Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs

Roger Moussalli, Mariam Salloum, Walid Najjar, Vassilis J. Tsotras

University of California
Riverside, California 92521, USA
(rmous,msalloum,najjar,tsotras)@cs.ucr.edu

Abstract— **In recent years, XML-based Publish-Subscribe Systems have become popular due to the increased demand of timely event-notification. Users (or subscribers) pose complex profiles on the structure and content of the published messages. If a profile matches the message, the message is forwarded to the interested subscriber. As the amount of published content continues to grow, current software-based systems will not scale. We thus propose a novel architecture to exploit parallelism of twig matching on FPGAs. This approach yields up to three orders of magnitude higher throughput when compared to conventional approaches bound by the sequential aspect of software computing. This paper, presents a novel method for performing unordered holistic twig matching on FPGAs without any false positives, and whose throughput is independent of the complexity of the user queries or the characteristics of the input XML stream. Furthermore, we present experimental comparison of different granularities of twig matching, namely path-based (root-to-leaf) and pair-based (parent-child or ancestor-descendant). We provide comprehensive experiments that compare the throughput, area utilization and the accuracy of matching (percent of false positives) of our holistic, path-based and pair-based FPGA approaches.**

I. Introduction

With the recent advent of social networking and blogging services, the Publish/Subscribe (or simply pub-sub) paradigm is being utilized for timely event-notification. A pub-sub system 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 profiles with published messages and forwarding the messages to the interested subscriber. Pub-sub systems have evolved from simple topic-based [1], to content-based systems [2], [3], [4], to recent XML-based systems [5], [6], [7], [8], [9], [10]. Given the adoption of XML as the standard format for data exchange, in this paper we focus on XML-based pub-sub systems. In XML-based pub-sub, messages are encoded as XML documents and profiles are expressed using XML query languages, such as XPath [11]. Such systems take advantage of the powerful querying that XML query languages offer, since profiles describe requests on structure as well as content.

Many software approaches have been presented to solve the XML filtering problem [5], [6], [7], [8], [9], [10]. These memory-bound approaches, however, suffer from the Von Neumann bottleneck and are unable to handle a large volume of input streams. On the other hand, *Field Programmable Gate Arrays* (FPGAs) have been shown to be particularly suited for stream processing applications and do not suffer from the memory bottleneck faced by software implementations [12], [13], [14]. In [13] we presented a proof-of-concept approach for the use of FPGAs for the XML filtering problem that expressed path queries using regular expressions and built a NFA in hardware to facilitate the query matching. The proposed approach, however, does not account for recursive elements in XML documents or wildcards(‘*’) in the XPath expressions. In [14] we presented a new approach for matching complex path profiles that supports the /child:: axis and /descendant-or-self:: axis [1], wildcard (‘*’) node tests and accounts for recursive elements in the XML document. This approach proved very efficient for matching path profiles, but is unable to match profiles expressed as a twig structure. In this paper, we present a new approach for processing twigs on FPGAs in a *holistic* manner. In addition, we provide experimental comparison between the different granularities of twig matching, namely, holistic twig matching, path matching, and pair matching all within the FPGA framework.

The main contributions of this paper can be summarized as follows:

- We present a novel dynamic programming approach for holistic twig matching on FPGAs without false positives. Our solution is based on mapping profiles into custom Push and Pop stacks in hardware.
- We present a comparison between the different granularities of twig matching, namely holistic twig matching, root-to-leaf path matching and parent-child or ancestor-descendant pair matching, on FPGAs.
- We present a thorough experimental evaluation of throughput and area utilization of the three FPGA-based proposed approaches. We then evaluate the throughput gain and area utilization of the various approaches against the accuracy of matching (percentage of false positives returned).
- We finally present additional evaluation of the FPGA-based approaches against the state of the art software counterparts, namely FiST and YFilter. Our methods offer up to *three orders of magnitude* improvement.

[1]In the rest of the paper we shall use ‘/’ and ‘//’ as shorthand to denote the /child:: axis and /descendant-or-self:: axis, respectively.

The rest of the paper is organized as follows: Section II presents related work. Section IV provides an in depth description of the proposed architectures targeted for XPath query matching. Section V discusses an alternative twig matching approached based on breaking twigs into paths or pairs. Experimental evaluation of the FPGA-based holistic twig matching approach compared to twig matching via pair/path join as well as a comparison of the FPGA-based approaches to the state of art software counterparts appears in Section VI. A discussion of future work appears in Section VII, while conclusions are discussed in Section VIII.

II. Related Work

A. Field Programmable Gate Arrays (FPGAs)

Field Programmable Gate Arrays (FPGAs) are integrated circuits consisting of up to hundreds of thousands of small (in the order of 3,4-input) memory blocks and numerous configurable interconnects. Each N-input memory block, also known as a Look Up Table (LUT), can be used to implement any N-input boolean function. Figure 1 shows a 2-input LUT configured as an AND gate. When combined, LUTs can represent more complex logic functions. We show in Figure 2 the implementation of the 3-input logic function f(A,B,C) = (A AND B) OR C, using two 2-input LUTs. For the purpose of more generic platforms, this is achieved through the use of the configurable interconnects, also known as switch matrices.

As hardware designers express the functionality of their circuit in a hardware descriptive language, their code (description) is passed through complex tools that will analyze the user's circuit description, optimize it for the FPGA at hand, and map it to the available hardware resources. The bit file is now the list of initialization bits of all LUTs and configuration bits of switch matrices.

The performance advantages of such platforms arise from the ability to execute thousands of computations in parallel, relieving the application at hand from the sequential limitations of software execution on Von-Neumann based platforms. The processor "instructions" are the logic functions processing the input data. Another strong advantage of FPGAs is the ability to process streamed data at wire speed, thus resulting in a minimal memory footprint. The aforementioned advantages are shared with Application Specific Integrated Circuits (ASIC). FPGAs however can be reconfigured, are more adaptable to changes in applications and specifications, and hence exhibit a faster time to market. This comes at a slight cost in performance and a considerable one in area, where one functional circuit would run faster on a tailored ASIC, and would require fewer gates.

As traditional platforms are increasingly hitting limitations when processing high volumes of streaming data, researchers are investigating FPGAs for database applications. The Glacier component library is presented in [15] which includes logic circuits of common operators such as selection, aggregation, and grouping for stream processing. In [16] the authors investigated the speedup of the frequent item problem using FPGAs, while in [17] they utilize FPGAs for complex event detection

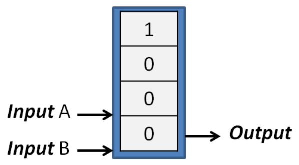


Fig. 1. Implementing a 2-input AND gate using a 2-input LUT.

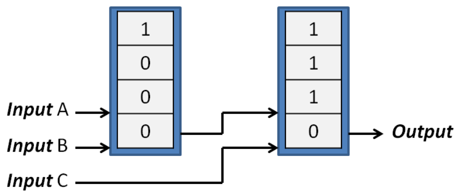


Fig. 2. Implementing *f(A,B,C)=(A AND B) OR C*, a 3-input boolean function, using two 2-input LUTs.

which uses regular expressions to represent events. The use of FPGAs in a distributed network system for traffic control information processing is demonstrated in [18]. Predicate-based filtering on FPGAs was investigated by [19] where user profiles are expressed as conjunctive set of boolean filters. Our focus differs from this work since we consider XML streams and complex query profiles expressed using a fragment of the XPath query language, which includes complex relationships between elements, such as parent-child, ancestor-descendant and wildcards.

B. XML Filtering

The popularity of XML has triggered research efforts to build efficient XML filtering systems. Several software-based approaches have been proposed and can be broadly classified into three categories: (1) FSM-based, (2) Sequence-based, and (3) other.

Finite State Machine(FSM)-based approaches use a single or multiple machines to represent the user profiles [5], [6], [7], [20], [9]. An early work, XFilter [5], proposed building an FSM for each profile, such that each element in the XPath expression becomes a state in the FSM. The FSM transitions 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. The profile is determined as a match when the final state of its FSM is reached. YFilter [6] built upon the work of XFilter and proposed a Non-Deterministic Finite Automata (NFA) representation of user profiles (path expressions) which combines all profiles into a single machine, thus reducing the number of states needed to represent the set of user profiles. Although, YFilter is scalable with the number of profiles, the throughput of the system decreases as the size of the XML document increases. Where as YFilter exploits prefix commonalities, the BUFF system builds the FSM in a bottom-up fashion to take advantage of suffix commonalities in profiles [9]. Several other FSM-based approaches were introduced that use different types of state-machines, such as [7], [21], [22], [23]. Green et. al. [7] proposed a lazy Deterministic Finite Automata (DFA) which has a constant throughput with respect to the size of the query workload; however, lazy-DFA may suffer from state explosion depending on the number of elements and level of recursion in the XML document, and the maximum depth of the XPath expressions. XPush [21] builds a single deterministic push down automaton using a lazy approach, while [23] builds a transducer, which employs a DFA with a set of buffers, and [22] employs a hierarchical organization of push down transducers with buffers.

Sequence-based approaches as in [8], [10] transform the XML document and user profiles into sequences and employ subsequence matching to determine which profiles have a match in the XML sequence. FiST [8] was the first to propose a sequence-based XML filtering system using Prufer sequence encoding. This approach was shown to be more efficient than automata-based approaches since whole twig profiles are processed at once, where as FSM-based approaches traditionally break the twig profiles into paths then perform a join. However, sequence-based approaches are limited to ordered-XPath filtering. Furthermore, the FiST system requires a post-processing phase to filter false positives returned in the subsequence matching phase.

Several other approaches have been proposed [24], [25], [26]. XTrie [24] uses a trie-based data structure to index common sub-strings of XPath profiles, but it only supports the /child:: axis. AFilter [25] exploits both prefix and suffix commonalities in the set of XPath profiles. More recently, Gou and Chirkova [26] have proposed two stack-based stream-querying (and filtering) algorithms, LQ and EQ, which are based on lazy strategy and eager strategy, respectively.

Previous works [27], [28], [29] that have used FPGAs for processing XML documents have mainly dealt with the problem of parsing and validation of XML documents. An XML parsing method which achieves a processing rate of two bytes per clock cycle is presented in [28]. This approach is only able to handle a document with a depth of at most 5, and assumes the skeleton of the XML is preconfigured and stored in a content-addressable memory. These approaches, however, only deal with XML parsing and do not address XPath matching.

The work in [29] proposed 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 used to parse the XML document and to provide partial evaluation of XPath predicates. The results are then reported to the software for further processing. This architecture can only support simple XPath queries with only parent-child axis.

When considering FPGAs, a tempting solution is to implement previously proposed XML filtering approaches on hardware without modification. However, although a given approach maybe efficient for the XML filtering problem on traditional platforms, the same approach may not be the best implementation in hardware, given that FPGAs have completely different design constraints. For instance, DFA was shown to provide advantages over NFA-based approaches like YFilter for XML filtering [7]. However, FPGAs are limited by area and DFAs may suffer from state explosion, thus NFAs are a better approach when considering FPGAs. In [13] we adopted an NFA approach to XML filtering by representing queries as a regular expressions, but this approach did not handle XML documents with recursive elements, which is an important construct in the XML data model. Thus, in the paper we consider a new approach to XML filtering that supports a core fragment of the XPath query language as well as recursive elements in XML documents, takes advantage of parallelism found in the XML filtering problem, and takes FPGA design constraints into consideration.

C. Previous Research and Current Contributions

Mitra el. al [13] were the first to propose a pure-hardware, FPGA-based solution to the XML filtering problem by efficiently evaluating path profiles with different types of navigation directions ('/' as well as '//' axis) over the streaming XML documents. Query profiles were represented as regular expressions and each profile compiled to a NFA. The FPGA, with a processing rate of one byte at a time, assumed a stream of sequential XML documents. As each document is being parsed, all NFAs were operating in parallel, which helped achieve considerable throughput when compared to the software counterparts. However, the presented method suffered from false dismissals in the presence of recursive tags in the XML document. Furthermore, it supported simple absolute paths (not twigs) and did not handle wildcards ('*') in query profiles.

In [14], we presented a stack-based approach to XPath processing which allows for '/' and '//' axis, as well as wildcard nodes in the query profile, and recursion in the XML document. This method proved to consume less area on the FPGA and provide higher throughput, when compared to the implementation proposed in [13]. However, to process a complex twig structure, the twig must be broken into root-to-leaf paths and a extra join step is required to join the results. The YFilter system takes such an approach by breaking twigs into absolute paths and inserting each path into the NFA to be matched. The match location for each path is recorded and utilized during the post-processing phase, where the Nested Path Filter is applied to join the paths. However, we are unable to adopt such an approach since the match location for each step in the XPath expression must be stored, and FPGAs have limited resources. In this paper, we present a method which performs twig matching holistically on the FPGA by compiling each query profile into a hardware circuit. We also provide a dynamic programming formulation for the stack operations, as explained in Section IV. We compared this approach against other FPGA-based possible approaches, such as root-to-leaf path matching or parent-child/ancestor-descendant pair matching. These approaches require less area,

```
Path        ::=  Step  |  Path  Step

Step        ::=   Axis  NodeTest  |  Axis  NodeTest  [Path]*

Axis        ::=  '/'  |  '//'

NodeTest    ::=  name  |  '*'
```

Fig. 3. Supported query grammar represents a core fragment of the XPath query language, where 'name' denotes element/tag labels, while '/', '//' and '*' denote the child axis, descendant axis and wildcard node test, respectively.

however, introduce false positives. This comparison covers the full spectrum of granularity matching when considering XML filtering on FPGAs.

III. XML Filtering

XML filtering is the core problem in a Pub-Sub system. Formally, given a collection of user profiles (expressed in the XPath query language) and a stream of XML documents, the objective of the filtering algorithm is to determine, for each document D, the set of XPath expressions that have at least one match in D.

A. XML Streams

XML documents are received in a streaming fashion, where they are parsed by a SAX parser [30]. The SAX parses the input stream and generates two core SAX events, startElement(*name*) and endElement(*name*), which are generated, respectively, when an open or close tag of a element arrives. For presentation simplicity, in this paper we treat attributes and elements similarly.

B. Query Language

XPath [11] is a popular language for querying XML data. In this paper, we address a core fragment of XPath that includes element labels, wildcards, and the /child:: and /descendant-or-self:: axis. The grammar of the supported query language is given in Figure 3. The query consists of a sequence of location steps, where '/' denotes the child axis, '//' denotes the descendant axis, and '*' is the wildcard node test. Extending the supported grammar to include equality-based predicates is straightforward, however, supporting complex value-based predicates is a challenging problem which we reserve for future work.

IV. Holistic Twig Matching

In this section, we present both a high-level and a detailed overview of the proposed filtering mechanism.

A. High-Level System Overview

Field Programmable Gate Arrays (FPGAs) are a suitable platform for a range of applications that can be parallelized. Using custom (and reconfigurable) circuitry tailored for the application at hand, speedup can be achieved when compared to the software counterpart, the latter suffering from the sequential aspect of computing, and from being memory

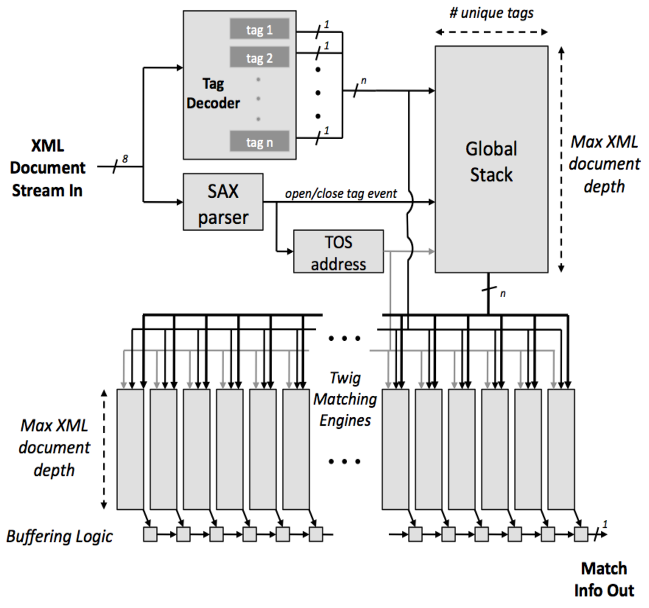


Fig. 4. High-Level XML Filtering System.

bound. Figure 4 depicts the underlying system and components that are mapped to the FPGA at hand. Here, streaming the XML document, parsing, and filtering for thousands of queries is simultaneously achieved. In addition to inter-query parallelism, we are able to extract parallelism internal to each query, where the matching of all tags belonging to the query is achieved simultaneously. Moreover, with the proposed approach supporting, but not being limited to, a streaming XML interface, the memory footprint is virtually non-existent. Once the stream of the XML document is complete, the matching states of all twig filtering engines are reported.

B. Parsing Sub-Structure

Looking at Figure 4, the parser consists of four main components: a hardware implementation of a SAX-like parser, a Top Of Stack (TOS) address generator, the Tag Decoder, and the Global Stack. The task of the SAX parser is to recognize XML delimiters, such as '<', '</', and '>'; this is achieved through the help of a finite state machine: when a new tag is opened, the TOS address generator is notified from the state machine, and will translate that event into a push event, thus incrementing the current TOS address. Similarly, a closed tag event is seen as a pop event, thus decrementing the TOS address.

Alongside the SAX parser is a tag decoder, which notifies the twig matching engines and the Global Stack of the tag ID respective to the open/close event that had just occurred. The tag decoder is implemented through a Content Addressable Memory (CAM) - a fully associative memory, requiring a single hardware cycle to search through all memory contents. When initially programming the FPGA, each CAM entry is initialized to a unique tag, such that all CAM entries cover the set of tags used across all twigs. Each CAM line outputs a

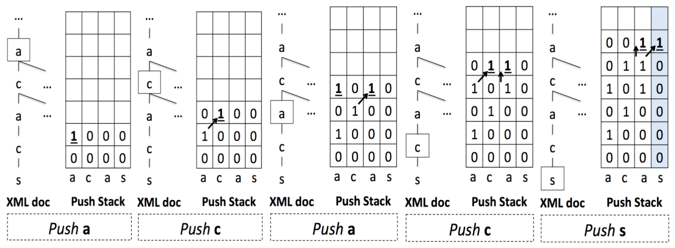


Fig. 5. An event by event overview of the matching of path *a/c/a//s*.

single bit, indicating whether or not the buffered tag matches the stored information in the line. Centralizing the tag decoder helps reducing the redundancy across twig matching engines. Moreover, making use of a decoded tag ID is highly beneficial, where a single bit indicates whether or not a given tag was found.

Finally, a global stack is introduced, where, in case of a push, the tag ID is stored in the stack address as noted by the TOS address generator. The number of bits representing the tag ID is the number of unique tags. The top of the global stack (TOGS) is forwarded to all twig matching engines.

C. Push Stacks for Path Matching

Matching for a twig consists of two parts working conjointly, namely, matching the root-to-leaf paths of the twig (Section IV-C), and appropriately joining the matched paths while reporting back to the root (Section IV-D).

The matching of each root-to-leaf path requires what we refer to as a *push stack*. We next explore the motivation, functionalities and properties of the push stack.

*1) **Root-to-Leaf Path Matching**:* In order to successfully match a twig, the partial matching of every path from root to leaf belonging to that twig should be achieved. In order to match a single path, we employ a novel dynamic programming approach, where the dynamic table is a stack, whose top of stack address is given by the TOS address generator operating as explained earlier. Every stack column represents a path node, and every stack row represents a document node.

The recurrence equation applied to each cell $C_{i,j}$ on a push event, is given by:

$$C_{i,j} = \begin{cases} 1 \; if \begin{cases} C_{i-1,j-1} = 1 \;\; \&\& \;\; \textit{tag of the node mapped} \\ \quad \textit{to the } j^{th} \textit{ column was opened} \\ OR \\ C_{i-1,j} = 1 \;\; \&\& \;\; \textit{the node mapped to the } j^{th} \\ \quad \textit{column is followed by // in the path} \end{cases} \\ 0 \;\; \text{otherwise} \end{cases}$$

where:

- $1 \leq i \leq$ maximum XML document depth
- $1 \leq j \leq$ number of Path nodes

When the j^{th} stack column stores a '1', this would indicate that a path of length *j*, from root to the j^{th} node has matched. A path starting from the root and being of length $j+1$, can only be matched if a path starting from the root and being of length j has matched; thus the common sub-problem property. When the last column in a stack stores a '1', the entirety of the path would have been matched successfully.

Note that the root node of the path does not require the propagation of the match state stored at $C_{i-1,j-1}$. Moreover, when a wildcard is mapped to a specific column, there is no need to check for the tag of the node mapped to that column, as any tag is a valid one - thus the added level of freedom offered by wildcards.

Figure 5 depicts an event-by-event example of the matching of the path *a/c/a//s*, as a sample XML document is being traversed. Notice how, from the 3^{rd} to the 4^{th} event, a '1' was allowed to propagate horizontally upwards in the 3^{rd} column. More on the matching of ancestor-descendants is covered in Section IV-C.3.

*2) **Push Stack Properties**:* When matching all the root-to-leaf paths in a given twig, all the paths are mapped to a single push stack, where every node in the twig appears exactly once. Furthermore, the diagonal propagation of a '1' does not necessarily propagate between two adjacent columns, rather from the column of a parent (ancestor) to the column of its child (descendant). The properties of Push Stacks are summarized as follows:

- Push stacks update on push events only.
- A '1' propagates diagonally upwards from and to any column connecting a parent or ancestor to a child or descendant, respectively.
- Only in a '//' column, a '1' propagates vertically upwards, to indicate matches to all descendants.
- A query node could require two push stack columns, one for '/' and another for '//', and this arises when:
  - The node in question has at least one child and at least one descendant in the query (thus requiring both a '/' and a '//' column)
  - Or, if the node has only descendants, and is itself the direct child (not descendant) of its parent node. The node requires a '//' column by default, having descendants. A '/' column helps in determining exactly where that node occurred in the XML document, a feature needed when reporting matches using pop stacks [2] (Section IV-D).

*3) **Supporting Ancestor/Descendant Relationships**:* The matching state of a sub-path ending in a '//' relation should be reported to any node in the XML tree, after the leaf (followed by '//') of the sub-path has been matched, and prior to popping it.

In Figure 6, sub-trees numbered according to the order by which they are encountered while streaming the XML doc. Moreover:

- The *Matched Sub-Path* can consist of 1 or more nodes.

[2]If the node N has only descendants, but is itself a descendant of another node R, then a '//' column suffices, since any tag similar to N's in the XML document, following a matched R, is a valid one. Here an extra check needs to be made, to ensure the tag being parsed is the same as N's, and that is unless N is a wildcard.

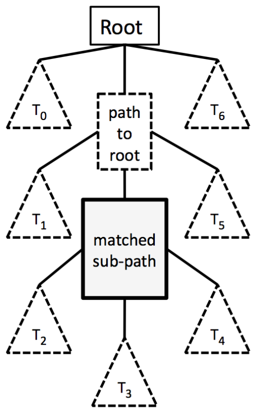


Fig. 6. Generic view of any XML document with regards to any matched path.

- Any of *T0 . . . T6* could consist of zero or more nodes.
- The *Path To Root* can consist of 0 or more nodes.

Reporting the matched state of sub-paths having a leaf that is followed by '//' to descendants using push stacks takes place as such:

- T0, T1, and T2 will not see a matched sub-path because by then the latter would have not matched; even after being in a match state, none of those subtrees are visited again.
- T3 is visited after the matched sub-path is matched, and the matched state of that sub-path is visible to this subtree, since push stacks report matches on pop, and T3 is reached after a series of push events.
- T4, T5, and T6 are visited after the matched sub-path is matched, and following a series of pop events. However, the matched state of the matched sub-path is also not visible here, as the leaf of the latter would be popped then, and the matched state of the sub-path with it.

Thus, the matched state can only be seen by any node visited after the leaf of the matched sub-path, and prior to popping the leaf of the latter.

4) ***Mapping Algorithm:*** So far, mapping twig nodes to stack columns has been achieved through a one-to-one mapping. Nonetheless, considerable FPGA resources can be saved through stack compaction, when several twig nodes are mapped to the same stack column. Some extra checks will be needed to insure the correct propagation of a '1' from a parent to its child. In [14], we witnessed a 30% average saving in resource utilization for paths matching.

Algorithm 1 refers to the twig node to stack column mapping algorithm, with column compaction enabled. The basic rules state that nodes followed by '//', and wildcards, map to their own restricted columns. Moreover, nodes having similar tags are not allowed to map to the same column. Nodes that are followed by both '/' and '//' in a twig are ones that require two push stack columns.

This algorithm returns the required stack width for a given query, and the mappings of the nodes to stack columns.

Algorithm 1 Twig Node to Push Stack Column Mapping

```
1  GLOBAL $stack\_width \leftarrow 0$ {Number of Push Stack Columns}
2  for every query node N do
3    if the node requires a single Push Stack column then
4      if the node is a wildcard, and-or is an ancestor of other query node(s) then
5        $stack\_width++$
6        Assign a new column restricted to this node
7      else
8        Map to the column given by Mapper(tag of N)
9      end if
10   else
11       • With regards to N//:
12         $stack\_width++$
13         Assign a new restricted column
14       • With regards to N/, map to the column given by Mapper(tag of N)
15   end if
16 end for
```

Procedure 2 Mapper(tag T)

```
1  $R \leftarrow 0$
2  Set R to be the stack column corresponding to the most recent occurrence of T/ in the push stack
3  for each stack column C, s.t. $R < C \leq stack\_width$ do
4    if no ancestor or wildcard nodes map to this column then
5      return C
6    end if
7  end for
8  $stack\_width++$
9  return $stack\_width$
```

D. Pop Stacks for Joins

Recall that matching a twig consists of two parts working conjointly, namely the push stack , and the pop stack, as seen in Figure 8. Using the push stack, matching any root-to-leaf path is achieved. However, matching all paths in a twig does not imply matching the twig, unless all paths match in the correct positions in order to form the twig at hand. The *pop stack* helps us achieve this task.

For the remainder of this section, we show why a pop stack is needed, and its properties.

1) ***Leaf-to-Split Node Matched Path Reporting:*** Let us assume a simple twig of the form *a[b/c]/d/e* that appears in a streamed XML document. Thus, (assuming) the path *a/b/c* will be visited first, then each of *c* and *b* will be popped (in that order), thus rendering the twig root node *a* at the top of the stack again. At this point, there is no way to tell whether the twig will be found in the document, since the path *a/d/e*

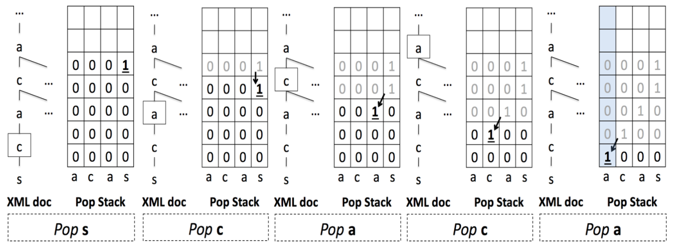


Fig. 7. An event by event overview of the reporting of the matched state of path *a/c/a//s*.

would not have matched yet. However, when the leaf node c was encountered, it was noted in the push stack that the first path matched. This information has been lost when c was popped. A pop stack is needed here to report back to the twig root that the first path matched.

For instance, Figure 7 illustrates an event-by-event example of the reporting of the matched state of path *a/c/a//s*, following what was shown in Figure 5.

In more generic terms, the initial task of the pop stack is to **report** the matched state of a root-to-leaf path, to the nearest split node, corresponding to the leaf of that path. In a pop stack, a split node is in a matched state only if all of its children/descendants have been **reported** as being matched.

We achieve this task using dynamic programing, where the dynamic programming table is a stack, whose top of stack address is given by the address generator. Every stack column represents a path node, and every stack row represents a document node.

For simplicity, let us assume that queries are broken down into P ***exclusive*** split-node (or inclusive root) to ***inclusive*** split-node (or leaf) paths. Let us also assume that each path utilizes its own push and pop stacks.

Each path K is mapped to both the Kth push and Kth pop stacks, and is of length N_K.

For the K^{th} path, the recurrence equation applied to each cell $D_{i,j,k}$ of the pop stack- *representing the reported match state of node* $n_{j,k}$ - on a pop event is given by:

$$D_{i,j} = \begin{cases} 1 \; if \begin{cases} \begin{cases} (a) C_{i+1,N_K,K} = 1 & \text{if ((j = N}_K\text{) and } n_{N_K,K} \text{ is a twig leaf)} \\ (b) D_{i+1,0,L} = 1 \; \forall \; L \in \{children \; of \; n_{N_K,K}\} & \text{if ((j = N}_K\text{) and } n_{N_K,K} \text{is a split node)} \\ (c) D_{i+1,j+1,K} = 1 & \text{if (j} < N_K\text{)} \end{cases} \\ OR \\ (d) D_{i+1,j,K} = 1 \;\; \&\& \;\; n_{N_K,K} \text{ is preceded by //} \end{cases} \\ 0 \;\; \text{otherwise} \end{cases}$$

where:

- $C_{i,j,K}$ represents a cell in the Kth push stack
- $1 \leq i \leq$ maximum XML document depth
- $1 \leq j \leq N_K$

The recurrence equation encapsulates four main cases to report matches on a pop. If a node is a leaf in the twig (a), then a match is reported by propagating the corresponding output from the push stack. If a node is a split node (b), then a match can be reported only if all of its children/descendants in the twig respectively report matches. Otherwise (c), a match is reported by propagating the reported match state of the node's single child/descendant. Matching for *unordered* twigs is supported, since in (b), checking for all children is achieved with no enforced order.

Only if a node is preceded by '//' (d), then the reported match state is allowed to propagate vertically downwards.

Figure 8 illustrates a high-level view of the underlying matching mechanism when targeting the matching of a twig as a whole. Here, a single push stack is required, the width of which is defined by the mapping algorithm (introduced in IV-C.4), and a single pop stack, the width of which is the number of nodes in the twig. The split nodes matching logic consists of the AND-ing logic as noted in (b) as part of the recurrence equation. This logic further requires some cells from the push stack to help determining the exact position of the split node in the document.

*2) **Pop Stack Properties**:* Here are the properties of Pop Stacks:

- Pop stacks update both on push and pop events, s.t.:
  - On a push, always force writing a '0' (to reset to a new state).
  - On a pop, only a '1' can propagate downwards, but never a '0' (in order to not erase previous states).
- A '1' propagates diagonally from and to any column connecting a parent or ancestor to a child or descendant, respectively.
- Only in a '//' column, a '1' propagates vertically downwards, to indicate matches to all ancestors residing only on the path from root to the node mapped in that respective column (see Section IV-D.3 and the supporting Figure 6).

*3) **Supporting Ancestor/Descendant Relationships**:* Reporting the matching state of a sub-path preceded by a '//' should be visible to the ancestor of that sub-path. Referring to Figure 6, using pop stacks, reporting matched path rooted by a node preceded by '//' occurs as such:

- T0, T1, and T2 will not see a matched path because by then the latter would have not matched; even after being in a match state, none of those subtrees are visited again.
- T3 is visited after the matched path is matched; however, the matched state of that path is not visible to this subtree, since pop stacks report matches on pop, and T3 is reached after a series of push events.
- T4, T5, and T6 are visited after the matched path is matched, and following a series of pop events. However, the matched state of the matched path is also not visible here, as some push events are required to go into the tree nodes, and pop stack contents do not propagate on pushes.

Thus, the matched state can only be seen by the root and the path to root as illustrated above, given that this path includes

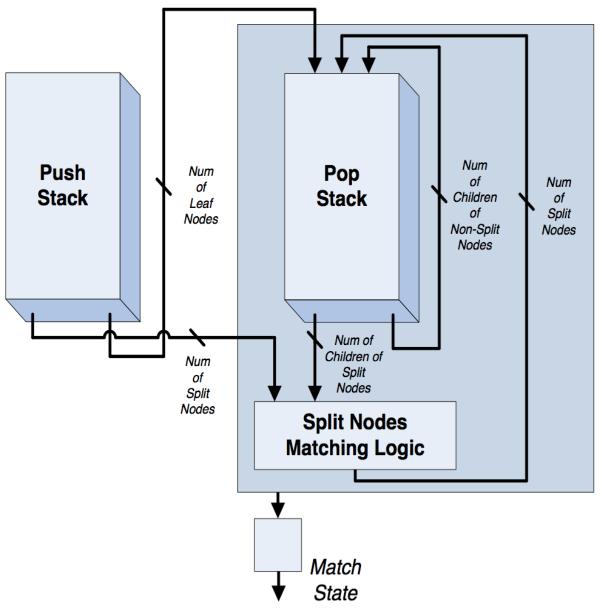

Push Stack
Num of Leaf Nodes
Pop Stack
Num of Children of Non-Split Nodes
Num of Split Nodes
Num of Split Nodes
Num of Children of Split Nodes
Split Nodes Matching Logic
Match State

Fig. 8. Generic view of a holistic twig matching engine, using a push and a pop stacks.

the ancestor of the sub-path.

*4) **Mapping Algorithm**:* Here, the mapping of twig nodes to pop stack columns is kept simple, where each node maps to its own respective column. Thus, the width of pop stacks is defined by the number of nodes to each twig mapping to that stack. We keep the exploration of pop stack column compaction as part of our future work.

V. Breaking Twigs into Paths and/or Pairs

In this section, instead of processing a twig holistically, we consider different granularities of twig matching. Based on the following approach, the twig profile is decomposed into smaller parts, and the 'filtering' algorithm is performed on the smaller parts. We have considered two methods:

- **Path matching:** Each twig profile is broken down into root-to-leaf paths. For instance, the twig {a/b[c]//a} is broken down into paths {a/b/c} and {a/b//a}.
- **Pairs matching:** Each twig profile is broken down into parent-child or ancestor-descendant pairs. For instance, the twig {a/b[c]//a} is broken down into pairs {a/b}, {b/c} and {b//a}.

Note that for both methods, paths and pairs may be common among twig profiles; specifically, pairs are more common among twigs, than are paths. Thus, this approach exploits commonality among profiles.

Figure 9 provides an overview of the Path/Pair filtering approach. Twigs are split into several root-to-leaf paths or parent-child/ancestor-descendant pairs. Every path/pair matching engine is followed by a match state buffer. In case a path/pair match state is true, that state is held for the document's entirety. After the document is processed, a join step is required to verify all parts (paths or pairs) that represent

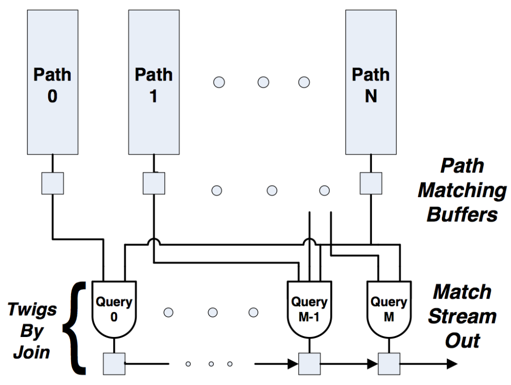


Fig. 9. Generic view of any XML document with regards to any matched path.

a twig profile were matched. Thus, every twig requires a single AND gate in order to join all the partial paths/pairs that constitute it. In case of a single path/pair not matching, the twig's matching state is marked as false.

A. Advantages

The path/pair approaches provide several advantages when compared with holistic matching. Here, a push stack suffices to match a path/pair, since no join step is required to match each of the latter. Therefore, no pop stack is required. Moreover, the smaller granularities constituting a twig profile may be common across the profile collection, thus exploiting commonalities. Therefore, the representation of more paths/pairs and ultimately more profiles on a single FPGA is achieved when compared to the holistic approach.

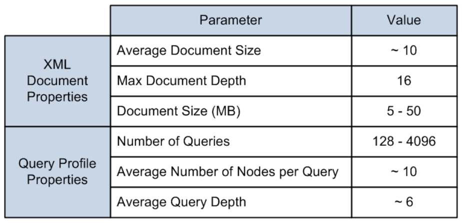
Finally, since each of the matching engines requires simpler hardware, a higher throughput can be achieved on the FPGAs.

B. Disadvantages

Although the path/pair approaches have advantages in area utilization and higher throughput, false positives are also introduced. Note, the join step performed is simply checking that all parts (paths/pairs) of a profile were matched; however, this step does not verify that these parts matched at the correct locations. Hence, the reported set of matched profiles will include a percentage of false positives. Thus, this technique has advantageous for applications where false positives are allowed or when the verification cost of the reduced profile set is small.

VI. Filtering System Evaluation

In this section, we evaluate the proposed hardware architectures, and compare them to two of the state-of-the-art software counterparts, namely FiST[8] and YFilter[6]. For the experiments, we utilized the DBLP DTD provided by [31] to generate XML documents and user profiles. XML documents and query profiles were generated using the ToXGENE XML Generator[32] and YFilter query generator [6], respectively. Furthermore, in all datasets, we set the number of unique tags to 64, each consisting of two bytes. The experimental



	Parameter	Value
XML Document Properties	Average Document Size	~ 10
	Max Document Depth	16
	Document Size (MB)	5 - 50
Query Profile Properties	Number of Queries	128 - 4096
	Average Number of Nodes per Query	~ 10
	Average Query Depth	~ 6

Fig. 10. Experimental Parameters

parameters are listed in 10. We make use of four datasets, namely:

- **Dataset 1**: the probability of occurrence of '*' and '//' in the queries is set to 5%.
- **Dataset 2**: the probability of occurrence of '*' and '//' in the queries is set to 10%.
- **Dataset 3**: the probability of occurrence of '*' and '//' in the queries is set to 15%.
- **Dataset 4**: the probability of occurrence of '*' and '//' in the queries is set to 20%.

A. Hardware System Evaluation

Our hardware platform consists of a Xilinx Virtex 5 LX330 FPGA[33]. All the push and pop stacks are implemented using on-chip Distributed Memory (DMEM) blocks, available on Xilinx FPGAs. We provide a thorough evaluation of the five hardware approaches that have been presented so far in this paper. These are:

- **Holistic**: each query is mapped onto both a push and pop stack.
- **Pairs_offChip**: each query is split into pairs, but the join step is not implemented.
- **Pairs_onChip**: each query is split into pairs, with the join step implemented on the same FPGA.
- **Paths_offChip**: each query is split into paths, but the join step is not implemented.
- **Paths_onChip**: each query is split into paths, with the join step implemented on the same FPGA.

Excluding the join step in two of the designs is, first, aimed at providing a better study of the effect of the join step on resource utilization and throughput, and second, proposed for designs where the join step is not needed, or can be performed off chip on a second FPGA, or in software, depending on the requirements of the application at hand.

We show in Figure 12(a) the resource utilization percentage on the target FPGA, while doubling the number of twigs[3]. The holistic approach is the least scalable, in contrast with the breaking of twigs into paths and pairs, where the commonalities across queries are exploited. Naturally, there are more common pairs than there are paths. However, looking at Figure 11, the percentage of false positives is largest when

[3]In the remainder of Section VI-A, we make use of query dataset 2, having 10% occurrence of '//' and '*'.

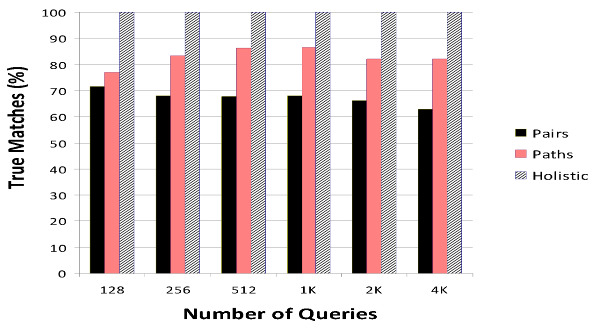


Fig. 11. The percentage of true matches as reported from several hardware approaches.

using pairs. The holistic approach, on the other hand, yields no false positives.

In Figure 12(b), while doubling the number of queries, we show the throughput of all proposed approaches, assuming a stream of one byte per cycle. As the FPGA utilization increases, the throughput decreases, as the task of placement and routing of components on the FPGA is hardened. For a given number of queries, the holistic approach exhibits the lowest throughput, being the more complex of all five. The pairs however, being the simplest, almost always demonstrates a higher throughput for a given query set.

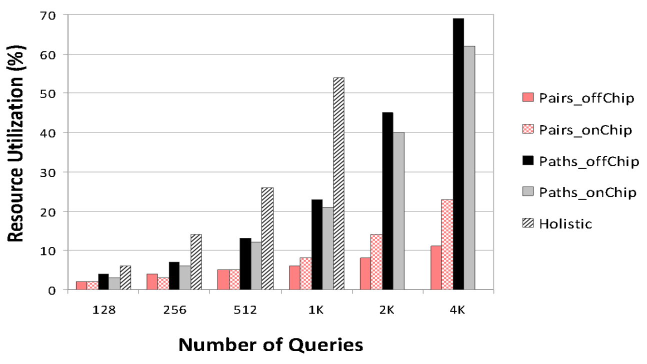
With regards to the pairs, excluding the join step shows to benefit throughput by much when compared to the inclusive join step approach. That behavior is however not always true with paths, as there are fewer paths than pairs, and the effect of the join step is not as harsh. Overall, almost all approaches record a throughput higher than 150 MB/s, and averaging more than 200 MB/s.

In order to further study all five approaches, we define the **true work per unit area** as:

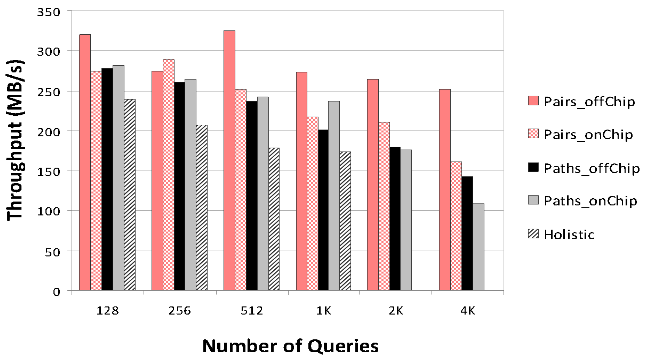
$$\frac{Throughput \times Number\ of\ Queries \times (1 - False\ Positives(\%))}{Area\ Utilization(\%)}$$

Hence, as the throughput and the number of queries handled increases, true work per unit area of a given approach increases. Conversely, as the area utilization and the percentage of false positives increases, the true work per unit area decreases.

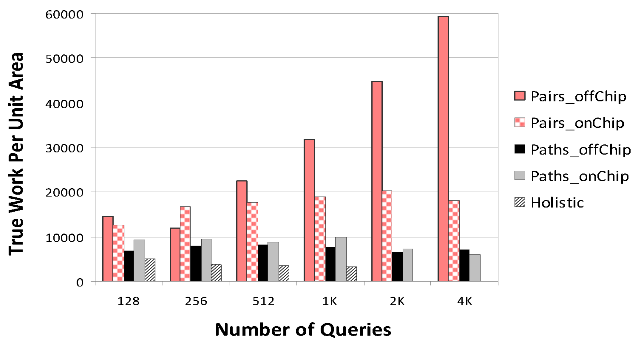
We plot this metric in Figure 12(c). As the Holistic approach is mostly affected by high resource utilization across all query sets, the true work per unit area is the least of all approaches.. On the other hand, the pairs approaches dominate, even while exhibiting the highest percentage of false positives. The high scalability aspect of this approach is due to both the low resource utilization, and the superior throughput. The pairs approaches' depict an almost constantly boosting true work per unit area with the number of queries increasing; as the number of queries increases, for a given label set, the number of common pairs across queries also increases, thus rendering this approach the most scalable. However, it should be kept in mind that *offChip* approaches do not perform a join step.



(a) Resource utilization of the proposed hardware architectures on a V5LX330 FPGA.



(b) Throughput of the proposed hardware architectures on aV5LX330 FPGA.



(c) True work per unit area of the proposed hardware approaches.

Fig. 12. Area Utilization (a), Throughput (b) and True Work per Unit Area (c) of the proposed hardware approaches for dataset 2 (10% occurrence of '//' and '*' in the queries). Note that the join step is not performed when using *off-chip* filtering methods.

B. Hardware/Software Performance Evaluation

We provide a comparison of the proposed hardware approaches against two state of the art software approaches, namely, YFilter[6] and FiST[8]. These approaches where chosen since they represent the two main techniques used for the XML filtering problem. The software approaches were run on a quad core 2.33GHz Intel Xeon machine with 8GB of RAM running Linux Red Hat 2.6. YFilter supports unordered query matching by breaking twigs into root-to-leaf paths and building a unified NFA over the set of paths. After path matching, an additional join step is required to join the paths at the split nodes. FiST, on the other hand, only supports ordered query matching. The XML document and set of queries are transformed into their Prufer sequence representation and subsequence matching is performed to determine if a match exists. FiST also requires a post-processing phase to filter false positives.

The throughput of YFilter and FiST is shown in Figure 13. On average, YFilter achieves a throughput of 1.7, 1.2 and 0.3 MB/s for 5, 25 and 50 MB documents, respectively. FiST achieves a higher throughput of 3.88, 2.58, 1.40 MB/s for 5, 25, and 50MB documents, respectively, since it processes twigs in a holistic manner rather than processing individual paths. Although YFilter and FiST scale with increasing query workload, it is clear, however, that both approaches do not scale with increasing document size. In comparison, the holistic FPGA-based approach achieves an average of 200x speedup for 1K queries, and up to three orders of magnitude speedup.

It should be noted that XML stream parsing is not the bottleneck for the software approaches. For the given experimental setup, using the Xerces Java parser, we were able to achieve a throughput rate of 23.1, 57.5, and 72.2 MB/sec for 5, 25 and 50 MB documents, respectively. Thus, the profile matching process is contributing to the low throughput of the software approaches, not the XML parsing. Whereas, for the FPGA-based approaches that we present, parsing is now the bottleneck, as the overall throughput is directly proportional to the number of bytes of XML that can be parsed per cycle. The query matching engines can process up to one SAX event per cycle; however, in practice, every event requires at least three bytes of XML ('<', '>', and a one-byte label). Nevertheless, the XML parsing problem is orthogonal to our current research, and other researchers have proposed complex FPGA-based parsers which are able achieve an average throughput of two bytes of XML per cycle (a peak 4 bytes per cycle) [28]. Using the proposed approaches in this paper, we are able to take advantage of all the effective bandwidth provided by the parsers.

In Figure 14, we show the effect of increasing the probability of occurrence of '//' and '*' in the query dataset, on software throughput, for a 25MB XML document. The performance of both YFilter and FiST highly depends on the complexity of the queries. As the occurrence of '//' and '*' reaches 15%, the software throughput degrades. Recall, both YFilter and FiST are composed of two phases, query matching and verification/join phase. High occurrence of '//' and '*' lends to less selective queries and introduces more false positives in the query matching phase, thus requiring more computation time for the verification/join phase. Furthermore, high occurrence of '//' and '*' degrades performance of the software approaches since expensive computations are performed to verify the specified pattern is satisfied. This performance degradation is not applicable to FPGA-based systems where the circuitry is the same for dealing with any tag, or any relation. Here, at

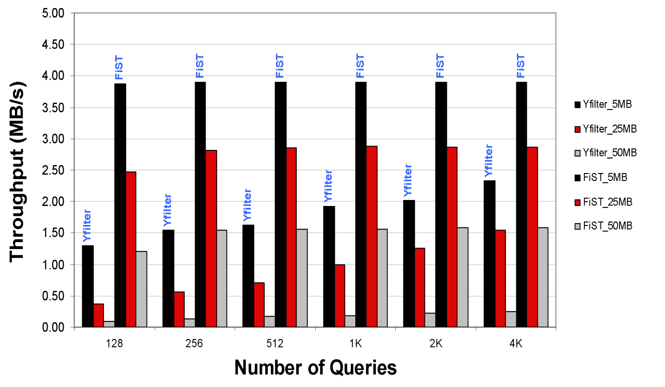


Fig. 13. Throughput of FiST and YFilter when using 5, 25, and 50MB XML documents, and queries for dataset 2 (10% occurrence of '//' and '*' in the queries).

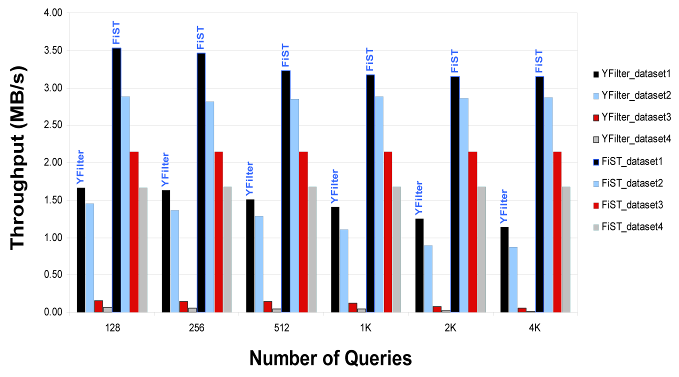


Fig. 14. Throughput of FiST and YFilter for a 25MB XML document, while increasing the probability of occurrence of '//' and '*' in the queries.

1K queries of dataset 4, the holistic FPGA-based approach yields an acceleration of 7000X and 2300X when compared to YFilter and FiST, respectively. It should be noted that FiST only deals with ordered twigs. While our current approach addresses unordered twig matching, enforcing order can be achieved through the addition of more checks at the recurrence equation level.

VII. Future Work

In our future research, we will focus on two main tasks, namely, supporting value based predicates and extending our study of XML filtering to GPU platforms.

In order to support value based predicates, every condition will be mapped to some hardware logic, and propagation of match states in the dynamic programming table would also depend on the condition assessment. Equality evaluation would be achieved through the use of variable word length content addressable memories; alternatively, inequality requires hardware comparators. All these added components will affect resource utilization, and we will be evaluating the impact of value based predicates on both resource utilization and performance (maximum achievable throughput).

Graphics Processing Units (GPUs) [34], [35] are being increasingly used in scientific research, as they are able to efficiently handle SIMD (Single Instruction Multiple Data), memory-intensive streaming data applications [36], [37], [38], [39]. GPUs incorporates several simple processing cores designed for mathematical computation, and do not handle control kernels efficiently. GPUs are programmed using high level software languages [40].

XML filtering is one data intensive application that requires many operations to be applied to every single data element; checking for state transitions (updating the dynamic programming table), joining partial matches, all for every node of every query, are such operations that need to be performed with the occurrence of every SAX event. XML filtering is thus categorized as a MIMD (Multiple Instruction Multiple Data) application, one that we suspect cannot be efficiently mapped to GPUs. However, one advantage to GPUs is the ability to support frequent query updates, where compiling and reprogramming suffices to *implement* new queries, knowing that compiling for GPUs requires negligible time when compared to FPGA synthesis, placement and routing. We would be looking into porting known XML filtering approaches to GPUs, or designing a newer one that can be efficiently handled by GPUs.

VIII. Conclusions

In this paper, we presented a novel FPGA-based architecture to address the XML filtering problem. Using custom stack generation, our architecture is the first providing full support for twig pattern matching, including parent-child('/') and ancestor-descendant('//') axes, wildcard nodes, and accounting for recursion in the XML document and queries. 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 twig matching engines operating in a parallel fashion. In addition to being able to match thousands all queries in parallel, through dynamic programming on FPGAs, we exploit parallelism by simultaneously matching for all nodes in the query.

We were able to show that holistic twig matching on the FPGA achieves an average of 175MB/s throughput for 1K queries. Compared to state of the art software approaches, the holistic FPGA-based approach yields up to three orders of magnitude throughput increase. We note that the performance of the software approaches do not scale when the size of the input stream increases, and as the queries are more complex, while the throughput of the FPGA-based approach is constant.

Furthermore, we presented a comparison of our holistic FPGA-based approach against path-based and pair-based approaches, which break twigs into root-to-leaf paths and parent-child/ancestor-descendant pairs, respectively. We compared the various approaches based on the true work per unit area on the FPGA. Our comprehensive experiments on the different granularities of query matching considers throughput, area utilization and false positives generated by the approaches, thus allowing the selection of the most suited approach for the application on hand. Future work will examine further optimizations of the holistic twig matching architecture.

REFERENCES

[1] T. Milo, T. Zur, and E. Verbin, "Boosting topic-based publish-subscribe systems with dynamic clustering," in *SIGMOD: Proceedings of the 2007 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2007, pp. 749–760.

[2] D. S. Rosenblum and A. L. Wolf, "A design framework for internet-scale event observation and notification," *SIGSOFT Softw. Eng. Notes*, vol. 22, no. 6, pp. 344–360, 1997.

[3] P. T. Eugster, P. A. Felber, R. Guerraoui, and A.-M. Kermarrec, "The many faces of publish/subscribe," *ACM Comput. Surv.*, vol. 35, no. 2, pp. 114–131, 2003.

[4] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and D. Shasha, "Filtering algorithms and implementation for very fast publish/subscribe systems," in *SIGMOD: Proceedings of the 2001 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2001, pp. 115–126.

[5] M. Altinel and M. J. Franklin, "Efficient filtering of XML documents for selective dissemination of information," in *VLDB: Proc. of the 26th Intl. Conf. on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000, pp. 53–64.

[6] Y. Diao, M. Altinel, M. J. Franklin, H. Zhang, and P. Fischer, "Path sharing and predicate evaluation for high-performance xml filtering," *ACM Trans. Database Syst.*, vol. 28, no. 4, pp. 467–516, 2003.

[7] T. J. Green, A. Gupta, G. Miklau, M. Onizuka, and D. Suciu, "Processing XML streams with deterministic automata and stream indexes," *ACM Trans. Database Syst.*, vol. 29, no. 4, pp. 752–788, 2004.

[8] J. Kwon, P. Rao, B. Moon, and S. Lee, "FiST: scalable XML document filtering by sequencing twig patterns," in *VLDB: Proc. of the 31st Intl. Conf. on Very Large Data Bases*. VLDB Endowment, 2005, pp. 217–228.

[9] M. M. Moro, P. Bakalov, and V. J. Tsotras, "Early profile pruning on XML-aware publish-subscribe systems," in *VLDB: Proc. of the 33rd Intl. Conf. on Very Large Data Bases*. VLDB Endowment, 2007, pp. 866–877.

[10] M. Salloum and V. Tsotras, "Efficient and scalable sequence-based XML filtering system," in *WebDB: Proc. of 12th Intl. Workshop on the Web and Databases*. ACM, 2009.

[11] "XML Path Language (XPath) Version 1.0," 1999. [Online]. Available: www.w3.org/TR/xpath

[12] Z. Guo, W. Najjar, F. Vahid, and K. Vissers, "A quantitative analysis of the speedup factors of FPGAs over processors," in *FPGA: Proc. of the 12th Intl. symposium on FPGAs*. New York, NY, USA: ACM, 2004, pp. 162–170.

[13] A. Mitra, M. R. Vieira, P. Bakalov, W. Najjar, and V. J. Tsotras, "Boosting XML filtering through a scalable FPGA-based architecture," in *CIDR: 4th Conference on Innovative Data Systems Research*. ACM, 2009.

[14] R. Moussalli, M. Salloum, W. Najjar, and V. Tsotras, "Accelerating XML query matching through custom stack generation on FPGAs," in *HiPEAC: High Performance Embedded Architectures and Compilers*. Springer Berlin / Heidelberg, 2010, pp. 141–155.

[15] R. Mueller, J. Teubner, and G. Alonso, "Streams on wires: a query compiler for FPGAs," *Proc. VLDB Endow.*, vol. 2, no. 1, pp. 229–240, 2009.

[16] L. Woods, J. Teubner, and G. Alonso, "Complex event detection at wire speed with fpgas," in *VLDB: Proceedings of the 2010 Very Large Data Bases (VLDB)*, 2010.

[17] J. Teubner, R. Müller, and G. Alonso, "FPGA acceleration for the frequent item problem," in *ICDE: 26th International Conference on Data Engineering Conference*, 2010, pp. 669–680.

[18] P. S. Vaidya, J. J. Lee, F. Bowen, Y. Du, C. H. Nadungodage, and Y. Xia, "Symbiote: a reconfigurable logic assisted data streammanagement system (RLADSMS)," in *SIGMOD: Proceedings of the 2010 international conference on Management of data*. New York, NY, USA: ACM, 2010, pp. 1147–1150.

[19] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H.-A. Jacobsen, "Efficient event processing through reconfigurable hardware for algorithmic trading," in *VLDB: International Conference on Very Large Data Bases (VLDB)*, 2010.

[20] B. He, Q. Luo, and B. Choi, "Cache-conscious automata for xml filtering," *IEEE Trans. on Knowl. and Data Eng.*, vol. 18, no. 12, pp. 1629–1644, 2006.

[21] A. K. Gupta and D. Suciu, "Stream processing of xpath queries with predicates," in *SIGMOD: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2003, pp. 419–430.

[22] F. Peng and S. S. Chawathe, "Xpath queries on streaming data," in *SIGMOD: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2003, pp. 431–442.

[23] B. Ludäscher, P. Mukhopadhyay, and Y. Papakonstantinou, "A transducer-based xml query processor," in *VLDB: Proceedings of the 28th international conference on Very Large Data Bases*. VLDB Endowment, 2002, pp. 227–238.

[24] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient filtering of xml documents with xpath expressions," *The VLDB Journal*, vol. 11, no. 4, pp. 354–379, 2002.

[25] K. S. Candan, W.-P. Hsiung, S. Chen, J. Tatemura, and D. Agrawal, "Afilter: adaptable xml filtering with prefix-caching suffix-clustering," in *VLDB: Proceedings of the 32nd international conference on Very large data bases*. VLDB Endowment, 2006, pp. 559–570.

[26] G. Gou and R. Chirkova, "Efficient algorithms for evaluating xpath over streams," in *SIGMOD: of the 2007 ACM SIGMOD international conference on Management of data*. New York, NY, USA: ACM, 2007, pp. 269–280.

[27] Z. Dai, N. Ni, and J. Zhu, "A 1 cycle-per-byte XML parsing accelerator," in *FPGA: Proc. of the 18th Intl. Symposium on FPGAs*. New York, NY, USA: ACM, 2010, pp. 199–208.

[28] F. El-Hassan and D. Ionescu, "SCBXP: An efficient hardware-based XML parsing technique," in *SPL: 5th Southern Conference on Programmable Logic*. IEEE, April 2009, pp. 45–50.

[29] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson, "XML accelerator engine," in *1st Intl. Workshop on High Performance XML Processing*. Springer Berlin / Heidelberg, 2004.

[30] "SAX. Simple API for XML." [Online]. Available: http://sax.sourceforge.net

[31] "University of washington xml repository." [Online]. Available: http://www.cs.washington.edu/research/xmldatasets

[32] D. Barbosa, A. Mendelzon, J. Keenleyside, and K. Lyons, "ToXgene: a template-based data generator for XML," in *SIGMOD: Proc. of the ACM SIGMOD Intl. Conf. on Management of data*. New York, NY, USA: ACM, 2002, pp. 616–616.

[33] "XILINX DELIVERS 65nm VIRTEX-5 LX330." [Online]. Available: http://www.xilinx.com

[34] "Nvidia." [Online]. Available: http://www.nvidia.com

[35] "ATI." [Online]. Available: http://www.amd.com/us/products/Pages/products.aspx

[36] J. Bolz, I. Farmer, E. Grinspun, and P. Schröoder, "Sparse matrix solvers on the gpu: conjugate gradients and multigrid," in *SIGGRAPH: ACM SIGGRAPH 2003*. New York, NY, USA: ACM, 2003, pp. 917–924.

[37] T. Foley and J. Sugerman, "Kd-tree acceleration structures for a gpu raytracer," in *HWWS '05: Proceedings of the ACM SIGGRAPH/EUROGRAPHICS conference on Graphics hardware*. New York, NY, USA: ACM, 2005, pp. 15–22.

[38] C. Kim, J. Chhugani, N. Satish, E. Sedlar, A. Nguyen, T. Kaldewey, V. Lee, S. Brandt, and P. Dubey, "Fast: Fast architecture sensitive tree search on modern CPUs and GPUs," in *SIGMOD: Proceedings of the 2010 ACM SIGMOD international conference on Management of data*, 2010.

[39] N. Satish, C. Kim, J. Chhugani, A. Nguyen, V. Lee, D. Kim, and P. Dubey, "Fast in-memory sort on modern cpus and gpus: A case for bandwidth-oblivious simd sort," in *SIGMOD: Proceedings of the 2010 ACM SIGMOD international conference on Management of data*, Indianapolis, Indiana, USA, 2010.

[40] "Cuda." [Online]. Available: http://www.nvidia.com/object/cuda-home-new.html