Parallelizing Data Processing on FPGAs with Shifter Lists

LOUIS WOODS, Systems Group, Dept. of Computer Science, ETH Zürich, Switzerland
GUSTAVO ALONSO, Systems Group, Dept. of Computer Science, ETH Zürich, Switzerland
JENS TEUBNER, DBIS Group, Dept. of Computer Science, TU Dortmund University, Germany

Parallelism is currently seen as a mechanism to minimize the impact of the power and heat dissipation problems encountered in modern hardware. Data parallelism—based on partitioning the data—and pipeline parallelism—based on partitioning the computation—are the two main approaches to leverage parallelism on a wide range of hardware platforms.

Unfortunately, not all data processing problems are susceptible to either of those strategies. An example is the *skyline* operator [Börzsönyi et al. 2001], which computes the set of Pareto-optimal points within a multi-dimensional data set. Existing approaches to parallelize the skyline operator are based on data parallelism. As a result, they suffer from a high overhead when merging intermediate results because of the lack of a global view of the problem inherent to partitioning the input data.

In this paper, we show how to combine pipeline with data parallelism on an FPGA for a more efficient utilization of the available hardware parallelism. As we show in our experiments, skyline computation using our proposed technique scales linearly with the number of processing elements and the performance we achieve on a rather small FPGA is comparable to the one of a 64-core high-end server running a state-of-the-art *data parallel* implementation of *skyline* [Park et al. 2009].

The proposed approach to parallelize the skyline operator can be generalized to a wider range of data processing problems. We demonstrate this through a novel, highly parallel data structure, a *shifter list*, that can be efficiently implemented on an FPGA. The resulting template is easy to parameterize to implement a variety of computationally intensive operators such as *frequent items*, *n-closest pairs*, or *K-means*.

Categories and Subject Descriptors: H.2 [**Database Management**]: Systems

General Terms: Algorithms, Design, Performance

Additional Key Words and Phrases: FPGA, database, shifter list, pipeline, parallelism, skyline query, frequent items, n-closest pairs, K-means

ACM Reference Format:
Louis Woods, Gustavo Alonso, and Jens Teubner. Parallelizing Data Processing on FPGAs with Shifter Lists. *ACM Trans. Reconfig. Technol. Syst.* 0, 0, Article 0 (June 2014), 22 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

There has been an increasing amount of research and commercial systems that exploit heterogeneous, low power, and massively parallel co-processors to accelerate data processing operations. Vendors such as IBM/Netezza [IBM 2014] and Convey [Con-

This work was supported by the Enterprise Computing Center (ECC) at ETH Zürich (http://www.ecc.ethz.ch) and by Deutsche Forschungsgemeinschaft (DFG) within the Collaborative Research Center SFB 876 "Providing Information by Resource-Constrained Analysis."
Author's addresses: Louis Woods and Gustavo Alonso, Systems Group, Dept. of Computer Science, ETH Zürich, 8092 Zurich, Switzerland, {firstname.lastname}@inf.ethz.ch; Jens Teubner, DBIS Group, Dept. of Computer Science, TU Dortmund University, 44221 Dortmund, Germany, jens.teubner@cs.tu-dortmund.de.

DOI:http://dx.doi.org/10.1145/0000000.0000000

vey Computer 2014] equip their systems with hardware accelerators using FPGAs. However, it can be very difficult to turn the hardware's parallelism into performance, especially, if a given problem is not embarrassingly parallel and exhibits many data dependencies. Moreover, the lack of suitable abstractions and design patterns prevents the efforts invested in one solution to carry over from one application to another.

To address these issues, we propose a data structure—*shifter list*—that helps in the design of massively parallel and *scalable* algorithms for a number of different problems. Shifter lists combine data organization, computational power, and synchronization into a new parallel processing model that naturally supports the characteristics of FPGAs. In our model, we think of input data as a data stream that propagates through the shifter list, which itself is distributed over many processing elements. These processing elements—we call them *shifter list nodes*—are arranged as a pipeline and locally update the shifter list as input data flows by. The only communication required is between neighboring shifter list nodes, *i.e.*, nearest neighbor communication.

In this paper, we first discuss in detail an efficient and scalable FPGA implementation of the *skyline* operator as an instance of a shifter list application. Skyline computation is a good example where straightforward input data partitioning neither matches the complexity properties of the problem—linear in the input data volume, but quadratic in the (intermediate) skyline result—, nor does it fit the characteristics of modern parallel hardware. In contrast, we can partition the *working set* of a *block-nested-loops* (BNL) [Börzsönyi et al. 2001] variant (a commonly used algorithm to solve the skyline problem) and leverage the lightweight partitioning mechanisms across many shifter list nodes.

Based on this concrete skyline implementation, we explicitly identify shifter list properties and introduce some generalizations that allow us to apply this design pattern to a wider range of applications. We then specify a shifter list template, and show how other well-known data processing tasks such as *frequent items*, *n-closest pairs* or *K-means* could be mapped to this template.

2. SKYLINE QUERIES

In this section, we will define skyline queries, a popular software algorithm to solve skyline queries (the BNL algorithm [Börzsönyi et al. 2001]), and our modified version of BNL for parallel execution on an FPGA. Our intention here is to describe our approach of parallelizing BNL at a high level, before we discuss technical details later. To do so, we will take the liberty of digressing into the world of Lemmings[1].

2.1. The Lemming Skyline

Lemmings are primitive creatures that go on migrations in masses. On Lemmings Planet every year a challenge takes place among the Lemmings with the goal to identify the "best" Lemmings. Every Lemming has different skills: some are very strong but slow and clumsy, others are agile but neither strong nor fast, then again others are generalists that do not have a particular skill that they are best in but have multiple skills they are pretty good in. As the committee of the competition could not agree on a weighting function that would determine the best Lemmings, all Lemmings that are not *dominated* (cf. Definition 2.1) by any other Lemming are considered best. In other words, the winners are the (*Pareto-optimal*) Lemmings that are part of the Lemming *skyline* (cf. Definition 2.2).[2]

[1]As in the video game "Lemmings": http://www.dmadesign.org/

[2]According to Definition 2.1, if two Lemmings are equal in all dimensions, neither Lemming dominates the other. As a result, both Lemmings would be part of the *skyline*, given they are not dominated by any other skyline Lemmings.

Definition 2.1. A Lemming l_i dominates ($\prec$) another Lemming l_j iff every skill (dimension) of l_i is better or equal than the corresponding skill of l_j and at least one skill of l_i is strictly better than the corresponding skill of l_j.

Definition 2.2. Given a set of Lemmings $L = \{l_1, l_2, \ldots l_n\}$, the skyline query returns a set of Lemmings S, such that any Lemming $l_i \in S$ is not dominated by any other Lemming $l_j \in L$.

2.2. The Competition—1st Year (Best)

When the competition took place for the first time, the committee did have a formal definition for the set of best Lemmings but it was still unclear how to determine this set. Thus, in the absence of sophisticated logistic means, one committee member suggested the following simple algorithm. Initially, all Lemmings queue up in front of a bridge, as illustrated in Figure 1.

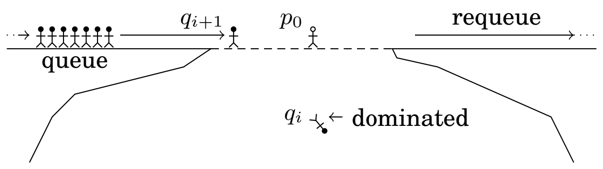

Fig. 1: Lemming skyline with *Best* [Torlone and Ciaccia 2002].

The first Lemming in the queue q_0 is considered a *potential* skyline Lemming p_0 and can advance onto the bridge. There, the *candidate* Lemming has to battle all other Lemmings in the queue $q_1 \ldots q_{n-1}$. A battle can have three possible outcomes. (1) p_0 dominates q_i. In this case, q_i will be pushed from the bridge and p_0 remains on its position to combat q_{i+1}. (2) q_i dominates p_0. Now, p_0 falls from the bridge and q_i becomes the new candidate Lemming p_0, *i.e.*, has to battle q_{i+1}. (3) If neither of the two Lemmings dominates the other, they are considered *incomparable*. In this case, p_0 stays on the bridge and q_i has to requeue.

The candidate Lemming p_0 has to remain on the bridge until it has fought all queued Lemmings once. When a challenger q_j confronts p_0 for the second time, we know that p_0 is not dominated by any other Lemming. Hence, p_0 is part of the Lemming skyline and can leave the bridge safely and q_j becomes the new p_0. The algorithm terminates when the queue is empty, *i.e.*, all dominated Lemmings have fallen from the bridge. The Lemmings still alive all belong to the Lemming *skyline*. This algorithm, known as *Best*, has been formally described in [Torlone and Ciaccia 2002].

2.3. The Competition—2nd Year (BNL)

The following year many new Lemmings were born and it was time to determine the Lemming skyline anew. The previous year some Lemmings complained that they had to spend too much time queuing. In particular, requeing was time-consuming and delayed the entire competition. To improve on this drawback, the set of candidate Lemmings was increased from 1 to w. The modified version of the algorithm is known as *block-nested-loops* (BNL) [Börzsönyi et al. 2001] and illustrated in Figure 2.

On the bridge there is room for a *working set* of w candidate Lemmings. A challenging Lemming q_i from the queue has to battle all candidate Lemmings on the bridge. If the challenging Lemming survives all battles, there are two possibilities. (1) If there are already w other candidate Lemmings on the bridge, q_i has to requeue. (2) Otherwise, q_i becomes a candidate Lemming p_i.

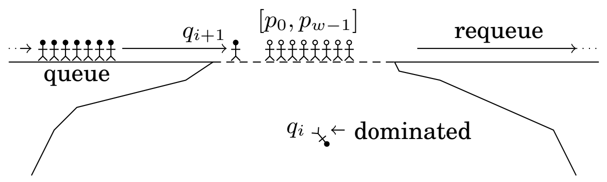

Fig. 2: Lemming skyline with *BNL* [Börzsönyi et al. 2001].

Unfortunately, now it is unclear when exactly a candidate Lemming has been on the bridge long enough to qualify as a true *skyline* Lemming. Luckily, the competition committee found a simple solution to this problem. After a Lemming q_i survives all candidate Lemmings on the bridge, it receives a *timestamp* independent of whether it becomes a candidate Lemming or has to requeue. A candidate Lemming p_i becomes a true *skyline* Lemming (and leaves the bridge) when it either encounters the first challenging Lemming q_j that has a larger timestamp or when the queue is empty. When Lemmings initially queue up for the first time, this timestamp is set to zero. A larger timestamp indicates that two Lemmings must have already competed against each other and since the queue is ordered, all following Lemmings in the queue will also have larger timestamps.

2.4. The Competition—3rd Year (Parallel BNL)

While the BNL algorithm used in the 2nd year significantly reduced the number of times that Lemmings had to requeue, there were new complaints coming from some Lemmings. In particular, candidate Lemmings criticized that most of the time on the bridge they were idle, waiting for their turn to battle the next challenger. Thus, in favor of higher throughput, the competition committee decided to slightly modify the BNL algorithm. The basic idea is that instead of one challenger q_i now up to w challengers $q_{(i+w-1)} \dots q_i$ are allowed on the bridge, and each challenger can battle a different candidate Lemming in parallel. This version of the algorithm is illustrated in Figure 3.

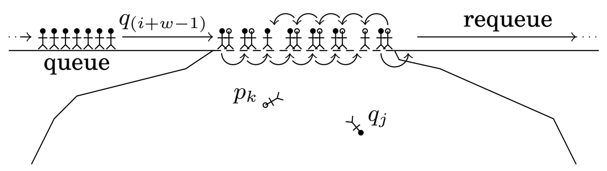

Fig. 3: Lemming skyline: parallel BNL for FPGAs.

To avoid chaos on the bridge the procedure is as follows: In each iteration there is a *shift phase* followed by a *evaluation phase*. In the *shift phase* all challenger Lemmings $q_{(i+w-1)} \dots q_i$ move one position to the right to face their next opponent (indicated by the lower arrows in the figure). This frees the leftmost position on the bridge and allows a new Lemming from the queue to step on the bridge every iteration. Then in the *evaluation phase* all w pairs of Lemmings battle concurrently. As can be seen in Figure 3, in some situations a Lemming will not have an opponent because the corresponding Lemming was previously dominated, *i.e.*, fell from the bridge. In that case, the Lemming does not need to battle in this iteration.

Once a challenging Lemming q_i safely reaches the right end of the bridge, it qualifies as a candidate Lemming if there is room on the bridge, otherwise it has to requeue. If during the *evaluation phase* a candidate Lemming p_i falls from the bridge, the other

Lemmings $p_{i+1} \ldots p_{w-1}$ to the right of that Lemming have to move up in the subsequent *shift phase* and fill the gap (indicated by the upper arrows in the figure), making room for new candidate Lemmings that reach the right end of the bridge.

Again, we can use timestamping to decide when candidate Lemmings turn into true skyline Lemmings and can leave the bridge. Since the order among the Lemmings on the bridge is maintained, it is always the leftmost candidate Lemming that may become the newest skyline member. Thus, candidate Lemmings begin on the right end of the bridge and then gradually move towards the left end, where they need to wait until they encounter a challenger with a larger timestamp.

3. IMPLEMENTATION—PARALLEL BNL WITH FPGAS

The parallelized BNL version, sketched previously in Section 2.4, exhibits properties such as *pipeline parallelism* and *nearest neighbor communication* that make it amenable to an FPGA implementation, the details of which we discuss in this section.

3.1. Pipeline of Processing Elements

To compute skyline queries with an FPGA, we assume the following setup. The FPGA reads input data from external DRAM and maintains a set of candidate skyline tuples inside the FPGA chip. Overflow tuples are written back to DRAM and processed in a later iteration. In BNL, each *input tuple* (a tuple read from DRAM) needs to be compared against every tuple of what we call the *working set* of a shifter list. With respect to our Lemmings example, the working set would consist of the candidate Lemmings on the bridge. This working set may contain several hundred tuples but we want to spend only a minimal number of clock cycles on each input tuple in order to achieve high throughput. Hence, we distribute the tuples of the working set over a pipeline of daisy-chained processing elements (shifter list nodes), as illustrated in Figure 4.

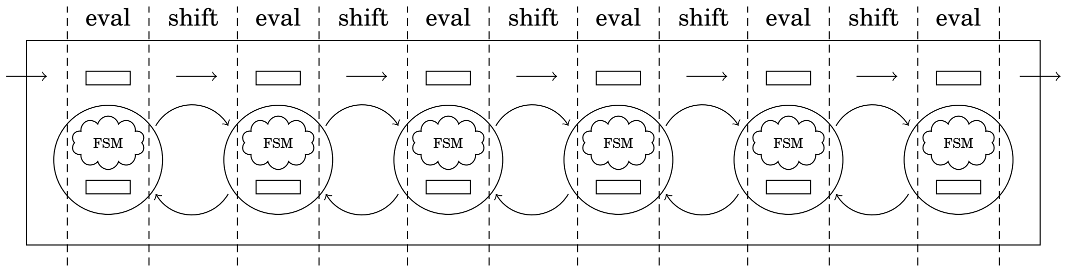

Fig. 4: Two-phase processing in parallel BNL. Working set items (*i.e.*, tuples with several dimensions) are distributed over a pipeline of shifter list nodes.

A shifter list node consists of one BRAM block to store tuples of the working set and a state machine that manipulates the working set data. An input tuple is fetched from DRAM and submitted to the first shifter list node in the pipeline, from where it is forwarded to the neighboring shifter list node after evaluation. Once an input tuple has propagated to the last shifter list node, it may be written back to DRAM into an *overflow queue* for processing in a subsequent round.

Thus, only the first and the last shifter list node directly interact with DRAM. Between the shifter list nodes, nearest neighbor communication is used, leading to a *scalable* solution with respect to the number of shifter list nodes since negative effects, *e.g.*, long communication paths or high fan-in/-out, can be avoided if the communication follows very simple topologies such as *pipelining* along a series of parallel units.

```
on each shifter list node do
    $q \leftarrow$ current input tuple ;
    $p \leftarrow$ local working set contents ;
    $s \leftarrow$ state of shifter list node ;
    if $q.valid$ then                                   /* next input tuple (challenger) */
        if $s = working\ set$ then                      /* local working set tuple (candidate) */
            if $q.timestamp > p.timestamp$ then
                $s \leftarrow$ output ;                 /* found skyline tuple */
            end
            else if $q.data \prec p.data$ then
                $s \leftarrow$ deleted ;                /* drop working set tuple */
            end
            else if $p.data \prec q.data$ then
                $q$.valid $\leftarrow$ false ;          /* drop input tuple */
            end
        end
        else if $s$ = free then                         /* add input tuple to working set */
            timestamp($q$) ;
            $p$.data $\leftarrow$ $q$.data ;
            $s \leftarrow$ working set ;
            $q$.valid $\leftarrow$ false ;
        end
    end
end
```

Fig. 5: Evaluation phase executed on each shifter list node.

3.2. Parallel BNL as Two-Phase Algorithm

As mentioned in Section 2.4, we can divide skyline computation into two phases: *(i)* an *evaluation phase* and *(ii)* a *shift phase*. During the *evaluation phase*, the next state is determined for each shifter list node (the exact definition of shifter list node states for BNL is subject of Section 3.3); but these changes are not applied before the *shift phase*, which is the phase that allows nearest neighbor communication. Those two phases run synchronously across the FPGA, as depicted in Figure 4.

Evaluation Phase. Figure 5 lists the partial algorithm that is executed locally on each shifter list node in the evaluation phase. It very closely resembles the global algorithm, *i.e.*, standard BNL. Only *boundary cases* have to be modified to obtain the code for node-local execution. For instance, node-local “overflow tuples” in our parallel BNL implementation have to be forwarded to the next shifter list node, rather than be written directly to an overflow queue as in the superordinate BNL skyline algorithm.

Shift Phase. All interactions between neighboring shifter list nodes are performed in the shift phase, displayed in Figure 6, which updates the global algorithm state based on the outcome of the evaluation phase. In essence, all input tuples are forwarded one shifter list node toward the right, whereas candidate results (working set tuples) move toward the left if there is space available. Since skyline candidates move toward the left, we report them on the leftmost shifter list node ν_0 once their timestamp condition has been satisfied. Likewise, on the rightmost shifter list node ν_{w-1}, we write input tuples to the overflow queue in DRAM if they were not invalidated during their journey along the pipeline of shifter list nodes, and cannot be inserted into the working set because there is no space.

```
foreach shifter list node $\nu_i$ do
    /* all skyline results are emitted on $\nu_0$ */
    if $i = 0 \wedge \nu_i$.state = output then
        emit $\nu_i$.working set.tuple as result ;
        $\nu_i$.state $\leftarrow$ deleted ;
    end
    if $i < w - 1$ then                                /* not last shifter list node */
        if $\nu_i$.state = deleted then
            /* move up candidates to left */
            $\nu_i$.working set $\leftarrow$ $\nu_{i+1}$.working set ;
            $\nu_i$.state $\leftarrow$ $\nu_{i+1}$.state ;
            $\nu_{i+1}$.state = deleted ;
        end
        /* challengers move one position to right */
        $\nu_{i+1}$.input tuple $\leftarrow$ $\nu_i$.input tuple ;
    end
    else                                 /* the last shifter list node (physically) */
        if $\nu_i$.state = deleted then
            $\nu_i$.state $\leftarrow$ free ;
        end
        if $\nu_i$.input tuple.valid then
            timestamp($\nu_i$.input tuple) ;
            write $\nu_i$.input tuple to overflow queue ;
        end
    end
end
```

Fig. 6: Shift phase. Results are reported on ν_0; candidates and input tuples move to the left and right, respectively; tuples after the last shifter list node are written to the overflow queue in DRAM.

3.3. The Finite State Machine (FSM) inside a Shifter List Node

While in Figures 5 and 6, we phrased parallel BNL as an algorithm in pseudo code, its implementation in hardware boils down to the simple finite state machine (FSM) depicted in Figure 7. In this state machine, each shifter list node can be in any of four states: F (free), W (working set), X (deleted), and O (output). Initially, all shifter list nodes are in state F. The dashed transitions enable *shifting* of shifter list nodes toward the end or the beginning of the shifter list. To implement shifting, two adjacent shifter list nodes *swap* their state and working set contents. Shifter list nodes in state O are shifted to the beginning of the shifter list, whereas shifter list nodes in state X are shifted to the end, where automatically the (dotted) transition $X \rightarrow F$ is executed. Note that we cannot directly perform the transition $W \rightarrow F$ because "free" shifter list nodes need to be at the end of the shifter list to ensure that new candidate tuples have been evaluated against the entire existing working set first. The solid transitions labeled "insert", "output", and "delete" are followed when a corresponding condition (listed below) is satisfied:

(i) Insert: when an input tuple reaches the first shifter list node in state F, it is inserted into the working set and the respective shifter list node changes its state accordingly ($F \rightarrow W$).

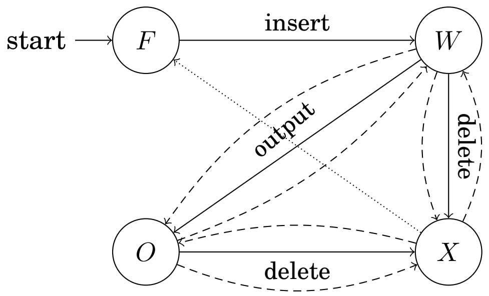

Fig. 7: State machine inside an shifter list node.

(ii) *Output*: when the timestamp condition of a working set tuple has been met (cf. Figure 5), that tuple is a skyline tuple and is ready for output ($W \to O$).

(iii) *Delete*: working set tuples are deleted when they are dominated by an input tuple ($W \to X$). Output tuples (*i.e.*, skyline tuples) are deleted after they have been output, *i.e.*, shifter list nodes in state O first are shifted to the beginning of the shifter list, where the tuple is output and the state of the shifter list node is changed ($O \to X$).

3.4. Correctness of the Proposed Approach

The shift phase of our parallel BNL version leads to a pipeline-style processing mode, where all input tuples visit all shifter list nodes, *i.e.*, all tuples in the working set, one after another. This allows us to exploit parallelism without altering the semantics of the original algorithm, in this case conventional BNL [Börzsönyi et al. 2001].

However, working set tuples and input tuples move in opposite directions, which bears a risk of *race conditions*, in particular *missed comparisons*. This problem can be avoided as follows: Working set tuples that move from ν_{i+1} to ν_i are compared to the current input tuple in addition to being copied to the predecessor shifter list node, *i.e.*, evaluation and copying are synchronized (cf. Section 3.5). If we detect that a working set tuple was dominated during the copy process, we cancel the copy transaction by not updating the state of ν_i from X to W. At the same time we change the state of ν_{i+1} from W to X such that now both shifter list nodes hold a deleted working set tuple.

By avoiding potential race conditions in the shift phase, the semantics of parallel BNL become identical to the original algorithm because it is guaranteed that when an input tuple is processed at an arbitrary shifter list node, all effects caused by tuples earlier in the input stream are “visible” to that shifter list node.

3.5. BRAM-based Component-wise Processing

Up to now, we have assumed atomic processing and forwarding of tuples. However, for performance reasons and because our implementation is based on BRAM, we *stream* all data one dimension at a time through the pipeline of shifter list nodes. Figure 8 illustrates this for the case of three-dimensional tuples and twelve shifter list nodes. Notice that after each tuple, we pass *meta data* such as timestamp information or the *tuple valid* flag.

We use BRAM (dashed boxes in Figure 8) for tuple storage within a shifter list node since potentially large tuples need to be saved in the working set. A BRAM block is big enough to store tuples of any realistic size. As a positive side effect, the number of dimensions has minor impact on resource consumption.

To *swap* two adjacent shifter list nodes, we cannot copy entire chunks of memory from one BRAM block to another in a single clock cycle—we have to do this word by

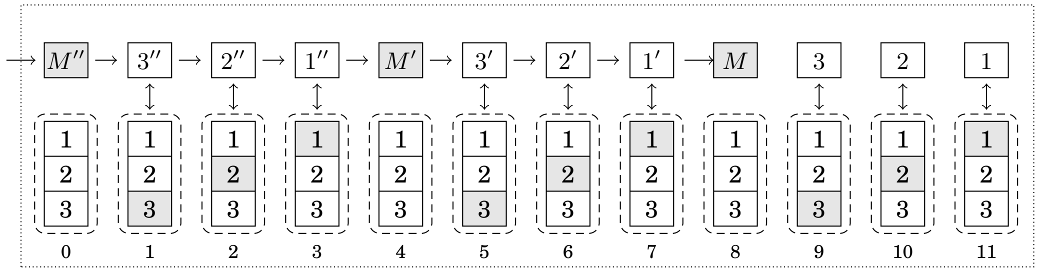

Fig. 8: Three tuples streaming by twelve shifter list nodes.

word. Nevertheless, as illustrated in Figure 9, copying is still possible without reducing throughput.

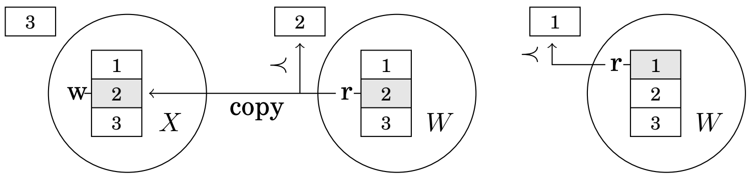

Fig. 9: BRAM copy mechanism: three shifter list nodes with a three-dimensional input tuple streaming by above.

In this example, the first shifter list node, is in state X (deleted), while the subsequent one is in state W (working set), which means they need to be *swapped* so that the deleted processing element can propagate to the end of the shifter list. For the BRAM block of the first shifter list node the write enable signal is asserted (w-flag). As data is read from BRAM of the second shifter list node (r-flag) for the dominance test, this data is written proactively to the BRAM of the "deleted" predecessor shifter list node. At the end of the dominance test, the relevant BRAM contents have been entirely copied, and the state of both shifter list nodes can be updated appropriately. Note that with this approach, it is sufficient to instantiate single-ported BRAM, as opposed to dual-ported BRAM, which provides twice as many available BRAM blocks, enabling a longer pipeline of processing elements.

Race Conditions. In Section 3.4, we stated that tuples are never missed, *i.e.*, that tuples of the working set are evaluated properly against all input tuples. Since we evaluate every working set tuple against the current input tuple and potentially copy the working set tuple to a "deleted" predecessor shifter list node at the same time, we can invalidate a copy transaction in case the copied tuple was dominated by the input tuple. Conversely, the meta data appending every input tuple ensures that the copy transaction can be completed before the next input tuple reaches the predecessor shifter list node, holding the freshly copied working set tuple.

3.6. Resource Consumption & Scalability

In the next section, we show that more shifter list nodes result in better performance. Therefore, it is crucial that we utilize FPGA resources efficiently. In Table I, we display resource consumption on a Virtex 5 FPGA (XC5VLX110T) for different configurations of our circuit using a shifter list of 4, 64 and 192 shifter list nodes, respectively. A single shifter list node consumes one out of 296 available single-ported BRAM blocks and roughly 320 LUTs, *i.e.*, 80 slices (post-map measurement). Physically, only 148 dual-ported BRAM blocks are available. However, each dual-ported 36 Kbit BRAM block can be used as two 18 Kbit single-ported BRAM blocks instead. A configuration with only four shifter list nodes consumes 20% of the available slices and 21% of the available BRAM because the measurements also include resources used for the DRAM controller [Bittner 2009] and the Ethernet-based communication framework [Eguro 2010] that we use to move data in and out of the FPGA board. Notice that we are LUT-bound, and that a configuration with 192 shifter list nodes saturates our FPGA. More importantly, even with 99% slice utilization, we were still able to operate the circuit at 150 MHz, which is only possible because of the scalability of shifter list-based implementations with their simple communication pattern.

	Slices		Flip-Flops		LUTs		BRAM	
available	17,280	100.0%	69,120	100.0%	69,120	100.0%	148	100%
4 PEs	3,385	20%	6,371	9%	8,501	12%	32	21%
64 PEs	9,204	53%	15,495	22%	27,385	40%	69	46%
192 PEs	17,151	99%	34,951	51%	67,398	98%	136	91%

Table I: Resource Consumption on the Virtex-5 (XC5VLX110T)

4. EVALUATION—FPGA VERSUS CPU

We first evaluate our parallel FPGA-based skyline operator against a sequential software implementation of BNL, in Section 4.2, to get a better understanding of the behaviour of both versions of the algorithm. Furthermore, in Section 4.4, we compare our FPGA solution to a state-of-the-art, multi-threaded skyline implementation [Park et al. 2009] on two different multicore platforms, in Section 4.4.

4.1. Experimental Setup

All experiments were run from main memory. We used the Xilinx XUPV5 development platform with a Virtex-5 FPGA (XC5VLX110T) clocked at 150 MHz and 256 MiB on-board DDR2 memory. The single-threaded CPU experiments were carried out on an Intel Xeon 2.26 GHz server processor (Gainestown, L5520, DDR3 memory). The multi-core experiments were conducted on the same 8-core Intel Xeon server, as well as on a 64-core (AMD Bulldozer, 2.2 GHz, DDR3 memory) PowerEdge R815 Server from Dell.

4.2. Effects of Data Distribution

To give a better understanding of the performance characteristics of sequential BNL (single-threaded, with and without SIMD support) versus our parallel FPGA implementation, we evaluate skyline queries on input data following three different data distributions. Synthetic input data was generated with the data generator provided by [Börzsönyi et al. 2001] according to the three different distributions: (1) *random*, (2) *correlated*, and (3) *anti-correlated*. These distributions are commonly used to evaluate skyline operators. The input data consists of 1,024,000 input tuples. A tuple has seven dimensions and a timestamp resulting in a total width of 32 bytes, *i.e.*, the size of the entire input set is 31.25 MiB.[3]

[3]We set the tuple width to 32 bytes so that it matches the DRAM word width because our logic to interact with DRAM is rather simple. However, with a more sophisticated memory unit, our implementation should handle an arbitrary number of dimensions equally well.

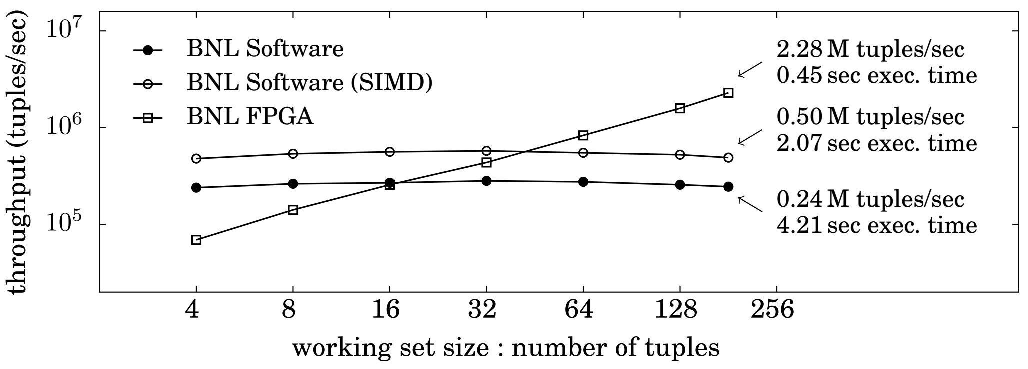

Fig. 10: Randomly distributed dimensions → tuples/sec.

Randomly Distributed Data. For our randomly distributed data set, the skyline consists of 15,154 tuples, *i.e.*, 1.48 % of the input data are skyline tuples. This measure is called the *density* of skyline tuples. On the y-axis we display *throughput* (input tuples/sec) and on the x-axis we vary the size of the working set used in the BNL algorithm.

As can be seen in Figure 10, the size of the BNL working set has little effect on the CPU-based version. On the FPGA, however, throughput increases linearly with the size of the working set because a larger working set also means more processing elements, *i.e.*, a higher degree of parallelism.

Notice that the software solution can be improved by a constant factor using SIMD (single instruction, multiple data), *e.g.*, with special SSE instructions we can perform up to four 32-bit comparisons in parallel. However, there is an overhead of using these instructions. Therefore, the actual improvement is not 4X but rather between 2X and 3X, as was also confirmed by Cho et al. [2010].

Correlated Data. The dimensions of a tuple are correlated if there is a high probability that the values in all dimension are similar. This means that a tuple that is “good” in one dimension is likely to be “good” also in the other dimensions and therefore dominates many tuples. As a result, the skyline is very small, *e.g.*, in this experiment, the skyline consists of only 135 tuples, corresponding to a density of 0.013%.

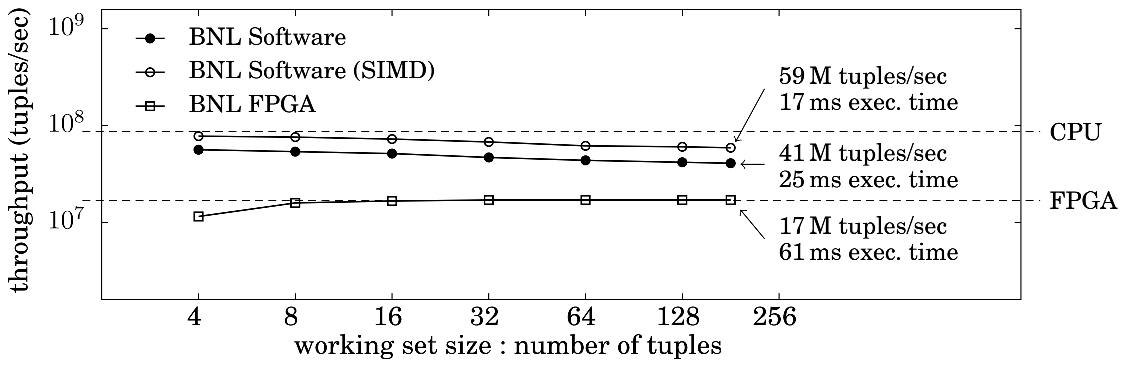

Fig. 11: Correlated dimensions → tuples/sec.

In Figure 11, the CPU-based version of BNL is faster than the FPGA-based one. Low skyline density favors the CPU-based implementation because parallel compute power no longer is the key criteria for a fast execution. Rather, the CPU-based implementation here benefits from the faster memory (DDR3 versus DDR2). In Figure 11, we display the upper bounds for throughput by dashed lines labeled CPU (87 million tuples/sec) and FPGA (17 million tuples/sec), respectively. These bounds were computed using a data set where the first input tuple is the only skyline tuple, which eliminates all other tuples. This results in a minimal number of tuple comparisons of $n-1$, where n is the number of input tuples, which is in line with the known *best case* complexity of $\mathcal{O}(n)$ for BNL [Godfrey et al. 2005].

While we cannot beat the CPU skyline operator with our FPGA implementation when the skyline tuples have a very low density, it is important to note that in absolute numbers both versions are very fast when dealing with correlated data. For instance, the fastest execution (working set size = 4, SIMD support) of the above query on the CPU takes 13 milliseconds and on the FPGA (working set size = 192) 61 milliseconds.

Anti-Correlated Data. This experiment is the opposite of the previous one. *Anti-correlated* means that a tuple, which is "good" in one dimensions, is likely to be "bad" in the other dimensions. In this case, a lot more tuples are part of the skyline, *e.g.*, now the skyline consists of 202,701 tuples, which corresponds to a density of 19.80%.

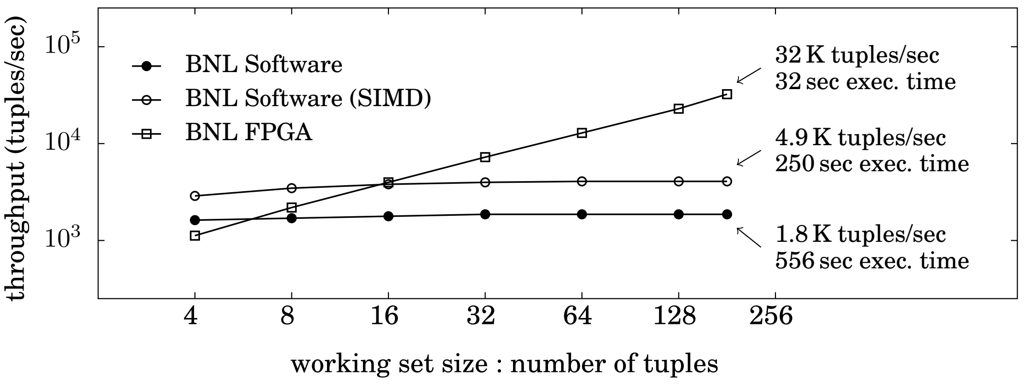

Fig. 12: Anti-correlated dimensions $\rightarrow$ tuples/sec.

The computation of the skyline is now significantly more expensive, *e.g.*, the best execution time of the CPU-based version has gone from 13 milliseconds to almost ten minutes. This slowdown is due to the increased number of comparisons since all skyline tuples have to be pairwise compared with each other. The number of comparisons among skyline tuples alone is $\frac{1}{2}s(s+1)$, where s is the size of the skyline—hence, the *worst case* complexity for BNL is $\mathcal{O}(n^2)$ [Godfrey et al. 2005].

4.3. Discussion on Utilization of Compute Resources

To further analyze the utilization of compute resources of the shifter list for the three workloads used in the previous section, in Figure 13, we plot the ratio between actual number of comparisons and peak number of comparisons for different working set sizes. That is, we show which fraction of the instantiated tuple comparators actually performs useful work as we vary data distributions and shifter list sizes (utilization is averaged over a full run of each experiment).

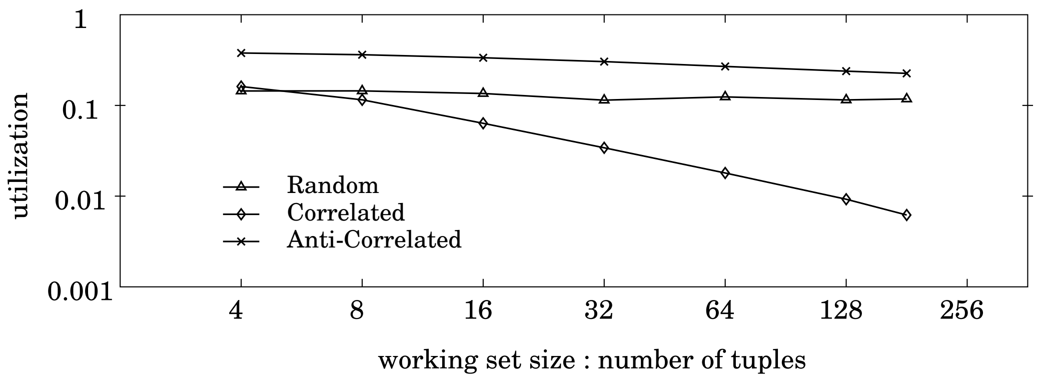

Fig. 13: Utilization of shifter list during query execution.

The figure confirms the throughput characteristics that we observed above. For correlated data, our circuit becomes memory bandwidth bound—additional shifter list nodes do not receive enough useful work and utilization drops as we increase the working set size. Not so with random or anti-correlated data, where the loss is significantly less dramatic.

4.4. FPGA versus Multicore Server

We also compared our FPGA results to *PSkyline* [Park et al. 2009], which is the fastest published skyline algorithm for multicore architectures.[4] We ran PSkyline on the same data sets as in the previous experiments that consisted of 1,024,000 seven-dimensional input tuples. We measured the performance of PSkyline on the 8-core (plus hyperthreading) Intel Xeon server used previously, as well as on a 64-core PowerEdge R815 Server from Dell. The FPGA was configured with 192 shifter list nodes. The results are depicted in Table II.

Data Distribution	FPGA	Intel Xeon	PowerEdge
Random	0.445 sec	0.722 sec	0.433 sec
Correlated	0.061 sec	0.003 sec	0.005 sec
Anti-correlated	31.633 sec	55.104 sec	18.574 sec

Table II: Execution time: FPGA versus multicore.

On the Intel Xeon server and on the PowerEdge server, best results were obtained using 16 and 64 threads, respectively. The performance for the more compute-intensive workloads (random and anti-correlated) achieved by the FPGA is better than the Intel Xeon Server and not far from the performance we measured on the high-end PowerEdge 64-core server.

Moreover, with 192 shifter list nodes a throughput of 32 thousand tuples/sec (anti-correlated distribution) is reached on the FPGA. This is more than two orders of magnitude below the upper bound of 17 million tuples/sec (cf. Figure 11), *i.e.*, with more real estate (using a larger FPGA), there is still a lot of leeway to further increase performance by adding more shifter list nodes.

[4]We would like to thank H. Im for providing the PSkyline code.

5. SHIFTER LISTS

In the previous sections, we described and evaluated an efficient and scalable shifter list-based *skyline* query processor. This section focuses on the shifter list abstraction itself and its most important properties. We will then explore how this design pattern could be applied to several problems, *e.g.*, computing *frequent items*, the *n-closest pairs* problem, and *K-means* clustering.

5.1. A Shifter List is a Data Structure

A shifter list targets the application patterns illustrated in Figure 14. From a given input data set, all *items* (*e.g.*, multi-dimensional tuples as for skyline computation) are consumed in turn. Each input item is evaluated against many or even all of the items in an on-chip *working set*. Possibly, this evaluation results in an update to the working set, such as *inserting* the current input item to the set or *removing*/*updating* others.

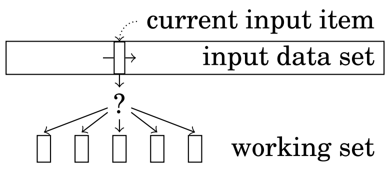

Fig. 14: Typical application pattern for a shifter list: For each input item, the working set is accessed and possibly modified.

The high-level structure of a shifter list is illustrated in Figure 15. Working set items are held in a number of shifter list nodes (processing elements). There is a defined total *order* among all nodes ν_i in a shifter list. Nodes are organized independently but communicate with each other through well-defined *message channels*. As illustrated in Figure 15, these channels constrain communication to *nearest-neighbor messaging*. Aside from application-defined messages, the channels are also used to propagate input data and to exchange working set items between nodes, which ultimately results in a dynamic repartitioning of the working set.

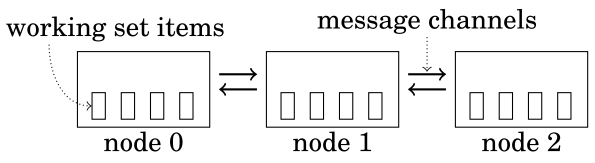

Fig. 15: Shifter lists group working set items into nodes. Neighboring nodes are connected via message channels.

5.2. A Shifter List is for Data Processing

To process the input, we submit each input item to the left-most shifter list node ν_0, where it is *evaluated* against the local working set item(s)—for parallel BNL this was a single tuple but for other applications multiple items per shifter list node are conceivable. Then the input item is *shifted* on to the right neighbor where the process repeats. Effectively, a sequence of input items flows through all nodes in a pipeline fashion.

The *actions* performed at each node depend on the specific task that is to be solved with the shifter list. *Action code* may decide to alter the local working set partition (*e.g.*, by deleting, inserting, or re-arranging working set items); drop the input item from the pipeline; or send and/or receive messages along the message channels.

5.3. A Shifter List is for Parallelism

Input items are evaluated over the individual shifter list node contents and strictly processed in a feed-forward fashion. This has important consequences that we can exploit in order to parallelize the execution over many shifter list nodes while preserving the causality of the corresponding sequential algorithm.

Causality Guarantees. Feed-forward processing implies that the global working set is scanned exactly once in a defined order. What is more, once an input item x_i has reached a shifter list node ν_h, its evaluation cannot be affected by any later input item x_j that is evaluated over a preceding node ν_d. Conversely, the later x_j is guaranteed to see all effects caused by the earlier x_i.

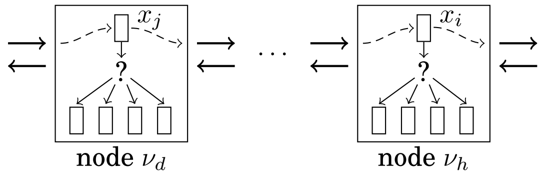

Fig. 16: Shifter list causality guarantees. The earlier x_i will see no effects caused by the later x_j but x_j sees all effects of x_i.

These causality guarantees hold even if we let the executions of x_i on ν_h and x_j on ν_d run *in parallel* on independent compute resources, as illustrated in Figure 16. To uphold the guarantees, x_j only must never *overtake* x_i in the processing pipeline. The preservation of causality hides much of the parallelization difficulties from the application developer, *e.g.*, with a shifter list we can parallelize BNL without any locking mechanism, complicated merging of intermediate results, etc.

Application-Level Guarantees and Invariants. Applications may use the shifter lists' causality guarantees to further establish their own invariants. For skyline queries, for instance, we add new items to the working set only at the end of the shifter list and then gradually shift them to the beginning. Since items never overtake each other, this ensures that the oldest working set item is always at the front of the shifter list.

6. SHIFTER LIST TEMPLATE

In this section, we describe a shifter list template that incorporates a number of generalizations. Their usefulness will become apparent in Section 7, where we discuss the mapping of several algorithms to this template. Note that the purpose of the template is to provide a starting point for a shifter list implementation of a given algorithm but application-specific engineering will still be necessary to produce an efficient solution.

6.1. Template Instantiation

A shifter list is a pipeline of identical shifter list nodes but the first and last shifter list nodes typically implement additional functionality, which is why they are instantiated separately. All intermediate shifter list nodes can be instantiated by means of a Verilog/VHDL *generate* statement. Shifter list nodes communicated using nearest neighbor communication, *i.e.*, the input and output ports of the shifter list nodes need to be connected in a pre-defined way, which can be achieved conveniently using Verilog/VHDL generate statements, as well.

6.2. Evaluating Input Data Items

Input data items are processed in a feed-forward manner and evaluated locally at every shifter list node, as illustrated in Figure 17. Node-local evaluation may trigger an action that affects the state of the input item and/or the state of the shifter list node, *e.g.*, for skyline the actions “drop input tuple” or “drop working set tuple” are executed when the corresponding condition with respect to the dominance test is satisfied.

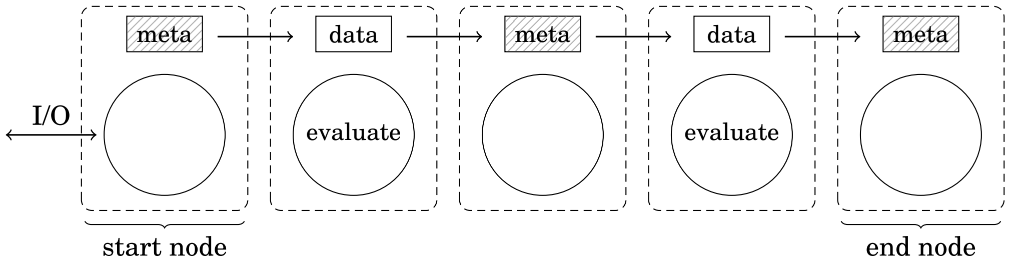

Fig. 17: Evaluation of input items results in the execution of user-defined *action code*.

Updating Input Item State. To forward the state of an input item to the next shifter list node, meta data is appended to every input item after evaluation (*e.g.*, as the flag in the case of skyline when a working set tuple dominated an input tuple). Thus, the stream of input items is interspersed with meta data words (cf. Figure 17).

Updating Shifter List Node State. Evaluation of input items may also affect the state of the evaluating shifter list node itself, *e.g.*, when a working set tuple is dominated in the skyline example, the respective shifter list node is set to “free”, causing it to be shifted towards the end of the shifter list. For the shifting to work correctly, the subsequent shifter list node needs to know about state changes in the predecessor shifter list node. Thus, this information can also be appended to the input item.

The input data bus is accompanied by a *data valid* and a *type* signal that indicates whether the current word is data or meta data. Meta data spanning over multiple words is easily feasible but in many cases a single word is sufficient.

6.3. Swapping Shifter List Nodes

The ability to move shifter list nodes within the shifter list, or more precisely, copy working set data from one shifter list node to an adjacent shifter list node has proven useful for skyline queries. The shifter list template thus supports a notion of *swapping* node contents. However, we cannot allow arbitrary swapping since this could introduce *race conditions*. Thus, the situation where both the left and right neighbor of a shifter list node attempt to swap contents with that core at the same time needs to be avoided. Granting only every other shifter list node to issue *swap* requests during the same clock cycle solves this problem. Hence, in shifter lists only the shifter list nodes that are processing *meta* words are allowed to trigger *swaps*, as depicted in Figure 18.

6.4. Atomic versus Component-wise Processing of Input Data Items

In this section, we have silently assumed atomic processing of data items for ease of presentation. However, multi-dimensional data items can be processed not only *atomically* but also *component-wise*, as we did for the case of skyline, resulting in a narrower input data bus. For skyline queries on our FPGA, component-wise processing using BRAM had some advantages but it also posed some engineering challenges such

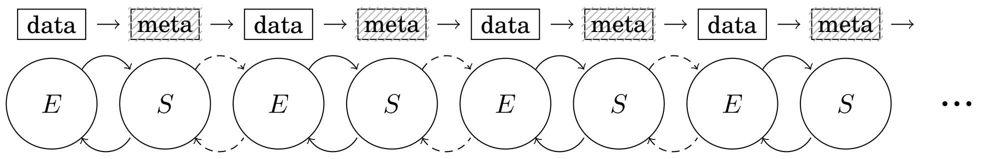

Fig. 18: Shifter list with half the shifter list nodes (E) *evaluating* data items, and the other half (S) potentially issuing *swap requests*.

as implementing component-wise *swaps*. In general though, the shifter list concept supports both atomic and component-wise processing of input items.

7. SHIFTER LIST MAPPINGS OF DIFFERENT ALGORITHMS

This section discusses the mapping and parallelization of several algorithms from different domains using shifter lists.

7.1. Frequent Item Computation with Shifter Lists

Teubner et al. [2010] solved computing *frequent items* (in a streaming context) on an FPGA with a variation of the *Space-Saving* algorithm (cf. Figure 19). In the paper, they evaluated a number of FPGA-based implementations. Their most efficient version relied heavily on pipelining, achieving three times higher throughput as the best known software results. The algorithm that Teubner et al. [2010] propose can be implemented with shifter lists, and we therefore revisit the most important aspects and results here.

```
1  foreach stream item $x \in S$ do
2      find bin $b_x$ with $b_x$.item $= x$ ;
3      if such a bin was found then
4          $b_x$.count $\leftarrow b_x$.count $+ 1$ ;
5      end
6      else
7          $b_{\mathsf{min}} \leftarrow$ bin with minimum count value ;
8          $b_{\mathsf{min}}$.count $\leftarrow$ min.count $+ 1$ ;
9          $b_{\mathsf{min}}$.item $\leftarrow x$ ;
10     end
11 end
```

Fig. 19: Algorithm *Space-Saving* [Metwally et al. 2006].

An exact solution that identifies the n most frequent items, would count the number of occurrences for every item, sort the result by item count, and emit the top n items. To avoid exhaustive space consumption the *Space-Saving* algorithm that approximates an exact solution was developed by Metwally et al. [2006]. The original algorithm is depicted in Figure 19. It uses k bins to count the frequencies of the most frequent items in a stream. If a corresponding bin b_x for a new stream item x exists, that item's frequency is increased (lines 3–4 in Figure 19). Otherwise, the bin with the lowest count value gets evicted in favor of the new item (lines 6–9), which inherits the incremented frequency of its predecessor (for more details see [Metwally et al. 2006]).

High-Level Shifter List Mapping. The key idea to implement the *Space-Saving* algorithm with shifter lists, is to map every bin to a shifter list node. To find corresponding bins we stream all input items through the shifter list (lines 3–4 in Figure 19). If at the end of the shifter list no appropriate bin is found, we update the last shifter list node

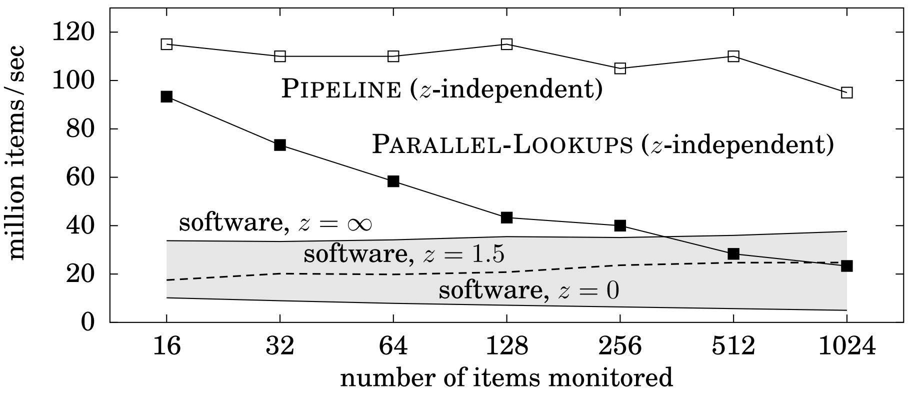

Fig. 20: Space-Saving algorithm. The software version is affected by the Zipf distribution ($z \in \{0, 1.5, \infty\}$), whereas the FPGA implementations *Pipeline* and *Parallel-Lookups* are Zipf-independent.

according to lines 7–9, in Figure 19. However, this is correct only if the last shifter list node stores the bin with the lowest count, which we ensure by keeping the shifter list *sorted* by bin count, using the *swapping* mechanism of shifter lists. As a result, the item with lowest count automatically propagates to the end of the shifter list.

Evaluation Phase. The working set item at every shifter list node consists of a bin ID and the current bin count (`working_set = {`bin_x`,`$count$`}`). Nodes compare input item IDs to `working_set.`bin_x, *i.e.*, line 2 in Figure 19 is performed in a pipeline-parallel manner. If a corresponding bin is found for an item, the count is incremented (line 4 in Figure 19 is executed directly on the respective shifter list node). This action affects the state of both the input item and the shifter list node: a meta data flag is set, indicating that the current input item was counted, and also the updated bin count is forwarded to the next shifter list node, which is relevant for the later shift phase.

Shift Phase. The shifter list nodes that process meta words are allowed to issue swap requests with their predecessor shifter list nodes. If the count of the predecessor shifter list node is smaller than the local bin count, *i.e.*, if the condition (`meta_data.`$count <$ `working_set.`$count$) is satisfied, working set contents will be exchanged, and in the following cycle the predecessor node will *evaluate* the next data item against the working set data just received from its successor.

Last Shifter List Node. This node has slightly modified functionality. In the evaluation phase, if the data item does not match the current bin ID, it is temporarily stored at the shifter list node. In the subsequent shift phase, if meta data indicates that the item has already been counted by some other bin, the temporary copy of the data item is discarded. Otherwise, the bin count is incremented, and the bin ID is overwritten with the ID of the temporary copy, as in the original *Space-Saving* algorithm. Due to the swapping mechanism it is guaranteed that the last shifter list node always stores the bin with the smallest count.

Performance & Scalability. In Figure 20, [Teubner et al. 2010] evaluated the performance of monitoring a varying number of frequent items, using different techniques. *Pipeline* corresponds to the shifter list implementation. It achieves the best performance and scales well with respect to the number of bins. In contrast, *Parallel-Lookups* is an implementation that uses a tree structure to look up the bin with the smallest count. Figure 20 shows that the tree-based approach does not scale well, *i.e.*, performance degrades significantly with an increasing number of bins.

7.2. n-Closest Pairs of Points with Shifter Lists

The n-closest pairs of points problem is defined as follows: given N points in a multi-dimensional space, find the n pairs of points with the smallest distance to each other. Distance can be defined in various ways, *e.g.*, the Euclidean distance (ℓ^2-norm), Manhattan distance (ℓ^1-norm), etc. The brute force algorithm that finds a single closest pair is illustrated in Figure 21. To find n such pairs we can execute this algorithm n times, while making sure that we exclude previously found pairs.

```
$mindist = \infty$ ;
for $i = 1 \rightarrow i < length(P) - 1$ do
    for $j = i \rightarrow j < length(P)$ do
        if $dist(P[i], P[j]) < minDist$ then
            $minDist = dist(P[i], P[j])$ ;
            $closestPair = (P[i], P[j])$ ;
        end
    end
end
$return\ closestPair$ ;
```

Fig. 21: Brute-force algorithm that determines the closest pair of points.

High-Level Shifter List Mapping. To compute n closest pairs, we map a different point from the input data to each of k shifter list nodes, where $k > n$. As input points visit the shifter list nodes, the distance d to the working set points is computed. If the distance to an input point is smaller than the smallest previously observed distance, the new distances is saved together with that input point at the respective shifter list node. The *swapping* mechanism of shifter lists keeps the global working set sorted by distance such that the closest pairs reside at the beginning of the shifter list.

Figure 22 illustrates this idea. The first n shifter list nodes store the n closest pairs observed so far. If d at one of the shifter list nodes ν_i (where $i > n$) becomes smaller, that node is shifted *upstream* to its proper position. Points that do not fit into the working set are *timestamped* and written to an overflow queue, as in our skyline implementation. The timestamp is used to drop pairs that did not make it into the top n shifter list nodes after having been compared to all input items. Furthermore, "free" shifter list nodes can be reused to store new input points. The algorithm terminates when there are no more input points.

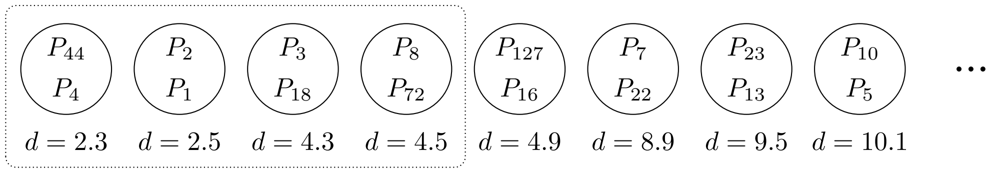

Fig. 22: The shifter list keeps the n closest pairs (here, $n = 4$) at the beginning of the list implicitly sorted. The pairs after the first n eventually expire and can be replaced.

Evaluation Phase. Every shifter list node stores a pair of points (P_A, P_B), the distance between those points d, and a timestamp when P_A was inserted into the shifter list (`working_set = {`$P_A, P_B, d, timestamp$`}`). All nodes that process an input item compute the distance d' between that item and P_A. If $d' < d$, P_B is replaced with the input item, and d is overwritten with d'. Moreover, d' is sent as meta data to the subsequent shifter list node, which is relevant for the later shift phase. Finally, if the timestamp condition was satisfied the working set distance is set to $d = \infty$.

Shift Phase. If the distance of the predecessor shifter list node is larger than the distance of the local closest pair, node contents are swapped to keep the shifter list sorted by distance. Notice that shifter list nodes with $d = \infty$ will automatically be swapped towards the end of the shifter list.

Last Shifter List Node. If at the last shifter list node $d = \infty$, a new point P_A can be assigned to this node, and the next data point will be the second point of the pair P_B. To avoid that P_B overwrites P_A, we set the distance $d = \infty - 1$. As for skyline, this last node also has additional I/O capabilities, *i.e.*, if the distance at the last shifter list node is not set to infinity, meaning that this core is still in use, we need to write the current input item to an overflow queue for processing in a subsequent round.

7.3. K-means Clustering with Shifter Lists

K-means clustering aims to partition N points in a d-dimensional space into k clusters such that each cluster has a *center* (the *mean* position of all points in the cluster) and each point in the cluster is closest to that center. Finding the optimal solution to this problem is NP-hard [Drineas et al. 2004], which is why typically approximation algorithms are used. A common iterative approximation algorithm is the following. Start with k random samples as initial centers. In every iteration, first assign each point to the closest center, then recompute all centers. Repeat this process until either the result converges or a specified threshold of iterations is reached. Note that a variant of K-means updates the cluster centers each time a point is reassigned to a new cluster, which leads to faster convergence. The center can be updated incrementally as follows:

$$C_{n+1} = \frac{\sum_{i=1}^{n+1} t_i}{n+1} = C_n + \frac{t_{n+1} - C_n}{n+1} \quad ,$$

where C_n corresponds to the current center, t_{n+1} corresponds to the new data point, and n is the running count of items in the cluster.

High-Level Shifter List Mapping. To implement the above algorithm we only instantiate k shifter list nodes. Each node stores one of the k centers and is initialized with some random data point. As input data points propagate through the shifter list, the distance to every center is computed, and the closest center is identified. After a data item has traversed the entire list the closest center is updated. To this end, the closest center is shifted together with the data item towards the end of the shifter list.

Evaluation Phase. The working set at every shifter list node stores a different center of the k clusters, *i.e.*, a d-dimensional data point together with the current count of points assigned to this cluster (`working_set = {`$center, count$`}`). The count is necessary to perform incremental updates of the center, as discussed above. All shifter list nodes that process an input item compute the distance d between the data point and the local center, which is temporarily stored at the shifter list node. Furthermore, d is sent as meta data to the subsequent shifter list node as it will be relevant for the shift phase.

Shift Phase. In the shift phase, the closest center is moved towards the end of the shifter list together with the respective data item. If the temporarily stored distance is smaller than the smallest distance computed so far, there is no need to swap. Otherwise, the predecessor node stores the closest center, and we therefore need to swap.

Last Shifter List Node. At the last shifter list node we simply recompute the center by taking the current data item into account using the formula discussed above.

8. RELATED WORK

The introduction of skyline queries in 2001 [Börzsönyi et al. 2001] has created a new direction for research (a comprehensive overview is given in [Godfrey et al. 2005]). There have been a few attempts to exploit parallelism for skyline query processing, *e.g.*, using SIMD instructions [Cho et al. 2010] or multiple threads [Park et al. 2009] on multicore machines. However, the compute-intensive nature of skyline queries suggests that even higher degrees of parallelism are required to effectively tackle this type of problem, making FPGAs an interesting alternative platform to explore.

There have been several approaches to execute SQL on FPGAs (*e.g.*, [Dennl et al. 2012; Sukhwani et al. 2012]). Furthermore, FPGA solutions in the context of databases have been proposed for sorting [Koch and Torresen 2011], XML filtering [Moussalli et al. 2011], or high-speed event processing [Inoue et al. 2011]. Nevertheless, those examples all confirm the observation of [Chung et al. 2011]: FPGAs still lack essential abstractions that have become pervasive in general-purpose computers; rather, most systems are developed in an ad-hoc manner for just one particular problem setting.

With shifter lists, we provide an abstraction that aids in building parallel solutions for difficult data processing tasks that demand high performance. Shifter lists combine well-studied FPGA concepts such as *stream processing*, *nearest neighbor communication*, and *pipeline-parallelism* (see, *e.g.*, [Hormati et al. 2008; Kahn 1974]) into a special kind of data structure for highly parallel hardware, comprising data storage and concurrent data processing. Shifter lists keep working set data co-located with the processing logic that uses it. In a sense, this blurs the classical separation of data and logic. Softening this strict separation indeed makes sense in the light of FPGAs and ongoing hardware trends. There is a general consensus that power and heat dissipation problems will force a move toward *heterogeneous* system architectures [Borkar and Chien 2011; Esmaeilzadeh et al. 2011; Singh 2011]. In such designs (an example are *Nanostores* [Ranganathan 2011]), data structures can be wrapped right into the corresponding processing logic to further improve energy efficiency and speed.

9. CONCLUSION

The prevalence of parallel hardware forces application developers to come up with efficient solutions that are able to exploit the available parallelism and scale to many parallel elements. However, in doing so, developers face several challenges such as the programmability of complex parallel systems, as well as dealing with the cost of communication among parallel units.

It is generally recognized that the ease of programming FPGAs is an important issue that will determine the success and impact of FPGAs in future heterogeneous systems. Yet, FPGAs still lack essential abstractions and design patterns, resulting in a high engineering overhead for every new problem.

Furthermore, developing FPGA solutions for data processing tasks that scale to high degrees of parallelism is often difficult. With increasing core counts, the average on-chip *distance* grows between arbitrary communication partners. What is more, for all-to-all communication patterns, the necessary routing logic scales quadratically in the number of compute nodes, which limits the observed *bandwidth*. Algorithms based on *scatter-gather* mechanisms are affected by the cost of communication in a similar way.

With shifter lists, we address the design of parallel data processing algorithms for FPGAs in two important ways: *(i)* Shifter lists can be used as a generic implementation strategy to build parallel data processing operators, *i.e.*, no need to re-start platform optimization for each new problem instance. *(ii)* Shifter lists have the awareness of communication cost built-in. It is applied by bringing *pipelining* and *nearest neighbor communication* to the *inside* of individual data processing operators.

REFERENCES

Ray Bittner. 2009. The Speedy DDR2 Controller For FPGAs. In *Proc. Int. Conf. on Engineering of Reconfigurable Systems and Algorithms (ERSA)*. Las Vegas, NV, USA.

Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (May 2011).

Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. 2001. The Skyline Operator. In *Proc. 17th Int. Conf. on Data Engineering (ICDE)*. Heidelberg, Germany.

Sung-Ryoung Cho, Jongwuk Lee, Seung-Won Hwang, Hwansoo Han, and Sang-Won Lee. 2010. VSkyline: Vectorization for Efficient Skyline Computation. *SIGMOD Rec.* 39, 2 (Dec. 2010).

Eric S. Chung, James C. Hoe, and Ken Mai. 2011. CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing. In *Proc. 19th ACM SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. Monterey, CA, USA.

Convey Computer. 2014. Convey HC-2. (2014). http://www.conveycomputer.com.

Christopher Dennl, Daniel Ziener, and Jürgen Teich. 2012. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. In *Proc. 20th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. Toronto, ON, Canada.

Petros Drineas, Alan M. Frieze, Ravi Kannan, Santosh S. Vempala, and V. Vinay. 2004. Clustering Large Graphs via the Singular Value Decomposition. *Machine Learning* 56, 1-3 (June 2004).

Ken Eguro. 2010. SIRC: An Extensible Reconfigurable Computing Communication API. In *Proc. 18th IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. Charlotte, NC, USA.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. 2011. Dark Silicon and the End of Multicore Scaling. In *Proc. 38th Symp. on Computer Architecture (ISCA)*. San Jose, California, USA.

Parke Godfrey, Ryan Shipley, and Jarek Gryz. 2005. Maximal Vector Computation in Large Data Sets. In *Proc. 31st Int. Conf. on Very Large Data Bases (VLDB)*. Trondheim, Norway.

Amir Hormati, Manjunath Kudlur, Scott Mahlke, David Bacon, and Rodric Rabbah. 2008. Optimus: Efficient Realization of Streaming Applications on FPGAs. In *Proc. Int. Conf. on Compilers, Architecture, and Synthesis for Embedded Systems (CASES)*. Atlanta, GA, USA.

IBM. 2014. IBM Netezza Data Warehouse Appliances. (2014). http://www.ibm.com/software/data/netezza.

Hiroaki Inoue, Takashi Takenaka, and Masato Motomura. 2011. 20Gbps C-Based Complex Event Processing. In *Proc. 21st Int. Conf. on Field Programmable Logic and Applications (FPL)*. Chania, Crete, Greece.

Gilles Kahn. 1974. The Semantics of Simple Language for Parallel Programming. In *IFIP Congress*. Stockholm, Sweden.

Dirk Koch and Jim Torresen. 2011. FPGASort: A High Performance Sorting Architecture Exploiting Runtime Reconfiguration on FPGAs for Large Problem Sorting. In *Proc. 19th ACM SIGDA Int. Symp. on Field Programmable Gate Arrays (FPGA)*. Monterey, CA, USA.

Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. 2006. An Integrated Efficient Solution for Computing Frequent and Top-k Elements in Data Streams. *ACM Transactions on Database Systems (TODS)* 31, 3 (Sept. 2006).

Roger Moussalli, Mariam Salloum, Walid A. Najjar, and Vassilis J. Tsotras. 2011. Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In *Proc. 27th Int. Conf. on Data Engineering (ICDE)*. Hannover, Germany.

Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. 2009. Parallel Skyline Computation on Multicore Architectures. In *Proc. 25th Int. Conf. on Data Engineering (ICDE)*. Shanghai, China.

Parthasarathy Ranganathan. 2011. From Microprocessors to Nanostores: Rethinking Data-Centric Systems. *IEEE Computer* 44, 1 (Jan. 2011).

Satnam Singh. 2011. Computing without Processors. *Commun. ACM* 54, 8 (Aug. 2011).

Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. 2012. Database Analytics Acceleration using FPGAs. In *Proc. 21st Int. Conf. on Parallel Architectures and Compilation Techniques (PACT)*. Minneapolis, MN, USA.

Jens Teubner, René Müller, and Gustavo Alonso. 2010. FPGA Acceleration for the Frequent Item Problem. In *Proc. 26th Int. Conf. on Data Engineering (ICDE)*. Long Beach, CA, USA.

Riccardo Torlone and Paolo Ciaccia. 2002. Which Are My Preferred Items. In *Workshop on Recommendation and Personalization in eCommerce (RPEC)*. Malaga, Spain.