XLynx—An FPGA-based XML Filter for Hybrid XQuery Processing

JENS TEUBNER, TU Dortmund University, Dept. of Computer Science, DBIS Group
LOUIS WOODS, ETH Zurich, Dept. of Computer Science, Systems Group
CHONGLING NIE, ETH Zurich, Dept. of Computer Science, Systems Group

While offering unique performance and energy saving advantages, the use of *field-programmable gate arrays (FPGAs)* for database acceleration has demanded major concessions from system designers. Either the programmable chips have been used for very basic application tasks (such as implementing a rigid class of selection predicates), or their circuit definition had to be completely re-compiled at runtime—a very CPU-intensive and time-consuming effort.

This work eliminates the need for such concessions. As part of our *XLynx* implementation—an FPGA-based XML filter—we present *skeleton automata*, which is a design principle for data-intensive hardware circuits that offers high expressiveness and quick re-configuration at the same time. Skeleton automata provide a generic implementation for a class of *finite-state automata*. They can be parameterized to any particular automaton instance in a matter of micro-seconds or less (as opposed to minutes or hours for complete re-compilation).

We showcase skeleton automata based on *XML projection* [Marian and Siméon 2003], a filtering technique that illustrates the feasibility of our strategy for a real-world and challenging task. By performing XML projection in hardware and filtering data *in the network*, we report on performance improvements of several factors while remaining non-intrusive to the back-end XML processor (we evaluate *XLynx* using the Saxon engine).

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

General Terms: Design, Performance

Additional Key Words and Phrases: FPGA, XML, XQuery, Projection, Skeleton Automaton

ACM Reference Format:
Teubner, J., Woods, L., and Nie, C. 2013. *XLynx*—An FPGA-based XML Filter for Hybrid XQuery Processing. *ACM Trans. Datab. Syst.* 38, 4, Article XX (December 2013), 40 pages.
DOI: http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Field-programmable gate arrays (FPGAs)—and hardware-accelerated database processing in general—have gained a lot of momentum in past years. The opportunity to implement tailor-made functionality directly in hardware is a very promising research and development direction to overcome the inherent limitations of commodity hard-

This work is supported by the Swiss National Science Foundation (SNSF) under *Ambizione* grant number 126405/144505, by the Enterprise Computing Center of ETH Zurich (http://www.ecc.ethz.ch/), and by the DFG, Collaborative Research Center SFB 876. Parts of this work have been done while Jens Teubner was at ETH Zurich.
Authors' address: TU Dortmund University; Dept. of Computer Science; Databases & Information Systems Group; Otto-Hahn-Strasse 20; 44227 Dortmund; Germany; jens.teubner@cs.tu-dortmund.de; louis.woods@inf.ethz.ch.

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

ware. As recent research has shown, FPGAs can perform database tasks with higher throughput, lower latency, and lower energy consumption than pure software systems (*e.g.*, [Moussalli et al. 2011; Mueller et al. 2009; Netezza 2012; Sadoghi et al. 2010; Sidhu and Prasanna 2001; Woods et al. 2010]).

Unfortunately, previous FPGA-based solutions often had a tendency to be rather inflexible. Designing good hardware circuits is tedious and difficult, and hardware-based solutions can only excel when the respective circuit is carefully tuned to the problem at hand. In previous work, this has forced system designers to face critical trade-offs:

(a) Systems like IBM's Netezza [Netezza 2012] use FPGAs only for a small part of the system's overall functionality. Selection and projection are implemented in the Netezza system by means of a statically compiled circuit that is *parameterized* at runtime within very tight bounds (*e.g.*, only basic "SARGable"[1] selection predicates).

(b) Several research prototypes (including [Moussalli et al. 2011; Mueller et al. 2009; Sadoghi et al. 2010; Woods et al. 2010]) instead opted to compile a *dedicated circuit* for each user query. Such an approach provides expressiveness, however, at a very high cost. Re-compiling a hardware circuit for each user query is a complex and CPU-intensive task, with typical compilation times ranging from minutes to even hours. Such a per-query startup cost seems only bearable for a very narrow set of applications (such as algorithmic trading [Sadoghi et al. 2010] or network intrusion detection [Sidhu and Prasanna 2001]).

In this work, we do *not* want to trade speed for expressiveness. Rather, we show a system design strategy that offers high expressiveness (sufficient to support an important subset of *XPath*), and yet does not require expensive re-compilation at runtime. Our system, *XLynx*, offers the same throughput characteristics as previous approaches that required per-query compilation. By contrast, however, *XLynx* also supports instant query workload changes with reconfiguration times in the micro-second range.

The heart of *XLynx*—and a core contribution of this work—is the *skeleton automaton* design pattern. A skeleton automaton is a generic implementation of a *nondeterministic finite-state automaton (NFA)* that can be tailored to implement a particular automaton instance with only few (and fast-to-realize) configuration changes. Skeleton automata are made possible by *separating* the *structure* of a finite-state automata—which is the difficult part to (re-)compile on-line—from its *semantics*, *e.g.*, number of states, transition conditions, etc. This allows us to perform all structure-related compilation steps off-line and only once, while at runtime we only modify configuration parameters.

This article describes the inner workings of *XLynx*, as well as the skeleton automaton design principle. We give sufficient details for readers to follow and reproduce all important building blocks of *XLynx*, and non-FPGA experts are provided with background information on hardware/FPGA technology.

We use the skeleton automaton concept to implement *XML projection*, a task that is meaningful and at the same time challenging from a hardware perspective. XML projection was proposed by Marian and Siméon [2003] almost ten years ago. But because *XML parsing* is the dominating cost factor in real-world systems [Nicola and John 2003], XML projection remained little more than an academic curiosity. By offloading the projection task to dedicated hardware, however, *XLynx* can truly unleash the potential of XML pre-filtering, leading to query speedups of several factors.

[1] http://en.wikipedia.org/wiki/Sargable

Parts of this work have been published earlier as a conference paper [Teubner et al. 2012]. This article emphasizes how a *complete system* can be built from the skeleton automaton principle. To this end, we discuss integration aspects that will be needed to leverage skeleton automata in a full system design. This includes a more in-depth discussion of *XML parsing*—a task which alone has challenged hard- and software makers for a long time [Leventhal and Lemoine 2009; Dai et al. 2010]—; *XML serialization* to interface with a software-based back-end; and *runtime reconfiguration*. New sections on *runtime query removal* and *on-line defragmentation* offer the dynamism necessary for real-world use. All these parts are carefully engineered to operate in concert. The article also includes a significantly extended experimental evaluation of *XLynx*, which demonstrates the advantages with respect to system integration, performance, and energy consumption.

We present *XLynx* in the following order. Section 2 refreshes the relevant parts of the XML projection concept. Section 3 gives a quick hardware background for non-expert readers, with a focus on the implementation of finite-state automata. Skeleton automata are introduced in Section 4, complemented by runtime configuration and automatic defragmentation in Sections 5 and 6, respectively. Section 7 gives hints on the low-level optimization of skeleton automata, before we evaluate our work in Section 8, discuss related work in Section 9, and wrap up in Section 10.

2. XML PROJECTION

Our work provides a hardware implementation for XML projection. To understand the idea of XML projection, consider the following query, which is based on XMark [Schmidt et al. 2002] data (XMark models an auction website):

```
for $i in //regions//item
  return <item>
           { $i/name }
           <num-categories>                                    (Q1)
             { count ($i/incategory) }
           </num-categories>
         </item>
```

This query looks up all auction items and prints their name together with the number of categories they appear in.

2.1. Projection Paths

Out of a potentially large XMark instance, Query Q_1 will need to touch only a small fraction that has to do with items and their categories. What is more, this fraction can be described using a set of very simple *projection paths*:

```
{ //regions//item,
  //regions//item/name #,
  //regions//item/incategory } .
```

Only nodes that match any of the paths in this set are needed to evaluate Query Q_1; all other pieces of the input document can safely be discarded without affecting the query outcome.

Since our aim is to reduce data volumes, by default we keep only the matching node itself in the projected document, but discard any descendant nodes that do not match any projection path as well. Whenever the query demands to keep the entire subtree below some matched path, we annotate this path explicitly with a trailing # symbol (consistent with the notation in [Marian and Siméon 2003]). In our example this is needed to include full `name` elements in the query result.

```
<site>
  <regions>
    ...
    <africa>
      ...
      <item id="item42">
        <name>vapour wept became empty </name>
        <incategory category="category3"/>
        <incategory category="category1"/>
      </item>
      ...
    </africa>
    ...
  </regions>
  ...
  <open_auctions>
    <open_auction id="open_auction0">
      ...
    </open_auction>
    ...
  </open_auctions>
  ...
</site>
```

Fig. 1. XML projection. Only the underlined parts are needed to evaluate Query Q_1.

projpath	::=	*path* #?
path	::=	`fn:root()` | *path*/*step*
step	::=	*axis* `::` *test*
axis	::=	`child` | `descendant` | `self` | `descendant-or-self`
test	::=	`*` | `text()` | `node()` | *NCName*

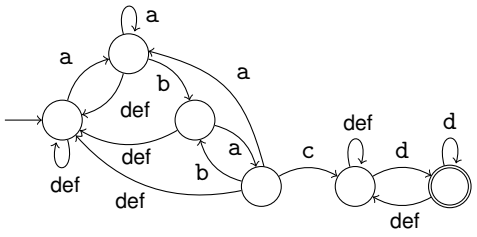
Fig. 2. Supported dialect for projection paths.

Figure 1 illustrates the process for an XMark excerpt. Only the underlined parts of the document are needed to evaluate Query Q_1. Everything else will be filtered out during XML projection.

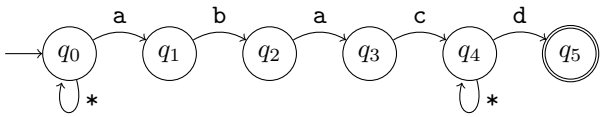
Path Inference and Supported XPath Dialect. Marian and Siméon describe a procedure to statically infer the set of projection paths for any given query Q. We adopt this procedure and refer to [Marian and Siméon 2003] for details. Several XQuery processors readily implement the inference procedure, including MXQuery [Botan et al. 2007] and Galax [Fernández et al. 2003]. The commercial version of Saxon, Saxon-EE, implements XML projection, too.

Paths emitted by the inference procedure adhere to a simple subset of the XPath language. Most importantly, the subset only permits downward navigation, *i.e.*, the `self`, `child`, `descendant`, and `descendant-or-self` axes.

Figure 2 lists the XPath dialect that our hardware implementation supports. This dialect essentially covers all features of the projection path language as proposed by Marian and Siméon [2003] (we do not support namespaces at this point, however). For illustration purposes, in this paper we frequently make use of the abbreviated notation in XPath, where, for example, '`//`' stands for '`/descendant-or-self::node()/`' (in our restricted dialect this is the same as '`/descendant::`').



(a) Deterministic finite-state automaton (DFA).



(b) Non-deterministic finite-state automaton (NFA).

Fig. 3. Finite-state automata (deterministic and non-deterministic variants) to implement query `fn:root()//a/b/a/c//d`.

2.2. Path Evaluation (Previous Work)

For evaluation, projection paths are often viewed as *regular expressions*, evaluated over each node's path starting from the root node. Thereby, the projection path/regular expression is compiled into a *finite-state automaton* that is driven by a SAX-style XML parser.

Finite-State Automata. Figure 3 illustrates this approach for the projection path `fn:root()//a/b/a/c//d`. This expression can be compiled into either a *deterministic* (Figure 3(a)) or a *non-deterministic* finite-state automaton (Figure 3(b)). Observe how, in the latter case, each $\circlearrowleft *$ corresponds to a `//` descendant step in the input query.

In deterministic finite-state automata, only a single state can be active at any given point in time. This significantly eases implementation in software (and requires only a single $\langle state, symbol\rangle \mapsto state$ lookup per input symbol). XFilter [Altınel and Franklin 2000], a publish/subscribe system for XML, is thus based on a set of deterministic automata, one for each registered query. Since XFilter is intended to support very large numbers of registered queries, a *query index* accelerates processing by only advancing those automata that may actually be affected by the current input symbol.

However, only non-deterministic finite-state automata exhibit the tight correspondence between automaton structure and query pattern. This makes them significantly easier to construct and maintain under workload changes. In YFilter [Diao et al. 2003], this allowed the use of a *single* non-deterministic finite-state automaton that simultaneously matches all registered input queries. Workload changes (*i.e.*, (un)registering queries) can be realized in YFilter by changing only local fragments of the whole automaton.

Backtracking. Either automaton type is to be evaluated on every root-to-node path. To this end, automata are advanced upon every seen *opening tag*. On *closing tags*, the system must *backtrack* to the originating automaton state. To implement this functionality, systems maintain a *stack* that holds a history of automata states. It is populated during the handling of opening tags and consumed when the corresponding closing tags are encountered.

Hardware Acceleration. Finite-state automata can be implemented very efficiently in hardware (more details later). In [Moussalli et al. 2010; 2011], this was used to im-

plement hardware-accelerated XML filtering. Essentially, their system compiles a set of path expressions into a YFilter-like NFA, which is then run on an FPGA. Similarly, in our own work [Woods et al. 2010] we used FPGAs to perform *complex event detection* based on regular expressions in hardware, again by generating a dedicated per-query circuit and reprogramming the FPGA to run it. As indicated before, both approaches incur a high compilation cost (of up to several hours) that has to be invested for every change of the query workload.

Conversely, BARTS [van Lunteren 2001] is an implementation technique for finite-state automata in hardware that can be updated at runtime (a use case is the ZUXA XML parsing engine [van Lunteren et al. 2004]). The key idea is an elegant encoding scheme for transition tables that can be stored and altered in on-chip memory. Unfortunately, the technique is bound to deterministic finite-state automata and queries cannot be (un)registered to/from a single deterministic finite-state automaton easily. The BARTS technique is used today in IBM's *wire-speed processor* [Franke et al. 2010] to implement XML parsing and accelerate network packet filtering.

The skeleton automaton technique that we describe in this work does not need to make compromises between expressiveness and workload re-configurability. To efficiently deal with (changing) XML projection workloads and high expressiveness, our system is based on non-deterministic finite-state automata, which support fast runtime (re)configuration enabled by our skeleton automata design technique.

3. SOME HARDWARE BACKGROUND

Before we delve into the inner workings of our prototype system *XLynx*, this section provides a very short introduction into FPGA technology and the implementation of finite-state automaton in hardware. Virtually any hardware circuit consists of the same three fundamental ingredients:

(i) *Combinational logic*, which is composed of basic logic gates ('AND', 'OR', etc.). Each (Boolean-valued) output $f_i(\bar{x})$ of a combinational circuit depends solely on its input signals x_j.
(ii) *Memory elements*, such as flip-flop registers, are 1-bit storage cells that allow a circuit to save and maintain state. For larger storage needs, circuits may further include dedicated *RAM*, which has a higher integration density and thus a lower cost but is less flexible.
(iii) A *wiring interconnect* combines logic and memories into a functional circuit.

The actual behavior of a circuit is determined by the Boolean functions f of its combinational parts and by the wiring between combinational logic and flip-flop registers.

In addition to the input data, most circuits depend on a *clock signal*, a periodically changing high/low signal, to *synchronize* all circuit components. The *speed* of a hardware circuit is determined by the clock frequency, but also by the amount of work that the circuit can perform within each clock cycle.

3.1. Field-Programmable Gate Arrays

Field-programmable gate arrays (FPGAs) are also considered "sea of gate" devices that provide a large amount of generic logic gates (so-called *lookup tables*) as well as flip-flop registers. An FPGA can be *programmed*[2] by defining *(a)* the logic function f for each lookup table and *(b)* the signal wiring in the on-chip *interconnect network*.

[2]FPGAs blur the distinction between "program" and "configuration." In this text, we "program" our chip once to determine the circuit it implements. When we only change parameters at runtime, we refer to this as "configuration."

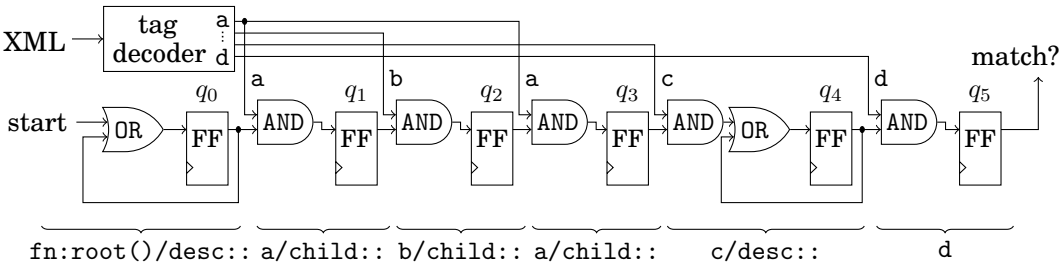


Fig. 4. Hardware implementation of the non-deterministic finite-state automaton in Figure 3(b).

Dedicated RAM is available on FPGAs in terms of so-called *Block RAM* (or *BRAM*). BRAM blocks can be allocated and integrated into a user circuit in chunks of a few kbits. For instance, the Xilinx XC5VLX110T FPGA chip we used for our experiments contains 296×18 kbit of BRAM.

In this work we do *not* actually exploit the reprogrammability of FPGAs. Rather, we compile and upload a generic circuit once, *i.e.*, we program the FPGA once. The query workload, including any workload changes, then only affects configuration parameters within this circuit. Economic aspects aside (tailor-made chips have substantial manufacturing costs), our system could be implemented equally well as an *application-specific integrated circuit (ASIC)*.

In fact, the given FPGA hardware imposes rather tight constraints on the available resources and their distribution on the chip. Managing these constraints adds to the challenge of building a hardware circuit. Kuon and Rose [2007] found that ASICs typically run more than three times faster than FPGAs, yet they dissipate only 1/14 of the power. Similar advantages could be expected from an ASIC implementation of our work.

3.2. Finite-State Automata in Hardware

Finite-state automata can be mapped mechanically to a corresponding (but hard-wired) hardware implementation, which after compilation can be uploaded onto an FPGA. Figure 4 illustrates this for the non-deterministic finite-state automaton that we saw earlier in Figure 3(b). For realistic automata, compiling and routing the respective circuit typically takes several minutes or even up to several hours. We observed such compilation times, *e.g.*, when generating finite-state automata for our work in [Woods et al. 2010].

In a circuit generated this way, every automaton state is represented by a flip-flop register (labeled 'FF' in Figure 4). Wires between flip-flops implement state transitions. An '`AND`' gate along these wires ensures that the transition is taken whenever the originating state is active *and* a matching input symbol is seen. $\circlearrowleft *$ transitions are not conditioned on the input symbol (thus, there is no '`AND`' gate along their path). Whenever multiple transitions can activate a state, these must be combined using an '`OR`' gate, as can be seen at the inputs to states q_0 and q_4.

The automaton is driven by a *tag decoder* that parses the XML input. Whenever it sees a tag named `a`, ..., `d`, it sets the corresponding output signal to '1'. The tag decoder itself is implemented as a finite-state automaton as well.

Not shown in Figure 4 is the clock circuitry that ensures that the automaton state is advanced on every clock tick. A stack data structure, needed to support the XML tree structure, can be attached to the finite-state automaton. Then, states q_0 through q_5 are pushed/popped to/from this stack during start/end element events. Refer to Moussalli et al. [2010; 2011] for details.

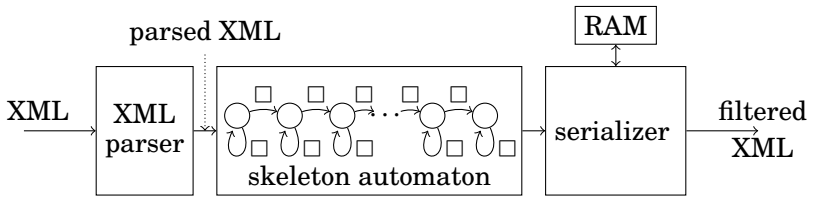


Fig. 5. XML projection engine. After *parsing*, the XML stream passes through a *skeleton automaton*, which controls what the *serializer* emits as the projection result.

Though slightly simplified, the explained procedure quite well describes the state of the art in hardware-based pattern matching. Optimized construction algorithms for FPGA targets exist (*e.g.*, those of Yang et al. [2008]) but their main concern is the consumption of on-chip resources. The immense routing effort is inherent to the concept and arises in any scheme that compiles automata from scratch.

4. DYNAMIC XML PROJECTION

Here we propose a new approach to automaton implementation on FPGAs that avoids the high cost of on-line automaton routing. We achieve this by *separating* the automaton structure from its semantics. The structural aspects of the automaton can then be compiled *off-line* into a *skeleton automaton*. At runtime, the skeleton only has to be *parameterized* to obtain a complete automaton for the particular query workload.

4.1. System Overview

The high-level structure of *XLynx* is illustrated in Figure 5. Raw XML data enters the system at the left end of the figure, where a *hardware XML parser* analyzes the syntactical structure of the stream. Enriched with parsing information, the XML stream passes through a series of *skeleton segments*—which together form a *skeleton automaton*—that performs the actual path matching. Finally, the *serializer* at the right end of the figure copies matches to the circuit output and ensures a well-formed XML result. We detail the inner workings of each building block in the following.

The *XLynx* design exploits an important characteristic of non-deterministic finite-state automata that are built from projection paths: each such automaton will always have a strictly linear structure, only interspersed with $\circlearrowleft *$ transitions for each `descendant` step in the path. Every *segment* (marked at the bottom of Figure 4) of the linear automaton corresponds to one part of the path expression that is evaluated.

The chain of skeleton segments in our system realizes this structure in a generic fashion, whereby skeleton segments can be runtime-(re)configured to include a $\circlearrowleft *$ loop or not.

4.2. XML Parsing

The input XML byte stream enters our system on the left side of Figure 5 and is fed into the hardware XML parser. As mentioned before, parsing is in itself a major throughput challenge for many XML processing systems [Leventhal and Lemoine 2009], but it is a prerequisite to perform effective XML projection. Only recently, Dai et al. [2010] were the first to report on a hardware XML parser that could sustain a 1 Gb/s Ethernet line rate.

Parsing can be done very efficiently in hardware if the language to recognize is regular. The language can then be implemented as a finite-state automaton, which matches the capabilities of electronic circuits well. Fortunately, XML is “almost regular”: only the proper nesting of element tags and the test for well-formedness (tag names in start and end tags must match) cannot be expressed using regular patterns. XML parsing

```
PITarget      = Name;                                                  # 17 *
PI            = '<?' PITarget (S (Char* - (Char* '?>' Char*)))?        # 16
                '?>' @pi_or_comment_end;

Comment       = '<!--' ((Char - '-') | ('-' (Char - '-')))*            # 15
                '-->' @pi_or_comment_end;

AttValue      = ( '"' ([^<&"] | Reference)* '"' )                      # 10
                | ("'" ([^<&'] | Reference)* "'" );
Attribute     = Name Eq AttValue;                                      # 41

STag          = '<' >tag_start Name @tag_name (S Attribute)* S?        # 40
                '>' >opening_tag_end;
ETag          = '<' >tag_start '/' >closing_tag_start                  # 42
                Name @tag_name '>' >closing_tag_end;

EmptyElemTag  = '<' >tag_start Name @tag_name (S Attribute)* S?        # 44
                '/' @empty_tag_slash '>' @empty_tag_end;

content       = ( PI | ConfPI | CharData | EmptyElemTag | STag | ETag
                  | Comment )*;
```

Fig. 6. Excerpt of actual *XLynx* source code: XML grammar specification (*Snowfall* input file). Comments on the right refer to XML grammar production rules in the W3C XML Recommendation [Bray et al. 2006]. Action code invocations are *italicized*.

becomes expressible as a finite-state automaton once we take such features out of the language specification (in *XLynx* they are handled outside the main parser logic).

The flip side is that the resulting automaton is potentially huge. Writing and maintaining a state automaton with hundreds of states in plain VHDL code is close to impossible. This is why we developed *Snowfall* [Teubner and Woods 2011], a *parser generator tool* that companions the development of *XLynx*. With help of *Snowfall*, parsers for real-world languages can be written and maintained efficiently.

Snowfall. Snowfall takes as input a *grammar specification* of the input language. The specification typically contains *action code annotations* to call user-defined VHDL routines whenever a particular (sub)pattern has been matched (this is similar in spirit to the `lex`/`yacc` tools in the software world). Figure 6 shows an excerpt of the actual *XLynx* source code. Large parts of the W3C XML Recommendation [Bray et al. 2006] can be copied literally into the input of the *Snowfall* parser generator, with only action code annotations added (numbers in comments on the right refer to production rules in [Bray et al. 2006]).

Internally, *Snowfall* converts the regular grammar into a finite-state automaton, then implements/emits this automaton as VHDL code. The complexity of such automata had severely limited the language support in previous work on FPGA-based XML filtering. For instance, the parser of Moussalli et al. [2011] can only accept XML tags with a length of two characters, and their paper leaves open which additional limitations their parser imposes. *Snowfall* allows us to include a full-fledged XML parser (with the exception of validation and namespace support). In addition, we used *Snowfall*'s high-level notation to recognize *configuration commands* directly in the parser, which we will discuss later in Section 5.

Parser Output. The output of our hardware XML parser is an *annotation* to the input XML data stream. A token field makes the lexical structure of the stream accessible to

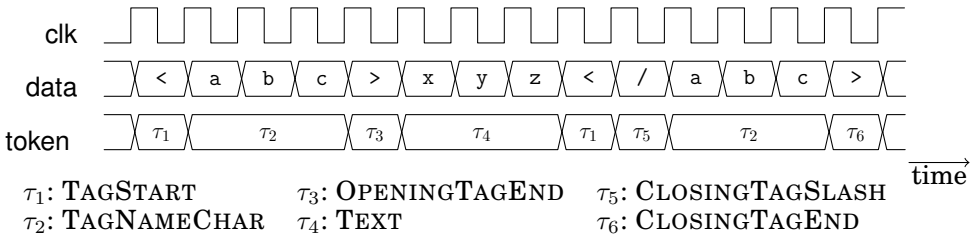


Fig. 7. Timing diagram of XML parser output. The XML stream is enriched with a token signal to make lexical information explicit.

subsequent processing units. We refer to an XML stream with token annotations as a *parsed* XML stream.

The behavior of the XML parser component is illustrated in Figure 7 as a *timing diagram*. The token signal carries values of an enumeration type, whose symbolic names we listed at the bottom of the figure. The main purpose of the XML parser component is to centralize the parsing task into a single hardware unit. This greatly simplifies the overall circuit design and reduces the size and complexity of the remaining hardware components.

To the parsed XML stream, the configured automaton adds a match flag to identify matching pieces in the data stream. This flag is interpreted by the serializer to produce the projected XML document.

4.3. Skeleton Automaton

Compiling individual automata into FPGA circuits is expensive because the placement and routing of states and transitions on the two-dimensional chip space is a highly compute-intensive task. Once the structure of an automaton and its placement on the chip is known, however, workload adaptations that only affect transition conditions can be realized with negligible effort.

Here we exploit this characteristic and build a generic *skeleton automaton*. The skeleton is provisioned for any transition and condition that would be permitted by the respective query language (in our case a dialect of XPath). Placeholders in the skeleton automaton (we illustrate them as □) are filled with parameter values at runtime to enable or disable (by putting a false condition on the edge) transitions or to reflect query-dependent conditions.

4.3.1. Skeleton Segments. In the case of XPath, we build the skeleton automaton from a large number of *segments*. Each segment consists of a single state and two parameterized conditions as shown here on the right (Figure 8). The actual implementation contains additional parameters that determine whether a state is accepting or handle specifics of XPath (such as `self` axis). For ease of presentation, we omit such parameters from the discussion here and in the following.

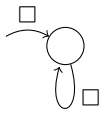


Fig. 8. Skeleton segment.

Skeleton segments are connected to form a chain much as we sketched it already in Figure 5. Observe how this structure coincides with the one that we saw earlier for our example query (Figure 3(b)). In fact, skeleton segments are sufficient as basic building blocks to construct a finite-state automaton for any legal XML projection path.

To support backtracking, each segment also includes a *history* stack (also not shown in the illustration), so backtracking is wrapped into the basic skeleton building blocks and scales trivially with the overall automaton size.

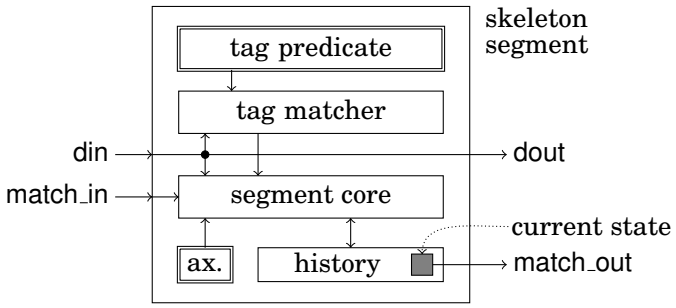


Fig. 9. Hardware implementation of a single skeleton segment. ▢ blocks hold configuration parameters (axis and node test).

4.3.2. Compiling Queries. Compiling a projection path into a set of segment parameters is particularly simple. Each step in the path is mapped to one segment in the skeleton automaton. Much as we saw in the example in Figure 3(b), each *node test* is set as a transition condition on a segment-to-segment edge. *Axes* (`child` or `descendant`) result in conditions false or $*$ annotated to a back loop $\circlearrowleft\square$ (we discuss `-self` variants later). Somewhat counterintuitive to the notion of XPath location steps, each skeleton segment corresponds to one '$nodetest/axis::$' pair (not '$/axis::nodetest$'), as we already indicated earlier on the bottom of Figure 4.

4.3.3. Implementing a Skeleton Automaton. Skeleton segments are the basic building blocks of our matching engine. Finding a proper hardware implementation for them is what now remains to realize scalable and efficient XML projection in hardware.

As illustrated in Figure 9, each segment consists of three sub-components (segment core, tag matcher, and history unit) that interpret the two query parameters *axis* and *tag predicate*. The two signals match_in and match_out represent the in- and outgoing transition edges of the segment, the din signal gives the circuit access to the input data stream (segments are daisy-chained so all segments have access to the stream).

The *segment core* is what ultimately implements the automaton segment. Based on the setting of the axis parameter, it will enable the respective logic gates to allow $\circlearrowleft*$ loops in the effective automaton.

As in the traditional scheme, the actual automaton state, which is part of each segment, is implemented using a flip-flop register. In Figure 9, this register is illustrated as a gray box ■. To support backtracking, the flip-flop is embedded inside a *history* unit, which replaces the global stack of previous hard- or software-based XPath engines.

In hardware, the history unit is implemented using a *shift register* whose contents can be shifted left/right as the parser moves down/up in the XML tree structure (*e.g.*, upon opening and closing tag events). The rightmost bit of this shift register corresponds to the current state and is propagated to the outside in terms of the match_out signal. In the software world, the history unit would best compare to a *stack* for single-bit values, where the stack top determines the match_out signal.

The size of the history unit is a compile-time parameter that limits the XML tree depth up to which matches can be tracked (default is 16 in our implementation). Cases where this depth is exceeded by a given XML instance will still not fail. XML projection is, by definition, a best-effort strategy to reduce input sizes prior to the actual query processing. If the hard limit for history tracking is reached, we can always pass those parts on to the software side and handle them there.

In contrast with the traditional compile-by-query scheme, our circuit does not use an external tag decoder. Instead, dedicated sub-circuits ('tag matcher') in each segment

ALGORITHM 1: Pseudo code for segment core.

```
1  switch din.token do
2      case OPENINGTAGEND
3          if (tag matches and match_in)
4              or (axis = desc and history[last]) then
5              match := true;
6          else
7              match := false;
8          push (history, match);
9      case CLOSINGTAGEND
10         pop (history);
```

provide information about matched tag names. We will detail those sub-circuits in a moment.

Algorithm 1 summarizes in pseudo code the behavior of a segment core.[3] Matching occurs when an opening XML tag is fully consumed. Lines 3–7 then combine the *axis* parameter, tag match information, the input match flag, and (to implement $\circlearrowleft *$ loops) the existing match state to determine a new match state. This new match state is then pushed/shifted into the history shift register (line 8), which implicitly makes the information also available on the match_out port. The match state is restored from the history shift register when a closing tag is consumed (lines 9–10).

The pseudo code in Algorithm 1 can straightforwardly be translated into a VHDL circuit description. Note that in hardware this code is *not* executed as sequential code. Rather, the code is compiled into combinational logic that drives the control signals of the hardware shift register.

4.3.4. Distributed Tag Decoding. Input to the segment core is a signal indicating whether an element with corresponding *tag name* was seen in the input. The classical approach to this sub-problem was shown in Figure 4. There, a dedicated *tag decoder* was compiled along with the main NFA. It included a hard-wired set of tag names, and produced a separate output signal for each tag name in the set. These signals were wired to segments in the NFA as needed (top part of Figure 4). Some earlier accelerators for XML filtering support tag decoding only in a very restricted form (*e.g.*, [Moussalli et al. 2011]) or push it to the software side altogether [Moussalli et al. 2011].

Two fundamental problems render dedicated tag decoding unsuited for our scenario: *(a)* the set of all relevant tag names must be known at circuit compilation time (no runtime-(re)configuration) and *(b)* routing the output signals of the tag decoder may require long signal paths which will deteriorate performance. In our system, tag name matching is wrapped *inside* each skeleton segment (cf. Figure 9), which keeps signal lengths short and independent of the overall circuit size.

Each tag matcher is connected to a *dedicated RAM* which holds the *tag predicate* that should be matched (*i.e.*, the tag name of a node test). In-silicon block RAMs on Xilinx FPGAs are 18 kbit in size. Thus, a single block is sufficient to store tag predicates.

The tag matcher signals true on its tag_match output when its local tag predicate was recognized and false otherwise. Algorithm 2 formalizes this behavior: the input data stream is compared character-by-character; tag_match is set to true when all seen characters matched and the length of the tag name is correct.

[3] For ease of presentation we simplified the algorithm to only `child` or `descendant` axes.

ALGORITHM 2: Tag matching. Parameters tag and taglen hold the tag name of an XPath name test and its length.

```
1 switch din.token do
2     case TAGSTART
3         pos ← 0;
4         partial_match ← true;
5     case TAGNAMECHAR
6         if din.char ≠ tag[pos] then
7             partial_match ← false;
8         pos ← pos + 1;
9 tag_match ← partial_match and (pos = taglen);
```

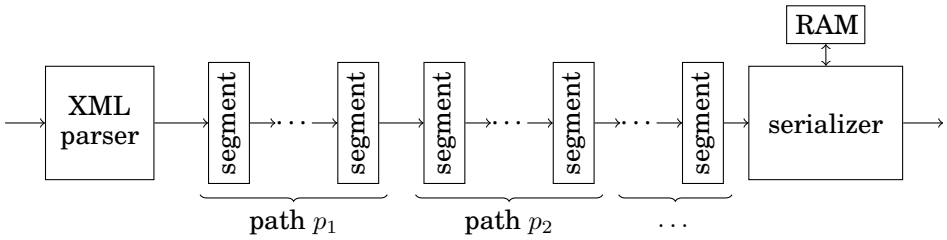


Fig. 10. Multiple paths can be matched within a single processing chain. Braces indicate the chain sections for projection paths p_1/p_2.

4.4. Matching Multiple Paths

Besides maintaining its own match state, each skeleton segment passes the (parsed) input XML stream directly on to its right neighbor. We can use this property to evaluate multiple projection paths within the same processing chain.

Figure 10 illustrates the idea. As the XML input is streamed through, sections of the entire chain of segments are responsible for evaluating different projection paths p_j. To realize this setting, all we have to do is ensure proper behavior at both ends of a chain section. We do so by introducing an explicit `fn:root ()` implementation and with help of *match merging* at the right end of a chain section.

Implementing `fn:root ()`. A segment for the XPath built-in function `fn:root ()` is the only one that does not depend on any previous matches. By placing it in front of every projection path, we break the finite-state automaton into separate automata that evaluate paths independently.

To evaluate `fn:root ()`, a segment must *(a)* enter a matching state exactly when parsing is at the XML root level and *(b)* become active in no other situation. We already have the tools available to implement both aspects of this behavior.

To implement *(a)*, we can initialize the history shift register such that history[last] ≡ true (so far we silently assumed that history[last] is initialized to false). The true flag will automatically be shifted accordingly such that the matching state re-appears whenever parsing moves back up to the root level. Property *(b)* can be assured by keeping the match_in signal false at the input of every chain section. The matcher will then match no tag in the document (Algorithm 1, line 3), but still follow a $\uparrow\!)*$ transition if it is configured to do so (*i.e.*, if `fn:root ()` is followed by a `descendant` step; line 4 in Algorithm 1).

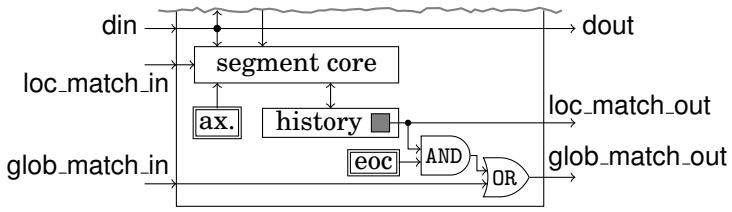


Fig. 11. Match merging to support multiple projection paths. Local matches are merged into the global match state if the parameter *end-of-chain-section* (eoc) is set.

Match Merging. At its right end, each chain section will compute the match state for its corresponding projection path. The serializer at the end of the processing chain must be informed whenever *any* of the paths along the chain found a match.

To establish this mechanism, we differentiate between *local* matches (for each of the p_j) and a *global* match. The former corresponds to the match_out signal that we used so far to find single-path matches. To implement the latter, we propagate an additional match flag along the chain and *merge* it with the local match result at the end of each chain section (using a Boolean 'OR' gate).

Figure 11 illustrates how match merging can be realized with only a few additional logic gates in each skeleton segment. At the end of each chain section (signified with an *end-of-chain-section* (eoc) configuration parameter), the local match state is merged into the global signal.

Resource Allocation. Note that the division of the entire chain into sections is not static. Rather, a sequence of segments is allocated as needed for each projection path. This lets us make efficient use of resources and enables high flexibility at the same time. Since segments are allocated on-demand, the same circuit can match either many short paths or fewer paths that are very long. The only limit is the aggregate number of XPath location steps in a set of projection paths, which must not exceed the number of segments n.

To illustrate this point, the twenty XMark queries that we look at in Section 8 use projection path sets with 3–15 paths per benchmark query (median: 4). The longest path in the XMark benchmark set contains 12 location steps. An allocation scheme where the per-path size is fixed would thus require at least $15 \times 12 = 180$ segment matchers, and no paths longer than 12 steps could ever be supported in such a design. With on-demand allocation, workloads are only limited by the total number of steps for a single projection path set, which ranges between $n = 7$ and $n = 79$ (median: $n = 15$) for the XMark benchmark. That is, 79 segment matchers would suffice to support the XMark workload (without individual constraints on path count or path size).

4.5. XML Serialization

Our engine is designed to support XML projection in a fully transparent manner, where the receiving query processor need not even know that it operates on pre-filtered XML data. Thus, the document must be filtered in such a way that an oblivious back-end processor will still produce the same query output (provided that all its projection paths have been configured in our engine).

To exemplify, in Figure 1 the document filter must preserve `site`, `regions`, and `africa` elements, even though they are not themselves matched by any projection path. Otherwise, Query Q_1 will miss its `regions` elements and return an empty result or—even worse—fail entirely because the projected document contains more than a single root element.

ALGORITHM 3: The *serializer* makes sure that full root-to-node paths are preserved for all output nodes. To this end, opening tags are copied to on-chip BRAM.

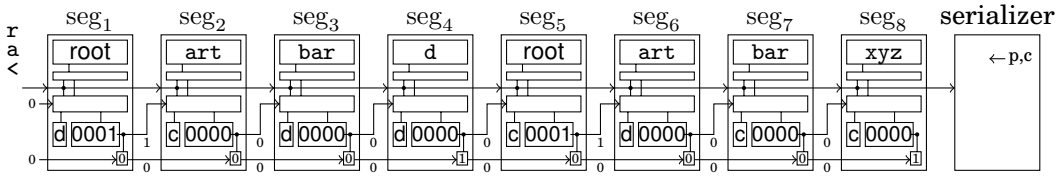
```
1  if match then
2      while printed_level < curr_level do
3          printed_level ← printed_level + 1;
4          print_opening_tag (printed_level);
5      copy din.char to dout;
6  switch din.token do
7      case TAGSTART
8          opening_tag ← true;
9      case CLOSINGTAGSLASH
10         opening_tag ← false;
11     case TAGNAMECHAR
12         if opening_tag then
13             copy din.char to tagmem[mempos];
14             mempos ← mempos + 1;
15     case OPENINGTAGEND
16         push (tagstack, mempos);
17         current_level ← current_level + 1;
18     case CLOSINGTAGEND
19         if not match then
20             print_closing_tag (printed_level);
21         printed_level ← printed_level - 1;
22         mempos ← pop (tagstack);
23         current_level ← current_level - 1;
```

Therefore, the serializer component of our circuit ensures that the root-to-node paths of all matching nodes are preserved in the circuit output. As the input stream is processed, the serializer writes all opening tag names into a dedicated RAM block. When a match is found, this information is read back and used to serialize full root-to-node paths.

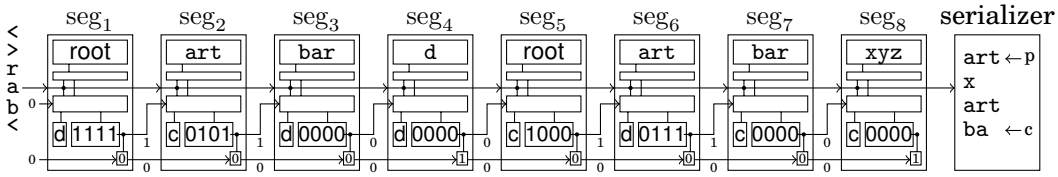
Algorithm 3 sketches the idea of serialization. When a match is discovered by the path matching engine, the input data stream is copied to the output, but not before opening tags were printed (from RAM), which is needed to ensure the root-to-node property (lines 1–5). In lines 7–17, opening tag names are copied from the input stream to the dedicated RAM tagmem. In lines 19–20, the printing of closing tags is enforced even when they are not fully contained in any matched document region (lines 21–23 do the necessary bookkeeping to prepare for coming opening tags).

In contrast to all other algorithms listed in this article, Algorithm 3 *cannot* straightforwardly be mapped to a VHDL circuit description. In the push-based design of *XLynx*, the serializer must be ready to accept a new input token every clock cycle. However, only one 8-bit symbol can be put back on the network wire at every clock tick, leaving no room to inject missing tags, as indicated with `print_opening_tag ()` in line 4.

Therefore, the serializer component uses a FIFO-based *queueing mechanism* (placed between "skeleton automaton" and "serializer" in Figure 5) to buffer incoming XML tokens while missing tags are printed. In this buffer, input tokens might queue up while the serializer fills in necessary start tags. The queue will drain, *e.g.*, whenever discarded XML content (which did not match any projection path) leaves "holes" in the



(a) Freshly configured *XLynx* projection engine.



(b) Same *XLynx* instance after having read the input '<art><x><art><ba'.

Fig. 12. Walk-through example for an *XLynx* instance that matches the two projection paths p_1 = fn:root//art/bar//d # and p_2 = fn:root()/art//bar/xyz.

input stream. Overall, the buffer will never grow larger than the concatenation of all currently open XML tags—a single BRAM block (approx. 4 kB) suffices to buffer this much data for any real-world XML instance.

4.6. Walk-Through Example

To better understand how the bits and pieces of *XLynx* work together, Figure 12 shows two snapshots of an *XLynx* instance while processing input.

Figure 12(a) shows the state of the projection engine right after configuration. Input is waiting on the left end of the engine, but no character has been processed, yet. For all segments seg_i, tag predicates have been configured to the respective tag name. A special root marker enforces match_in = false to implement fn:root(), as motivated in Section 4.4. Axis predicates have been configured to either c (child) or d (descendant).[4] Match mergers (on the bottom-right of each segment) contain *end-of-chain* = 0/1 ("false"/"true") flags to indicate the end of a segment chain. All history units are initialized to 0 (indicating false), except history[last] = 1 (or true) for fn:root() segments. In the serializer, printed_level and curr_level both point to the top of an empty stack.

Figure 12(b) shows the same *XLynx* instance after it has processed the XML byte sequence '<art><x><art><ba' (characters on the left indicate the input stream). The two starting 'art' tags have triggered matches in segments seg_2 and seg_6, leading to 1s being shifted into the corresponding history units. seg_6 has been configured to a successive descendant step. Hence, a sequence of 1s was shifted into the history unit of seg_6. By contrast, seg_2 is followed by a child step, such that only tags labeled 'art' lead to a 1 in the history unit, interspersed with 0s for remaining tags (cf. Algorithm 1).

Up to this point, no matches have been found (the global match flag on the bottom is 0 to indicate false). Hence, printed_level in the serializer still points to the stack bottom. Opening tags from the input stream have been copied into the serializer's BRAM, however (pushing curr_level to the new stack top). Once a match is discovered, the serializer will emit all opening tags between printed_level and curr_level to ensure complete root-to-leaf paths (cf. Section 4.5). Closing tags are always forwarded to the output, pushing printed_level and curr_level toward the stack bottom again.

[4]The axis predicate of the last segment of any path (*i.e.*, seg_4 and seg_8 in our example) implements the presence of a # in the projection path specification.

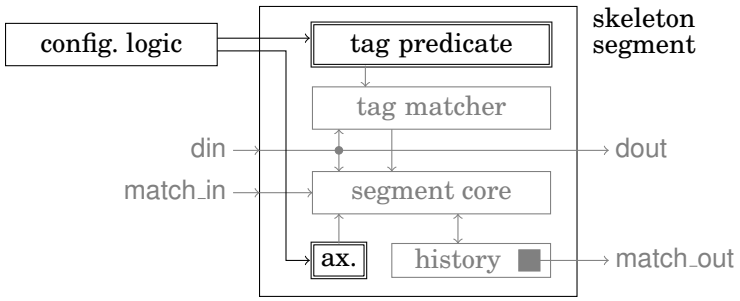


Fig. 13. Configuration logic changes workload parameters outside the main processing and data path.

5. RUNTIME (RE)CONFIGURATION

Now that we have seen how individual skeleton segments *interpret* configuration parameters to match sets of projection paths, it is time to look at the mechanisms to *set* those parameters at runtime. First, however, we need to briefly discuss suitable on-chip storage technology for each of the different flavors of configuration parameters.

5.1. Parameter Storage

Our skeleton automaton for XML projection depends on two flavors of query workload information: *(a)* the *XPath axis* of each navigation step and *(b)* the *tag predicate* that has to be evaluated along with the step, *i.e.*, a tag name or some information that encodes a node test. Both pieces of information could be placed either in *flip-flop registers* or in *dedicated RAM* (block RAM). To use the FPGA resources efficiently, we use both storage types, namely flip-flop registers for the XPath axis and block RAM for the tag predicate of each navigation step.

Flip-flop registers can be allocated at a granularity of a single bit. This is a good fit for small-sized pieces of information, such as the configured XPath axis or the `fn:root()`*/end-of-chain-section* flags. The benefit is two-fold: *(a)* we can allocate the exact number of bits really needed for those parameters and *(b)* flip-flops are directly woven into the remaining FPGA fabric, which lets them efficiently interact with lookup tables that, *e.g.*, implement the gates in a segment core.

Tag predicates, by contrast, can become much larger. Thus, we choose dedicated RAM to store them. Virtex-5 FPGAs contain hundreds of built-in (concurrently accessible) BRAM blocks, each of which is 18 kbit in size. This is suitable for storing tag predicates and leaves some room to accommodate even large query tag names. BRAM is single-ported. That is, it must be wired to exactly one logic module. In *XLynx*, we pair one BRAM block with each skeleton segment.

5.2. Changing Parameters at Runtime

Since all sub-circuits in an FPGA can operate in parallel and independently of each other, we can keep query workload updates completely outside the main processing and data path. As illustrated in Figure 13, separate *configuration logic* can maintain both configuration parameters without interfering with the processing logic.

The best way to provide query workload information to the chip depends on the particular system design (*e.g.*, Ethernet, PCI, or USB). To keep our system self-contained, we chose to communicate projection paths also via Ethernet. As illustrated in Figure 14, we inject the query workload directly into the input XML stream. Special *processing instructions* `<?query ... ?>` distinguish the query workload from the actual XML

```
<?xml version="1.0"?>
<?query reset?>
<?query fn:root()/descendant::regions/descendant::item?>
<?query fn:root()/descendant::regions/descendant::item/child::name #?>
<?query fn:root()/descendant::regions/descendant::item/child::incategory?>
<site>
  <regions>
    ...
  </regions>
  ...
</site>
```

Fig. 14. XML document with projection processing instructions `<?query ...?>` included.

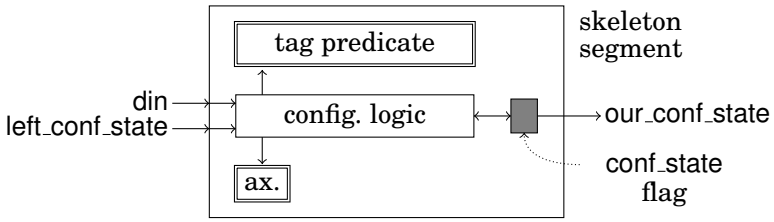


Fig. 15. Configuration logic for runtime query workload (re)configuration.

stream. For instance, the processing instruction

```
<?query fn:root()/descendant::regions/descendant::item/child::incategory?>
```

registers the new projection path `//regions//items/incategory` in the engine. These processing instructions are recognized by a small set of XML parser extensions. In the parsed XML data stream they are represented as special token values, which are interpreted by configuration logic. The configuration logic is wrapped into the individual skeleton segments of our system (see Figure 15). This makes the semantics of query workload changes deterministic, since the order between data stream items and workload changes becomes explicit in the parsed stream.

As parts of the query workload information (namely XPath steps) map almost one-to-one to the configuration parameters of individual skeleton segments (cf. Section 4.3.2), compiling input queries and inferring parameter values is simple enough to be performed directly on the FPGA chip. This is a significant deviation from previous approaches, where large amounts of CPU resources were needed to re-compile hardware circuits at runtime.

5.3. Configuration Logic and Segment Allocation

The configuration logic itself is distributed and integrated into the skeleton segments (again, this keeps on-chip signal paths short). The logic snoops the bypassing XML stream on the din signal line and writes configuration information into the respective storage units.

Figure 15 illustrates this interaction. Configuration logic in the middle interprets the din signal and updates tag predicates as well as the flip-flop-based configuration flags. Workload changes become effective immediately and will be considered for any data that follows the processing instruction in the input stream.

Segment Allocation. The left_conf_state and our_conf_state signals are used to coordinate segment allocation between segments. For new query workloads, skeleton segments are allocated and configured from left to right (that is, the first workload query

ALGORITHM 4: Semantics of configuration logic.

```
1  if din.type = CONFRESET then
2      conf_state ← unconfigured;
3  if conf_state = unconfigured and left_conf_state = configured then
4      switch din.token do
5          case AXISCHILD
6              axis ← child;
7          ...
8          case NAMETESTCHAR
9              update tag[...];
10         case FNROOT
11             history[last] ← true;
12         case COLONCOLON
13             conf_state ← configured;
14         case ENDOFPATH
15             end_of_chain_section ← true;
```

p_1 will occupy an automaton subset just after the XML parser; later p_i will follow the processing chain toward the serializer, cf. Figure 10).

To implement this behavior, the distributed pieces of the configuration logic synchronize between themselves with the help of a conf_state flag (implemented using flip-flop registers, see Figure 15) and left_conf_state/our_conf_state signals that are propagated from left to right. A local piece of configuration logic reacts to configuration tokens whenever it finds itself unconfigured and sees that its predecessor has changed its configuration state to configured. Once the local configuration is complete, the baton is passed to the right by setting the conf_state register to configured (which is also passed to the successor segment via the our_conf_state port).

Writing the Local Configuration. Parameters are written into local configuration storage while the parser tokens are passed through (tokens arrive in the same order as they are seen in the processing instruction, *i.e.*, in the XPath language format). As shown in Algorithm 4, different tokens will trigger writes to different storage locations (lines 1–3 and 13 implement the aforementioned synchronization).

The `<?query reset?>` processing instruction clears all configured projection paths. Lines 1–2 in Algorithm 4 implement this by re-setting the conf_state flag when the CONFRESET token is seen in the stream.

Reconfiguration Speed. The time needed by the processing instruction within the XML stream may thus be interpreted as the workload reconfiguration time. The 70-byte processing instruction above, for instance, requires 70 FPGA clock cycles to be processed, or 422 ns at an FPGA clock speed of 166 MHz.

6. WORKLOAD UPDATES AND AUTOMATIC DEFRAGMENTATION

The baton-passing mechanism described in the previous section works well to allocate skeleton segments from left to right when a set of projection paths is loaded into an initially empty segment automaton. In practice, users will demand the possibility to load or unload projection paths dynamically (without wiping out the entire existing configuration via `<?query reset?>`). In this section we describe a *deletion mechanism*

to unload projection paths at runtime and a *defragmentation mechanism* to reclaim automaton space that was occupied by unloaded projection paths.

6.1. Unloading Individual XPath Expressions

Unloading a projection path from the workload set presupposes that individual projection paths can be identified once loaded into *XLynx*. To this end, we extend our syntax for query registration to carry a *path id* as follows:

```
<?query 42 fn:root()/descendant::item/child::incategory?> .
```

Every skeleton segment that implements a part of this projection path will memorize the path id (here: `42`) in local configuration registers (implemented as flip-flops).

Once path ids have been associated with skeleton segments, a processing instruction like

```
<?query 42 remove?> .
```

can be used to remove the respective projection path from the workload set.

Internally, the `remove` command will only *deactivate* all skeleton segments that match the given path id. Deactivated segments will still occupy space in the *XLynx* processing chain. But they no longer react to incoming XML data or raise any of their match flags. Deactivation can be realized by adding

```
2a  if din.type = CONFREMOVE and path_id matches then
2b      conf_state ← deactivated;
```

to Algorithm 4 after line 2. In addition, any path matching (Algorithms 1 and 2) must be conditioned on conf_state = configured. Effectively, the deactivated skeleton segments become a *gap* in the segment chain that no longer actively participates in matching.

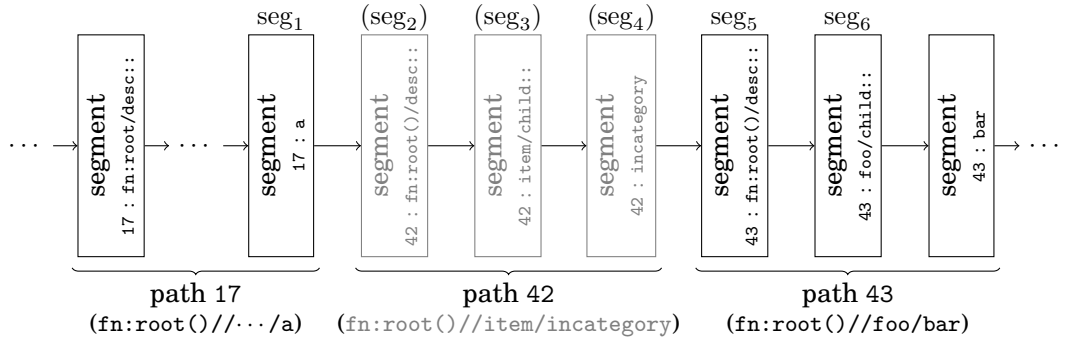
In principle, segments in this gap could immediately be re-used to register new projection paths. However, we are now experiencing the down sides of on-demand segment allocation. The size occupied by a registered projection path is not a pre-defined constant, resulting in a situation where a newly registered path might not fit into the gap left behind by a previous `remove` command. This is why path removal puts segments into a deactivated (rather than unconfigured) state. A *defragmentation mechanism*, which we will describe next, reclaims deactivated segments in a proper way to enable full dynamism for path registration and removal.

6.2. Automatic Defragmentation

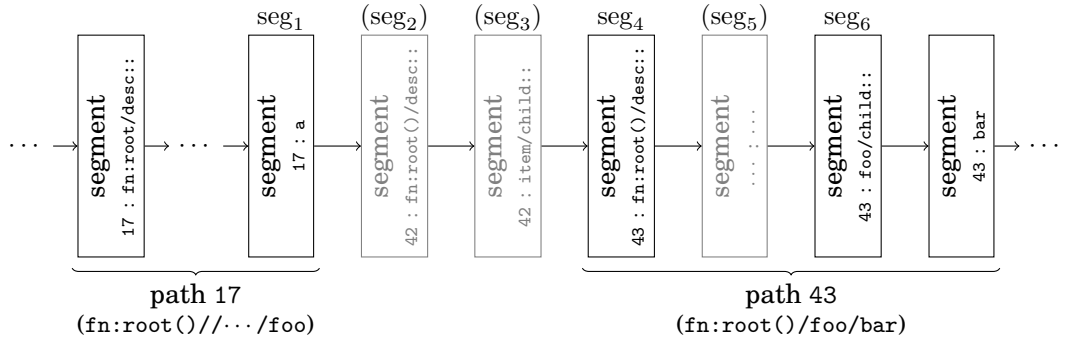
If an XPath expression is deactivated in an existing segment chain, this first creates a gap of unused segments. This is illustrated in Figure 16(a), where path `42` (previously covering segments seg_2 through seg_4) has been deactivated using a `remove` instruction (indicated using gray color).

Intuitively, we would like to reclaim the segments that were previously occupied by the removed path. By "pushing" deactivated segments toward the end of the segment chain, the set of unused segments would become contiguous and thus available for re-use by new projection paths.

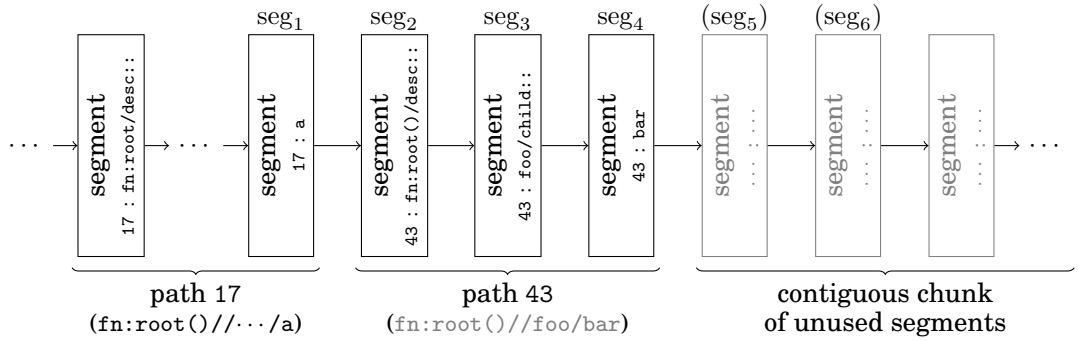
Idea: Configuration Copying. Figure 16(b) illustrates how this can be achieved. If a deactivated segment is followed by a configured one, we can *copy* all configuration settings from right to left, and then *swap* the states of the two segments. By repeating this process, unused segments gradually move toward the right where they become available for re-use.



(a) Segment chain just after segments for path 42 (segments $\mathrm{seg}_2, \ldots, \mathrm{seg}_4$) have been deactivated.



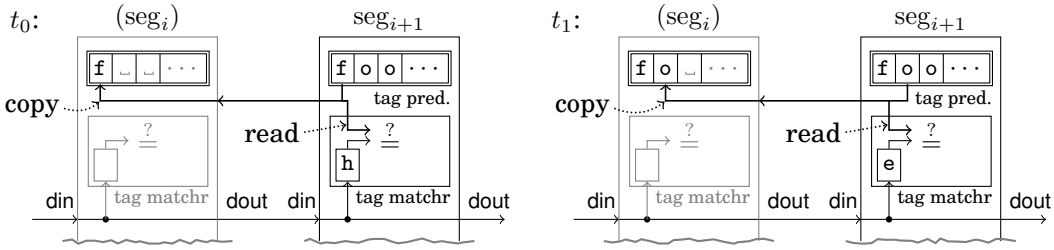
(b) The (previously deactivated) skeleton segment seg_4 swapped its configuration with segment seg_5, making the latter now deactivated.



(c) Eventually, all unused segments will become part of a contiguous chunk at the end of the segment chain.

Fig. 16. Automatic defragmentation exemplified.

Figure 16(b) illustrates the chain of skeleton segments just after path 42 has been removed and the first swap (between the segments marked seg_4 and seg_5) has been performed. Next, segment seg_3 will swap with seg_4 and segment seg_5 will swap with seg_6. Eventually, swapping will lead to the situation shown in Figure 16(c), where all unused segments have been pushed all the way to the right. They are now ready for re-use by newly loaded projection paths.



(a) Time t_0: seg_{i+1} is processing input token `h`.

(b) Time t_1: seg_{i+1} is processing input token `e`.

Fig. 17. BRAM copying as a side effect of input processing. While the (configured) segment seg_{i+1} processes the first tag name characters of the input "`<hello···`", the deactivated segment seg_i copies the BRAM content of seg_{i+1} character by character.

Semantics. Pushing unused segments this way leads to situations where a sequence of segments that implements one projection path is interspersed with segments marked as deactivated. For instance, in Figure 16(b), the deactivated segment seg_5 sits in-between the two active segments seg_4 and seg_6. To make sure such sub-automata still correctly implement their projection path, deactivated segments always propagate all match_out signals unchanged to the right. This way, such segments become transparent to their surrounding projection path.

6.3. Implementing Automatic Defragmentation

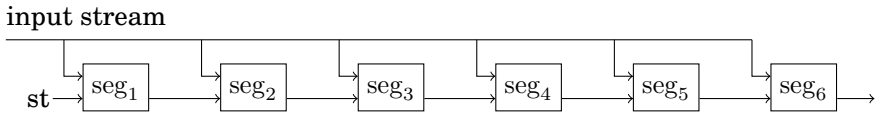
A challenge in realizing the idea in actual hardware is that swapping has to be performed *while* input data is being processed. To guarantee line-rate performance, the input stream *cannot*, for instance, be blocked while the skeleton automaton is being defragmented. This bears a high risk of *race conditions* when a segment state changes just while configuration and state are being swapped.

Furthermore, swapping is—at FPGA time scales—a rather time-consuming process. In particular, tag names cannot be copied from one segment to another as an atomic operation, but must be copied one word per FPGA clock cycle. What is more, there is only a single access port to each BRAM block. And since the configured skeleton segment seg_{i+1} is still processing data, the deactivated segment seg_i cannot independently read out BRAM contents to implement word-by-word copying.

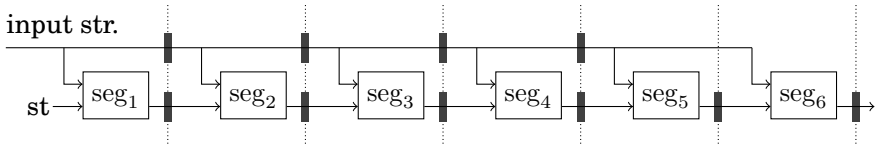
Copying as a Side Effect. However, BRAM copying can be performed as a *side effect* to input processing. To this end, a deactivated skeleton segment seg_i passively "listens" to BRAM reads initiated by its configured neighbor seg_{i+1}. As soon as the next XML start tag passes by from the input stream, seg_{i+1} will read out its BRAM content, automatically making the information available also to seg_i.

BRAM copying as a side effect of input processing is illustrated in Figure 17, assuming that the skeleton automaton just parses the character sequence "`<hello···`" (*i.e.*, an opening XML tag). To process the two leading tag name characters `h` and `e`, segment seg_{i+1} reads out the first two characters from its local BRAM. While doing so, it copies all BRAM output to segment seg_i, which is in deactivated state. seg_i writes this information into its own BRAM. As soon as all contents of seg_{i+1}'s BRAM have been copied, seg_i and seg_{i+1} can swap their states, making seg_i then in charge of matching `foo` tags.

Helper Scans. To read out BRAM contents, a deactivated segment always needs the assistance of the segment that "owns" the BRAM. This is because that segment might need to read out tag names to process input data that just flows by. If implemented



(a) standard approach to hardware NFA



(b) fully pipelined NFA for XML projection

Fig. 18. Standard hardware NFA implementation (top) requires long signal paths. Pipelining (bottom) reduces signal paths by inserting *registers* ▮.

as described in the previous paragraph, this would mean that contents are *only* copied whenever an opening tag (of sufficient length) occurs in the input stream. In practice, this might delay automatic defragmentation or even prevent copying altogether.

There are also situations, however, when a node like seg_{i+1} in the illustration above does not strictly need its access port to the BRAM, *e.g.*, while processing text node content or other node types. As an optimization, we can set segments to perform "helper scans" on their BRAM in such situations. Simply by scanning their BRAM, they make BRAM contents available to a potentially listening predecessor segment. In practice, we found "helper scans" to be a sufficient mechanism to quickly defragment the skeleton automaton even for very dynamic workloads.[5]

7. TUNING FOR PERFORMANCE

As in software-based systems, the observable performance of an FPGA-based solution hinges on a proper low-level implementation that matches the characteristics of the underlying hardware. Most importantly in FPGA design, a circuit must *(a)* meet tight *timing constraints* (such that it can be operated at high clock speeds) and *(b)* utilize *chip space* efficiently (to support real-world problem sizes at low cost). In this work we use *pipelining* and *BRAM sharing* to address both aspects.

7.1. Pipelining

The standard approach to hardware-based finite-state automata is to forward incoming stream tokens simultaneously to *all* involved automaton states. In Figure 4, for instance, the output of the tag decoder was sent to all '`AND`' gates at the same time. Figure 18(a) emphasizes the same concept but hides the inner details of circuit segments seg_i.

Signal Paths. Figures 4 and 18(a) both also show the problem that this incurs. For larger automata, the length of the 'input stream' communication paths will increase. In general, the processing speed of any hardware circuit is determined by its *longest signal path* between any two clock synchronization points.

When arbitrary automata shapes must be supported, long signal paths are inevitable. The new value of a state q_i might depend on any other state q_j, hence, q_i must be reachable from q_j within one clock cycle. Non-deterministic finite-state automata

[5]When operating *XLynx* over Ethernet, packet headers and inter-frame gaps lead to enough idle time, such that defragmentation appears to happen almost instantaneously (order of micro-seconds).

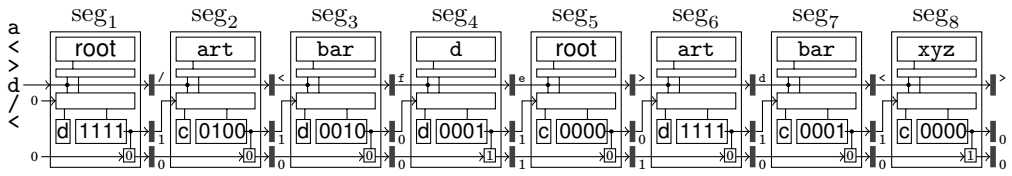


Fig. 19. Skeleton automaton with pipelining enabled; pipeline registers indicated as ▌. Illustrated is the automaton after the byte sequence '<art><x><art><bar><d>ef</d' has been consumed from the input. (History units in seg_1 through seg_6 have overflown; to keep the illustration readable we printed only the last four bits of every history stack.)

generated from XML projection paths, however, will always follow a very particular pattern. Their shape is strictly *sequential* and all data flows in the *same direction*.

Pipelining. The corresponding circuits are thus amenable to *pipelining*, a very effective circuit optimization technique. Figure 18(b) illustrates the idea. The one-directional data flow is broken up into disjoint *pipeline stages* (indicated with a dotted line). Whenever any signal crosses a stage boundary, a *register* (marked as ▌) is inserted. Every register will buffer its input signals in clock cycle i and make the values available to the successor segment in clock cycle $i + 1$.

Registers act as a synchronization point. The longest signal path is now reduced to the longest path *between any two registers*. In contrast to the original design, the longest path length no longer depends on the overall circuit size, but remains unchanged even if the automaton size is scaled up. This way, in an n-stage pipeline (n is also called the *pipeline depth*) the available FPGA hardware parallelism is turned into a parallel processing of n successive input data items (*i.e.*, input bytes).

Throughput vs. Latency. Pipelining primarily increases the *throughput* of a hardware circuit. The clock frequency is increased and, in a fully pipelined circuit, a new input item can enter the circuit every clock cycle. This benefit comes at the expense of a small *latency* penalty that increases proportionally to the pipeline depth. In general this penalty is negligible: with a 6 ns clock period, even a 500-stage pipeline will have a latency of only 3 μs—far less than, say, the same data item traveling over the network in a client-server setup.

Pipelining in Action. Figure 19 illustrates the *XLynx* instance of Figure 12 with pipeline registers installed. Input bytes are no longer broadcast to all segments in parallel, but propagate through the pipeline stage-by-stage. Pipeline registers are indicated as ▌; at every register output, we indicate the current register value.

Segment seg_4 in this figure has finished processing all bytes up to 'e', hence has discovered a match for p_1. This match is indicated via the glob_match_out line on the bottom, but the matching information was so far only forwarded until seg_5 (the successful match is indicated together with the '>' byte of the matching <d> tag). Once the matching information has been forwarded until the serializer component, the serializer will first emit opening tags from the root-to-leaf path that have not yet been printed to the output. Then it will let all input bytes pass to the output as long as glob_match_out $=$ true.

XPath Semantics. Pipelining has an interesting side effect with respect to the semantics of XPath evaluation. Consistent with the original work on XML projection [Marian and Siméon 2003], our supported language dialect covers the XPath `self` and `descendant-or-self` axes. These axes *cannot* be expressed using a standard hardware automaton like the one shown in Figure 4, because a segment circuit seg_i will

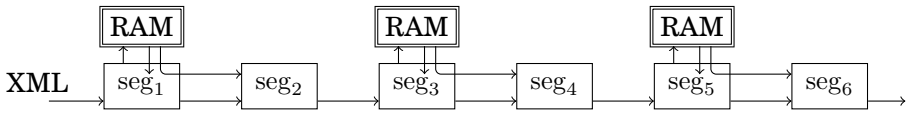


Fig. 20. BRAM sharing. Two segments store their tag predicates in the same RAM block. Since each block has only one interface, segments seg_{2k-1} mediate traffic for segments seg_{2k}.

report a new match state only *after* an input item x has been consumed; this is too late for the successor seg_{i+1} to perform a match on the same input item x.

In a pipelined circuit x is processed by seg_{i+1} one cycle later. This gives us the opportunity to *fast-forward* the match state of seg_i in case of a `self` or `descendant-or-self` axis. A fast-forwarded state bypasses one intermediate register. This way, input item x arrives together with the matching state of seg_i at segment seg_{i+1}, leaving enough time to implement the '`self`' functionality.

Existing automaton-based XPath engines either do not support `-self` axes at all (to our knowledge, no existing system does), or they compile `-self` axes into complex multi-way predicates, *e.g.*, a sub-path `child::`τ_1`/self::`τ_2 would translate into a conjunctive predicate 'matches τ_1 $\wedge$ matches τ_2'; `descendant-or-self` axes become even more complex. Without an upper bound on the number of conjunctions, resources for predicate evaluation have to be allocated dynamically. By avoiding a second case of dynamic resource allocation, we can save precious chip space, which allows our circuit to scale better for larger workload sizes.

7.2. BRAM Sharing

As discussed before, we use dedicated RAM to store tag predicate configuration parameters for all skeleton segments. This may lead to an upper limit on the number of segments that can be instantiated (and thus on the supported size of projection path sets), because the available number of RAM blocks is fixed. The Virtex-5 chip that we used in our experiments, for instance, contains 296 blocks of RAM, which would limit the number of segments to 296 (minus a few BRAM blocks that are needed for the *serializer* and surrounding glue logic).

At the same time, we are underutilizing the available RAM blocks. The full 18 kbit of a Virtex-5 BRAM unit are rarely needed for a tag predicate in the real world, and we read out only one byte at a time, even though BRAMs would support a (configurable) word size of up to 36 bits.

BRAM usage can be improved by *sharing* each BRAM unit between two or more segments, which effectively multiplies the supported NFA size. Figure 20 illustrates how this idea can be realized in FPGA hardware. Since there is only one port to each BRAM block, some segments act as *mediators* for the communication information of their neighbors.[6]

BRAM sharing is useful only up to the point where the number of segments is bound by the amount of logic resources (lookup tables and flip-flop registers) available. As we will see in Section 8, BRAM and logic resources are in balance on our hardware when three segments share one BRAM unit.

8. EVALUATION

We implemented and tested *XLynx* on widely available and low-cost ($750 academic price) FPGA hardware. The Xilinx XUPV5 development board is equipped with a Virtex-5 XC5VLX110T FPGA (69,120 LUTs, 69,120 flip-flops; 296 $\times$ 18 kbit BRAM) and

[6]The maximum word size for each BRAM block is 36 bits. Up to four segments can thus share one BRAM block by concatenating their 8-bit data into one large (32-bit) word.

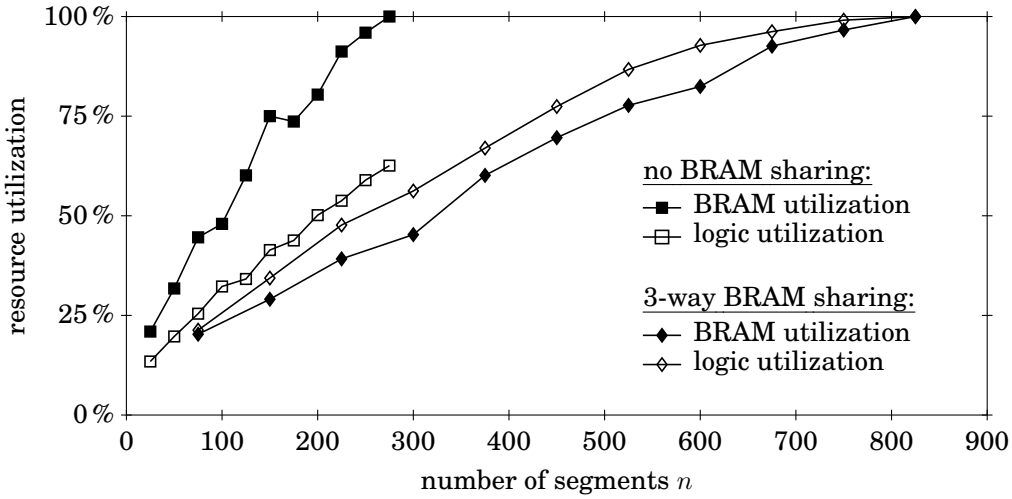


Fig. 21. FPGA chip resource consumption of engine configurations with and without BRAM sharing. BRAM sharing allows to balance the use of logic and BRAM resources to obtain a larger overall engine size.

has a number of high-speed I/O connectors to communicate with outside systems. The Virtex-5 chip series has been released already in 2006. More recent chip series—Xilinx' most recent series is Virtex-7—offer substantially more chip resources and better timing/speed characteristics. When evaluating *XLynx* in a full system setting, we assume an Intel Sandy Bridge system, equipped with an i7-2700k CPU (3.5 GHz; 8 MB L3 cache) and running Ubuntu Linux.

In the following Section 8.1, we first characterize *XLynx*, *i.e.*, the core XML projection engine running on the FPGA. Then, in Section 8.2, we show how the engine could be used in a working system, together with a full-blown XQuery engine such as Saxon-EE. As a workload, we use a 116 MB XMark instance ([Schmidt et al. 2002]; scale factor 1) and the twenty XMark queries.

8.1. *XLynx*: Core XML Projection Engine

To analyze the characteristics of *XLynx*, we compiled it to actual FPGA circuits in various configurations. Besides an obvious expectation of sufficient data throughput, two aspects are particularly interesting to judge the quality of an FPGA design:

economic resource utilization The given FPGA hardware imposes strict limits on the types and amounts of available hardware resources. A good FPGA design is properly balanced to make near-optimal use of the available resources.

scalability An FPGA circuit should provide stable performance even when its size is scaled up, *e.g.*, when it is ported to larger and more powerful FPGA hardware.

Economic Resource Utilization. Using our available hardware, we implemented various configurations of the XML projection engine, varying the number of skeleton segments; with and without BRAM sharing enabled. For each configuration we determined the amount of FPGA resources the resulting circuit uses.

Figure 21 illustrates the utilization of BRAM units (denoted by filled markers) and logic blocks (*i.e.*, slices, denoted by empty markers) as a percentage of the total available BRAMs/slices on the chip. The results are consistent with the expectations that we stated in Section 7.2. Without BRAM sharing, all BRAM resources are used up

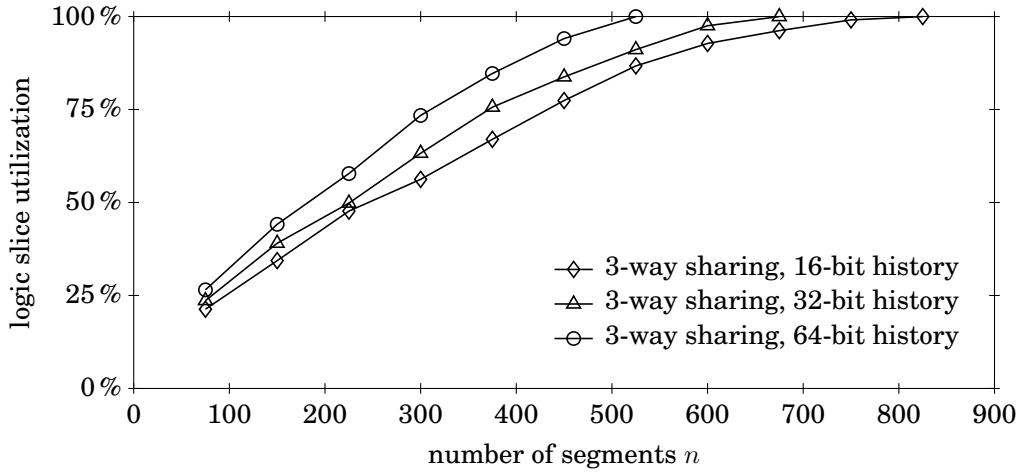


Fig. 22. The depth of the *history unit* may affect the logic resource consumption of the XML projection engine. With 3-way BRAM sharing enabled, changing a 16-bit history configuration to 64 bits results in a space overhead of 20–30 %.

for circuit configurations beyond approximately 275 segments. At the same time, more than ⅓ of the available logic resources are unused.

BRAM sharing can bring resource utilization into balance. With 3-way BRAM sharing (diamond symbols in the plot), the maximum number of segments is now limited by logic resources (specifically, lookup tables) and we can instantiate more than 800 segments on our chip, *i.e.*, we can support three times as many concurrent projection paths.

Effect of Configuration Parameters. The resources reported in Figure 21 assume a circuit configuration where the history unit in each skeleton segment is 16 bits in size. Increasing this value may increase the filtering accuracy of our projection engine, though only when matches need to be tracked in XML sub-trees deeper than 16 levels.

Increasing the depth of the history shift register will have no effect on the throughput of the XML projection engine (shift registers can easily be implemented in a scalable manner; we verified this with a separate set of experiments). But larger shift registers will require additional flip-flops (to hold the additional state) as well as as additional lookup tables, which will drive the added registers.

Figure 22 illustrates this effect for configurations where we set the shift register depth to 16, 32, and 64 bits. As can be seen in the figure, increasing the depth from 16 to 64 bits increases the overall chip slice consumption by about 20–30 %. It seems very unlikely, however, that any real-world use case will use XML documents this deep *and* require matches to be accurately tracked in those deep sub-trees.

Scalability. To evaluate the scalability criterion, we used the FPGA design tools to determine the maximum *clock frequency* at which each of our engine configurations could be operated.[7] Figure 23 illustrates the numbers we obtained.

The clock frequency directly determines the *maximum speed* of the XML projection engine. One input byte can be processed on every clock cycle (independent of the

[7]Physical constraints on FPGA hardware (clock frequencies are generated by a *phase-locked loop*) restrict allowable frequencies to $^n/_m \times 100$ MHz (*i.e.*, 150 MHz, 160 MHz, 166 MHz, 175 MHz, 180 MHz, 200 MHz, and 225 MHz).

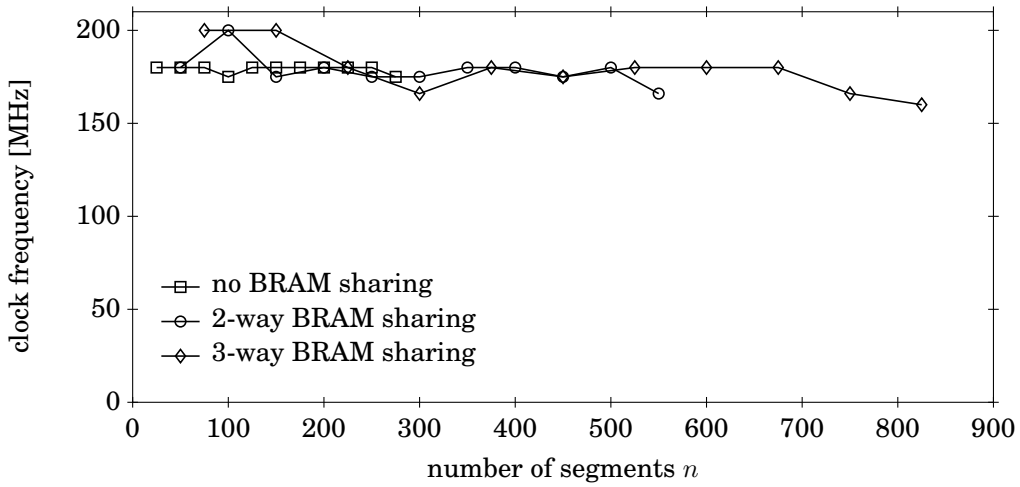


Fig. 23. Maximum clock frequency for various engine configurations. Frequency is not strongly influenced by circuit size, which indicates scalability to support also newer or future chip devices.

query workload). With clock frequencies around 180 MHz, our system could thus sustain 180 MB/s XML throughput. This is more than enough for the use cases our system is designed for: it could easily, for instance, keep up with an XML stream that is served from disk or via a network link. 180 MB/s is the *guaranteed* throughput rate at the XML input side. It will be sustained independent of the XML document characteristics and/or the set of projection paths being matched. Hard performance guarantees are one of the key properties that make FPGA accelerators for data processing so appealing.

The clock frequencies shown in Figure 23 are also a good indicator for the scalability characteristics of our system. Since chip space and parallelism are the main asset of FPGAs, the achievable clock frequency should not (significantly) drop when the circuit size is scaled up. Only then can a circuit really benefit from expected advances in hardware technology (Moore's law predicts that the transistor count per chip doubles approximately every two years).

In our case we see that the achievable clock frequency stays high even for configurations that significantly exceed the 70-80 % chip utilization, beyond which performance often decreases [DeHon 1999]. It is reasonable to expect that our system will keep its performance characteristics even when it is scaled up to 6000 or more segments on current Virtex-7 chips [Xilinx Inc. 2011].

8.2. *XLynx* Integration with an XQuery Engine (Saxon Enterprise Edition)

FPGAs may offer significant advantages over software-based systems in terms of performance and/or power consumption. Even more attractive are their unique *system integration* opportunities that cannot be matched with commodity hardware. To demonstrate this advantage, we connected our engine directly to the Ethernet interface. The so-obtained system can perform XML filtering *in the network* as data is sent from a network server to a client.

Thus, we inserted *XLynx* in the data path between the data storage and the XQuery engine. Rather than replying directly to the XML processor, the server sends the raw XML stream to the FPGA pre-filter. There, the data is projected and forwarded to the XQuery engine. Figure 24 illustrates how a query is processed in such a setup. First,

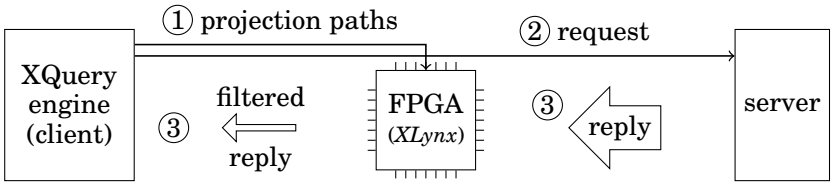


Fig. 24. A hybrid system with *XLynx* inserted in the data path. For each query, projection path information is sent to the FPGA (1) and a data request to the server (2). Data is sent back and filtered on the FPGA (3). All communication is using Gigabit Ethernet.

Table I. Effect of FPGA-based XML projection on overall power consumption. The FPGA has a fixed power consumption of 16.7 nano-Joule per input byte. CPU power consumption was computed as *execution time* $\times\, 30\,W$.

	software only	*XLynx* FPGA/CPU hybrid			
Query	CPU	FPGA	CPU	total	saving
Q1	71.5 J	1.83 J	12.6 J	14.5 J	80 %
Q8	97.0 J	1.83 J	39.1 J	40.9 J	58 %
Q11	1363.5 J	1.83 J	570.1 J	571.9 J	58 %
Q15	69.0 J	1.83 J	2.4 J	4.2 J	94 %

the software system sends projection path information to the FPGA (1), then requests the XML data from the server (2). The reply is sent via the FPGA (3), which filters the data "in the network."

Power Efficiency. FPGAs are clocked at significantly lower rates than main-stream processors (*e.g.*, in our case: 180 MHz vs. 3.5 GHz). On top of that, a dedicated circuit for a specific problem can spend much less transistors for control logic and many more transistors for the task itself. This makes them consume only a fraction of the power that a general-purpose CPU would need to perform the same task. Heterogeneous CPU/FPGA systems thus promise lower costs for energy and cooling. More importantly, all modern processor designs are *power-limited* [Borkar and Chien 2011]. In a modern system, any savings in power become immediately available to increase overall performance.

Power consumption is notoriously hard to measure accurately. The CPU that we use on the software side is rated at 95 W TDP[8], but there is no public information about its actual power consumption, which will depend on the type of load that is running on the CPU. As a conservative estimate, in this paper we assume a power consumption of only 30 W at 100 % system load (less than a third of the rated maximum power dissipation). Power measurements at the external wall plug confirmed that the actual power consumption at full load is much higher. Hence, for a real system the power advantage of using an FPGA will be even higher than reported here.

Our FPGA hardware is not equipped with power measurement facilities either. But software design and simulation tools can very accurately determine the maximum power consumption that a specific FPGA design will exhibit at a particular clock frequency. For our designs, Xilinx Power Analyzer reported a maximum power consumption of 3 W.

With a clock rate of 180 MHz, this means that our XML projection engine consumes about 16.7 nJ (nano-Joule) per XML input byte, independent of the XML projection workload. The energy consumed by the back-end CPU depends on the amount of work that it has to perform. Table 8.2 lists the total energy consumption needed to

[8] "Thermal Design Power"

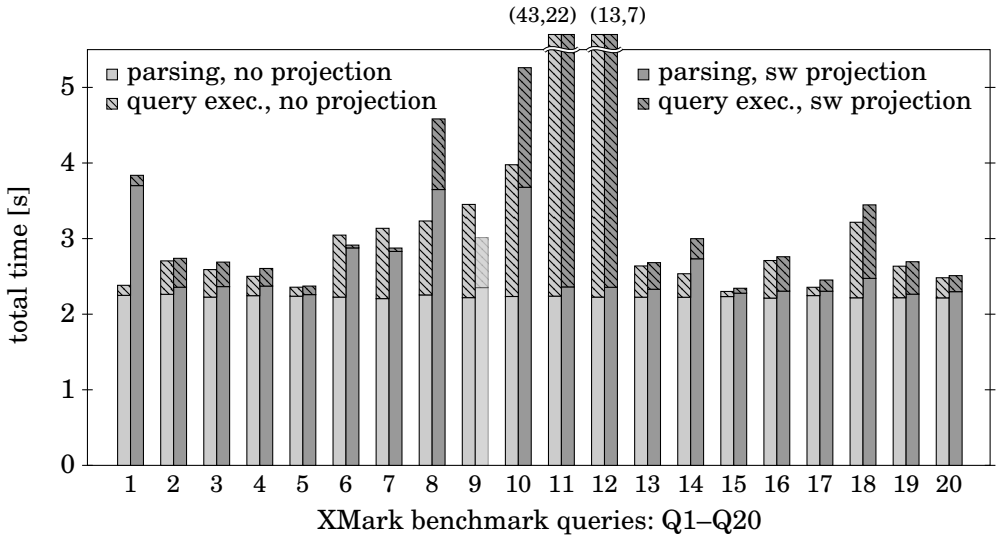


Fig. 25. Parsing time and query execution time (XMark scale factor 1; 116 MB XML): no projection versus software-only projection of Saxon-EE. Software-only projection for query $Q9$ produced an incorrect result. Runtimes for $Q11$ and $Q12$ exceed scale (actual values printed above).

run four of the XMark queries in a CPU-only setting and when using *XLynx*'s hybrid FPGA/CPU architecture. As can be seen in the figure, pre-filtering in hardware substantially reduces the overall energy consumption for all queries.

Filtering Throughput. XLynx operates in a strict *streaming mode* and processes one input character per clock cycle. Thus, by design the filtering throughput of our system is independent of the query workload. As detailed above, *XLynx* can sustain throughput rates of 180 MB/s. This is more than the Gigabit Ethernet link of our system can provide, so effectively our system is only limited by the physical network speed.

In measurements on real hardware we validated that *XLynx* can sustain full Gigabit Ethernet line rate. We observed a maximum payload throughput of 109 MB/s. With protocol overhead accounted for, this corresponds to a bandwidth of 123 MB/s on the physical network link, or 98.4 % of its maximum capacity. To fully saturate our filtering engine, we would have to connect our chip to a faster network (*e.g.*, 10 Gb/s Ethernet) or to a different I/O channel (*e.g.*, 3 Gb/s SATA Gen 2).

8.3. Effects of Projection on Memory Consumption and Performance

To judge the runtime characteristics of a hybrid FPGA/CPU system, we plugged *XLynx* in front of Saxon-EE (version 9.4.0.3), a state-of-the-art XQuery processor for in-memory processing. We measured parsing time, query execution time, and memory consumption of Saxon when running the 20 XMark queries. Since Saxon cannot directly process the streaming XML protocol of our engine, we measured the filtering throughput of our FPGA (previous section) and Saxon performance independently (and ran all Saxon experiments from a memory-cached file).

Feasibility of XML Projection. The light bars in Figure 25 illustrate the processing speed of an off-the-shelf Saxon-EE processor for the twenty XMark queries, broken down into time spent on XML input parsing (▯) and actual query execution time (▨). For all queries, except for join queries Q11 and Q12, which are known to be complex, input parsing dominates the total execution time. On raw data, Saxon requires 2.23 sec for

input parsing, independent of the query, and actual query execution times were 68 ms–41 sec, with a median value of 390 ms. These measurements confirm the observation of Nicola and John [2003] that processing speed for XML data is often limited by the system's *parsing cost*, not by query execution per se.

Unfortunately, this situation is hard to address by software-only solutions. *Any* software-based XML processor will have to parse the input document, and—due to the sequential nature of XML—the opportunities to accelerate XML parsing are very limited.

XML Projection in Software. Under these premises, it is not surprising that software-based projection brings only limited benefit for end-to-end processing speed. The Enterprise Edition of Saxon (Saxon-EE) includes such a software-based projection feature. After enabling the feature, we obtained the performance numbers shown in dark gray in Figure 25 (again broken up into parsing time ▮ and query execution time ▨).

In a software implementation, projection is performed while parsing the input document. As can be seen in the figure, enabling projection thus even *increases* the parsing cost for all twenty queries (now 2.3–3.7 sec; median: 2.36 sec), resulting in an overall slowdown for most of them. The evaluation of projection paths during input parsing causes additional CPU load that cannot be compensated by a reduced cost to build Saxon's internal tree representation. Since XML parsing is an inherently sequential task that dominates overall execution cost, Amdahl's law indicates that there is little room to improve XMark performance with software-only solutions, such as multi-core parallelism or distribution, as suggested by Cameron et al. [2008].

For most queries, input projection has very little effect on the time spent on the query execution part, which is consistent with the observations of Kay [2008]. Saxon is very good at touching only those parts out of the whole document that are actually relevant to the given query. Any XML data that projection could filter away might occupy memory resources, but they will not typically cause any processing overhead.

XML Projection in Hardware. The game changes when we perform XML projection in hardware. Hardware-based projection reduces the amount of XML data seen by the back-end processor by as much as 63–99.9 % (average: 97.0 %; median: 98.3 %). The reduced amount immediately translates into a reduced parsing overhead.

The effect is illustrated in Figure 26 (shown in dark gray next to the baseline situation without projection). Parsing times now range between 31 and 599 ms (median: 283 ms), a significant reduction over the software-only situation.

As with software projection, filtering has less effect on the actual query execution time. Here we measured 45 ms–18 s (median: 346 ms) after filtering. Again, this is in line with previous reports on document projection in Saxon [Kay 2008]. Nevertheless, for most queries, where *parsing time* is the dominant factor, total execution time can be significantly reduced.

Memory Consumption. Our experiments confirm that XML projection is an effective technique to *reduce memory* overhead during query processing. This was one of the incentives for XML projection [Marian and Siméon 2003]. Our measurements regarding memory savings are displayed in Figure 27.

Main-memory consumption is query-dependent and amounted to 363–685 MB on our system (median: 518 MB), when no projection was used. With hardware projection main-memory consumption could be significantly reduced for all 20 XMark queries—memory consumption went down to 12–207 MB (median: 25.6 MB). This effect manifests itself even for those queries that lead to a significant number of projection paths (cf. Section 4.4).

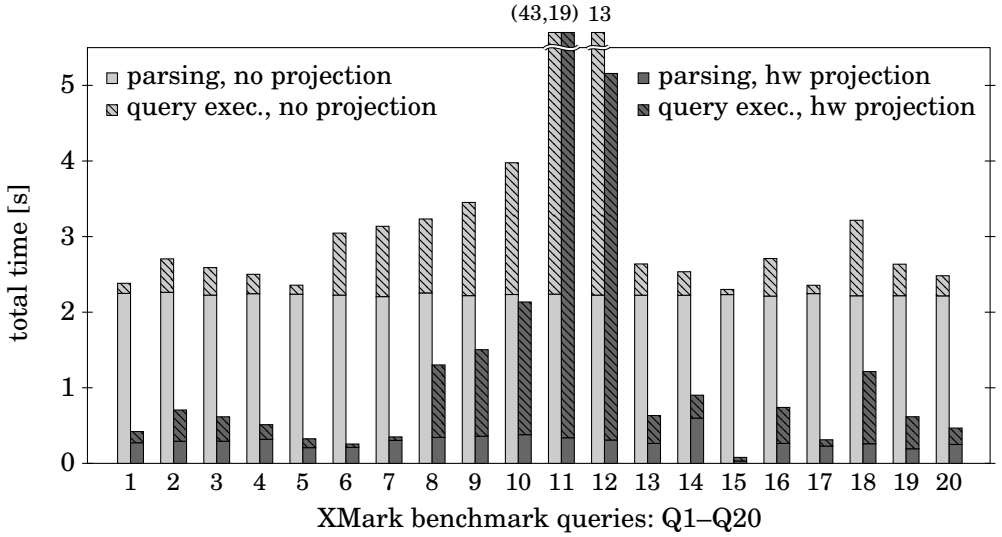


Fig. 26. Parsing time and query execution time: no projection versus hardware projection with *XLynx*. Runtimes for $Q11$ and $Q12$ (without projection) exceed scale.

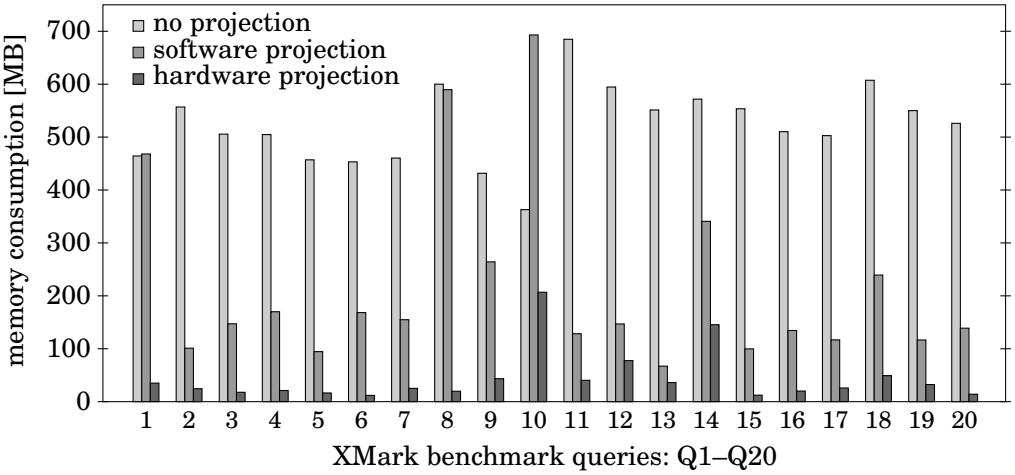


Fig. 27. Memory consumption of Saxon-EE with no projection (□), Saxon's software projection (▒), and hardware projection (■).

Intuitively, XML projection should reduce the in-memory tree sizes by the same amount, whether computed in hard- or software. However, when we tested Saxon's software-based projection mechanism, the memory savings were less than the results we obtained with hardware-based filtering. We attribute this to the way how garbage collection is realized in the Java runtime (Saxon is written in Java), which introduces some non-determinism in the memory consumption. We even found situations where memory consumption increases after we enabled software-only projection (Queries Q1 and Q10).

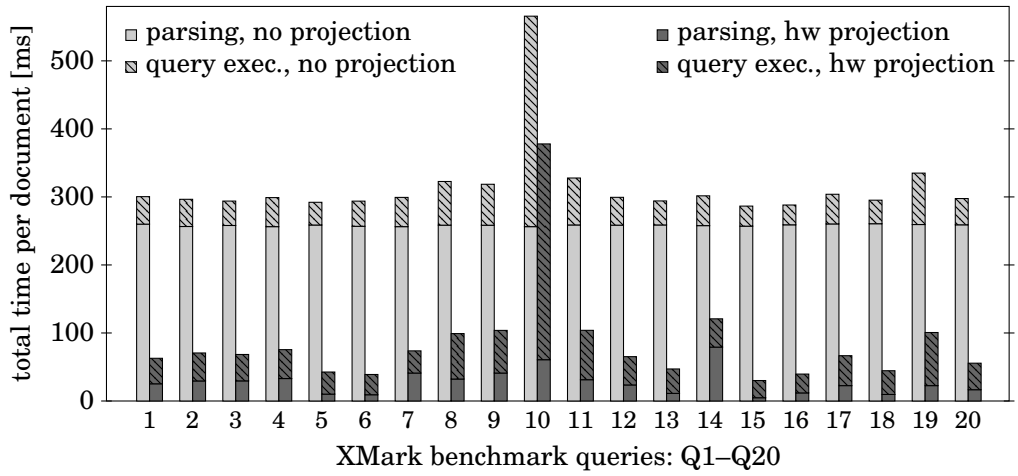


Fig. 28. Effect of hardware-based XML projection on parsing and query execution times when using small input documents (`xmlgen -f 1 -s 1000`).

Collections of Small XML Instances. XML projection was invented originally to handle also large XML instances in memory-limited, stream-oriented XML processors, a situation that is well reflected by the experimental setup above. Arguably, however, large sets of small XML documents might be more appropriate in stream-based environments.

To see how *XLynx* reacts to such environments, we instructed the XMark data generator to produce a collection of small XML documents, rather than a single file. Invoked with the command line option `-s 1000`, the XMark document generator `xmlgen` produces XML files which vary between 280 kB and 3.3 MB in size (average: 1.7 MB). On this XML collection, again we ran Saxon with and without hardware-based projection applied.

The results are illustrated in Figure 28. The general picture of this graph resembles the one we saw for larger XMark instances (Figure 26). For most XMark queries, parsing and query execution depend linearly on the input document size. An exception are the value-based joins in Queries Q11 and Q12, which exhibit quadratic complexity with growing document sizes. As can be seen in the figure, this eliminates the dominance of the query execution part, making both queries benefit even stronger from hardware-based projection.

8.4. Comparison with Existing Work

Given the importance of the XML in many application areas, various types of accelerators have been proposed to (pre-)process XML on non-CPU hardware. The exact way how that hardware is used—and under which parameter setting it runs most efficiently—is highly application-specific. Nevertheless, in this section we try to compare *XLynx* to alternative approaches that use *(a)* FPGAs and *(b)* graphics processors (GPUs) for acceleration.

FPGA-Based Solutions. The FPGA solution of Moussalli et al. [2011] targets *publish/subscribe* scenarios. A (possibly very large) number of subscribers registers queries with a publisher system. The publisher matches incoming documents (often very small) against those queries and forwards documents only to matching subscribers. The role of the FPGA in [Moussalli et al. 2011] is to perform XML matching, then forward a *bit vector* to a host CPU to indicate which paths/subscriptions have

matched the current document. This avoids the need to re-serialize matches from the XML input. But it bears a risk of volume amplification: previous systems have assumed document sizes of 1–100 kB [Kwon et al. 2005] or even less (77 XML element nodes according to [Diao et al. 2003]) and up to 150,000 subscriptions; the FPGA will actually *increase* the communication volume under such parameters.

The system of Moussalli et al. [2011] currently supports at most 4,000 subscriptions on a Virtex-5 LX330 FPGA (which is three times as large as our chip). The system is further limited in its support for XML: tag names must be two characters in size and there must be at most 64 distinct tag names. Under these constraints, their matching engine achieves a throughput around 200–250 MB/s, significantly less if the design is scaled to more than 50 % chip space utilization. As expected, per-query compilation offers better XML throughput and a higher resource efficiency than the dynamic *XLynx* approach (on LX330 hardware, *XLynx* could host about 2,500 skeleton segments or ≈ 625 four-step subscriptions). Given the above limitations, however, this advantage is less than one might expect.

The price for the runtime efficiency of [Moussalli et al. 2011] is the need to re-compile the full FPGA circuit upon every workload change. The time required to re-compile is not explicitly stated in [Moussalli et al. 2011], but amounts to "several hours" according to the authors. *XLynx* can embrace workload changes within micro-seconds instead.

Graphics Processors (GPUs). The same authors also implemented publish/subscribe-style matching on graphics processors (GPUs) [Moussalli et al. 2011], exploiting the hardware parallelism available in modern GPUs. The proposed implementation delivers its maximum throughput of about 20–25 MB/s only for small subscription counts (up to 512 subscriptions).

In this implementation, however, the GPU accelerator does *not* operate on raw XML input data (as *XLynx* does). Rather, a CPU-based pre-processor "compresses" the XML input into a representation where only opening and closing XML tags are preserved, each encoded as a one-byte "entry" (one bit for opening/closing information, seven bits for a tag name id; tag names are limited to 128 distinct tag names). In effect, for the given benchmark data (DBLP), the GPU "sees" only 1⁄21 of the raw data stream—resulting in an actual throughput of only about 1 MB/s of raw XML on the GPU.

A similar design would significantly simplify also the design of *XLynx*. In particular, the availability of tag identifiers should reduce logic and memory consumption by significant amounts. At the same time, there is no reason why one-byte "entries" would lead to a reduced clock frequency (in fact, the simplified logic would likely improve the clock rate), so likely *XLynx* would sustain 180 MB/s when operating on one-byte "entries"—or close to 4 GB/s in the metric of Moussalli et al. [2011].[9]

A key challenge in XML processing, parsing in particular, is *context dependence*. Graphics processors, however, depend on very pure *data parallelism* to leverage their SIMT-based execution engine. Generally, it is not clear whether GPUs are a good execution platform for workloads like XML/XPath.

Index-Based XQuery Processing. The target of *XLynx* clearly are streaming scenarios, where no access structures (indexes) can be constructed over the data ahead of time. Index structures—if chosen and generated properly—can accelerate a known query workload by several factors, unbeatable by any mechanism that cannot preprocess its data. Nevertheless, it is interesting to relate the performance improvement through hardware acceleration to the execution characteristics of well-engineered disk-based engines.

[9] 4 GB of raw XMark data contain about 180×10^6 XML tags and thus compress to about 180×10^6 one-byte "entries," which could be processed in one second using a clock frequency of 180 MHz.

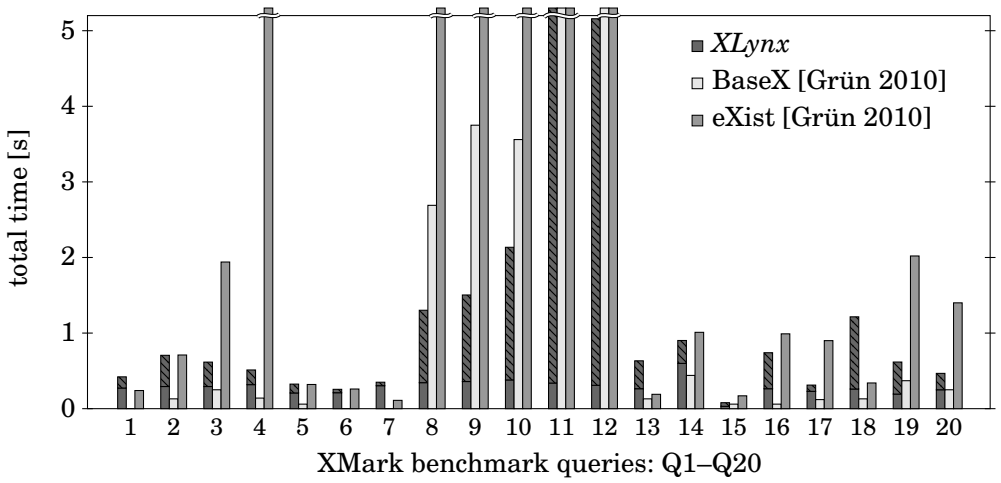


Fig. 29. Performance of a Saxon instance that operates on data prefiltered with *XLynx*, versus execution times reported for BaseX and eXist. Numbers for the latter two systems taken from [Grün 2010]. Some numbers exceed scale.

Such a comparison is illustrated in Figure 29. In this figure, we show the performance of an *XLynx*-accelerated Saxon instance side-by-side with execution times reported for BaseX (http://www.basex.org/) and eXist (http://www.exist-db.org/). Numbers for the latter two systems were taken from [Grün 2010]. They were measured on dual core T7300 Intel CPU.

9. MORE RELATED WORK

After Marian and Siméon [2003] proposed the concept of XML projection, the idea was expanded into different directions by the research community.

On the path evaluation side, Koch et al. [2008] suggested an interesting alternative to the automaton-based path matching, as discussed in Section 2.2. The key insight is the problem's similarity to *string matching*. This allows the use of string matching algorithms that have proven efficient for the matching task, such as the classical Boyer-Moore algorithm [Boyer and Moore 1977] or—to match sets of paths—the string matching algorithm of Commentz-Walter [1979]. The ideas of Koch et al. are similar to our work in the sense that they exploit specific characteristics of the XPath matching problem. But unlike our work, their approach depends on in-memory pointer navigation, which is contrary to the truly stream-oriented processing model of our system.

The work of Benzaken et al. [2006] primarily improves the query analysis part. Their proposed *type-based XML projection* looks at type information rather than plain `child`/`descendant` paths. This allows building a more selective projection filter, which further reduces the size of the projected XML document. In the runtime part, Benzaken et al. push much of the matching complexity into *type annotation* as a preprocessing step to the actual projection. Type annotation again can be implemented with the help of finite-state automata and, therefore, could be realized using skeleton automata much as we described it in this paper. The effectiveness of type-based projection on filtering selectivity is data-dependent: the concept should benefit hard- and software implementations alike.

FPGAs are an increasingly attractive alternative to overcome the architectural limitations of commodity hardware. Commercial systems like IBM/Netezza [Netezza 2012] as well as a number of research prototypes [Moussalli et al. 2010; 2011; Mueller et al.

2009; Sadoghi et al. 2010; Sadoghi et al. 2011; Woods et al. 2010] demonstrate this for a wide range of use cases.

All these systems were forced to compromise between query expressiveness and interactivity. On one end of the spectrum, systems like Netezza provide full interactivity, but can use their FPGAs for only very basic operations (such as selection and projection). Others (such as most of the research prototypes) opted for the opposite extreme. They offer much higher expressiveness, but at the cost of very high compilation overhead for each user query. The work of Sadoghi et al. [2010; 2011] stands in the middle and explicitly analyzes the existing trade-offs. For the same use case (publish/subscribe for algorithmic trading), they propose different FPGA implementations that are tuned for (and named) "flexibility," "adaptability," "scalability," or "performance." The reported trade-offs are significant: "performance" is about 70 $\times$ faster than "flexibility," but requires expensive hardware re-compilation for every workload change (Sadoghi et al. [2011] do not report compilation times; they usually range from several minutes to hours).

The focus of our work is *not* to make any compromises. Rather, we support XML and a rich subset of XPath, and yet also offer micro-second reactivity to query workload changes.

We believe the skeleton automaton idea could be combined also with existing approaches, for instance on hardware-accelerated SQL processing. Netezza [Netezza 2012] offers selection and projection predicates that can be parameterized for a given query, but lacks flexibility to construct complex predicates. *Glacier*, our own prototype with per-query circuit construction [Mueller et al. 2009], allows for such predicates, but requires expensive re-compilation. Skeleton automata could be used to construct complex networks of Netezza-style operators, thus allow for complex predicates without expensive re-compilation.

Many FPGA solutions face the trade-off between flexibility and performance. *High-frequency trading (HFT)*, for instance, is a race for ultra-low latency [Schneider 2012]. To minimize latency, many developers tend toward building new, tailor-made circuits for each application; but the competitive market does not allow long development cycles to build these circuits. Lockwood et al. [2012] proposes to counter the problem with help of an *IP (intellectual property) library* with pre-built components for individual tasks in the application domain.

Pre-built libraries can also be used to implement faster workload updates by exploiting the *partial re-configuration* capabilities of modern FPGA chips.[10] Dennl et al. [2012] showed how this idea can be used to improve the flexibility of a *Glacier*-like query processing system (*Glacier* is our own prototype of an execution platform with per-query compilation [Mueller et al. 2009]). Partial re-configuration is appealing to solve the flexibility/performance trade-off. But its use brings in another level of complexity into the development process (such as additional tools needed at runtime), which so far has kept system makers from using partial reconfiguration in real-world systems.

One way of looking at *XLynx* is that it leverages the XML projection idea of Marian and Siméon [2003] and hardware-based parsing to reduce the high *parsing cost* that bottlenecks many real-world XML processors. XML parsing has been studied separately by Dai et al. [2010]. Their system, *XML Parsing Accelerator (XPA)*, reaches similar throughput rates as our input parser. In addition to our work, however, XPA also includes facilities to build up (DOM-based) in-memory data structures that could

[10]Using partial re-configuration, parts of an FPGA chip can be updated at runtime, rather than stopping and re-loading the entire FPGA chip.

directly be handed over to a software XML processor. Given the modular designs of both *XLynx* and XPA, we think that those facilities could also be integrated into *XLynx*, completely avoiding the serialization/parsing cycle along the FPGA $\rightarrow$ CPU path.[11]

Several research projects have addressed the parsing of XML in software-only systems, by leveraging available hardware parallelism in the form of SIMD (vector-oriented processing) or multi-core processors. The crux in parallelizing the parsing task is context dependence. Pan et al. [2007; 2008] thus suggested to *pre-parse* the XML input into a *skeleton* where suitable boundaries for document *partitioning* are explicitly identified. After parsing these partitions in parallel, a simple post-processing steps suffices to integrate partial results into an overall DOM tree. Pan et al. [2008] report an almost perfect scaling of this *PXP* strategy to at least 30 CPU cores, though no absolute numbers are given for the achieved parsing speed.

Parabix leverages SIMD functionality in modern processors to identify the key syntactic elements of XML (*e.g.*, angle brackets </> or ampersands &) in a parallel fashion. As such, we think that Parabix might not only be attractive for standalone XML parsing, but also for use as a pre-parser in setups like PXP.

Shah et al. [2009] demonstrated that pre-parsing can be avoided by exploiting known *numbering schemes* for XML. Independent threads create, for arbitrary chunks of input data, local *preorder ranks* for all nodes in the chunk. By exchanging parsing stacks between threads, the overall DOM tree can afterward be obtained through a lightweight post-processing step. Shah et al. [2009] report a speedup of around 2.5 for 4 CPU cores, with absolute performance in the order of 100–160 MB/s (*i.e.*, close to the 180 MB/s that we achieve in hardware).

The (de)fragmentation issues discussed in Section 6 resemble the problem of free space management experienced in operating systems [Wilson et al. 1995], databases [McAuliffe et al. 1995], or file systems. However, we avoid many of the problems that affected OS, database, or file system implementors in the past. In particular, in *XLynx* we are free to move segments after allocation, which is in contrast to main memory allocation or record-id allocation. Further, copying segment contents to a new destination does *not* cause any overhead or slowdown, as experienced by file systems. Rather, the copying work is performed only by segments that would be idle otherwise. Once again this adds flexibility without deteriorating processing performance.

10. SUMMARY

To avoid the critical trade-off between query expressiveness and the capability for ad-hoc querying, we propose a new implementation strategy for FPGA-based database accelerators. Rather than building hard-wired circuits for only narrow query types, we statically compile a *skeleton automaton* that can be configured at runtime to implement query-dependent state automata. The so-constructed and configured automata run as fast as existing hard-wired automata, yet offer high expressiveness and complexity (*e.g.*, hundreds of parallel XPath steps on one low-end chip).

Our use case for this work is *XML projection*, a method that has proven effective to reduce processing and main-memory overhead of XML processors. As such, we make the architectural advantages (*e.g.*, in-network processing); the lower energy consumption; and the performance benefits of FPGAs accessible to XML processing. We demonstrated all three aspects with a micro-benchmark of the core projection engine (*XLynx*) and by pairing our system with a state-of-the-art XQuery processor (Saxon Enterprise Edition).

[11]As a down side, we would lose the *back-end independence* of *XLynx*, since the DOM representation will be a back-end-dependent data structure.

On the micro-architectural side, the skeleton automaton design principle scales favorably with the available chip space, making our work ready for upcoming chip generations that will provide significantly more real estate. The XML projection that we propose runs at throughput rates of about 180 MB/s of XML input—more than enough to realize "in-network filtering," a scenario that we used to exemplify our approach.

On the full system scale, *XLynx* leads to significant savings (up to 94 %) of electrical power consumption, hence we address the key limitation in modern system designs. In-network filtering with *XLynx* significantly eases the *XML parsing burden* on the back-end XML processor. Since parsing is the main bottleneck for many real-world scenarios, reducing the parsing cost directly translates to an overall speedup of the total query execution time. The effect is independent of the XML processor used as *XLynx*'s back-end and leads to a performance improvement of several factors with Saxon-EE as the back-end system.

The role of FPGAs in a complete system is part of our ongoing research project *Avalanche*. In this project, we currently are working on strategies to leverage the potential of FPGAs in hybrid FPGA/CPU system designs. *XLynx* is one example, designed to integrate in a complete end-to-end system. In this article, we illustrated how the skeleton principle elegantly interplays with dynamic query additions and removals and how resource allocation can be implemented in a dynamic fashion. *XLynx* consumes and produces real XML data. It can thus be paired with arbitrary back-end processors.

REFERENCES

Mehmet Altınel and Michael J. Franklin. 2000. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the 26th Int'l Conference on Very Large Data Bases (VLDB)*. Cairo, Egypt.

Véronique Benzaken, Giuseppe Castagna, Dario Colazzo, and Kim Nguy˜ên. 2006. Type-Based XML Projection. In *Proc. of the 32nd Int'l Conference on Very Large Data Bases (VLDB)*. Seoul, Korea.

Shekhar Borkar and Andrew A. Chien. 2011. The Future of Microprocessors. *Commun. ACM* 54, 5 (May 2011), 67–77.

Irina Botan, Peter M. Fischer, Daniela Florescu, Donald Kossmann, Tim Kraska, and Rokas Tamosevicius. 2007. Extending XQuery with Window Functions. In *Proc. of the 33rd VLDB Conference*. Vienna, Austria.

Robert S. Boyer and J. Strother Moore. 1977. A Fast String Searching Algorithm. *Commun. ACM* 20, 10 (Oct. 1977), 762–772.

Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, Fraçois Yergeau, and John Cowan. 2006. Extensible Markup Language (XML) 1.1 (Second Edition). (Sept. 2006). W3C Recommendation.

Robert D. Cameron, Kenneth S. Herdy, and Dan Lin. 2008. High Performance XML Parsing Using Parallel Bit Stream Technology. In *Proc. of the 2008 Conference of the Centre for Advanced Studies on Collaborative Research (CASCON)*. Richmond Hill, ON, Canada, 17.

Beate Commentz-Walter. 1979. A String Matching Algorithm Fast on the Average. In *Proc. of the 6th Int'l Colloquium on Automata, Languages and Programming (ICALP)*. Graz, Austria.

Zefu Dai, Nick Ni, and Jianwen Zhu. 2010. A 1 Cycle-Per-Byte XML Parsing Accelerator. In *Proc. of the ACM/SIGDA 18th Int'l Symposium on Field-Programmable Gate Arrays (FPGA)*. Monterey, CA, USA, 199–208.

André DeHon. 1999. Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100 % LUT utilization). In *Proc. of the Int'l Symposium on Field Programmable Gate Arrays (FPGA)*. 125–134.

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

Yanlei Diao, Mehmet Altınel, Michael J. Franklin, Hao Zhang, and Peter Fischer. 2003. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Transactions on Database Systems (TODS)* 28, 4 (Dec. 2003), 467–516.

Mary F. Fernández, Jérôme Siméon, Byron Choi, Amélie Marian, and Gargi Sur. 2003. Implementing XQuery 1.0: The Galax Experience. In *Proc. of the 29th Conference on Very Large Data Bases (VLDB)*. Berlin, Germany, 1077–1080.

Hubertus Franke, J. Xenidis, Claude Basso, Brian M. Bass, Sandra S. Woodward, Jeffrey D. Brown, and Charles L. Johnson. 2010. Introduction to the Wire-Speed Processor and Architecture. *IBM Journal of Research and Development* 54, 1 (2010), 3:1–3:11.

Christian Grün. 2010. *Storing and Querying Large XML Instances*. Ph.D. Dissertation.

Michael Kay. 2008. Ten Reasons Why Saxon XQuery is Fast. *IEEE Data Eng. Bull.* 31, 4 (2008), 65–74.

Christoph Koch, Stefanie Scherzinger, and Michael Schmidt. 2008. XML Prefiltering as a String Matching Problem. In *Proc. of the 24th Int'l Conference on Data Engineering (ICDE)*. Cancún, Mexico.

Ian Kuon and Jonathan Rose. 2007. Measuring the Gap Between FPGAs and ASICs. *IEEE Trans. on Computer-Aided Design of Integrated Circuits and Systems* 26, 2 (Feb. 2007).

Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. 2005. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. In *Proc. of the 31st Int'l Conference on Very Large Data Bases (VLDB)*. Trondheim, Norway.

Michael Leventhal and Eric Lemoine. 2009. The XML Chip at 6 Years. In *Int'l Symposium on Processing XML Efficiently: Overcoming Limits on Space, Time, or Bandwidth*. Montréal, Canada.

John W. Lockwood, Adwait Gupte, Nishit Mehta, Michaela Blott, Tom English, and Kees Vissers. 2012. A Low-Latency Library in FPGA Hardware for High-Frequency Trading (HFT). In *Proc. of the IEEE 20th Annual Symposium on High-Performance Interconnects*. 9–16.

Amélie Marian and Jérôme Siméon. 2003. Projecting XML Documents. In *Proc. of the 29th Int'l Conference on Very Large Data Bases (VLDB)*. Berlin, Germany.

Mark L. McAuliffe, Michael J. Carey, and Marvin H. Solomon. 1995. Towards Effective and Efficient Free Space Management. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD 1995)*. Montreal, QC, Canada, 389–400.

Roger Moussalli, Robert Halstead, Mariam Salloum, Walid Najjar, and Vassilis J. Tsotras. 2011. Efficient XML Path Filtering Using GPUs. In *Proc. of the 2nd Int'l Workshop on Accelerating Data Management Systems Using Modern Processor and Storage Architectures (ADMS)*. Seattle, WA, USA.

Roger Moussalli, Mariam Salloum, Walid A. Najjar, and Vassilis J. Tsotras. 2010. Accelerating XML Query Matching through Custom Stack Generation on FPGAs. In *Proc. of the 5th Int'l Conference on High-Performance Embedded Architectures and Compilers (HiPEAC)*. Pisa, Italy, 141–155.

Roger Moussalli, Mariam Salloum, Walid A. Najjar, and Vassilis J. Tsotras. 2011. Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In *Proc. of the 27th Int'l Conference on Data Engineering (ICDE)*. Hannover, Germany, 948–959.

Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Data Processing on FPGAs. *Proc. of the VLDB Endowment (PVLDB)* 2, 1 (Aug. 2009).

Netezza 2012. Netezza. (2012). http://www.netezza.com/.

Matthias Nicola and Jasmi John. 2003. XML Parsing: A Threat to Database Performance. In *Proc. of the 12th Int'l Conference on Information and Knowledge Management (CIKM)*. New Orleans, LA, USA, 175–178.

Yinfei Pan, Wei Lu, Ying Zhang, and Kenneth Chiu. 2007. A Static Load-Balancing Scheme for Parallel XML Parsing on Multicore CPUs. In *Proc. of the 7th Int'l IEEE Symposium on Cluster Computing (CCGRID)*. Rio de Janeiro, Brazil, 351–362.

Yinfei Pan, Ying Zhang, and Kenneth Chiu. 2008. Simultaneous Transducers for Data-Parallel XML Parsing. In *Proc. of the 22nd Int'l IEEE Symposium on Parallel and Distributed Processing (IPDPS)*. Miami, FL, USA, 1–12.

Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. 2010. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. *Proc. of the VLDB Endowment (PVLDB)* 3, 2 (Sept. 2010).

Mohammad Sadoghi, Harsh Singh, and Hans-Arno Jacobsen. 2011. Towards Highly Parallel Event Processing through Reconfigurable Hardware. In *Proc. of the 7th Int'l Workshop on Data Management on New Hardware (DaMoN)*. Athens, Greece.

Albrecht R. Schmidt, Florian Waas, Martin L. Kersten, Michael J. Carey, Ioana Manolescu, and Ralph Busse. 2002. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th Int'l Conference on Very Large Data Bases (VLDB)*. Hong Kong, China, 974–985.

David Schneider. 2012. The Microsecond Market. *IEEE Spectrum* 49, 6 (June 2012), 66–81.

Bhavik Shah, Praveen R. Rao, Bongki Moon, and Mohan Rajagopalan. 2009. A Data Parallel Algorithm for XML DOM Parsing. In *Proc. of the 6th Int'l XML Database Symposium on Database and XML Technologies (XSym)*. Lyon, France.

Reetinder Sidhu and Viktor Prasanna. 2001. Fast Regular Expression Matching Using FPGAs. In *IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*. Rohnert Park, CA, USA.

Jens Teubner and Louis Woods. 2011. Snowfall: Hardware Stream Analysis Made Easy. In *Proc. of the 14th Conference on Databases in Business, Technology, and Web (BTW)*. Kaiserslautern, Germany.

Jens Teubner, Louis Woods, and Chongling Nie. 2012. Skeleton Automata: Reconfiguring without Reconstructing. In *Proc. of the ACM SIGMOD Conference on Management of Data (SIGMOD 2012)*. Scottsdale, AZ, USA, 229–240.

Jan van Lunteren. 2001. Searching Very Large Routing Tables in Wide Embedded Memory. In *Proc. of the IEEE Global Telecommunications Conference (GLOBECOM'01)*, Vol. 3. San Antonio, TX, USA, 1615–1619.

Jan van Lunteren, Ton Engbersen, Joe Bostian, Bill Carey, and Chris Larsson. 2004. XML Accelerator Engine. In *Proc. of the 1st Int'l Workshop on High-Performance XML Processing*. New York, NY, USA.

Paul R. Wilson, Mark S. Johnstone, Michael Neely, and David Boles. 1995. Dynamic Storage Allocation: A Survey and Critical Review. *Lecture Notes in Computer Science* 986 (1995), 1–116.

Louis Woods, Jens Teubner, and Gustavo Alonso. 2010. Complex Event Detection at Wire Speed with FPGAs. *Proc. of the VLDB Endowment (PVLDB)* 3, 1 (Sept. 2010).

Xilinx Inc. 2011. 7 Series FPGAs Overview. (Sept. 2011).

Yi-Hua E. Yang, Weirong Jiang, and Viktor K. Prasanna. 2008. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *ACM/IEEE Symp. on Architectures for Networking and Communication Systems (ANCS)*. San Jose, CA, USA, 30–39.