Shifter Lists—A Data Structure for Massive Parallelism

Louis Woods Jens Teubner Gustavo Alonso
Systems Group, Department of Computer Science, ETH Zurich, Switzerland
{louis.woods,jens.teubner,gustavo.alonso}@inf.ethz.ch

ABSTRACT

A wide range of technology trends are creating the opportunity to use highly parallel co-processors next to conventional CPUs to improve the performance of data processing systems. However, it is often difficult to exploit the inherent parallelism in these devices. Most systems available focus on ad-hoc implementations of single operators. Rather than trying to parallelize a given operator, in this paper, we propose a new data structure—*shifter lists*—designed to support data processing on massively parallel hardware (hundreds to thousands of processing elements). A shifter list can be customized to accommodate different algorithms while guaranteeing throughput-optimized communication among processing elements. In the paper, we present shifter lists, characterize their behavior, show a first implementation (on an FPGA, for large-scale parallelism), apply the result to a use case (skyline queries), and show how shifter lists can be used to implement other operators.

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. For instance, server vendors such as Netezza [21] (acquired by IBM in 2010) and Convey [7] equip their systems with configurable hardware accelerators known as *field-programmable gate arrays* (FPGAs), which are inherently parallel devices.

This new hardware provides a large aggregated compute power but it is difficult to turn the hardware's parallelism into true performance. A common answer to this challenge is to build specialized implementations focused on one specific application 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 to carry over from one application to another.

In this work we provide one such abstraction. We propose a new data structure—*shifter lists*—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 emerging parallel hardware. 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. 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.

We illustrate shifter lists based on a common database operator, *skyline*. 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. With shifter lists, by contrast, we partition the *working set* of a *block-nested-loops* (BNL) [4] variant and leverage the data structure's lightweight partitioning mechanisms across many parallel processing units.

Shifter lists are a generic data structure that can be used at different levels of granularity. In this paper, we aim at scalability to very high degrees of parallelism. For evaluation, we thus use *field-programmable gate arrays* (FPGAs), where we can experiment with degrees of parallelism far beyond those of commodity multi-core hardware (on our FPGA hardware, we could accommodate almost two hundred parallel processing units). Though absolute performance is not the main focus of this paper, our experiments show that we outperform a CPU-based solution by almost a factor of 20, even on low-end FPGA hardware. As such, our prototype could readily be used as a *co-processor* to accelerate skyline applications.

We present shifter lists 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. We realize the idea on concrete (FPGA) hardware in Section 5, and evaluate our implementation 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."

2.1 Parallel Hardware & Heterogeneity

The trend toward increasing degrees of parallelism is complemented with a notion of hardware *heterogeneity*. Specialized co-processors and even programmable logic are already used successfully to assist general-purpose cores on compute- or data-intensive tasks.

A particular instance of this trend are *field-programmable gate arrays* (FPGAs), which proved already very successful in the database domain. FPGAs are digital logic devices that can be used to realize a circuit by a mere (software-based) device (re)configuration. The configured hardware logic can then solve a particular problem at very high speeds and with favorable energy consumption properties.

IBM/Netezza's 1000 (formerly *Twinfin*) [21] is probably the most prominent example of a commercial FPGA-powered database appliance. On the research side, FPGA solutions have been proposed, *e.g.*, for sorting [20], XML filtering [19], or high-speed event processing [23]. 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. With shifter lists, we work towards such an abstraction that aids in building parallel solutions for difficult database tasks that demand high performance.

From a research perspective, FPGAs are particularly interesting because of their inherent massive parallelism. The degree of parallelism is limited mainly by the amount of available *chip space* on the given FPGA. At the same time, the observed effects of increased parallelism are representative for what we can expect from future commodity hardware. Thus, FPGAs can be used already today to illustrate the interplay of communication and high degrees of parallelism on hardware yet to come.

2.2 Parallel Data Processing

The database community has investigated parallel data processing techniques already long before the current multi-core race. A good overview of parallel databases has been published, *e.g.*, by DeWitt and Gray [9]. Parallelism, thereby, was used in essentially two ways: *(i) data partitioning*, large input data is partitioned to run parallel instances of the same operator on many nodes (*e.g.*, [17]), and *(ii) inter-operator pipelining*, where pipelined query plans are taken literally and individual operators are assigned to distinct processing resources (*e.g.*, [12, 16]). As noted in [9], inter-operator pipelining bears a risk of uneven load distribution because different operators within the same pipeline may have significant differences in cost.

Modern Hardware. All of the above techniques (and research on parallel databases in general) were a good fit for the available hardware at the time. Meanwhile, however, hardware characteristics—in particular with respect to the availability of parallelism—have changed considerably. In this work, we thus propose to *re-think* parallel algorithm design and devise a new programming model that matches the actual trends in hardware.

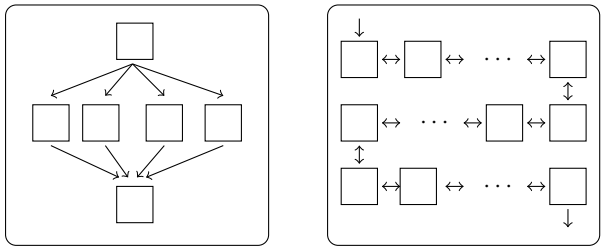

Figure 1: Scatter-gather versus pipeline of cores.

Most importantly, this affects the interplay of parallelism with *communication*. 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.

2.3 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 units (Figure 1), or *ring* and *tree* topologies.

Most existing algorithms for parallel database processing barely reflect these effects induced by communuication. Most recently, for instance, Kim et al. [15] or Blanas et al. [2] devised parallel variants of *join algorithms* for multi-core hardware. While the primary focus of these techniques—cache awareness—can be viewed as one particular type of communication (to main memory), neither technique is really aware of the implied inter-core communication.

One of the first works that addressed core-to-core communication in databases is *QPipe* by Gao et al. [12]. *QPipe* places database operators on individual cores 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.

Inter-operator pipelining has been discussed in a few recent papers. In *cyclo-join* [11], 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 data structure for highly parallel hardware. As such, they are more related to—and in fact a generalization of—the data structures in our earlier work [25, 29]. Shifter list-like constructs were used in [29] to perform efficient key-value search in the context of `GROUP BY`, and in [25] to solve the frequent item problem (see Section 7).

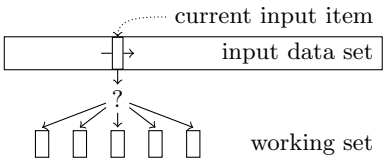

Figure 2: Typical application pattern for a shifter list: For each input item, the working set is accessed and possibly modified.

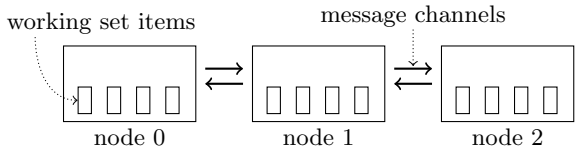

Figure 3: Shifter lists group working set items into nodes. Neighboring nodes are connected via message channels.

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

2.4 Skyline Processing

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 of the general directions of approaches is given in [14]). To demonstrate shifter lists, we revisit block-nested-loops (BNL) [4], one of the early skyline algorithms, which is simple and yet still effective today.

Recently, there have also been some approaches to exploit parallelism for skyline queries. On multi-core machines, the main problem is that multi-threaded skyline algorithms using traditional approaches often only scale up to a few cores [22]. Using SIMD, the dominance test of skyline queries [5] can be improved. However, maximal theoretical speedup is limited by the vector size of the SIMD registers.

3. SHIFTER LISTS

The main target of our shifter list data structure are application patterns as illustrated in Figure 2. From a given input data set, all items are consumed in turn. The single input item is evaluated against many or even all the items in an in-memory *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. Many common database tasks match this pattern, *e.g.*, *skyline* queries, top-k queries, k-nearest neighbor searches (k-NN), and finding frequent items, just to name a few.

In this work, we use *skyline computation* to showcase shifter lists. We are going to base our work on the block nested loops (BNL) algorithm [4] that, for each input item, examines and updates a working set—consistent with the pattern in Figure 2.

Note that for some algorithms used in the examples given above the processing *order* may influence the final result. Many applications demand that this *causality* implied by in-order processing is preserved also in a parallel execution scheme.

3.1 A Shifter List is a Data Structure

The high-level structure of shifter lists is illustrated in Figure 3. Working set items are held in a number of *shifter list nodes* (each of which we will later assign to a separate compute resource). There is a defined total *order* among all nodes ν_i in a shifter list. Oftentimes, node contents themselves will have a defined order, resulting in a total order 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 *neighbor-to-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).

Ready for Modern Hardware. A working set organized as in Figure 3 is well prepared for the runtime characteristics of modern and future hardware. Grouping and neighbor-to-neighbor communication both ensure spacial *locality*. Awareness of communication locality is exactly among the properties that Borkar and Chien demand from the software side if we want to see continuous performance growth on future hardware generations [3].

A possible way to implement messaging channels on commodity systems is to use *asynchronous FIFO queues*. Such queues were shown to match the capabilities of modern multi-core systems particularly well [13] and—if organized in a linear structure like shifter lists—scale to large core counts almost trivially. In FPGA designs, the point-to-point nature of the channels avoids costly multiplexing logic and reduces circuit complexity.

3.2 Shifter Lists are for Data Processing

To process input, we submit each input item to the left-most shifter list node. 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 (more details later).

This self-similarity property not only eases application development. It also means that we can slice the original task into smaller and smaller units in a hierarchical fashion. Again, this is in line with what we see on the hardware side where, *e.g.*, functional units are combined to form one CPU core, several CPU cores make one chip die, dies are packaged into processors, etc. (*e.g.*, [8]). Conceivably, shifter lists could be applied across all these levels, and even beyond machine boundaries at the network and data center level.

3.3 Shifter Lists are 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 units 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 may be a desirable property for some algorithms). What is more, once an input item x_i has reached a shifter list node ν_m, its evaluation cannot be affected by any later input item x_j that is evaluated over a preceding node $\nu_n, n < m$ (while conversely, the later x_j is guaranteed to see all effects caused by the earlier x_i).

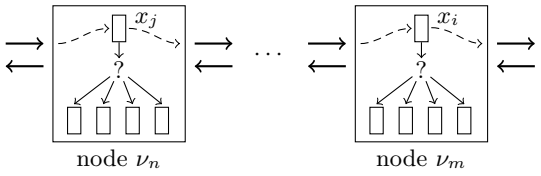

Figure 4: Shifter list causality guarantees. The earlier x_i will see no effects caused by the later x_j but x_j sees all effects of x_i.

These causality guarantees hold even if we let the executions of x_i on ν_m and x_j on ν_n 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. Explicit *barriers*, placed between successive input items, are an easy method to prevent such risks (also see Section 7.1).

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 on the rightmost position. Since items never overtake each other, this ensures that the oldest working set element can always be found at the front of node 0 (the leftmost shifter list node).

3.4 Shifter Lists are 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 logic and data.

Releasing this strict separation indeed makes sense in the light of 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, 10]. In such designs, data structures can be wrapped right into the corresponding processing logic to further improve energy efficiency and speed.

In the experimental part of this work, we make the integration of data and logic very explicit by implementing shifter lists on top of FPGAs. The circuit that we propose would be ready to become one part of a heterogeneous multi-core architecture.

4. USE CASE: SKYLINE QUERIES

In a data warehouse *appliance* equipped with configurable hardware, *e.g.*, IBM/Netezza's 1000 (formerly *Twinfin*) [21], a significant performance optimization can be achieved when *compute-intensive*, long-running queries are outsourced to a dedicated co-processor previously loaded onto the configurable hardware. *Skyline* queries, discussed in this section, are typically very compute-intensive and are related to many other well-known database problems, *e.g.*, computing the *convex hull* for a set of points, making skyline queries a good candidate for shifter lists.

4.1 The Lemming Skyline

To figuratively explain *skyline queries*, the BNL algorithm [4], and the modified version of BNL using shifter lists, we 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 are pretty good in multiple skills. 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 those 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 $L = \{l_1, l_2, \ldots l_n\}$, the skyline query returns a set of Lemmings S, such that any Lemming $l_i \in S$ is not dominated by any other Lemming $l_j \in L$.*

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

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.

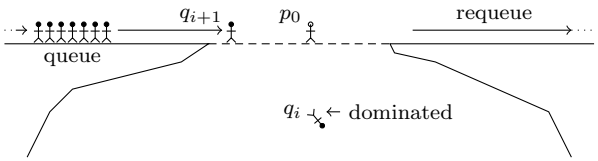

Figure 5: Lemming Skyline with *Best* [26].

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].

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.

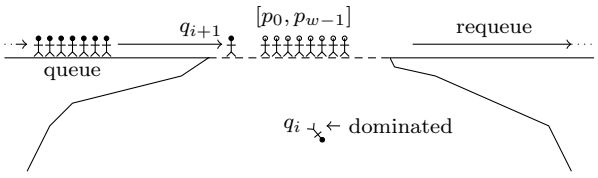

Figure 6: Lemming Skyline with *BNL* [4].

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.

```
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).

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

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

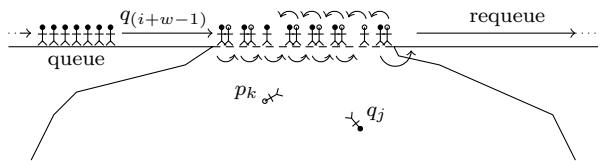

Figure 8: Lemming Skyline: BNL & Shifter List.

$q_{(i+w-1)} \dots q_i$ 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 can go first. Thus, candidate Lemmings get on the bridge on the right end 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 (no need for external storage) and efficient memory subsystems, the bandwidth to read input or overflow data limits skyline computation only for extremely small working set sizes (few candidate skyline tuples). In most practical cases therefore, CPU load becomes the bottleneck when computing skylines.

This makes FPGAs a good alternative to conventional CPUs, because the relevant window sizes (say, 100 skyline candidates) conveniently fit into on-chip memories. In this section, we show how shifter lists then 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 brings the algorithm back to memory-bound behavior, which in turn 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-granular 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.

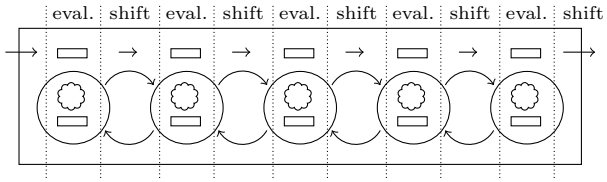

Figure 9: Two-phase processing of shifter lists.

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.

```
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.

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 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).

```
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.

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 ($W \to O$).

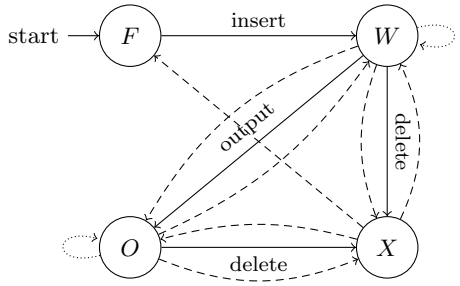

Figure 12: State diagram: shifter list node.

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 might need the global working set content of a shifter list to remain sorted (see Section 7.2). 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 multi-dimensional input, this would lead to a very high bandwidth demand (*e.g.*, 15 dimensions × 32 bits × 150 MHz clock frequency = 9 GB/s in- *and* outgoing traffic), which is far out of reach for the hardware that we use. Instead, our implementation *streams* all data one dimension at a time.

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).

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. Modern FPGAs ship with two types of on-chip memory: *registers* and *block RAM* (BRAM). *Registers* are composed from flip-flops and configurable logic. They provide flexible usage but have limited storage capacity. *Block*

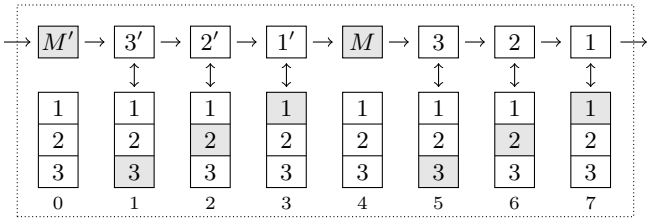

Figure 13: Dimension-at-a-time processing. Two tuples streaming by eight shifter list nodes.

RAM (BRAM), on the other hand, refers to blocks of dedicated memory (distributed across the FPGA) with address, data, and control ports and much more capacity.

Shifter lists can be implemented using either memory type 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.

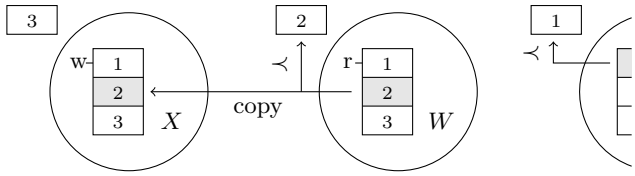

Figure 14: BRAM copy mechanism.

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 from BRAM of the node in state W for the dominance tests, at the same time this data can be written 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* of working set contents.

6. EXPERIMENTS

In our experiments, we compare the FPGA-based skyline operator against a software implementation. On the one hand, we demonstrate the *scalability* of our approach and show how *throughput* increases as we add more shifter list nodes. On the other hand, our results are put in relation to throughput measurements of a (faster clocked) CPU-based BNL implementation.

On the FPGA side we used a low-cost ($750.00 university price) Virtex-5 FPGA (XC5VLX110T) from Xilinx clocked at *151.8 MHz*. Our CPU-based experiments were carried out on an Intel Xeon *2.26 GHz* server processor (Gainestown, L5520).

6.1 Characteristics of BNL

Before we present throughput measurements, in the next section, we first want to explain some characteristics of the block-nested-loops (BNL) algorithm. The main objective of original BNL [4] as an *external* algorithm was to reduce I/O operations. The larger the window, the fewer runs are needed, as can be seen in Figure 15, where 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 a higher throughput. However, observe that the number of tuple *comparisons* (see Figure 15) does not improve with a larger window. In fact, the number of comparisons might even slightly increase. Hence, if we run BNL in main memory the size of the window has little effect on runtime.

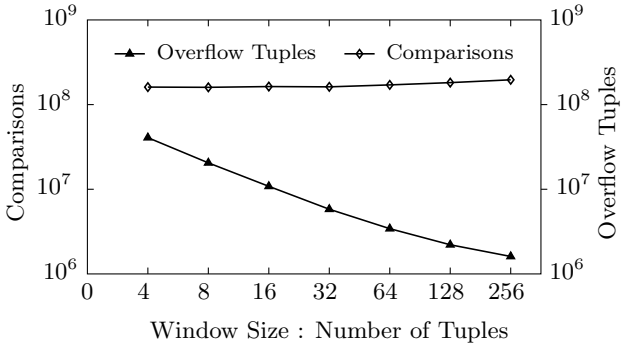

Figure 15: BNL: Comparisons vs. overflow tuples.

Nevertheless, BNL as a main memory algorithm still exhibits reasonable performance. For the experiment in Figure 15, the skyline was computed over 1,024,000 input tuples of seven dimensions each following a uniform random distribution. It took *3.7 seconds* to compute the resulting skyline of 15,154 tuples on this data. As a comparison, we also ran a newer block-nested-loops algorithm designed for fast in-memory processing called *SSkyline*, which was recently presented in [22]. For *SSkyline* we measured an execution time of *3.3 seconds*. While *SSkyline* here is indeed a bit faster (speedup = 1.12X), it is not orders of magnitude better than BNL, and if we increase the number of dimensions or the correlation of the tuple dimensions, the two algorithms run practically at same speed.

In Figure 15 the number of comparisons increases with a larger window. This would suggest that a window size of one—BNL with a window size of one is referred to as an algorithm called *Best* [26]—should yield the best results. However, despite more comparisons BNL with a small window size executes faster than *Best*. In such a configuration, the system becomes bottlenecked by memory bandwidth, because of a very large number of overflow tuples. In the following measurements we will always indicate for which window size BNL achieved the best results.

6.2 Effects of Data Distribution

In the following experiments we evaluate *throughput* performance of our FPGA-based skyline operator versus a CPU-based block-nested-loops (BNL) implementation. Again,

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 *sequence number* a single tuples is 32 bytes wide and the size of the entire input set is 31.25 MiB. We use synthetic input data with three different distributions: (1) *random*, (2) *correlated*, and (3) *anti-correlated*. These distributions are commonly used to evaluate skyline operators. To generate the data we used the data generator[2] provided by [4].

6.2.1 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.

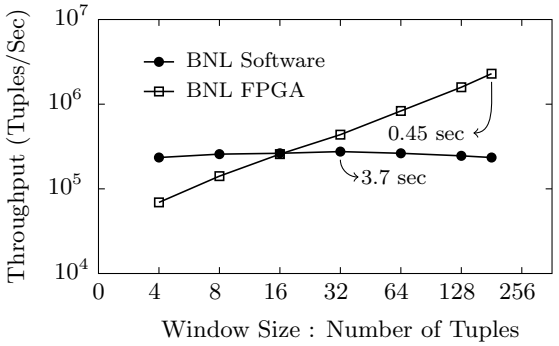

Figure 16: Random distr. → tuples/second.

As we already noted before, the size of the BNL window has little effect in the CPU-based version. On the FPGA, however, throughput increases linearly with the size of the window. This is not surprising because in the FPGA case a larger window also means more shifter list nodes or higher degrees of parallelism. Since the BNL algorithm here is compute-bound, we can significantly increase throughput by performing more dominance tests in parallel. Also notice that the frequency of the CPU with 2.26 GHz is roughly 15 times higher than that of the FPGA, which is 151.1 MHz.

Here, BNL executes the fastest (3.7 seconds) with a window size of 32. The best FPGA results are at a window size of 192 with an execution time of 0.45 seconds. The break-even-point between the two versions is at a window size of 16.

6.2.2 The Correlated Case

In the second experiment, we compute the skyline on data that favors the CPU-based implementation. Our CPU runs at higher clock speed and has a faster memory subsystem than our FPGA. 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. For example, in our experiment, depicted in Figure 17, the skyline consists of only 135 tuples, which corresponds to a skyline tuple density of *0.013%*.

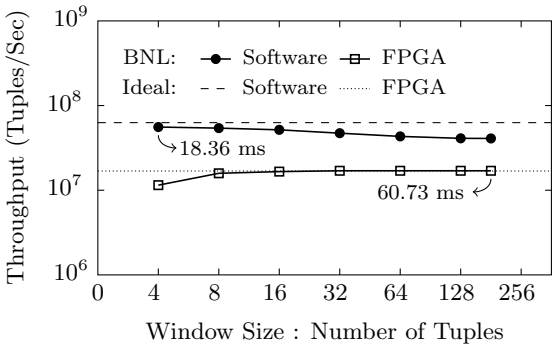

Figure 17: Correlated distr. → tuples/second.

The first observation is that the CPU-based version is indeed faster than the FPGA-based version. This is not surprising since the upper bound for throughput is also higher for the CPU than for the FPGA because of the reasons mentioned above. 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 [14].

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. 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 took *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. However, as we will see in the next section, with an increased *density* of skyline tuples the computation of the skyline becomes extremely expensive and calls for an optimized solution.

6.2.3 The Anti-Correlated Case

This experiment is the inverse of the previous experiment. 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%*.

Now, the computation of the skyline has become significantly more expensive, *e.g.*, the best execution time of the

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

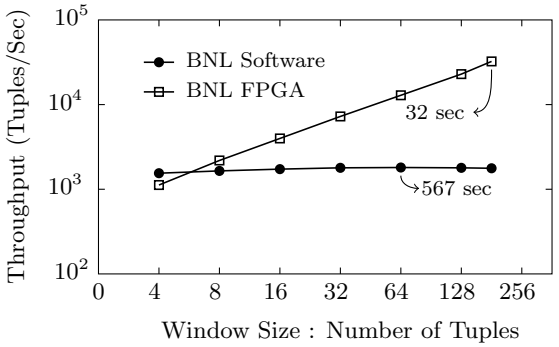

Figure 18: Anti-correlated distr. → tuples/second.

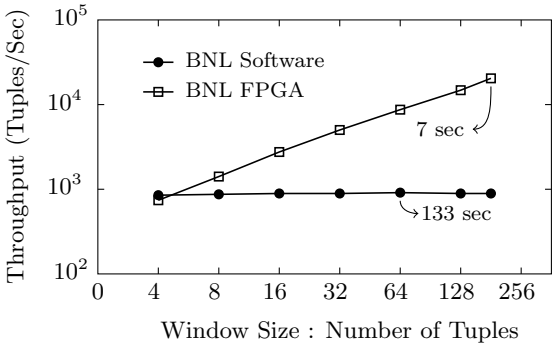

Figure 19: 15 dimensions (random distribution).

CPU-based version has gone from *18.36* milliseconds to almost *10* minutes, *i.e.*, more than four orders of magnitude slower than the correlated case. 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 $\mathcal{O}(n^2)$ [14]. Therefore, the number of comparisons becomes the dominating factor as the size of the skyline increases.

With this many comparisons—we measured an average of $\sim 25,000$ comparisons per input tuple—the skyline query becomes compute-bound. In the FPGA case, this cost can be reduced with every additional shifter list node. This observation is also confirmed by the measurements in Figure 18: y-axis and x-axis are in logarithmic scale and the throughput increases linearly with the size of the window, *i.e.*, the number of shifter list nodes.

With a 192 shifter list nodes we reach a throughput of $\sim$ 32 thousand tuples/second. This is almost two orders of magnitude below the upper bound of $\sim$ 16 million tuples/second for throughput (coming from 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. Our results suggest that a larger FPGA would further increase throughput, as (below the upper bound) there is a one-to-one relationship between chip space and throughput performance.

6.3 The Curse of Dimensionality

By now we know that the *size of the skyline* severely impacts the performance of the BNL algorithm. Increasing the number of dimensions of the input tuples naturally increases the size of the skyline. This phenomenon is known as *the curse of dimensionality* [28]. Here, we show an experiment with tuples of 15 dimensions each. The dimensions of every tuple follow a random distribution. Because of the increased computational intensity we had to reduce the input data set by a factor of ten. Out of the 102,400 input tuples the skyline here consists of 76,657 tuples which translates to a density of *74.86%*. Our results are displayed in Figure 19.

The graph above looks similar to the one in Figure 18. The break-even-point has moved even a bit further to the left, close to a window size of four. Thus, even though the CPU is clocked about 15 times faster than the FPGA, it cannot do 15 times more comparisons per clock cycle, otherwise the break-even-point should never be below a window size of 15. A single comparison of two 15-dimensional tuples on the FPGA takes 17 clock cycles, *i.e.*, 112.5 ns at a 151.1 MHz clock. We also measured how long a comparison takes on our CPU and the result was 34.4 ns. Thus, we need at least $\lceil \frac{112.5}{34.4} \rceil = 4$ shifter list nodes to achieve better results than the CPU.

6.4 Concluding Remarks

The important lesson to be learned from the experiments is that the *computational intensity* of skyline queries is mainly driven by the *density* of the skyline tuples, *i.e.*, the size of the skyline. The *density* of skyline tuples depends on how the dimensions of the tuples are distributed and more importantly on the number of dimensions. The FPGA-based skyline operator improves performance most when the *density* of skyline tuples is high as then the parallel compute power of the FPGA is used most effectively. For instance, with a *density* of 74.86% in the experiment in Section 6.3, the FPGA with 192 shifter list nodes achieves a 17.7X speedup over the CPU based version.

7. OTHER ALGORITHMS (SKETCHES)

So far, we have focused on how to solve one particular problem (skyline queries) with shifter lists—so that we could present the inner workings and performance characteristics of shifter lists in sufficient detail. However, the goal for developing this data structure was to make it applicable to more than one algorithm. In this section, we briefly sketch how shifter lists can be applied to other common database operators: *(i)* frequent item computation and *(ii)* top-k queries.

7.1 Frequent Item Computation

In [25], the computation of *frequent items* (in a streaming context) was solved on an FPGA with a variation of the *Space-Saving* algorithm [18]. The FPGA-based version relied heavily on pipelining (achieving three times higher throughput as the best known results). We now sketch how this algorithm can be reformulated using shifter lists. The original *Space-Saving* algorithm is depicted in Figure 20.

In this algorithm n bins are used to count the frequencies of the most frequent items in a stream. If a corresponding bin for a new stream item x exists, that item's frequency is increased (lines 3–4 in Figure 20). Otherwise, the bin with

```
1 foreach stream item x ∈ S do
2     find bin b_x with b_x.item = x ;
3     if such a bin was found then
4         b_x.count ← b_x.count + 1 ;
5     else
6         b_min ← bin with minimum count value ;
7         b_min.count ← b_min.count + 1 ;
8         b_min.item ← x ;
```

Figure 20: Algorithm *Space-Saving* [18].

the lowest count value gets evicted in favor of the new item (lines 5–8), which inherits the incremented frequency of its predecessor (see [18, 25] for details).

For the parallel version of *Space-Saving* with shifter lists, each bin is mapped to a shifter list node. We then need to specify the evaluation phase and the shift phase to implement the algorithm. Given the invariant that the bin with the minimum count value will always be located at the last shifter list node (this invariant will be established in the shift phase), the evaluation phase is straightforward, as illustrated in Figure 21. Note that bin count values are initialized to zero (not shown in the figure). As opposed to BNL, here, we explicitly output all items with frequencies above a given *threshold* by submitting a special input item (*stop token*) to the shifter list.

```
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
6          if s = working set then
7              if q.terminate then          /* stop token */
8                  if p.count > q.threshold then
9                      s ← output ;
10                 else
11                     s ← deleted ;
12             else if p.item = q.item then
13                 p.count = p.count + 1 ;
14         else if s = free then
15             p.count = p.count+1 ;
16             p.item ← q.item ;
17             s ← working set ;
18             q.valid ← false ;
```

Figure 21: Frequent item: evaluation phase.

We can use the shift phase (Figure 22) to ensure that the bin with the minimum count value is always located at the rightmost shifter list node. To this end, the count values of adjacent shifter list nodes are compared. If the *order invariant* (which may depend on state and working set content) of two nodes has been violated after the evaluation phase, the working set contents and states of those nodes are *swapped* in the subsequent shift phase (lines 11–12 in Figure 22). As discussed in [25], these swaps may introduce race conditions (which are not present in shifter list-based BNL due to the naturally enforced order by BNL). Fortunately, these race conditions can be avoided by leaving gaps in the input stream and coupling the swap operation with the valid flag of the input item (line 10 in Figure 22), *i.e.*, $k/2$ items can traverse a shifter list of length k simultaneously. Notice, that we alway update the *physically* last node of the shifter list—as specified in the original *Space-Saving* algorithm—by setting its state to 'free' every time in the shift phase.

```
1  foreach node ν_i do
2      if i = 0 ∧ ν_i.state = output then
3          emit ν_i.working set.item as result ;
4          ν_i.state ← deleted ;
5      if i < w − 1 then
6          if ν_i.state = deleted then
7              ν_i.working set ← ν_{i+1}.working set ;
8              ν_i.state ← ν_{i+1}.state ;
9              ν_{i+1}.state = deleted ;
10         else if ν_i.input item.valid then
11             if not order_invariant(ν_i, ν_{i+1}) then
12                 swap(ν_i, ν_{i+1}) ;
13         ν_{i+1}.input item ← ν_i.input item ;
14     else
15         ν_i.state ← free ;
```

Figure 22: Frequent item: shift phase.

7.2 Top-k Queries

A. Metwally et al. already showed in [18] that the *Space-Saving* algorithm could also be used to solve top-k queries, *i.e.*, return the k items with the highest frequencies instead of all items with a frequency above a certain threshold. Likewise, we can use a shifter list (with k shifter list nodes) 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 previous section only slightly. Instead of *count values* we store *scores* of tuples in the bins, as shown in Figure 23.

```
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
6          if s = working set then
7              if q.terminate then          /* stop token */
8                  s ← output ;
9          else if s = free then
10             if p.score < scorefunc(q.item) then
11                 p.score ← scorefunc(q.item) ;
12                 p.item ← q.item ;
13                 s ← working set ;
14                 q.valid ← false ;
```

Figure 23: Top-k: evaluation phase.

The shift phase is virtually identical to the one of the frequent item problem listed in Figure 22. We simply need to modify the swap condition such that it is based on scores rather than count values, *i.e.*, we need to replace the *order_invariant* function on line 11. It is possible that the item with the highest score is inserted last. Thus, to ensure that the items are output in proper order, we would need to have the shifter list process n dummy items after every run. Notice that the swapping mechanism showed here will eventually sort the shifter list and essentially is equivalent to a parallel version of *bubble sort*.

8. CONCLUSIONS

Parallelization of algorithms is often a non-trivial task, especially, when the problem is not *embarrassingly parallel*. In particular, when reasoning about parallelizing a specific task, the focus is often on the computation while the communication overhead is overlooked. But the communication overhead can become a severe bottleneck, particularly, as the core count increases.

The usual way to address the parallelism challenge has been to build a specialized parallel algorithm for each and every given problem. With *shifter lists*, we propose a general-purpose solution instead. A shifter list is a novel data structure tailored to tightly-coupled many-core systems. A special property of shifter lists is that they can be used to unify algorithmic data processing components with the data structure itself. We used an FPGA as a test platform for a many-core system of up to 192 processing elements and showed by example of a skyline algorithm how shifter lists help accessing this massive parallelism in a highly scalable way.

9. REFERENCES

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

[2] Spyros Blanas, Yinan Li, and Jignesh M. Patel. Design and Evaluation of Main Memory Hash Join Algorithms for Multi-Core CPUs. In *SIGMOD'11*, Athens, Greece, 2011.

[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] P. Conway et al. Cache Hierarchy and Memory Subsystem of the AMD Opteron Processor. *IEEE Micro*, 2010.

[9] David J. DeWitt and Jim Gray. Parallel Database Systems: The Future of High Performance Database Systems. *Commun. ACM*, 1992.

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

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

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

[13] John Giacomoni, Tipp Moseley, and Manish Vachharajani. FastForward for Efficient Pipeline Parallelism. In *PPoPP'08*, Salt Lake City, UT, USA, 2008.

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

[15] Changkyu Kim et al. Sort vs. Hash Revisited: Fast Join Implementation on Modern Multi-Core CPUs. In *VLDB'09*, Lyon, France, 2009.

[16] Ming-Ling Lo et al. On Optimal Processor Allocation to Support Pipelined Hash Joins. In *SIGMOD'93*, Washington, DC, USA, 1993.

[17] Manish Mehta and David J. DeWitt. Managing Intra-operator Parallelism in Parallel Database Systems. In *VLDB'95*, Zurich, Switzerland, 1995.

[18] 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.

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

[20] René Müller, Jens Teubner, and Gustavo Alonso. Data Processing on FPGAs. In *VLDB'09*, Lyon, France, 2009.

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

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

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

[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.

[28] Roger Weber, Hans-Jörg Schek, and Stephen Blott. A Quantitative Analysis and Performance Study for Similarity-Search Methods in High-Dimensional Spaces. In *VLDB'98*, New York City, USA, 1998.

[29] Louis Woods, Jens Teubner, and Gustavo Alonso. Complex Event Detection at Wire Speed with FPGAs. In *VLDB'10*, Singapore, 2010.