# Breaking the Synchronization Bottleneck with Reconfigurable Transactional Execution

Zhaoshi Li, Leibo Liu, Yangdong Deng, Shouyi Yin, and Shaojun Wei

**Abstract**—The advent of FPGA-based hybrid architecture offers the opportunity of customizing memory subsystems to enhance the overall system performance. However, it is not straightforward to design efficient FPGA circuits for emerging FPGAs applications such as in-memory database and graph analytics, which heavily depend on concurrent data structures (CDS'). Highly dynamic behaviors of CDS' have to be orchestrated by synchronization primitives for correct execution. These primitives induce overwhelming memory traffic for synchronizations on FPGAs. This paper proposes a novel method for systematically exploring and exploiting memory-level parallelism (MLP) of CDS by transactional execution on FPGAs. Inspired by the idea that semantics of transactions can be implemented in a more efficient and scalable manner on FPGAs than on CPUs, we propose a transaction-based reconfigurable runtime system for capturing MLP of CDS'. Experiments on linked-list and skip-list show our approach achieves 5.18x and 1.55x throughput improvement on average than lock-based FPGA implementations and optimized CDS algorithms on a state-of-the-art multi-core CPU respectively.

**Index Terms**—Reconfigurable hardware, data structures, heterogeneous systems

## 1 Introduction

The advent of the big data era spurs an unprecedentedly strong momentum for power efficient high performance computing. As a result, FPGA-based reconfigurable platforms, which are capable of adapting to the inherent structures of applications, are considered as a promising candidate to accelerate general-purpose applications. The emergent data-intensive computing workloads tend to pose a strong demand for memory performance. From this point of view, FPGAs offers the unique advantage over their CPU and GPU equivalents by allowing customized memory architectures to accommodate the underlying data structures.

FPGAs are equipped with computing elements, memories blocks, and a configurable on-chip interconnection network. With the distributed nature of computing and storage resources, FPGAs are inherently aligned to parallel computing. To efficiently exploit FPGA resources, memory-level parallelism (MLP), which can be defined as the number of overlapping memory accesses [1], has to be extracted from applications [2]. A significant body of research has been dedicated to developing efficient solutions. Such approaches, however, typically depend on compilation-based analysis to extract MLP such that distributed memory blocks of FPGAs can be instantiated as streaming buffers or local scratchpads to reduce the latency and/or improve the effective memory bandwidth [3], [4].

Unfortunately, emerging big data applications such as graph analytics [5] and in-memory database [6], feature dynamic data structures with their behaviors agnostic at compile-time. Current high-level programming models for CPU/GPU resort to the thread-level parallelism to expose MLP from such dynamic data structures. Dynamic accesses to shared data from different threads are coordinated by synchronization primitives like locks and barriers, which have to be manipulated by the programmers, to allow concurrent access and reduce amortized memory latency among threads [7]. Although the thread abstraction can be smoothly ported to FPGA as either finite state machines or pipelines, the migration of synchronization primitives proves to be challenging. Previous endeavors have shown that directly porting the semantics of these primitives to FPGA could lead to insufficient performance [8], [9], [10] compared to CPU counterparts. As a result, in spite of their prevalence in multi-threaded applications, concurrent data structures (CDS') are seldom used in high-level programming tools for FPGAs.

While traditional synchronization primitives are widely used in constructing many useful CDS' on CPU, they are flawed in usability, composability and portability [7]. These flaws have driven CPU programmers to explore alternatives. A promising alternative is *transaction* [11]. A *transaction* is a sequence of steps that should appear to execute one-at-a-time in the system. With transactions, programmers could specify which part of their application should appear to be serialized and leave the implementation of parallelization to the runtime system.

To break the synchronization bottleneck and exploit MLP more efficiently, we propose incorporating transactions into programming tools for FPGA to construct CDS'. Based on analyses of CDS' on modern FPGA architectures, we dive into why the semantics of transactions are more suitable for FPGA implementations than prevailing synchronization primitives in Section 2. Then a novel method for exploiting MLP with transactional execution on FPGAs is demonstrated in Section 3. Concurrent method calls of CDS' are constructed with transactions and managed by a scalable runtime system implemented as logics on FPGA. The resulted memory subsystem is evaluated and compared to state-of-the-art implementations in Section 4. Then we shed light upon how our work will influence future programming tools for FPGA in Section 5.

## 2 Synchronization on FPGA

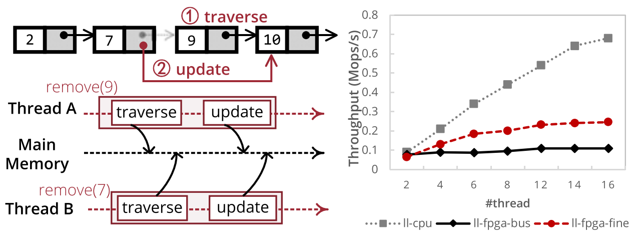
### 2.1 Synchronization in CDS Algorithms

Pointer-based data structures such as link-list and graphs exhibit highly dynamic memory behaviors that are unpredictable at compile-time. Contentions are inevitable under concurrent access to such data structures. We will use a linked-list set as a demonstration. A linked-list set consists of three methods: `search` traverses the linked-list and decide whether the given key is in the set, `insert` adds a key to the set if no node with the same key is found, and `remove` deletes a key in the set. Fig. 1a shows one contentious case of linked-list set. A `remove` method is consist of two steps: (①) it *traverses* the linked list in search of the targeted node; (②) it *updates* the next pointer of previous node so that targeted node is removed from the list. When two `remove` methods are interleaved as Fig. 2a, both threads will report a successful removal while node 9 is not removed since thread B is not aware of the removal of node 9 by thread A.

To resolve contention on dynamic data structures under the context of the shared-memory architecture, existing high-level CPU and GPU programming models use synchronization primitives to coordinate contentious access to shared data regions at run-time. Respective CDS' synchronizations models can be categorized as coarse-grained synchronization, where a single lock is used to synchronize every access to an object, and fine-grained synchronization, where the object is split into independent components for synchronization [7]. Usually the resulted CDS' methods have *linearizability*, a semantic property for correctness that specifies each method call should appear to take effect instantaneously at some moment between invocation and response. The point when the effects are visible to other threads is identified as the *linearization point*. In the case of linked-list

• *The authors are with the National Laboratory for Information Science and Technology, Tsinghua University, Beijing100084, China. E-mail: li-zs12@mails.tsinghua.edu.cn, {liulb, dengyd, yinsy, wsj}@tsinghua.edu.cn.*

*Manuscript received 14 Feb. 2018; revised 10 Apr. 2018; accepted 13 Apr. 2018. Date of publication 18 Apr. 2018; date of current version 8 May 2018.*
*(Corresponding author: Leibo Liu.)*
*For information on obtaining reprints of this article, please send e-mail to: reprints@ieee.org, and reference the Digital Object Identifier below.*
*Digital Object Identifier no. 10.1109/LCA.2018.2828402*



(a) A contentious situation on linked list (b) Throughput of linked list

Fig. 1. Scalability of link-list implemented in coarse-grained (ll-bus) and fine-grained (ll-fine) synchronization on FPGA. Optimized fine-grained synchronization algorithms [12] on CPU are plotted to demonstrate the throughput gap between FPGA and CPU.

`remove` method, correctness should be guaranteed if both traversal and update observe a consistent memory state at the linearization point, which can be achieved by locking both the targeted node and the previous node before updating so that other traversal to these nodes will be blocked.

## 2.2 Synchronization Bottleneck on FPGA

The centralized nature of these synchronization primitives, however, makes them inefficient for a straightforward FPGA implementation. We analyze the problem by evaluating FPGA and CPU implementations of linked-list (experimental setup is to be detailed in Section 4). For coarse-grained synchronization, contentious methods on a CDS are mediated by trying to lock the bus when accessing off-chip memory; for fine-grained synchronization, we use the same algorithm as the CPU where each nodes of the list is associated with a lock resides in off-chip memory. As shown in Fig. 1b, there is a huge throughput gap between CDS' on FPGA and CPU. Even though fine-grained synchronizations expose sufficient MLP and lead to scalable implementations on CPUs, its FPGA implementation incurs a huge overhead that outweighs the benefits of extra MLP.

To understand the overhead of fine-grained synchronization, we profile the memory requests of FPGA threads and count accesses to locks of nodes. Of all the memory requests, the synchronizations traffic could amount to 59 percent when the number of threads reaches 16. Since both data and locks reside on the shared memory, all accesses have to contend for the memory port. The profiling suggests that the fine-grained synchronization greatly increases the pressure to off-chip memory, which already tends to a bottleneck of FPGA applications.

The crux of the above-mentioned bottleneck is the centralized synchronization scheme. Mainstream high-level programming models are abstracted from the dominant imperative paradigm of computation where control-based threads communicating through shared-memory. With modern thread management system, each thread could suspend or migrate to different cores during execution. Thus states of synchronization primitives are associated with shared-memory locations. As a result, simply porting semantics of these synchronization primitives to FPGAs will inevitably lead to the bottleneck of a centralized synchronization scheme.

Nevertheless, there is a fundamental difference between threads on CPUs and FPGAs. Threads on FPGAs are bound to hardware logics that are active all the time. Since there are no preemptions and reschedules of threads, a thread could capture all modifications of shared states by observing other threads through direct on-chip communications without going to the memory subsystem. No synchronization primitives in shared-memory are needed as long as one thread (or parallel component) knows what the others are doing.

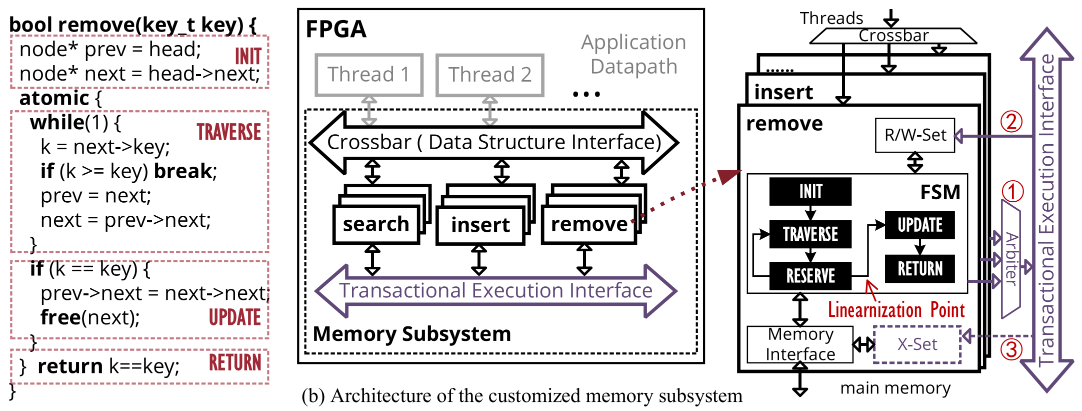
## 2.3 Transaction

To exploit the on-chip communication network on FPGA while making the best use out of existing CDS algorithms, we propose using transactions as a programming tool for constructing CDS' on FPGA and build efficient infrastructure on FPGA for transactional execution.

The phrases *transaction* and *transactional memory* (TM) are often used interchangeably as both a programming model and an executional model in the literature [11]. *As a programming model* they provide constructs (e.g., `atomic` in Fig. 2a) for programmers to delineate a sequence of code as a transaction whose execution must be atomic and isolated. Atomicity means either all actions in a transaction complete or none of them, and isolation means partial updates of a transaction are not visible to other transactions. *As an executional model* they refer to mechanisms in hardware (HTM) or software (STM) to coordinating concurrent executions of transactions. Memory locations read (read-set) or written (write-set) by each transaction are tracked by the runtime system of TM. When the write-set of a transaction intersect with the read-set of other concurrent transactions, a conflict is detected and has to be resolved by rollback.

However, on CPUs the primary goal of transactions is to facilitate programmability and scalability rather than boosting performance. As a high-level executional model STMs have to be built upon existing synchronization constructs like locks, and HTMs rely on ingenious mechanisms built upon already complex cache coherence protocols. Thus, transaction-based CDS' on CPUs are usually inferior in terms of performance to optimized counterparts directly constructed on conventional primitives.

Although directly porting TM infrastructure from CPU to FPGA is cumbersome, customizing memory subsystems on FPGA for transaction-based CDS' algorithms minimize overhead of transactional execution and avoid pitfalls of centralization from lock-based



```
bool remove(key_t key) {
  node* prev = head;
  node* next = head->next;
  atomic {
    while(1) {
      k = next->key;
      if (k >= key) break;
      prev = next;
      next = prev->next;
    }
    if (k == key) {
      prev->next = next->next;
      free(next);
    }
  } return k==key;
}
```

(a) Code for transactional removal (b) Architecture of the customized memory subsystem (main memory interfaces are omitted for simplicity) (c) Detail of the `remove` operator

Fig. 2. Linked list on FPGA: From source code to customized memory subsystem.

synchronization. First, dependences among methods of a specific CDS can be analyzed statically, which indicates that the documentation and intersection of read-set and write-set of each method can be implemented as hardwired logics on FPGA. In this way atomicity and isolation can be guaranteed by simple hardware mechanisms like broadcast and arbitration rather than lock-based synchronization in STM or modified coherence protocols in HTM. Second, there exists a large number of well-defined transaction-based CDS' algorithms. As a result a CDS' library for FPGA can be built upon them.

## 3 Proposed Method

In this section we demonstrate how a customized memory subsystem for linked-list set can be built from established transactional algorithms. Section 3.1 presents the general architecture for transactional execution on FPGA. Section 3.2 highlights customizations applicable to linked-list set on top of the generalized transactional execution. Since no support for transaction exists in current HLS tools, we analyze transaction-based CDS algorithm and write RTL code for demonstration.

### 3.1 Reconfigurable Transactional Execution

Fig. 2b shows the customized architecture of memory subsystem for linked-list. Other CDS' can be constructed from this architecture by substituting corresponding method calls. Here names of finite state machine (FSM) states are inherited from the linked-list `remove` method in Fig. 2a. However, these states can be generalized to other CDS' with TRAVERSE representing execution with buffered writes and UPDATE representing commitment of these writes to the main memory.

The key to coordinate MLP through on-chip communications is reconfigurable transactional execution. As shown in Fig. 2b, all operators are connected to the transactional execution interface (TE), which facilitates the implementation of transactional semantics for each operator.

Fig. 2c depicts the diagram of `remove` operator and its interaction with the TE. Each operator of the linked-list maintains two private buffers for read-set (R-set) and write-set (W-set) respectively. In addition, the TE maintains an exclusive-set (X-set) that is logically shared by all operators. To avoid centralized bottleneck of X-set, each operator maintains a privatized one instead. When a `remove` operation is issued from a thread and allocated to a `remove` operator, it operates according to the FSM. During TRAVERSE, the operator issues read requests to memory, buffers write requests, and logs both read and write addresses in R/W-set respectively. If no conflict with the TE is detected during TRAVERSE state, the operator enters the RESERVE state where it tries to broadcast its write-set though TE. If the broadcast is granted by the TE, it enters the UPDATE state where all buffered writes are sent to the memory. Afterwards the operator returns the result to the thread.

To preserve the transactional semantics, at each cycle the TE chooses at most one operator at the RESERVE state (①) and broadcasts its W-set to the other operators. On receiving a broadcast from TE, operators at TRAVERSE and RESERVE states should compare their R/W-sets to the broadcast addresses (②). If there is a collision, a conflict is detected and should be resolved by re-executing the operation. By this means, *atomicity* is preserved since an operator either commits conflict-free or re-executes under conflicts. In addition, operators at TRAVERSE state have to log the broadcast W-set in the X-set, which indicates another operator has acquired exclusive access to these addresses (③). At each cycle, the operator may delete an address in X-set when it observes the address being granted write access at the memory interface, which indicates this address has been released. During TRAVERSE the operator compares the read/write addresses to X-set before updating the R/W-set, and stalls if collision occurs until when the conflicting address is released. In this way, *isolation* is preserved since no partial results of committing operators are observed by other operators.

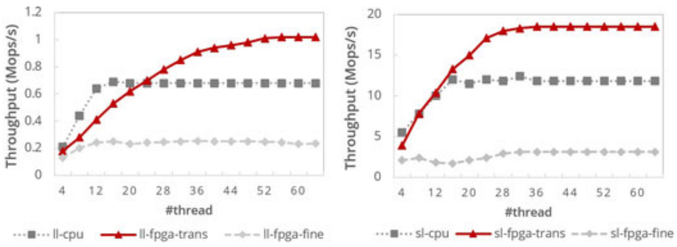


Fig. 3. Throughput of an optimized CPU algorithm [12] and lock-based and transaction-based FPGA implementations.

Operators are *linearized* at the transition from RESERVE to UPDATE when their writes are visible to other operators. Significant amount of MLP can be extracted by enabling concurrent execution of all operators as they are only serialized at the linearization point.

### 3.2 Customization for Linked-List

For linked-list set with ordered keys, it is sufficient to only monitor nodes pointed by the `prev` and `next` shown in the source code from Fig. 2a. This is because updates to nodes before `prev` will not affect the result of traversal (i.e., whether or not the given key is found) on the set. As a result, the read-set of all methods for linked-list only contains address of `prev` and `next`, with previously touched addresses discarded.

Moreover, when read-write conflict is detected, rollback is no longer needed in case of linked-list. The operator with contentious read-set only needs to update its `prev` and `next` with the broadcast results and resend memory requests if necessary. In this way, the contention management is greatly simplified.

In a word, in-depth customizations of transaction-based CDS can be exploited from semantics of each method. These customizations take the opportunity to tailor the hardware resources on FPGA for the additional steps of transactional execution.

## 4 Evaluation

We evaluate our method by porting linked-list (ll) and skip-list (sl) set to FPGA with our method.

CPU implementations are evaluated on an Intel Core-i7 5930K CPU running at 3.5 GHz. Source codes of state-of-the-art algorithms [12] are compiled with gcc 5.4.0 by setting the optimization level as -O3. FPGA implementations are evaluated on Intel HARP2,[1] a server cluster that connects processor with an in-package Arria 10 FPGA. In this evaluation, all FPGA implementations operate at 400 MHz.

Performance is measured by running T threads for 1 second on CPU and FPGA. Note that each thread corresponds to a queue of operations on FPGA. Method calls and their parameters are generated locally from pseudo-random generators on both CPU and FPGA, with a distribution of 80 percent searches, 10 percent inserts, and 10 percent removes. All CSDS are initialized by inserting 2,048 random keys. Keys of each method calls are uniformly distributed between 0 and 4096. In both CPU and FPGA implementations, the layout of data structures is the same, with the address of each node aligned to a 64B cache line.

Fig. 3 depicts the comparison of throughput on CPU and FPGA. The proposed transaction-based (`ll-fpga-trans` and `sl-fpga-trans`) implementations achieve better scalability than CPU counterparts and lock-based FPGA implementations. With high bandwidth and abundant MLP on HARP2, the proposed framework

1. The presented HARP-2 results were obtained on resources hosted at the Paderborn Center for Parallel Computing (PC) in the Intel Hardware Accelerator Research Program (HARP2).

even outperforms state-of-the-art CPU at a much lower frequency with a large number of active threads.

Note that throughput of `ll-fpga-trans` and `sl-fpga-trans` saturate at 64 threads and 32 threads respectively due to limited off-chip bandwidth other than on-chip resources. In our evaluation we instantiate 64 `search`, 8 `insert` and 8 `remove` operators according to experimental setup. The customized memory subsystem of Fig. 3b for linked-list consumes 11 percent logic (in ALM) and 5 percent RAM blocks. With higher bandwidth of emerging devices, our framework could achieve better scalability.

The full potential of FPGA-based CDS has to be unleashed by in-depth customization. In this evaluation each thread is simply a queue of operations. On the other hand, more complex applications tend to have a higher degree of heterogeneity in terms of computation patterns and enable a greater degree of freedom for customization.

## 5 Future Work

This paper demonstrates how transaction-based CDS′ for FPGA can be constructed and achieves better scalability than CDS based on conventional synchronization primitives. Transactions as alternative concurrency control have been overlooked by the HLS community. In addition, customizations for transaction-based CDS′ can be exploited by, which will motivate innovative transactional semantics customized for FPGA.

Our method achieves better throughput than state-of-the-art CPU on Intel HARP2, which could broaden the domain of applications for FPGA (e.g., database) in the future, especially in data centers. Nevertheless, the applicability of our method is not limited to tightly-coupled FPGA fabric like HARP2. On discrete FPGA devices with high off-chip memory bandwidth (e.g., new FPGAs with HBM), transaction-based CDS′ will also enjoy the booming throughput promised by our method.

As part of a systematic effort to automatically design reconfigurable accelerators for emergent data-intensive applications, we are going to extend our method in three directions. First, we will explore efficient memory allocators for CDS′ to build memory management systems for reconfigurable platforms. Second, we are going to provide a library of CDS′ with methods calls as first-class citizen to accelerator designers for FPGA. Third, it is intriguing to explore how to share CDS′ between FPGA and host CPU with transactions. With high performance libraries for CDS, the FPGA will emerge as an attractive data engine for both regular and irregular applications.

## Acknowledgments

This work was supported in part by National Natural Science Foundation of China (No. 61672317) and National Science Technology Major Project (No. 2016ZX01012101).

## References

[1] Y. Chou, B. Fahs, and S. Abraham, "Microarchitecture optimizations for exploiting memory-level parallelism," in *Proc. 31st Annu. Int. Symp. Comput. Archit.*, 2004, Art. no. 76.

[2] G. Weisz, J. Melber, Y. Wang, K. Fleming, E. Nurvitadhi, and J. C. Hoe, "A study of pointer-chasing performance on shared-memory processor-FPGA systems," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 264–273.

[3] R. Prabhakar, D. Koeplinger, K. J. Brown, H. Lee, C. De Sa, C. Kozyrakis, and K. Olukotun, "Generating configurable hardware from parallel patterns," in *Proc. 21st Int. Conf. Architectural Support Programm. Languages Operating Syst.*, 2016, pp. 651–665.

[4] J. Vasiljevic and P. Chow, "MPack: Global memory optimization for stream applications in high-level synthesis," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2014, pp. 233–236.

[5] G. Dai, T. Huang, Y. Chi, N. Xu, Y. Wang, and H. Yang, "ForeGraph: Exploring large-scale graph processing on multi-FPGA architecture," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 217–226.

[6] J. Casper and K. Olukotun, "Hardware acceleration of database operations," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2014, 151–160.

[7] M. Herlihy and N. Shavit, *The Art of Multiprocessor Programming*. San Francisco, CA, USA: Morgan Kaufmann, 2008.

[8] N. Ramanathan, S. T. Fleming, J. Wickerson, and G. A. Constantinides, "Hardware synthesis of weakly consistent C concurrency," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2017, pp. 169–178.

[9] N. Ramanathan, J. Wickerson, F. Winterstein, and G. A. Constantinides, "A case for work-stealing on FPGAs with OpenCL atomics," in *Proc. ACM/SIGDA Int. Symp. Field-Programmable Gate Arrays*, 2016, pp. 48–53.

[10] F. Winterstein, S. Bayliss, and G. A. Constantinides, "High-level synthesis of dynamic data structures: A case study using vivado HLS," in *Proc. Int. Conf. Field-Programmable Technol.*, 2013, pp. 362–365.

[11] T. Harris, J. Larus, and R. Rajwar, *Transactional Memory*, 2nd ed., San Rafael, CA, USA: Morgan and Claypool, 2010.

[12] T. David, R. Guerraoui, and V. Trigonakis, "Asynchronized concurrency: The secret to scaling concurrent search data structures," in *Proc. 20th Int. Conf. Archit. Support Program. Languages Operating Syst.*, 2015, pp. 631–644.