FPGAs for Data Processing: Current State

Jens Teubner

Abstract: To escape a number of physical limitations (e.g., bandwidth and thermal issues), hardware technology is strongly trending toward heterogeneous system designs, where a large share of the application work can be off-loaded to accelerators, such as graphics or network processors.
In the database domain, field-programmable gate arrays (FPGAs) were recently discovered as a powerful class of co-processors. Data intensive tasks can benefit from their very high bandwidth paired with a very low latency.
However, a number of challenges still have to be solved before FPGA-accelerated systems can go mainstream. Applications must leverage the massive parallelism provided by FPGAs; algorithms must be re-designed with awareness for communication; and new abstractions are needed to make system development effective.
In this article, we report on some of the results we obtained working toward solutions to these challenges. They have been carried out in the context of the Avalanche project, a pioneer in the use of FPGAs for database applications.

ACM CCS:

Keywords: FPGA; database; acceleration

1 Introduction

While hardware technology has always been marked by fast and significant advances, none of the previous changes has been as disruptive as the ongoing trend toward heterogeneity. All of today's processors are power-limited [1], and the use of specialized hardware is seen as the only promising escape to the growing energy limitation.

A particular brood of specialization are *field-programmable gate arrays (FPGAs)*. Unlike most other co-processor classes, FPGAs do not run sequential programs. Rather, they provide "uncommitted chip space." Through a (software-based) configuration process, this chip space can be used to to realize arbitrary logic circuits directly in hardware.

FPGAs were recently discovered as a promising platform to accelerate database tasks. But the characteristics, strengths, and weaknesses of the technology are still not widely known, nor are the challenges understood or solved that arise in FPGA-accelerated systems.

This article provides an overview not only of FPGAs as a technology. We further exemplify how FPGAs can be used effectively and we illustrate design principles that lead to good FPGA solutions. Much of the work that we present here is an outcome of the *Avalanche* project that has pioneered the use of FPGAs in a database context.

2 FPGA Technology

Any electronic circuit, whether realized using FPGAs or as a hard silicon component, is built from three principle types of resources:

(a) *Combinational logic* is composed of basic logic gates ('*and*', '*or*', etc.) and maps a set of (Boolean) input values $\bar{x} = (x_1, \ldots, x_n)$ to an output signal $f(\bar{x})$. f is a function in the mathematical sense. That is, $f(\bar{x})$ depends solely on the input signals x_i.

(b) *Memory elements* allow a circuit to keep *state*. Specifically, a *flip-flop* is a single-bit storage usually directly wired into the circuit.

(c) Finally, an *interconnect* wires up all logic resources to obtain a complete circuit.

Most circuits make use of an external *clock signal*, a periodically changing high/low signal. Flip-flops will save their input data value on every rising edge of the clock signal. This allows to build *sequential circuits* that, *e.g.*, can perform step-by-step operations or computations.

2.1 FPGA Architecture

FPGAs integrate configurable implementations of all three resource types. Figure 1 illustrates how an FPGA is built internally. The majority of the chip area is covered by a two-dimensional grid of *configurable logic blocks*,

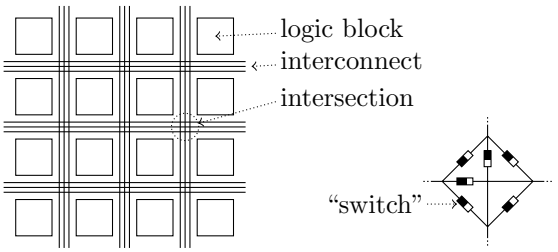

Figure 1: FPGA architecture. The chip hosts a 2d array of logic blocks with a configurable interconnect in-between. Each intersection is controlled by a set of "switches" (transistors) as illustrated on the right.

illustrated as squares in Figure 1. Logic blocks provide combinational logic and memory elements.

Logic blocks are interspersed with an *interconnect network*. Through this network, a direct line can be instantiated from any logic block to any other logic block. The illustration on the bottom-right of Figure 1 shows how this can be implemented in hardware. At any intersection point between a horizontal and a vertical wire, arbitrary wirings between the four ends of the intersection can be realized through a set of "switches" (illustrated as ⊣■⊢). In practice, these "switches" are transistors whose base is connected to an SRAM cell. The content of those SRAM cells thus defines the routing configuration of the interconnect. By (re-)writing the cells, the interconnect can be (re-)configured at any time.

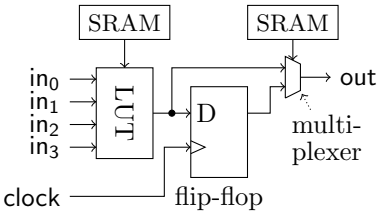

Figure 2: FPGAs package combinational logic (in the form of *lookup tables* or *LUTs*) and flip-flops into *configurable logic blocks*.

FPGAs provide combinational logic and memory elements (flip-flops) in the form of configurable logic blocks. A typical combination is illustrated in Figure 2. The *lookup table (LUT)*, shown on the left, is a generic implementation for a combinational circuit with up to four inputs and one output. It materializes $f(\bar{x})$ for all possible combinations of input values x_i and stores them in a 16-bit SRAM cell. Depending on the contents of a further SRAM cell, the *multiplexer* shown on the right enables or bypasses the flip-flop element shown in the middle. Modern FPGA devices use lookup tables with six input lines.

2.2 Additional Functionality

Typical FPGA devices include additional components, such as clock generators or I/O units to interface with the outside world. To exemplify, current Xilinx Virtex-7 chips provide an aggregate I/O bandwidth of up to 2.8 Tb/s.

Lookup tables, flip-flops, and interconnect network may all be scarce resources. For frequently-used functionality, FPGA vendors thus often package discrete silicon implementations into their devices. Such *hard cores* achieve much higher resource efficiency and better performance, *e.g.*, for on-chip memories (so-called *block RAM* or *BRAM*) or multiply/add functionality. Typical chips contain hundreds of small BRAM units, with an aggregate capacity of a few megabytes. This idea can go a far way: there are chips that include network controllers or full-fledged processors (*e.g.*, Xilinx' Zynq platform).

2.3 Strengths and Weaknesses

FPGAs provide the unique opportunity to build tailor-made hardware, efficient and at low cost. Circuits can be dedicated to an application (such as a database engine), to a typical workload, or even to a specific query; and those circuits can be re-programmed at any time if needed. Often, the massive parallelism offered by FPGAs (every lookup table can operate independent of any other) can lead to significant application speed-ups.

Obviously, these advantages come at a cost. A circuit instantiated on an FPGA will be slower by factors when compared to the identical circuit realized in pure silicon; at the same time, the latter will be about fourteen times more energy efficient [6]. To make up for this up-front overhead, FPGA-based solutions *must* use highly specialized hardware and exploit the available parallelism as far as possible. However, once such an implementation is found, the lower FPGA clock speed may lead to significant savings in energy consumption (*e.g.*, [20]).

FPGAs may further excel with their possibilities for *system integration*. Chips may be installed, *e.g.*, close to I/O devices such as network, disk, or memory. Such configurations can address data-intensive (sub-)tasks and filter or pre-process data directly at its source. Latency-sensitive tasks may benefit from the avoidance of an expensive round-trip to a general-purpose CPU [12].

Finally, FPGAs are best when the same and simple task must be performed over a large set of input data. Complex application tasks lead to larger chip area consumption, so less of the available parallelism can be exploited. A good fit are streaming-type applications that can process large amounts of data "on the wire."

3 Challenges

FPGA-backed data processors can achieve spectacular performance—if the developer manages to leverage the key advantages of the devices without hitting any of its limitations. After having worked in the field for a while, three aspects turned out to be the hardest nuts to crack.

Massive Parallelism. Efficient support for parallelism is among the key challenges in computer science as a whole. In the context of FPGAs, however, the problem gets exacerbated by the massive scale of the available hardware parallelism. FPGA designs can easily use thousands of parallel processing units, whereas many algorithms for CPUs begin to struggle when parallelism goes beyond four or eight independent units [4].

In this article, we will not discuss parallelization schemes for FPGAs in detail. But we refer to [18] for examples in the database domain.

Communication. An often-overlooked effect in modern computing systems is the cost of communication. Increased parallelism naturally goes with communication between the parallel units, and *bandwidth*, *latency*, or *locality* of communication become an issue.

In FPGAs, this surfaces not only because of their high degree of parallelism. Rather, *signal propagation delays* as a physical effect are usually the limiting factor for the clock frequency of the device. This emphasizes locality effects. Only circuits that have been designed with locality in mind will scale to large problem sizes.

It is interesting to note that locality effects have strong similarities with *NUMA (non-uniform memory access)* effects in multi-core systems. FPGA designs may thus be a good blueprint for large "many-core" systems that we may soon see on the market.

Abstractions. Decades of research have produced a large number of tools and abstractions that are the basis for any realistic software project today. For the development of FPGA/CPU co-designed systems, by contrast, such support is virtually non-existent. Suitable abstractions are heavily sought for at all levels of the programming stack: *(a)* good *design tools* (in particular, compilers for *high-level language synthesis*) are needed to close the gap between hardware and software developers; *(b)* more *libraries* and *drivers* will allow developers to focus on their actual application problems; finally, *(c)* massive degrees of parallelism will require new *abstractions for parallel programming*.

4 XML Filtering with FPGAs

The processing of XML data streams is an application domain where several groups suggested the use of FPGA technology for acceleration. For instance, Mitra *et al.* [10] and Moussalli *et al.* [11] used FPGAs to implement state machines in hardware and match XPath-based subscriptions in an XML publish/subscribe engine. Their use case assumes a large set of subscriptions (several thousand queries) and small input documents. The FPGA consumes input documents and emits, for each document, a *bitvector* that indicates for each subscription whether it matched or not. That bitvector is consumed by a software-based system, resulting in an FPGA/CPU co-designed setup.

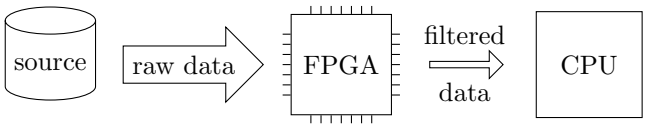

Figure 3: Data path architecture. The FPGA filters (or aggregates) the input to shield bandwidth off the software side.

Clearly, in such designs the FPGA will quickly become the I/O bottleneck of the system, particularly when the subscription count is scaled up to the values reported in earlier articles. In this case, the FPGA will actually *amplify* the original data volume and make the problem even more bandwidth bound.

4.1 Data Path Architecture

FPGAs seem much better suited for tasks where they can sustainably and significantly reduce the data volume that would otherwise hit the software system directly. A very effective means to do so is a *data path architecture* as illustrated in Figure 3. Thereby, the FPGA filters (aggregates, summarizes, ...) a high-volume input stream and leaves only a small fraction to the bandwidth-critical software part of the system.

In the XML domain, a good candidate for such behavior is *XML projection* [9] in FPGA hardware. Thereby, XML nodes are stripped off an XML data stream if they can be shown to not affect the outcome of a given XQuery expression. For actual queries, about 97 % of all input data can be discarded from the stream this way [5].

4.2 Expressiveness ↔ Speed

Integration also involves important trade-offs between flexibility (*i.e.*, how large a class of filter tasks can the accelerator support?) and performance.

Most existing systems chose to focus on only one side of the trade-off. *Glacier* compiles a large class of SQL queries into equivalent hardware circuits [12]. The price of *Glacier*'s very high flexibility is a very high circuit re-compilation cost. Every incoming query may result in minutes or hours of compilation time. Other systems [10, 11] have followed a similar route and require similar compilation costs.

Systems like Netezza [13] go for the other extreme. Their FPGA-based filter supports only a very restricted set of filter criteria, expressible through *parameters* exchanged by the hard- and software sides. Changes in the workload can then be accommodated in negligible time.

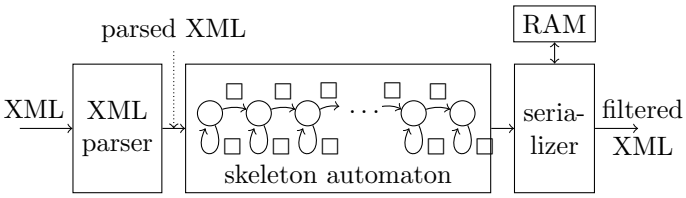

Figure 4: FPGA-accelerated XML filtering. After *parsing*, the XML stream passes through a *skeleton automaton*, which controls what the *serializer* emits as the projection result.

4.3 Skeleton Automata

Only few systems have attempted to strike a balance between the two ends, such as the *fpga-ToPSS* system [16]. For the XML projection problem, we presented a solution that provides expressiveness *and* fast workload changes [17]. It is based on *skeleton automata*, a novel mechanism to realize automata on FPGAs.

The idea is illustrated in Figure 4. The heart of the filtering engine is a finite-state automaton. Rather than re-compiling that automaton from scratch for every workload (or workload change), the FPGA device is prepared with a set of uncommitted automaton fragments, forming a *skeleton automaton*. In the skeleton automaton, transition conditions remain undefined (□ in Figure 4). The skeleton automaton becomes a functional automaton by loading a *parameter file* with transition conditions, depending on the individual user workload.

Skeleton automata separate concerns in an elegant way. Circuit compilation is expensive because of the structural aspects of finite-state automata. Many applications, including XML processing, do not exhibit high structural diversity. With skeleton automata, such applications can push expensive compilation parts to an off-line phase. Workload- or query-specific aspects merely affect configuration parameters, that can be determined and set in the hardware in negligible time. These parameters may still modify the pre-compiled automaton structure, *e.g.*, by setting transition conditions to false.

4.4 Hybrid XML Processing

We realized the skeleton automaton idea as part of our *XLynx* system. Plugged in-between the network path from an XML data source and an off-the-shelf XQuery processor, *XLynx* pre-filters the XML data stream on the wire. Effectively, a substantial part of the query processing workload is off-loaded to the FPGA-based accelerator. Figure 5 illustrates this for the first ten queries of the XMark benchmark. The figure shows improvements in parsing time, query execution time, and consumed main memory of the XQuery processor Saxon.

XML projection has been proposed as a software-based mechanism before [9]. Such functionality can optionally be enabled in the commercial version of Saxon, resulting in speedups/memory savings as illustrated using

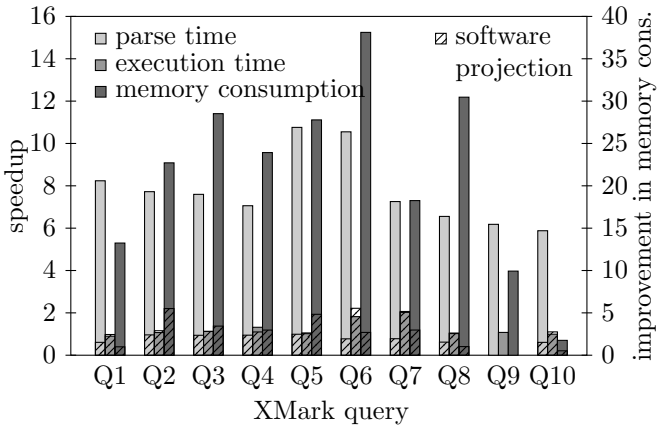

Figure 5: Speedup and improvement in memory consumption due to XML projection for the first ten XMark queries. Software projection for Q9 failed.

shading ▨ in Figure 5. A software-based projection strategy still has to parse the full XML input. Unfortunately, most XML applications are dominated by their high parsing overhead [14], which explains why software-based projection yields hardly any benefit in Figure 5.

5 Abstractions

Many existing FPGA solutions assume—at least to a significant extent—that an FPGA circuit is generated from scratch for the particular problem at hand (*e.g.*, XML pre-filtering). Given the tedious programming effort for FPGA solutions, such a methodology is not going to be practical for wide-spread FPGA use. Rather, *abstractions* are needed that ease the development task, much like standard data structures or boilerplate algorithms are the basis for any software-only project.

Unfortunately, usual data structures or algorithms do not carry over to FPGA solutions. This is because important parameters (*e.g.*, the degree of parallelism) are significantly different in the hardware world. But even more importantly, FPGAs completely blur the separation of algorithms and data structures. Rather, computation can be wrapped right into data structures, and data elements can be directly weaved into processing circuits.

5.1 Shifter Lists

As part of the *Avalanche* project, we made a first step at devising meaningful abstractions that can help implementors to solve data-intensive tasks on FPGA hardware. The outcome are *shifter lists*, a novel combination of data structure and massively parallel computation.

Figure 6 illustrates a typical application pattern for a shifter list. A (possibly large) input data set is scanned item-by-item. For each input item, a *working set* is consulted and possibly modified. The output will be a

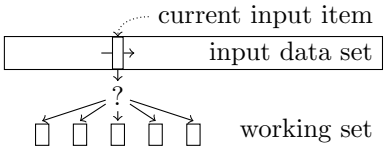

Figure 6: Application pattern for a shifter list: For each input item, the working set is accessed and possibly modified.

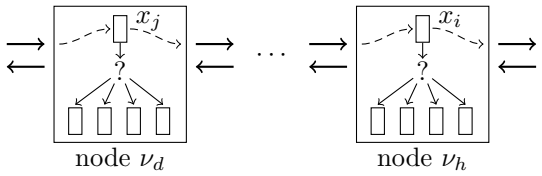

Figure 7: Shifter lists group working set items onto nodes ν. Input items x are forwarded from node to node and applied to the local working set part.

modified copy of the input stream or information collected within the working set while the input data has been consumed. Many stream processing tasks match this pattern, including the XML use case above.

5.1.1 Shifter Lists are a Data Structure

One way of looking at shifter lists is how they represent the working set and the input data set, which resembles a linked list where each list node ν holds a share of the working set and one "current" working set item x. Figure 7 illustrates this view.

During operation, input items x are forwarded from node to node. In effect, each input item x is guaranteed to "see" every item of the overall working set. In addition, shifter lists give a guarantee for causality. As illustrated in Figure 7, input items travel through the shifter list one after another. As a consequence, the later input item x_j is guaranteed to see any effect caused by the earlier input item x_i. Conversely, x_i is not going to see any effect caused by x_j. As we detailed in [19], this enables to re-formulate existing (sequential) algorithms using shifter lists.

5.1.2 Shifter Lists are a Processing Model

To enable automatic parallelization, shifter lists impose a particular processing model on the algorithmic part, which we illustrated in Figure 8.

The algorithm alternates between two phases, dubbed *eval* and *shift*. The former phase usually corresponds to

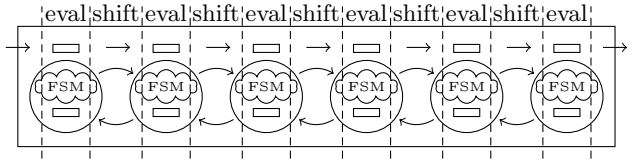

Figure 8: Two-phase processing in parallel BNL. Working set items (*i.e.*, tuples with several dimensions) are distributed over a pipeline of shifter list nodes.

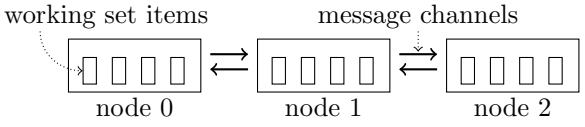

Figure 9: Shifter lists result in strict neighbor-to-neighbor communication, which can be implemented efficiently on modern hardware platforms (*e.g.*, FPGAs, NUMA architectures).

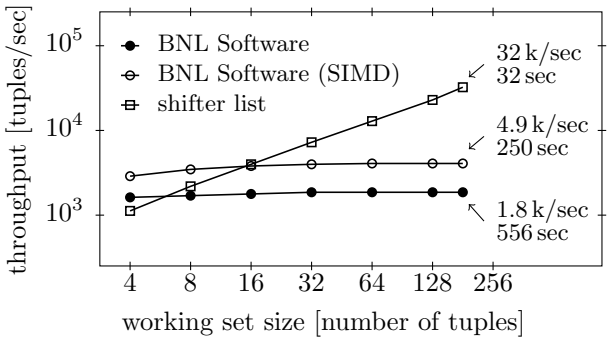

Figure 10: Shifter list-based implementation for skyline problem (anti-correlated data set with 7 dimensions).

the sequential implementation of the algorithm in question. During the *shift* phase, neighboring nodes in the shifter list exchange working set data (indicated using arrows ⇄) and forward input data along the chain of shifter list nodes (illustrated along the top).

To solve a particular application problem using shifter lists, the implementor must provide *eval* and *shift* functions. This remotely resembles the *map* and *reduce* functions that must be provided to a MapReduce framework. Contrast to MapReduce, however, shifter lists allow intuitive solutions to problems that may not run efficiently on MapReduce's rigid data parallelization scheme.

5.1.3 Shifter Lists are Massively Parallel

The *eval*/*shift* processing model results in a restricted *communication pattern*, where only immediate neighbors exchange working set data or input items. Logically, the shifter list forms a chain of processing elements, connected by pairs of FIFO message channels (Figure 9).

The structure fits particularly well to the constraints and capabilities of real hardware. FIFO queues have very little synchronization overhead. But more importantly, even large shifter list instances with large node counts do not require longer-distance messaging. FPGA hardware, but also NUMA architectures, favor communication patterns that exhibit a strong *locality*—a characteristic naturally given by the shifter list structure.

5.2 Shifter Lists in Practice

The *skyline problem* [2] demonstrates the potential of shifter lists. In [19], we detailed a solution to the well-known problem that uses the shifter list abstraction and FPGAs as an implementation platform.

Figure 10 illustrates the performance of the shifter list-based implementation, compared to the usual BNL algorithm and the recent *VSkyline* proposal of Cho *et al.* [3]. The experiments assume a Xeon 2.26 GHz CPU and a Xilinx XUPV5 development board, respectively. The shifter list implementation can exploit larger working set sizes to distribute processing over parallel units inside the FPGA. As can be seen in the figure, this results in a significant advantage over existing solutions.

6 Related Work

In 2008, we ramped up the *Avalanche* project, a pioneering effort on FPGA-accelerated database processing. *Glacier* [12] is an SQL-to-hardware compiler, which provides ultra-low latencies that cannot be reached with commodity components. *Ibex* [20] is an FPGA-based storage engine which can accelerate MySQL databases by pre-processing data along the SSD-to-CPU path.

The work of Sadoghi *et al.* [15, 16] also targets financial trading applications with high throughput/low latency demands. Their results also emphasize the importance of the expressiveness $\leftrightarrow$ speed trade-off (Section 4).

FPGAs have been used very successfully in application domains where data streams with massive volumes appear far out of reach for commodity hardware. A good poster child for this is the *LHCb* experiment at CERN, where more than 1 TB/s of sensor data must be pre-filtered in real time to search for rare decay events [7].

Within the database community, FPGAs have become popular also because of the success of Netezza (now an IBM company, [13]). In the product, FPGAs pre-filter data during scans. The usage pattern fits the "simple task on a large set of input data" scheme and can leverage the I/O and system integration opportunities.

Powerful IP cores for specific application needs (*e.g.*, those of Lockwood [8]) and the use of *partial reconfiguration* (runtime chip re-configuration with help of pre-built circuit blocks; proposed, *e.g.*, by Ziener *et al.* [21]) complement our quest for powerful abstractions in order to increase developer productivity in the field.

7 Summary

There is no doubt that FPGAs are a highly interesting technology for data processing systems. In this article, we highlighted some of the challenges (and steps toward their solutions) that we think still need attention, before FPGA-accelerated data processing can go mainstream.

We focussed on two aspects. *System integration* can be seen as an opportunity for FPGAs, because the technology allows entirely new ways to integrate the hardware into the remaining system stack. We used *XML processing* as a poster child for our discussion. We also discussed *abstractions*, *e.g.*, to make the FPGA development process more efficient. We looked at *shifter lists* as a possible step toward new abstractions for FPGAs.

Literature

[1] S. Borkar and A. Chien. The future of microprocessors. *Comm. of the ACM*, 54(5):67–77, May 2011.

[2] S. Börzsönyi, D. Kossmann, and K. Stocker. The skyline operator. In *ICDE*, pages 421–430, Heidelberg, Germany, April 2001.

[3] S.-R. Cho et al. VSkyline: Vectorization for efficient skyline computation. *SIGMOD Record*, 39(2):19–26, June 2010.

[4] C. Cole and R. Williams. Photoshop scalability: Keeping it simple. *Comm. of the ACM*, 53(10):32–38, October 2010.

[5] P. Fischer and J. Teubner. MXQuery with hardware acceleration. In *ICDE*, Arlington, VA, USA, April 2012.

[6] I. Kuon and J. Rose. Measuring the gap between FPGAs and ASICs. *IEEE TCAD*, 26(2), February 2007.

[7] J. Laubser, P. Perret, H. Chanal, R. Cornat, and O. Deschamps. The level 0 trigger decision unit for the LHCb experiment. In *IEEE NPSS Real Time Conference*, pages 1–8, Batavia, IL, USA, May 2007.

[8] J. W. Lockwood and M. Monga. Implementing ultra-low-latency datacenter services with programmable logic. *IEEE Micro*, 36:18–26, July 2016.

[9] A. Marian and J. Siméon. Projecting XML documents. In *VLDB*, Berlin, Germany, September 2003.

[10] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML filtering through a scalable FPGA-based architecture. In *CIDR*, Asilomar, CA, USA, January 2009.

[11] R. Moussalli, M. Salloum, W. A. Najjar, and V. J. Tsotras. Massively parallel XML twig filtering using dynamic programming on FPGAs. In *ICDE*, pages 948–959, Hannover, Germany, April 2011.

[12] R. Mueller, J. Teubner, and G. Alonso. Streams on wires—a query compiler for FPGAs. *Proc. of the VLDB Endowment*, 2(1):229–240, August 2009.

[13] Netezza, 2012. `http://www.netezza.com/`.

[14] M. Nicola and J. John. XML parsing: A threat to database performance. In *CIKM*, pages 175–178, New Orleans, LA, USA, November 2003.

[15] M. Sadoghi, H.-A. Jacobsen, M. Labrecque, W. Shum, and H. Singh. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. of the VLDB Endowment*, 3(2):1525–1528, September 2010.

[16] M. Sadoghi, H. Singh, and H.-A. Jacobsen. Towards highly parallel event processing through reconfigurable hardware. In *DaMoN*, Athens, Greece, June 2011.

[17] J. Teubner, L. Woods, and C. Nie. XLynx—an FPGA-based XML filter for hybrid XQuery processing. *ACM TODS*, 38(4), December 2013.

[18] Jens Teubner, Rene Mueller, and Gustavo Alonso. Frequent item computation on a chip. *IEEE TKDE*, 23(8):1169–1181, August 2011.

[19] L. Woods, G. Alonso, and J. Teubner. Parallelizing data processing on FPGAs with shifter lists. *ACM TRETS*, 8(2), March 2015.

[20] L. Woods, J. Teubner, and G. Alonso. Less watts, more performance: An intelligent storage engine for data appliances. In *SIGMOD*, New York, NY, USA, June 2013.

[21] D. Ziener, F. Bauer, A. Becher, C. Dennl, K. Meyer-Wegener, U. Schürfeld, J. Teich, J.-S. Vogt, and H. Weber. Fpga-based dynamically reconfigurable SQL query processing. *ACM TRETS*, 9(4):25, 2016.