BionicDB: Fast and Power-Efficient OLTP on FPGA

Kangnyeon Kim
University of Toronto
knkim@cs.toronto.edu

Ryan Johnson
Amazon Web Services
frj@amazon.com

Ippokratis Pandis
Amazon Web Services
ippo@amazon.com

Abstract

Hardware specialization has been considered as a promising way to overcome the power wall, ushering in heterogeneous computing paradigm. Meanwhile, several trends, such as cloud computing and advanced FPGA technology, are converging to eliminate the barriers to custom hardware deployment, allowing it to be both technologically and economically feasible. In this paper, we conduct a pioneering study of hardware specialization for OLTP databases and present a fast and power-efficient transaction processing system built on FPGA, called BionicDB. With an order of magnitude higher power-efficiency inherently offered by FPGA, BionicDB performs faster or comparable to state-of-the-art software OLTP system by accelerating indexing and inter-worker communication.

1 INTRODUCTION

For the past decade, multicore has been a dominant scaling path. However, multicore hardware is getting more and more stagnant over time due to the growing scalability pressure and continuing power wall [16, 21]. To tackle the situation, hardware specialization has garnered a great deal of attention. It is widely regarded as a way to provide substantial power saving, while providing application-specific acceleration at the expense of generality [22, 30, 32, 34, 38, 44, 45].

In the meantime, field-programmable gate array (FPGA) technology has been matured, making custom hardware deployment more viable and economical. Its reconfigurability can return economic gains in accommodating quickly changing application demands by lifting manufacturing burden; we can reconfigure hardware to update or patch on-the-fly, as we do so for software. More importantly, inherent power efficiency of FPGA provides a great opportunity to overcome the power wall. Although running at low clock frequency, fine-grained, massive parallelism of FPGA could potentially compensate the low clock frequency with suitable custom hardware design on top.

For these reasons, datacenters increasingly integrate FPGA as a primary platform to run various custom accelerators for some data-intensive applications, such as search engine, deep learning and OLAP databases [4, 38]. Characteristically, those are compute-bound, dataflow applications. For such workloads, FPGA's massive fine-grained parallelism holds great promise for compute acceleration while CPU's limited and stagnant parallelism becomes a major computation bottleneck in dealing with huge data volume.

However, hardware specialization for transaction processing (OLTP) has been rarely explored. It has even been regarded as questionable because of OLTP's very different characteristics from previous cases: OLTP is generally bound by memory stalls and communication, rather than computation. In this situation, computation acceleration does not promise meaningful performance gain. Meanwhile, multicore CPU is returning diminishing performance gain with growing power consumption, putting scalability efforts at risk.

Inspired by the findings and technology trends, we conclude that it is imperative to explore an OLTP-oriented hardware that is power-efficient to operate under restrictive power budget and fast enough to meet performance requirement at the same time. FPGA can easily satisfy the power goal, but the question is how fast it can be while preserving the power efficiency.

In this paper, we report the design and implementation of BionicDB on FPGA as a case of hardware specialization for OLTP. BionicDB is an OLTP-oriented hardware optimized for in-memory, partitioned databases. Preserving the inherent power efficiency of FPGA, it achieves high performance thanks to various OLTP-oriented custom hardware that accelerate 1) index and 2) inter-worker communication. Also, it takes a hybrid processor-accelerator (software-hardware) architecture to complement each other. Our experimental results show that BionicDB can achieve an order of magnitude power saving while providing competitive performance compared to state-of-the-art software system; with the same number of worker threads, BionicDB can be faster by up to 4.5x when fully utilized and maintains comparable performance in TPC-C transactions where BionicDB is substantially underutilized.

The main contributions of this paper are as follows.

- We establish a holistic strategy for fast and power-efficient OLTP through hardware specialization.
- We propose index pipelining and transaction interleaving for index acceleartion.
- We suggest on-chip message-passing for faster inter-worker communication in partitioned databases.
- We show that hybrid processor-accelerator (software-hardware) approach is required for OLTP.

The remainder of this paper is organized as follows. Section 2 covers background on FPGA. Section 3 discusses design decisions, laying the foundations of BionicDB. Section 4 describes how BionicDB works in detail. Section 5 evaluates BionicDB, comparing to a software OLTP system. Section 6 summarizes related work. Section 7 contains our conclusion and suggests possible future research directions.

2 FIELD PROGRAMMABLE GATE ARRAYS

FPGA is a reconfigurable hardware platform and becoming an enabling technology for hardware specialization for a wide variety of application. Its main advantages are low power consumption and massive fine-grained parallelism that can be leveraged for acceleration. While CPU still remains as a central processing resource, certain CPU-unfriendly work can be offloaded to custom accelerator built on FPGAs for higher efficiency.

The main building blocks of FPGA that enable reconfigurability are lookup tables (LUT), flip-flops (FF) and programmable routing fabric. The first two components are used to implement logic functions, and the routing fabric interconnects them programmably. In addition to the programmable elements, modern FPGA chips commonly include hardwired blocks for higher efficiency, such as

Challenge	Solution
Memory stalls	Index pipelining
Memory stalls	Transaction interleaving
Inter-worker communication	On-chip message-passing over DORA
Heavy control-flow	Hybrid processor-accelerator design

Table 1: Design summary

block RAMs (BRAMs), digital signal processors (DSPs) and even embedded processors. Hardware can be designed with hardware description language (HDL) and CAD tools provided by a vendor compile HDL code into a bitstream that physically implements the hardware design on a target chip.

FPGA trades efficiency for reconfigurability, largely because of the area/speed/power overhead from programmable fabric, residing in a middle ground between ASIC and CPU in terms of efficiency and flexibility; ASIC provides the highest efficiency with lowest flexibility, while CPU provides the opposite. But FPGA can be an excellent platform for applications where the volume of production does not justify ASIC manufacturing. In datacenters where servers are dynamically re-purposed for a changing set of applications, a FPGA-augmented server can reprogram both software and hardware at runtime, allowing more efficient resource provisioning [38]. It also provides a chance to free up a number of CPUs from cycle-consuming jobs and spare them for more suitable (SW-friendly) tasks, returning higher datacenter efficiency.

3 DESIGN

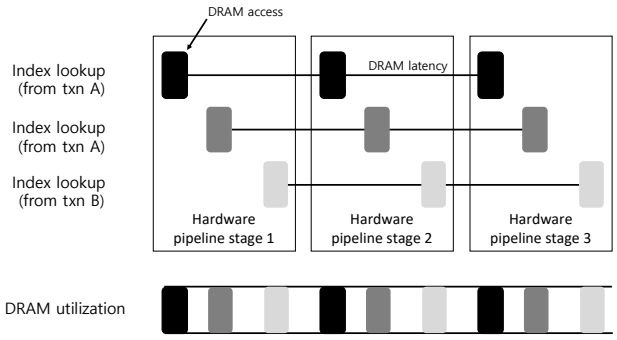
BionicDB aims to provide 1) power saving 2) and high performance at the same time for OLTP. Software-only systems on low-power processors usually end up trading one for another; [40] reports that an ARM processor sacrifices performance by 3x with merely 25% energy efficiency gain, compared to a Xeon processor when running a high-performance in-memory OLTP. Therefore, we leverage hardware specialization with FPGA to achieve both goals. Table 1 summarizes the specific challenges and solutions to address them.

3.1 Acceleration Strategy

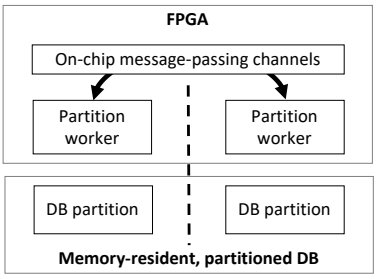
We start from understanding bottlenecks in modern OLTP to identify "what to accelerate" and establish a strategy on "how to accelerate". Typically, OLTP systems serve massively concurrent transactions that make small random updates to databases [1, 20]. Such workload characteristic naturally gives high pressure on indexing and thread coordination.

Index. Although OLTP databases heavily rely on index, the CPU is frequently bound by memory stalls from dependent pointer chasing within an index probe, wasting huge amounts of cycles. For example, a recent study reported roughly 40% overhead from index in a state-of-the-art software OLTP system [24]. But existing software solutions are limited to overcome the memory wall [19, 25]: 1) cache optimizations and bigger cache are easily undermined by OLTP's access randomness; 2) prefetching often fails to hide memory latency due to the lack of computation to overlap with; 3) software pipelining is inefficient with irregularity; 4) and group/dynamic prefetching is bound by the limited size of the instruction window of a CPU.

BionicDB alleviates the memory stalls through index pipelining. The key concept of index pipelining is to overlap multiple index accesses through hardware pipelining. It can provide higher memory-level parallelism, thereby improving single-worker performance. Also, we aim to exploit index parallelism not only



(a) Index acceleration by overlapping index operations.



(b) Fast on-chip message-passing for partitioned DB.

Figure 1: OLTP acceleration strategy

within a transaction, but also across transactions. For that, transaction interleaving captures inter-transaction index parallelism, improving the utilization of index pipelining. Figure 1a illustrates how we can achieve higher memory-level parallelism with these techniques.

Communication. Partitioned databases can simplify the hardware complexity; theoretically, database resources are core-private eliminating the burden of complex thread coordination, such as concurrent index implementation. For that reason, we adopt DORA [35, 36] which is a scalable partitioned database.

In regard to performance, it has very low overhead for a single-partition transaction because inter-worker communication does not take place at all. However, it requires inter-worker communication overhead when a transaction spans across partitions (multi-site transaction). Recent studies [7, 8, 12] have revisited message-passing semantic for partitioned databases, but software message-passing can involve 1) memory latency when a message is evicted from cache and 2) thread synchronization at concurrent message queues that can be a scalability bottleneck. Despite the problems, there is no alternative or a bypass for software, because the shared-memory is the only available communication semantic in most CPUs. In other words, the shared-memory bottlenecks are inevitable with software message-passing even when application-level communication semantic is purely message-passing.

For faster communication in partitioned database, As illustrated in Figure 1b, BionicDB provides an on-chip message-passing method that can completely eliminate the overhead of software message-passing. As a result, multisite transactions, a remaining concern in partitioned databases, can be accelerated.

3.2 Hybrid Hardware-Software Approach

While software-only is inefficient for the challenges of memory stalls and communication, the heavy control-flow in transaction logic, such as conditional branches and dynamic loops, makes

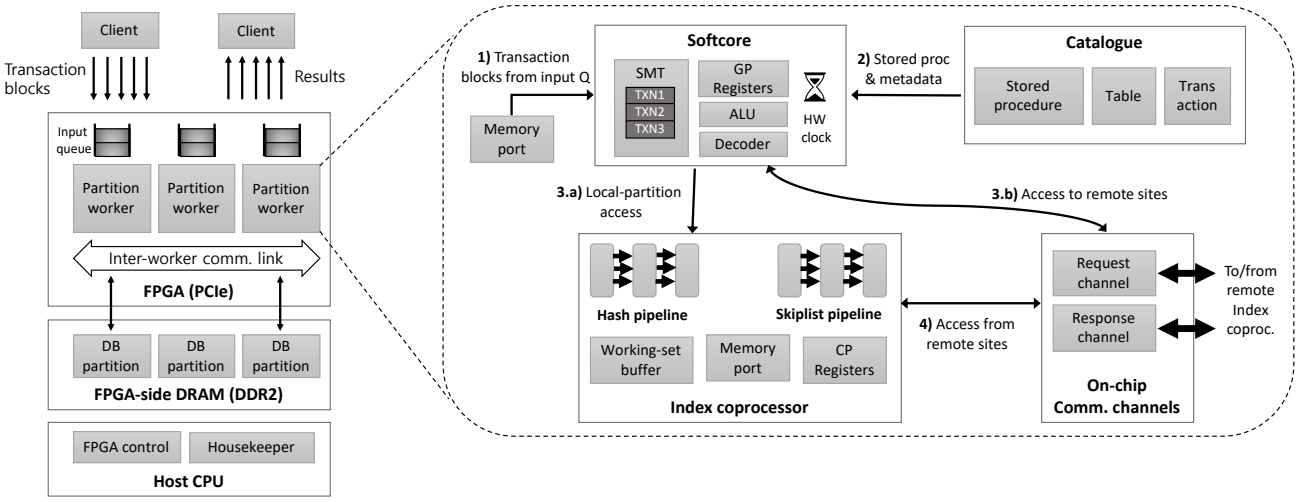


Figure 2: Architecture of BionicDB and the processing flow within a partition worker: 1) ingesting an input transaction block from memory, 2) fetching stored procedure code and metadata from the catalogue, 3.a) dispatching an index operation to the local index coprocessor 3.b) or to a remote site through the on-chip communication channels and 4) processing an index operation from remote workers.

general-purpose processor technology indispensable. This naturally leads to hybrid processor-accelerator approach that can deal with the inefficiency of both hardware-only (dynamic control-flow) and software-only (memory stalls, inter-worker communication) solutions.

The next question is how to integrate them. In our hardware environment where FPGA is connected through PCIe, the excessive PCIe latency (1us) between FPGA and host CPU can absorb most acceleration benefits. Therefore, we implement a custom softcore (microprocessor built around reconfigurable fabric) and acceleration fabric altogether on a PCIe-attached FPGA chip for tight integration, moving the majority of the OLTP components. The mission of the softcore is to execute pre-compiled stored procedures, while interacting with the index/communication acceleration fabric. As a result, hardware and software can closely collaborate, complementing each other. We design a custom core from the scratch, rather than using a vendor-provided softcore, to easily implement custom features and to integrate with the acceleration fabric more efficiently.

Although unexplored in this work, upcoming in-socket FPGA hardware [33] where FPGA is placed closer to host CPU is promising for hybrid software-hardware approach in that it allows to re-use existing software ecosystem, while enabling fast FPGA roundtrips thanks to low-latency interconnects, such as QPI.

4 ARCHITECTURE

4.1 Hardware Description

We build BionicDB on Micron HC-2 machine (formerly, Convey HC-2)[1] which contains four Xilinx Virtex-5 LX330 FPGA chips on a PCIe card and two-socket Intel Xeon X5670 processors. The memory subsystem of the FPGA card includes 64GB of DDR2 RAM, 8 memory controllers and 16 scatter-gather DIMMs. It can provide up to 80GB/s memory bandwidth for random 64-bit accesses when its 16 scatter-gather DIMMs are fully utilized with all FPGA chips enabled. However, current implementation of BionicDB uses only a single FPGA chip out of 4 and 8 DIMMs out of 16, bound by the maximum bandwidth of 10GB/s. It is also worth noting that in-memory OLTP is typically bound by memory latency, rather than bandwidth, due to small random accesses.

[1] https://www.micron.com/about/about-convey-computer-accelerator-products/hc-series

4.2 Overview

Figure 2 illustrates the architecture of BionicDB. BionicDB implements OLTP functionalities (stored procedure execution, indexing, concurrency control and transaction management) on a PCIe-attached FPGA chip, leaving a few background housekeeping jobs (signaling FPGA to start/stop, memory management, interactions with clients) to the host CPU. The database is partitioned, entirely residing in FPGA-side on-board DRAM. Each partition is accessed by a single partition worker in FPGA, and we fit multiple partition workers on a FPGA chip as many as its logic capacity allows for thread-level parallelism. The main components of a partition worker are 1) stored procedure execution engine and 2) acceleration fabric. The former includes the softcore and the catalogue that stores stored procedures and metadata, and the latter is composed of a local index coprocessor and communication channels.

We briefly describe the processing flow during transaction execution. The client should upload a pre-compiled stored procedure along with all metadata to the catalogue in advance. After that, the client can submit a transaction block to BionicDB for executing the transaction. Then, a partition worker fetches the transaction block from its input queue (step 1 in Figure 2). Once a transaction block arrives, the softcore fetches the corresponding stored procedure code from the catalogue and executes it (step 2 in Figure 2). During the execution of a stored procedure, it makes a database access by asynchronously passing an index request to the local index coprocessor or a remote site through the communication channels (step 3 in Figure 2). Meanwhile, the index operations issued by the local softcore (foreground requests) can be overlapped in the index coprocessor. The index operations from remote sites (background requests) also can be overlapped in the index coprocessor (step 4 in Figure 2). After a batch of transactions is entirely executed, the transactions are committed in serial order.

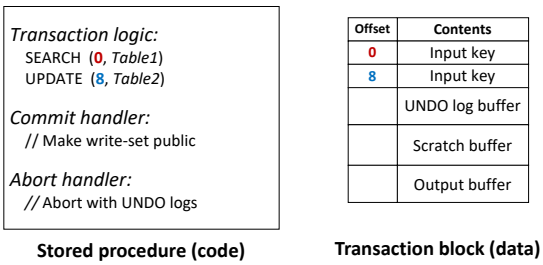


Figure 3: Stored procedure interface.

Instruction	Type
INSERT	DB
SEARCH	DB
SCAN	DB
UPDATE	DB
REMOVE	DB
ADD/SUB/MUL/DIV/MOV	CPU
CMP	CPU
LOAD/STORE	CPU
JMP/BE/BLE/BLT/BGT/BGE	CPU
RET	CPU
COMMIT/ABORT	CPU

Table 2: BionicDB instructions. DB instructions invoke index operations provided by the index coprocessor.

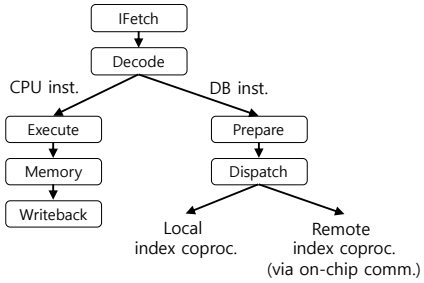


Figure 4: Instruction execution steps of the softcore.

4.3 Softcore

In BionicDB, a transaction can be executed by invoking an associated stored procedure that is pre-compiled with BionicDB instructions (will be described in this section). Then, the softcore executes the instructions while interacting with the index accelerator where most performance gains come from.

Stored procedure execution. A stored procedure is composed of three parts: 1) transaction logic code, 2) commit handler and 3) abort handler. The first part represents the original transaction logic and the rest implements a commit protocol (currently, commit/abort handlers should be defined by users). Depending on the execution results during transaction logic, BionicDB moves on to either the commit or the abort handler.

Once a stored procedure is registered, a client can submit a transaction block to BionicDB to invoke the transaction at runtime. A transaction block contains a transaction ID, input data and buffers for result sets, intermediate data and transaction logs. When it receives a transaction block, the softcore executes the matching stored procedure code with the transaction ID, using the input data and buffers in the transaction block. In the example shown in Figure 3, the stored procedure (left) tries to search a tuple with a key at offset 0 and update a tuple with a key at offset 8 in the transaction block (right).

When generating a stored procedure, a compiler should translate SQL statements into machine code, as Hekaton does with T-SQL [14] (we used manually-written stored procedures, as the compiler is beyond the scope of this paper). A client can register a new transaction or change an existing one by uploading the stored procedure code to BionicDB along with metadata to work with, such as transaction information and table schema. It does not require FPGA reconfiguration that involves hours-long synthesis, so BionicDB can accommodate workload changes quickly.

Instruction set architecture. The instruction set of BionicDB is composed of a subset of typical CPU instructions and DB instructions that encapsulate index operations (see Table 2). Figure 4 briefly illustrates how the softcore processes each instruction type during stored procedure execution. CPU instructions are directly executed by the softcore in five steps as simple RISC CPUs do: Instruction Fetch, Decode, Execute, Memory and Writeback. We ruled out instruction pipelining and out-of-order execution as previous studies showed that such features do not translate into meaningful improvement in OLTP [6, 17].

DB instructions are added for invoking indexing services provided by the index coprocessor. For DB instructions, the softcore collects metadata in Prepare stage, such as index type (hash or skiplist) and transaction begin timestamp. In the next stage (Dispatch), the softcore passes the DB instruction with the metadata to the local index coprocessor or a remote one via on-chip communication channels, depending on the destination partition. DB instructions are asynchronously forwarded for overlapping multiple index operations.

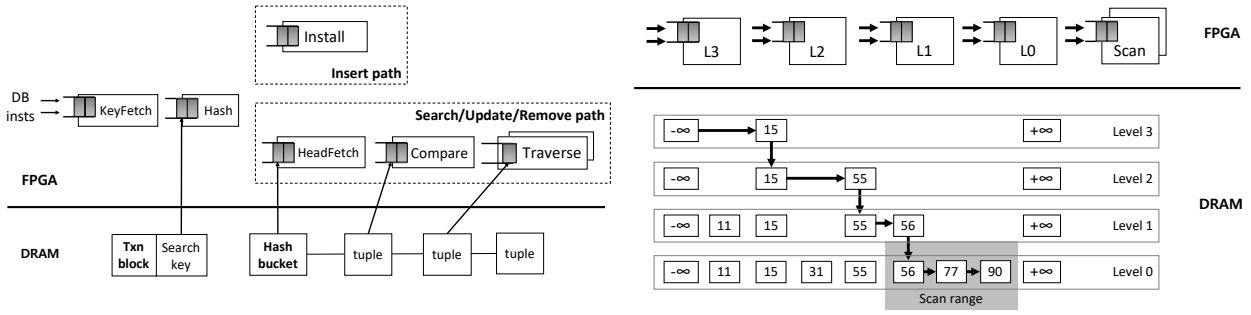
General-purpose registers (GP registers) are available to store data or pointers, and special-purpose registers, including a program counter (PC) and a status register that contains carry/zero/sign/overflow flags, are also available as typical CPUs. Additionally, coprocessor registers (CP registers) are provided; the result of a DB instruction is returned asynchronously to a CP register specified in the DB instruction. Then, the softcore can copy the result from the CP register to a GP register by executing a RET instruction. Therefore, a DB instruction must be paired with a RET instruction on the same CP register. The GP/CP register files are implemented on BRAMs, instead of flip-flops, for resource efficiency, and 256 GP/CP registers are provided to a single softcore. General-purpose cache memory was not implemented.

The addressing mode is base-offset. At the beginning of a transaction, a BionicDB worker sets a base address register with the start address of a transaction block and reaches memory locations within the block by adding the base address and an offset value. The offset value could be either the content of a GP register or an immediate value which is inlined to an instruction directly.

4.4 Index Coprocessor

The index coprocessor processes DB instructions from the local softcore or a remote one. The key technique for acceleration is hardware pipelining. We decompose an index algorithm into sub-functions that can work in parallel and implement each sub-function as a hardware pipeline stage. Each pipeline stage is a finite-state machine that is awakened on source data arrival from off-chip DRAM, performs a certain task, issues memory requests designating the next stage as a destination and moves on to the next incoming instruction. There could be multiple outstanding DB instructions between neighboring stages.

Index pipelining can overlap multiple index accesses, achieving higher memory-level parallelism. But pipeline hazards can happen when different stages access the same memory location. We describe details of each index and discuss how to prevent pipeline hazards in the following sections. Figure 5 shows the



(a) Hash index. Sub-functions of hash index operations are mapped to hardware pipeline stages.

(b) Skiplist. Sub-ranges of skiplist are mapped to hardware pipeline stages. Scan is done by dedicated modules.

Figure 5: Index pipelining to overlap multiple index accesses. Hash index deals with point access and skiplist provides range scan.

index pipelining with hash and skiplist indexes. For point access, BionicDB implements hash index that processes INSERT, SEARCH, UPDATE and REMOVE instructions. Skiplist index can handle SCAN instructions for range query, INSERT and REMOVE instructions. Both indexes support variable-length key.

4.4.1 Hash index for point access We illustrate how the HW hash index works in Figure 5a. KeyFetch stage takes a DB instruction and issues memory requests to fetch a search key from a transaction block, designating Hash stage as a destination. Then, the memory response containing the search key is queued at Hash stage. Hash stage takes the search key from its input queue, computes a hash value and loads a hash table entry with the hash value. (we use Sdbm hash function[2] for its minimal use of hardware resources; it requires neither a huge lookup table nor an expensive operation like modulo).

At this point, Hash stage forwards an INSERT instruction to Install stage and the other instructions to HeadFetch stage. For INSERT instruction, Install stage takes a hash table entry and appends a new tuple to the entry. For SEARCH/UPDATE/REMOVE instructions, HeadFetch stage checks the content of hash table entry. If the value is NULL indicating that there is no tuple installed, it returns a “NotFound" message to the destination CP register specified in the DB instruction. Otherwise, it issues memory requests to read the first item of a hash bucket. KeyComp stage compares the search key against the first item’s key. If they match, it examines visibility check (will be explained in Section 4.7) for concurrency control, otherwise, it passes the instruction to Traverse stage to follow a hash conflict chain.

Traverse stage follows a hash conflict chain until it finds a matching tuple or reaches the end of the chain. Unlike other stages that contain only computation without memory stalls, this stage could involve multiple memory stalls. Thus, Traverse could take much longer time with poorly distributed hashing. We decouple this stage from Compare stage so that an instruction that follows a long hash conflict chain does not block succeeding instructions that terminate at Compare stage. If hash conflict is frequent, multiple Traverse stages could be populated for balanced dataflow over the pipeline. Also, a hash function with good distribution and sufficiently large hash table could minimize the activation of Traverse stage by reducing hash conflicts.

Pipeline hazards and prevention. There are two hazards in hash index: insert-after-insert and search-after-insert (see Figure 6). In the former case, lost update can happen between in-flight inserts accessing the same hash table entry, if the following request reads a hash table entry before the preceding request updates it (see Figure 6a). Although not illustrated in the figure, a SEARCH instruction could read an inconsistent hash table entry at Hash stage before Install stage finishes updating it. In both cases, the reason for the hazards is that Hash and Install stages access the same hash table entry without coordination.

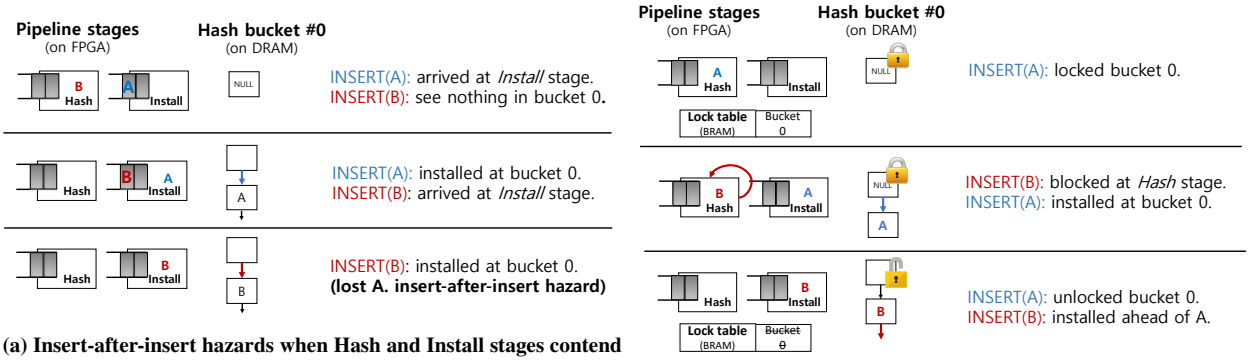
To prevent the hazards, we use a pipeline-stall based coordination scheme. BionicDB tracks the hash values of in-flight instructions that passed Hash stage in a lock table on BRAM (content-addressable memory could be used for faster check), and Hash stage checks the lock table before accessing a hash bucket. If a duplicate hash value is detected, it blocks until the lock entry disappears (see Figure 6b). The lock table entry is deleted by terminal stages when an instruction completes.

4.4.2 Skiplist for range scan For range scan, we choose skiplist because it is efficient with index pipelining for range scan (will be discussed later in this section). A skiplist index is a collection of linked lists at multiple levels [37]. A skiplist node (tower) includes a tuple and an array of pointers to the next towers at different levels. The bottom link is a list of all towers, while upper ones are short-cuts and contain towers with probabilistic distribution; the number of towers decreases exponentially from the bottom through the top, offering logarithmic time complexity on average for traversal by preferring to traverse taller towers first.

Traversal pipelining. BionicDB maps skiplist indexing on deeply pipelined datapath. Each pipeline stage covers an exclusive range of levels. We illustrate an example of index pipelining for skiplist in Figure 5b. In the example, four levels are mapped on four skiplist pipeline stages. The level 3 stage starts pointer chasing horizontally at the top level, stops at the tower having key 15 because its next tower contains an upper bound and drills down to the lower level. As it goes out-of-range, it passes the instruction to the next pipeline stage (level 2) and immediately moves on to the next incoming instruction. Meanwhile, level 2 stage takes over the instruction and performs same tasks within its coverage. The level 0 stage that exclusively owns the bottom level finally receives the request and locates the right tower for the given key, which is 56. For balanced pipelining, it is important to customize range binding properly. If skiplist towers are substantially sparser at upper levels than lower ones, upper pipeline stages could be assigned larger ranges.

Insert. During the traversal for INSERT instruction, the insert path, the pointers to the predecessor and successor towers at each

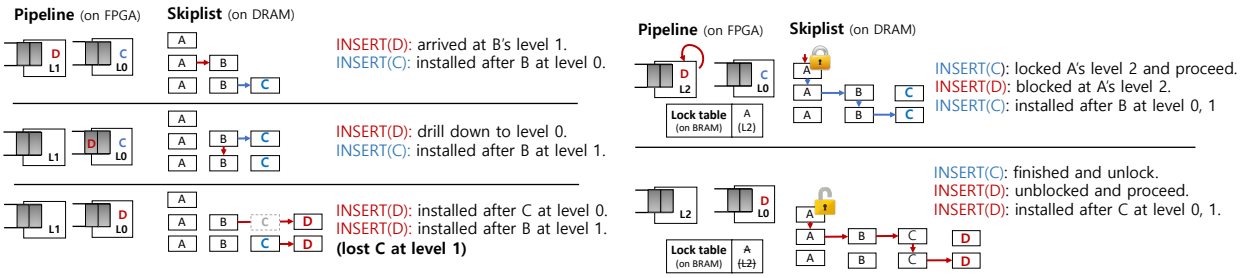
[2] http://www.cse.yorku.ca/~oz/hash.html



(a) Insert-after-insert hazards when Hash and Install stages contend on the same bucket.

(b) Hazard prevention with pipeline stalls.

Figure 6: Hash index hazards and prevention.



(a) Insert-after-insert hazard when different stages contend on the same predecessor node.

(b) Hazard prevention with pipeline stalls.

Figure 7: Skiplist hazard and prevention.

level below the new tower's height, is recorded in BRAMs. From the bottom level, the bottom-level stage installs a new tower on the insert path.

Scan. Scan does not require to record the path because it only needs to reach the bottom level. We assign a dedicated scanner module after the bottom-level stage. The bottom-level stage passes a scan request with the first tower in the scan range to the scanner. Then, a scanner collects committed and visible tuples within the scan range. The result scan set is stored in a designated buffer space in a transaction block (on DRAM), and the size of scan set is returned through a CP register. Like Traverse stage in hash index, decoupled scanner stages prevent long-running scan requests from blocking following requests that complete at the bottom-level stage. When necessary, redundant scanners could distribute heavy scan loads, ameliorating unbalanced dataflow.

Insert-insert hazard and prevention. In skiplist, only insert-insert hazard can happen. When consecutive inserts traverse down through common path, the traversal path of the following request could be invalidated when the preceding one has overwritten it with a new tower's address. Figure 7a shows an example of the insert-insert hazard. In the example, INSERT(D) request drills down a wrong tower (which is B) because the preceding insert C has not updated the tower at level 1. In turn, INSERT(D) installs a new tower on its inconsistent path, consequently, C is lost at level 1.

To prevent insert-insert hazard in skiplist, we apply a similar blocking policy to the hash index to prevent the skiplist hazard (see Figure 7b). For all in-flight INSERTs, we record the entry points of insert paths in a lock table. All skiplist pipelines should check the lock table before switching to the next tower or a lower level and block when encountering a locked traversal path. The lock is eliminated by the terminal stage (the bottom-level stage) when an INSERT operation completes.

Stall-free range scan. Unlike insert, skiplist scan can be stall-free, allowing efficient pipelining. A scan request might traverse down an inconsistent image of the skiplist, when missing towers installed by previous in-flight inserts at short-cut levels. However, all towers inserted previously are visible at the bottom link because a single dedicated pipeline stage serializes requests in order. After the bottom-level stage passes a scan request, a scanner starts pointer chasing from the first tower in the scan range. Therefore, scan does not miss any previous inserts in the end. During scan, the scanner could see towers inserted after scan started, but they are ignored by timestamp-based visibility check. In terms of performance, a scan request could choose a slower traversal path by missing recent short-cuts, but higher concurrency is more beneficial for overall performance.

4.5 Transaction Interleaving

The index coprocessor could be heavily underutilized with insufficient intra-transaction index parallelism when transactions are serially executed. As an extreme example, a single record transaction in key-value workloads that issues only a single index request can entirely eliminate the chance to overlap index accesses. To prevent the underutilization, the softcore exploits inter-transaction index parallelism by interleaving multiple transactions. We describe the details of transaction interleaving and discuss what factors could affect its efficiency.

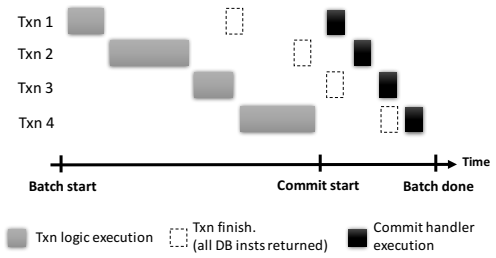


Figure 8: Transaction interleaving and 2 phase execution.

Transaction grouping. Whenever a transaction block enters the softcore, the softcore tries to add the transaction to the current batch as follows. It first checks the metadata in the catalogue to figure out how many GP/CP registers are required for the transaction. If there are enough registers remaining, the transaction joins the current batch with an exclusive range of GP/CP registers allocated. For the purpose, the softcore maintains the base GP/CP register addresses and renames the registers in the stored procedure instructions by adding the base register addresses. After that, the base register addresses are updated to indicate next available range, and the stored procedure is executed immediately. If the allocation fails, the current batch is closed and the new transaction is scheduled after the current batch commits.

Two-phase execution and interleaving. Figure 8 shows how interleaving is done. BionicDB executes transactions in two phases: 1) transaction logic and 2) commit/abort. Starting from the first transaction in a batch, the softcore executes a stored procedure code. When it reaches the end of a stored procedure, it saves the transaction context in a BRAM buffer, including program counter register, the base address of transaction block and register address range, and switches to the next transaction without waiting for outstanding DB instructions to complete.

When the first phase of a batch is finished by batch closure, it returns back to the first transaction and restores the context saved in the BRAM buffer. At this point, the program counter (PC) indicates the address of the first instruction of the commit handler. The commit handler then waits for all outstanding DB instruction to return. Depending on the results from them, the softcore continues to commit or jumps to the abort handler. Any exception, such as a DB instruction failure by CC or a voluntary abort, caught will trigger the abort handler. Then, the commit/abort handler performs a CC protocol to finish the transaction batch. After the second phase of a batch is finished, the softcore opens a new batch and executes the pending input transactions in the batch.

Discussion. With transaction interleaving, DB instructions across transactions can be overlapped. It is particularly beneficial for small transactions with insufficient intra-transaction index parallelism. Also, the overhead of transaction interleaving is marginal: transaction contexts are stored entirely in a context table on BRAMs, and a single switch takes 10 cycles to save current context and restore the next one from the context table.

The benefit of interleaving could be absorbed by data dependency within a transaction; data dependency is formed when a DB instruction requires an output from the previous DB instruction in the same transaction. Data dependency becomes a barrier that forces to wait for outstanding DB instructions to complete within a transaction, getting rid of the chance to overlap index requests in the next transactions. However, we found that current interleaving is still promising for certain workloads where transactions are small and data dependency-free (for example, YCSB transactions). To deal with heavy data dependency, it might be helpful to switch between transactions dynamically whenever desired, but current implementation does not support such dynamic scheduling.

4.6 On-chip Message-passing Channels

Partitioned databases, such as H-store and DORA, can minimize synchronization overhead across concurrent threads with single-threading policy; a partition is not thread-safe, but it is guaranteed to be accessed by a dedicated worker exclusively. Therefore, a partition worker cannot directly access a remote partition, instead, it should send a request message to a remote site. Then, a delegate worker on the remote partition processes the request on behalf of the initiator and returns a response message back. Hence, inter-worker message-passing is required when a transaction spans over multiple partitions.

To reduce the communication overhead and, thereby, accelerate cross-partition transactions, BionicDB provides on-chip message-passing channels. Unlike software message-passing that can include memory latency and thread synchronization during communication, request/response messages are directly exchanged between workers at on-chip speed without memory round-trips and thread synchronization. Despite the low clock frequency of FPGA, the communication latency is still low due to low-overhead message-passing protocol. The latency in exchanging a single pair of request and response messages takes 6 cycles in total (the latency could vary slightly depending on congestion). We believe that message-passing is a suitable communication semantic for single-threaded databases where data is strongly isolated between partition workers, while most CPUs have to conform the shared-memory semantic for backward compatibility.

Multisite transaction handling. Let us explain how a multisite transaction is processed using with the on-chip communication channels in detail. Each worker is assigned a communication link that consists of request and response channels. When the softcore decodes a DB instruction and finds out that the target partition is remote, it creates a request packet with the instruction and sends it through the request channel asynchronously. A request packet is piggybacked with a transaction timestamp for concurrency control and source/destination worker IDs for routing. At a remote site, a background unit monitors the request channel and catches an inbound request packet having a matching worker ID. In turn, the DB instruction is dispatched immediately to the remote index coprocessor as a background request. At this point, local (foreground) and remote (background) requests can be overlapped in the index coprocessor. But, it does not cause inconsistency because the index coprocessor is capable of dealing with pipeline anomalies and concurrency control. After completion of a background request, its result is sent back to the initiator through the response channel. The response message returned to the initiator is written back to the destination CP register asynchronously, and the initiator's softcore takes the result later when executing a RET instruction with the CP register.

Scaling on-chip message-passing. We discuss several scaling issues, leaving them to future work. First, the current topology of the on-chip communication is crossbar which does not scale. When scaling up BionicDB on datacenter-grade FPGAs that can fit tens or hundreds of BionicDB workers in a single chip (resource consumption will be provided in Section 5), a scalable on-chip communication topology, such as ring or tree, will be required. Also, BionicDB is currently a single-chip, single-node system.

Given that typical FPGA offerings have merely around tens of GBytes of on-board DRAM, it is vital to scale BionicDB across multiple FPGA nodes in a shared-nothing cluster like H-store [23]. Fortunately, some cloud FPGA services, such as AWS F1, support multi-chip, multi-node deployment [4]. Therefore, the message-passing channels should be diversified with additional connectivities for inter-node communication.

4.7 Concurrency Control

Currently, BionicDB employs a variant of the basic single-version timestamp CC [11]. Its strengths and weaknesses are well-known [47], but we use it for the sake of simplicity as CC is not the focus of this paper. A transaction is assigned a hardware timestamp value at the beginning. During transaction lifecycle, DB instructions issued by the transaction are packed with the transaction timestamp and passed to the index coprocessor. Then, visibility check is performed as follows by the index coprocessor against a matching tuple with the search key.

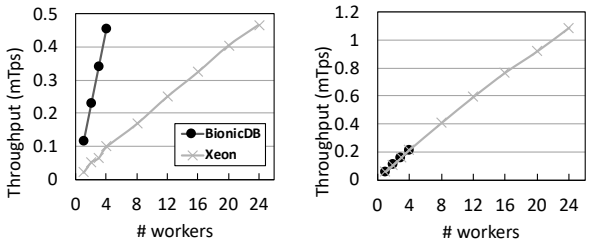
Visibility check. Each tuple is associated with latest read and write timestamps, and they are compared against the transaction timestamp when accessed. The read permission is granted on a tuple having a lower write time, and the write permission is granted on a tuple having a lower read time. If the transaction is the latest reader, read time of the tuple is updated with the transaction timestamp immediately. There are minor deviations from the original algorithm. First, any access to an uncommitted (dirty) tuple during runtime is blindly rejected without the care of the serial order and triggers transaction abort immediately. Also, read set is not buffered. If the second access to a previously visited tuple is denied by concurrent updates, the transaction should abort to ensure repeatable read.

If a DB instruction passes the visibility check, the address of the matching tuple with a “success" return code is written back to the CP register specified in the DB instruction. Otherwise, an error code is written. An UPDATE instruction only marks the dirty bit and returns without actual modification. With the tuple address returned, the softcore later performs in-place update after backing up the original tuple in UNDO log buffer in a transaction block. An REMOVE instruction returns after marking both dirty and tombstone bits.

Commit protocol. When a commit handler takes over the softcore after transaction logic execution, it collects the results from CP registers by executing RET instruction. If an error code is detected, the softcore jumps to the abort handler. After making sure successful execution, the softcore iterates write-set to cleanup dirty marks and overwrite their write time with the transaction’s begin timestamp. If aborted, the abort handler restores the write-set from the UNDO logs in a transaction block and removes the dirty marks.

4.8 Logging and Recovery

Although logging and recovery are currently missing, we discuss a possible way to guarantee durability. BionicDB can adopt the VoltDB’s command logging approach for recovery [29]. After executed by BionicDB, each transaction block contains the commit state and the commit timestamp of the transaction, preserving the input arguments. BionicDB can recover the database simply by re-executing the committed transaction blocks in the commit timestamp order. To this end, the host CPU should store the input transaction blocks that were processed in a durable storage before returning them to clients. After system failure, the host CPU should load the last checkpoint image and the persisted transaction blocks (command logs) from the disk. Then, the host CPU can replay the committed transaction blocks, ignoring the uncommitted ones. It must replay them in the commit timestamp order to ensure correct recovery. After recovery, the hardware clocks of BionicDB should be re-initialized to the latest commit timestamp, and the host CPU can signal BionicDB to resume transaction processing.



(a) YCSB-C (read-only). **(b) TPC-C NewOrder and Payment mix.**

Figure 9: Comparison of overall performance of BionicDB to Silo on Xeon (4 chips).

5 EVALUATION

This section evaluates BionicDB for the following purposes.

- We show that BionicDB can provide high performance and substantial power saving in OLTP workloads, comparing to state-of-the-art SW system.
- We figure out how much speedup comes from each acceleration feature and which factors can limit the performance.

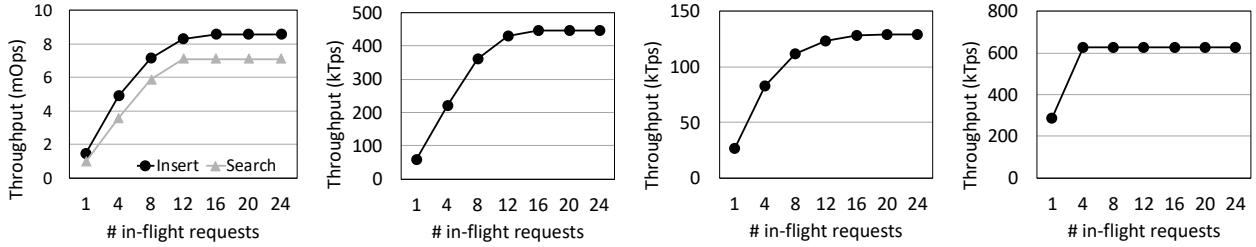
5.1 Experimental Setup

In all experiments, we fit the entire databases in DRAM. For BionicDB, we populated input transaction blocks in advance by host CPUs (ideally, remote clients should submit transaction blocks through network cards, and BionicDB should process them without the intervention of the host CPU). Unless stated otherwise, we report the aggregate throughput of four BionicDB workers.

5.2 Hardware

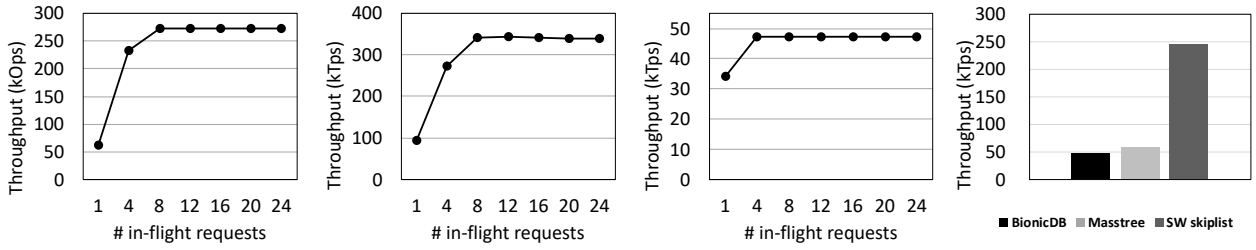
Xilinx Virtex5 LX330. We built BionicDB on a single Virtex5 LX330 FPGA chip which was manufactured with 65nm technology and released a decade ago. It contains merely 200K logic cells, allowing to fit only four BionicDB workers within a single chip. However, recent datacenter-grade FPGA chips containing millions of logic cells, such as Xilinx Virtex Ultrascale+ used in AWS F1 instances or Intel Arria 10, could accommodate tens or hundreds of BionicDB workers with a single chip [2, 5], providing ample thread-level parallelism. The clock frequency of BionicDB was set to 125MHz.

Intel Xeon E7 4807. To compare BionicDB to a software engine, we ran Silo [43] on 4 hexa-core Xeon chips (24 physical cores in total). Its clock frequency is 1.87GHz, and each core is assigned private 32KB L1I/D cache and 256KB L2 cache. L3 cache is 18MB and shared by all cores on a chip. In choosing a CPU for comparison, our focus was making sure the CPU is roughly in the same generation with Virtex5 for fairness. Xeon E7 4807 was released later (in 2011) than Virtex5 and manufactured with more advanced process technology (32nm).



(a) KeyValue (b) YCSB-C (read-only) (c) TPC-C NewOrder (d) TPC-C Payment

Figure 10: Throughput of hash index with varying index parallelism.



(a) Insert (b) Point query (c) Scan (50 tuples) (d) vs. SW (scan)

Figure 11: Throughput of skiplist with varying index parallelism and comparison of scan performance to SW.

5.3 Benchmarks

TPC-C. We ran a mix of the TPC-C NewOrder and Payment (50:50). Most transactions are local, but 1% of the NewOrder and 15% of the Payment transactions are cross-partition by default. We partitioned the database by Warehouse and replicated Item table which is read-only across partitions. The number of partitions was fixed to the number of workers. We modified the Payment transaction by enforcing to pick up a customer with *customer id* for both BionicDB and Silo.

YCSB. The YCSB transaction includes 16 independent DB accesses with no data dependency. The YCSB table schema consists of a 8B integer key and 1KB payload. We populated 300K records per partition, and the size of a partition is 300MB (the number of partitions is equal to the the number of workers). We ran the YCSB-C (read-only) and YCSB-E transactions. The YCSB-B (read-intensive) was ommitted due to similar results to YCSB-C. We modified the YCSB-E transaction to make it scan-only and fixed the scan range to be 50 records which is the average scan length with the default workload setting.

5.4 Overall Performance

We compared the overall performance of BionicDB and Silo in Figure 9. As explained in Section 5.1, we only presented BionicDB from 1 to 4 workers because of limited logic capacity of Virtex5 FPGA. In Figure 9a, we ran YCSB-C transactions. With the same number of workers, BionicDB outperformed Silo by 4.5x. Silo matched the throughput of 4 BionicDB workers with 24 cores enabled over 4 CPU chips. YCSB-C transaction exploited both intra-, inter-transaction parallelism provided by index pipelining and transaction interleaving thanks to sufficient index parallelism and the absence of data dependency that permits aggressive transaction interleaving.

We also ran a mix of the TPC-C NewOrder and Payment (50:50) and plotted in Figure 9b. Unlike the YCSB result, BionicDB achieved only comparable performance to Silo with the same number of workers because of insufficient index parallelism of Payment transaction (only 4 index lookups) and heavy data dependency that nearly eliminated the chance for transaction interleaving (in fact, the TPC-C transactions were executed almost in serial).

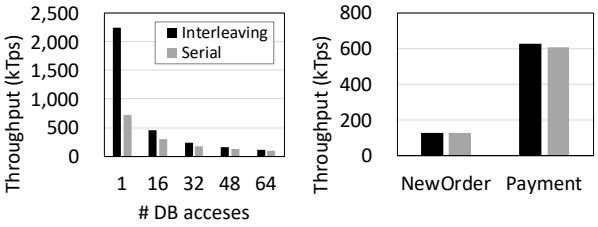
These results reveal both the promise and challenge of BionicDB at the same time. BionicDB can outperform state-of-the-art SW OLTP system when fully utilized, but it can be underutilized by insufficient index parallelism and heavy data dependency. However, the results confirm that BionicDB can provide still competitive performance even with much lower frequency (125MHz) and a limited microprocessor (no instruction-level parallelism and general-purpose cache memory), by taking a suitable acceleration approach for OLTP. From the next section, we evaluate each acceleration component to understand their impacts quantitatively.

5.5 Impact of Index Pipelining

We evaluated index pipelining in Figure 10 and Figure 11, We controlled the degree of coprocessor parallelism by changing the maximum number of in-flight DB requests over the index coprocessor. To focus on the index coprocessor, all experiments in this section run local transactions only.

Hash. In Figure 10a, we ran a non-transactional key-value workload to see the peak performance of the hash index. A single transaction repeated issuing 60 insert/search instructions in bulk for 20,000 times (in total 1.2M inserts and searches). The peak performance for insert and search reached 8.5Mops and 7Mops, respectively, and saturated between 12 and 16 index parallelism. This implies that there were 3 or 4 in-flight requests between pipeline stages in average.

We plotted the throughput of the YCSB-C (read-only) and the TPC-C NewOrder transactions in Figure 10b and Figure 10c. Both figures show the similar trend with the previous result in the



(a) YCSB-C (b) TPC-C

Figure 12: Transaction interleaving vs. serial execution.

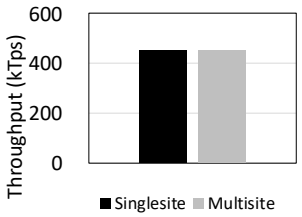


Figure 13: Single-site (100% local access) vs. multi-site transaction (75% remote, 25% local access).

KV workload. It proves that the transactions have sufficient intra-transaction parallelism. Figure 10d shows the performance of the TPC-C Payment transactions. Due to the limited index parallelism, the performance did not improve after 4 requests.

Skiplist. We instantiated 8-stage skiplist pipelines with an extra scanner module. The maximum height of a skiplist tower was set to 20. Figure 11a shows the performance of sequential loading. The pipeline was saturated at 8 in-flight requests. It increased sharply from 1 to 4 and modestly from 4 to 8 requests. The reason for lower pipeline parallelism than hash index is that skiplist pipeline stages contain multiple memory stalls during horizontal pointer chasing, leaving nearly no in-flight requests between stages, unlike the hash pipeline. Hence, the index parallelism was bound by the depth of pipeline. The modest curve from 4 to 8 in-flight requests is related to unbalanced dataflow over skiplist pipeline. Pipeline stages covering upper levels were less busy than lower levels as towers were sparser. Figure 11b shows the performance of point query. It shows similar trend with the previous result, but the throughput is higher because tower installation was skipped.

To measure scan performance, we ran modified YCSB-E transactions (Section 5.1) and presented the performance in Figure 11c. We can find that the pipelining efficiency was deteriorated. This is because a single scanner became a bottleneck in the pipeline. Thus, heavy scan loads should be distributed over multiple scanners for balanced pipelining. In Figure 11d, we compared the scan performance to Masstree and SW skiplist on the Xeon CPU introduced in Section 5.1. The number of workers was four across all indexes. Since its pipelining efficiency was largely eliminated, HW skiplist was slower than Masstree by 20% and SW skiplist by 5x. With extra FPGA resources, HW skiplist could be improved with deeper pipelining and multiple scanners (in this experiment, pipelining depth and the number of scanners were bound by current FPGA resource). To catch up with SW skiplist, at least 5 scanners would be required.

5.6 Impact of Transaction Interleaving

Figure 12 shows the performance comparison between transaction interleaving and serial execution. In this experiment, all transactions were local. In Figure 12a, we changed the the size of transaction footprint, or intra-transaction parallelism, by varying the number of DB instructions in a YCSB-C transaction. With a single record transaction, transaction interleaving was 3x faster than serial execution. In serial execution, the coprocessor was underutilized by small transactions. Whereas, transaction interleaving was able to overlap index requests across transactions, resulting in much higher utilization. As we increase the number of requests, the performance gap shrank. This is because coprocessor was utilized better in serial execution with more intra-transaction parallelism.

Figure 12b illustrates the throughput of the TPC-C NewOrder and Payment transactions. In both transactions, there was no noticeable difference because heavy data dependency hindered transaction interleaving. The NewOrder transaction had enough intra-transaction index parallelism, but failed to exploit inter-transaction parallelism due to data dependency, leaving the coprocessor idle during commit phase. The Payment transaction's case was worse. It was not able to exploit both intra-, inter-transaction parallelism because of limited parallelism and data dependency combined, leading to significant underutilization. It is also evidenced by the modest difference in overall performance from the YCSB-C transaction that includes four times more index roundtrips.

5.7 Impact of On-chip Message-passing

Latency analysis. Table 3 compares the communication latency of on-chip message-passing against software message-passing. We assumed 20ns and 80ns for the latency of shared-cache and DDR3, respectively. Based on the estimates, we calculated the total communication latency for exchanging a single request/response pair that takes two iterations of message-passing, assuming that cache communication takes two cache reads on a modified-state cache-line, and DRAM communication takes two rounds of memory read and write. Despite the slow frequency (125MHz), the latency of on-chip message-passing is 48ns which is comparable to cache communication and much faster than DDR3 communication. If cache miss happens, the latency of software communication can rapidly increase. Also, we did not take synchronization cost into account, giving favor to software message-passing. In practice, software message-passing could suffer much higher communication latency in the presence of cache misses and thread contention.

Primitive		Latency (ns)	Total comm. delay (ns)
On-chip MP		24	48
Software MP	L3 cache	20	40
	DDR3	80	320

Table 3: Latencies of message-passing methods.

Throughput of cross-partition transactions. It is widely known that frequent multisite transactions in non-partitionable workloads can be a serious bottleneck in partitioning-based systems. We now evaluate the performance of multi-site transactions to see if the on-chip message-passing can accelerate them. We ran the cross-partition YCSB-C transactions with uniform random keys and plotted the throughput in Figure 13. In the cross-partition transaction, 75% of DB accesses are remote, and the rests are local accesses. As an ideal case, we also plotted the performance of local transactions that do not involve inter-worker communication at all. The result shows that on-chip, message-passing communication

imposed negligible overhead, achieving almost same performance with the ideal case. In all other workloads, we observed the same result. This confirms that the message-passing, on-chip communication of BionicDB eliminated the overhead of communication and accelerated cross-partition transactions effectively.

5.8 Power Consumption and Resource Utilization

We estimated the power consumption of BionicDB on Virtex5 LX330 with Xilinx Power Estimator (XPE). The total power consumption was approximately 11.5W. The thermal design power (TDP) of a single Xeon E7 4807 is 95W, and the aggregate TDP of four chips is 380W.

Module	Flip-flops	Look-up tables	Block RAMs
Hash	12,932	14,504	24
Skiplist	27,300	35,968	36
Softcore	7,080	8,796	12
Catalogue	1,484	1,964	8
Communication	2,482	3,191	8
Memory arbiters	1,192	5,800	0
HC-2 modules	98,507	76,639	103
Virtex5 LX330 Total	207,360	207,360	288
Utilization	72%	70%	70%

Table 4: Resource utilization of BionicDB with 4 workers

Table 4 reports the resource utilization of BionicDB with four workers on a Virtex5 LX-330 chip. The entire hardware design consumed around 70% of FFs, LUTs and BRAMs. Almost half of the total logic cells were taken by HC-2's infrastructures, such as host interface, crossbar memory interconnects and a custom processor which were not used by BionicDB at all. Four BionicDB workers consumed approximately 70k LUTs and 53k FFs in total. Out of BionicDB resources, the skiplist index consumed almost 50%, and hash index consumed around 20%. The stored procedure execution modules, the softcore and catalogue, took only 15% thanks to the resource-efficient design choices.

6 RELATED WORK

Many existing database accelerators have focused on offloading SQL computation. They commonly exploit massive fined-grained parallelism for compute acceleration, avoiding instruction decoding overhead and memory wall. (application-specific functions are wired on datapath directly, and data stream flow over them) [18, 27].

Oracle SPARC M7 processor integrates in-memory database acceleration fabric (DAX), offloading filtering and de/compression [3]. Ibex [44] is a FPGA-based MySQL storage engine that offloads filtering and aggregation functions to FPGA. It provides the notion of near-data processing by placing the SQL accelerator between CPU and SSD. Bharat et al., suggested FPGA coprocessor for OLAP acceleration in in-memory HTAP system [41]. They offloaded filtering and data decompression to FPGA coprocessors on a PCIe board while the host CPU focus on transaction processing. The FPGA coprocessor directly accesses compressed memory-resident data, decompresses them, and performs filtering. To saturate PCIe bandwith, data scan tiles are replicated. Q100 [45] is a dataflow hardware for SQL. ASIC tiles that perform SQL operators are provided, along with a custom instruction set to control data stream over the tiles. DoppioDB offloaded regular expression evaluation and analytic operators on Intel's Xeon-FPGA machine [33, 39]. Do et al. explored in-storage SQL processing inside flash memory SSD [15], but the system used embedded processors for acceleration without custom hardware. Kocberber et al. suggested Widx which is an on-chip hash indexing accelerator for OLAP workloads [26]. They identified hash index lookup as the main bottleneck due to poor memory-level parallelism in OLAP workloads and offloaded the it to on-chip acceleration fabric.

For stream acceleration, Glacier [30] implemented SQL circuits on an FPGA fabric, installed on datapath between network card and host CPU. Handshake join [42] is a highly parallel stream join algorithm with massive parallel processing resources such as FPGA or GPGPU. The key idea is fine-grained parallel join processing between two streams flowing from the opposite directions. A large number of tuple pairs are evaluated at once, exploiting massive computation parallelism. FQP [31] is a flexible stream query processor that can support changing query logic without FPGA reconfiguration. It implemented filtering and stream join operators.

For transaction processing, there have been a few studies with custom hardware approach. Cipherbase uses FPGA as a coprocessor for security, offloading expression evaluation, some index operations and en/decryption [9]. Also, there have been hardware key-value store systems with hash index [10, 13, 22, 46] and transactional graph processing on FPGA [28]. However, standalone transaction processing hardware is still missing in the landscape. Many existing SQL accelerators do not solve OLTP's main problems: memory stalls and communication. Low-end processor, such as ARM, could be a power efficient option, but it often sacrifices performance [40].

7 CONCLUSION

In this paper, we explored hardware specialization for OLTP and presented the design and implementation of BionicDB built on FPGA. We discussed index pipelining and transaction interleaving for index acceleration, and on-chip message-passing for faster communication in partitioned databases. Also, we argued that OLTP requires tightly integrated SW-HW architecture. The experimental results confirmed that transaction processing can be done at substantially lower power cost while providing competitive performance. Possible future directions include scaling up BionicDB on a modern FPGA chip and scaling out over multiple chips and nodes.

References

[1] 1992. TPC Transcation Processing Performance Council. http://www.tpc.org/default.asp

[2] 2014. Xilinx Ultrascale datasheet. https://www.xilinx.com/support/documentation/data_sheets/ds890-ultrascale-overview.pdf

[3] 2015. Oracle SPARC M7 processor. http://www.oracle.com/us/products/servers-storage/sparc-m7-processor-ds-2687041.pdf

[4] 2017. AWS EC2 F1 Instances. https://aws.amazon.com/ec2/instance-types/f1/

[5] 2017. Intel Stratix 10 FPGA. https://www.altera.com/products/fpga/stratix-series/stratix-10/overview.html

[6] Anastassia Ailamaki, David J. DeWitt, Mark D. Hill, and David A. Wood. 1999. DBMSs on a Modern Processor: Where Does Time Go?. In *Proceedings of the 25th International Conference on Very Large Data Bases (VLDB '99)*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 266–277. http://dl.acm.org/citation.cfm?id=645925.671662

[7] Raja Appuswamy, Angelos C. Anadiotis, Danica Porobic, Mustafa K. Iman, and Anastasia Ailamaki. 2017. Analyzing the Impact of System Architecture on the Scalability of OLTP Engines for High-contention Workloads. *Proc. VLDB Endow.* 11, 2 (Oct. 2017), 121–134. https://doi.org/10.14778/3149193.3149194

[8] Raja Appuswamy, Manos Karpathiotakis, Danica Porobic, and Anastasia Ailamaki. 2017. The Case For Heterogeneous HTAP. (2017).

[9] Arvind Arasu, Spyros Blanas, Ken Eguro, Raghav Kaushik, Donald Kossmann, Ravi Ramamurthy, and Ramarathnam Venkatesan. 2013. Orthogonal Security With Cipherbase, In 6th Biennial Conference on Innovative Data Systems Research (CIDR'13). https://www.microsoft.com/en-us/research/publication/orthogonal-security-with-cipherbase/

[10] Arvind Arasu, Ken Eguro, Raghav Kaushik, Donald Kossmann, Pingfan Meng, Vineet Pandey, and Ravi Ramamurthy. 2017. Concerto: A High Concurrency Key-Value Store with Integrity. In *Proceedings of the 2017 ACM International Conference on Management of Data (SIGMOD '17)*. ACM, New York, NY, USA, 251–266. https://doi.org/10.1145/3035918.3064030

[11] Philip A. Bernstein and Nathan Goodman. 1981. Concurrency Control in Distributed Database Systems. *ACM Comput. Surv.* 13, 2 (June 1981), 185–221. https://doi.org/10.1145/356842.356846

[12] Irina Calciu, Dave Dice, Tim Harris, Maurice Herlihy, Alex Kogan, Virendra Marathe, and Mark Moir. 2013. Message Passing or Shared Memory: Evaluating the Delegation Abstraction for Multicores. In *Proceedings of the 17th International Conference on Principles of Distributed Systems - Volume 8304 (OPODIS 2013)*. Springer-Verlag New York, Inc., New York, NY, USA, 83–97. https://doi.org/10.1007/978-3-319-03850-6_7

[13] Sai Rahul Chalamalasetti, Kevin Lim, Mitch Wright, Alvin AuYoung, Parthasarathy Ranganathan, and Martin Margala. 2013. An FPGA Memcached Appliance. In *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '13)*. ACM, New York, NY, USA, 245–254. https://doi.org/10.1145/2435264.2435306

[14] Cristian Diaconu, Craig Freedman, Erik Ismert, Per-Ake Larson, Pravin Mittal, Ryan Stonecipher, Nitin Verma, and Mike Zwilling. 2013. Hekaton: SQL Server's Memory-optimized OLTP Engine. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1243–1254. https://doi.org/10.1145/2463676.2463710

[15] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. 2013. Query Processing on Smart SSDs: Opportunities and Challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data (SIGMOD '13)*. ACM, New York, NY, USA, 1221–1230. https://doi.org/10.1145/2463676.2465295

[16] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark silicon and the end of multicore scaling. In *ISCA*.

[17] Brian Gold, Anastassia Ailamaki, Larry Huston, and Babak Falsafi. 2005. Accelerating Database Operators Using a Network Processor. In *Proceedings of the 1st International Workshop on Data Management on New Hardware (DaMoN '05)*. ACM, New York, NY, USA, Article 1. https://doi.org/10.1145/1114252.1114260

[18] Rehan Hameed, Wajahat Qadeer, Megan Wachs, Omid Azizi, Alex Solomatnikov, Benjamin C. Lee, Stephen Richardson, Christos Kozyrakis, and Mark Horowitz. 2010. Understanding Sources of Inefficiency in General-purpose Chips. In *Proceedings of the 37th Annual International Symposium on Computer Architecture (ISCA '10)*. ACM, New York, NY, USA, 37–47. https://doi.org/10.1145/1815961.1815968

[19] Nikos Hardavellas, Ippokratis Pandis, Ryan Johnson, Naju Mancheril, Anastasia Ailamaki, and Babak Falsafi. 2007. Database Servers on Chip Multiprocessors: Limitations and Opportunities. (01 2007).

[20] Stavros Harizopoulos, Daniel J. Abadi, Samuel Madden, and Michael Stonebraker. 2008. OLTP Through the Looking Glass, and What We Found There. In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data (SIGMOD '08)*. ACM, New York, NY, USA, 981–992. https://doi.org/10.1145/1376616.1376713

[21] Mark D. Hill and Michael R. Marty. 2008. Amdahl's Law in the Multicore Era. *Computer* 41 (2008), 33–38. Issue 7.

[22] Z. Istvan, G. Alonso, M. Blott, and K. Vissers. 2013. A flexible hash table design for 10GBPS key-value stores on FPGAS. In *2013 23rd International Conference on Field programmable Logic and Applications*. 1–8.

[23] Evan P.C. Jones, Daniel J. Abadi, and Samuel Madden. 2010. Low Overhead Concurrency Control for Partitioned Main Memory Databases. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of Data (SIGMOD '10)*. ACM, New York, NY, USA, 603–614. https://doi.org/10.1145/1807167.1807233

[24] Kangnyeon Kim, Tianzheng Wang, Ryan Johnson, and Ippokratis Pandis. 2016. ERMIA: Fast Memory-Optimized Database System for Heterogeneous Workloads. In *Proceedings of the 2016 International Conference on Management of Data (SIGMOD '16)*. ACM, New York, NY, USA, 1675–1687.

[25] Onur Kocberber, Babak Falsafi, and Boris Grot. 2015. Asynchronous Memory Access Chaining. *Proc. VLDB Endow.* 9, 4 (Dec. 2015), 252–263. https://doi.org/10.14778/2856318.2856321

[26] Onur Kocberber, Boris Grot, Javier Picorel, Babak Falsafi, Kevin Lim, and Parthasarathy Ranganathan. 2013. Meet the Walkers: Accelerating Index Traversals for In-memory Databases. In *Proceedings of the 46th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-46)*. ACM, New York, NY, USA, 468–479. https://doi.org/10.1145/2540708.2540748

[27] H. T. Kung. 1982. Why systolic architectures? *Computer* 15, 1 (Jan 1982), 37–46. https://doi.org/10.1109/MC.1982.1653825

[28] 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

[29] N. Malviya, A. Weisberg, S. Madden, and M. Stonebraker. 2014. Rethinking main memory OLTP recovery. In *2014 IEEE 30th International Conference on Data Engineering*. 604–615. https://doi.org/10.1109/ICDE.2014.6816685

[30] Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on Wires: A Query Compiler for FPGAs. *Proc. VLDB Endow.* 2, 1 (Aug. 2009), 229–240. https://doi.org/10.14778/1687627.1687654

[31] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. 2013. Flexible Query Processor on FPGAs. *Proc. VLDB Endow.* 6, 12 (Aug. 2013), 1310–1313. https://doi.org/10.14778/2536274.2536303

[32] Eriko Nurvitadhi, Ganesh Venkatesh, Jaewoong Sim, Debbie Marr, Randy Huang, Jason Ong Gee Hock, Yeong Tat Liew, Krishnan Srivatsan, Duncan Moss, Suchit Subhaschandra, and Guy Boudoukh. 2017. Can FPGAs Beat GPUs in Accelerating Next-Generation Deep Neural Networks?. In *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays (FPGA '17)*. ACM, New York, NY, USA, 5–14. https://doi.org/10.1145/3020078.3021740

[33] N. Oliver, R. R. Sharma, S. Chang, B. Chitlur, E. Garcia, J. Grecco, A. Grier, N. Ijih, Y. Liu, P. Marolia, H. Mitchel, S. Subhaschandra, A. Sheiman, T. Whisonant, and P. Gupta. 2011. A Reconfigurable Computing System Based on a Cache-Coherent Fabric. In *2011 International Conference on Reconfigurable Computing and FPGAs*. 80–85. https://doi.org/10.1109/ReConFig.2011.4

[34] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric Chung. 2015. Accelerating Deep Convolutional Neural Networks Using Specialized Hardware. https://www.microsoft.com/en-us/research/publication/accelerating-deep-convolutional-neural-networks-using-specialized-hardware/

[35] Ippokratis Pandis, Ryan Johnson, Nikos Hardavellas, and Anastasia Ailamaki. 2010. Data-oriented Transaction Execution. *Proc. VLDB Endow.* 3, 1-2 (Sept. 2010), 928–939. https://doi.org/10.14778/1920841.1920959

[36] Ippokratis Pandis, Pinar Tözün, Ryan Johnson, and Anastasia Ailamaki. 2011. PLP: Page Latch-free Shared-everything OLTP. *Proc. VLDB Endow.* 4, 10 (July 2011), 610–621. https://doi.org/10.14778/2021017.2021019

[37] William Pugh. 1990. Skip Lists: A Probabilistic Alternative to Balanced Trees. *Commun. ACM* 33, 6 (June 1990), 668–676. https://doi.org/10.1145/78973.78977

[38] Andrew Putnam, Adrian Caulfield, Eric 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 Xiao, and Doug Burger. 2015. A Reconfigurable Fabric for Accelerating Large-Scale Datacenter Services. *IEEE Micro* 35 (May 2015), 10–22.

[39] D. Sidler, M. Owaida, Z. IstvÃąn, K. Kara, and G. Alonso. 2017. doppioDB: A hardware accelerated database. In *2017 27th International Conference on Field Programmable Logic and Applications (FPL)*. 1–1. https://doi.org/10.23919/FPL.2017.8056864

[40] Utku Sirin, Raja Appuswamy, and Anastasia Ailamaki. 2016. OLTP on a Server-grade ARM: Power, Throughput and Latency Comparison. In *Proceedings of the 12th International Workshop on Data Management on New Hardware (DaMoN '16)*. ACM, New York, NY, USA, Article 10, 7 pages. https://doi.org/10.1145/2933349.2933359

[41] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database Analytics Acceleration Using FPGAs. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques (PACT '12)*. ACM, New York, NY, USA, 411–420. https://doi.org/10.1145/2370816.2370874

[42] Jens Teubner and Rene Mueller. 2011. How Soccer Players Would Do Stream Joins. In *Proceedings of the 2011 ACM SIGMOD International Conference on Management of Data (SIGMOD '11)*. ACM, New York, NY, USA, 625–636. https://doi.org/10.1145/1989323.1989389

[43] 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

[44] Louis Woods, Zsolt István, and Gustavo Alonso. 2014. Ibex: An Intelligent Storage Engine with Support for Advanced SQL Offloading. *Proc. VLDB Endow.* 7, 11 (July 2014), 963–974. https://doi.org/10.14778/2732967.2732972

[45] Lisa Wu, Andrea Lottarini, Timothy K. Paine, Martha A. Kim, and Kenneth A. Ross. 2014. Q100: The Architecture and Design of a Database Processing Unit. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '14)*. ACM, New York, NY, USA, 255–268. https://doi.org/10.1145/2541940.2541961

[46] Shuotao Xu, Sungjin Lee, Sang-Woo Jun, Ming Liu, Jamey Hicks, and Arvind. 2016. Bluecache: A Scalable Distributed Flash-based Key-value Store. *Proc. VLDB Endow.* 10, 4 (Nov. 2016), 301–312. https://doi.org/10.14778/3025111.3025113

[47] 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