [15]. In this way, the signature of an address can be computed with several AVX instructions.

A major concern for designing the bloom-filter signature is to decide appropriate sizes of signatures so that an acceptable false positive rate can be attained. We use an established probabilistic model [30] to analyse the false positivity of query and set intersection[6] of bloom-filter signatures. The respective false positivity of an bloom-filter signature storing n elements are depicted in Figure 7 (a) and (b). Here m and k are chosen to ease the implementation with AVX2 instructions on CPUs. False set-overlap for set intersections can be frequent even with a small number of elements. To lower the false positivity, we choose $m = 512, n = 8$ for ROCoCoTM and only perform intersection operations on signatures with at most 8

[6]Here we concern false set-overlap of intersection, i.e. a bitwise AND of two signatures of disjoint sets indicates an non-empty intersection



Ready-to-commit transactions after GlobalTS
Committing transactions after GlobalTS
Committed transactions before PrefixTS
Executor
ReadSet
WriteSet
RedoLog
ValidTS
LocalTS
MissSet
Ready transactions from FPGA
UpdateSet
GlobalTS
(a) A transaction loads global timestamp as its local timestamp at start
0x02
No Overlap
TempSet
(b) The transaction updates its validated timestamp if no conflict is found
0x03
(c) The transaction traces missed updates with MissSet after finding a conflict
addr
TM_ABORT()
(d) The transaction aborts if it attempts to read a missed update

Figure 8: LSA for strict serializability based on bloom-filter signatures

elements. Coincidentally, each 512-bit cacheline can store exactly eight 64-bit addresses.

5.3 Algorithms on CPU-side

To decouple validation while maintaining *opacity*[7] [19] on CPU-side, ROCoCoTM performs eager conflict detection to committed transactions by adapting LSA from TinySTM [17]. Unlike TinySTM where per-location metadata are required, ROCoCoTM relies on thread-local bloom-filter signatures to detect collisions. In what follows, all set operations are based on bloom-filters.

The gist of the algorithm is that each transaction incrementally maintains a valid snapshot of memory states by comparing its read set to write sets of committed transactions. As shown in Figure 8 (a), each committed transaction creates a snapshot indexed by an incremental global timestamp (GlobalTS). An executing transaction initially acquires the GlobalTS as its local timestamp (LocalTS) and validated timestamp (ValidTS). When the transaction reads an address, it copies the signatures of the write sets between its LocalTS and current GlobalTS as a temporary set (TempSet), and attempts to extend ValidTS by intersecting its read set to TempSet. If no overlap is found, it updates the ValidTS to indicate that no collision has been found before ValidTS (Figure 8 (b)). Otherwise, one of addresses in its read set might have been updated after

[7]A transaction's read-set must stay consistent during its execution

Algorithm 1 Transactional Load and Store in ROCoCoTM

```
Procedure TM_READ(var, addr)
1:  if WriteSet.Query(addr) then
2:      var = RedoLog[addr]
3:      return
4:  end if
5:  while UpdateSet.Query(addr) do
6:      if MissSet != ∅ then TM_ABORT() else back_off()
7:  end while
8:  v = Memory[addr]
9:  TempSet = ∅
10: while LocalTS < GlobalTS.acquire() do
11:     TempSet.unite(CommitQueue[LocalTS])
12:     LocalTS = LocalTS + 1
13: end while
14: if MissSet != ∅ || ReadSet.Interset(TempSet) != ∅ then
15:     MissSet = MissSet.Unite(TempSet)
16:     if MissSet.Query(addr) then
17:         TM_ABORT()
18:     end if
19: end if
20: ReadSet.Insert(addr)
Procedure TM_WRITE(var, addr)
21: WriteSet.Insert(addr)
22: RedoLog.Insert(var, addr)
```

ValidTS. In this case, it copies current TempSet and future write sets of committed transactions to a miss set (Figure 8 (c)) for missed updates since ValidTS. If a transaction reads an address in miss set, it cannot maintain a valid snapshot and should abort (Figure 8 (d)).

Algorithm 1 shows the detailed algorithm for transactional read and write during the execution phase. Different conditions of line 14 corresponds to various scenarios in Figure 8. Since the set intersection on bloom-filter signatures features a sharp rise of false positivity after recording eight elements, the read set summarizes a signature for every subset of eight addresses. If the signature of the whole read set overlaps with TempSet, the transaction iterates signatures in each sub-set for more accurate intersection with TempSet. In this way most conflict resolution are conducted with an time complexity of $O(1)$. There is only a small chance of an $O(r)$ (r is the size of read-set) overhead when the miss set is empty and the intersection between TempSet and the read set is non-empty.

When a transaction finishes execution, a read-only transaction will be committed directly. On the other hand, a transaction with a non-empty write set has to be validated by out-of-core FPGA to ensure transactional semantics with regard to the other transactions. A transaction sends its read set and write set along with the ValidTS to the FPGA and waits for validation. A transaction's read set and write set are transferred in terms of address rather than signature, so that the query operation on signatures can be used to minimize the possibility of false positivity of conflict detection on FPGA. The FPGA records the history of committed transactions with two signatures (one for read set and the other for write set) per transaction so that an upper bound of required resources can be determined a priori.

When the FPGA decides that a transaction can be committed, it pushes the ready-to-commit transaction into a queue. Before committing, a transaction publishes its signature to the update set (Figure 8 (a)), which serves as a commit-time locking on updating addresses (line 5 of Algorithm 1). Therefore, isolation between committing and executing transactions is preserved, since no partial results of committing transactions are observed. Multiple transactions can be committed concurrently, as long as no write-write conflicts are detected among them by FPGA. A transaction will increment and release the GlobalTS after commitment. Thus, atomicity on the CPU-side is guaranteed since a transaction's updates only become visible to other transactions after it commits.

This CPU-side design is specialized for speculation in loop parallelization, which is the programming model used in STAMP benchmark suites. As a result, the need for strong atomicity [20] is obviated since all codes in parallelized loops could run inside transactions. Modifications such as stall before non-transactional codes can be added to ROCoCoTM if strong atomicity is preferred. Nevertheless, the ROCoCo algorithm on FPGA-side is general enough to accommodate all programming models of TM.

6 EVALUATION

In this section, we report the evaluation results of ROCoCoTM. The experiments are designed to answer the following questions.

(1) Does the ROCoCo algorithm attain a lower abort rate than mainstream CC algorithms (section 6.1)?
(2) Does ROCoCoTM perform better on the STAMP benchmark suites on commodity systems than established STMs and HTMs (section 6.2 and section 6.3)?
(3) Does the centralized validation mechanism on FPGA constitute a bottleneck of ROCoCoTM (section 6.4)?
(4) What is the overhead of ROCoCoTM in terms of FPGA resources (section 6.5)?

6.1 Micro-Benchmark for CC Algorithms

To isolate the impacts of the concurrency control from those of other components in a TM system, we use memory traces extracted from a simple synthetic micro-benchmark similar to EigenBench [24] to test three concurrency control algorithms: 2PL, TOCC, and the ROCoCo algorithm proposed in this work. The functionality of the micro-benchmark is to emulate transactions with a random set of array accesses. The array consists of 1024 memory locations. Each transaction accesses N memory locations with 50% read and 50% write. Given every two transactions, the possibility of at least one collision on memory locations is $(1 - (1 - N/1024)^N)$. In this experiment, we set $N = 4, 8, 12, \ldots 32$, corresponding to a collision rate of 1.5%-63.8%. Fifty traces are generated with varying random seeds for each collision rate. Then we assume T transactions are executed concurrently. In other words, the tentative updates of the last T transactions, no matter they commit or not, are not visible to current transactions. We test two cases, $T = 4$ and $T = 16$, to approximate four and sixteen concurrent threads, respectively.

Figure 9 shows the experimental results. Compared with 2PL and TOCC, ROCoCo achieves the lowest abort rate in all scenarios. In the 4-thread cases, ROCoCo is only slightly better (a reduction of up to 8.6%) than TOCC, since the possibility for a transaction to

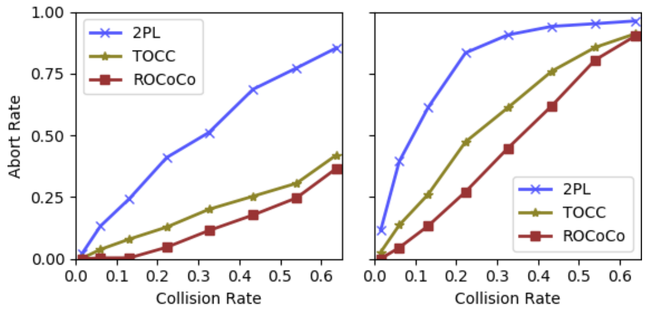


Figure 9: Abort rate to collision rate of the three CC algorithms for different number of concurrent threads, i.e. $T = 4$ (left) and $T = 16$ (right)

neglect the updates of concurrent transactions (i.e. write-after-read dependency in section 3.1) is low. Hence, fewer transactions need to be rearranged against the timestamped order in 4-thread cases than 16-thread cases. Nevertheless, both OCC algorithms enables considerable reduction of the abort rate over 2PL. With the 16-thread simulation targeting a modest number of cores, ROCoCo shows up to 56.2% and 20.2% lower aborts with regard to 2PL and TOCC, respectively, at a collision rate of 22.3%. Thus, under low or medium collision rate with a high amount of concurrency, over-serialization introduced by the timestamped order of strict serializability is not negligible for the OCC algorithm, which can be effectively circumvented by the ROCoCo algorithm. As the collision rate increases to over 50%, the possibility of forming dependence cycles among 16 threads rises abruptly and the three algorithms perform similarly.

6.2 Experimental Setup on HARP2

We evaluate the performance of ROCoCoTM with STAMP benchmarks running on the Intel HARP2 Platform [26], an experimental server featuring in-package integration of a 2.4 GHz 14-core Haswell Xeon processor (at most 28 concurrent threads with hyper-threading) and an Arria 10 FPGA (10AX115U3F45E2SGE3). The processor and FPGA are interconnected with a CCI (Cache-Coherent Interface) interface [27], which greatly reduces the communication latency and increases the bandwidth. Both CPU and FPGA share the Last-Level Cache (LLC) of CPU. The round-trip latency of transferring a cacheline between CPU and FPGA is measured to be less than 600 ns[8] through the low-latency channel built upon QPI, which is several orders of magnitude smaller than the latency of FPGA as discrete PCIe accelerating card. Such in-package integrated platforms are preferable for applications with fine-grained CPU-FPGA interactions such as TM systems.

For the STAMP benchmark suite, we choose TinySTM (v1.0.4) as the baseline STM, as it outperforms a number of STM and HTM systems on STAMP in recent evaluations [14, 48]. We use a configuration that is similar to ROCoCoTM, which is commit-time locking (lazy conflict detection) with write-back of tentative states on commit (lazy version management). Evaluations of TinySTM on HARP2 show no significant difference between commit-time

[8]Around 200 ns for read-hit to LLC on FPGA, and less than 400 ns for write back to LLC from FPGA. In contract, the de facto PCIe interconnect for ASIC accelerators incur a round-trip latency of over 1us.

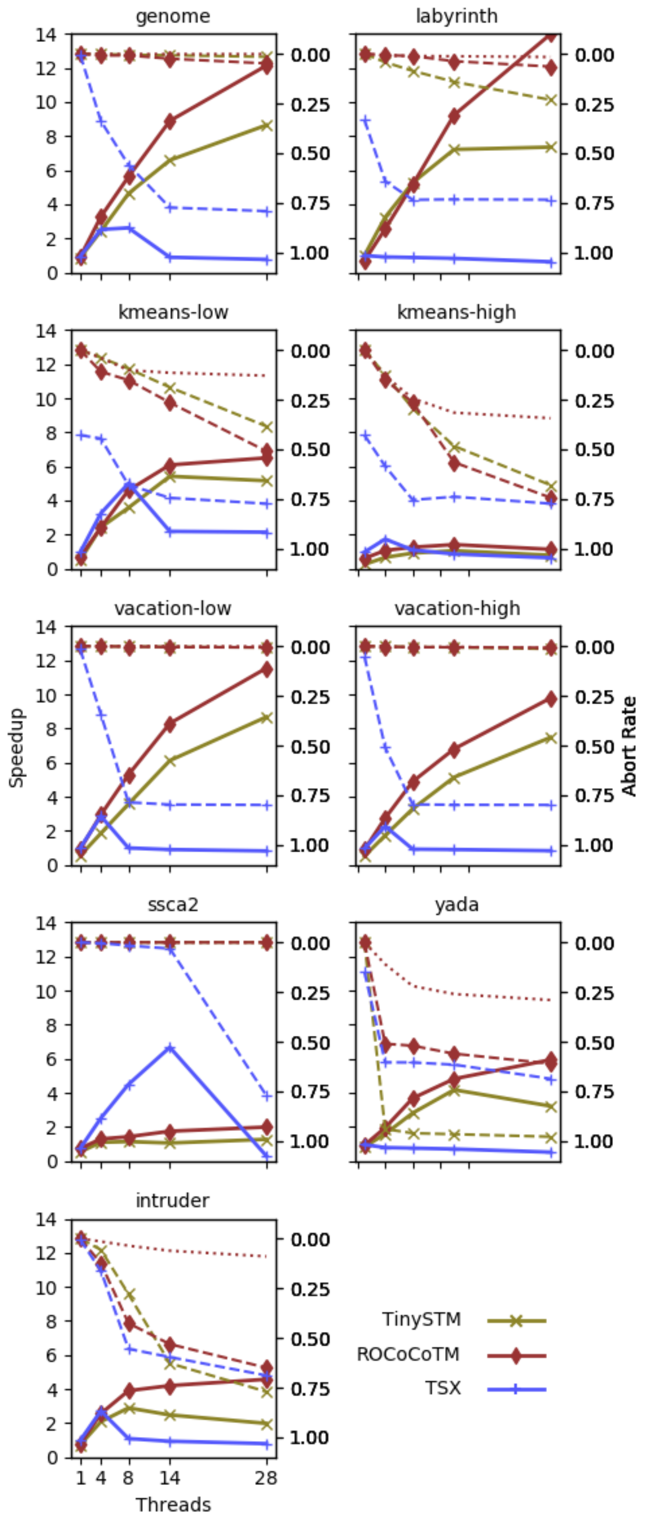


Figure 10: Speedup (solid line, left y-axis) with regard to sequential execution and abort rate (dashed line, right y-axis) under the varying number of threads ({1, 4, 8, 14, 28} on the x-axis) in the STAMP benchmark suite. Abort rates attributed to FPGA in ROCoCoTM is drawn as dotted lines.

locking and the default encounter-time locking. We implement an HTM based on Intel TSX. As TSX will trigger abort under various indeterministic micro-architectural conditions, a transaction will fall back to use a global lock if it aborts for too many times. This fallback path is necessary due to the best-effort nature of TSX. Individually tuning the retry policy for each benchmark will result in better performance, but it is impractical for common cases. So we stay with a constant retry policy and find that the 4-time retry performs best on HARP2. All benchmark programs are compiled with GCC (v5.4.0) -O3 and tested with the respective largest input dataset.

6.3 Performance on STAMP

Figure 10 shows speedups and abort rates of executing STAMP benchmark applications with TinySTM, TSX and ROCoCoTM, respectively. Speedups depicted on the left y-axis are relative to the sequential baseline of STAMP. Abort rates are calculated as the ratio of the number of aborted transactions over the total number of executed transactions, which is depicted on the right y-axis (note the y-axis for the abort rate is inverted). To show how out-of-core validation contributes to aborts in ROCoCoTM, the abort rate on FPGA-side is explicitly depicted as dotted lines. We exclude the bayes benchmark due its high variability.

The first observation is that ROCoCoTM scales better than TinySTM and TSX. In the case of 14-thread[9], ROCoCoTM outperforms TinySTM and TSX by 1.41x and 4.04x in geomean, respectively. While for 28-thread, ROCoCoTM achieves a geomean speedup of 1.55x and 8.05x to TinySTM and TSX, respectively. Considering these two cases, it seems that the centralized validation mechanism on FPGA has less impact on performance than the cache thrashing due to hyper-threading.

For TSX, we can observe the avalanche of aborts when concurrency reaches certain thresholds for all applications[10], especially for 28-thread ssca2. This can be attribute to TSX's eager conflict detection and eager version management, where an aborted transactions will cause more transactions to abort in a chain. Although TSX attains the best performance at 4-thread cases, due to the booming abort rate, it performance fails to scale up to 28 threads.

On the other hand, when considering the addition of cache thrashing from 14 threads to 28 threads, ROCoCoTM can scale better than TinySTM since it exhibits smaller memory footprints with the bloom-filter signatures. In some cases ROCoCoTM and TinySTM exhibits similar abort rate since ROCoCoTM adopts the TSA algorithm from TinySTM on CPU-side. As indicated by the dotted line for aborted transactions on FPGA-side, most aborts of ROCoCoTM fails fast on CPU, without going through the validation process on FPGA. Besides, transactions with empty write sets, which accounts for a large percentage of transactions in genome and intruder, will commit directly on CPU-side. There optimizations successfully reduce performance overhead of ROCoCoTM in there cases where ROCoCoTM and TinySTM have similar abort rate .

[9]We have to alter the default log2 barrier of original STAMP to a pthread barrier to enable 14- and 28-thread tests. Such modification leads to slightly and similar performance loss for all TMs at 8-thread.

[10]the ceiling of TSX abort rate is 83.3% since each transaction retries at most 4 times before acquiring a global lock as a fallback

Particularly, the workloads of labyrinth and yada are considered more transaction-friendly, since they involve pointer-chasing on concurrent data structures with non-negligible conflict rates. These conflicts can only resort to transactions. Whereas conflicts induced by sharing atomic counters or dynamic buffers in kmeans and intruder can be resolved by other programming constructs [48]. ROCoCoTM achieves noteworthy lower abort rate than that of TinySTM in labyrinth and yada. In this way ROCoCoTM outperforms TinySTM on high-contentious applications with reduced abort rates.

Another point to note is that ROCoCoTM incurs a higher performance penalty than TinySTM in 1-thread case. For 1-thread applications, TinySTM outperforms ROCoCoTM by a factor of 1.32x. This is because TinySTM never needs to resolve a conflict or validate a transaction with only one thread. As ROCoCoTM still has to validate read-write transactions by out-of-core FPGA, the communication latency dominates the overhead. However, the overhead becomes less prominent as the number of transaction increases. With an increased number of threads, concurrent transactions will trigger the conflict resolution and validation much more frequently for both TinySTM and ROCoCoTM. Since ROCoCoTM resolves conflicts by bloom-filter signatures and offloads the validation process to FPGA, the amortized overhead seems to be lower than that of TinySTM in cases of 14 and 28 threads.

One exception to this trend is ssca2, which features an enormous number of short transactions with low contention rate. Accordingly, its scalability is limited by the overhead of maintaining transactions rather than the contentious memory accesses. While ROCoCoTM maintains less metadata, its amortized overhead for each transaction is increased by out-of-core communications. Such an overhead makes ssca2 scales poorly on ROCoCoTM.

6.4 Amortized Overhead of Validation

To quantitatively study the potential bottleneck of centralized validation, we instrumented the TinySTM and ROCoCoTM to report the time spent in validation. Since TinySTM is configured to be commit-time locking, a dedicated validation phase before transactions' commit is required, where the CPU goes over all timestamped objects in read set. The amortized validation overheads per transaction for some benchmarks are shown in figure 11. The per-transaction overhead stays below one micro-second for all cases in ROCoCoTM. Thus, we can safely conclude that the centralized validation bottleneck can be avoided by pipelining on FPGA. On the other hand, with its huge read set during execution, labyrinth incurs a larger amortized overhead on TinySTM than that of ROCoCoTM. With a hardwired bloom-filter based validate mechanism, the amortized overhead of ROCoCoTM is insensitive to the size of read set.

6.5 Resource Consumption on FPGA

In the current implementation of ROCoCoTM on HARP2, its FPGA component is clocked at 200MHz, where the 512-bit bloom filter is the critical path. Since ROCoCoTM are fully-pipelined, it consumes 113485 (62.9%) registers, 249442 (58.39%) ALMs, 223 (14.7%) DSPs (for hashing), and only 2055802 (3.7%) BRAM bits. These BRAM bits are mainly used for storing historical signatures.

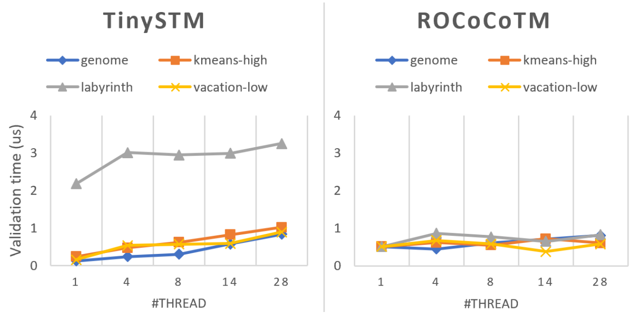


Figure 11: Per-transaction validation overhead measured in micro-second for some STAMP benchmarks.

More accurate bloom-filter is possible under current resource consumption. However, even though we extend the bloom-filter signatures to 1024-bit at the cost of lower clock frequency, no noteworthy improvement on the abort rate can be observed, which proves that our quantitative analysis on bloom filter stands. With the incoming Stratix 10 FPGA devices featuring low cost pipeline registers [38], ROCoCoTM may operate at higher frequency, leading to lower out-of-core validation overhead.

7 CONCLUSION

In this paper we study transactional semantics and CC algorithms with a new formalization based on the order theory. Based on this formalization, we prove mainstream OCC algorithms can at best provide strict serializability and propose a centralized OCC algorithm named ROCoCo. We implement ROCoCoTM, a hybrid TM with out-of-core hardware validation, for fast-growing CPU-FPGA platforms,.

This work can be extended in several directions. From a programming perspective, there are still a vast number of in-between semantics besides what we list in Figure 3. For example, there are many rigorously proved relaxed consistency models residing between quiescent and sequential consistency in multi-processor designs. It is appealing to formalize these semantics with the axiom-based semantics to provide insights for other transactional systems. At the algorithm level, it is intriguing to study non-greedy CC algorithms for the avoidance of sub-optimal aborts. At the implementation level, it is worth trying to apply ROCoCo to transactional systems with a centralized control unit, such as directory-based HTMs.

8 ACKNOWLEDGEMENT

We thank the anonymous reviewers and the shepherd for their thoughtful comments. This work is supported in part by the National Science and Technology Major Project of the Ministry of Science and Technology of China(Grant No. 2018ZX01028201), and in part by the National Natural Science Foundation of China (Grant No. 61672317, No.61834002), and in part by the National Key R&D Program of China (Grant No. 2018YFB2202101). The presented HARP-2 results were obtained on resources hosted at the Paderborn Center for Parallel Computing (PC^2) in the Intel Hardware Accelerator Research Program (HARP2).

REFERENCES

[1] Ali-Reza Adl-Tabatabai, Brian T. Lewis, Vijay Menon, Brian R. Murphy, Bratin Saha, and Tatiana Shpeisman. 2006. Compiler and Runtime Support for Efficient Software Transactional Memory. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '06)*. ACM, New York, NY, USA, 26–37. https://doi.org/10.1145/1133981.1133985

[2] J. Bobba, N. Goyal, M. Hill, M. Swift, and D. Wood. 2008. TokenTM: Efficient Execution of Large Transactions with Hardware Transactional Memory. In *35th International Symposium on Computer Architecture, 2008. ISCA '08*. 127–138. https://doi.org/10.1109/ISCA.2008.24

[3] Jayaram Bobba, Kevin E. Moore, Haris Volos, Luke Yen, Mark D. Hill, Michael M. Swift, and David A. Wood. 2007. Performance Pathologies in Hardware Transactional Memory. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 81–91. https://doi.org/10.1145/1250662.1250674

[4] Michael J. Cahill, Uwe Röhm, and Alan D. Fekete. 2008. Serializable Isolation for Snapshot Databases. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 729–738. https://doi.org/10.1145/1376616.1376690

[5] Brian D. Carlstrom, Austen McDonald, Michael Carbin, Christos Kozyrakis, and Kunle Olukotun. 2007. Transactional collection classes. In *Proceedings of the 12th ACM SIGPLAN symposium on Principles and practice of parallel programming - PPoPP '07 (PPoPP '07)*. ACM Press, San Jose, California, USA, 56. https://doi.org/10.1145/1229428.1229441

[6] Calin Cascaval, Colin Blundell, Maged Michael, Harold W. Cain, Peng Wu, Stefanie Chiras, and Siddhartha Chatterjee. 2008. Software Transactional Memory: Why Is It Only a Research Toy? *Queue* 6, 5 (Sept. 2008), 40:46–40:58. https://doi.org/10.1145/1454456.1454466

[7] Jared Casper, Tayo Oguntebi, Sungpack Hong, Nathan G. Bronson, Christos Kozyrakis, and Kunle Olukotun. 2011. Hardware Acceleration of Transactional Memory on Commodity Systems. In *Proceedings of the Sixteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS XVI)*. ACM, New York, NY, USA, 27–38. https://doi.org/10.1145/1950365.1950372

[8] A. M. Caulfield, E. S. Chung, A. Putnam, H. Angepat, D. Firestone, J. Fowers, M. Haselman, S. Heil, M. Humphrey, P. Kaur, J. Y. Kim, D. Lo, T. Massengill, K. Ovtcharov, M. Papamichael, L. Woods, S. Lanka, D. Chiou, and D. Burger. 2017. Configurable Clouds. *IEEE Micro* 37, 3 (2017), 52–61. https://doi.org/10.1109/MM.2017.51

[9] Luis Ceze, James Tuck, Josep Torrellas, and Calin Cascaval. 2006. Bulk Disambiguation of Speculative Threads in Multiprocessors. In *Proceedings of the 33rd Annual International Symposium on Computer Architecture (ISCA '06)*. IEEE Computer Society, Washington, DC, USA, 227–238. https://doi.org/10.1109/ISCA.2006.13

[10] Sui Chen, Lu Peng, and Samuel Irving. 2017. Accelerating GPU Hardware Transactional Memory with Snapshot Isolation. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 282–294. https://doi.org/10.1145/3079856.3080204

[11] Young-kyu Choi, Jason Cong, Zhenman Fang, Yuchen Hao, Glenn Reinman, and Peng Wei. 2016. A Quantitative Analysis on Microarchitectures of Modern CPU-FPGA Platforms. In *Proceedings of the 53rd Annual Design Automation Conference (DAC '16)*. ACM, New York, NY, USA, 109:1–109:6. https://doi.org/10.1145/2897937.2897972

[12] Luke Dalessandro, Michael F. Spear, and Michael L. Scott. 2010. NOrec: Streamlining STM by Abolishing Ownership Records. In *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '10)*. ACM, New York, NY, USA, 67–78. https://doi.org/10.1145/1693453.1693464

[13] Diego Didona, Nuno Diegues, Anne-Marie Kermarrec, Rachid Guerraoui, Ricardo Neves, and Paolo Romano. 2016. ProteusTM: Abstraction Meets Performance in Transactional Memory. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 757–771. https://doi.org/10.1145/2872362.2872385

[14] Nuno Diegues, Paolo Romano, and Luís Rodrigues. 2014. Virtues and Limitations of Commodity Hardware Transactional Memory. In *Proceedings of the 23rd International Conference on Parallel Architectures and Compilation (PACT '14)*. ACM, New York, NY, USA, 3–14. https://doi.org/10.1145/2628071.2628080

[15] Martin Dietzfelbinger, Torben Hagerup, Jyrki Katajainen, and Martti Penttonen. 1997. A Reliable Randomized Algorithm for the Closest-Pair Problem. *J. Algorithms* 25, 1 (Oct. 1997), 19–51. https://doi.org/10.1006/jagm.1997.0873

[16] Alan Fekete, Elizabeth O'Neil, and Patrick O'Neil. 2004. A Read-only Transaction Anomaly Under Snapshot Isolation. *SIGMOD Rec.* 33, 3 (Sept. 2004), 12–14. https://doi.org/10.1145/1031570.1031573

[17] P. Felber, C. Fetzer, P. Marlier, and T. Riegel. 2010. Time-Based Software Transactional Memory. *IEEE Transactions on Parallel and Distributed Systems* 21, 12 (Dec. 2010), 1793–1807. https://doi.org/10.1109/TPDS.2010.49

[18] Peter C. Fishburn. 1985. *Interval orders and interval graphs: a study of partially ordered sets*. Wiley. Google-Books-ID: DuSAAAAAIAAJ.

[19] Rachid Guerraoui and Michal Kapalka. 2008. On the correctness of transactional memory. ACM Press, 175. https://doi.org/10.1145/1345206.1345233

[20] Tim Harris, James Larus, and Ravi Rajwar. 2010. *Transactional Memory, 2Nd Edition* (2nd ed.). Morgan and Claypool Publishers.

[21] Maurice Herlihy and J. Eliot B. Moss. 1993. Transactional Memory: Architectural Support for Lock-free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture (ISCA '93)*. ACM, New York, NY, USA, 289–300. https://doi.org/10.1145/165123.165164

[22] Maurice Herlihy and Nir Shavit. 2008. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA.

[23] Maurice P. Herlihy and Jeannette M. Wing. 1990. Linearizability: A Correctness Condition for Concurrent Objects. *ACM Trans. Program. Lang. Syst.* 12, 3 (July 1990), 463–492. https://doi.org/10.1145/78969.78972

[24] Sungpack Hong, T. Oguntebi, J. Casper, N. Bronson, C. Kozyrakis, and K. Olukotun. 2010. Eigenbench: A simple exploration tool for orthogonal TM characteristics. In *2010 IEEE International Symposium on Workload Characterization (IISWC)*. 1–11. https://doi.org/10.1109/IISWC.2010.5648812

[25] Theo Härder. 1984. Observations on Optimistic Concurrency Control Schemes. *Inf. Syst.* 9, 2 (Nov. 1984), 111–120. https://doi.org/10.1016/0306-4379(84)90020-6

[26] Intel Corporation. 2015. Xeon+fpga platform for the data center. https://www.ece.cmu.edu/~calcm/carl/lib/exe/fetch.php?media=carl15-gupta.pdf

[27] Intel Corporation. 2017. Intel® FPGA IP Core Cache Interface (CCI-P) Specification.

[28] Christian Jacobi, Timothy Slegel, and Dan Greiner. 2012. Transactional Memory Architecture and Implementation for IBM System Z. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-45)*. IEEE Computer Society, Washington, DC, USA, 25–36. https://doi.org/10.1109/MICRO.2012.12

[29] Syed Ali Raza Jafri, Gwendolyn Voskuilen, and T. N. Vijaykumar. 2013. Wait-n-GoTM: Improving HTM Performance by Serializing Cyclic Dependencies. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '13)*. ACM, New York, NY, USA, 521–534. https://doi.org/10.1145/2451116.2451173

[30] Mark C. Jeffrey and J. Gregory Steffan. 2011. Understanding bloom filter intersection for lazy address-set disambiguation. In *Proceedings of the 23th ACM Symposium on Parallelism in Algorithms and Architectures (SPAA '11)*. ACM Press, 345. https://doi.org/10.1145/1989493.1989551

[31] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A Scalable Architecture for Ordered Parallelism. In *Proceedings of the 48th International Symposium on Microarchitecture (MICRO-48)*. ACM, New York, NY, USA, 228–241. https://doi.org/10.1145/2830772.2830777

[32] Arpit Joshi, Vijay Nagarajan, Marcelo Cintra, and Stratis Viglas. 2018. DHTM: Durable Hardware Transactional Memory. In *Proceedings of the 45th Annual International Symposium on Computer Architecture (ISCA '18)*. IEEE Press, Piscataway, NJ, USA, 452–465. https://doi.org/10.1109/ISCA.2018.00045 event-place: Los Angeles, California.

[33] A. B. Kahn. 1962. Topological Sorting of Large Networks. *Commun. ACM* 5, 11 (Nov. 1962), 558–562. https://doi.org/10.1145/368996.369025

[34] Sanidhya Kashyap, Changwoo Min, Kangnyeon Kim, and Taesoo Kim. 2018. A Scalable Ordering Primitive for Multicore Machines. In *Proceedings of the Thirteenth EuroSys Conference (EuroSys '18)*. ACM, New York, NY, USA, 34:1–34:15. https://doi.org/10.1145/3190508.3190510

[35] Martin Kleppmann. 2017. *Designing Data-Intensive Applications* (1 edition ed.). O'Reilly Media, Boston.

[36] H. T. Kung and John T. Robinson. 1981. On Optimistic Methods for Concurrency Control. *ACM Trans. Database Syst.* 6, 2 (June 1981), 213–226. https://doi.org/10.1145/319566.319567

[37] Yossi Lev, Victor Luchangco, Marek Olszewski, Virendra J Marathe, Mark Moir, and Dan Nussbaum. 2009. Anatomy of a Scalable Software Transactional Memory. In *TRANSACT '09: 4th Workshop on Transactional Computing*. 10.

[38] David Lewis, Gordon Chiu, Jeffrey Chromczak, David Galloway, Ben Gamsa, Valavan Manohararajah, Ian Milton, Tim Vanderhoek, and John Van Dyken. 2016. The Stratix™ 10 Highly Pipelined FPGA Architecture. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '16)*. ACM, New York, NY, USA, 159–168. https://doi.org/10.1145/2847263.2847267

[39] Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, Yao Wang, and Shaojun Wei. 2017. Aggressive Pipelining of Irregular Applications on Reconfigurable Hardware. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, Toronto, ON, Canada, 575–586. https://doi.org/10.1145/3079856.3080228

[40] Heiner Litz, David Cheriton, Amin Firoozshahian, Omid Azizi, and John P. Stevenson. 2014. SI-TM: Reducing Transactional Memory Abort Rates Through Snapshot Isolation. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 383–398. https://doi.org/10.1145/2541940.2541952

[41] Heiner Litz, Ricardo J. Dias, and David R. Cheriton. 2015. Efficient Correction of Anomalies in Snapshot Isolation Transactions. *ACM Trans. Archit. Code Optim.* 11, 4 (Jan. 2015), 65:1–65:24. https://doi.org/10.1145/2693260

[42] Xiaoyu Ma, Dan Zhang, and Derek Chiou. 2017. FPGA-Accelerated Transactional Execution of Graph Workloads. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 227–236. https://doi.org/10.1145/3020078.3021743

[43] Martin Kleppmann. 2014. Hermitage: Testing the "I" in ACID — Martin Kleppmann's blog. https://martin.kleppmann.com/2014/11/25/hermitage-testing-the-i-in-acid.html https://martin.kleppmann.com/2014/11/25/hermitage-testing-the-i-in-acid.html.

[44] Alexander Matveev and Nir Shavit. 2015. Reduced Hardware NOrec: A Safe and Scalable Hybrid Transactional Memory. In *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*. ACM, New York, NY, USA, 59–71. https://doi.org/10.1145/2694344.2694393

[45] Mojtaba Mehrara, Jeff Hao, Po-Chun Hsu, and Scott Mahlke. 2009. Parallelizing Sequential Applications on Commodity Hardware Using a Low-cost Software Transactional Memory. In *Proceedings of the 30th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '09)*. ACM, New York, NY, USA, 166–176. https://doi.org/10.1145/1542476.1542495

[46] Chi Cao Minh, JaeWoong Chung, C. Kozyrakis, and K. Olukotun. 2008. STAMP: Stanford Transactional Applications for Multi-Processing. In *2008 IEEE International Symposium on Workload Characterization*. 35–46. https://doi.org/10.1109/IISWC.2008.4636089

[47] Chi Cao Minh, Martin Trautmann, JaeWoong Chung, Austen McDonald, Nathan Bronson, Jared Casper, Christos Kozyrakis, and Kunle Olukotun. 2007. An Effective Hybrid Transactional Memory System with Strong Isolation Guarantees. In *Proceedings of the 34th Annual International Symposium on Computer Architecture (ISCA '07)*. ACM, New York, NY, USA, 69–80. https://doi.org/10.1145/1250662.1250673

[48] Donald Nguyen and Keshav Pingali. 2017. What Scalable Programs Need from Transactional Memory. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '17)*. ACM, New York, NY, USA, 105–118. https://doi.org/10.1145/3037697.3037750

[49] Christos H. Papadimitriou. 1979. The serializability of concurrent database updates. *J. ACM* 26, 4 (Oct. 1979), 631–653. https://doi.org/10.1145/322154.322158

[50] Keshav Pingali, Donald Nguyen, Milind Kulkarni, Martin Burtscher, M. Amber Hassaan, Rashid Kaleem, Tsung-Hsien Lee, Andrew Lenharth, Roman Manevich, Mario Méndez-Lojo, Dimitrios Prountzos, and Xin Sui. 2011. The Tao of Parallelism in Algorithms. In *Proceedings of the 32Nd ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '11)*. ACM, New York, NY, USA, 12–25. https://doi.org/10.1145/1993498.1993501

[51] Raghu Prabhakar, David Koeplinger, Kevin J. Brown, HyoukJoong Lee, Christopher De Sa, Christos Kozyrakis, and Kunle Olukotun. 2016. Generating Configurable Hardware from Parallel Patterns. In *Proceedings of the Twenty-First International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '16)*. ACM, New York, NY, USA, 651–665. https://doi.org/10.1145/2872362.2872415

[52] Andrew Putnam, Adrian M. Caulfield, Eric S. Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth, Gopal Jan, Gray Michael, Haselman Scott Hauck, Stephen Heil, Amir Hormati, Joo-Young Kim, Sitaram Lanka, James Larus, Eric Peterson, Simon Pope, Aaron Smith, Jason Thong, Phillip Yi, and Xiao Doug Burger. 2014. A Reconfigurable Fabric for Accelerating Large-scale Datacenter Services. In *Proceeding of the 41st Annual International Symposium on Computer Architecuture (ISCA '14)*. IEEE Press, Piscataway, NJ, USA, 13–24. http://dl.acm.org/citation.cfm?id=2665671.2665678

[53] X. Qian, B. Sahelices, and J. Torrellas. 2014. OmniOrder: Directory-based conflict serialization of transactions. In *2014 ACM/IEEE 41st International Symposium on Computer Architecture (ISCA)*. 421–432. https://doi.org/10.1109/ISCA.2014.6853223

[54] Hany E. Ramadan, Christopher J. Rossbach, and Emmett Witchel. 2008. Dependence-aware Transactional Memory for Increased Concurrency. In *Proceedings of the 41st Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 41)*. IEEE Computer Society, Washington, DC, USA, 246–257. https://doi.org/10.1109/MICRO.2008.4771795

[55] Torvald Riegel. 2006. Snapshot isolation for software transactional memory. In *In Proceedings of the First ACM SIGPLAN Workshop on Languages, Compilers, and Hardware Support for Transactional Computing (TRANSACT'06*.

[56] Wenjia Ruan, Trilok Vyas, Yujie Liu, and Michael Spear. 2014. Transactionalizing Legacy Code: An Experience Report Using GCC and Memcached. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 399–412. https://doi.org/10.1145/2541940.2541960

[57] B. Saha, A. r Adl-Tabatabai, and Q. Jacobson. 2006. Architectural Support for Software Transactional Memory. In *2006 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'06)*. 185–196. https://doi.org/10.1109/MICRO.2006.9

[58] Daniel Sanchez, Luke Yen, Mark D. Hill, and Karthikeyan Sankaralingam. 2007. Implementing Signatures for Transactional Memory. In *Proceedings of the 40th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO 40)*. IEEE Computer Society, Washington, DC, USA, 123–133. https://doi.org/10.1109/MICRO.2007.24

[59] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L Hudson, Katherine F Moore, and Bratin Saha. 2007. Enforcing Isolation and Ordering in STM. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*. Pages 78–88. https://doi.org/1250744

[60] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. 2008. Ordering-Based Semantics for Software Transactional Memory. In *Proceedings of the 12th International Conference on Principles of Distributed Systems (OPODIS '08)*. Springer-Verlag, Berlin, Heidelberg, 275–294. https://doi.org/10.1007/978-3-540-92221-6_19

[61] Michael F. Spear, Luke Dalessandro, Virendra J. Marathe, and Michael L. Scott. 2009. A Comprehensive Strategy for Contention Management in Software Transactional Memory. In *Proceedings of the 14th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP '09)*. ACM, New York, NY, USA, 141–150. https://doi.org/10.1145/1504176.1504199

[62] Michael F. Spear, Maged M. Michael, and Christoph von Praun. 2008. RingSTM: Scalable Transactions with a Single Atomic Instruction. In *Proceedings of the Twentieth Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '08)*. ACM, New York, NY, USA, 275–284. https://doi.org/10.1145/1378533.1378583

[63] Alexander Spiegelman, Guy Golan-Gueta, and Idit Keidar. 2016. Transactional Data Structure Libraries. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '16)*. ACM, New York, NY, USA, 682–696. https://doi.org/10.1145/2908080.2908112

[64] Suvinay Subramanian, Mark C. Jeffrey, Maleen Abeydeera, Hyun Ryong Lee, Victor A. Ying, Joel Emer, and Daniel Sanchez. 2017. Fractal: An Execution Model for Fine-Grain Nested Speculative Parallelism. In *Proceedings of the 44th Annual International Symposium on Computer Architecture (ISCA '17)*. ACM, New York, NY, USA, 587–599. https://doi.org/10.1145/3079856.3080218

[65] S.M. Trimberger. 2015. Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology. *Proc. IEEE* 103, 3 (March 2015), 318–331. https://doi.org/10.1109/JPROC.2015.2392104

[66] Stephen Tu, Wenting Zheng, Eddie Kohler, Barbara Liskov, and Samuel Madden. 2013. Speedy Transactions in Multicore In-memory Databases. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles (SOSP '13)*. ACM, New York, NY, USA, 18–32. https://doi.org/10.1145/2517349.2522713

[67] Gwendolyn Voskuilen, Faraz Ahmad, and T. N. Vijaykumar. 2010. Timetraveler: Exploiting Acyclic Races for Optimizing Memory Race Recording. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 198–209. https://doi.org/10.1145/1815961.1815986

[68] Stephen Warshall. 1962. A Theorem on Boolean Matrices. *J. ACM* 9, 1 (Jan. 1962), 11–12. https://doi.org/10.1145/321105.321107

[69] W. E. Weihl. 1989. Local Atomicity Properties: Modular Concurrency Control for Abstract Data Types. *ACM Trans. Program. Lang. Syst.* 11, 2 (April 1989), 249–282. https://doi.org/10.1145/63264.63518

[70] Yingjun Wu and Kian-Lee Tan. 2016. Scalable In-memory Transaction Processing with HTM. In *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (USENIX ATC '16)*. USENIX Association, Berkeley, CA, USA, 365–377. http://dl.acm.org/citation.cfm?id=3026959.3026993

[71] Richard M. Yoo, Christopher J. Hughes, Konrad Lai, and Ravi Rajwar. 2013. Performance Evaluation of Intel® Transactional Synchronization Extensions for High-performance Computing. In *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC '13)*. ACM, New York, NY, USA, 19:1–19:11. https://doi.org/10.1145/2503210.2503232

[72] Xiangyao Yu, George Bezerra, Andrew Pavlo, Srinivas Devadas, and Michael Stonebraker. 2014. Staring into the Abyss: An Evaluation of Concurrency Control with One Thousand Cores. *Proc. VLDB Endow.* 8, 3 (Nov. 2014), 209–220. https://doi.org/10.14778/2735508.2735511

[73] Xiangyao Yu, Andrew Pavlo, Daniel Sanchez, and Srinivas Devadas. 2016. TicToc: Time Traveling Optimistic Concurrency Control. In *Proceedings of the 2016 International Conference on Management of Data - SIGMOD '16 (SIGMOD '16)*. ACM Press, San Francisco, California, USA, 1629–1642. https://doi.org/10.1145/2882903.2882935

[74] G. Zhang, V. Chiu, and D. Sanchez. 2016. Exploiting semantic commutativity in hardware speculation. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*. 1–12. https://doi.org/10.1109/MICRO.2016.7783737