FPGAs for Dynamic (XML) Query Workloads

ABSTRACT

While the performance opportunities of *field-programmable gate arrays (FPGAs)* for high-volume query processing are well known, complicated and tedious query compilation procedures still defeat the use of the technology for dynamic query workloads, which are relevant in practice.

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* [10] that can be reconfigured in less than a micro-second and thus supports even highly dynamic query workloads.

Our work brings the architectural advantages of FPGA technology to the XML world. Using our system, XML streams can be filtered *in the network*, saving network bandwidth, client-side parsing, and expensive pre-processing. The system is ready now and can be run on widely available FPGA hardware (on XUPV5 development boards).

1. INTRODUCTION

Thanks to their performance and architectural advantages, *field-programmable gate arrays (FPGAs)* have become a compelling technology for high-volume data processing. FPGA-based stream processors [12, 15] or XML filtering engines [11] were shown to excel with high throughput at low latency.

The catch in all of these systems is that they were designed for *off-line query compilation*. Each query workload must be run through a time-consuming compilation procedure, before the respective dedicated hardware circuit can be uploaded to the FPGA. Each compilation run may take up to several hours. As such, the approach is limited to situations where the query workload is mostly static and where the task assignment to the FPGA is known ahead of time.

In our ongoing project X[1] we aim for a *hybrid* CPU/FPGA solution that is much more ambitious. An *optimizer* decides on the assignment of tasks to processing resources (CPUs or FPGAs) and it may change this assignment *dynamically* as queries enter and leave the system. Clearly this calls for an FPGA design where the query workload can be modified *on-line*.

This work contributes an FPGA solution that allows *on-line query workload reconfiguration*. In contrast to earlier work, our system considers query workload as a mere *configuration parameter* to an otherwise hard-wired FPGA circuit. No prior compilation is needed to register or unregister queries, and workload changes become effective immediately, within less than a micro-second.

This is made possible through a generic hardware implementation of a *non-deterministic finite-state automaton (NFA)*. After the circuit has been uploaded to the FPGA, the circuits physical structure remains fully static. The language that it accepts—and thus the query workload—is stored in configuration parameters that can be arbitrarily changed at runtime. The static circuit interprets these parameters to yield a "soft" automaton with runtime characteristics comparable to circuits that required expensive pre-compilation in earlier work.

Our target application for this report is high-volume *XML filtering*. More specifically, we describe an FPGA implementation for *XML projection* [10], a proven and effective method to off-load computation work in XML streaming scenarios. Exploiting architectural advantages of FPGA technology, our system can be used to filter XML streams *in the network*. This further improves the effectiveness of XML projection, because clients no longer need to spend CPU cycles on costly XML parsing and pre-processing.[2]

On the technical side, our circuit leverages *pipelining* techniques and the high degree of *parallelism* inherent to FPGAs. This way our circuit can guarantee high and load-independent throughput.

This paper is structured as follows. Sections 2 and 3 give the necessary background on XML projection and FPGA hardware. Our main contributions are covered in Sections 4 and 5, where we detail our FPGA-based finite-state automaton 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, con-

[1]Project name suppressed for double-blind reviewing.

[2]XML parsing is highly CPU-intensive, to the extent that it may become a "threat to database performance" [14].

```
<site>
  <regions>
    ...
    <africa>
      ...
      <item id="item42">
        <name>vapour wept became empty </name>
        <incategory category="category3"/>
        <incategory category="category2"/>
        <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, everything else can be pruned.

sider the following query, which is based on XMark [16] data:

```
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[3] 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 effecting 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 [10]). In our example this is needed to include full `name` elements into the query result.

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.

[3]xmlgen (the XMark data generator) produces XML documents modeling an auction website.

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.

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 adapt this procedure and refer to [10] 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.

Figure 2 lists the XPath dialect that our hardware implementation supports. This dialect essentially covers all features of the projection path language in [10] (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::`').

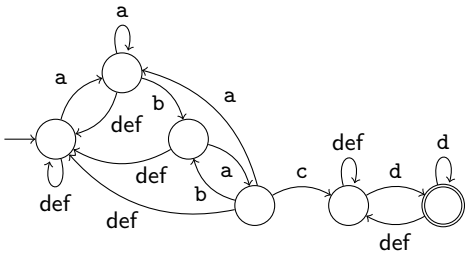
2.2 Path Evaluation (Previous Work)

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

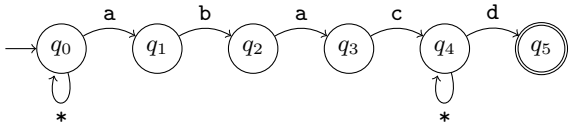
Finite-State Automata. Figure 3 illustrates this approach for the projection path `fn:root()//a/b/a/c//d`. This expression can be compiled into either a *deterministic* (Figure 3(a)) or a *non-deterministic* finite-state automaton (Figure 3(b)). Observe how, in the latter case, each $\circlearrowleft$`*` 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 [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 [7], this allowed the use of a *single* non-deterministic finite-state automaton that concurrently matches all registered input queries. The automaton structure is changed whenever a query is (un)registered.



(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`.

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 [11], 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, Woods et al. [19] use FPGAs to perform *complex event detection* based on regular expressions in hardware. They too generate a dedicated hardware circuit for each given event pattern and reprogram 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 [17] 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 [18]). Its key 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 eat the cake and have it too. To efficiently deal with (changing) XML projection workloads, our system is based on non-deterministic finite-state automata, which support fast runtime (re)configuration. We thus refer to these NFAs as "soft" automata, implemented as static circuits with runtime (re)configurable behavior.

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) 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 hardware 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*[4] by defining *(a)* the logic function f for each lookup table and *(b)* the signal wiring in the programmable 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 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 [9], 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).

In a circuit generated this way, every automaton state is represented by a flip-flop register (labeled 'FF' in Figure 4).

[4]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."

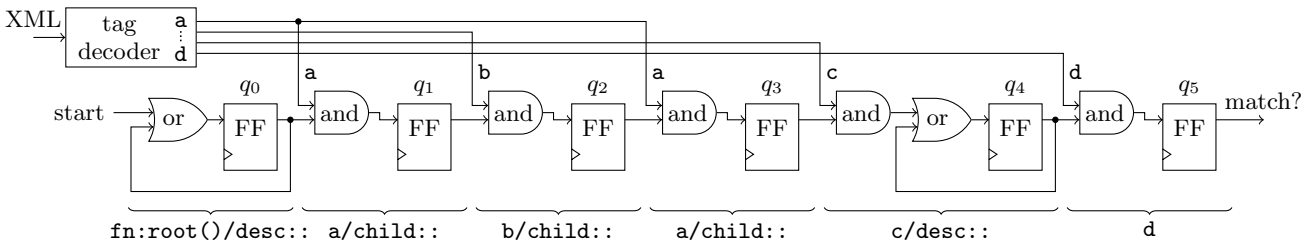


Figure 4: Hardware implementation of the non-deterministic finite-state automaton in Figure 3(b).

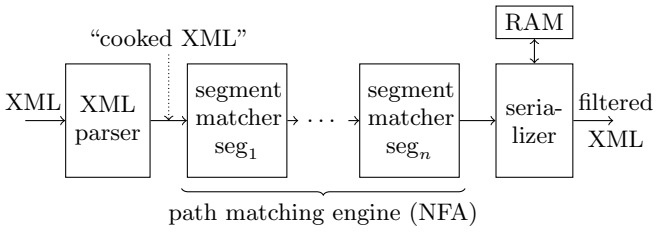


Figure 5: XML projection engine. After *parsing*, the XML stream traverses a number of *segment matchers* that inform the *serializer* which document pieces to emit.

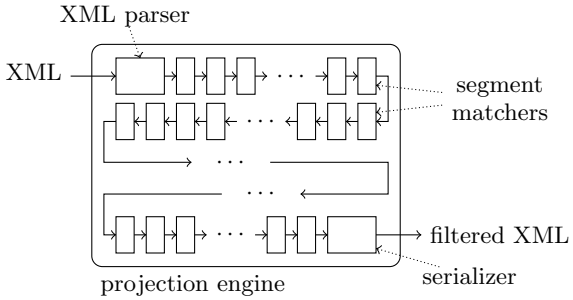


Figure 6: The sequential structure of the XML projection engine can efficiently be mapped to the two-dimensional chip space.

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 at state, these must be combined using an 'or' gate, as 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, too.

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 on the side 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 [11] for details).

4. DYNAMIC XML PROJECTION

The above idea works well if the whole finite-state automaton including its structure is known in advance, *i.e.*, when the circuit is compiled and uploaded to the FPGA. In this work we aim for *dynamic* XML projection, where the query workload can be modified at runtime (after FPGA programming).

4.1 System Overview

To support runtime (re)configuration, we built a special FPGA circuit whose high-level design is illustrated in Figure 5. 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 a series of *segment matchers* that perform 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 segment matchers take the lion's share of the available chip space (in practice there are hundreds of them). Together with the XML parser and the serializer they are arranged in a strictly sequential circuit structure. Such a structure can be mapped particularly efficient to the available two-dimensional *chip space*, for instance using a snake shape as illustrated in Figure 6. A so-obtained chip layout has a simple *routing structure* with only short-distance links. As we will see in Section 7, this allows us to operate our system at very high *clock speeds* to achieve correspondingly high *throughput rates*.

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

In the following sections, we explain how each component of the XML projection engine works and how together they provide a runtime-(re)configurable hardware implementation for XML projection.

4.2 XML Parsing

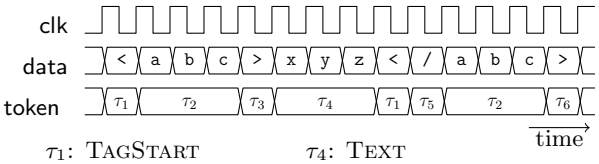


Figure 7: Timing diagram of XML parser output. The XML stream is enriched with a token signal to make lexical information explicit.

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, our 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 7 as a *timing diagram* (our circuit is fully synchronized; signal values may change at any rising edge of the clock signal clk). 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 *Y* hardware parser generator tool [2]. *Y* is part of a tool set that eases the development of database functionality on top of FPGAs. *Y* reads in the grammar of any (regular) language, computes the corresponding finite-state automaton, and emits a VHDL specification of a hardware circuit that recognizes that language.

Here we provided *Y* with a modified version of the XML language specification. In this version we added rules to recognize configuration commands (see below) and we removed support for namespaces and DTDs. Tests for well-formedness are handled outside the main parsing logic in order to make the input to *Y* a regular language. The resulting parser is a full-fledged XML parser with the exception of namespace and DTD support.

4.3 Path Matching

The key building block of our system is its path matching engine, which consumes a "cooked" XML stream and enriches it with a match flag. This flag is interpreted by the serializer to produce the projected XML document. To understand the inner workings of the path matching engine, we first assume there is only a single path expression to match. In Section 4.4, we extend the setup to support multiple projection paths.

4.3.1 Segment Matchers

The path matching engine consists of a series of *segment matchers*. Together, this series implements a hardware-based NFA (cf. Figure 4), but now in a fully runtime-(re)configurable way.

Each segment matcher thereby implements one of the NFA *segments* that we indicated at the bottom of Figure 4. On the XPath language level, each segment corresponds to a node test followed (!) by an XPath navigation axis. On the hardware side, an NFA segment contains some 'and' and 'or' gates, a flip-flop register, and—depending on the XPath axis to match—a back loop ⟲∗ or not. Rather than wiring and combining these logic components statically, a segment matcher can implement any possible combination of gates (and loops); the exact behavior is determined by runtime parameters that can be modified on-line.

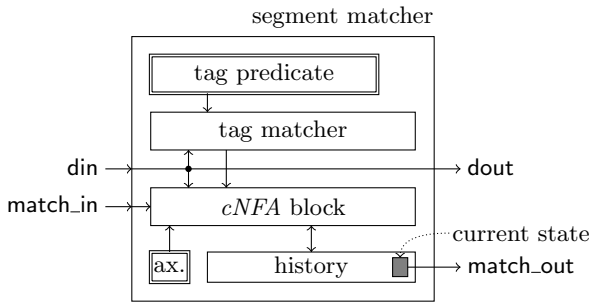


Figure 8: Internals of *segment matcher* component. ▣ blocks hold configuration parameters (axis and tag predicate).

4.3.2 Configurable NFA Block

Wiring and gate combination are implemented in our system by a component that we call *configurable NFA block*. It is parameterized by an *axis* information that decides on the logic gates to combine and it enables or disables the ⟲∗ loop.

The role of the configurable NFA block ("cNFA block") inside a segment matcher is illustrated in Figure 8. Through its match_in signal, the block receives the match state of the preceding segment matcher (this corresponds to the forwarded flip-flop states in Figure 4). The input signal din contains the cooked XML input stream.

The configurable NFA block interfaces to two flip-flop registers that encode the four supported XPath axes. This information is runtime-(re)configurable, which we indicate as ▣ in our illustration. The state of the two axis flip-flop registers will determine in which way 'and' and 'or' gates are to be combined for this NFA segment.

Our circuit does not use an external tag decoder. Instead, dedicated sub-circuits in each segment matcher provide information about matched tag names. We will detail those sub-circuits in a moment.

As in the hard-wired circuit (Figure 4), the current matching state for each NFA segment is represented by a flip-flop register, which we represented as ■ in Figure 8. Here we embedded the flip-flop inside a *history* unit that implements *backtracking* inside the segment matcher.

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.

Algorithm 1 summarizes in pseudo code the behavior of 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 configurable NFA.

configurable NFA block.[5] 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 Verilog/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.

4.3.3 Tag Decoding

Input to the configurable NFA block is an information whether an element with corresponding *tag name* was seen in the input. The classical approach to this sub-problem is shown in Figure 4. A dedicated *tag decoder* is compiled along with the main NFA. It includes a hard-wired set of tag names and produces a separate output signal for each tag name in the set. These signals are 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 interesting 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 of the segment matchers (cf. Figure 8), 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, reduces chip space consumption, 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; a single block is sufficient to store tag predicates.

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

```
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.

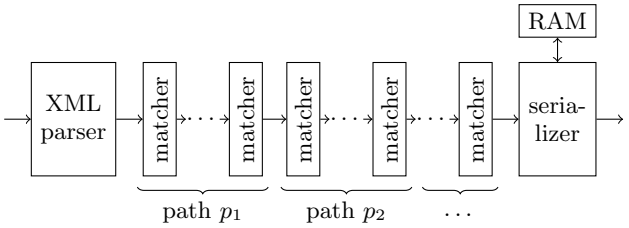


Figure 9: Multiple paths can be matched within a single processing chain.

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.

4.3.4 Matching a Whole Path

The combination of a number of segment matchers as a chain (see system overview in Figure 5) yields a non-deterministic finite-state automaton that can match a single path expression. A recognized match is indicated by a raising of the match_out signal of the right-most segment matcher.

The number of matchers required to match a path p depends on the length of p: one matcher is needed for each path step, plus a special matcher configuration that implements `fn:root()` (we detail this in a moment).

4.4 Matching Multiple Paths

Besides maintaining its own match state, each segment matcher in the path matching engine 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 9 illustrates the idea. As the XML input is streamed through, sections of the entire chain of segment matchers 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 matcher for the XPath built-in function `fn:root()` is the only one that does not depend

```
1 if end_of_chain_section then
2 |   glob_match_out ← glob_match_in or match;
3 |   match_out ← false;
4 else
5 |   glob_match_out ← glob_match_in;
6 |   match_out ← match;
```

Algorithm 3: Tag merging. At the end of each chain section, local match information is *merged* into the global match state.

on any previous matches. By placing it in front of every projection path, we break the linear finite-state automaton into separate automata that evaluate paths independently.

To evaluate `fn:root ()`, a matcher 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 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 around 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 $\uparrow*$ transition if it is configured to do so (*i.e.*, if `fn:root ()` is followed by a `descendant` step; line 4 in Algorithm 1).

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 *some* path along the chain found a match.

To establish such an information, 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.

The logic to implement this functionality (see Algorithm 3) is simple and boils down to an 'or' gate that is predicated on a piece of configuration information (denoting whether or not the segment matcher is last in a chain section). Observe also how we set the local match flag to false at the end of each path. As discussed previously, a following `fn:root ()` matcher will need this information to operate properly.

Resource Allocation. Note that the division of the entire chain of segment matchers into sections is not static. Rather, a sequence of segment matchers is allocated as needed for each projection path. This lets us make efficient use of resources and use the same circuit to match either many short paths or fewer paths if they are very long.

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).

```
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;
```

Algorithm 4: The XML Serialization unit 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.

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 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 4 illustrates this. When a match as discovered by the path matching engine, the input data stream is copied to the output, but not before opening tags were printed (from RAM) as needed to ensure the root-to-node property (lines 1–5).

Lines 7–17 copy all opening tag names from the input stream to the dedicated RAM tagmem. Lines 19–20 force the printing of closing tags 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).

5. RUNTIME (RE)CONFIGURATION

All workload-specific parameters are kept in on-chip storage that can be modified at runtime to change the active

```
<?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>
```

Figure 10: XML document with projection processing instructions `<?query ...?>` included.

query workload. In Figure 8, we indicated these parameter storage cells using double-lined boxes ▣.

Parameter Storage. Re-writable storage is a scarce resource on FPGAs and provided mainly by flip-flop registers and dedicated RAM (block RAM). To use the available capacities efficiently, we use both storage types and put different aspects of a query workload configuration into different storage types.

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 just the 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 to, *e.g.*, implement the gates in a configurable NFA node.

Tag predicates, by contrast, can become much larger. Our system uses dedicated RAM to store them. Virtex-5 contain hundreds of built-in BRAM blocks, each of which is 18 kbit in size. This is a good fit for tag predicates and leaves some room to accommodate even large query tag names. As mentioned before, one dedicated BRAM is allocated to each segment matcher.

5.1 Changing Parameters at Runtime

Query workload parameters are written by a *configuration logic* that sits next to the main processing logic, but operates independent to it. The source of the query workload information is application-dependent. Only the configuration logic must be changed to support different query workload sources.

For the purpose of this paper, we chose a configuration source that keeps our implementation fully self-contained. As illustrated in Figure 10, we inject the query workload directly into the input XML stream. Special *processing instructions* `<?query ...?>` distinguish the query workload from the actual XML stream. These processing instructions are recognized by a small set of XML parser extensions. In the "cooked" XML data stream they are represented as special token values.

Configuration Logic. The configuration logic itself is distributed and integrated into the segment matchers. The logic snoops the bypassing XML stream on the din signal line and writes configuration information into the respective storage units.

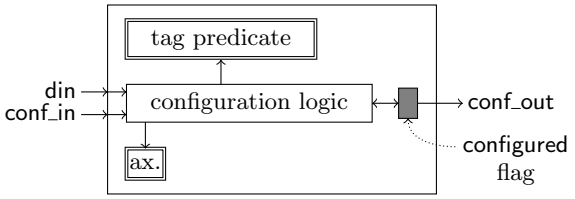


Figure 11: Configuration logic for runtime query workload (re)configuration.

```
1  if din.type = ConfReset then
2      configured ← false;
3  if conf_in and not 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             configured ← true;
14         case EndOfPath
15             end_of_chain_section ← true;
```

Algorithm 5: Semantics of configuration logic.

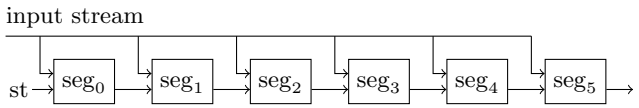
Figure 11 illustrates this interaction. Configuration logic in the middle interprets the din signal and updates tag predicates as well as the flip-flop-based configuration flags. Any configuration update will become effective immediately. Any following XML data will always be processed according to the new query workload.

Matcher Allocation. For new workload queries, segment matchers are allocated and configured from left to right (that is, the first workload query p_1 will occupy a matcher subset just after the XML parser; later p_i will follow in the processing chain toward the serializer; cf. Figure 9).

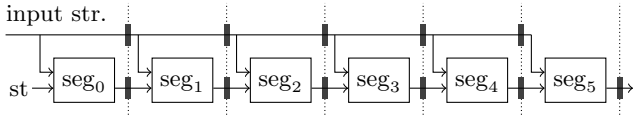
To implement this behavior, the distributed pieces of the configuration logic synchronize between themselves with help of a configured flag (implemented as a flip-flop register) and conf_in/conf_out signals that are propagated from left to right. A local piece of configuration logic "listens" to configuration tokens as soon as its predecessor has raised the conf_in signal. Once the local configuration is complete, the baton is passed to the right by setting the configured flag (and thus raising the conf_out signal).

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 5, different tokens will trigger writes to different storage locations (lines 1–3 and 13 implement the aforementioned synchronization).

A segment matcher corresponds to one node test and its following XPath axis. Thus, the local configuration is complete when the `::` is seen in the input stream. As shown in



(a) standard approach to hardware NFA



(b) fully pipelined NFA for XML projection

Figure 12: Standard hardware NFA implementation (top) requires long signal paths. Pipelining (bottom) reduces signal paths by inserting *registers* ▮.

lines 12–13, this is the situation where the configured flag is set and the configuration baton passed on to the right.

The `<?query reset?>` processing instruction clears all configured projection path. Lines 1–2 in Algorithm 5 implement this by clearing the configured flag when the ConfReset token is seen in the stream.

6. TUNING FOR PERFORMANCE

Like 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 12(a) emphasizes the same concept but hides the inner details of circuit segments seg_i.

Figures 4 and 12(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, this problem is inevitable, since 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 into the *same direction*.

Pipelining. The corresponding circuits are thus amenable to *pipelining*, a very effective circuit optimization technique. Figure 12(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. 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.

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 at every clock cycle. This benefit comes at the expense of a small *latency* penalty that increases proportional 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 will be travelling over the network in a client-server setup.

Pipelining in Our System. Pipelining is particularly effective in FPGA designs. In FPGA hardware, each lookup tables is inherently 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 (since the flip-flop register would remain unused otherwise).

In our system, we place a pipeline register after every segment matcher. 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 ($\approx$ 160 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 [10], our supported language dialect covers the XPath `self` and `descendant-or-self` axes. These axes can *not* be expressed using an off-the-shelf 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.

6.2 BRAM Sharing

As discussed before, we use dedicated RAM to store tag predicate configuration parameters for all segment matchers. This may lead to an upper limit on the number of matchers 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 segment matchers to 296.[6]

At the same time, we are underutilizing the available RAM blocks. The full 18 kbit of a Virtex-5 BRAM unit are rarely

[6]In practice, this number is further limited, because the serializer component and surrounding glue logic require few additional RAM blocks.

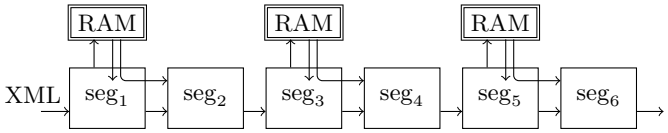


Figure 13: BRAM sharing. Two segment matchers store their tag predicates in the same RAM block. Since each block has only one interface, matchers seg_{2k-1} mediate traffic for matchers seg_{2k}.

needed for a tag predicate in 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 segment matchers, which effectively multiplies the supported NFA size. Figure 13 illustrates how this idea can be realized in FPGA hardware. Since there is only a single port to each BRAM block, some segment matchers act as *mediators* for the communication information of their neighbors.

BRAM sharing is useful only up to the point where the number of matchers 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 two segment matchers share one BRAM unit (like the situation shown in Figure 13).

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.

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 good 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 recent FPGA hardware).

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

Figure 14 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

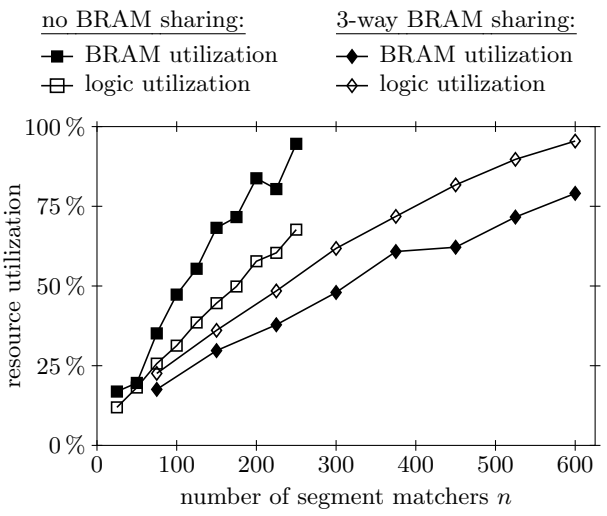


Figure 14: FPGA chip resource consumption of various engine configurations. BRAM sharing allows to balance the use of logic and BRAM resources to obtain a larger overall engine size.

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 ≈ 250 segment matchers (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 (plotted using diamond symbols), the maximum number of segment matchers is now limited by logic resources (lookup tables for that matter) and we can instantiate up to 600 segment matchers 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 frequency* at which each of our engine configurations could be operated.[7] Figure 15 illustrates the numbers we obtained.

The clock frequency directly determines the *speed* of the XML projection engine. One input byte can be processed on every clock cycle (independent of the query workload). With clock frequencies in the 150–175 MHz range, our system can thus provide 150–175 MB/s sustained XML throughput. This is more than enough for the use cases our system is designed for: our FPGA implementation 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 15 are also a good indicator for the scalability properties 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).

[7]Physical constraints on FPGA hardware (clock frequencies are generated by a *phase-locked loop*) restricts allowable frequencies to discrete values (namely 150, 160, 166, 175, and 180 MHz).

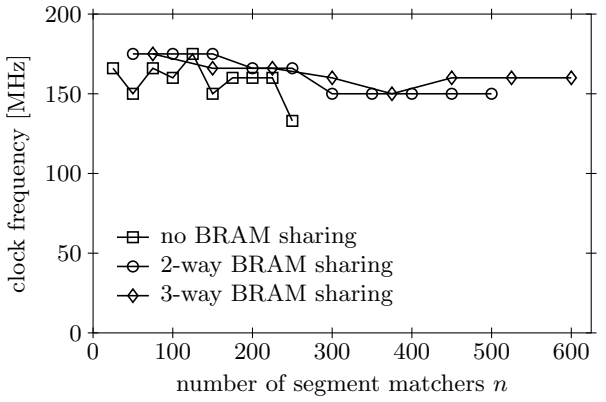


Figure 15: Maximum clock frequency for various engine configurations. Frequency is not strongly influenced by circuit size.

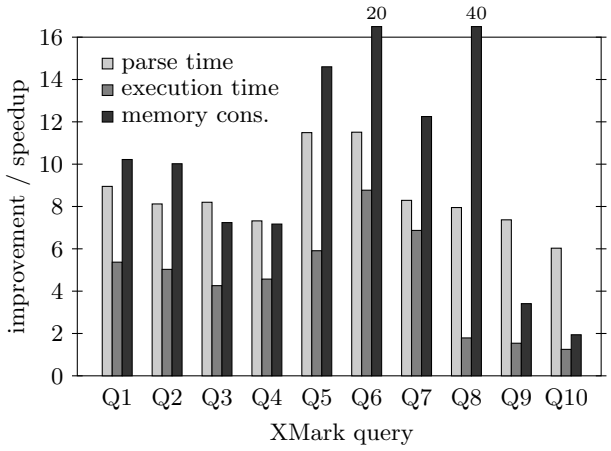


Figure 16: Speedup and improvement in memory consumption due to XML projection for first 10 XMark queries (memory reduction for Q6 and Q8 out of scale).

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 [6]. It is reasonable to expect that our system would keep its performance characteristics even when it was scaled up to 5000 or more segment matchers on current Virtex-6 chips [20].

7.2 XML Projection in the Network

FPGAs may offer significant advantages over software-based systems in terms of performance and/or power consumption. 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 a hardware Ethernet interface. The so-obtained system can perform XML filtering directly *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 [10]). Moreover, filtering in the network additionally saves much of the *parsing cost* that is a significant cost factor in typical XML applications [14].

We verified this on the basis of the Saxon-HE (version 9.2.1.2J), a state-of-the-art XQuery processor for in-memory processing, and the XMark benchmark suite [16]. 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 150–175 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 100 MB XMark instance. With protocol overhead counted in, this corresponds to a bandwidth of 123 MB/s on the physical network link, or 98.7 % 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.*, to a disk controller).

Application Speedup. On the application side, in-network filtering leads to significant improvements in parsing time, execution speed, and memory consumption. On raw data, Saxon requires 2.79 sec for input parsing (independent of the query) and uses between 450 MB and 1.3 GB of main memory (query-dependent; average over all XMark queries: 560 MB). Filtering reduces the parsing time to 37–680 msec (query-dependent; average: 322 msec) and main memory requirements to 20–365 MB (average: 82 MB).

Figure 16 visualizes the speedup in processing time and the improvement in main memory consumption for the first 10 benchmark queries. As expected, the main benefit of hardware-accelerated XML projection is the reduced parsing time as well as significantly reduced main memory requirements.

8. MORE RELATED WORK

After Marian and Siméon proposed the concept of XML projection in [10], the idea was expanded into different directions by the research community.

On the path evaluation side, Koch et al. [8] suggested an interesting alternative to the automaton-based path matching as we discussed it in Section 2.2. The key insight are the problem's similarities to *string matching*. This allows the use of proven-efficient string matching algorithms for the matching task, such as the classical Boyer-Moore [4] algorithm or—to match sets of paths—the string matching algorithm of Commentz-Walter [5].

The work of Benzaken et al. [3] primarily improves the query analysis part. The proposed *type-based XML projection* looks at type information rather than plain `child/de-`

`scendant` paths. This allows to build a more selective projection filter, which further reduces the size of the projected XML document.

FPGAs are an increasingly attractive alternative to overcome the architectural limitations of commodity hardware. The FPGA accelerators in the IBM/Netezza [13] data warehousing appliance are based on a parametrization mechanism, too. Based on parametrization, the included FPGAs can perform very simple filtering and table projection tasks near the attached disk controllers. Our ambition in *X* is to leverage FPGAs for tasks far beyond the most simple ones, such that the critical pieces of a query execution plan can be ran fully in hardware.

9. SUMMARY

We presented a hardware implementation for *XML projection*, a proven-effective method to reduce processing and main-memory overhead of XML processors. As such, we make the architectural advantages—such as 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.

The key innovation of our work, however, goes far beyond the presented XML projection scenario. Other than any existing FPGA solution found in the database literature, our system is fully *runtime-reconfigurable*. Our "soft automaton" approach allows runtime workload changes that become instantly effective (while existing solutions may require hours of off-line compilation to load a new workload).

On the hardware technology side, our work contributes a filtering engine with very favorable scalability properties. This makes our work ready for upcoming chip generations that will provide significantly more chip space.

The work we presented in this paper is part of our research effort on system *X*, a hybrid CPU/FPGA database engine currently under development in our group. The goal is to design a highly dynamic engine with support for ad-hoc querying and runtime resource optimization.

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] Anonymous. Reference suppressed for double-blind reviewing.

[3] 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.

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

[5] 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.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] 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.

[11] 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.

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

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

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] Xilinx Inc. Virtex-6 Family Overview, January 2010.