Int. Conf. on High-Performance Embedded Architectures and Compilers
(HiPEAC), January 2010, Pisa, Italy

Accelerating XML Query Matching through Custom Stack Generation on FPGAs

Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis Tsotras

Department of Computer Science and Engineering
University of California, Riverside
CA 92521, USA
{rmous,msalloum,najjar,tsotras}@cs.ucr.edu
http://www.cs.ucr.edu

Abstract. Publish-subscribe systems present the state of the art in information dissemination to multiple users. Such systems have evolved from simple topic-based to the current XML-enabled systems. Here, users pose complex queries (expressed in XPath) on the structure and content of the streaming documents. The parts of the documents that match the user queries are then returned to the users. This paper proposes a novel hardware architecture that would exploit the parallelism found in XPath filtering systems. Using an incoming XML stream, parsing and matching with thousands of user profiles are performed simultaneously on a single FPGA, thus yielding up to three orders of magnitude higher throughput when compared to conventional approaches bound by the sequential aspect of software computing. By converting XPath expressions into custom stacks, our architecture is the first providing full support for all structural XPath constructs, including parent-child and ancestor descendant relations, whilst allowing wildcarding and recursion.

Keywords: FPGA, XML, Query, XPath, Compilation.

1 Introduction

Increased demand for timely and accurate event-notification systems has lead to the wide adoption of Publish/Subscribe Systems(or simply pub-sub). A pub-sub is an asynchronous event-based dissemination system which consists of three components: *publishers*, who feed a stream of messages into the system, *subscribers*, who post their interests (also called *profiles*), and an infrastructure for matching subscriber interests with published messages and delivering *matched messages* to the interested subscriber. Pub-sub systems have enabled notification services for users interested in receiving news updates, stock prices, weather updates, etc; examples include google.news.com, pipes.yahoo.com, and www.ticketmaster.com. Pub-sub systems have greatly evolved over time, adding further challenges and opportunities in their design and implementation. Earlier pub-subs involved simple topic-based communication. That is, subscribers could subscribe to a predefined collection of topics (e.g., news, weather, etc.). The

Y.N. Patt et al. (Eds.): HiPEAC 2010, LNCS 5952, pp. 141–155, 2010.

second generation consists of predicate-based systems which employ the Event-Condition-Action paradigm to perform profile matching and selective dissemination of information. Profiles are usually described as conjunctions of (attribute, value) pairs. For example, a profile could be: (concert, Police) AND (city, Los Angeles), for a user interested in being notified of Los Angeles concerts of the Police rock band. The wide adoption of the eXtensible Markup Language (XML) as the standard format for data exchange has led to the third generation, namely XML-enabled pub-sub systems. Here messages are encoded as XML documents and profiles are expressed using XML query languages, such as XPath [19]. Such systems take advantage of the powerful querying that XML query languages offer: profiles can now describe requests not only on the message values but also on the structure of the messages [1].

The wide adoption of XML is due to its self-describing and extensible nature; document content is tagged to provide a detailed description of its organization. An XML document has a hierarchical (tree) structure that consists of a root element and sub-elements. In addition, elements (or tags) can appear multiple times inside the same enclosing element (also referred as *recursion*). In the XPath query language queries are composed of a sequence of location steps. Each location step consists of an axis and an element. An axis specifies the hierarchical relationship between the document nodes. We focus on the two most common axes, namely, parent-child ('/') and ancestor-descendant ('//'). The parent-child axis specifies that two elements should appear at adjacent levels in the XML document tree. Likewise, the ancestor-descendant axis specifies that two elements can be separated by any number of levels in the XML tree. Wildcard characters ('*') are elements of XPath queries, providing a level of freedom by allowing any tag of the XML tree to replace them.

XML-filtering becomes a challenging problem when considering that it should support thousands of subscriptions, high volume of input streams, and should perform complex structural matching in a timely manner. Many software approaches have been presented to solve the XML filtering problem [1,6,7,9].These memory-bound approaches, however, suffer from the Von Neumann bottleneck and are unable to handle large volume of input streams. On the other hand, Field Programmable Gate Arryas (FPGAs) have been shown to be particularly suited for the stream processing of large amounts of data and do not suffer from the memory bottleneck faced by software implementations [15], [8]. Recently, in [14] we presented a proof-of-concept approach for the use of FPGAs on XML filtering. This approach, however, does not account for recursive elements in XML documents, neither for wildcards in the XPath profile expressions; both features are important constructs for XML documents and the XPath query language.

In this paper we present a novel implementation of XPath queries on FPGA that does support expressions with '/', '//', '*' and recursive elements in the XML documents. We present various alternatives and optimizations of this implementations and report on their respective costs benefits and trade-offs in terms of clock speed and area occupancy on the FPGA. We compare the achieved

[1] In the rest we use the terms "profile" and "query" interchangeably.

throughput to two popular software implementations: the LazyDFA [7] and FiST [9]. The results show up to three orders of magnitude of increasing throughput, with the geometric mean of the acceleration reported being 59x.

The rest of the paper is organized as follows: Section 2 presents related work while Section 3 provides in depth description of the proposed architectures targeted for XPath query matching. Section 4 presents an experimental evaluation of the FPGA based hardware approach compared to the state of the art software counterparts. Finally conclusions and open problems for further research appear in Section 5.

2 Related Work

The popularity of XML as a de facto standard for information exchange has triggered several research efforts to build scalable and efficient XML filtering systems. Several approaches have been proposed to solve the XML filtering problem. An early work, XFilter [1], proposed building a Finite State Machine (FSM) for each user profile, such that each element in the XPath expression becomes a state in the FSM. Each profile axis defines the transition between states, where the final state is the accept state for that FSM. The FSM states are executed as XML tag events are generated. An *open(tag)* event drives the FSM to the next state, while a *close(tag)* event drives the FSM back to the previous state. YFilter [6] built upon the work of XFilter, proposed a Non-Deterministic Finite Automata (NFA) representation of query path expressions which combines all queries (profiles) into a single machine. This approach yields better results since it exploits the commonality among path expressions. Green et.al. [7] proposed a lazy Deterministic Finite Automata. [9] proposed a sequence-based approach where both the XML document and query are transformed into Prufer sequences and subsequence matching is performed to determine if the query has a match in the document.

Nevertheless, the approaches discussed above are entirely software-based solutions abiding by the standard Von Neumann organization. One naive solution would be implementing the software-based FSM approaches on FPGAs. Such an approach however, is not efficient because the software approaches must dynamically allocate memory during XML filtering: consider the LazyDFA approach [7], which constructs the DFA in a lazy fashion during XML filtering. This approach cannot be implemented on the FPGA because the number of DFA states is not known in advance. In addition, recursion in XML streams requires dynamic initiation of multiple NFA processing engines during filtering which is not possible on FPGAs.

There are several approaches that use specialized parallel architectures for XML processing [10], [12]. In particular, [10] aimed to accelerate XML parsing using the Cell Broadband Engine multi-processor which consists of 8 independent processors (SPEs) that implement the FSM of the Zurich XML Accelerator (ZuXA) engine. This approach achieves parallelism by parsing (eight) XML documents in parallel at a time. In addition to be only suitable for XML parsing, this

solution is a combination of hardware-software approach. Similarly, the work in [12] addresses ways to load-balance parallel threads for low-level XML processing (e.g., XML parsing).

Previous work that have used FPGAs for processing XML documents have mainly dealt with the problem of XML parsing. In particular, [13] proposes the ZuXA engine to parse XML documents. This engine employs state machines for efficient parsing based on set of rules. The paper however does not provide any discussion how this engine can be adapted to evaluate XPath query expressions over the XML input.

The works in [11] propose the use of a mixed hardware/software architecture to solve simple XPath queries having only parent-child axis. A finite state machine implemented in FPGAs is facilitated to parse the XML document and to provide partial evaluation of XPath predicates. The results are then reported to the software part for further processing. Similarly to the ZuXA engine, this architecture is limitted to support simple XPath expressions with the parent-child axis.

Our previous work [14] was the first to propose a pure-hardware solution to the XML filtering problem. Improvements of more than one order of magnitude were reported when compared to software. However, this method is unable to handle recursion in XML documents or wildcards '*' in XPath profiles; such issues as well as various optimizations are handled by the novel architecture we present in this paper.

3 XPath Matching Hardware Architecture

Using an XML stream as input, we present a full-hardware XPath matching system on FPGAs; this section describes the details of the proposed approach. We start by providing an overview of our SAX Parser implementation, built upon a tag decoder, leading to a resource-optimized XML event notifier and overall architecture. The intuition behind mapping XPaths into stacks is then described, while contrasting with the shortcomings of previous approaches. We then propose some area saving optimizations through the reduction of the average width of required stacks. These optimizations would potentially imply a decrease in the operational frequency of the overall system on FPGA, a limitation which we prove to overcome using fan-out trees. Finally, we present the incentive behind the clustering of XPath matching engines, and the underlying technique used to report matches.

3.1 SAX Parser and Tag Decoder Implementation

The (*Simple API for XML*) (SAX) Parser [17], is an event-driven XML parser, ideal for streaming applications. Unlike other parsers (such as DOM [5]), where the entire XML document need be stored in memory before processing can start, SAX Parsers would generate *open(tag)* and *close(tag)* events on the fly, with all XPath query matching engines updating states accordingly. As a result, matching ends when the XML stream is complete.

With FPGAs being limited in hardware resources, a tag decoder is a desirable feature operating in conjunction with the SAX Parser. Since all query matching engines would need comparisons against respective tags, all engines executing in a parallel fashion, many redundant comparisons would take place across several engines, thus unnecessarily wasting resources. Decoders solve this issue by centralizing comparisons, and mapping decoded tags into single bit lines. All remaining comparisons are then translated into simple *AND* gates, hence, allowing the FPGA resources to be used for more useful computations. Our tag decoder is inspired from character decoding, the latter becoming conventional in pattern matching on FPGAs [16], [14], and which was shown to offer up to 83% of area savings in [14].

Fig. 3 shows how a tag decoder would operate in parallel with a SAX Parser in order to generate *open* and *close* tag events, with a tag being a single bit line out of the possible n decoded ones. Note that only one of those bit lines is high at a given point in time. Furthermore, the tag decoder is configured at compile time to recognize the n unique tags that would appear in the stream of XML documents.

3.2 Matching XPaths Using Path Specific Stacks

Due to the occurrence of parent-child relations, a stack is an essential feature of XML filtering systems, where an *open(tag)* is translated into a *push* event, and conversely, a *close(tag)* would be translated onto a *pop* event. For instance, matching the XPath *a/b* would take place when the *open(b)* event arises, with '*a*' being the top of stack. There is no bound to the number of children '*a*' could have within the XML document, any of those coming about prior to '*b*'. Thus, it would not be sufficient to check for the latest tag opened, but instead, the latest tag opened that still has not been closed.

A single global stack is needed to support the matching of parent-child relations for all XPath profiles. On the other hand, when using conventional state machine approaches, matching ancestor-descendant relations of the form *c//d* can be translated into a 3-state FSM, as we described in [14].

However, such methods fail to support recursion, a key aspect of XML documents, where certain tags are allowed to appear as their own children and/or descendants. Instead of using one global stack and one state machine per XPath user profile, we propose mapping each XPath into a customized stack, namely a *Path Specific Stack* (PSS). The PSS depth would be that of the XML document; furthermore the PSS width is equivalent to the depth of the XPath profile, where each tag of the query expression is mapped to a unique column, with regard to the order in which they appear in the XPath. A '1' would be stored in a column when matching for the tag mapped to it is true. This occurs with an *open(tag)* event for that tag being generated from the SAX Parser, with all previous tags having matched earlier. The storing of a '1' in the right-most column indicates a successful match for the entire XPath expression. The width of the stack, the surrounding logic alongside the tag decoded bit lines routed to it for matching purposes are all specific to each XPath query.

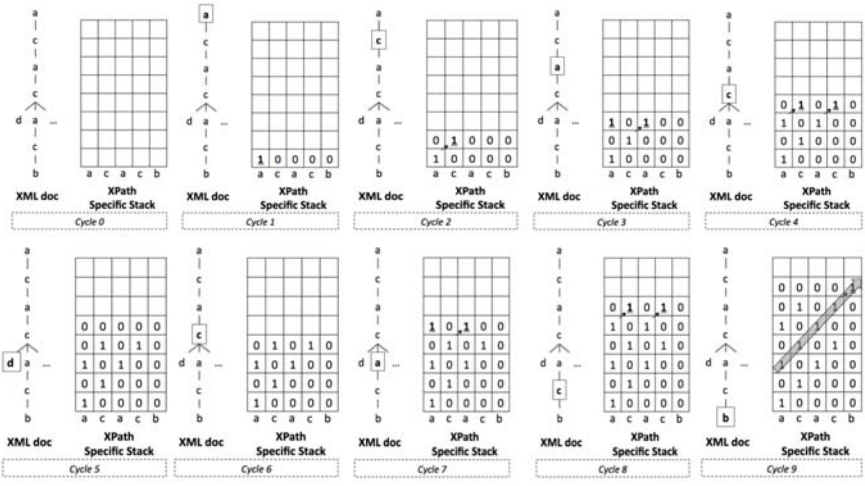

Fig. 1. Overview of the matching of XPath *a/c/a/c/b*. Each cycle refers to an *open(tag)* or *close(tag)* event, relative to the highlighted tag.

Fig. 1 shows a cycle by cycle overview of all the steps required for the matching of the XPath *a/c/a/c/b*, where, for simplification purposes, a cycle refers to a SAX Parser event. Cycle 0 reflects on the initial state of the system prior to any streaming taking place: the XML document to be streamed is drawn on the left hand side, whereas a PSS of width five is shown to the right. Each column is labeled with the corresponding tag of the *a/c/a/c/b* expression.

When the *open(a)* event takes place in cycle 1, the first column of the PSS would store a '1'. Consequently, with an *open(c)* event occurring in cycle 2, a '1' is stored in the second column, allowing the previous partial match stored in column 0 of the previous Top Of PSS (TOPSS) to propagate diagonally. In other words, an *open(c)* event alone is not enough to validate the matching of tag '*c*'. The fourth column in that same cycle demonstrates this behavior, for no matching was reported, due to no diagonally propagating '1'.

Support for recursion is depicted in cycle 3, where both the first and third columns indicate a match for tag '*a*' simultaneously, thus, allowing two possible matches of the same XPath to be in progress concurrently: one having started at cycle 1, the other at cycle 3. The state maching approach described in [14] would not take the new possible matches into consideration, since an FSM cannot reside in two states simultaneously. Moreover, each XPath query expression is mapped into a single state machine; therefore, multiple possible matches require multiple state machines, an issue which we solve using Path Specific Stacks.

With an *open(c)* event on cycle 4, both previous partial possible matches propagate diagonally. The occurrence of tags irrelevant to the XPath query has no negative effect on the matching process. For instance, with '*d*' pushed onto the stack on cycle 5, no partial matches are propagated. Moreover, roll-back to

the previous state took place on cycle 6 with the *close(d)* event taking place, thus popping the TOPSS.

A third partial possible match spawns off on state 7 (first column), while the first partial match that awaited an *open(b)* event had to stop propagation for the moment being, and can only resume matching until the currently pushed '*a*' is popped.

Propagation of partial matches resumes in cycle 8. Ultimately, a match has been found in cycle 9, thanks to the partial matching starting propagation from cycle 3. A match can be seen as a diagonal of 1's, ending in the fifth column.

Since our proposed architecture is not based on state machines as in [14], we offer support specific to our system for ancestor-descendant relations, as explained in Section 3.4.

3.3 Applied Optimizations for PSS Reduced Resource Utilization

As described in Section 3.2, PSS's have a width equivalent to the depth of the XPath profile mapped to it. With FPGAs being limited in resources, we propose some area reduction optimizations to be applied to the PSS. In this section, we focus on optimizing the PSS mapping of the same XPath profile used as a base example in Section 3.2.

One key observation reflected in Fig. 1 is that at most, two columns can be written to with regard to the occurrence of a single event. In other words, tag '*a*' maps to no more of two of the possible five stack columns, specifically columns one and three. Similarly, tag '*c*' maps to columns two and four, whereas tag '*b*' solely maps to column 5.

We base our optimizations on the introduction of a global stack. The decoded representation of tags would be pushed or popped onto this stack, in the case of *open(tag)* and *close(tag)* events respectively. Each of the global stack and Path

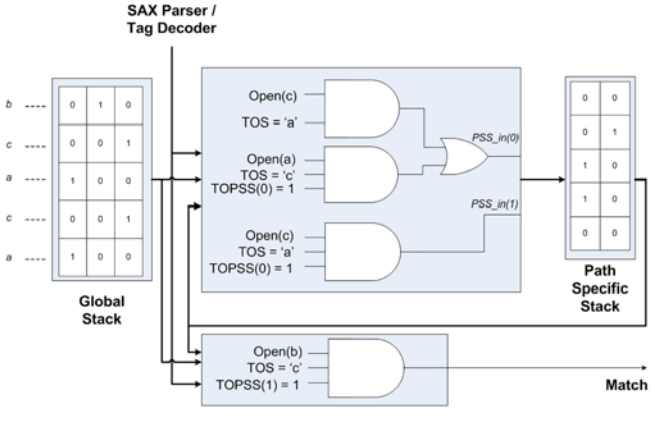

Fig. 2. Optimized hardware mapping for the matching of XPath *a/c/a/c/b*. Stack contents reflect on a *macthed* state of the expression.

Specific Stacks are pushed to and popped from concurrently. We refer to the top of the global stack as TOS. Fig. 2 shows the optimized mapping of the XPath expression *a/c/a/c/b*.

One intuitive optimization is not to map the final tag of the XPath expression to the PSS, rather to the match output signal, indicating whether or not a match of the query has occurred.

Moreover, with the help of the global stack, the first tag in an XPath need not be mapped to the PSS. Checking for the second tag consists of *AND*-ing the first label's decoded bit of the TOS, with the tag decoder output bit corresponding to the second tag; the top most *AND* gate in Fig. 2 connects the first bit of the TOS and the bit corresponding to tag '*c*' from the tag decoder output, in order to match *a/c*, the two initial tags of the XPath studied.

One more general optimization that aims to reduce the PSS width is to map multiple occurrence of different tags to the same column. The rule is to map one occurrence of a given tag from the XPath query onto the column following the mapping of the *last* occurrence of that *same* tag. By doing so, we fold the diagonal previously noted in Fig. 1, into the minimal number of columns; with the exception of the first and last tag in an XPath query expression, the needed PSS width is defined as the greatest number of repetitions accross all tags. Considering the expression *a/c/a/c/b*, tag '*c*' has the highest number of occurrences, being equal to two. Note that tag '*a*' has only one occurrence to be considered for PSS mapping, since the first tag in an XPath would not be mapped, as explained earlier. Since tag '*c*' has the most repetitions, namely two, the required PSS would have a width of two; the first initial of '*c*' and the second occurrence of '*a*' would map to the first PSS column, whereas the second occurrence of '*c*' is mapped onto the second column.

When propagating a '1', the global stack is essential in order to distinguish between multiple tags mapped to the same column. For instance, in the previous example, a '1' in the first column of the TOPSS accompanied by a decoded '*c*' in the TOS, would refer to the second tag in the XPath expression. On the other hand, a '1' in the first column of the TOPSS accompanied by a decoded '*a*' in the TOS states that the partial match has reached the third tag in the XPath expression.

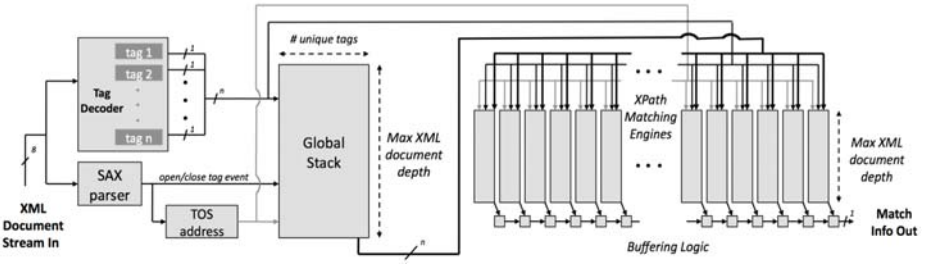

Fig. 3. High-Level system overview

The need of the global stack is reflected by some penalty on the logic surrounding the PSS. Looking at Fig. 2, it can be seen that, with the exception of the top-most *AND* gate, all remaining gates require an extra input, taking into account one bit from the TOS. Moreover, since two tags map to the same column, a 2-input *OR* gate was inserted prior to the first column of the PSS. Nonetheless, the reduction in required PSS width is noticeable; the overall area savings would be used to place structures relevant to achieving high throughput, as explained in Section 3.5.

3.4 Supporting Wildcards and Ancestor-Descendant Relationships

Wildcards, represented as '*' within XPath query expressions, imply that any tag from the XML document can be used as a valid replacement. In our architecture, this would mean that any decoded tag would help propagate the diagonal '1'. In other words, no global stack output is needed at the input of the corresponding *AND* gate. In the case of the unoptimized PSS, wildcards are translated into the output of the previous column directly routed into the input of the wildcard column, thus no extra logic is required. However, each wildcard should be mapped to a column of its own, since the tag used to replace a wildcard at a given point could be similar to any other tag from the XPath query. Therefore, wildcards exhibit a negative impact on the total occupied area.

Likewise, ancestor-descendant relations also have negative impact on resource utilization. Tags followed by ancestor-descendant relations should be mapped onto exclusive columns. The reasoning is that one column would reflect that tag being an ancestor, having appeared earlier in the document. In order to do so, the input of the PSS column consists of the regular matching logic propagating the previous '1', *OR*-ed with the output of the column itself. Note that PSS entries are updated solely upon push events. Thus, once the ancestor column stores a '1', all later pushed entries of that column would reflect the match, until the initial '1' is popped.

The optimization introduced in Section 3.3 regarding the first tag of an XPath expression, does not hold when that tag is followed by an ancestor-descendant relation, unless that tag is a wildcard; in that case, the second tag's propagating input is the *stack_not_empty* signal generated from the global stack.

3.5 System Architecture

We propose a scalable architecture appropriate for the simultaneous matching of thousands of XPath profiles.

Addressing Inner and Outer Fan-Outs via Clustering. One observation is that all stacks on chip would be updating concurrently; hence, all of the stacks' addresses would be generated from a common structure, which in turn requires push and pop notifications from the SAX Parser. Figure 3 illustrates this matter, where the TOS address is routed to the global stack and all remaining Path Specific Stacks (in the case of unoptimized PSS's, there is no need of a

global stack). This approach however creates a fan-out issue, where the address signal, the global stack output and the tag decoder output are replicated as many times as there are XPath profile matching engines (see Fig. 3), thus, affecting the allowed operational frequency.

A solution to this problem would be clustering, where the global stack, the SAX parser and the tag decoder would be replicated for clusters of PSS's, thus reducing the fan-out. This in turn raises the issue of the fan-out on the input stream, which would have to be replicated to feed into all clusters.

We refer to the fan-out within clusters as the *inner-fanout*; moreover, as the name indicates, the *outer-fanout* is caused by the out-of-cluster replication of the input stream.

One attempt to reduce the outer-fanout is the insertion of a binary fan-out tree on the input stream. Each node in that tree is a 9-bit buffer, capable of storing the input stream and an input valid bit. With each leaf of that tree feeding a single cluster, the outer fan-out would be eliminated, at the cost of many on-chip resources. Section 4.1 provides a thorough design space exploration on the allowed inner-fanout vs. tree size compromise. A reduced fan-out tree is introduced, which occupies less resources than a full tree, while keeping outer fan-out within reasonable bounds.

Reporting Matches. With thousands of matching engines co-existing on chip, reporting matches becomes a more complicated issue, where mapping each match signal exclusively to an FPGA pin is not an option. Our previous approach [14] suggested the use of priority encoders, where upon the event of a match, the unique encoded ID of the expression is returned. However, such an approach fails to acknowledge multiple matches occurring concurrently. XPaths *a//b* and *c/a/d/b* are such examples.

For the application of interest (filtering), the number of matches of each profile is of no interest, rather whether or not there was at least one match. Thus, we enhance our matching logic with one bit buffers relative to each PSS (Fig. 3); these buffers are connected serially. Upon the completion of the input stream, all of these results would be streamed out in a pipelined fashion, with a single bit port required. There would be N cycles of overhead required for this mechanism to complete streaming out, with N being the number of profiles. Nonetheless, this overhead is minimal when compared to the size of the documents streamed through the FPGA. In the case of clustering, we provide the option of having one match output signal per cluster. This would help reduce the overhead of sending the information out of the FPGA.

4 Experimental Results

We proceed with a design space exploration, where the effects of inner and outer fan-out, resource utilization and throughput are studied. We present four hardware systems, namely:

– No Optimization No Tree (**NONT**), where the PSS optimizations described in 3.3 are not applied.

- With Optimization No Tree (**WONT**), where the PSS optimizations for area reduction are applied, but the outer fan-out issue is not addressed.
- With Optimization With Tree (**WOWT**), where we apply both PSS optimizations and a binary fan-out tree having as many leaf nodes as there are clusters. This system cancels outer fan-out by using part of the optimized resources.
- With Optimization With reduced Tree (**WOWrT**); this is an architecture similar to WOWT, however the fan-out tree is reduced, having fewer leaf nodes than the number of clusters. While this approach would not eliminate outer fan-out, we expect that it would scale much better with almost no penalty on performance.

Our target platform is the Xilinx Virtex 5 LX330 [18] FPGA. With the proposed architecture heavily relying on memory structures, we make use of on-chip Block RAMs (BRAMs) [3]. These are highly configurable hard-wired memory blocks embedded in most Xilinx FPGAs. However, since the number of BRAMs is far fewer than that of all (global and path specific) required stacks, we only map global stacks to BRAMs. XPath queries on the other hand would be implemented using Distributed Memories (DMEMs) [4], memory structures built from slice LUTs. We provide a thorough resource utilization and performance study on the underlying tradeoffs of all of the four aforementioned hardware systems.

The reported performance is measured in throughput (MB/s), i.e., the average amount of data that can be processed over one second. All hardware systems assume a single character of 8 bits per cycle from the input stream's end. We compare the performance of our hardware systems against both of the LazyDFA [7] and FiST [9] software approaches.

We used a highly recursive XML Document Type Definition (DTD; which defines the allowed XML document structure) to generate XML documents and XPath queries for our experiments. The XML document datasets were generated by the ToXGENE XML Generator [2], setting the number of unique tags to 32, each consisting of two bytes. We generated documents of sizes of 5 and 50 MB, with a maximum XML document depth of 16. The same XML DTD was used to generate the set of user profiles using the XPath generator package provided by [6]. The maximum depth of a user profile was fixed at 6 and the probability of '*' and '//' occurrences was set to 10 percent. We varied the number of user profile datasets from 128 to 8192 queries.

All software experiments were ran on a quad core 2.33GHz Intel Xeon machine with 2GB of RAM, running Linux Red Hat 2.6.

4.1 Design Space Exploration

In order to evaluate the tradeoffs of excessive vs. sparse clustering, we ran a series of experiments, fixing the number of XPath queries at 2048, while varying the number of queries per cluster, up to 256 clusters (eight queries per cluster). We could not provide results beyond that point due to the limitation in the number of available BRAMs. We first compare NONT, WONT and WOWT.

The larger number of queries in each cluster, the higher the inner fan-out, thus the lower the outer fan-out, and vice versa. As expected, Fig. 4 shows that with the absence of clustering, inner fan-out is dominant and the operational frequency is much lower than achievable for all of three systems studied. Clustering proves to be beneficial up to a certain point, where the balance between outer and inner fan-out allows operational frequencies around the best achievable of 200 MHz. This behavior occurs around 128 queries per cluster. Beyond that point, where outer fan-out becomes dominant, both of NONT and WONT's performance deteriorates. On the other hand, WOWT would exhibit a rather constant superior performance at around 200 MHz. This is due to the full binary fan-out tree introduced as an effort to eliminate the effects of outer fan-out (at the expense of a higher area utilization). This penalty is tolerable and benefiting while the tree is kept small, up to 128 queries per cluster (where the tree has 16 leaf nodes and the WOWT area still is smaller than NONT's). Beyond that point, the tree grows too large, displaying up to 200% increased resource utilization.

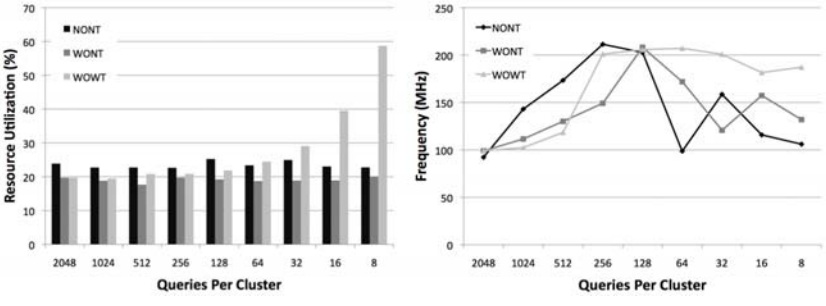

Fig. 4. Design space exploration with regards to the inner vs. outer fan-out compromise across three systems at 2048 queries

We then explore the scalability of the proposed architectures with the number of XPath queries ranging from 128 to 8192 (Fig. 5) and 200 MHz being our target operational frequency. We fix the number of queries per cluster at 128 (being the point where the best performance was realized at 2048 queries). However, we now evaluate the WOWrT setup, where the fan-out tree has a fixed 16 leaf nodes, the most adequate tree size from the previously shown exploration. Such an approach would not eliminate outer fan-out, but would keep the area utilization minimal, while almost no performance deterioration is noticed. For this approach only, we fix the number of queries per cluster to a more conservative 64. The intuition is that with outer fan-out reduced thanks to the tree, inner fan-out should be kept minimal with the help of extra clustering. Furthermore, since the target operational frequency of 200 MHz was achieved with no tree at 2048 queries, we only evaluate WOWrT for systems having at least 4098 queries (knowing that no tree is needed otherwise).

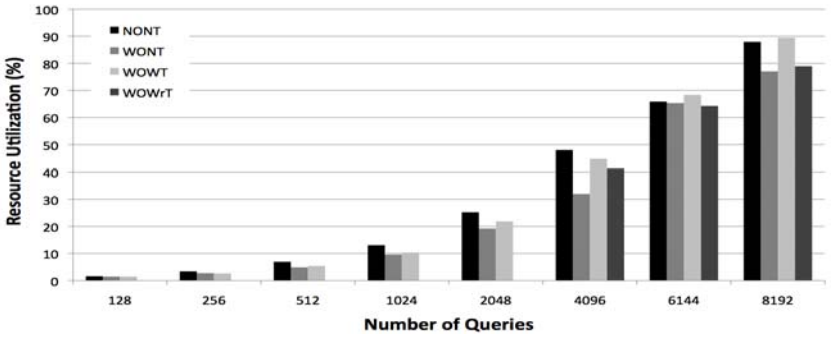

Fig. 5. Resource utilization comparison across all four proposed systems. NONT, WONT and WOWT results shown for 128 queries per cluster. WOWrT makes use of a 16 leaf-tree at 64 queries per cluster.

As shown in Fig. 5, all approaches scale surprisingly well, almost doubling the resource utilization while doubling the number of XPath queries.

PSS optimizations offer an average 20% of area savings. For the most part, WOWrT seems to scale as well as WONT, whereas WOWT suffers from the full binary fan-out tree.

Figure 6 presents the throughput for all approaches: as expected, a throughput superior to 200 MB/s is achieved up to 2048 queries across all systems. Beyond that point, a fan-out tree is needed, thus illustrating the benefits of WOWT and WOWrT, the latter being more consistent, having a smaller fixed size tree. Otherwise, a decrease in throughput is revealed across the remaining systems.

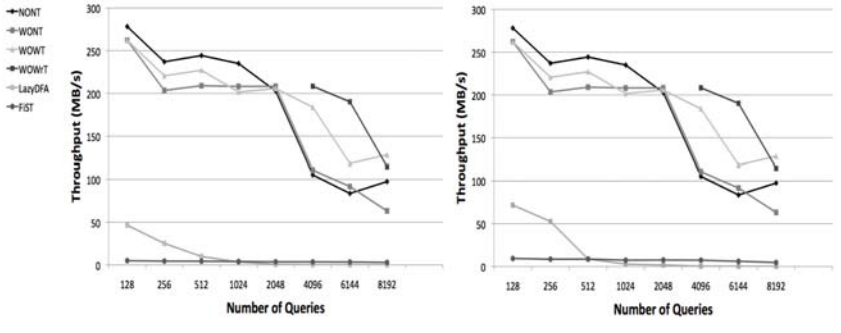

Fig. 6. Hardware vs. software performance with an increasing number of queries for streams of 5 MB (left) and 50 MB (right)

4.2 Performance Evaluation

In order to evaluate the performance of our proposed hardware architectures, we provide a comparison against two state of the art software approaches, namely

LazyDFA abd FiST (see Fig. 6). We report the throughput as the number of XPath queries increases from 128 to 8192, for two sets of XML streams of sizes 5 and 50 MB respectively.

Hardware throughput, being the operational frequency of the system, is independent of the input stream. On the other hand, the negative effects from the sequential computing aspect of the software approaches, is noted as the number of queries increases. Moreover, we show to overcome the Von Neumann memory bottleneck as hardware systems exhibit a speedup of up to three orders of magnitude, with a geometric mean of 59x. LazyDFA performs much better with the number of queries kept small; that approach does not scale too well, where FiST would outperform it beyond 2048 queries.

5 Conclusions and Future Work

In this paper, we presented a novel FPGA based hardware architecture to address the XML filtering problem. Using custom stack generation, our architecture is the first providing full support for all structural XPath constructs, including parent-child and ancestor descendant relations, whilst allowing wildcarding and recursion. Hardware architectures do not suffer from the memory bottleneck problem (better known as the Von Neumann bottleneck), since they are highly suitable for stream processing; they would also not suffer from the limitations of sequential processing, as the proposed architecture would support thousands of matching engines operating in a parallel fashion.

We were able to show that through moderate clustering and proper fan-out reduction, an average throughput of 200 MB/s can be maintained for up to 8192 matching engines, thus yielding up to three orders of magnitude accelerated throughput when compared to state of the art software approaches for various stream sizes. The reported geometric average of the acceleration is 59x.

As part of our future work, we would be looking into enhancing our Path Specific Stacks to support twig matching; here, user profiles are more complicated as they resemble trees. One common approach used by software systems is to split the twig into multiple XPaths. However, we are interested in holistic twig matching, where each engine would detect a twig without splitting it into multiple paths. The resulting system would be fully implementable on hardware, as no false positives are generated to reconstruct the twigs using any accompanying software.

References

1. Altinel, M., Franklin, M.J.: Efficient Filtering of XML Documents for Selective Dissemination of Information. In: Proceedings of the 35th Int'l Conference on Very Large Data Bases (VLDB), pp. 53–64 (2000)
2. Barbosa, D., Mendelzon, A., Keenleyside, J., Lyons, K.: ToXgene: a template-based data generator for XML. In: Proceedings of ACM Management of Data (SIGMOD), p. 616 (2002)

3. Block RAM v1.00a, http://www.xilinx.com/support/documentation/ip_documentation/bram_block.pdf
4. Distributed Memory Generator v4.1, http://www.xilinx.com/support/documentation/ip_documentation/dist_mem_gen_ds322.pdf
5. W3.org on DOM, http://www.w3.org/DOM
6. Diao, Y., Altinel, M., Franklin, M.J., Zhang, H., Fischer, P.: Path sharing and predicate evaluation for high-performance XML filtering. ACM Trans. on Database Systems (TODS) 28, 467–516 (2003)
7. Green, T.J., Gupta, A., Miklau, G., Onizuka, M., Suciu, D.: Processing XML streams with deterministic automata and stream indexes. ACM Trans. on Database Systems (TODS), 752–788 (2004)
8. Guo, Z., Najjar, W., Vahid, F., Vissers, K.: A quantitative analysis of the speedup factors of fpgas over processors. In: Proc. of the 12th ACM/SIGDA Int'l Symp. on Field programmable gate arrays (FPGA), pp. 162–170 (2004)
9. Kwon, J., Rao, P., Moon, B., Lee, S.: FiST: scalable XML document filtering by sequencing twig patterns. In: Proceedings of the 31st international conference on Very Large Databases (VLDB), pp. 217–228 (2005)
10. Letz, S., Zedler, M., Thierer, T., Schutz, M., Roth, J., Seiffert, R.: XML offload and acceleration with Cell broadband engine. XTech.: Building Web 2.0 (2006)
11. Linderman, R.W., Lin, C.S., Linderman, M.H.: FPGA acceleration of information management services. In: High Performance Embedded Computing, HPEC (2004)
12. Lu, W., Gannon, D.: ParaXML: A Parallel XML Processing Model on Multicore CPUs, Techincal Report (2008)
13. Lunteren, J.V., Engbersen, T., Bostian, J., Carey, B., Larsson, C.: XML accelerator engine. In: 1st Int. Workshop on High Performance XML Processing (2004)
14. Mitra, A., Vieira, M.R., Bakalov, P., Najjar, W., Tsotras, J.T.: Boosting XML Filtering with a Scalable FPGA-based Architecture. In: 4th Biennial Conference on Innovative Data Systems Research, Asilomar (2009)
15. Muller, R., Teubner, J., Alonso, G.: Streams on Wires – A Query Compiler for FPGAs. In: Proceedings of the 35th Int'l Conference on Very Large Data Bases, VLDB (2009)
16. Clark, C.R., Schimmel, D.E.: Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In: 13th international conference on Field Programmable Logic and Applications, pp. 956–959. Springer, Lisbon (2003)
17. SAX home page, http://www.saxproject.org
18. XILINX DELIVERS 65nm VIRTEX-5 LX330, http://www.xilinx.com/prs_rls/2006/silicon_vir/06130lx330delivery.htm
19. XML Path Language (XPath) Version 1.0, W3C Recommendation (1999), http://www.w3.org/TR/xpath