Shifter Lists—Interleaving Data Storage and Computation

ABSTRACT

Database system vendors have started to explore massively parallel co-processors such as FPGAs to further increase performance of their systems. However, the lack of suitable abstractions often makes it difficult to map a given database task to these devices and fully exploit their inherent parallelism.

In this paper, we focus on one such abstraction that can be applied to a common class of compute-intensive database operators that are based on nested loops. We propose a new data structure—a shifter list—for FPGAs that interleaves *data storage* and *computation*, allowing multiple operations to be executed on the same data structure in parallel while implicitly avoiding race conditions and other potential conflicts.

The shifter list abstraction allows many related database tasks, *e.g.*, top-k queries, k-nearest neighbor search (k-NN), and skyline queries to be brought to FPGAs for improved performance. In this paper, we apply shifter lists to *skyline queries*, as one possible use case. Our experiments show that we achieve very promising results compared to CPU-based solutions.

1. INTRODUCTION

There has been an increase in both the number of research projects and commercial systems that exploit heterogeneous, low power, and massively parallel co-processors to accelerate data processing operations. For instance, server vendors such as Netezza [20] (acquired by IBM in 2010) and Convey [7] equip their systems with configurable hardware accelerators, *i.e.*, *field-programmable gate arrays* (FPGAs).

Modern FPGAs provide high aggregated compute power but it is often difficult to turn the hardware's parallelism into true performance for a given database task. A common answer to this challenge is to build specialized implementations focused on one specific task, *e.g.*, by fitting the problem to an input data partitioning scheme and then fixing problem-specific bottlenecks as they arise. This tends to leave much of the true hardware potential unused. In addition, the lack of suitable abstractions prevents the invested efforts from carrying over from one application to another.

Contributions. In this work, we present a new data structure abstraction (shifter lists) that helps designing *massively parallel* algorithms for a number of related database tasks. Shifter lists combine data organization, computational power, and synchronization into a novel 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 parallel processing elements. The processing elements are arranged as a pipeline and locally update the shifter list as input data *flows by*. The only communication required is between *neighboring* processing elements. As a result, shifter lists exhibit high *throughput* and very good *scalability*.

We illustrate shifter lists based on a common database operator, *skyline* [4]. Skyline computation is a good example where straightforward input data partitioning does not fit the characteristics of FPGAs very well. With shifter lists, by contrast, we dynamically partition the active *working set* in a way that allows throughput optimized processing of the input data and concurrent manipulation of the working set. In our experiments, we show how throughput scales linearly with respect to the amount of FPGA resources allocated. Using a low-end FPGA, we clearly outperform a single-threaded CPU-based skyline operator and even achieve performance close to a parallel skyline operator running on a high-performance 64-core server.

The rest of this paper is organized as follows. Section 2 motivates our work and relates it to existing ideas. In Section 3, we introduce shifter lists. Section 4 illustrates using shifter lists for skyline computation. A concrete implementation on the Xilinx (XUPV5) development platform is discussed in Section 5, and evaluated in Section 6. Additionally, in Section 7, we briefly sketch a shifter list implementation for *(i)* frequent item computation and *(ii)* top-k queries to demonstrate the generality of shifter lists, before we wrap up in Section 8.

2. RELATED WORK AND MOTIVATION

The prevalence of parallel hardware is pushing the software side harder and harder to come up with efficient parallel problem solutions. Berkeley researchers phrased this provocatively in a recent article [1] in *Communications of the ACM*: "If researchers meet the parallel challenge, the future of IT is rosy. If they don't, it's not." The programma-

FPGA'13 February 11 - 13, Monterey, California USA

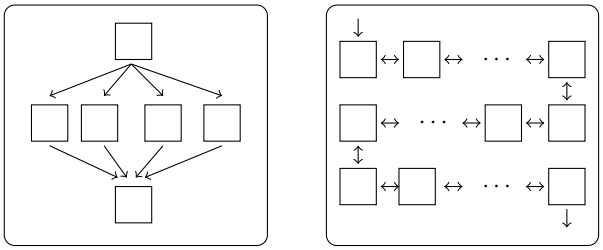

Figure 1: Scatter-gather versus pipelining along a series of processing elements (PEs).

bility of complex parallel systems as well as dealing with the cost of communication among parallel units are two of the challenges that we face.

2.1 Parallel Hardware & Heterogeneity

The trend toward increasing degrees of parallelism is complemented with a notion of hardware *heterogeneity* [23]. Specialized co-processors and even programmable logic are already used to assist general-purpose cores on compute- or data-intensive tasks. A particular instance of this trend are FPGAs, with IBM/Netezza's 1000 (formerly *Twinfin*) [20] being one of the most prominent examples of a commercial FPGA-powered database appliance.

On the research side, there have been some approaches to execute standard SQL queries on FPGAs by assembling query plans from a component library either statically [19] or at runtime using partial reconfiguration [8]. Furthermore, FPGA solutions in the context of databases have been proposed, *e.g.*, for sorting [16], XML filtering [18], or high-speed event processing [15, 22].

Nevertheless, those examples all confirm the observation of Chung et al. [6]: 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. In the field of high-level synthesis, projects like Kiwi [13] or Liquid Metal [14] recognize that the ease of programming FPGAs is an important issue that will determine the success and impact of FPGAs in future heterogeneous systems. With shifter lists, we work towards the same goal as these projects, by providing an abstraction that aids in building parallel solutions for difficult database tasks demanding high performance.

2.2 Communication Patterns

With increasing core counts, the average on-chip *distance* grows between arbitrary communication partners, which requires *additional energy* and increases *latency* [3]. What is more, the necessary *routing logic* scales quadratically in the number of compute nodes, which limits the observed bandwidth for many communication patterns.

As illustrated in Figure 1, algorithms based on *scatter-gather* mechanisms are strongly affected by the cost of communication. However, 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 processing elements (Figure 1), known as *nearest neighbor communication*.

One of the first works that addressed core-to-core communication in databases is *QPipe* by Gao et al. [11]. *QPipe* places database operators on individual cores of a multicore processor and explicitly pipelines data between cores. However, the proposed scheme was designed for *inter-operator* pipelining only and is hence limited to rather coarse-grained parallelism.

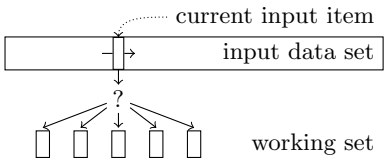

Figure 2: Typical application pattern for a shifter list: For each input item, the working set is accessed and possibly modified.

Intra-operator pipelining has been discussed in a few recent papers. In *cyclo-join* [10], the join operator is mapped to a ring topology, where the inner relation is stationary and fragments of the outer relation are rotated in the ring. *Handshake join* [24] is a stream join algorithm devised for many-core systems that distributes the join operator over available parallel compute resources. Each parallel unit performs a local join while the two join relations *flow* through a series of such units in opposite directions. From these works, shifter lists adopt the (pipelined) dataflow processing paradigm. Yet, shifter lists are better viewed as a special kind of data structure for highly parallel hardware, comprising data storage and concurrent data processing.

With shifter lists, we address the design of parallel database algorithms for FPGAs in two important ways: *(i)* shifter lists can be used as a generic implementation strategy to build parallel database operators—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 to the *inside* of individual operators.

3. SHIFTER LISTS

Our shifter list data structure targets application patterns as illustrated in Figure 2. From a given input data set, all items are consumed in turn. Each input item is evaluated against many or even all of the items in an in-memory (on-chip) *working set*. Possibly, this evaluation results in an update to the working set, such as adding the current input item to the set or removing others.

Note that the processing order of the input data may be important for some database algorithms, *i.e.*, a parallel computation must preserve the *causality* inferred by a sequential implementation. The importance thereof will become clearer later when we use *skyline computation* to showcase shifter lists—in the block nested loops (BNL) algorithm [4], for every input item, a working set of items has to be examined and updated (consistent with the pattern in Figure 2).

3.1 A Shifter List is a Data Structure

The high-level structure of a shifter list is illustrated in Figure 3. Working set items are held in a number of processing elements, which we will call *shifter list nodes*. There is a defined total *order* among all nodes ν_i in a shifter list. Oftentimes, node content itself will have a defined order,

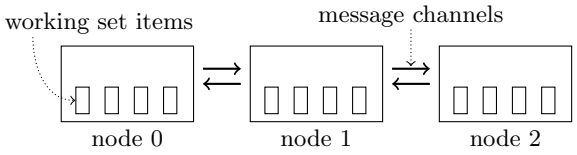

Figure 3: Shifter lists group working set items into nodes. Neighboring nodes are connected via message channels.

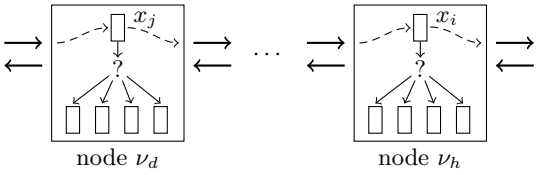

Figure 4: Shifter list causality guarantees. The earlier x_i will see no effects caused by the later x_j but x_j sees all effects of x_i.

resulting in a total ordering across all working set items.

Nodes are organized independently but communicate with each other through well-defined *message channels*. As illustrated in Figure 3, these channels constrain communication to *nearest-neighbor messaging*. Besides for application-defined messages, the channels will also be used to propagate input data and to exchange (*swap*) working set items between nodes (which ultimately results in a dynamic repartitioning of the working set).

3.2 A Shifter List is for Data Processing

To process input, we submit each input item to the leftmost shifter list node ν_0. There, the item is *evaluated* against all local working set items, then *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.

Self-Similarity. While the concrete action code has to be written specifically for each shifter list use case, we often see a "self-similarity" effect. Thereby, the local action code resembles very closely the superordinate algorithm that solves the overall application task. Typically, only *side effects* have to be modified to obtain the code for node-local execution. For instance, node-local "overflow tuples" in the BNL skyline algorithm have to be forwarded to the next shifter list node, rather than be written to an overflow file as in the subordinate algorithm (see Section 5.1).

3.3 A Shifter List is for Parallelism

As described above, input items are evaluated over the individual shifter list node contents in a strictly forward-oriented fashion. This has important consequences that we can exploit in order to parallelize the execution over many processing elements (shifter list nodes) while preserving the causality of the corresponding sequential algorithm.

Causality Guarantees. Forward-only processing implies that the global working set is scanned exactly once in a defined order (which turns out to be a desirable property for many algorithms). 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).

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 4. To uphold the guarantees, x_j only must never *overtake* x_i in the processing pipeline, a requirement that is easy to meet if all message channels are implemented as FIFO queues.

The preservation of causality hides much of the parallelization difficulties from the application developer. But the two-way interaction between two neighboring shifter list nodes may still bear a risk for race conditions. More specifically, an item x_i might be affected by the evaluation of x_j if x_j follows too closely in the processing pipeline and the interaction code is not engineered carefully (also see Section 5.3).

Application-Level Guarantees and Invariants. Applications may use the shifter lists' causality guarantees to further establish their own invariants. When solving the skyline problem in Section 4, for instance, we add new items to the working set only at the end of the shifter list (the rightmost position) and then gradually shift them to the left. Since items never overtake each other, this ensures that the oldest working set item can always be found at the front of the shifter list.

3.4 A Shifter List is Data *and* Logic

The intended use of shifter lists is to keep chunks of data—the contents of a node—strictly co-located with the processing logic that uses it. In a sense, this blurs the classical separation of data and logic.

Releasing 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, which might even soon be dominated by highly specialized co-processors or configurable hardware [3, 9, 23]. In such designs, data structures can be wrapped right into the corresponding processing logic to further improve energy efficiency and speed.

4. USE CASE: SKYLINE QUERIES

After skyline queries were first introduced in 2001 [4], a decade of research has produced a variety of different approaches to solving skyline queries (a comprehensive overview is given in [12]). Only recently, there have been a few attempts to exploit parallelism for skyline query processing, *e.g.*, using SIMD instructions [5] or multiple threads [21] on multicore machines. Nevertheless, the compute-intensive nature of skyline queries suggests that even higher degrees of parallelism are required to effectively tackle this type of problems. This makes skyline queries a good candidate for FPGAs and shifter lists.

4.1 The Lemming Skyline

To figuratively explain *skyline queries*, a popular algo-

rithm to solve skyline queries (the BNL algorithm [4]), and the modified version of BNL using shifter lists, we are going to digress into the world of Lemmings. Lemmings[1] are primitive creatures that go on migrations in masses. On Lemmings Planet every year a challenge—*Lemmings got Talent*—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* (see Definition 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* (see Definition 2).

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

4.2 The Competition—1st Year (*Best*)

When the competition took place for the first time, the committee had a definition for the set of best Lemmings (see previous section) 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 5.

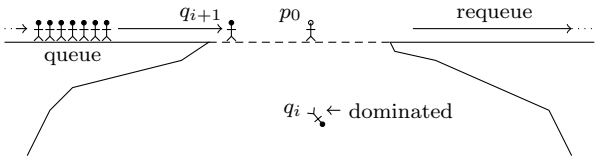

Figure 5: Lemming Skyline with *Best* [26].

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 $q_1 \dots q_{n-1}$. 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 Lemmings 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 [26].

[1]As in the video game "Lemmings" originally developed by DMA Design: http://www.dmadesign.org/

4.3 The Competition—2nd Year (BNL)

The following year many new Lemmings were born and it was time to redetermine the Lemming skyline. 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) [4] and illustrated in Figure 6.

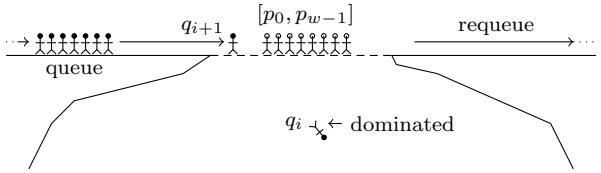

Figure 6: Lemming Skyline with *BNL* [4].

On the bridge there is room for a window 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.

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 Lemmings p_i can now be said to be a true *skyline* Lemming (and leave the bridge) when it 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 the 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. More formally, the BNL algorithm is given in Figure 7.

4.4 The Competition—3rd Year (Shifter List)

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 using the *shifter list* approach. 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

```
1  foreach Lemming q_i ∈ queue do
2      isDominated = false;
3      foreach Lemming p_j ∈ bridge do
4          if q_i.timestamp > p_j.timestamp then
               /* p_j ∈ Lemming skyline                    */
5              bridge.movetoskyline(p_j);
6          else if q_i ≺ p_j then
7              bridge.drop(p_j);
8          else if p_j ≺ q_i then
9              isDominated = true;
10             break;
11     if not isDominated then
12         timestamp(q_i);
13         if bridge.isFull() then
14             queue.insert(q_i);
15         else
16             bridge.insert(q_i);
```

Figure 7: BNL Algorithm ($\prec$ means dominates).

Lemming in parallel. This version of the algorithm is illustrated in Figure 8.

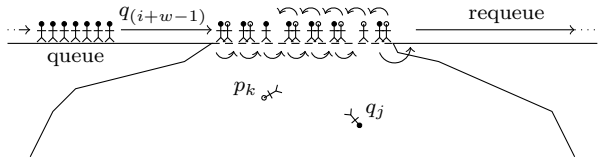

Figure 8: Lemming Skyline: BNL & Shifter List.

To avoid chaos on the bridge the procedure is as follows: In each iteration there is a *shift phase* followed by a *battle 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 *battle phase* all w pairs of Lemmings battle concurrently. As can be seen in the figure, 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 (as in standard BNL). If during the *battle phase* a candidate Lemming p_i falls from the bridge, the other Lemmings $p_{i+1} \dots p_{w-1}$ to the right of that Lemming should move up in the subsequent *shift phase* and fill the gap (indicated by the upper bent arrows in the figure). This is to make room for new candidate Lemmings that reach the right end of the bridge.

As in standard BNL, we can also 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 Lemmings that may become the newest skyline member. Thus, potential skyline Lemmings begin on the right end of the bridge and then gradually move towards the left end again, where they need to wait until they encounter a challenger with a larger timestamp.

5. PARALLEL BNL WITH FPGAS

When Börzsönyi et al. [4] first proposed the block-nested-loops (BNL) algorithm, their main motivation was to support skyline computation for problem sizes that would require external (and slow) memory. With today's large main memories and efficient memory subsystems, the bandwidth to read input or overflow data only defines the performance of skyline computations with extremely small working set sizes (very few candidate skyline tuples). In most practical cases, CPU load becomes the bottleneck when computing skylines.

This makes FPGAs a good alternative to conventional CPUs, because a window of reasonable size (say, 100 skyline candidates) conveniently fits into on-chip memories. In this section, we show how shifter lists help to parallelize the computation within the FPGA to make best use of its available compute capacity. As our results in Section 6 demonstrate, this restores the original characteristics of the algorithm, where reducing the number of overflow tuples translates into faster execution of the algorithm, *i.e.*, the larger the window, the better the performance.

5.1 BNL Using Shifter Lists

Given the opportunity for very fine-grained compute and communication parallelism inside FPGAs, we configure our shifter list implementation such that each node holds only a single working set item. This allows us to fully leverage the available hardware parallelism and also helps us illustrate how shifter lists scale to very high degrees of parallelism.

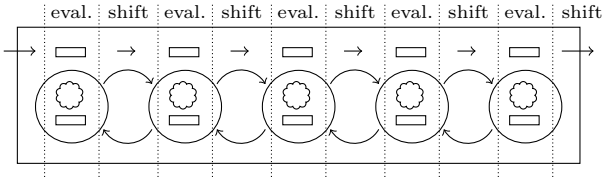

Figure 9: Two-phase processing of shifter lists.

As mentioned in the previous section, data processing in shifter lists can be viewed as a two-phase algorithm: during the *evaluation phase* (the "battle" phase in the Lemming example), a new state is determined for each shifter list node; but these changes are not applied before the *shift phase*, which is the phase that also allows neighbor-to-neighbor communication. In our FPGA-based implementation, those two phases will run synchronously across the chip, as depicted in Figure 9. A similar concept has been proposed in the work of Wang et al. [27] on large-scale parallelization of behavioral simulations. By running parallel code in two phases ("query phase" and "update phase" in [27]), algorithms can be phrased in a way that is intuitive, and yet efficient to parallelize.

Evaluation Phase. Thanks to the *self-similarity* property of shifter lists, the partial algorithm executed locally on each shifter list node in the evaluation phase very closely resembles the global algorithm—BNL in our case. As shown in

```
1  on each node do
2      q ← current input item ;
3      p ← local working set content ;
4      s ← state of shifter list node ;
5      if q.valid then                          /* next challenger */
6          if s = working set then              /* valid candidate */
7              if q.timestamp > p.timestamp then
8                  s ← output ;                 /* found skyline tuple */
9              else if q.data ≺ p.data then
10                 s ← deleted ;                /* drop window tuple */
11             else if p.data ≺ q.data then
12                 q.valid ← false ;            /* drop input tuple */
13         else if s = free then /* add input to window */
14             timestamp(q) ;
15             p.data ← q.data ;
16             s ← working set ;
17             q.valid ← false ;
```

Figure 10: Shifter list-based BNL: evaluation phase.

Figure 10, the only changes to the original algorithm (Figure 7) are that all *side effects* are now handled by the shift phase (and we handle boundary cases more explicitly here).

Shift Phase. All interactions between neighboring nodes are performed in the following shift phase (Figure 11), which updates the global algorithm state based on the outcome of the evaluation phase. In essence, all input items are forwarded one shifter list node toward the right, whereas candidate results (working set items) move toward the left if there is space available, *i.e.*, the left neighbor node is in state 'deleted'. Since skyline candidates move toward the left, we report them on the leftmost node ν_0 once their timestamp condition has been satisfied. Likewise, on the rightmost node ν_{w-1}, we write input items to the overflow file if they were not invalidated during their move along the shifter list pipeline, and cannot be inserted into the shifter list because it is already full.

5.2 Shifter List Node State Automaton

While in Figures 10 and 11, we phrased BNL with shifter lists as an algorithm in pseudo code, the implementation in hardware logic boils down to a *state automaton*. It turns out that most of the transitions in this automaton are not specific to a particular application problem, but are determined by the general data processing pattern that we saw in Figure 2. The few transitions that really depend on the concrete BNL algorithm are labeled 'insert', 'output', and 'delete' in Figure 12. In this state automaton, each shifter list node can be in any of four states:

F: ***"free"*** The current and all following nodes are free. In BNL, an input item that reaches this point will be 'inserted' into the shifter list causing the state transition $F \to W$ for the current node.

W: ***"working set"*** The current node holds an item of the working set (in BNL, one candidate tuple). If the candidate tuple is dominated by an input tuple, the working set item is 'deleted' ($W \to X$). If the timestamp condition in Figure 10 is satisfied the tuple can be 'output' as a skyline tuple ($W \to O$).

X: ***"deleted"*** The working set item of the current node has been 'deleted'. Due to the causality property mentioned earlier, we cannot directly perform the transition $W \to F$ but first need to propagate the freed resource to the end of the pipeline, where transition $X \to F$ can take place.

O: ***"output"*** The working set item of the current node is ready for 'output'. In BNL, this means that a skyline tuple has been identified and is shifted to the leftmost shifter list node, where it is 'output'. Then the current resource is freed ($O \to X$).

The dashed transitions are built-in shifter list transitions. They enable shifting of nodes (more precisely, their working set content) toward the end or the beginning of the shifter list. Nodes in state O are shifted to the beginning. On the other hand, nodes in state X are shifted to the end, where automatically the transition $X \to F$ is executed.

While not required for skyline queries, other algorithms

```
1  foreach node ν_i do
       /* all skyline results are emitted on ν_0                    */
2      if i = 0 ∧ ν_i.state = output then
3          emit ν_i.working set.tuple as result ;
4          ν_i.state ← deleted ;
5      if i < w − 1 then              /* any but the last node */
6          if ν_i.state = deleted then
               /* move up candidates to left                        */
7              ν_i.working set ← ν_{i+1}.working set ;
8              ν_i.state ← ν_{i+1}.state ;
9              ν_{i+1}.state = deleted ;
           /* challengers move one position to right                */
10         ν_{i+1}.input item ← ν_i.input item ;
11     else                           /* the last node (physically) */
12         if ν_i.state = deleted then
13             ν_i.state ← free ;
14         if ν_i.input item.valid then
15             timestamp(ν_i.input item) ;
16             write ν_i.input item to overflow file ;
```

Figure 11: Shifter list-based BNL: shift phase. Results are reported on ν_0; candidates and input items move to the left and right, respectively; items after last node are written to the overflow file.

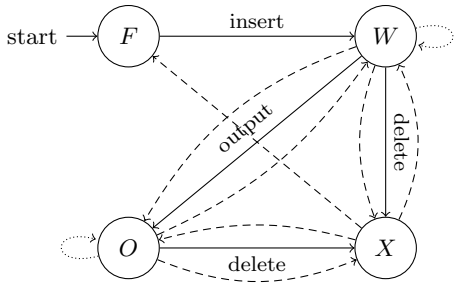

Figure 12: State diagram: shifter list node.

might need the global working set content of a shifter list to remain sorted. In such a scenario, two nodes in state W or O might need to *swap* their working set content, *e.g.*, based on some sort criteria. Therefore, we also added the dotted transitions in the state diagram (omitted between W and O for readability purpose).

5.3 Dimension-at-a-time vs. Tuple-at-a-time

Taken literally, the shift phase illustrated in Figure 11 passes items atomically between adjacent shifter list nodes. For the multi-dimensional input of our benchmark (see Section 6), this would lead to a very high bandwidth demand (*e.g.*, 15 dimensions × 32 bits × 150 MHz clock frequency = 9 GiB/s). Since we are using BRAM (see below), this is far out of reach for the hardware that we use. Instead, our implementation *streams* all data one dimension at a time through the shifter list.

Data Representation. Figure 13 illustrates this for the case of three-dimensional data and a shifter list configuration of eight nodes. After each data point, we pass *meta data* (such as timestamp information or the *data valid* flag).

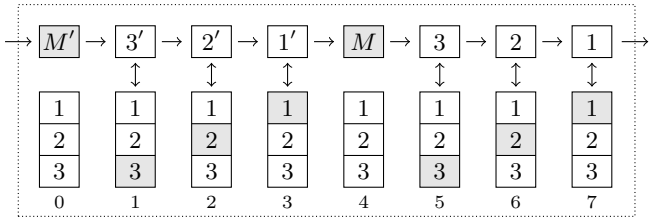

Figure 13: Dimension-at-a-time processing. Two tuples streaming by eight shifter list nodes.

Managing State. So far we have assumed that the working set content can be stored in memory local to a shifter list node without saying anything about the type of this memory. Shifter lists can be implemented using *registers* or *block RAM* (BRAM) depending on the application's needs. For our BNL implementation, we used BRAM since entire tuples (consisting of 15 dimensions and more) need to be stored in the working set. Unfortunately, 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 word. Nevertheless, as illustrated in Figure 14, copying is still possible without reducing throughput.

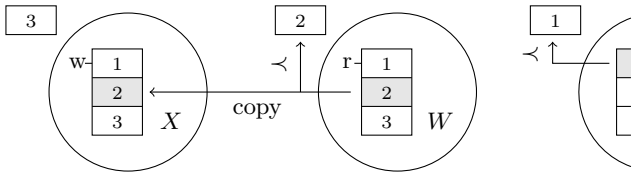

Figure 14: BRAM copy mechanism.

The first node (on the left), in Figure 14, is in state X (deleted). The second node is in state W (working set), which means that by shifter list semantics these two nodes need to be *swapped*. The node in state W is performing dominance tests against input tuples streaming by (each numbered box above the nodes corresponds to one dimension of the input tuple). As data is read (indicated by the r-flag) from BRAM of the node in state W for the dominance tests, at the same time this data can be written (indicated by the w-flag) to the BRAM of the 'deleted' predecessor node. If the input tuple did not dominate the current working set tuple, the two nodes exchange states, otherwise the second node is also 'deleted'. Notice that with *dual-ported* BRAM this mechanism can also be used to copy data in both directions simultaneously, *e.g.*, to *swap* working set contents.

6. EXPERIMENTS

In our experiments, we first compare the FPGA-based skyline operator against a single-threaded software implementation. On the one hand, we want to give a better understanding of the performance characteristics of the BNL algorithm, and highlight what the relevant parameters are that influence performance. On the other hand, we demonstrate the *linear scalability* of our approach, and show how throughput increases as we add more parallel resources, *i.e.*, shifter list nodes. Finally, in Section 6.5, we also put our results in relation to a state-of-the-art parallel skyline implementation (PSkyline [21]) for multicore systems.

6.1 Experimental Setup

On the FPGA side we used a Xilinx XUPV5 development platform with a Virtex-5 FPGA (XC5VLX110T) clocked at 150 MHz. All experiments were run from main memory. The XUPV5 board ships with 256 MiB DDR2 memory, which we accessed using MSR's Speedy DDR2 controller [2].

The single-threaded CPU experiments were carried out on an Intel Xeon 2.26 GHz server processor (Gainestown, L5520) equipped with DDR3 memory. The multicore experiments, in Section 6.5, were conducted on the same Intel Xeon server (8 cores plus hyper-threading) and on a 64-core (AMD Bulldozer, 2.2 GHz, DDR3 memory) PowerEdge R815 Server from Dell.

6.2 Characteristics of BNL

Before we present throughput measurements, in the next section, we first want to explain some characteristics of the BNL algorithm. The main objective of original BNL [4] as an *external* algorithm was to reduce I/O operations. The number of I/O operations depends on the number of *overflow tuples*. Figure 15 shows that the number of *overflow tuples* decreases almost linearly as we increase the size of the window. Thus, with I/O being the main bottleneck, a larger window directly translates into higher throughput. However, observe that the number of tuple *comparisons* in Figure 15 does not improve with a larger window. Hence, if we run BNL in main memory of a CPU-based system, we expect that the size of the window has little effect on runtime unless we add more compute resources.

6.3 Effects of Data Distribution

In the following experiments we evaluate the *throughput* of our FPGA-based skyline operator versus a CPU-based BNL [4] implementation. The input data consists of 1,024,000 seven-dimensional input tuples. Each dimension is represented by a 32-bit integer. Thus, together with a 32-bit *timestamp* a single tuples is 32 bytes wide and the size of the entire input set is 31.25 MiB, which is stored in (off-chip) DRAM memory. Synthetic input data was generated according to three different distributions: (1) *random*, (2) *correlated*, and (3) *anti-correlated*. These distributions are

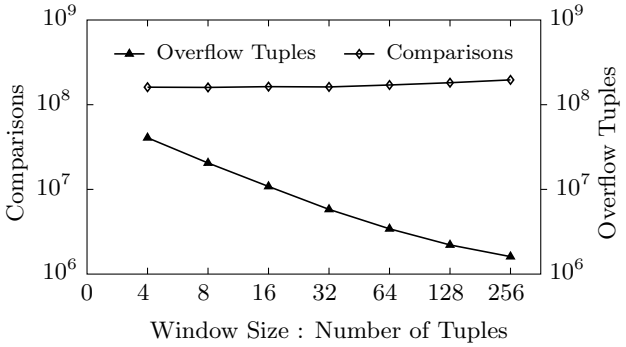

Figure 15: BNL: #Overflow tuples (I/O) versus #tuple comparisons (compute).

commonly used to evaluate skyline operators. To generate the data we used the data generator[2] provided by [4]. In the following measurements we will always indicate for which window size BNL achieved the best results by showing the exact execution time for the respective window size.

6.3.1 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* as number of input tuples processed per second and on the x-axis we vary the size of the window used in the BNL algorithm.

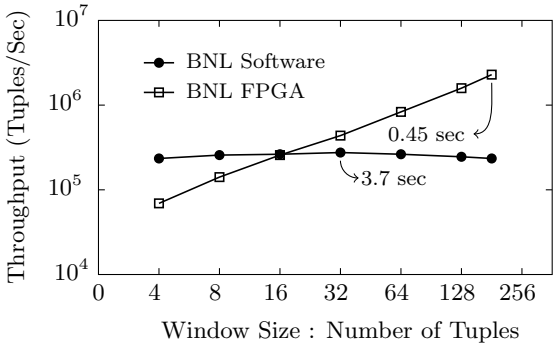

Figure 16: Random distr. → tuples/second.

As we already mentioned before, we expect the size of the BNL window to have little effect in the CPU-based version. On the FPGA, however, throughput increases linearly with the size of the window because in the FPGA case a larger window also means more shifter list nodes or a higher degree of parallelism. Since the BNL algorithm here is compute-bound, we can significantly increase throughput by performing more dominance tests in parallel.

6.3.2 The Correlated Case

In the second experiment, we compute the skyline on data that favors the CPU-based implementation. If the computational effort per input tuple is very low, aggregated compute power is no longer the key criteria for a fast execution of the algorithm and the CPU will be faster than the FPGA. This is the case, when the dimensions of the input tuples are strongly correlated, *i.e.*, when a tuple is "good" in one dimension, it is likely to be "good" also in the other dimensions. As a result, the skyline is very small, *e.g.*, in this experiment, the skyline consists of only 135 tuples, which corresponds to a density of *0.013%*.

In Figure 17, the CPU-based version of BNL is faster than the FPGA-based one because the upper bound for throughput is also higher for the CPU than for the FPGA due to the faster memory subsystem (DDR3 plus caches versus DDR2 and no caches). The upper bounds are depicted in the figure for both CPU and FPGA by a dashed line and a dotted line, 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 [12].

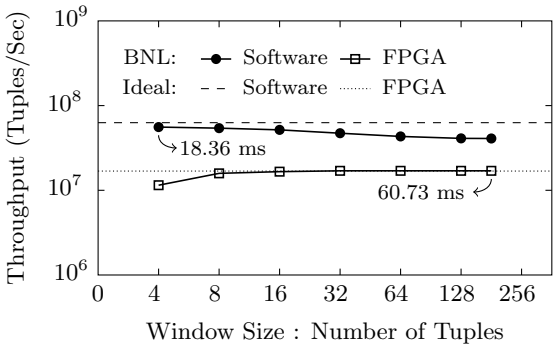

Figure 17: Correlated distr. → tuples/second.

For the CPU version, throughput decreases slightly with a larger window. The reason for this is that a larger window means that more unnecessary tuple comparisons are performed (for all of those potential skyline tuples that will later be eliminated). By contrast, in the FPGA case, the additional comparisons due to the larger window do not hurt since they are computed in parallel. Therefore throughput increases until we hit the limits of our memory subsystem.

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 above query takes *18.36 ms* on the CPU and *60.73 ms* on the FPGA. Thus, for many use cases with a reasonable number of input tuples, when the data is strongly correlated, this performance difference will not be very noticeable.

6.3.3 The Anti-Correlated Case

This experiment is the opposite of the previous one. Here, the dimensions of the input tuples are *anti-correlated* meaning 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%*.

Observe that the computation of the skyline is now sig-

[2]http://www.pubzone.org/pages/publications/showWiki.do?task=showComment&commentId=201&publicationId=298353&versionId=298378

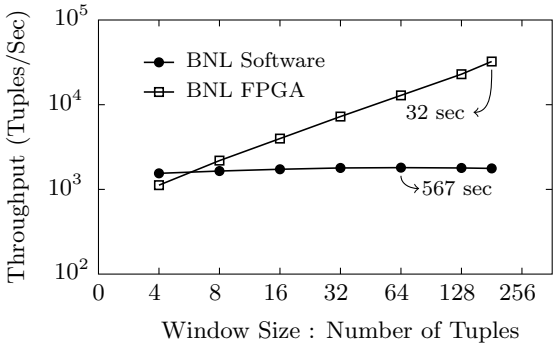

Figure 18: Anti-correlated distr. → tuples/second.

nificantly more expensive, *e.g.*, the best execution time of the CPU-based version has gone from *18.36* milliseconds to almost *10* minutes. The slowdown can be explained by the increase in 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 $\mathcal{O}(n^2)$ [12]. With this many comparisons—we measured an average of $\sim$25,000 comparisons per input tuple—the skyline query becomes highly compute-bound. However, in the FPGA case, the cost of computation can be reduced with every additional shifter list node.

6.4 The Curse of Dimensionality

Besides data distribution also the number of dimensions severely affect performance in skyline queries. Increasing the number of dimensions of the input tuples naturally increases the size of the skyline. For example, in an experiment with 102,400 15-dimensional input tuples following a *random distribution*, the density of the skyline was *74.86%*. As can be seen in Figure 19, the FPGA achieved a 19X improvement over the software algorithm in this case.

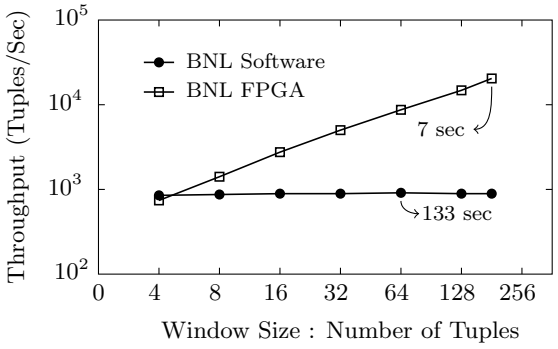

Figure 19: 15 dimensions (random distribution).

6.5 FPGA versus Multicore Server

The previous experiments demonstrated how our FPGA-based implementation in most cases significantly outperformed the CPU-based BNL implementation. In this section, we compare our FPGA results to *PSkyline* [21], which is the fastest published skyline algorithm for multicore architectures. We ran PSkyline on the same data sets as in the previous experiments that consisted of 1,024,000 seven-dimensional input tuples following a *random*, *correlated*, and *anti-correlated* distribution. We measured the performance of PSkyline on the 8-core (plus hyper-threading) 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 1.

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 1: Execution time: FPGA versus multicore.

On the Intel Xeon server best results were obtained using 16 threads respectively, and on the PowerEdge server using 64 threads. Notice that the performance for the compute-intensive workload (anti-correlated) we get out of our $750 (academic price) FPGA is not so far from the performance we measured on the $12,000 PowerEdge 64-core server.

It is also important to note that with 192 shifter list nodes a throughput of $\sim$32,000 tuples/second is reached on the FPGA. This is almost two orders of magnitude below the upper bound of $\sim$16,000,000 tuples/second (derived from the performance of the memory subsystem). Therefore there is still a lot of leeway to further increase performance by adding more shifter list nodes. The number of shifter list nodes that we can put on an FPGA is limited by FPGA real estate. Therefore, a larger FPGA configured with a few thousand shifter list nodes would again dramatically change the picture.

7. OTHER ALGORITHMS (SKETCHES)

While we have applied shifter lists successfully to various other database tasks, it is beyond the scope of this paper to present these implementations in detail. Nevertheless, in this section, we want to at least briefly sketch how shifter lists could be applied to two other common database operators: *(i)* frequent item computation and *(ii)* top-k queries.

Frequent Item Computation. Many data mining techniques start their data analysis by first looking at the most frequently occurring items in a given input data set. In [25], the computation of *frequent items* was solved on an FPGA with a variation of the *Space-Saving* algorithm [17], which achieved three times higher throughput as the best known software results. Out of the three ad-hoc implementations discussed in [25], the best one (Pipeline) can elegantly be expressed using the shifter list abstraction.

The *Space-Saving* algorithm works as follows: n bins are used to count the frequencies of the most frequent items. For every input, item the count of the corresponding bin is incremented if such a bin exists. Otherwise, the bin with the *lowest count* is evicted and assigned to the new item, which then inherits the incremented count of the evicted one [17].

For the parallel version of *Space-Saving* with shifter lists, each bin is mapped to a shifter list node and we stream input items through the list, just as we did for skyline queries. In the *evaluation phase*, on each node we check if the node-local input item matches the respective bin. If this is the case, we simply increment the count and stop forwarding the item to the subsequent shifter list node.

If the entire shifter list does not contain a matching bin, we need to find the bin with the *lowest count* for eviction. Finding that bin becomes trivial if we use the *shift phase* to ensure that the bin with the minimum count is always located at the rightmost shifter list node. To this end, the bins of two adjacent shifter list nodes are compared after the evaluation phase. If necessary, these bins are swapped in the following shift phase to ensure that the bin with the lowest count always resides at the rightmost shifter list node.

Top-k Queries. We may also use a shifter list for general top-k selection queries that rank results according to some compound *scoring function* over the attributes of a tuple. For this purpose, we need to modify the shift and evaluation phases from the frequent item algorithm only slightly: Instead of *counts*, we now compute *scores* of tuples and store those in the bins. Likewise, we need to modify the swap condition such that it is based on scores now. Notice how this swapping mechanism will eventually sort the shifter list and is essentially equivalent to a parallel version of *bubble sort*.

8. CONCLUSIONS

In this paper, we presented a new data structure—a shifter list—for FPGAs that helps in the parallelization of several related database tasks. Shifter lists comprise both data storage and computation in the sense that replicated logic units are directly woven into the storage resources of the data structure. This enables very high degrees of *parallelism*. In addition, the pipelined architecture of a shifter list exhibits very good *scalability* and enforces *causality guarantees* that for many algorithms makes parallelization easy and effective.

Out of several related database tasks suitable for an implementation with shifter lists on an FPGA, we chose skyline queries as one possible use case. The compute-intensive nature of skyline queries makes them attractive to be offloaded to a dedicated co-processor such as an FPGA. Our experiments show that our implementation on a rather low-end FPGA significantly outperforms a single-threaded software version of BNL on a CPU-based system and delivers performance results close to those obtained from running a highly-optimized parallel skyline algorithm (PSkyline [21]) on a modern multicore server using all available 64 cores.

9. REFERENCES

[1] Krste Asanovic et al. A View of the Parallel Computing Landscape. *Commun. ACM*, 2009.

[2] Ray Bittner. The Speedy DDR2 Controller For FPGAs. In *ERSA*, 2009.

[3] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Commun. ACM*, 2011.

[4] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The Skyline Operator. In *ICDE'01*, Heidelberg, Germany, 2001.

[5] Sung-Ryoung Cho et al. VSkyline: Vectorization for Efficient Skyline Computation. *SIGMOD Rec.*, 2010.

[6] Eric S. Chung, James C. Hoe, and Ken Mai. CoRAM: An In-Fabric Memory Architecture for FPGA-based Computing. In *FPGA'11*, Monterey, CA, USA, 2011.

[7] Convey Computer Corp. http://www.conveycomputer.com.

[8] Christopher Dennl, Daniel Ziener, and Jürgen Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using A Partially Reconfigurable Module Library. In *FCCM'12*, Toronto, ON, Canada, 2012.

[9] Hadi Esmaeilzadeh et al. Dark Silicon and the End of Multicore Scaling. In *ISCA'11*, San Jose, CA, USA, 2011.

[10] Philip W. Frey et al. A Spinning Join That Does Not Get Dizzy. In *ICDCS'10*, Genova, Italy, 2010.

[11] Kun Gao et al. Simultaneous Pipelining in QPipe: Exploiting Work Sharing Opportunities Across Queries. In *ICDE'06*, Atlanta, GA, USA, 2006.

[12] Parke Godfrey, Ryan Shipley, and Jarek Gryz. Maximal Vector Computation in Large Data Sets. In *VLDB'05*, Trondheim, Norway, 2005.

[13] David Greaves and Satnam Singh. Kiwi: Synthesis of FPGA Circuits from Parallel Programs. In *FCCM'08*, 2008.

[14] Shan Shan Huang et al. Liquid Metal: Object-Oriented Programming Across the Hardware/Software Boundary. In *European Conference on Object-Oriented Programming*, Paphos, Cyprus, 2008.

[15] Hiroaki Inoue, Takashi Takenaka, and Masato Motomura. 20Gbps C-Based Complex Event Processing. In *FPL'11*, Chania, Crete, Greece, 2011.

[16] Dirk Koch and Jim Torresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting. In *FPGA'11*, Monterey, CA, USA, 2011.

[17] Ahmed Metwally, Divyakant Agrawal, and Amr El Abbadi. An integrated efficient solution for computing frequent and top-k elements in data streams. *ACM Transactions on Database Systems (TODS)*, 2006.

[18] Roger Moussalli et al. Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In *ICDE'11*, Hannover, Germany, 2011.

[19] René Müller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *SIGMOD'10*, Indianapolis, IN, USA, 2010.

[20] Netezza Corp. http://www.redbooks.ibm.com/abstracts/redp4725.html.

[21] Sungwoo Park, Taekyung Kim, Jonghyun Park, Jinha Kim, and Hyeonseung Im. Parallel Skyline Computation on Multicore Architectures. In *ICDE'09*, Shanghai, China, 2009.

[22] Mohammad Sadoghi et al. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. In *VLDB'10*, Singapore, 2010.

[23] Satnam Singh. Computing without Processors. *Commun. ACM*, 2011.

[24] Jens Teubner and René Müller. How Soccer Players Would do Stream Joins. In *SIGMOD'11*, Athens, Greece, 2011.

[25] Jens Teubner, René Müller, and Gustavo Alonso. FPGA Acceleration for the Frequent Item Problem. In *ICDE'10*, Long Beach, CA, USA, 2010.

[26] R. Torlone and P. Ciaccia. Which Are My Preferred Items? In *Workshop on Recommendation and Personalization in eCommerce (RPEC)*, Malaga, Spain, 2002.

[27] Guozhang Wang et al. Behavioral Simulations in MapReduce. In *VLDB'10*, Singapore, 2010.