FPGAs for Dynamic (XML) Query Workloads

Chongling Nie Jens Teubner Louis Woods
Systems Group, Department of Computer Science, ETH Zurich
firstname.lastname@inf.ethz.ch

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* [9] 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.

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 [11, 14] or XML filtering engines [10] 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 *Avalanche* 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 circuit's 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. Our system still achieves runtime characteristics comparable to circuits that required expensive pre-compilation in earlier work. It is fast enough to, *e.g.*, process data at full gigabit Ethernet wire speed.

Our target application for this report is high-volume *XML filtering*. More specifically, we describe an FPGA implementation for *XML projection* [9], 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.[1]

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 Section 4, where we detail our FPGA-based automaton and how it allows for runtime (re)configuration. Section 5 evaluates our system, Section 6 relates our work to others'. We wrap up in Section 7. For the interested reader, an appendix discusses implementation and optimization aspects.

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 [15] 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 and prints their name together with the number of categories they appear in.

[1]XML parsing is highly CPU-intensive, to the extent that it may become a "threat to database performance" [13].

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. Articles from this volume were invited to present their results at The xth International Conference on Very Large Data Bases.
Proceedings of the VLDB Endowment, Vol. X, No. Y
Copyright 20xy VLDB Endowment 2150-8097/11/XX... $ 10.00.

```
<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, everything else can be pruned.

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.

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 (as in [9], the `#` symbol indicates that subtrees below `name` elements are to be preserved in the output, while all other subtrees can be discarded).

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 adapt this procedure and refer to [9] 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. The complete XPath dialect supported by our hardware implementation is shown in Figure 2. It essentially covers all features in [9], but without namespace support.

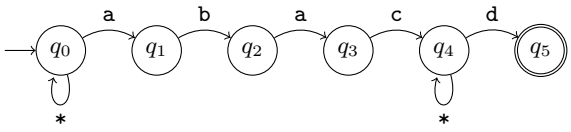


Figure 3: Non-deterministic finite-state automaton to implement query `fn:root()//a/b/a/c//d`.

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. At runtime, a *finite-state automaton* is driven by a SAX-style XML parser to evaluate a path expression.

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

Software implementations typically prefer *deterministic* automata instead, since each input symbol can then be processed using a single $\langle state, symbol\rangle \mapsto state$ lookup. This has been realized, for instance, in the XFilter [1] system, which creates one deterministic automaton for each registered query. On the flip side, non-deterministic finite-state automata are significantly easier to construct and maintain. In YFilter [6], this allowed the use of a *single* non-deterministic automaton that concurrently 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 [10], 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 [19] we compiled dedicated hardware circuits for *complex event detection* queries over data streams. However, either approach incurs 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

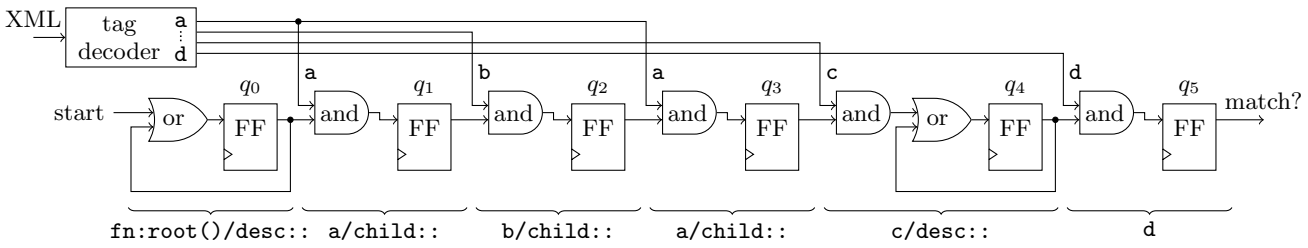


Figure 4: Hardware implementation of the non-deterministic finite-state automaton in Figure 3.

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 logic and flip-flop registers.

In addition to the input data, most circuits depend on a *clock signal*, a periodically changing high/low signal, to *synchronize* all circuit components. The *speed* of a 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* by defining *(a)* the logic function f for each lookup table and *(b)* the 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 [8], 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.

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. $\uparrow\!)*$ 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 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 can be 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 [10] 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 Fig-

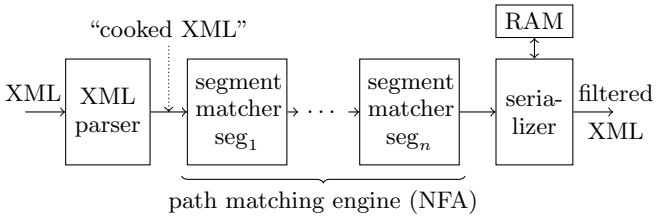


Figure 5: XML projection engine. After *parsing*, the XML stream traverses a number of *segment matchers* that inform the *serializer* which document pieces to emit.

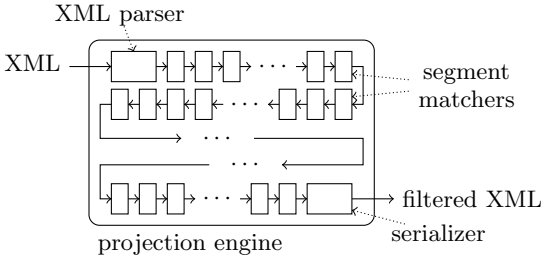


Figure 6: The sequential structure of the XML projection engine can efficiently be mapped to the two-dimensional chip space.

ure 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 5, 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 $\uparrow\!\!\downarrow*$ 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 $\uparrow\!\!\downarrow*$ loop or not.

4.2 XML Parsing

The input XML byte stream enters our system on the left

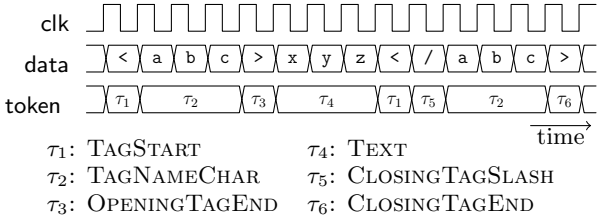


Figure 7: Timing diagram of XML parser output. The XML stream is enriched with a token signal to make lexical information explicit.

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* (clk is the FPGA clock signal). 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 component with help of our *Snowfall* parser generator [16]. The parser is standards-compliant, but lacks support for namespaces and DTDs.

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 and the XPath navigation axis that follows it. 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 $\uparrow\!\!\downarrow*$ 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.

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 $\uparrow\!\!\downarrow*$ loop.

The role of the configurable NFA block ("cNFA block") in-

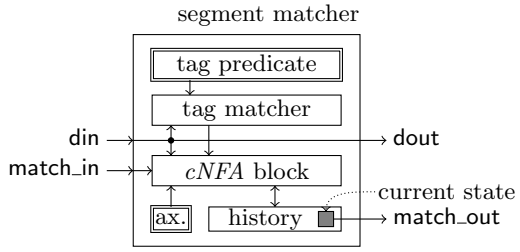


Figure 8: Internals of *segment matcher* component. ▣ blocks hold configuration parameters (axis and tag predicate).

```
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.

side 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.

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 configurable NFA block.[2] 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 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.

Inside each segment matcher, tag matching then boils down to a *string comparison*, which is simple to do in hardware and thus allows for high clock speeds. Each incoming tag name is compared to a *tag predicate* (a tag name, wildcard, or node test). The tag predicate is part of the query workload and held in *dedicated RAM* blocks, one of which is connected to each tag matcher.

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 nondeterministic 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

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

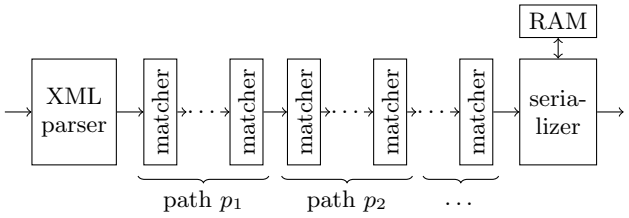


Figure 9: Multiple paths can be matched within a single processing chain.

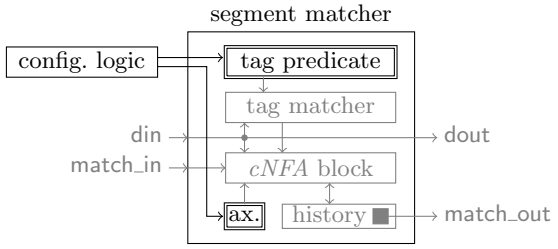


Figure 10: Configuration logic changes workload parameters outside the main processing and data path.

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, at the end of each chain section, *merge* it with the local match result by means of a logical 'or' gate.

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).

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. Serializer implementation details can be found in Appendix C.

4.6 Runtime (Re)Configuration

The main goal of this work is to allow for dynamic query workload changes at runtime. This is made possible by keeping all query workload-specific parameters in on-chip storage cells that can be modified at runtime (indicated as double-lined boxes ▣ in Figure 8). Query workload changes can be applied by writing new parameters into these storage cells. For reasons discussed in Appendix A.1, we use dedicated RAM to store tag predicates and flip-flop registers for the navigation axes.

A side effect is that query workload configuration can be kept completely outside the main processing and data path. As illustrated in Figure 10, separate *configuration logic* can maintain both configuration parameters without interference with the processing logic. In Appendix A we describe one particular way to implement the configuration logic, with query workload information multiplexed into the network data stream. Alternatively, the configuration logic could receive its input, *e.g.*, via PCI or USB channels.

5. 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 5.1 we first characterize the core XML projection engine, before in Section 5.2 we show how the engine could be used in a working system.

5.1 Core Engine

We focus our evaluation of the core XML projection engine here on its *scalability*. We want the circuit to sustain network-speed throughput even when sizes or amounts of queries are large. Another indicator for the quality of an FPGA design is its *resource utilization*. We discuss this indicator along with optimization strategies in Appendix B.

Clock Frequencies. To evaluate the scalability, we created

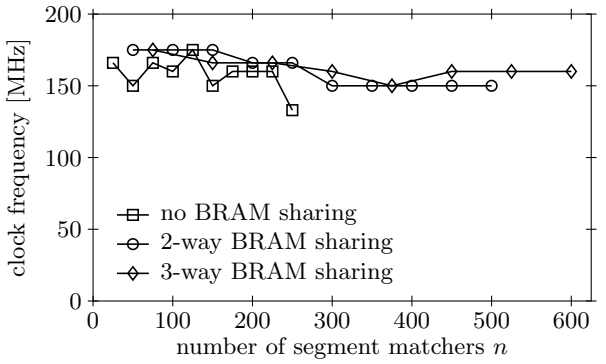


Figure 11: Maximum clock frequency for various engine configurations. Frequency is not strongly influenced by circuit size.

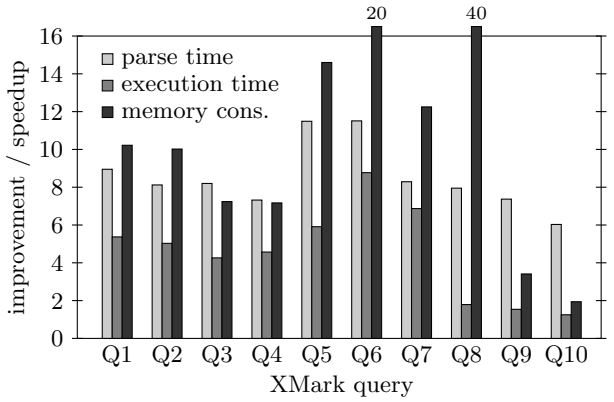


Figure 12: Speedup and improvement in memory consumption due to XML projection for first 10 XMark queries (memory reduction for Q6 and Q8 out of scale).

instances of our projection engine with a varying number of segment matchers n. This number controls the maximum size of the query workload (one path step per matcher) but also the chip space requirements of the circuit. For each configuration, we used FPGA design tools to determine the maximum *clock frequency* at which it could be operated.[3] Figure 11 illustrates the numbers for our basic circuit ("no BRAM sharing") but also with resource optimizations (cf. Appendix B.2) applied ("k-way BRAM sharing").

The clock frequency directly determines the maximum *speed* of the XML projection engine. One input byte can be processed on every clock cycle, *i.e.*, up to 150–175 MB/s of 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.

Scalability. The clock frequencies shown in Figure 11 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).

In our case the achievable clock frequency stays high even for configurations that significantly exceed the 70-80 % chip utilization beyond which performance often decreases [5]. 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.

5.2 XML Projection in the Network

Besides performance and power consumption advantages, the main benefit of FPGA technology 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 *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 (the main incentive in [9]) but also from lesser *parsing overhead*, which is a significant cost factor in typical XML applications [13].

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 [15]. 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. By design, our system operates in a strict *streaming mode* and, as detailed above, 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.

We verified this on real hardware and 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 12 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, combined with significantly reduced main memory requirements.

[3] We report only the discrete clock frequencies supported by the on-chip PLL clock generator.

6. MORE RELATED WORK

After Marian and Siméon proposed the concept of XML projection in [9], the idea was expanded into different directions by the research community.

On the path evaluation side, Koch et al. [7] 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 [3] algorithm or—to match sets of paths—the string matching algorithm of Commentz-Walter [4]. On the flip side, neither of those algorithms operates in a strict streaming mode, which makes them more difficult to use for hardware-based in-network processing.

The work of Benzaken et al. [2] proposes *type-based XML projection*. At runtime, the system checks type annotations in a validated input stream, which shifts much of the expensive work to the validation part. Much like projection paths, the necessary validation can be performed using state automata. With some modifications to our "soft automaton," we think that validation could be done in hardware, too. This would combine the advantages of [2] into our architecture.

FPGAs are an increasingly attractive alternative to overcome the architectural limitations of commodity hardware. The FPGA accelerators in the IBM/Netezza [12] 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 *Avalanche* is to leverage FPGAs for tasks far beyond the most simple ones, such that the critical pieces of a query execution plan can be run fully in hardware.

7. 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. Unlike any existing FPGA solution found in the database literature, our system is fully *runtime-reconfigurable*. Our "soft automaton" approach allows runtime query 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 *Avalanche*. Our goal is to design a highly dynamic engine with support for ad-hoc querying and runtime resource optimization.

8. REFERENCES

[1] M. Altınel and M. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *Proc. of the 26th VLDB Conference*, Cairo, 2000.

[2] V. Benzaken, G. Castagna, D. Colazzo, and K. Nguyễn. Type-Based XML Projection. In *Proc. of the 32nd VLDB Conference*, Seoul, Korea, 2006.

[3] R. Boyer and J. Moore. A Fast String Searching Algorithm. *Comm. of the ACM*, 20(10), 1977.

[4] B. Commentz-Walter. A String Matching Algorithm Fast on the Average. In *Proc. of the 6th ICALP Colloquium*, Graz, Austria, 1979.

[5] A. DeHon. Balancing Interconnect and Computation in a Reconfigurable Computing Array (or, why you don't really want 100 % LUT utilization). In *Proc. of the 7th FPGA Symposium*, 1999.

[6] Y. Diao, M. Altınel, M. Franklin, H. Zhang, and P. Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Trans. on Database Systems (TODS)*, 28(4), 2003.

[7] C. Koch, S. Scherzinger, and M. Schmidt. XML Prefiltering as a String Matching Problem. In *Proc. of the 24th ICDE Conference*, Cancún, Mexico, 2008.

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

[9] A. Marian and J. Siméon. Projecting XML Documents. In *Proc. of the 29th VLDB Conference*, Berlin, Germany, 2003.

[10] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras. Accelerating XML Query Matching through Custom Stack Generation on FPGAs. In *Proc. of the 5th HiPEAC Conference*, Pisa, Italy, 2010.

[11] R. Mueller, J. Teubner, and G. Alonso. Data Processing on FPGAs. *Proc. of the VLDB Endowment (PVLDB)*, 2(1), 2009.

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

[13] M. Nicola and J. John. XML Parsing: A Threat to Database Performance. In *Proc. of the 12th CIKM Conference*, New Orleans, LA, USA, 2003.

[14] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. *Proc. of the VLDB Endowment (PVLDB)*, 3(2), 2010.

[15] A. Schmidt, F. Waas, M. Kersten, M. Carey, I. Manolescu, and R. Busse. XMark: A Benchmark for XML Data Management. In *Proc. of the 28th VLDB Conference*, Hong Kong, China, 2002.

[16] J. Teubner and L. Woods. Snowfall: Hardware Stream Analysis Made Easy. In *Proc. of the 14th BTW Conference*, Kaiserslautern, Germany, 2011.

[17] J. v. Lunteren. Searching Very Large Routing Tables in Wide Embedded Memory. In *Proc. of the GLOBECOM 2001 Conference*, volume 3, San Antonio, TX, USA, 2001.

[18] J. v. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. XML Accelerator Engine. In *Proc. of the 1st Int'l Workshop on High-Performance XML Processing*, New York, NY, USA, 2004.

[19] L. Woods, J. Teubner, and G. Alonso. Complex Event Detection at Wire Speed with FPGAs. *Proc. of the VLDB Endowment (PVLDB)*, 3(1), 2010.

APPENDIX

To better focus on the key principles of our system, the main body of this paper abstracts away from many low-level aspects. In this appendix we add the missing details to make this work a documentation of our complete system, including a concrete implementation of the configuration part (Part A), hardware-based optimizations (Part B), and details on the serializer component (Part C).

A. RUNTIME (RE)CONFIGURATION

Runtime (re)configuration is achieved by placing query workload-specific information into suitable types of on-chip storage elements and by implementing a configuration logic that can update this storage at runtime. We discuss both aspects in turn.

A.1 Parameter Storage

The 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 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 that, *e.g.*, implement the gates in a configurable NFA node.

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 segment matcher. But we will shortly see how resource utilization and circuit performance can be improved if this allocation strategy is changed.

A.2 Configuration Logic

Deliberately we did not describe any specific implementation for the configuration logic in Section 4.6. Such logic depends on the way how query workload information is provided to the hardware implementation (*e.g.*, via PCI, USB, or a dedicated network link), which is orthogonal to path processing itself. Here we illustrate one possible implementation where query workload information is multiplexed into the input data stream.

Query Workload Format. This representation of the query workload information is illustrated in Figure 13. Projection paths are injected directly in the input XML stream, primarily because this keeps our prototype implementation self-contained. 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.

```
<?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 13: XML document with projection processing instructions `<?query ...?>` included.

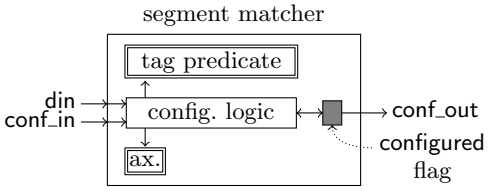


Figure 14: Configuration logic for runtime query workload (re)configuration.

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 14 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. Configuration updates will become effective immediately. Any following XML data will always be processed according to the new query workload.

Matcher Allocation. For new query workloads, 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 2, 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

```
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 EndOfPath
13             end_of_chain_section ← true;
14         case ColonColon
15             configured ← true;
```

Algorithm 2: Semantics of configuration logic.

following XPath axis. Thus, the local configuration is complete when the `::` is seen in the input stream. As shown in lines 14–15, this is the situation where the configured flag is set and the configuration baton passed on to the right. Lines 10–11 and 12–13 modifiy the history shift register and set the end_of_chain_section flag as needed to match multiple paths in one pipeline (cf. Section 4.4).

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

B. TUNING FOR PERFORMANCE

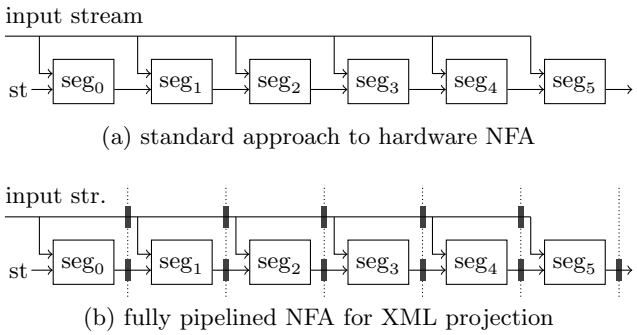
The high-level design of our engine is prepared to exploit some of low-level characteristics of the underlying FPGA hardware. In this section we fill in some of the implementation details that work toward goals in FPGA design that are quite distinct from those in software-based environments. Most importantly, an FPGA 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). Here we particularly elaborate on *pipelining* and *BRAM sharing* as ways to address both aspects.

B.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 15(a) emphasizes the same concept but hides the inner details of circuit segments seg_i.

Figures 4 and 15(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



(a) standard approach to hardware NFA

(b) fully pipelined NFA for XML projection

Figure 15: Standard hardware NFA implementation (top) requires long signal paths. Pipelining (bottom) reduces signal paths by inserting *registers* ▮.

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 15(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. What is more is that 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 travelling time of the same data item 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 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 a consequence, all signal lengths remain well within the range of clock frequencies that the FPGA hardware has been designed for. This is the main reason why in Section 5.1 we observed throughput rates (clock speeds) that are independent of the circuit size.

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 [9], our supported language dialect covers the XPath

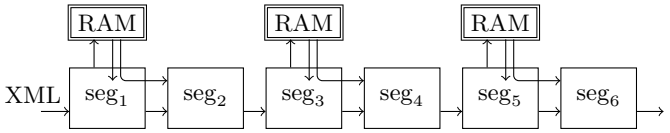


Figure 16: BRAM sharing. Two segment matchers store their tag predicates in the same RAM block. Since each block has only one interface, matchers seg_{2k-1} mediate traffic for matchers seg_{2k}.

`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.

Without pipelining and fast-forwarding, a system would either have to make k successive transitions on a single input symbol to evaluate k successive `self` steps, or support k-ary conjunctive predicates for state transitions. Both approaches would introduce another parameter dimension with significant complexity in the hardware-based designs. As far as we are aware, existing path evaluators based on state machines do not support '`self`' at all (*e.g.*, XFilter/YFilter).

B.2 BRAM Sharing

Dedicated on-chip RAM is a good fit to store tag predicate configuration parameters directly along each segment matcher. 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.[4]

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 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 16 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* that relay communication for their neighbors.

Resource Utilization. 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. A given piece of FPGA hardware is best exploited if the use of BRAM and logic resources are properly balanced.

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

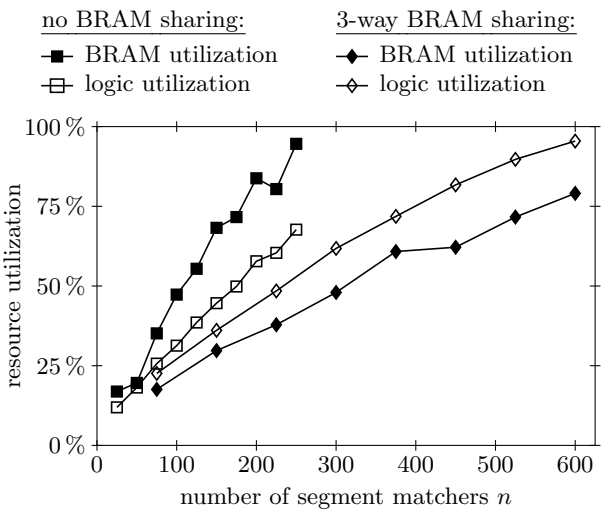


Figure 17: FPGA chip resource consumption of various engine configurations. BRAM sharing allows to balance the use of logic and BRAM resources to obtain a larger overall engine size.

We evaluated resource consumption for the hardware we had available. Again we instantiated various configurations of the XML projection engine (varying number of segment matchers; with and without BRAM sharing enabled) and this time determined the FPGA resources that the resulting circuit was using. Figure 17 reports the utilization for BRAM units (filled markers) and logic blocks (empty markers) as a percentage of the total available BRAMs/slices on the chip.

As can be seen in the graph, without BRAM sharing (square markers) 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). Besides a clear indication that the configuration underutilizes the available hardware, this situation also explains the performance degradation in Figure 11 with 250 segment matchers and no BRAM sharing. To reach the BRAM blocks that are spread all across the chip die, the matching pipeline must be *stretched* artificially to cover the entire chip. This again increases signal paths and thus limits the achievable clock rate.

Balancing Resources. Resource utilization is better balanced when BRAM sharing is enabled. With 3-way BRAM sharing (*i.e.*, three segment matchers share one BRAM block; 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).

With 2-way sharing (not shown in the graph for space reasons), BRAM and logic utilization are almost perfectly in balance. As we saw earlier in Figure 11, 3-way sharing still has a slight advantage in terms of the number of matchers supported. For this particular hardware, 3-way sharing is thus the best choice.

C. XML SERIALIZATION

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

Algorithm 3: 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.

As discussed on a high level in Section 4.5, the serializer component in our system temporarily writes opening tags into a dedicated RAM block. Whenever a match is detected, it copies those tags into the output stream to ensure that the projection remains transparent to the receiving application.

Algorithm 3 makes this functionality explicit. Lines 2–12 copy all opening tag names from the input stream to the dedicated RAM tagmem. When a match is discovered by the path matching engine, lines 20–22 check whether any opening tags are still missing in the output stream and emit them (using data from BRAM) if need-be. Line 23 copies all matching data directly to the output stream.

Lines 14 and 15 make sure that tags are properly closed again (even when they are not fully contained in any matched document region). Lines 16–18 do the necessary bookkeeping for closing tags.