Skeleton Automata for FPGAs: Reconfiguring without Reconstructing

Jens Teubner
jens.teubner@inf.ethz.ch

Louis Woods
louis.woods@inf.ethz.ch

Chongling Nie
cnie@student.ethz.ch

Systems Group, Department of Computer Science, ETH Zurich, Switzerland

ABSTRACT

While the performance opportunities of *field-programmable gate arrays (FPGAs)* for high-volume query processing are well-known, system makers still have to compromise between desired query expressiveness and high compilation effort. The cost of the latter is the primary limitation in building efficient FPGA/CPU hybrids.

In this work we report on an FPGA-based stream processing engine that does not have this limitation. We provide a hardware implementation of *XML projection* [14] that can be reconfigured in less than a micro-second, yet supports a rich and expressive dialect of XPath. By performing XML projection *in the network*, we can fully leverage its filtering effect and improve XQuery performance by several factors.

These improvements are made possible by a new design approach for FPGA acceleration, called *skeleton automata*. Skeleton automata separate the structure of finite-state automata from their semantics. Since individual queries only affect the latter, with our approach query workload changes can be accommodated fast and with high expressiveness.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

FPGA, XML, XQuery, Projection, Skeleton Automaton

1. INTRODUCTION

With the looming end of microprocessor scaling [4], makers of databases are starting to prepare and explore *field-programmable gate arrays (FPGAs)* as an escape from the inherent limitations of commodity hardware. Systems such as *IBM Netezza* [18], *Glacier* [17], or *fpga-ToPSS* [20, 21] have demonstrated exceptional throughput rates while keeping energy consumption low.

Processing queries on bare hardware is compelling. But building tailor-made circuits is a complex task that has forced major concessions in all of the existing systems. On one end of the spectrum, systems like *Glacier* compile full SQL query expressions into dedicated hardware, a strategy that offers high flexibility and maximum processing throughput. But query compilation is highly CPU-intensive, and compile times of several hours are not an exception. The other extreme are systems like Netezza, where quick compile times have to be paid for with very rigid limitations on query expressiveness (selections and projections only).

In this work, we do *not* want to trade expressiveness for speed. Rather, the *skeleton automata* design technique that we propose enables interactive querying for a language as complex as *XPath*. Automata generated this way can operate at full Gigabit Ethernet line rate, but do not need hours of query compilation like existing approaches (*e.g.*, [15, 17]).

More specifically, we use skeleton automata here to implement *XML projection* [14] in hardware. The effectiveness of XML projection—essentially input filtering based on XPath expressions—is well-known. But in existing work, the benefit has always been off-set by the high XML parsing overhead that a software-based implementation would face.[1] By offloading XML projection to dedicated hardware, we avoid this cost and enable XML processing at true line rate.

We present our system as a closed-box solution that transparently filters XML data streams *in the network*. This way, our filter can be paired with any existing XQuery processor or act as a semantic firewall that strips off sensitive information as XML passes the network (in the spirit of [7]).

Like most interesting filtering tasks, XPath evaluation can be expressed using *finite-state automata*. Such automata are known to run very efficiently on FPGA hardware (*e.g.*, [23, 27]), but are also known to be very costly to compile. Placing and routing states and transitions on the FPGA chip is highly compute-intensive, resulting in the long compilation times that we mentioned already.

A skeleton automaton *separates* the automaton's *structure*—which is the difficult part to compile—from its *transition conditions*. The latter can be realized as simple configuration parameters and altered even after the main automaton has been instantiated on the chip. Much of this automaton structure is determined alone by the XPath language semantics [3]. We exploit this observation and *statically* compile a skeleton automaton that can universally represent any legal projection path query. At runtime, we only modify the automaton's transition conditions, which can be performed in a micro-second or less.

[1] XML parsing is highly CPU-intensive, to the extent that it has been considered a "threat to database performance" [19].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
SIGMOD'12, May 20–24, 2012, Scottsdale, Arizona, USA.
Copyright 2012 ACM 978-1-4503-1247-9/12/05 ...$10.00.

Contributions. Our main contribution is a new approach to FPGA-assisted database processing. We leave behind the typical query-to-hardware compilation strategy that incurs very high compilation overhead in existing work. Instead, we offer truly interactive query workload modifications, even for tasks as difficult as XML processing. This report discusses the new "skeleton automaton" approach, but also contains enough implementation details to reproduce our work and adopt it to other use cases.

In addition, our work puts automaton-based XPath evaluation and XML projection into a new perspective. By performing projection in hardware, we are the first to actually benefit from the filtering technique that had been discovered already years ago. Our observations on the structure of automata for XPath evaluation may further guide others toward new execution strategies also in software systems.

This paper is structured as follows. Sections 2 and 3 give the necessary background on XML projection and FPGAs. Our main contributions are covered in Sections 4 and 5, where we detail the skeleton automaton concept and its runtime (re)configuration, respectively. Sections 6 and 7 discuss optimization techniques and evaluate our system. Section 8 relates our work to others', before we wrap up in Section 9.

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 [22] data:

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

$$(Q_1)$$

This query looks up all auction items[2] 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 [14]). In our example this is needed to include full `name` elements into the query result.

[2]xmlgen (the XMark data generator) produces XML documents that model an auction website.

```
<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>
```

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

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 [14] for details.

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.

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

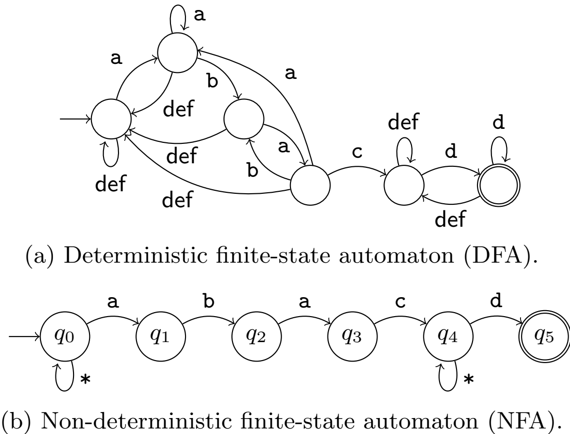
Figure 2: Supported dialect for projection paths.

Figure 2 lists the XPath dialect that our hardware implementation supports. This dialect essentially covers all features of the projection path language in [14] (we do not support namespaces at this point, however). Although our prototype includes experimental support for attribute-based filter predicates, which may even further increase filter selectivity, in this text we will not further elaborate on attribute-based filter predicates and all of our experiments were conducted without this enhancement.

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::`').

2.2 Path Evaluation (Previous Work)

For evaluation, projection paths are often viewed as *regular expressions*, evaluated over each node's path starting



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

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

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

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 ↻* 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 $\langle \mathit{state}, \mathit{symbol} \rangle \mapsto \mathit{state}$ lookup per input symbol). XFilter [1], 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.

On the flip side, non-deterministic finite-state automata are significantly easier to construct and maintain. In YFilter [9], this allowed the use of a *single* non-deterministic finite-state automaton that simultaneously matches all registered input queries. The automaton structure is changed whenever a query is (un)registered.

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 tag is encountered.

Hardware Acceleration. Finite-state automata can be implemented very efficiently in hardware (more details later). In [15, 16], this was used by Moussalli et al. to implement 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 [27] 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 [25] 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 [26]). 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.

In this work we can have our cake and eat it, too. 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

Virtually any hardware circuit consists of the same two 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*, *e.g.*, flip-flop registers, which 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.

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 actual 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*[3] by defining *(a)* the logic function f for each lookup table and *(b)* the signal wiring in the on-chip *interconnect network*.

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 the FPGA. Rather, we compile and upload a generic

[3]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."

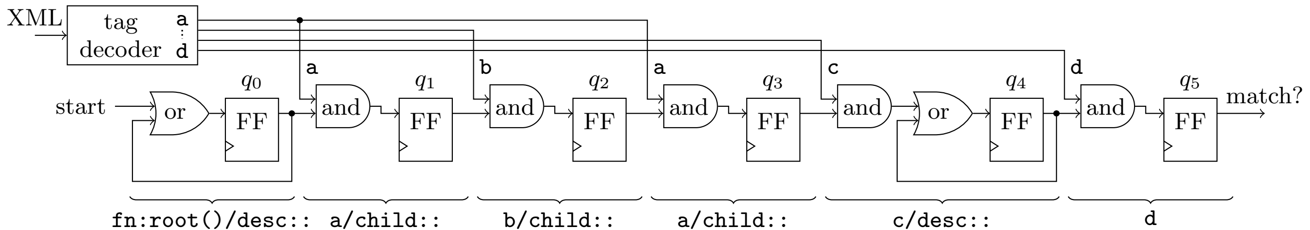


Figure 4: Hardware implementation of the non-deterministic finite-state automaton in Figure 3(b).

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. In [13], the authors 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.

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. ↻∗ 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. States q_0 through q_5 are pushed/popped to/from this stack during start/end element events then (refer to [15, 16] for details).

Though slightly simplified, the described procedure quite well describes the state of the art in hardware-based pattern matching. Optimized construction algorithms for FPGA targets exist (*e.g.*, [29]) 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.

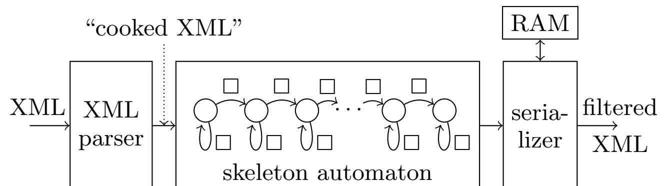


Figure 5: XML projection engine. After *parsing*, the XML stream passes through a *skeleton automaton*, which controls what the *serializer* emits as the projection result.

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

Figure 5 illustrates how the skeleton automaton participates in a complete XML filtering data flow. Raw XML data enters the system at the left end of the figure, where an *XML parser* analyzes the syntactical structure of the stream. Enriched with parsing information ("cooked"), the XML stream passes through the configured automaton, which determines a match state for each item in the stream. Finally, the *serializer* at the right end of the figure copies matches to the circuit output and ensures a well-formed XML result.

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. Much like a SAX parser in the software world, this parser identifies lexical elements in an input stream. While doing so, the parser *annotates* the raw XML input stream with a `token` field that makes the lexical structure of the stream accessible to subsequent processing units. We refer to an XML stream with `token` annotations as a *cooked* XML stream.

The behavior of the XML parser component is illustrated in Figure 6 as a *timing diagram*. The `token` signal carries values of an enumeration type, whose symbolic names we listed at the bottom of the figure.

We implemented the XML parser in our system with the help of the *Snowfall* hardware parser generator tool [24]. *Snowfall* reads in the grammar of any (regular) language,

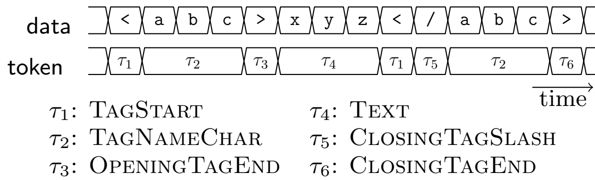


Figure 6: Timing diagram of XML parser output. The XML stream is enriched with a token signal to make lexical information explicit.

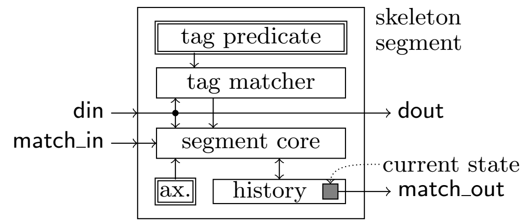


Figure 7: Hardware implementation of a single skeleton segment. ☐ blocks hold configuration parameters (axis and node test).

computes the corresponding finite-state automaton, and emits a VHDL specification of a hardware circuit that recognizes that language. Here we provided *Snowfall* with a slightly modified version of the XML language specification.

To the cooked 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 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, the document filter must preserve `site`, `regions`, and `africa` elements in Figure 1, 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.

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

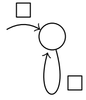
4.4 Skeleton Automaton

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.

4.4.1 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. Additional parameters, omitted here for ease of presentation, determine whether a state is accepting or help us correctly handle some specifics of XPath (see later).



Skeleton segments are connected to form a chain much like 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 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.

4.4.2 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 like 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 ↻☐ (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 had already indicated earlier on the bottom of Figure 4.

4.4.3 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 7, 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 ↻* loops in the effective automaton.

Like in the traditional scheme, the actual automaton state, which is part of each segment, is implemented using a flip-flop register. In Figure 5, this register is illustrated as a

```
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);
```

Algorithm 1: Pseudo code for segment core.

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 (currently set to 16 in our implementation). Cases where this depth is exceeded by a given XML instance will still not fail, because deeper document fragments can always be passed on to the software side and handled 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 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.[4] 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 ↻∗ 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.

4.4.4 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).

Two fundamental problems render this approach 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 7), which keeps signal lengths short and independent of the overall circuit size.

A consequence is that each tag matcher has to watch out for exactly one tag predicate (a tag name or a node test). Rather than building an automaton that could recognize a set of tag names, we can now implement tag matching as a simple string comparison circuit. This simplifies the hardware implementation and allows for higher clock speeds.

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.

```
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);
```

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

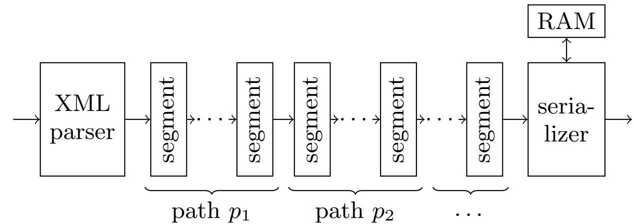


Figure 8: Multiple paths can be matched within a single processing chain.

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.

4.5 Matching Multiple Paths

Besides maintaining its own match state, each skeleton segment passes the (cooked) 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 8 illustrates the idea. As the XML input is streamed

[4] For ease of presentation we simplified to only child or descendant axes.

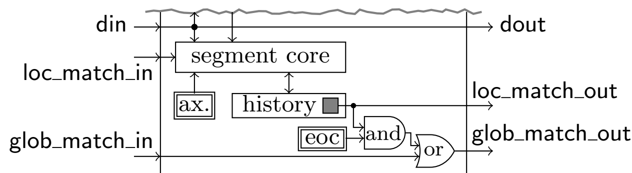


Figure 9: Match merging to support multiple projection paths. Local matches are merged into the global match state if the parameter *end-of-chain-section* (eoc) is set.

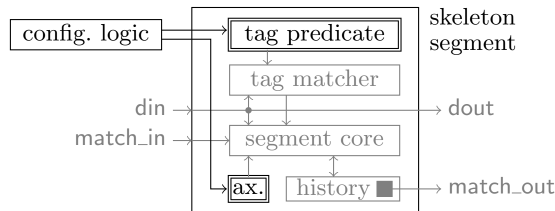


Figure 10: Configuration logic changes workload parameters outside the main processing and data path.

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] $\equiv$ 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 low 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 $\circlearrowright$∗ transition if it is configured to do so (*i.e.*, if `fn:root()` is followed by a `descendant` step; line 4 in Algorithm 1).

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 9 illustrates how match merging can be realized with only 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 utilize the same circuit to match either many short paths or fewer paths that are very long. In either case, the number of segments n provisioned in the skeleton automaton limits the amount of projection paths that can be processed simultaneously. The total number of steps in all paths must not exceed n. To illustrate, the twenty XMark queries that we look at in Section 7 use projection path sets with 3–15 paths per benchmark query (median: 4). In total, each query requires between $n = 7$ and $n = 79$ (median: $n = 15$) path steps.

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.

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 available capacities 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 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. By default, we allocate one BRAM block for each skeleton segment but we will shortly see how resource utilization and circuit performance can be improved if this allocation strategy changes.

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 10, separate *configuration logic* can maintain both configuration parameters without interfering with the processing logic.

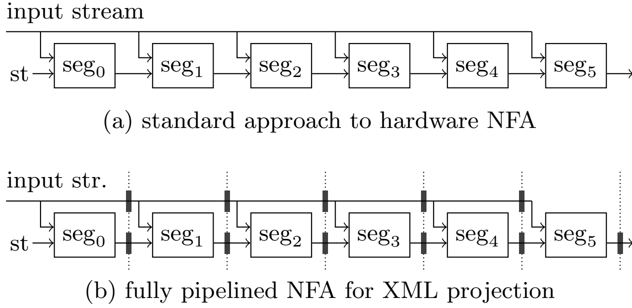


Figure 11: Standard hardware NFA implementation (top) requires long signal paths. Pipelining (bottom) reduces signal paths by inserting *registers* ▌.

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 4.4.2), compiling input queries and inferring parameter values is simple enough to be performed directly on the FPGA chip.

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. More specifically, our system can be configured via *XML processing instructions* that can be injected into the input XML data stream. For instance, the processing instruction

```
<?query fn:root()/descendant::item/child::name #?>
```

registers the new projection path `//item/name` in the engine.

Workload changes become effective immediately and will be considered for any data that follows the processing instruction in the input stream. The time needed by the processing instruction within the XML stream may thus be interpreted as the workload reconfiguration time. The 50-byte processing instruction above, for instance, requires 50 FPGA clock cycles to be processed, or 300 ns at an FPGA clock speed of 166 MHz.

6. 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)* be economic on *chip space* (to support real-world problem sizes at low cost). In this work we use *pipelining* and *BRAM sharing* to address both aspects.

6.1 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 11(a) emphasizes the same concept but hides the inner details of circuit segments seg_i.

Figures 4 and 11(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.*

NFAs for XML Projection. When arbitrary automata shapes must be supported, long signal paths are inevitable, since a new value of a state q_i might depend on any other state q_j. Non-deterministic finite-state automata 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 11(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.

With the registers in place, 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 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 our System. Pipelining is particularly effective in FPGA designs. In FPGA hardware, each lookup table is co-located with a flip-flop register (together they are packed into so-called *slices*). Thus, by enabling those registers, throughput can be improved at very little cost (as the flip-flop register would remain unused otherwise).

In our system, we place a pipeline register after every skeleton segment. As we will see in the next section, this leads to signal lengths that are well within the range of clock frequencies that the FPGA hardware has been designed for (around 150–200 MHz).

XPath Semantics. At this point we would like to note an interesting side effect of pipelining to the semantics of XPath evaluation. Consistent with the original work on XML projection [14], 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 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 to make up for the missing clock cycle needed to implement the '`self`' functionality.

Existing automaton-based XPath engines either do not support `-self` axes at all (to our knowledge, no existing sys-

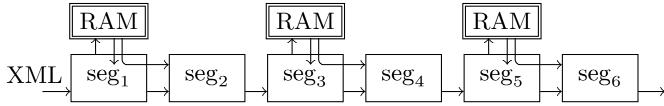


Figure 12: BRAM sharing. Two segments store their tag predicates in the same RAM block. Since each block has only one interface, segments seg_{2k-1} mediate traffic for segments seg_{2k}.

tem 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 ∧ 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. This tends to be even more expensive on FPGAs than it is in software-based systems and should thus be avoided whenever possible.

6.2 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 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 character 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 12 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.[5]

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 7, BRAM and logic resources are in balance on our hardware when three segments share one BRAM unit.

7. EVALUATION

We implemented and tested our system 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 × 18 kbit BRAM) and has a number of I/O connectors to communicate with outside systems. In the following Section 7.1 we first characterize the core XML projection engine, before in Section 7.2 we show how the engine could be used in a working system.

[5]The maximum word size for each BRAM block is 36 bits. Up to four segments can thus share one BRAM block by simply merging their 8-bit data into one large word.

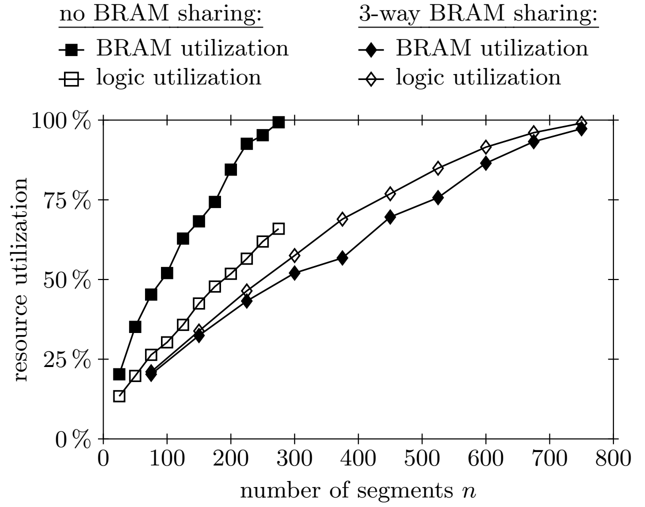


Figure 13: FPGA chip resource consumption of various engine configurations. BRAM sharing allows to balance the use of logic and BRAM resources to obtain a larger overall engine size.

7.1 Core Engine

To analyze the characteristics of our core XML projection engine, 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 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 13 illustrates the utilization of BRAM units (filled markers) and logic blocks (slices; 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 6.2. Without BRAM sharing, all BRAM resources are used up for circuit configurations beyond ≈ 275 segments (while more than 1/3 of the available logic resources are unused).

BRAM sharing can bring resource utilization into balance. With 3-way BRAM sharing (diamond symbols in plot), the maximum number of segments is now limited by logic resources (lookup tables for that matter) and we can instantiate up to 750 segments on our chip, *i.e.*, we can support more than two times as many concurrent projection paths.

Scalability. To evaluate the scalability criterion, we used the FPGA design tools to determine the maximum *clock*

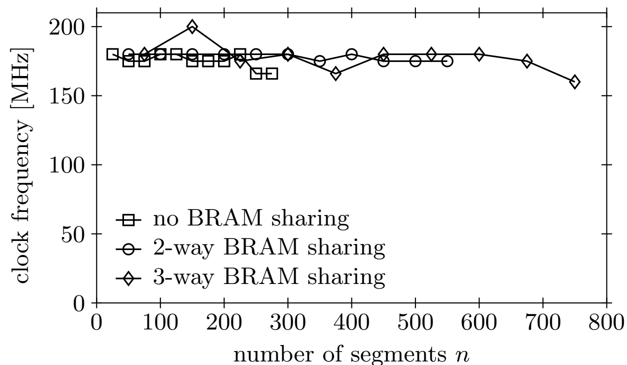


Figure 14: Maximum clock frequency for various engine configurations. Frequency is not strongly influenced by circuit size.

frequency at which each of our engine configurations could be operated.[6] Figure 14 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 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.

The clock frequencies shown in Figure 14 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 [8]. 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 [28].

7.2 XML Projection in the Network

FPGAs may offer significant advantages over software-based systems in terms of performance and/or power consumption.[7] Their main benefit, however, lies in *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.

In the resulting system, the client will not only benefit from *reduced memory* overhead during query processing (which was the main incentive in [14]). Moreover, filtering

[6] 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, 160, 166, 175, 180 MHz, 200 MHz, and 225 MHz).

[7] The Xilinx Power Analyzer tool reports a power consumption of less than 3 Watts for our projection engine.

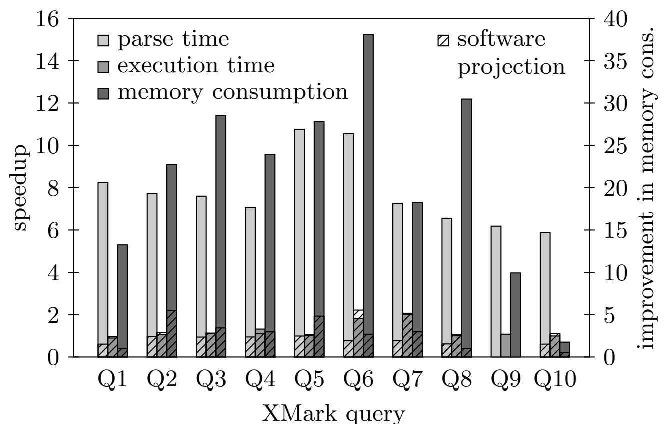


Figure 15: Speedup and improvement in memory consumption due to XML projection for the first ten XMark queries. Software projection for Q9 failed.

in the network also saves much of the *parsing cost*, which is an important cost factor in typical XML applications [19].

We verified this on the basis of Saxon-EE (version 9.4.0.3), a state-of-the-art XQuery processor for in-memory processing, and the XMark benchmark suite [22]. We used an XMark instance of scale factor 1 and 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 and Saxon performance independently (and ran all Saxon experiments from a memory-cached file).

Filtering Throughput. Our system 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, our system 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 limited by the physical network speed.

This was confirmed by the measurements we performed on real hardware. We observed a maximum payload throughput of 109 MB/s on a 110 MB XMark instance. 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).

Application Speedup. On the application side, in-network filtering mainly reduces parsing cost and memory consumption. On raw data, Saxon requires 2.23 sec for input parsing (independent of the query), which dominates the overall query answer time for most XMark queries (actual query execution times were 68 ms–41 sec; median: 390 ms). Main-memory consumption is query-dependent and amounted to 363–685 MB on our system (median: 518 MB).

Figure 15 illustrates the effect of in-network filtering for the first 10 XMark queries (gray bars). Parsing times and memory consumption are significantly reduced for all twenty queries. We measured parsing times of 31 and 599 ms (me-

dian: 283 ms), which correlates well with the filter selectivities of the individual queries. Filtering has even more effect on Saxon's main-memory consumption, which went down to 12–207 MB (median: 25.6 MB) with filtering. Both effects manifest themselves even for those queries that lead to a significant number of projection paths (cf. Section 4.5).

By contrast, filtering has much less effect on the actual query execution time (which excludes parsing cost). Here we measured 45 ms–18 s (median: 346 ms) after filtering, which is in line with previous reports on document projection in Saxon [11].

Figure 15 also puts in-network filtering in relation to software-based projection (shown as ▨), a feature of Saxon's enterprise edition.

Software-based projection, however, even slightly *increases* input parsing cost (now 2.3–3.7 sec; median: 2.36 sec). The evaluation of projection paths during input parsing causes additional CPU load that cannot be compensated by a reduced build-up cost for 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).

Interestingly, Saxon's software-based projection mechanism does not lead to the same memory savings as in-network filtering. We attribute this to garbage collection-based memory management (Saxon is written in Java). Intuitively, XML projection should reduce the in-memory tree sizes by the same amount, whether computed in hard- or software.

8. MORE RELATED WORK

After Marian and Siméon proposed the concept of XML projection in [14], the idea was expanded into different directions by the research community.

On the path evaluation side, Koch et al. [12] suggested an interesting alternative to the automaton-based path matching as we discussed in Section 2.2. The key insight is the problem's similarity to *string matching.* This allows the use of proven-efficient string matching algorithms for the matching task, such as the classical Boyer-Moore [5] algorithm or—to match sets of paths—the string matching algorithm of Commentz-Walter [6]. 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. [2] primarily improves the query analysis part. The 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 help of finite-state automata and, therefore, could be realized using skeleton automata much like the one that we described in this paper.

FPGAs are an increasingly attractive alternative to overcome the architectural limitations of commodity hardware. Commercial systems like IBM/Netezza [18], but also a number of research prototypes [15, 16, 17, 20, 21, 27] 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) went 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. [20, 21] 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 focus of our work is to not make any compromises. Rather, we support XML and a rich subset of XPath, yet offer micro-second reactivity to query workload changes.

9. SUMMARY

To avoid the critical trade-off between query expressiveness and capability for ad-hoc querying, we propose a new implementation strategy for FPGA-based database accelerators. Rather than building hard-wired circuits or templates 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 proven-effective method to reduce processing and main-memory overhead of XML processors. As such, we make the architectural advantages—like in-network processing—and performance benefits of FPGAs accessible to XML processing. We demonstrated both aspects with a micro-benchmark of the main projection engine and by pairing our system with a state-of-the-art XQuery processor.[8]

The system that we describe shows favorable scalability properties making our work ready for upcoming chip generations that will provide significantly more chip space.

Looking forward, we think that skeleton automata can play an important role in the quest for novel system designs that leverage (rather than suffer from) on-going development in hardware technology. Our goal for this line of work is to build a hybrid CPU/FPGA database engine that fully supports runtime resource optimization and ad-hoc querying.

Outside our main research interest, we think that some of the observations that we made in this paper—*e.g.*, automata for XML projection exhibit a very uniform structure—could be inspiring also for software-only systems. Similar observations have already lead, *e.g.*, to the use of string-matching techniques for XPath evaluation in the past [12].

Acknowledgements

This research is part of the *Avalanche* research project at ETH Zurich, funded by an *Ambizione* grant of the Swiss National Science Foundation and by the Enterprise Computing Center (ECC) of ETH Zurich (`http://www.ecc.ethz.ch/`).

[8]Skeleton automata are ready to use also in combination with the MXQuery engine [10].

10. REFERENCES

[1] Mehmet Altınel and Michael J. Franklin. 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, September 2000.

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

[3] Anders Berglund, Scott Boag, Don Chamberlin, Mary F. Fernández, Michael Kay, Jonathan Robie, and Jérôme Siméon. XML Path Language (XPath) 2.0 (Second Edition), December 2010. W3C Recommendation.

[4] Shekhar Borkar and Andrew A. Chien. The Future of Microprocessors. *Communications of the ACM*, 54(5):67–77, May 2011.

[5] Robert S. Boyer and J. Strother Moore. A Fast String Searching Algorithm. *Communications of the ACM*, 20(10):762–772, October 1977.

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

[7] IBM WebSphere DataPower SOA Appliance. `http://www.ibm.com/software/integration/datapower/`.

[8] André DeHon. 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)*, pages 125–134, February 1999.

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

[10] Peter Fischer and Jens Teubner. MXQuery with Hardware Acceleration. In *Proc. of the 28th Int'l Conference on Data Engineering (ICDE)*, Arlington, VA, USA, April 2012.

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

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

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

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

[15] Roger Moussalli, Mariam Salloum, Walid A. Najjar, and Vassilis J. Tsotras. 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)*, pages 141–155, Pisa, Italy, January 2010.

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

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

[18] Netezza. `http://www.netezza.com/`.

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

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

[21] Mohammad Sadoghi, Harsh Singh, and Hans-Arno Jacobsen. 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, June 2011.

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

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

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

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

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

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

[28] Xilinx Inc. 7 Series FPGAs Overview, September 2011.

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