FPGA-Accelerated Transactional Execution of Graph Workloads

Xiaoyu Ma, Dan Zhang and Derek Chiou
The University of Texas at Austin
{xma, dzhang, derek}@utexas.edu

ABSTRACT

Many applications that operate on large graphs can be intuitively parallelized by executing a large number of the graph operations concurrently and as transactions to deal with potential conflicts. However, large numbers of operations occurring concurrently might incur too many conflicts that would negate the potential benefits of the parallelization which has probably made highly multi-threaded transactional machines seem impractical. Given the large size and topology of many modern graphs, however, such machines can provide real performance, energy efficiency, and programability benefits. This paper describes an architecture that consists of many lightweight multi-threaded processing engines, a global transactional shared memory, and a work scheduler. We present challenges of realizing such an architecture, especially the requirement of scalable conflict detection, and propose solutions. We also argue that despite increased transaction conflicts due to the higher concurrency and single-thread latency, scalable speedup over serial execution can be achieved. We implement the proposed architecture as a synthesizable FPGA RTL design and demonstrate improved per-socket performance (2X) and energy efficiency (22X) by comparing to a baseline platform that contains two Intel Haswell processors, each with 12 cores.

Keywords

Graph Application; FPGA Accelerator; Transactional Memory; Throughput Compute; Multi-threaded Architecture

1. INTRODUCTION

Graph applications, widely used in many fields such as social network analysis, machine learning, data mining, electronic design automation and etc., have massive data parallelism because they iterate over a large number of vertices and/or edges using the same operators. Unlike regular data-parallel applications which can be well accelerated by SIMD (e.g. vector processing) or SIMT (e.g. GPUs) mechanisms, however, graph workloads are dominated by pointer operations, leading to irregular memory accesses and control divergences which make those mechanisms inefficient. Therefore, parallelizing graph workloads requires more general, single-program-multiple-thread mechanisms, often achieved by thread-level parallelization techniques.

Transactions simplify parallel programming of graph applications by eliminating the need for fine-grained locks [1]. The scheduling of transactions is often combined with optimistic concurrent execution, which assumes transactions do not conflict, executes them in parallel and relies on transactional memory (TM) to detect conflicts and rollback conflicting transactions as needed. TM performs two main tasks: conflict detection and version management. Conflict detection signals an overlap between writes of one transaction and writes or reads of other concurrent transactions. Conflict detection is either eager (sometimes called pessimistic) if it detects offending accesses immediately or lazy (sometimes called optimistic) if it defers detection until later (e.g. when transactions commit). Version management defines where and how transactional writes are stored. The two main options for version management are lazy and eager. In a lazy approach, new values are stored in a temporary write buffer and are committed (or discarded) if the transaction succeeds (or fails). In an eager approach, new values are written into memory directly and old values are held in an undo-log. The undo-log is discarded (or flushed to memory) if the transaction succeeds (or fails).

The goal of this work is to accelerate transactional execution of graph applications. Towards this goal, we propose running many transactions in parallel to exploit the irregular data-parallelism available in graph workloads and an architecture to achieve this idea. The proposed architecture consists of lightweight multi-threaded processing engines that support the non-lockstep parallel execution of many transactions and high memory latency tolerance, a global shared memory with a hardware TM, and a work scheduler. While our idea is also applicable to CPUs and ASICs, we focus on using dedicated logic on FPGAs to exploit fine-grained parallelism and reduce general-purpose compute overheads like instruction fetch and decode. We implement the proposed architecture as a synthesizable FPGA RTL design and compare in performance and energy efficiency against a baseline platform using Intel 24-core Haswell processors.

2. MOTIVATIONAL EXAMPLE

As an example to motivate the techniques to be presented, the Vertex Exploration kernel (Figure 1) performs a BFS-

FPGA '17, February 22–24, 2017, Monterey, CA, USA.
© 2017 ACM. ISBN 978-1-4503-4354-1/17/02...$15.00
DOI: http://dx.doi.org/10.1145/3020078.3021743

```
Vertex-Exploration (graph);

  graph.init (); // init graph
  workQ.add (graph.root); // init workQ

  // main loop
  foreach (Vertex v in workQ) {
    // start of transaction
    if (v.visited == 0) {
      v.visit ();
      v.visited = 1;
      foreach (Vertex u : v.neighbors)
        workQ.add (u);
    }
    // end of transaction
  }
```

Figure 1: Vertex Exploration

like process that traverses all graph vertices reachable from the root and visits each vertex exactly once by updating a *visited* bit. During graph initialization, the *visited* bit is set to *0* for all vertices. The algorithm starts from the root node by putting it into a work queue. The work queue is a pool, with a scheduling policy e.g. FIFO, holding vertices to be processed in the future and supports dynamic work production and consumption, which is common for graph applications. The *foreach* loop body is transactional. Every iteration gets one vertex from the work queue and checks the *visited* bit of the vertex. If that vertex has been visited already, nothing happens and the iteration is done. Otherwise, the *visit()* function is called. Afterwards its *visited* bit is updated to *1* and its neighbor vertices are added into the work queue. If multiple vertices share a common adjacent vertex, each of them may generate a work item corresponding to that neighbor, leading to duplicated work. While duplicated work exists, transactional semantics ensure the vertex is visited only once since the *visited* bit is set by the first visitor to prevent others. If two transactions try to set the *visited* bit at the same time, a conflict is detected and handled so that only one can proceed and the other is aborted for re-execution.

The application exhibits massive loop-level data parallelism for large input datasets with e.g. millions of nodes. The available parallelism can be characterized by the number of work items available in the work queue as well as the likelihood of conflicts when they are executed in parallel. While initially the parallelism is limited to one vertex since only the root vertex is in the work queue, processing one vertex can cause all its neighbors to be inserted as new work into the work queue, leading to a quick increase of the available parallelism. Both the amount of parallelism and the likelihood of conflicts depend heavily on the size and topology of the input graph. In general, as the input graph gets bigger, the chance of conflicts decreases and the available parallelism increases accordingly until the algorithm runs out of work.

We propose running many transactions in parallel to exploit the exhibited parallelism. Transactions are viewed as software threads running on hardware substrate. Similar to GPUs, we emphasize high throughput rather than low single-thread latency. Unlike vector processors and GPUs that execute concurrent threads in a lockstep fashion, which we call *synchronous execution*, we run transactions as independent threads not in lockstep, defined as *asynchronous execution*. We don't use *synchronous execution* models because they are inefficient due to the under-utilization issue of compute and memory resources [2–4] caused by control and data divergences of graph workloads. For the above example, assuming a real-world social network where some vertices are connected to thousands of edges while some are connected to only a few, the variation of transaction latency is significant, causing the synchronous model to not be a good fit because the overall latency of a group of concurrent threads is determined by the worst case in that group. In addition to high thread count and asynchronous execution, we employ hardware multi-threading for high latency tolerance to address the low spatial locality of point-based graph structures and support a large number of outstanding memory requests to explore memory-level parallelism, both ideas introduced by the Tera multi-threaded architecture [5].

3. KEY PROBLEMS

- While running transactions with increased concurrency exploits more parallelism, transaction conflicts increase accordingly because of more active transactions and higher single-thread latencies. Increased conflicts hurt scalability by undermining the performance gain brought by throughput compute, making the worthwhileness of running many transactions concurrently unclear.
- A scalable approach for conflict detection among a large number of asynchronous transactions is required to realize the proposed idea. While transactional memory (TM) designs have been proposed for GPU bulk synchronous execution with thousands of threads, they are not applicable to asynchronous systems. Other prior TMs do not target architectures with hundreds or thousands of threads.
- The slower a transaction, the higher the probability that it causes other transactions to fail because it does not release resources quickly enough. Transaction re-execution can also cause other transaction failure, leading to potential livelocks.
- The cache hierarchy is often affected by supporting transactions in existing TMs. A decoupled TM from the cache hierarchy would make the design cleaner and more flexible. In addition, it would enable replacing the silicon area that caches consume for additional computational capabilities, since caches are often area and energy inefficient for graph applications.

4. RELATED WORK

4.1 HW-Accelerated Graph Workloads

Hardware acceleration of graph workloads has gained a lot attention due to the explosion of digital data and the ever-growing need for fast data analysis. Several architectural requirements [6] were presented for hardware specialization of graph computations. Such requirements include an active work set, asynchronous execution with specialized synchronization mechanisms, memory latency tolerance, dynamic load balancing and customized memory subsystem. Many of these requirements have been realized in a variety of graph accelerator architectures. Betkaoui et al. proposed a reconfigurable architecture [7] which doesn't have a traditional cached memory hierarchy and tolerates off-chip memory latency by using a memory crossbar that connects many parallel identical processing elements to the shared off-chip memory. GraphStep [8] presented a concurrent system architecture for sparse graph algorithms that places graph nodes in small distributed memories paired with specialized graph processing nodes interconnected by a lightweight network. It also claimed the proposed architecture can be well mapped to FPGAs which have high-bandwidth and low-latency embedded memories. GraphGen [9] automatically compiles a vertex-centric graph specification onto an application-specific processor with a specialized memory subsystem and pipelined logic for user-defined instructions. Ozdal et al. [10] used System-C HLS flow to generate graph accelerators from programs based on the gather, apply and scatter model. The generated accelerator has a vertex-centric microarchitecture where tens of vertices and hundreds of edges are processed simultaneously and asynchronously for memory-level parallelism exploration. Tesseract [11] applied the concept of processing-in-memory (PIM) to graph processing by designing a programmable PIM accelerator with many in-order cores inside a memory chip, a new message passing mechanism to hide remote access latency and specialized graph prefetchers.

4.2 HW Transactional Memory (HTM)

HTM on CPUs. There are many HTM designs on CPUs [12–24], with focuses on version management, conflict detection, unbounded transaction support, scalability, flexibility and etc. Besides academic projects, HTMs are also available in industry implementations, such as Sun Rock [25], Intel Haswell and Broadwell [26,27], IBM BlueGene/Q [28], SystemZ [29] and POWER8 [30] processors. In general, these designs target a shared memory multiprocessor with a separate cache for each core and achieve HTM through modifying standard cache coherence protocols. Version management is performed by either buffering speculative state, typically in the cache or store buffer, or maintaining an undo-log. Metadata required by conflict detection is typically stored in Bloom filters or in bits added to each cache line.

HTM on GPUs. KILO TM [31] is an HTM design for GPUs to support thousands of concurrent transactions. It employs lazy version management and lazy conflict detection and leverages GPU's SIMT execution model. During the speculative execution, transactions within a thread block execute instructions like non-transactional code and buffer memory reads and writes in the local memory. After the thread block is done, a procedure is performed to detect conflicts among threads of the thread block and to commit/discard speculative results before the next thread block can be launched. Such a bulk synchronization assumed by KILO TM, however, doesn't exist in asynchronous execution which requires conflicts to be detected simultaneously with threads' speculative execution.

HTM on FPGAs. TMACC [32] offloaded conflict detection onto Bloom filter FPGA accelerators while having version management handled by software, and claimed by doing so the processor, caches, or coherence protocol do not need modifications in order to support accelerated transactional execution. Pusceddu et al. [33,34] presented a software layer to run TM on FPGA-based soft cores for small-scale embedded systems. Njoroge et al. [35] prototyped TCC [13] on an FPGA hosting two PowerPC soft cores. Kachris and Kulkarni [36] proposed an HTM design on four FPGA soft multiprocessors without private L1 caches. Grinberg and Weiss [37] took a similar approach in using FPGA-based HTM to simulate transactional systems. Labrecque and Steffan built NetTM [38] in an FPGA-based, multi-threaded, multicore architecture. They claimed that eager version management and eager conflict detection is a practical way of implementing FPGA-based HTM. NetTM supports eight threads running on two single-issue in-order cores, as well as large transactions through in-memory undo-logs.

Key Differences. Features that distinguish our work from prior HTM work are the capability of running hundreds to thousands of concurrent transactions in an asynchronous execution model and the worklist-based runtime control of transaction scheduling.

5. ACCELERATOR ARCHITECTURE

In this section we describe the accelerator architecture (Figure 2) that supports the execution of hundreds to thousands of transactions in the asynchronous execution environment.

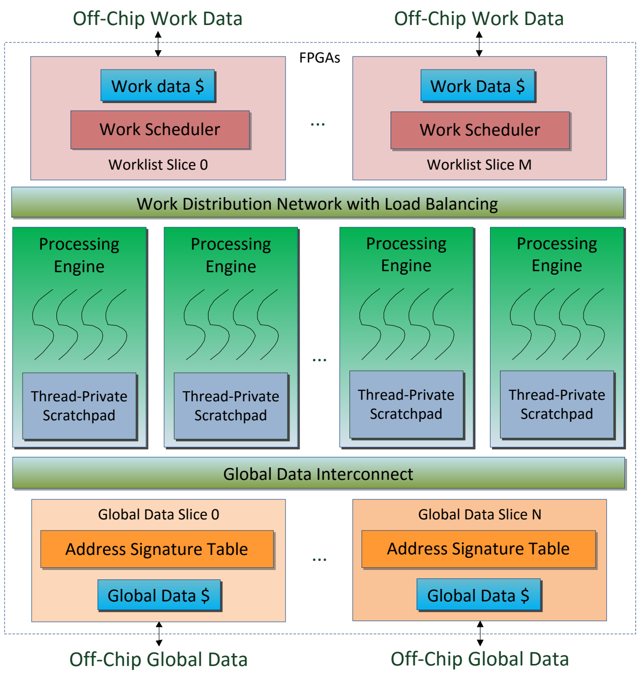


Figure 2: Architecture Overview

5.1 Memory Model

There are several memory spaces in the proposed architecture. **Global Shared Memory** presents a single view of the graph data structure. Global memory consistency is ensured by transactional semantics. Concurrent write/writes and read/writes to the same location from different threads are detected and handled by the transactional execution mechanism, eliminating the possibility of inconsistent states. **Work Memory** is a place for work data to be fetched by the worklist. Work memory requires off-chip storage since the work data can have an unbounded size. **Thread-Private Scratchpad Memory** holds each transaction's private data including intermediate results, speculatively created work, lock-log and versioning buffer (explained later). Since it's private to each thread, it cannot be used for communication among threads. Each thread starts with a clean scratchpad, eliminating the need of checkpointing before the transaction starts. Thread-private scratchpad uses on-chip buffer and may spill to off-chip for large transactions which cause hardware capacity overflow.

5.2 Processing Engines

Processing engines can be implemented using soft or hard processors or customized logic on FPGAs. They are fed with work data and process one work item at a time as a transaction. The work item definition and the operations that process the work item are application specific. For many applications the work item is a vertex or an edge, along with some attributes if needed. The processing of the work item typically touches that vertex or edge as well as a set of connected vertices and/or edges.

Processing engines access the global shared memory and the thread-private scratchpad memory as needed. One engine has multiple hardware threads, of which only one is active at any time. The engine switches among those threads in the event of off-chip memory accesses to hide memory latency. After a engine's currently active thread issues a memory operation, the engine switches to a different thread that is ready to proceed. When the result of the memory operation arrives, the thread that issued the operation earlier is rescheduled so that it becomes a candidate to be selected for continuation on a future context switch. Arithmetic operations of a thread can be executed either in-order or out-of-order to exploit parallelism within a thread. Global memory operations are issued in program order, eliminating single-thread memory write-after-write and write-after-read hazards and, therefore, simplifying the management of transactional states.

5.3 Worklist

The worklist feeds processing engines by fetching work data from work memory. The execution of a transaction may produce one or multiple new work items. Speculatively created new work is saved into the per-thread scratchpad memory. Later if that transaction commits, new work is sent to the worklist. Otherwise, new work is discarded and the work item processed by the failed transaction is sent back to the worklist to be scheduled for future re-execution.

The worklist has one or multiple instances, each called a worklist slice. Since producing/consuming work from the worklist usually has higher throughput than performing the computation, one worklist slice can feed many threads running on one or multiple processing engines. When there are multiple worklist slices, they are connected via an interconnect, e.g. a ring, and load balancing techniques can be employed to improve resource utilization. In each worklist slice, there is a scheduler that decides the order in which work items are sent to execution. For applications that require a specific execution order, bucket-based priority scheduling is widely used. A bucket-based priority scheduler [39] supports concurrent execution by allowing work items in the same bucket to execute in parallel. It also prioritizes buckets to achieve ordered execution, which is important to keep parallel execution efficient by reducing bad work, defined as updates of vertex/edge attributes to non-final values. While the bucket-based priority scheduler suppresses bad work, it doesn't scale well because bad work increases as the worklist slice count increases. For some applications, the scheduling order has little performance impact and, therefore, simpler schedulers such as FIFO/LIFO are used for logic and power savings. In addition to scheduling, the worklist has done detection logic to terminate the execution when all worklist slices are empty and all processing engines are idle.

5.4 HW Transactional Memory

Threads are optimistically dispatched onto processing engines to execute by assuming the threads are isolated and do not conflict. Transactional memory detects and handles transactional semantics violations through version management, conflict detection and conflict resolution. Nested transactions are not supported. The proposed HTM does not rely on caches or coherence protocols.

5.4.1 Conflict Detection and Resolution

We adopt eager conflict detection because it identifies conflicts earlier and hence enables an earlier start of rollback to reduce wasted work. Conflict detection is performed for every global memory access. We provide scalable conflict detection by (1) using a shared directory to avoid broadcasting address signatures and (2) supporting concurrent conflict detection for memory accesses with different addresses. These ideas are achieved by **Address Signature Table** (AST), a shared directory for metadata that tracks memory footprints of all executing transactions. AST is like a cache whose set is chosen by a hash function. Given an address, the hash function maps it to a set that stores information needed to detect conflicts associated with that address. The stored information tells whether a live transaction has issued a memory request corresponding to the current AST set, indicated by a *valid* bit, and, if so, the *thread ID* of that transaction and whether it is a write or read using a *write* bit. Using the thread ID is space efficient because it has $\log_2 n$ (n is thread count) space complexity and leads to logarithmically increased AST size as the thread count increases. To allow multiple transactions to read the same address, an AST set can have multiple ways with each having these three fields, similar to a set associative cache. Although read-read overlaps should not cause conflict, a conflict must be assumed if there are not enough ways in that set. AST can be implemented using either on- or off- chip memory. On-chip AST supports faster conflict detection. Off-chip AST enables a larger signature table that reduces false sharing, a problem occurring when two different addresses are mapped onto the same AST set by the hash function. To support parallel conflict detection, the AST is partitioned into AST slices, each connected to processing engines through a network.

The conflict detection scheme is lock-based. Before a memory operation can be issued to memory, it arrives at its corresponding AST slice and set to reserve the entry by acquiring a lock on that set. During the AST entry reservation, the conflict detection algorithm (presented in the next paragraph) is performed to determine whether the memory operation conflicts with memory accesses of other concurrent threads. If a conflict does not occur, then the lock is acquired, the AST set is updated as needed, and the memory request will be sent to memory. Otherwise, the lock is not acquired by the requester thread. No further execution of that thread will be performed and that thread will abort. While detecting conflicts for every global data access is the default, it is not needed by an access to read-only data or an access with the same address as a preceding access that has been issued by the same thread and successfully acquired a lock. Such accesses can be marked as conflict free to bypass conflict detection.

The conflict detection protocol is as follows. (1) **Reads.** For a read, a valid way in the corresponding set of the read address with a matching thread ID indicates this thread already owns this location and, therefore, no conflict will be detected. Otherwise, if there is no write bit set for any valid ways and the set is not full, no conflict will be detected and one invalid way is allocated, setting the valid bit and the thread ID. In doing this, read-read overlaps are not detected as conflicts unless the associativity runs out. If there is a valid way with a different thread ID and a set write bit, which indicates another thread is writing to this location, then a conflict is detected. (2) **Writes.** Writes must be exclusive. If there exists a valid way in the corresponding set of the write address with a different thread ID, a conflict is detected because this implies another thread is writing to this location. Otherwise, no conflict is detected and one way of that set is allocated to the requester thread if needed, setting both the valid bit and the write bit.

One transaction can acquire locks on many addresses belonging to different AST slices and sets. Those locks need to be reclaimed during commit/abort so that they can be acquired later by other transactions. We use a per-thread **Lock-Log** residing in the thread-private scratchpad memory to keep track of each transaction's acquired locks by logging their addresses. The lock-log eliminates broadcasting the thread ID to all AST slices. Unlock requests are sent to AST slices associated with addresses recorded in the lock-log. Redundant entries of lock-log can be eliminated by logging an acquired lock only if a new AST entry has been allocated to the memory request. There is no required order in which lock-log entries are accessed during unlock. A simple solution is to organize the lock-log as an FIFO.

5.4.2 *Version Management*

Many prior HTM work leverages lazy version management by taking advantage of, and at the same time being constrained by, speculative execution that already exists in the microprocessor. Since we don't have that constraint, either eager or lazy can be used. The eager approach, which updates memory locations with speculative values and saves old values in a per-thread undo-log, allows faster commit by simply discarding the undo-log. It may hurt memory performance in two ways. First, each write introduces an extra read for the old value. The extra read can be eliminated if a preceding read has acquired the old value already, which is fortunately true for many graph applications. Second, memory bandwidth is wasted in performing and undoing the writes for aborted transactions. The lazy approach, which saves speculative values in a per-thread write-buffer and writes them to memory during commit, has faster abort by simply discarding the write-buffer. It also has the advantage of reducing memory traffic by coalesced writes but the disadvantage that the write-buffer must be searched for each read. In general, whether to use eager or lazy version management depends on how frequently rollback occurs. To achieve scalable performance, conflicts should be minimized and commit should be the common case. This makes the eager approach approach more attractive. In addition, the eager undo-log has a simpler structure than the lazy write-buffer. The undo-log requires only sequential accesses. It can be achieved by an LIFO that saves old values of writes in program order and flushes to memory in reverse program order. In contrast, the lazy write-buffer requires an associative search on every read to load values from the latest write with the same address if single-thread read-after-write dependencies exist.

Version management is not needed when two conditions are satisfied. The first condition is limiting transactions to be cautious [1], that is, all reads occur before writes in program order and any write requires a preceding read to the same address. In fact, any non-cautious transaction is transformable to a cautious transaction by the programmer or compiler. The second condition is exclusive locking, meaning that a transaction holds an exclusive lock on data to prevent other transactions accessing the data (even if all accesses are read). Exclusive locking can be achieved by an one-way AST. When both conditions are met, the decision of transaction commit or rollback can be made before any write, eliminating the need of versioning buffer.

5.5 Livelock Avoidance

Livelocks are created by wasted work generating wasted work infinitely. For example, a thread that is aborting may not have had time to clear AST entries that another thread needs, causing the second thread unnecessarily to abort. If such abortings are circular and occur indefinitely, a livelock may occur. Making the abort process faster, e.g. through fast unlock and undo techniques, can reduce the chance of livelock, but cannot eliminate since it is impractical to make abort instantaneously fast. To tackle this problem, we introduce a concurrency control technique to dynamically turn on/off threads when needed. Each processing engine has logic that counts commits and aborts to compute the conflict rate. When the conflict rate during the past time interval is higher than the threshold, a certain number of threads are disabled. In case of livelock, no forward progress can be made so every transaction aborts. This would keep reducing the active thread count and eventually fall back to sequential execution which can always move forward since it has no conflicts. Likewise, threads that are disabled earlier can be re-activated if the conflict rate decreases. Besides avoiding livelocks, dynamic concurrency control can have positive performance impact by dynamically adjusting the execution width to the parallelism available to reduce conflicts.

5.6 Cache Hierarchy

Various caches can be introduced into the proposed architecture. A work data cache included in each worklist slice can be useful since work data has good spatial locality and fetching work data in a low-latency and high-throughput manner is important to keep processing engines busy. Per-engine L1 and/or global shared L2 caches for thread-private data such as intermediate results, lock-log and versioning buffer can help exploit short-term producer-consumer locality and speed up transaction commit/rollback. For graph data stored in the global shared memory, since it is required to send every graph data access to the global shared directory to detect possible conflicts, we eliminate per-engine L1 caches and, by doing so, eliminate the need for coherence. Instead, we support global shared L2 caches to service memory accesses that the AST determines are conflict free. To increase memory parallelism, conventional cache optimization techniques, such as banking, pipelining and non-blocking can be applied. Caches that have producer/-consumer semantics and sequential memory footprint, e.g. work data cache for an FIFO worklist, can be further optimized to save unnecessary memory traffic. In addition, with the separation of HTM from cache hierarchy, each of these caches can be turned off to trade latency for logic and energy savings and the design would still work functionally.

6. EXPERIMENTAL EVALUATION

6.1 Experiment Methodology

6.1.1 Benchmarks

We study several benchmarks selected from the graph sub-category of graph exploration, shortest paths, connectivity analysis and graph coloring in the evaluation of our proposed approach. **Vertex Exploration** (VE) performs a BFS-like process to visit each vertex reachable from the root exactly once. **Bipartite Coloring** (BC) assigns one of two colors to each vertex of a bipartite graph such that no two adjacent vertices share the same color. **Transitive Closure** (TC) solves the reachability problem in graph theory. Given a graph in the form of adjacent matrix, it performs an iterative search to incrementally compute a two-dimensional reachability matrix, of which an element (i, j) represent whether there exists a path from vertex i to j in one or more hops. **Single Source Shortest Path** (SSSP) is a problem of finding shortest paths from a source vertex to all other vertices such that the sum of the weights of its constituent edges is minimized. The algorithm selected to solve this problem uses a demand-driven modification of the Bellman-Ford algorithm. **Breadth-First Search** (BFS) is a special case of SSSP with the distance being the number of hops required to reach from the source vertex. Its work scheduler requires very small bucket intervals and large per-bucket sizes, leading to significantly different dynamic behavior from SSSP. **Connected Components** (CC) performs a label propagation process to compute for each vertex the smallest component ID of all vertices reachable from that vertex.

S/L	**ROAD**		**RAND**		**RMAT**	
	Nodes	Edges	Nodes	Edges	Nodes	Edges
Small	1.9M	4.7M	1M	4M	256K	2M
Large	24M	58M	16M	64M	4M	32M

Table 1: Graph Categories and Sizes

Graphs are selected under three categories with different shapes characterized by graph diameter (the greatest distance between any pair of vertices in the graph) and vertex degree (the number of edges incident to the vertex). **Road** networks have large diameters and small vertex degrees. We use real-world road graphs such as the US road network. **Random** networks have constant degrees and medium diameters. They are synthetic graphs produced by a graph generator. **Scale-free** networks have small diameters and large vertex degrees following a power-law distribution. They are generated by a recursive matrix (RMAT) model [40]. For each of these categories, we run a small graph to obtain the scalability results shown in Section 6.2.1 and a large graph to get results listed in Section 6.2.2 (Table 1), except Transitive Closure that has a $O(V^3)$ worst case complexity so it takes the first 1K and 4K nodes of small and large graphs respectively.

6.1.2 Baseline

The baseline platform uses 2.6GHz Intel Xeon E5-2690 v3 processors based on the Haswell-EP microarchitecture. The system has two processor sockets and each socket has 12 cores. Each core has private 32KB L1-Data, 32KB L1-Instruction and 256KB L2 caches. Cores on the same socket share a 30MB L3 cache. The system contains DDR4-2133 DRAM with a theoretical peak transfer rate of 17GB/s per channel. There are two connected DRAM channels on each of the two processor sockets, leading to an aggregated peak memory bandwidth of 68GB/s for the entire system. Total DRAM capacity of the system is 64GB. When needed, NUMA and huge pages are enabled for best performance.

Software is developed in C++ using Galois [1], a state-of-art high-performance graph computation framework. Code is compiled using gcc 4.9.3 version with -O3 flag enabled. A parameter sweep is done for each application and input pair to achieve the highest performance. The Intel vector extension instructions are not used because the SIMD execution model cannot efficiently support the asynchronous nature of graph applications. While HTM was originally available on the baseline processor, it is not used because a bug announced by Intel [41] caused the HTM to be disabled and the lack of runtime control on transaction scheduling leads to very poor performance. Baseline programs use the same algorithm as accelerator implementations. They are, however, not limited to transactional execution and use fine-grained locks and atomic instructions such as compare-and-swap to achieve higher performance. The execution time of baseline runs is measured by timers inserted into the source code. Power measurement leverages Intel Performance Counter Monitor (PCM) which collects the actual energy usage through the processor's on-chip sensors.

6.1.3 Accelerator Implementation

We implement the proposed design using tools provided by the FPGA Research Infrastructure Cloud [42]. The source is developed in Bluespec System Verilog and is passed to the Bluespec compiler to generate Verilog for synthesis and C++ for cycle-accurate simulation orders of magnitude faster than RTL simulation. The implemented processing engine is decoupled into engine infrastructure that is constant across applications and application-specific operators (not soft or hard processors) generated by a script from an FSM-based specification. Implemented worklists support bucket-based priority scheduling used by SSSP, BFS and CC and FIFO scheduling used by other applications. We use a ring network to connect worklist slices for load balancing. Conflict detection is through on-chip AST and implemented as a pipeline. We eliminate version management by making transactions cautious and using exclusive locking achieved by an one-way AST. A crossbar is used as the on-chip network connecting processing engines to AST. The implementation of cache hierarchy includes work data caches and per-engine caches for thread-private scratchpad memory. Graph data is not cached since graph memory access latency is hidden by hardware multi-threading. Implemented caches are direct-mapped, non-blocking, pipelined, and multi-banked.

We synthesize our design on Xilinx Virtex UltraScale 440 and tune our design to meet timing requirements of a 200

MHz clock speed. The whole design consists of one or more FPGAs depending on design parameters such as the number of engines and worklist slices, sizes of various on-chip buffers and etc. In a typical configuration, one FPGA has eight processing engines with each running 16 threads to keep the engine being fully utilized by hiding memory latency. The total 128 threads are fed by two worklist slices for applications using FIFO scheduling and four worklist slices for applications using bucket-based priority scheduling, a configuration that minimizes worklist stall cycles and keeps bad work at a low level. The total work data cache size per FPGA is 1MB. Since versioning buffer is eliminated, thread-private scratchpad memory only contains lock-log and intermediate results which are small enough to fit on LUT resources. AST is 640KB in size and is distributed in four slices. Such a single-FPGA configuration saturates the off-chip memory bandwidth we assume for each socket. The FPGA LUT utilization of such a configuration is around 45% for applications using FIFO scheduling and around 50% for applications using bucket-based priority scheduling, with little variation since application-specific logic is a small fraction of the entire design, and BRAM utilization is typically around 20%. When multiple FPGAs are used, we assume they all have the same image. This assumption is only for the purpose of evaluation simplicity. There are no reasons preventing proposed techniques being applied to a system consisting of different FPGA parts or images.

Cycle-by-cycle simulation is performed to produce performance results. In the simulation, we assume DDR SDRAM is used as the off-chip memory. The assumed memory I/O bus clock frequency is 800 MHz, which is supported by both Altera and Xilinx FPGAs when running user logic at 200 MHz. DRAM data are transferred at a burst length of eight, with each transfer being 64 bits and occurring at both positive and negative clock cycle edges, giving a theoretical peak transfer rate of 12.8GB/s per channel. It is also assumed that each FPGA socket has the same number of DRAM channels as one CPU socket of the baseline. The assumed DRAM device latency is 160ns for every DRAM access. Note that this latency is the wait time before the DRAM produces a response after it receives a request. It does not include the latency of sending DRAM requests and responses through AST (needed only for global data accesses), the on-chip network, various buffers and the memory controller. The actual round-trip DRAM read latency from the processing engine's perspective is around 300ns. For writes that need to reserve AST entries, the round-trip latency doesn't include the assumed DRAM access latency because write responses are sent back to engines without waiting for DRAM to complete the writes. For writes that are lock-free, no responses are expected by engines.

We use the power estimator incorporated in Xilinx Vivado Design Suite to measure power. The estimation is performed on fully placed-and-routed design and with the input of vector-based signal activity files generated by simulation.

6.2 Results and Analysis

6.2.1 Scalability

Due to the 13X lower FPGA clock speed and the focus on multi-threaded throughput, single-threaded execution of our approach is 8X to 32X slower than that of the software baseline (Figure 3). Therefore, executing many threads and achieving scalable speedup is desired to achieve performance equal to or higher than the baseline. To study the scalability of our approach, we vary the thread count and report speedup over serial execution (Figure 4), the percentage rate of conflicts out of the total transaction count (Figure 5), the percentage increase of transaction commits compared to serial execution (Figure 6) and memory bandwidth utilization as a fraction of the theoretical peak (Figure 7).

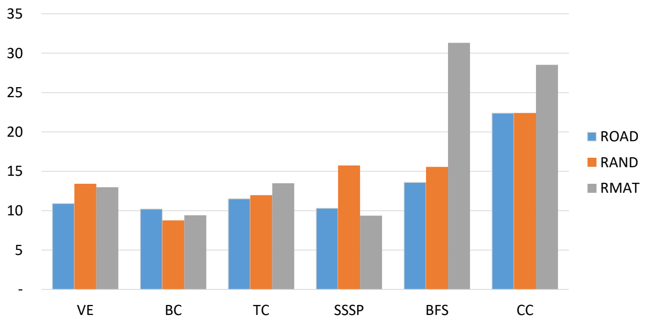
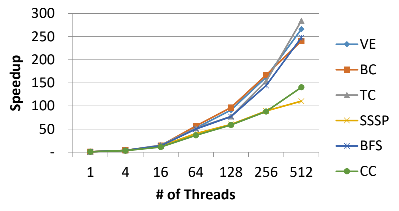


Figure 3: Single-Thread Execution Slowdown (Normalized)

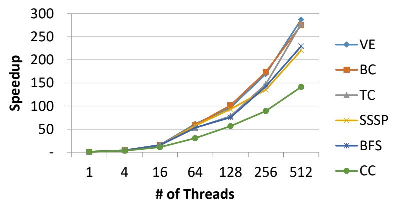
One major limiting factor of scalability is wasted work which can be classified in two categories. The first category is transaction conflicts, which increase as concurrency increases. In some cases, moving from 128 threads to 256 or 512 threads decreases the conflict rate. This is because thread counts high than 128 use multiple FPGAs so the AST size is doubled or quadrupled to reduce conflicts due to false sharing. Another observation is that the conflict rate of road graphs is significantly higher than random and RMAT graphs because road graphs have smaller degrees and larger diameters, increasing the chance of conflicts. The second category of wasted work is bad work and is often associated with bucket-based priority scheduling. Take SSSP for example. Ideally we want all updates that lower the distance value of a node to its final value to minimize the work needed. With carefully chosen bucket intervals single-thread execution can achieve zero or very little bad work. Unfortunately, as concurrency increases it becomes harder to ensure the least distance update is applied in the first place, leaving to increased bad work. Besides SSSP, bad work also exists in BFS and CC, and as a consequence, increased commits are observed for them. Bad work becomes significant at 256 and 512 threads and hurts scalability.

Despite increased wasted work, scalable speedup over serial execution is achieved for most applications and inputs. Scalability persists even in some cases where the conflict rate is over 40%. This is because aborted transactions are usually lightweight, achieved by eager conflict detection and on-chip AST that enable detecting conflicts earlier and stopping further execution to avoid unnecessary work. Applications using worklists with bucket-based priority scheduling is in general not as scalable as applications with worklists with FIFO scheduling due to significant bad work at high thread counts. In addition, input graphs used in the scalability study are small graphs listed in Table 1. Our simulator runs high thread counts fast, but low thread count runs take a long time to complete because there are 10X/100X more cycles to simulate. For this reason we don't use larger graphs but we expect improved scalability as the graph gets bigger.

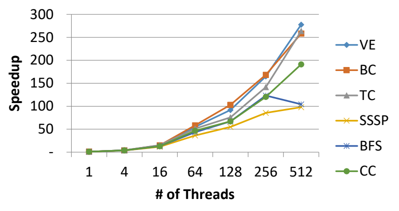
Memory bandwidth usage is proportional to the speedup. Single-FPGA memory bandwidth is saturated at 128 threads,



(a) Using Road Graphs

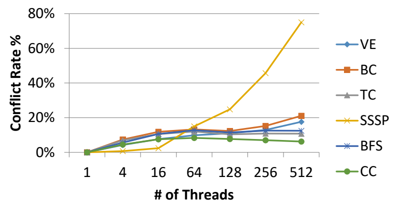


(b) Using Random Graphs

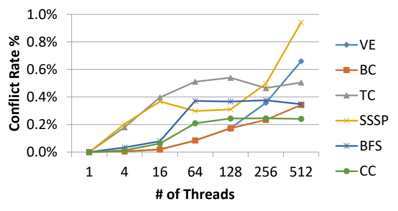


(c) Using RMAT Graphs

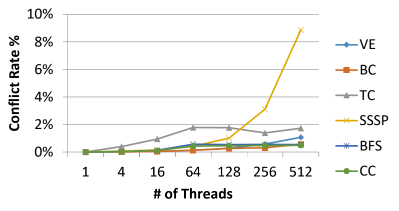
Figure 4: Speedup Over Serial Execution



(a) Using Road Graphs

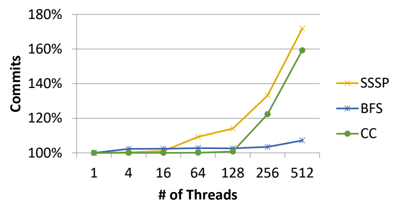


(b) Using Random Graphs

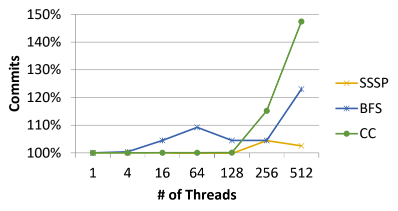


(c) Using RMAT Graphs

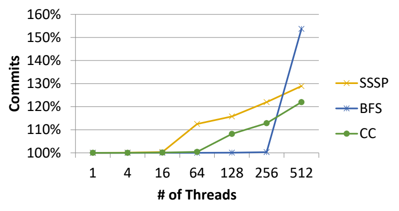
Figure 5: Conflict Rate



(a) Using Road Graphs

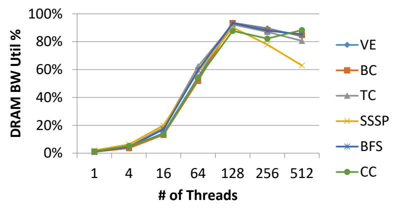


(b) Using Random Graphs

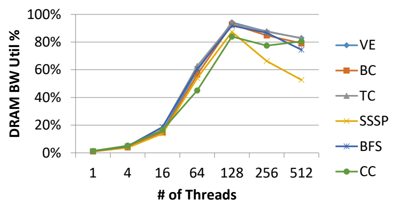


(c) Using RMAT Graphs

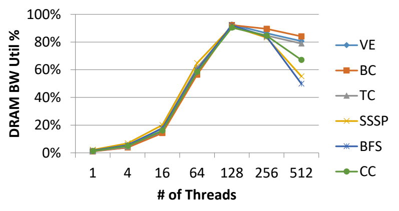
Figure 6: Commits



(a) Using Road Graphs



(b) Using Random Graphs



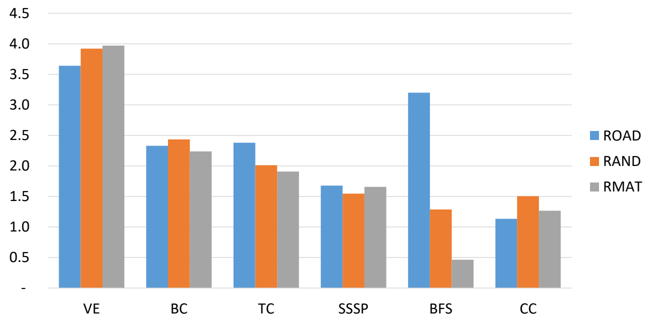
(c) Using RMAT Graphs

Figure 7: Memory Bandwidth Utilization

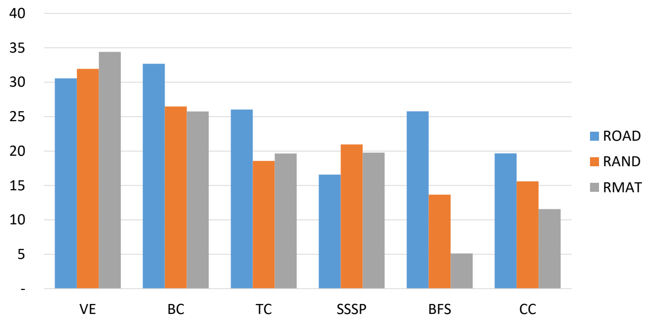
which explains why we run 128 threads per FPGA. Moving from 128 threads to 256 or 512 threads requires doubled or quadrupled FPGAs with the assumed available DRAM bandwidth increasing in a similar factor. So scaling continues and the actual memory use increases while its fraction out of the peak starts decreasing. At high thread counts, peak memory bandwidth utilization drops faster for applications and inputs that don't scale well.

6.2.2 *Comparison Against CPUs*

We compare performance achieved by our solution with a dual-FPGA-socket (256 threads) configuration against the dual-CPU-socket (24 threads) baseline (Figure 8a) by running on large graphs listed in Table 1. Note that time spent on loading graphs into memory and initializing data structure and worklist is excluded from the execution time. On average our approach performs 2.14X faster than the baseline. The only case of lower performance is BFS with RMAT graphs because serial execution on FPGA is over 30X slower than on CPUs and executing 256 threads is only 130X faster than serial execution (results from last section). The comparison of energy efficiency computed as performance per watt (Figure 8b) is based on single-socket. Moving from single-socket to multi-socket leads to linearly increased power but sub-linear speedup for both CPU and FPGA, so single-socket has the highest energy efficiency. For CPU, we use all threads on the first socket with NUMA disabled. We don't count the idle power consumption, which is around 20 watts, of the second CPU socket. In our measurement, a CPU socket with all cores being used consumes 80 to 120 watts while FPGA per-socket power is between 5 and 8 watts. Our comparison doesn't include the off-chip DRAM power, which is 5 to 7 watts, reported by Intel PCM, for a CPU socket, and unknown for FPGAs since it's not included in the power report. Due to the 10X power saving, our approach beats the baseline in all cases and is on average 21.93X more energy efficient. The performance and energy efficiency comparison results are summarized in Table 2.



(a) Speedup (Normalized)



(b) Performance-Per-Watt (Normalized)

Figure 8: FPGAs VS. CPUs

	Clock	DRAM Bandwidth	Threads	Perf.	Perf. Per Watt
CPUs	2.6G	68.0GB/s	24	1X	1X
FPGAs	200M	51.2GB/s	256	2.14X	21.93X

Table 2: Comparison Summary

7. CONCLUSION

In this work we described a way to exploit the irregular loop parallelism of iterative graph workloads by executing loop iterations transactionally and asynchronously on many concurrent threads. We proposed microarchitectural techniques to realize such an idea on FPGAs and demonstrated their effectiveness in accelerating graph kernels. Our experimental results showed our approach beats the Intel 24-core Haswell CPU in per-socket performance while consuming 10X less power. Our future work includes an extensive study of various design choices in transactional state management, conflict detection, cache hierarchy and concurrency control and an evaluation of their impact on conflict rate, single-thread latency and overall performance.

8. REFERENCES

[1] K. Pingali, D. Nguyen, M. Kulkarni, M. Burtscher, M. A. Hassaan, R. Kaleem, T.-H. Lee, A. Lenharth, R. Manevich, M. MÃl'ndez-Lojo, D. Prountzos, and X. Sui, "The tao of parallelism in algorithms," in *Proceedings of the 32nd ACM SIGPLAN conference on Programming language design and implementation*, 2011.

[2] Q. Xu, H. Jeon, and M. Annavaram, "Graph processing on gpus: Where are the bottlenecks?," in *IEEE International Symposium on Workload Characterization*, 2014.

[3] M. Burtscher, R. Nasre, and K. Pingali, "A quantitative study of irregular programs on gpus," in *IEEE International Symposium on Workload Characterization*, 2012.

[4] N. Chatterjee, M. O'Connor, G. H. Loh, N. Jayasena, and R. Balasubramonian, "Managing dram latency divergence in irregular gpgpu applications," in *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, 2014.

[5] R. Alverson, D. Callahan, D. Cummings, B. Koblenz, A. Porterfield, and B. Smith, "The tera computer system," in *Proceedings of the 4th international conference on Supercomputing*, 1990.

[6] M. M. Ozdal, S. Yesil, T. Kim, A. Ayupov, S. Burns, and O. Ozturk, "Architectural requirements for energy efficient execution of graph analytics applications," in *Proceedings of International Conference on Computer-Aided Design*, 2015.

[7] B. Betkaoui, D. B. Thomas, W. Luk, and N. Przulj, "A framework for fpga acceleration of large graph problems: Graphlet counting case study," in *Proceedings of IEEE International Conference on Field-Programmable Technology*, 2011.

[8] M. deLorimier, N. Kapre, N. Mehta, D. Rizzo, I. Eslick, R. Rubin, T. E. Uribe, T. F. J. Knight, and A. DeHon, "Graphstep: A system architecture for sparse-graph algorithms," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 2006.

[9] E. Nurvitadhi, G. Weisz, Y. Wang, S. Hurkat, M. Nguyen, J. C. Hoe, J. F. Martinez, and C. Guestrin, "Graphgen: An fpga framework for vertex-centric graph computation," in *IEEE Symposium on Field Programmable Custom Computing Machines*, 2014.

[10] M. Ozdal, S. Yesil, T. Kim, A. Ayupov, J. Greth, S. Burns, and O. Ozturk, "Energy efficient architecture for graph analytics accelerators," in *Proceedings of International Symposium on Computer Architecture*, 2016.

[11] J. Ahn, S. Hong, S. Yoo, O. Mutlu, and K. Choi, "A scalable processing-in-memory accelerator for parallel graph processing," in *Proceedings of International Symposium on Computer Architecture*, 2015.

[12] M. Herlihy and J. E. B. Moss, "Transactional memory: architectural support for lock-free data structures," in *Proceedings of International Symposium on Computer Architecture*, 1993.

[13] L. Hammond, V. Wong, M. Chen, B. D. Carlstrom, J. D. Davis, B. Hertzberg, M. K. Prabhu, H. Wijaya, C. Kozyrakis, and K. Olukotun, "Transactional memory coherence and consistency," in *Proceedings of International Symposium on Computer Architecture*, 2004.

[14] C. S. Ananian, K. Asanovic, B. C. Kuszmaul, C. E. Leiserson, and S. Lie, "Unbounded transactional memory," in *Proceedings of International Symposium on High Performance Computer Architecture*, 2005.

[15] R. Rajwar, M. Herlihy, and K. Lai, "Virtualizing

transactional memory," in *Proceedings of International Symposium on Computer Architecture*, 2005.

[16] K. E. Moore, J. Bobba, M. J. Moravan, M. D. Hill, and D. A. Wood, "Logtm: Log-based transactional memory," in *Proceedings of International Symposium on High Performance Computer Architecture*, 2006.

[17] M. Lupon, G. Magklis, and A. Gonzalez, "Fastm: A log-based hardware transactional memory with fast abort recovery," in *Proceedings of International Symposium on Parallel Architecture and Compilation Techniques*, 2009.

[18] C. Blundell, J. Devietti, E. C. Lewis, and M. M. K. Martin, "Making the fast case common and the uncommon case simple in unbounded transactional memory," in *Proceedings of International Symposium on Computer Architecture*, 2007.

[19] J. Bobba, N. Goyal, M. Hill, M. Swift, and D. Wood, "Tokentm: Efficient execution of large transactions with hardware transactional memory," in *Proceedings of International Symposium on Computer Architecture*, 2008.

[20] L. Ceze, J. Tuck, C. Cascaval, and J. Torrellas, "Bulk disambiguation of speculative threads in multiprocessors," in *Proceedings of International Symposium on Computer Architecture*, 2006.

[21] L. Yen, J. Bobba, M. R. Marty, K. E. Moore, H. Volos, M. D. Hill, M. M. Swift, and D. A. Wood, "Logtm-se: Decoupling hardware transactional memory from caches," in *Proceedings of International Symposium on High Performance Computer Architecture*, 2007.

[22] A. Shriraman, S. Dwarkadas, and M. L. Scott, "Flexible decoupled transactional memory support," in *Proceedings of International Symposium on Computer Architecture*, 2008.

[23] H. Chafi, J. Casper, B. D. Carlstrom, A. McDonald, C. C. Minh, W. Baek, and C. K. andK. Olukotun, "A scalable, non-blocking approach to transactional memory," in *Proceedings of International Symposium on High Performance Computer Architecture*, 2007.

[24] C. C. Minh, M. Trautmann, J. Chung, A. McDonald, N. Bronson, J. Casper, C. Kozyrakis, and K. Olukotun, "An effective hybrid transactional memory system with strong isolation guarantees," in *Proceedings of International Symposium on Computer Architecture*, 2007.

[25] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay, "Rock: A high-performance sparc cmt processor," in *Proceedings of International Symposium on Microarchitecture*, 2009.

[26] Intel, "Chapter 8. intel transactional synchronization extensions," in *Intel Architecture Instruction Set Extensions Programming Reference*, 2013.

[27] V.Leis, A. Kemper, and T. Neumann, "Exploiting hardware transactional memory in main-memory databases," in *Proceedings of International Conference on Data Engineering*, 2014.

[28] A. Wang, M. Gaudet, P. Wu, J. N. Amaral, M. Ohmacht, C. Barton, R. Silvera, and M. M. Michael, "Evaluation of blue gene/q hardware support for transactional memories," in *Proceedings of international conference on Parallel architectures and compilation techniques*, 2012.

[29] C. Jacobi, T. J. Slegel, and D. F. Greiner, "Transactional memory architecture and implementation for ibm system z," in *Proceedings of International Symposium on Microarchitecture*, 2012.

[30] H. W. Cain, M. M. Michael, B. Frey, C. May, D. Williams, and H. Le, "Robust architectural support for transactional memory in the power architecture," in *Proceedings of International Symposium on Computer Architecture*, 2013.

[31] W. W. L. Fung, I. Singh, A. Brownsword, and T. M. Aamodt, "Hardware transactional memory for gpu architectures," in *Proceedings of International Symposium on Microarchitecture*, 2011.

[32] J. Casper, T. Oguntebi, S. Hong, N. G. Bronson, C. Kozyrakis, and K. Olukotun, "Hardware acceleration of transactional memory on commodity systems," in *Proceedings of international conference on Architectural support for programming languages and operating systems*, 2011.

[33] M. Pusceddu, S. Ceccolini, G. Palermo, D. Sciuto, and A. Tumeo, "A compact transactional memory multiprocessor system on fpga," in *Proceedings of International Conference on Field Programmable Logic and Applications*, 2010.

[34] M. Pusceddu, S. Ceccolini, A. Tumeo, G. Palermo, and D. Sciuto, "Emulating transactional memory on fpga multiprocessors," in *Proceedings of international conference on Architecture of computing systems*, 2011.

[35] N. Njoroge, S. Wee, J. Casper, J. Burdick, Y. Teslyar, C. Kozyrakis, and K. Olukotun, "Building and using the atlas transactional memory system," in *Proceedings of Workshop on Architecture Research using FPGA Platforms*, 2006.

[36] C. Kachris1 and C. Kulkarni, "Configurable transactional memory," in *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2007.

[37] S. Grinberg and S. Weiss, "Investigation of transactional memory using fpgas," in *IEEE 24th Convention of Electrical and Electronics Engineers in Israel*, 2006.

[38] M. Labrecque and J. G. Steffan, "Nettm: Faster and easier synchronization for soft multicores via transactional memory," in *Proceedings of International Symposium on Field Programmable Gate Arrays*, 2011.

[39] A. Lenharth, D. Nguyen, and K. Pingali, "Priority queues are not good concurrent priority schedulers," tech. rep., The University of Texas at Austin, 2011.

[40] D. Chakrabarti, Y. Zhan, and C. Faloutsos, "R-mat: A recursive model for graph mining," in *Proceedings of the 2004 SIAM International Conference on Data Mining*, 2004.

[41] Desktop 4th Generation Intel Core Processor Family, Desktop Intel Pentium Processor Family, and Desktop Intel Celeron Processor Family: Specification Update (Revision 014).

[42] http://www.openfabric.org.