A Study on Parallelizing XML Path Filtering Using Accelerators

ROGER MOUSSALLI, IBM T. J. Watson Research Center
MARIAM SALLOUM, ROBERT HALSTEAD, WALID NAJJAR, and
VASSILIS J. TSOTRAS, University of California Riverside

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-based systems. XML-based pub-sub systems provide users with more flexibility by allowing the formulation of complex queries on the content as well as the structure of the streaming messages. Messages that match a given user query are forwarded to the user. This article examines how to exploit the parallelism found in XPath filtering. Using an incoming XML stream, parsing and matching thousands of user profiles are performed simultaneously by matching engines. We show the benefits and trade-offs of mapping the proposed filtering approach onto FPGAs, processing streams of XML at wire speed, and GPUs, providing the flexibility of software. This is in contrast to conventional approaches bound by the sequential aspect of software computing, associated with a large memory footprint. By converting XPath expressions into custom stacks, our solution is the first to provide support for complex XPath structural constructs, such as parent-child and ancestor descendant relations, whilst allowing wildcarding and recursion. The measured speedups resulting from the GPU and FPGA accelerations versus single-core CPUs are up to 6.6X and 2.5 orders of magnitude, respectively. The FPGA approaches are up to 31X faster than software running on 12 CPU cores.

Categories and Subject Descriptors: H.2.4 [**Database Management**]: Systems—*Query processing*; B.5.1 [**Register-Transfer-Level Implementation**]: Design; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems

General Terms: Design, Performance, Experimentation

Additional Key Words and Phrases: Publish-subscribe systems, hardware accelerators, field-programmable gate arrays (FPGAs), graphics processing units (GPUs), XML

ACM Reference Format:
Roger Moussalli, Mariam Salloum, Robert Halstead, Walid Najjar, and Vassilis J. Tsotras. 2014. A study on parallelizing XML path filtering using accelerators. ACM Trans. Embedd. Comput. Syst. 13, 4, Article 93 (February 2014), 28 pages.
DOI: http://dx.doi.org/10.1145/2560040

1. INTRODUCTION

Increased demand for timely and accurate event-notification systems has led 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 documents into the system; *subscribers*, who post their

This work was partially funded by the National Science Foundation under CCR grants 0905509, 0811416, and IIS grants 0705916, 0803410, and 0910859.
Authors' addresses: R. Moussalli (corresponding author), R. Halstead, M. Salloum, W. Najjar, and V. J. Tsotras, Computer Science Department, Bournes College of Engineering, University of California Riverside; corresponding author's email: rmous@cs.ucr.edu.

DOI: http://dx.doi.org/10.1145/2560040

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 `alerts.google.com`, `news.google.com`, `pipes.yahoo.com`, and `www.ticket-master.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 or channels (e.g., news, weather, etc.) and would receive every document published on the channel. The second generation of pub-subs consists of predicate-based systems, where user profiles are described as conjunctions of (attribute, value) pairs, thus improving profile selection. The wide adoption of the *eXtensible Markup Language* (XML) as the standard format for data exchange, due to its self-describing and extensible nature, has led to the third generation, namely, XML-enabled pub-sub systems. Here, messages are encoded with XML, and profiles are expressed using XML query languages, such as XPath.[1] Such systems take advantage of the powerful querying that XML query languages offer: profiles can now describe requests not only on the document values but also on the structure of the messages.[2]

XML-based pub-sub systems have been adopted for the dissemination of *Micronews* feeds, which are short fragments of frequently updated information in XML-based formats, such as RSS. Feed readers, such as Bloglines and NewsGator, check the contents of micronews feeds periodically and display the returned results to the user.

The core of the pub-sub system is the *filtering* algorithm, which supports complex query matching of thousands of user profiles against a high volume of published messages. For each message received in the pub-sub system, the filtering algorithm determines the set of user profiles that have one or more matches in the message. Many software approaches have been presented to solve the XML filtering problem [Al-Khalifa et al. 2002; Diao et al. 2003; Green et al. 2004; Kwon et al. 2005]. These memory-bound solutions, however, suffer from the Von Neumann bottleneck and are unable to handle large volumes of input streams. On the other hand, field-programmable gate arrays (FPGAs) have been shown to be particularly suited for stream processing large amounts of data and do not suffer from the memory offloading problem faced by software implementations.

Graphical processing units (GPUs) are also a favorable option for applications requiring massively parallel computations [He et al. 2008; Ao et al. 2011; Kim et al. 2010; Lieberman et al. 2008]. GPUs serve as co-processors to the CPU such that sequential computations are run on the CPU while the computationally-intensive part is accelerated by the highly parallel GPU architecture. The architecture is favorable for single-instruction multiple data (SIMD) applications, where multiple threads are running on multiple cores executing the same program on different data.

The contributions of this work are as follows.

—We provide a novel dynamic programming-inspired approach for XML path filtering which does not result in false positives. Wildcard and recursion (nesting) support is offered using this solution.
—We present the first implementation and study of an FPGA-based solution to XML path filtering, using the aforementioned approach. In particular, we examined the trade-offs of two FPGA-based implementations:

[1]XML Path Language Version 1.0. `http://www.w3.org/TR/xpath`.
[2]In this manuscript, we use the terms "profile" and "query" interchangeably.

Table I. Summary of Impact of Several Factors on All the Studied Approaches

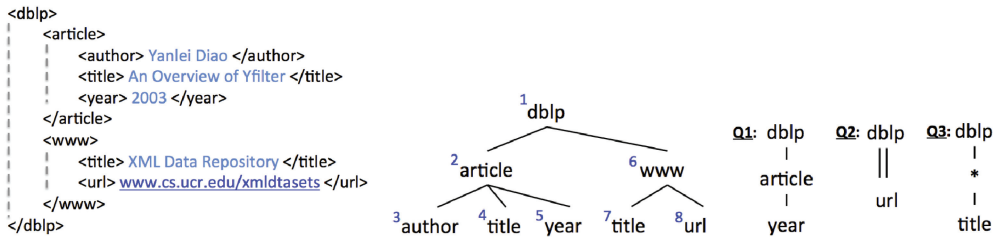
Factor	Software	GPU	Prog. FPGA	Cust. FPGA
Document Size	Decreases throughput rapidly and greatly affects the memory footprint.	Minimally increases throughput due to the reduced CPU-GPU transfers.	No effect on the filtering core.	No effect on the filtering core.
Query Length	Some impact on throughput (28%) and large impact on memory footprint.	Minimal effect on throughput, until over-utilization.	Linear impact on utilization of pre-mapped resources; no effect on throughput.	Small impact on area, minimal on throughput.
Number of Queries	Small impact on throughput.	No effect, until over-utilization / the common prefix opt. helps with scalability.	Linear impact on utilization of pre-mapped resources; no effect on throughput.	Linear effect on area, less on throughput until over-utilization.
Percentage of '' and '//'*	6% to 30% decrease in throughput per 10% added.	No effect.	No effect.	No effect.

(i) using fully customized query matching engines, thus resulting in low resource utilization.
(ii) using programmable query matching engines to allow (fast) dynamic query updates.

—We present the first implementation and study of a GPU-based solution to XML path filtering.
—We provide an extensive performance evaluation of the preceding approaches with leading software implementations. Our experimental focus is directed towards batches of small XML documents, which is the most common scenario in practical pub-sub applications.

This article is an extension of work presented in Moussalli et al. [2010, 2011a]. The additional material specific to this article includes the presentation of a unified solution to perform path matching queries on accelerators, which can apply to both FPGAs and GPUs. We also present the first implementation of a programmable FPGA-based accelerator for XML query matching, allowing on-the-fly updates. An in-depth evaluation considering mapping queries fully into GPU processing cores is offered. In addition, a complete new set of experiments is presented, focused on batches of small XML documents (a common consideration of pub-sub systems), rather than single large documents (which was the focus in Moussalli et al. [2010, 2011a]). Finally, we include an extensive end-to-end performance evaluation of the FPGA- and GPU-based approaches with leading software implementations. A complete comparison (see summary Table I) is offered, detailing the effect of several factors on CPU, GPU, custom-FPGA- and programmable-FPGA-based filtering.

The rest of the article is organized as follows. In Section 2, we define the requirements and challenges of the XML filtering problem. Section 3 presents related work. Section 4 provides an in-depth description of the proposed solution targeted for XML query filtering, while Section 5 describes the filtering architecture on FPGAs. Section 6 presents an experimental evaluation of the FPGA-based and GPU-based hardware approaches compared to the state-of-the-art software counterparts. Finally, conclusions appear in Section 7.



(a) XML document. (b) XML tree representation. (c) Path queries.

Fig. 1. Example XML Document in (a) textual and (b) tree representations. (c) Sample XML path queries are displayed, querying the XML document.

2. PROBLEM DEFINITION

XML filtering is the core problem in a pub-sub system. Formally, given a collection of user profiles and a stream of XML documents, the objective of the filtering algorithm is to determine, for each document D, the set of profiles that have at least one match in D.

An XML document has a hierarchical (tree) structure that consists of markup and content. Markups, also referred to as tags, begin with the character '<' and end with '>'. Nodes in an XML document begin with a *start-tag* (e.g., <author>) and end with a corresponding *end-tag* (e.g., </author>). Figure 1(a) shows a small XML document example, while Figure 1(b) shows the XML document's tree representation. In this article, we shall use the terms 'tag' and 'node' interchangeably. For simplicity, Figure 1(b) shows the tags/nodes (i.e., the structural relationship between nodes) in the XML document of Figure 1(a), but not the content (values). The values can be thought as special leaf nodes in the tree (not shown).

XPath[1] is a popular language for querying and selecting parts of an XML document. In this article, we address a core fragment of XPath that includes node names, wildcards, and the /child:: and /descendant-or-self:: axis. The grammar of the supported query language is given next.

$$
\begin{aligned}
\mathit{Path} &:= \mathit{Step} \mid \mathit{Path}\ \ \mathit{Step} \\
\mathit{Step} &:= \mathit{Axis}\ \mathit{Node\ Test} \\
\mathit{Axis} &:= \text{`/'} \mid \text{`//'} \\
\mathit{Node\ Test} &:= \mathit{name} \mid \text{`}*\text{'}
\end{aligned}
$$

The query consists of a sequence of location steps, where each location step consists of a node test and an axis. The node test is either a node name or a wildcard '*' (wildcards can match any node name). The axis is a binary operator that specifies the hierarchical relationship between two nodes. We support two common axes: the parent/child axis (denoted by '/'), and the ancestor/descendant axis (denoted by '//').

Example path queries are shown in Figure 1(c). Consider Q_1 (/dblp/article/year), which is a path query of depth three and specifies a structure which consists of nodes 'dblp', 'article', and 'year', where each node is separated by a '/' operator. This query is satisfied by nodes (dblp, 1), (article,2), and (year, 5) in the XML tree shown in Figure 1(b). Q_2 (/dblp//url) is a path query of depth two and specifies a structure which consists of two nodes: 'dblp' and 'url' are separated by the '//' operator. Q_2 specifies that the node 'url' must be descendant of the 'dblp' node. The nodes (dblp,1) and (url,8) in Figure 1(b) satisfy this query structure. Q_3 (/dblp/*/title) specifies a structure that consists of two nodes and a wildcard. The nodes (dblp,1), (article,2), and (title,4) satisfy one match, while nodes (dblp,1), (www,6), and (title,7) satisfy another match for Q_3.

In a pub-sub system, XML documents are received in a streaming fashion, and they are parsed by a SAX parser,[3] the latter generating startElement(*name*) and endElement(*name*) events. Designing an XML pub-sub system raises many technical challenges due to the high volume of XML messages and the complexity and size of user profiles.

3. RELATED WORK

As traditional platforms are increasingly hitting limitations when processing high volumes of streaming data, researchers are investigating FPGAs and GPUs for database applications. The performance advantages of such platforms arise from the ability to execute thousands of computations in parallel, relieving the application from the sequential limitations of software execution on Von-neumann-based platforms.

The Glacier component library is presented in Mueller et al. [2009], which includes logic circuits of common operators, such as selection, aggregation, and grouping, for stream processing. The use of FPGAs in a distributed network system for traffic control information processing is demonstrated in Vaidya et al. [2010]. Predicate-based filtering on FPGAs was investigated by Sadoghi et al. [2010], where user profiles are expressed as a 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.

GPUs have evolved to the point where many real-world applications are easily implemented and run significantly faster than on multicore systems; thus, a large number of recent work has investigated GPUs for the acceleration of database applications [He et al. 2008; Ao et al. 2011; Kim et al. 2010]. He et al. [2008] utilized GPUs to accelerate relational joins, while Kim et al. [2010] present a CPU-GPU architecture to accelerate tree-search, which was shown to have low latency and to support online bulk updates to the tree. Recently, Ao et al. [2011] proposed the utilization of GPUs to speed up indexing by offloading list intersection and index compression operations to the GPU.

3.1. Software Approaches to XML Filtering

The popularity of XML has triggered research efforts in building 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 user profiles [Altinel and Franklin 2000; Diao et al. 2003; Green et al. 2004; He et al. 2006; Moro et al. 2007]. An early work, XFilter [Altinel and Franklin 2000], proposed building an FSM for each profile such that each node in the XPath expression becomes a state in the FSM. The FSM transitions are executed as XML tag events are generated. The profile is as a match when the final state of its FSM is reached. YFilter [Diao et al. 2003] built upon the work of XFilter and proposed a nondeterministic finite automata (NFA) representation of user profiles (i.e., path expressions) which combines all profiles into a single machine, thus reducing the number of states needed to represent the set of user profiles. Whereas YFilter exploits prefix commonalities, the BUFF system builds the FSM in a bottom-up fashion to take advantage of suffix commonalities in profiles [Moro et al. 2007]. Several other FSM-based approaches were introduced that use different types of state machines [Green et al. 2004; Gupta and Suciu 2003; Peng and Chawathe 2003; Ludäscher et al. 2002].

Sequence-based approaches (e.g., [Kwon et al. 2005; Salloum and Tsotras 2009]) transform the XML document and user profiles into sequences and employ subsequence

[3]Simple API for XML. `http://sax.sourceforge.net`.

matching to determine which profiles have a match in the XML sequence. FiST [Kwon et al. 2005] was the first to propose a sequence-based XML filtering system which transforms the query profiles and XML streams into Prufer sequences, then employs subsequence matching to determine if the query has a match in the XML stream.

Several other approaches have been proposed [Chan et al. 2002; Candan et al. 2006; Gou and Chirkova 2007]. XTrie [Chan et al. 2002] uses a trie-based data structure to index common substrings of XPath profiles, but it only supports the /child:: axis. AFilter [Candan et al. 2006] exploits both prefix and suffix commonalities in the set of XPath profiles. More recently, Gou and Chirkova [2007] have proposed two stack-based stream-querying (and filtering) algorithms, LQ and EQ, which are based on lazy strategy and eager strategy, respectively.

3.2. Hardware-Accelerated Approaches to XML Processing

Previous works [Dai et al. 2010; El-Hassan and Ionescu 2009; Lunteren et al. 2004] 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 El-Hassan and Ionescu [2009]. 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 filtering.

Lunteren et al. [2004] 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 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 is efficient 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 [Green et al. 2004]. However, FPGAs are limited by area and DFAs may suffer from state explosion, thus NFAs are a better approach when considering FPGAs.

Our previous work [Mitra et al. 2009] was the first to propose a pure-hardware solution to the XML filtering problem. We adopted an NFA approach to XML filtering by representing queries as regular expressions, and improvements of over one order of magnitude were reported when compared to software. However, that method is unable to handle recursion (nesting) 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 article.

We presented [Moussalli et al. 2011a] the first implementation of the XML filtering problem onto GPUs. By extending the approach described in Moussalli et al. [2010], we made use of the programmability aspect of GPUs to tackle the issue of static queries. We also studied common prefix optimizations to the query set, with the goal of speeding up execution, by minimizing computation. Speedup was reported over state-of-the-art CPU approaches. In this work, we provide a more thorough evaluation of the implementation of an GPU-based XML query matching engine. We also provide a performance comparison with FPGA-based approaches.

In Moussalli et al. [2011b], we considered complex twig matching on FPGAs (i.e., the profiles can be complex trees). Instead, we concentrate here on path profiles due to

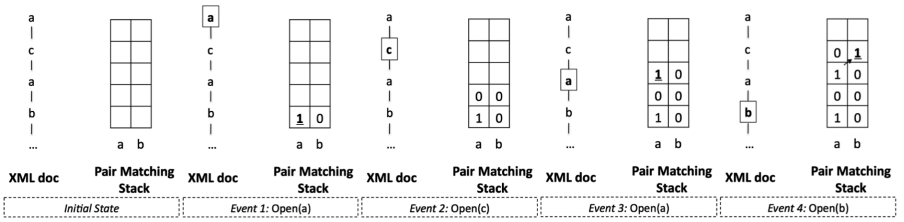


Fig. 2. Overview of the matching of pair $\{a/b\}$. Each step refers to an *open(tag)* or *close(tag)* event, relative to the highlighted tag. A '1' in the b column indicates a match.

their less complex nature and inherent parallelism that can be exploited by the FPGA and GPU. We also show how to support path profiles with on-the-fly updates using FPGAs.

4. PARALLEL XPATH FILTERING SOLUTION

We now introduce a solution that identifies the parallelism inherent in path matching and thus applies to both FPGA and GPU approaches.

A user query expressed in the XPath language is comprised of J nodes and J-1 relations, where each pair of nodes shares a relationship: parent/child or ancestor/descendant. A path query of length J is said to have matched if a sequence of nodes in the XML document sharing the same relations as the tags in the query has occurred; this is only true if the subpath of length J-1 has moreover matched. Next, we present a stack-based generic XPath filtering algorithm which will be used for our parallel implementations. We first focus on the matching of the base case (i.e., paths of length 2, or simply pairs), and then extend it to general paths.

4.1. Pairs Matching

Using an XML stream as input, we look at the matching of pairs' relationships.

4.1.1. Parent/Child Relationships. Stacks are an essential feature of XML filtering systems, where the respective states of all open (non-closed) nodes in the XML tree are saved. Using the presented solution, an *open(tag)* is translated into a *push* event, and conversely, a *close(tag)* is equivalent to a *pop* event. Matching for $\{a/b\}$ now requires a stack as deep as the maximal depth of the XML document and as wide as 2: one column for each a and b. This is a binary stack that can be filled with 1's and 0's based on the match state, as explained next, where a '1' indicates a match.

Each query node is allocated a match state for every tree level (node in the tree). As such, nesting (recursion) in the XML document is supported, where the level of each tag in the tree differentiates it from other similar tags. Furthermore, two tags cannot simultaneously coexist at the same tree level (one has to be popped before pushing the other).

Through the streaming of the XML document, for every *open(a)* event, a '1' is pushed on the first column (the a column), indicating that a has been opened at that level. On the other hand, every time an *open(b)* event occurs, if the first column contains a '1' on the previous top of the stack, only then can a '1' be pushed onto the second column (diagonally upwards propagating '1'), indicating that b as a child of a has been found. Checking for levels is implied, since neighboring rows share a parent/child relation by design. Note that on each push event all columns are simultaneously updated at the top of the stack.

Figure 2 shows the event-by-event matching of the pair $\{a/b\}$ in a sample XML document. The XML document to be streamed is drawn on the left-hand side, whereas a stack of width 2 is shown to the right. Each column is labeled with the corresponding

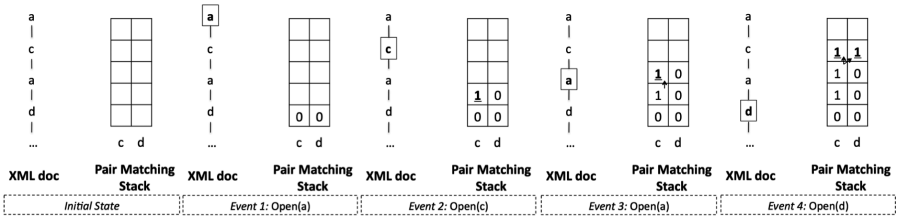


Fig. 3. Overview of the matching of pair $\{c//d\}$. Each step refers to an *open(tag)* or *close(tag)* event, relative to the highlighted tag. A '1' in the d column indicates a match.

tag of the $\{a/b\}$ pair. Note that support for recursion is depicted under event 3, where each occurrence of the a tag in the XML document has a corresponding state (row) in the stack.

4.1.2. Ancestor/Descendant Relationships. Using the stack-based approach, every ancestor/descendant pair is also mapped to a two-column stack as deep as the XML document. When matching for $\{c//d\}$, a '1' is pushed on the first column (the c column) at every *open(c)* event. However, d does not require c to be its parent, rather its ancestor; therefore, as long as c has not been closed (i.e., for all its descendants), a '1' is pushed alongside each pushed descendant. Hence, any descendant *open(d)* event should result in a '1' being written to the second column. The tree node c is popped (upon a *close(c)* event) after all its descendants are popped (i.e., each respective stack row of each descendant), and with it any record of c being pushed.

To highlight this property, a '1' is allowed to propagate vertically upwards in the column of the ancestor (here, the first column). It is also true that the top of the stack at both columns can be updated simultaneously. Figure 3 shows the event-by-event matching of the pair $\{c//d\}$ in a sample XML document.

4.2. Custom Stacks for Path Matching

We can now move to general paths by considering their pairs. For instance, the path $\{a/b//c/d\}$ can be broken down into pairs $\{a/b\}$, $\{b//c\}$, and $\{c/d\}$. The mechanisms described in Section 4.1 hold for all pairs, where, based on the relation, a '1' is allowed to propagate vertically or diagonally upwards (or both). Matching a path of length J requires a stack of width J columns (one for each node): all pair stacks are merged at the common node's columns. A '1' in the j^{th} column ($j \leq J$) indicates that the path of length j was found in the XML document. Thus, for a successful match to occur, a '1' has to propagate from the path's root (1^{st} column) to the leaf (J^{th} column).

The matching approach can be thought of as dynamic programming, where the stacks are binary stacks, and a '1' in the query leaf (j^{th}) column indicates a match. The recurrence equation encompasses both checks described for parent/child and ancestor/descendant relations, as needed per query node.

Figure 4 shows an event-by-event overview of all the steps required for the matching of the XPath $\{a/c/a/c/b\}$.

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

Support for recursion is depicted under the third event, where both the first and third columns indicate a match for tag '*a*' simultaneously, thus, allowing two possible

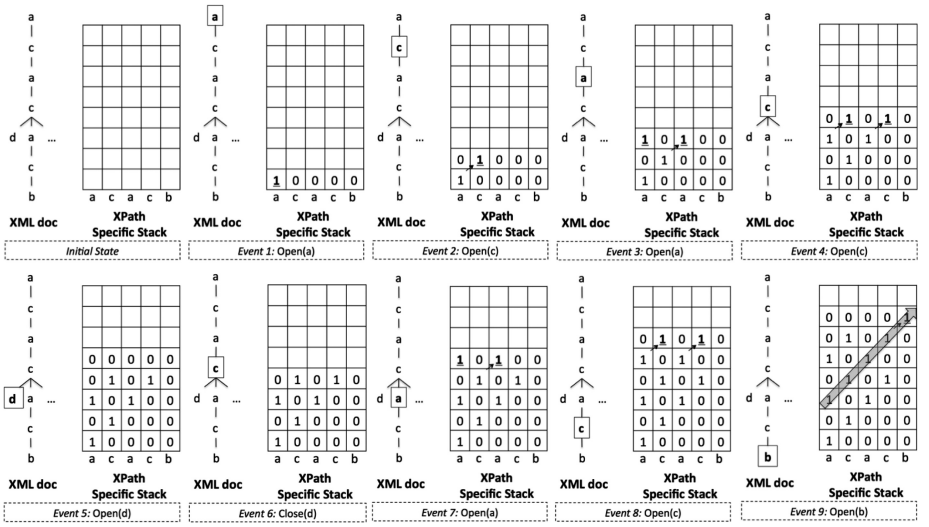


Fig. 4. Overview of the matching of XPath $\{a/c/a/c/b\}$. Each step refers to an *open(tag)* or *close(tag)* event, relative to the highlighted tag. A '1' in the right-most column indicates a match.

matches of the same XPath to be in progress concurrently: one having started at event 1, the other at event 3.

With an *open(c)* as the fourth event, 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 at the fifth event, no partial matches are propagated. Moreover, roll-back to the previous state took place with the *close(d)* event taking place, thus popping the top of stack.

A third partial possible match spawns off on at event 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 event 8. Ultimately, a match has been found in event 9, thanks to the partial matching starting propagation from event 3. A match can be seen as a diagonal of 1's, ending in the fifth column.

4.3. Matching Stack Properties

We refer to our stacks as path-specific stacks (PSS), where every path is mapped to a stack whose width is defined by the path length, and conditions to write to every column are determined by the path nodes and the relations connecting them. Here are some properties of the PSS.

—A PSS is written to push events only.
—Pop events only affect the pointer to the top of the stack.
—A '1' can propagate diagonally upwards from and to any two adjacent columns connecting a parent or ancestor to a child or descendant, respectively.
—If the node mapped to a column is an ancestor, then a '1' can propagate vertically upwards; this helps indicate matches to all descendants.

4.4. Inherent Parallelism

Since an XML-enabled pub-sub system involves multiple profiles processed over the same document data stream, it is possible to utilize parallel architectures for

accelerating its filtering performance. Using our proposed stack-based approach, two levels of parallelism can be pursued here.

(1) *Inter-query parallelism*. All queries (stacks) can be processed in a parallel fashion, even when stack columns are shared among queries (e.g., when applying the common prefix optimization). This parallelism is available due to the embarrassingly parallel nature of the filtering problem.
(2) *Intra-query parallelism*. Updating the state of all nodes within a query (top of stack at every column) can be achieved in parallel.

Each user profile can be implemented on the FPGA unit as a hardware datapath circuit, and with appropriate optimizations, it is possible to fit up to tens of thousands of queries on a single FPGA chip. Moreover, having the parallel processing modules implemented on the same chip eliminates the need for expensive communications between them. This in turn allows for full pipelining of the parsing and filtering processes: as an event is produced by the parser, it is immediately forwarded to the filtering module, implemented on the same FPGA chip (*added level of parallelism*). Section 5 elaborates on the details of a full-hardware XPath filtering engine using FPGAs.

Similarly, GPUs are suitable for general-purpose applications where thousands of simple computing cores perform one common operation (at a time). We look into mapping query stacks and columns to each of those computing cores to process XML documents and perform filtering at a high throughput.

When mapped to FPGAs, the proposed approach has virtually no memory footprint: as the XML document is streamed, filtering is performed in the FPGA at wire speed without relying on external memory. Similarly for GPUs, memory offloading is minimal, with stacks localized to low-latency shared memories, whereas pure CPU approaches build data structures up to two orders of magnitude larger than the XML document streamed. It is typical for large data structures to result from software techniques due to intermediate state saving. While two orders of magnitude is not characteristic, it has been reached.

4.5. Support for Predicate Expression Evaluation

The preceding discussion focused on identifying whether a profile structure appears within a document. Nevertheless, user profiles can specify not only the XML structure, but may also content predicate expressions. The XPath query language allows the specification of predicates to filter the node set with respect to the current axis.

Predicate expressions perform comparisons using $<$, $>$, $\geq$, $\leq$, $=$, and $!=$ operations, and these expressions can be combined by *and* and *or*. Though the XPath language provides support for even more complex functions, such as *mod* for evaluating numbers, generally simple predicate comparisons are most common for XML filtering.

While structure evaluation is more challenging and requires the use of stacks, etc., predicate evaluation comprises of content identification and can thus be migrated to the parser. Predicates can then be treated as additional tag identifiers in intercolumn relations. Conditions to propagate a '1' across two columns will be slightly modified to incorporate predicates; thus, in the parser, predicate output must also evaluate to '1'. In addition, by migrating predicate evaluation to the parser, we can take advantage of the commonalities in predicates across queries.

Thanks to the massive parallelism of FPGAs, all predicates can be evaluated in parallel. In the case where one or more predicates require more than a cycle to be computed, we envision that predicate evaluation can be performed in a pipelined manner; hence, there would be no impact on throughput. Further, since XML events are less frequent than the parsing rate, the time window allocated to compute predicates could be large enough not to incur any extra pipelining.

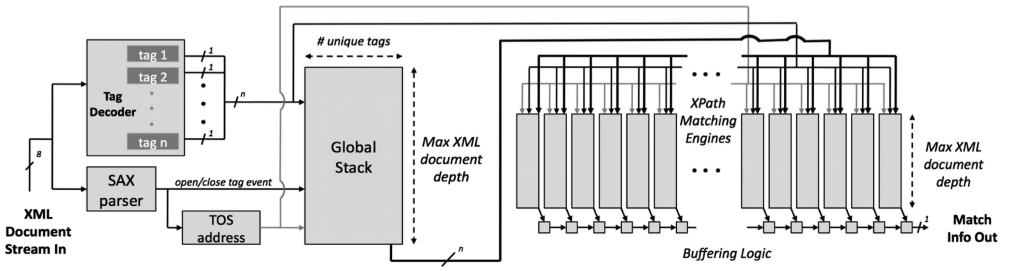


Fig. 5. High-level FPGA-based system overview.

The discussion on predicate support here helps as proof of concept. There are several types of predicates, and different efficient evaluation engines can be tailored to each type. This opens several research questions. Moving predicate evaluation to the parsers allows it to be abstracted from querying the structure of XML documents. We leave further related discussion and in-depth study on predicate evaluation to future work.

Our work targets mainly the (more complex) filtering on the structure of the XML profiles, rather than content; thus parsers are orthogonal to our filtering system. High performance and optimized parsers can be deployed in our system with minimal modification required.

5. XPATH FILTERING ON FPGAS

Using an XML stream as input, we present a full-hardware XPath filtering system on FPGAs; this section describes the details of the proposed approach. Two implementations of the stack algorithm described in Section 4 are explored; the first targeting setups where the query lifetime is considerably longer than that of the streamed XML document (Section 5.2); the second implementation targets queries that are updated regularly (Section 5.3). In the first approach, the *soft* circuit is fully customized and thus more profiles are "packed", but to update profiles, one has to regenerate the circuit description and go through the lengthy synthesis/place and route process. Instead, the focus of the second approach is on supporting dynamic profile updating through a generic circuit where each profile is configured, at the cost of fitting fewer profiles on the FPGA.

5.1. System Architecture

Our hardware filtering architecture assumes an XML document stream as input. As the document is streamed, it is being parsed on the fly, and *open(tag)* and *close(tag)* events are generated and passed to the query matching engines (i.e., path-specific stacks). Using these, all query matching engines are updating states to find occurrences of paths within the streamed document. As a result, matching ends when the XML stream is complete, and all match states can then be reported. Figure 5 illustrates a high-level view of the system architecture.

Parsing is achieved using a lightweight hardware implementation of the *Simple API for XML* (SAX) Parser.[3] The SAX parser is an event-driven XML parser, ideal for streaming applications. Unlike other parsers (such as DOM[4]), where the entire XML document needs to be stored in memory before processing can start, SAX Parsers would generate *open(tag)* and *close(tag)* events on the fly.

The deployed FPGA lightweight parser operates at a rate of one byte of XML per cycle. As (*open()*, *close()*) events are less frequent than bytes, the parser will not

[4]w3.org/DOM. `http://www.w3.org/DOM`.

produce one event per cycle. Efficient hardware parsers based on the implementation in El-Hassan and Ionescu [2009] parse at a rate of up to 4 bytes per cycle, with an average of 2 bytes per cycle.

The SAX parser makes use of a tag decoder, resulting in considerable savings in FPGA resources, since tags are shared across several queries. The tag decoder is implemented as a content addressable memory (CAM) which, when given a tag, searches through all entries in parallel and requires a single cycle to generate the decoded tag address.

Figure 5 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 event, and all lines are cleared otherwise.

Since all stacks on chip would be updating concurrently, the top of stack address (common to all stacks) is centralized, being generated from a common structure, which in turn requires push (*open*) and pop (*close*) notifications from the SAX parser. This is depicted in Figure 5, where the top of stack (TOS) address is routed to a structure referred to as the global stack and to all remaining path-specific stacks.

The decoded tag ID output of the tag decoder is pushed onto the global stack upon *open()* events. Moreover, the global stack uses the common top of stack address structure and passes its output to all the matching engines. The global stack is added to keep track of the XML node at one level lower and is only used in the matching engines described in Section 5.2.2. The global stack is mapped to on-chip block RAMs (BRAMs)[5]—highly configurable hard-wired memory blocks that are embedded in most Xilinx FPGAs.

Finally, with up to tens of thousands of matching engines coexisting 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 [Mitra et al. 2009] 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. XPath profiles $\{a//b\}$ and $\{c/a/d/b\}$ are such examples.

For the application of interest (filtering), the number of matches of each profile is of no relevance, rather whether or not there was at least one match. Thus, the matching logic is enhanced with one-bit buffers relative to each PSS (*Buffering Logic*, Figure 5); 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. This overhead is typically minimal when compared to the size of the documents streamed through the FPGA. In order to reduce this overhead, reporting results back can be parallelized with the streaming in of a new XML document. This is achieved through the buffering of the final match state of each query.

5.2. Fully Customized FPGA Hardware

In this section, we describe the low-level implementation details of the path-specific stacks (PSS), that is, the matching engines as described in Section 4.

5.2.1. Matching XPaths Using Path-Specific Stacks. Stacks are implemented using distributed memory blocks, that is, memory structures on Xilinx FPGAs that comprise of slice LUTs. The stack width (number of stack columns) is equal to the length of the XPath mapped to it, whereas the stack depth is the maximum streamed XML document depth. The latter is determined offline at compile time.

[5]Block RAM v1.00a. `http://www.xilinx.com`.

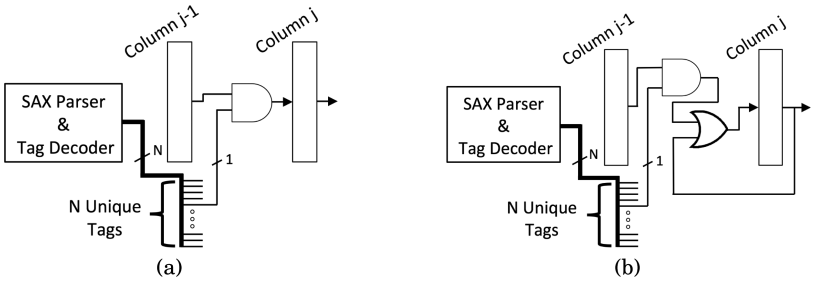


Fig. 6. Hardware logic connecting two PSS columns with the j^{th} and $(j+1)^{\text{th}}$ column sharing (a) a parent/child relation, and (b) an ancestor/descendant relation. If the child/descendant node is a wildcard, the *AND* gate and a tag bit are not needed.

Based on the relation of every two nodes in the path mapped to the PSS, the input to every column is determined as depicted in Figures 6(a) and 6(b). In case of a parent/child relation (Figure 6(a)), a '1' is pushed to the j^{th} column:

—on the *open()* event of the tag mapped to the j^{th} column (as determined by the parser and decoder)
—only if a '1' is stored at the top of stack of the $(j-1)^{\text{th}}$ column.

On the other hand, in case of an ancestor/descendant relation (Figure 6(b), where the j^{th} node is an ancestor), the same conditions as a parent/child relation hold, with the addition of *OR*-ing the output of the j^{th} column to the output of the *AND* gate, which would force pushing a '1' once it was written, thus preserving the property of the ancestor.

If the child/descendant node is a wildcard (e.g., {.../A/*/...}, {.../A//*/...}), any tag would result in the propagation of the match from the $(j-1)^{\text{th}}$ column. Thus, there would be no need for a comparison with any decoder bit, resulting in the omission of the AND gates shown in Figures 6(a) and 6(b). In the case of a parent/child relation with a wildcard as child (Figure 6(a)), the output of the $(j-1)^{\text{th}}$ column is connected to the input of the j^{th} column, with no extra logic in between. In the case of an ancestor/descendant relation with a wildcard as descendant (Figure 6(b)), the output of the $(j-1)^{\text{th}}$ column is connected to the *OR* gate preceding the j^{th} column.

5.2.2. Applied Optimizations for PSS-Reduced Resource Utilization. As described in Section 5.2.1, the width of every PSS is equivalent to the depth of the XPath profile mapped to it. In this section, three optimizations are proposed with the goal of minimizing the number of required stack columns, hence utilized FPGA resources. We focus on optimizing the PSS mapping of the same XPath profile used as a base example in Figure 4.

The first optimization relates to removing the column respective to the last query node. This is a simple optimization. When the last node is evaluated to match, the match bit is instead stored in some buffering logic. There is no need to keep track of the match state of the last node at every document level, since no other nodes depend on it.

The remaining two optimizations make use of the global stack, a structure shared by all matching engines (hence global), first introduced in Section 5.1 and Figure 5. At every *open()* XML event, the decoded representation of the respective opened tag is pushed onto the Global Stack. Conversely, every *close()* XML event results in popping from the global stack. The top of the global stack output (TOS) is made to reflect the parent tag of the currently active tag.

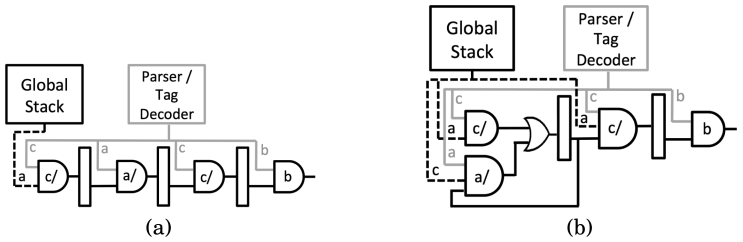


Fig. 7. Hardware logic depicting the implementation of the filtering engine respective to query {a/c/a/c/b}, (a) without and (b) with the third stack optimization.

The second optimization relates to removing the column respective to the query root node. The top of the global stack reflects the decoded representation of the active XML tree tag. It hence also represents the match state of all query root nodes through a respective TOS bit. Evaluating the match state of the second node in a query is achieved by reading the TOS bit of the root node (implicit root stack column) and the current decoded tag. Figure 7(a) depicts a query implementation based on this optimization. Further explanation is provided next.

The third optimization helps reduce the number of PSS columns to the maximum number of node matches that can be set per XML event. A given node in a query can become in a matched state if the previous node is in a matched state and if the current active XML document node's tag matches that of the node in question (as seen in Figure 6(a)). In other words, at a given node in the XML document, the only query nodes that could result in being matched are the nodes whose tag is identical to the XML node's tag.

For example, assuming queries Q1{A/B/C} and Q2{D/C/E}, upon an *open(C)* XML document event, only node 3 of Q1 and node 2 of Q2 could result in being in a matched state (the nodes with tag C).

This implies that some column entries are not utilized, and this can be deduced from the decoded XML tag at a given level, alongside the tag of the node mapped to this given column. For instance, an entry in a 'C' column can be only set to '1' (matched) at a level where the XML document has a 'C' tag. The latter can be deduced from the global stack.

Therefore, query nodes are nonconflicting if they cannot be in a matched state at the same XML level. Nonconflicting nodes can share a stack column.

Wildcard and ancestor/descendant nodes conflict with any other node, since they can be in a matched state at any given XML node. Therefore, wildcard and ancestor/descendant nodes can under no conditions share stack columns.

When building the PSS with the third optimization on, the added rule is to map every node to the first column to which no conflicting nodes are mapped. Tested columns for mapping start from the root of the query up to the node in question. If no such column is found, a new column is instantiated.

We show in Figures 7(a) and 7(b) the hardware logic depicting the implementation of the PSS's respective to query {a/c/a/c/d}, without and with the third optimization on; the first two optimizations are applied to both implementations. Note that the event-by-event detailed matching steps of this query were previously presented in Figure 4.

Looking at Figure 7(a), stmatching for the first query node {a/} is achieved by using the global stack, as described in the second optimization earlier. The advantage of using a global stack is relevant with multiple query engines on the FPGA, rather than just one. The last query node does not require a stack, as described by the first stack optimization earlier. All other query nodes require a 2-input *AND* gate alongside a

stack column. When reading from the global stack and tag decoder, a single bit of each tag is forwarded to the *AND* gate based on the respective tag. Every *AND* gate in Figure 7(a) reflects the node implemented through it.

On the other hand, when making use of the third stack optimization, the second and third nodes ({c/} and {a/}) can share a stack column, since they are nonconflicting. The fourth node {c/} conflicts with the second, since they share the same tag. Thus, two *AND* gates are connected at the input of the first column, with their outputs merged (*OR*-ed).

The *AND* gate respective to node {a/} reads the output of the first column (though it writes to it), while also reading the global stack *tag(c)* output bit; the latter will ensure whether a match indicated at the output of the first column was resulting from a previous {c} as a parent of the current XML document node.

Similarly, since the first column stores information respective to more than one node, the *AND* gate reading from that first column requires a global stack bit to filter out the matches resulting from the previous {c/} node.

Finally, since the second column stores the match state of only one node, the *AND* gate reading from it does not make use of the global stack.

Savings in Resources. Every stack column is implemented through an FPGA LUT (look-up table); the number of LUTs needed to implement the logic between columns is dependent on the number of unique input bits to this logic versus the physical number of LUT input bits. For instance, a 6-input boolean function can be implemented using one 6-input LUT or two 5-input LUTs. The LUT size is a physical constraint of the FPGA used. Typically, modern FPGAs make use of 5-input LUTs. Assuming such LUTs, the PSS implementation in Figure 7(a) requires seven LUTs (four for logic, three for stack columns), while the implementation in Figure 7(b) requires five LUTs (three for logic, two for stack columns).[6]

5.3. Programmable FPGA Hardware for Fast Update Time

In this section, we introduce an FPGA-based approach for XML filtering, targeting applications requiring frequent query updates. In the approach presented in Section 5.2, the profiles are identified prior to synthesis, and every hardware PSS is connected to exactly the signals needed for filtering. Updating queries would require an updated hardware description. Going from hardware description to FPGA configuration includes synthesis/place and route—processes that can take up to several hours depending on the resulting circuit size. Here, the latter is mostly bounded by the total number of query profiles. Hence, using a fully-customized accelerator works well when targeting applications where the lifetime of the query is much longer than that of the document.

5.3.1. Programmable Path-Specific Stacks. For applications where user profiles are updated regularly, we present a generic customizable PSS whose functionality is similar to that of a custom non-optimized PSS. So far, select wire signals are routed to each stack column from the tag decoder and global stack. Here, focus is shifted to allow a stack column to match for any tag followed by any relation. Every column will be programmed to support matching for one tag and relation per configuration.

The optimization of mapping several distinct tags to one column, as described in Section 5.2.2, is not applied to the programmable path-specific stacks. Instead, exactly one query node is mapped to a respective column. A '1' propagates diagonally only between two adjacent columns.

[6]These results were generated through Synplify Pro 2010-09 (synthesis) and Xilinx ISE 14 (PAR). Although LUT sharing did not take effect here, column optimizations would still result in area savings when LUT sharing is applied on XPath queries.

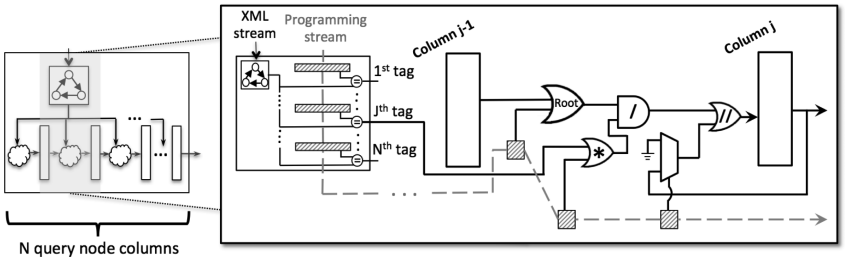


Fig. 8. Programmable FPGA hardware overview, with emphasis on the column connections. The XML input passes through the parser and programmable tag decoder. Every decoded tag bit is connected to the logic preceding one column; every query node is mapped to a column. The connections between two columns can be programmed by writing to the (striped) flip-flops. A node can be a root node in the query (see **Root**), can be a wildcards (see '*'), or can be followed by a parent/child relation or ancestor/descendant (see '*//*'). The use of any of these is optional and node-dependent.

The programmable FPGA logic consists of a set of stack columns connected serially (see Figure 8). In between each two columns lies the programmable logic implementing a single query node, enabling the propagation of '1's.

The XML input stream passes through the parser and tag decoder. The tag decoder is now made programmable such that every tag-decoded tag bit is connected one query node. Hence, tag decoder contents could contain duplicates. The number of tags in the decoder is equal to the number of available hardware columns.

Figure 8 shows the configurability of the connecting logic between two columns and the support for the following.

—*Any Tag.* The tag required by a query node is stored in the programmable decoder and forwarded to this column logic only.
—*Roots.* A query node can be a root by logically disconnecting it from the previous column. A '1' stored in the leftmost flip-flop would overwrite any output of the previous column (see *OR* gate with **Root**).
—*Wildcards.* In case of a wildcard, a '1' is stored in a flip-flop and *OR*-ed with the respective tag decoder bit (see *OR* gate with '*'). The OR gate has no effect in case of a '0' stored in the input flip-flop and would otherwise nullify the effect of the multiplexer output.
—*Parent/Child Relations.* As in the custom PSS, an *AND* gate is required to ensure that a '1' is stored in the top of stack of the previous column and that the required tag has been opened (see *AND* gate with '*/*').
—*Ancestor/Descendant Relations.* Similar to a PSS, if a node mapped to a column is an ancestor, then the input to the column is *OR*-ed with its top of stack output. Support for ancestor/descendant relations is provided by using an *OR* gate (labeled with '*//*') that takes as input the output of a multiplexer. The select bit (stored in a flip-flop) is used to forward to the *OR* gate either the output of that column (feedback signal) or ground (a '0'), the latter having no effect on the output of the *AND* gate.

Every column has a 'match' bit-buffer indicating whether or not a match occurred at its query node. Once streaming of the XML document is completed, all the column match bits are read as results. The match state of the leaf nodes would be of interest.

All configuration flip-flops are shown as striped, and all flip-flops across all tags in the decoder and all column configuration flip-flops are connected as a single shift register; generic stacks are programmed in a serial fashion. A query is now represented as a sequence of bits that control the hardware. The FPGA logic needs to be

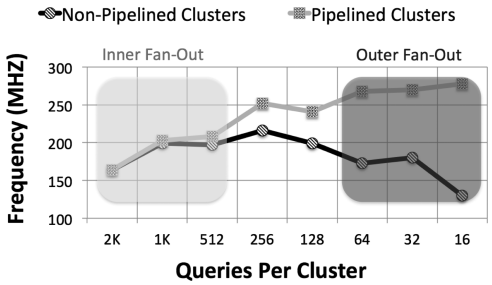


Fig. 9. Design space exploration on 2K queries mapped to customized stacks on a Xilinx V6LX240T FPGA: as the cluster size is varied, the effect on performance is studied. The low frequency in the inner fan-out region is due to the many queries per cluster. Pipelining the input stream across clusters will fight over-clustering (effect visible through a maintained high frequency in the outer fan-out region).

synthesized/placed and routed only once initially, and all query updates are applied by streaming the bits that represent the queries, one bit at a time.

We provide [Moussalli et al. 2011b] a description of custom FPGA hardware for matching queries expressed as twigs. These are more complex queries that require two types of stacks: push stacks which are updated on push events as described earlier, and pop stacks which update mostly on pop events. Twig queries are broken up as several split node to split node paths; each path requiring one push stack and one pop stack for filtering. As a first natural exploration step, we focus on adapting path queries to programmable hardware and GPUs; as part of our future work and based on the work detailed in this article, we will be investigating a similar, yet more complex, framework targeting twig queries. With twigs, the stacks respective to the smaller broken-down paths should be connected. These connections between several stacks should become programmable, which adds another level of complexity to the resulting architecture.

5.4. Performance Optimizations

A limiting factor in FPGA performance is the frequency at which the circuit will run on-chip. This operational frequency is bound by the length of the longest wire (i.e., the critical path).

With respect to our design, the tag decoder and global stack outputs are forwarded tens of thousands of matching engines. This creates a fan-out problem, where a single wire is used all over the FPGA chip. In order to minimize the effect of fan-out, we resort to clustering by replicating the parser, tag decoder, and global stack, and distributing queries across clusters.

Figure 9 depicts a design space exploration to determine the adequate cluster size in order to achieve a good balance between fan-out within clusters (i.e., too many queries per cluster), and overclustering (i.e., too few queries per cluster). Results are generated using Synplify Pro 2010-09 (synthesis) and Xilinx ISE 14 (PAR) targeting a Xilinx V6LX240T FPGA. While setting the total number of user profiles to 2K, the cluster size was varied from 8 to 2K in steps of doubling the cluster size. The operational frequency peaks for clusters of size 256 (tolerable inner-fanout). As the number of clusters doubles, this peak in operational frequency is only maintained when buffering the XML stream across clusters; otherwise, the operational frequency deteriorates due to overclustering. Moreover, overclustering increases resource utilization to replicate the parser and global stack, and it reduces opportunities to exploit commonalities across queries if desired; this occurs when mapping less than 64 queries to a single cluster. Therefore, we conclude that the cluster size should be of size 128 or 256 queries

in order to achieve resource/performance balance. Note that when doubling the query size, the cluster size should be halved.

5.5. Query Compiler Overview

Hardware is automatically generated from user-defined XPath queries using a query compiler developed in C++. The automatic hardware generation step requires around one second. The most notable compiler options include setting a cluster size (Section 5.4), generating custom (Section 5.2) or programmable (Section 5.3) hardware, enabling column-level optimizations (Section 5.2.2), setting the max XML document depth, and setting the number of stack columns (programmable stacks).

When specifying customized hardware, the query-to-hardware compilation is required for every query set. Conversely, programmable stacks need to be generated once, initially. Another tool is needed to generate the configuration bits for every query set. This tool is aware of the underlying architecture specifications (number of columns, number of configuration bits per column, etc.), and generates configuration bits within a second.

6. EXPERIMENTAL EVALUATION

In this section, the performance of all the aforementioned FPGA approaches is evaluated, alongside an adaptation of our filtering mechanism on GPUs, and two state-of-the-art software (CPU-based) approaches namely, YFilter [Diao et al. 2003] and FiST [Kwon et al. 2005].

For the experiments, we utilize the DBLP DTD provided by the University of Washington Repository[7] to generate XML documents and user profiles. XML documents of maximum depth 16 and varying sizes were generated using the ToXGENE XML Generator [Barbosa et al. 2002]. We make use of two main batches of documents for our experiments.

—*Batch 'small_documents'*. 5,000 documents of average size 220 KB each.
—*Batch 'medium_documents'*. 500 documents of average size 2.2 MB each.

Each XML document consists only of open and close tag events, one per line. Each tag was replaced with a 2-byte ID. Using this scheme, the number of XML events per document can be deduced by dividing the document size (bytes) by 5.5, the latter being the average line size: an open tag is 5 bytes long '<__>\n', whereas a close tag is 6 bytes long '</__>\n'. The total size of all documents of each batch is around 1,100MB; hence, every batch corresponds of around 200 million events, across all documents.

Query datasets, each containing distinct queries, with varying depth, percentage occurrence of ancestor-descendant axis and wildcards, were generated using the YFilter query generator [Diao et al. 2003].

The properties of query profiles are as follows.

—Max query depth = 4 or 8 nodes.
—Number of queries = 32, 64, 128, 256, . . . 32K.
—Percent occurrence of ancestor-descendant axis ('//') and wildcard path nodes ('*') = 5, 15, & 25 % occurrence.

Due to the streaming nature of pub-sub systems, throughput (MB/s, events/s) is used in our experiments as a performance metric. Throughput is inversely proportional to the wall-clock running time and is derived using the total size of all documents per batch. Throughput denotes how much information can be processed per unit time.

[7] http://www.cs.washington.edu/research/xmldatasets.

Here, a filtering setup with higher throughput is one less likely to drop packets and able to process documents faster. Furthermore, the choice of a fixed XML tag size (regardless of the size itself) is made in order to derive throughput as measured in events/s, from throughput as measured in MB/s. The former is important for measuring the performance of filtering engines regardless of the XML data based solely on the structure.

End-to-end performance is measured for all platforms (FPGA, GPU, CPU) such that the XML documents start off as located on the CPU RAM, filtering is performed, and finally the filtering results reside on the CPU RAM. Since XML parsing is orthogonal to this work, parsing time is not included in performance measurements. With respect to the FPGA implementation, a lightweight parser is deployed in order to be able to stream a raw XML document. Otherwise, the FPGA would be at a higher and an unrealistic advantage. Furthermore, the parser is just another (single) stage in the FPGA XML filtering pipeline, affecting latency and throughput. By omitting parsing time for CPU-based approaches, we assume that CPU parsers are at least as efficient as hardware parsers. That is, in fact, opposite to practical implementations, where hardware parsers outperform their software counterpart. In conclusion, reported FPGA speedup would be even higher when compared to a parsing+filtering software setup.

6.1. Experimental Evaluation of FPGA-Based Approaches

A study on the resource utilization and performance of the proposed FPGA-based solutions follows.

6.1.1. Setup and Platform. Our FPGA platform consists of a Pico M-501 board connected to an Intel Xeon processor via eight lanes of PCI-e Gen. 2.[8] We make use of one Xilinx Virtex 6 FPGA LX240T, a low- to mid-size FPGA relative to modern standards. The PCIe hardware interface and software drivers are provided as part of the Pico framework.

Our hardware XML filtering circuit communicates with the input and output PCIe interfaces through one stream each way, with dual-clock BRAM FIFOs in between our logic and the interfaces. Hence, the clock of the filtering logic is independent of the global clock. The PCIe interfaces incur an overhead of ≈8% of available FPGA resources.

The RAM on the FPGA board does not reside in the same virtual address space of the CPU RAM. Data is streamed from the CPU RAM to the FPGA. Since the proposed solution does not require memory offloading, RAM on the FPGA board is not used (i.e., stacks are built using the FPGA logic).

Synplify Pro 2010-09 is used for synthesis, and Xilinx ISE 14 for PAR. FSM exploration, resource pre-packing, and resource-sharing optimizations are activated during synthesis.

6.1.2. Trade-Offs and Resource Utilization. The resource utilization of FPGA slices is shown in Figure 10(a), corresponding to the three implementations of the filtering algorithm on FPGAs, namely, as follows.

(1) *Customized (Query length = 4)*. An implementation of the custom hardware approach described in Section 5.2 with PSS optimizations on and clusters of size 256 queries.
(2) *Customized (Query length = 8)*. An implementation of the custom hardware approach described in Section 5.2 with PSS optimizations on and clusters of size 128 queries.

[8]Pico Computing M-Series Modules. `http://www.picocomputing.com/m_series.html`.

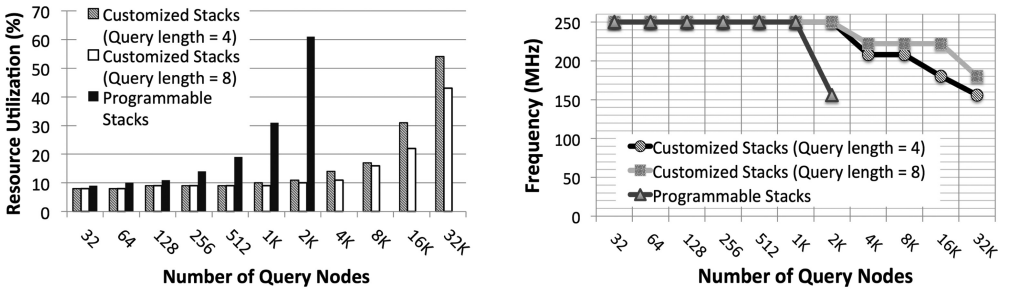


Fig. 10. (a) FPGA slice resource utilization and (b) operational frequency (MHz) of FPGA-based XML filtering approaches on a Xilinx Virtex 6 LX240T FPGA, as part of a PICO M-501 platform. Lower frequencies are due to the larger filtering circuits.

(3) *Programmable*. An implementation of the programmable hardware approach described in Section 5.3 with a cluster size of 128 columns.

Note that the programmable implementation makes use of smaller clusters (i.e., size 128) than the customized counterpart. This smaller cluster size is preferable, since programmable clusters are bigger (i.e., use more resources) than customized ones, hence negatively affecting the critical path.

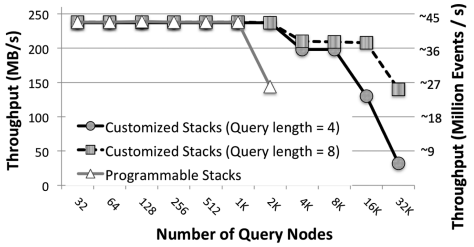
The XML maximum depth is assumed to be 16—a relaxed limitation on the average XML document depth to be processed. Deeper XML documents can be supported with minimal penalty on resource utilizations due to the availability of 32-row LUTs on modern FPGAs.

The data depicted in Figure 10(a) is respective to query nodes rather than queries. The programmable hardware consists of hardware columns, regardless of the number of queries or size of the queries mapped. Moreover, looking from a node perspective, we can see that stack optimizations are more effective with longer queries, saving around 25% in resources when supporting the same number of nodes (custom length 8 vs. length 4).

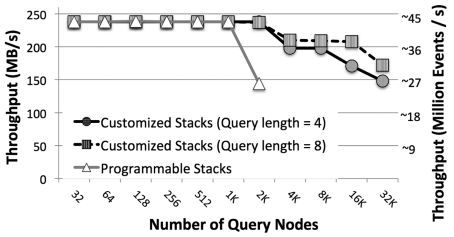
As expected, the custom hardware benefits from the reduced resource utilization, and that is not solely due to the PSS optimizations. Custom hardware uses on average seven times less resources than a programmable approach (up to 12 times less). Note that we can further optimize the custom circuitry by making use of the common prefix optimization. This optimization can be combined with the stack optimizations presented in Section 5.2.2. The expected reduction in query nodes would be as studied in Moussalli et al. [2011a]. We omit further exploration of this option here for brevity.

Doubling the query length requires on average two times more resources with the programmable implementations, and that is due to the doubling of the stack size (i.e., number of columns) and the resulting need for intercolumn logic. Conversely, doubling the query length would incur on average 1.4 times more resources when considering custom logic. This ratio is smaller than that of generic hardware due to stack compaction which minimizes stack depth for any query width. Note that nonlinear behavior in resource utilization while doubling the number of queries is due to the heuristic-based nature of the tools. Moreover, in the case of a circuit easily fitting on chip, certain resource utilization optimization constraints are relaxed in order to achieve higher performance at the cost of added resources.

Though custom hardware approaches utilize considerably less resources than their programmable counterpart, this comes at the cost of high reconfiguration time, where updating queries in the custom hardware reconfiguration requires a new run through



(a) Batch of 5K documents, ≈220KB each.



(b) Batch of 500 documents, ≈2.2MB each.

Fig. 11. Throughput of the FPGA-based XML filtering approaches for (a) a batch of 5K documents ≈220KB each, and (b) a batch of 500 documents, ≈2.2MB each. Throughput measured in XML events/s can be directly derived such that every 5.5 bytes of XML constitute one event (by design of the test documents).

the synthesis, place, and route tools, which could take hours to complete with larger designs. On the other hand, updating queries in the programmable architecture requires updating the configuration stream and streaming it to hardware, a process requiring less than a second overall.

6.1.3. Performance Evaluation. The filtering mechanisms on the FPGA chip depict respective deterministic throughputs; this is in contrast to CPU- and GPU-based filtering, where throughput is affected by the document size and contents.

The parser deployed on the FPGA is able to process a stream of up to one character (8 bits) per hardware cycle, generating events (push/pop) that are then forwarded to the stacks. The stacks guarantee processing one event per cycle, as no memory external to the FPGA chip is used. Note that XML events generated by the parser are less frequent than document characters; in other words, the rate of events is once per several cycles. As a result, the throughput of the filtering mechanisms is deterministic, is rated at one event/cycle, and is independent of the document size and contents. However, the throughput of the FPGA platform as a whole is not deterministic, since data has to be sent from the CPU to the FPGA and filtering results back to the CPU memory. Communication between the CPU and FPGA is penalized by the setup time of every transfer and the amount of transfers.

With respect to parsing performance, the number of characters in a tag only affects parsing and not filtering. However, it does not affect the performance of parsers, which is measured in characters/cycle, and is irrespective of the tag size. The number of unique tags could have an effect on parser performance. In practice, the number of unique tags needed by queries in a cluster is not large (in the range of at most a few hundreds), and will not limit parsing performance.

Reading an XML document, parsing it and filtering are all performed in parallel, a noted advantage versus CPU- and GPU-based approaches. Furthermore, since the input and output PCIe interfaces are independent, streaming results back to the CPU can also be parallelized with the parsing/filtering, as long as match states are buffered.

The operational frequencies at which the FPGA filtering circuits run are shown in Figure 10(b). The physical platform limitation on the operational frequency is 250MHz, which is easily achieved by many filtering circuits (for 1K query nodes and less, and 2K custom FPGA query nodes). As the FPGA utilization increases through doubling the number of queries, the frequency then deteriorates due to the added complexity and area (longer delays) of the resulting circuits.

Figures 11(a) and 11(b) depict the throughput of FPGA-based XML filtering approaches for a batch of 5K small (≈220KB) and 500 medium (≈2.2MB) XML documents, respectively. Measuring performance in XML events/s can be directly derived such that every 5.5 bytes of XML constitute one event (by design of the test documents).

Throughput includes the end-to-end time of streaming the XML documents from CPU RAM to the FPGA, filtering, and reading the results back for each document from the FPGA to the CPU RAM. Note that filtering results can be kept on the local FPGA memory (and not streamed to a CPU host memory) in case the routing mechanism to subscribers is implemented on chip (this is not applicable to GPU filtering systems).

Initially, the throughput of individual architectures is limited by the maximum operational frequency in addition to a small overhead incurred for transfer setup and reading back results from the FPGA. As noted earlier, the on-chip throughput is independent of the document size and contents. Nonetheless, overall system performance deteriorates for batches of small documents with a high number of query nodes ($\geq$16K), where the time to report matches becomes comparable to the time to receive each document. This is in contrast to the performance noted for larger documents (Figure 11(b)), where the operational frequency is always the main limitation, regardless of the number of matches to report, being minimal compared to each document. Also, since there are fewer 2.2MB documents than 220KB ones, there will be fewer transfers to/from the FPGA respective to the former.

The next consideration is the effect of wildcards and ancestor-descendant relations on the FPGA resource utilization and performance (operational frequency). By design, no effect is to be witnessed on the programmable FPGA approach, where support for '*' and '//' is deployed by default for all stack columns, even when not used by a given configuration. We ran several experiments to study the effect of varying the percentage of occurrence of wildcards and ancestor/descendant relationships on the customized approach (data omitted for brevity). The witnessed fluctuations in resource utilization and operational frequency were minimal and subsumed by the heuristic nature of the synthesis/place-and-route tools. As a result, all FPGA architectures were not affected by wildcards and ancestor-descendant relations.

It should be noted that the performance of our filtering system can be further improved through the use of high-performance XML parsers, as in El-Hassan and Ionescu [2009]. Such parsers are able to sustain a processing rate of of two bytes per cycle, on average. This would double the maximum filtering throughput. High performance XML parsers can be deployed in our system with little modification required. Another approach would be to run the parser at a higher frequency than the filtering mechanism, a highly plausible approach, since XML events are less frequent than document characters. Nonetheless, such parser optimizations are orthogonal to our work and are out of the scope of this article. Using optimized parsers would help fight the lower operational frequency of certain circuits (Figure 10(b), $>$16K customized and 2K programmable nodes), and would help filter at higher bandwidth links when needed (10G ethernet).

6.2. Performance of GPU-Based Approaches

The performance of the GPU-based approaches is measured on an NVIDIA TESLA C1060 GPU; a total of 30 streaming multiprocessors (SMs) comprising of 8 streaming processors (SPs) each are available.

An in-depth description of the adaptation of the proposed filtering solution (Section 4) is provided [Moussalli et al. 2011a]. Here, a complete new set of experiments is presented, focused on batches of small XML documents (a common consideration of pub-sub systems), rather than single large documents (which was the focus in Moussalli et al. [2011a]) .

A new study is further presented here, comparing the performance of mapping queries holistically to a GPU thread each, versus mapping queries to GPU blocks (one stack column per GPU thread). Through the latter, several streaming processors (SPs) are processing a single query.

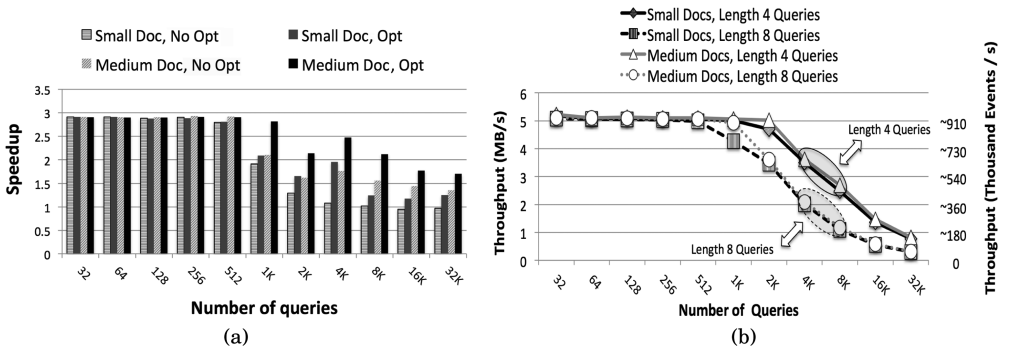


Fig. 12. (a) Speedup of mapping queries to GPU blocks versus mapping queries to GPU threads. (b) Throughput (MB/s) of the GPU-based approaches with queries mapped to blocks, for queries of length 4 and 8, with respect to a batch of 5K XML (small) documents of size 220KB each and 500 XML (medium) documents of size 2.2MB each.

Figure 12(a) depicts the speedup of mapping queries (of length 8) to GPU blocks versus one query per thread, with (Opt) and without (No Opt) the common prefix optimization applied, for a 220KB and a 2.2MB XML document (no batches). Note that the common prefix optimization is not applicable to the query set when one query is matched per thread.

With too few query matching engines on the GPUs (less than 1K queries), the GPU is underutilized, and the speedup is highest. At 1K queries, we exhibit a *breaking point*, where speedup lowers due to the overutilization of the GPU; more parallelism (queries) results in linearly more time (less speedup) due to the unavailability of processors.

Mapping queries to blocks results in speedup initially (around 2.9X) due to the fact that the parallelism offered by the GPU is exploited in a more efficient manner. For instance, when mapping 32 queries of length 8 to threads, 32 SPs are used (out of the available 240). On the other hand, when mapping 32 queries of length 8 to blocks, more SPs are used (one per column), and more columns can be evaluated in parallel. This advantage is exploited less with more stack columns to be evaluated.

Furthermore, from Figure 12(a), we observe that the document size has virtually no effect on speedup until the breaking point, where a larger document results in more speedup.

The common prefix optimization results in fewer stack columns, hence less work to be done on the GPU. This in turn results in more speedup, notably at 1K queries, where speedup remained almost constant for a 2.2MB document, and beyond 16K queries, where slowdown is depicted with No Opt and a 220KB document.

For the remainder of this section, we will only consider mapping queries to GPU blocks (one stack column per GPU kernel), with the common prefix optimization applied.

Figure 12(b) depicts the GPU filtering throughput (MB/s) for queries of length 4 and 8, with respect to a batch of 5K XML (small) documents of size 220KB each and 500 XML (medium) documents of size 2.2MB each. Throughput measured in XML events/s can be directly derived such that every 5.5 bytes of XML constitute one event (by design of the test documents). This metric is relevant to the GPU platform because it only receives events from parsed documents and no XML data.

As XML parsing is orthogonal to our work, throughput measurements do not include the time to parse XML documents. Throughput includes the time to send parsed documents from the CPU RAM to the GPU, and to retrieve the parsing results back to the CPU RAM.

Throughput here is much lower than the throughput provided by the FPGA-based XML filters; when filtering using GPUs, throughput of few MB/s is witnessed versus hundreds of MB/s when using FPGA-based filtering. Even though GPUs offer a certain level of parallelism, all processing cores are general-purpose cores, whereas the circuitry on the FPGA is highly optimized for the application at hand. Also, in spite of GPUs being able to (virtually) filter any number of queries, the parallelism provided by FPGAs is not bound by time multiplexing, where a single computing module is used by different queries at different time instances. Hence, all queries on the FPGA are being filtered in parallel, and the document is being read exactly once. Finally, (all the) GPU processors have to read the document from global memory, thus limiting performance.

Throughput starts off as constant until reaching a *breaking point* where the GPU becomes overutilized. Beyond the breaking point, throughput almost halves as the number of queries doubles. The breaking point affects queries of length 8 earlier than queries of length 4, since queries in length 8 imply almost double the number of stack columns (effective GPU work), with minor differences due to the common prefix optimization. The throughput of queries of length 4 is approximately one step behind that of length 8 queries in terms of deterioration. Making use of a GPU with more cores while using the same architecture and memory hierarchy will not result in better performance prior to the breaking point. However, it would linearly help delay the occurrence of the breaking point (i.e., moving the breaking point to the right of the plot).

For a given query set, throughput is minimally higher for larger documents. This is due to the extra CPU-GPU communication implied by using smaller documents, since the latter are more in number than the larger documents. However, as seen in Figure 12(b) (gap in the throughput between length 4 queries, and gap in the throughput between length 8 queries), this overhead is minimal, and we can deduce that the time to send documents to the GPU and receive filtering results back is minor compared to the filtering time spent on the GPU. The CPU-GPU send-receive time was measured to be less than 0.2% of the overall filtering time (data omitted for brevity).

We ran several experiments varying the percentage of occurrence of '*' and '//' (results omitted due to the lack of space). Negligible fluctuations in performance were witnessed (average $<1\%$). Increasing the percentage of occurrence has no effect, since '*' and '//' are supported by default for all stack columns.

6.3. Performance of CPU-Based Approaches

Numerous software approaches have been proposed for the XML filtering problem. Here, two leading software approaches are considered: YFilter and FiST, that are representative of the different strategies proposed for XML filtering. Despite their differences, for each XML stream event consumed, both approaches identify a set of active states, the event buffered, and the next set of active states are maintained in memory before the next input event is considered. Our results showed that the maintenance and processing of a large number of active states degrades the performance of these approaches. The number of active states increases when query complexity (i.e., query length or percentage occurrence of '//' and '*') or XML document size is increased, incurring up to 7GB memory footprint and reducing the throughput. The memory usage increases slightly when doubling the number of queries but is more sensitive to the document size.

The CPU-based approaches were run on a single core of a 2.33GHz Intel Xeon machine with 8GB of RAM running Linux Red Hat 2.6. In our experiments, YFilter and FiST exhibited comparable performance and behavior; hence, for the remainder of this section, we will only show results for the YFilter approach. Since YFilter and FiST are optimized to run on single processors, we first run software experiments on a

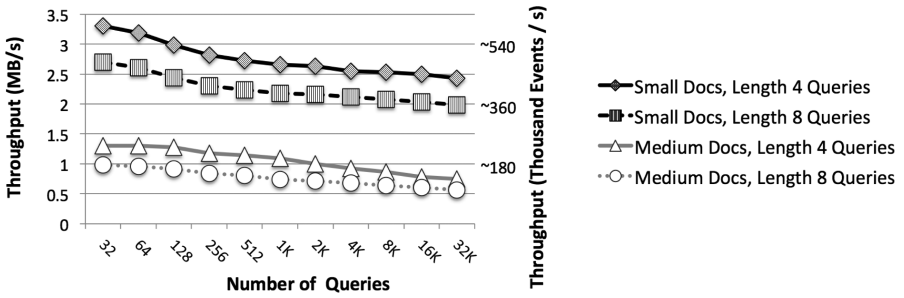


Fig. 13. Throughput (MB/s) of the CPU-based approach (YFilter) for queries of length 4 and 8 with 15% probability of occurrence of '*' nodes and '//' relations, and varying documents sizes (namely 5K documents ≈220KB each, and 500 documents, ≈2.2MB each).

single CPU, as intended by design. These numbers give us a good understanding of the characteristics of the three used platforms (summarized in Table I). We later show many (independent) instances of YFilter running on a multicore, splitting the document set equally among all YFilter threads/instances. As XML parsing is orthogonal to this work, throughput measurements do not include parsing time; further, prior to filtering, XML documents reside on the CPU RAM.

Figure 13 depicts the throughput using YFilter with input XML document sizes of ≈220KB (5K documents) and ≈2.2MB (500 documents), respectively, both for queries of depth of 4 and 8, whilst doubling the number of queries.

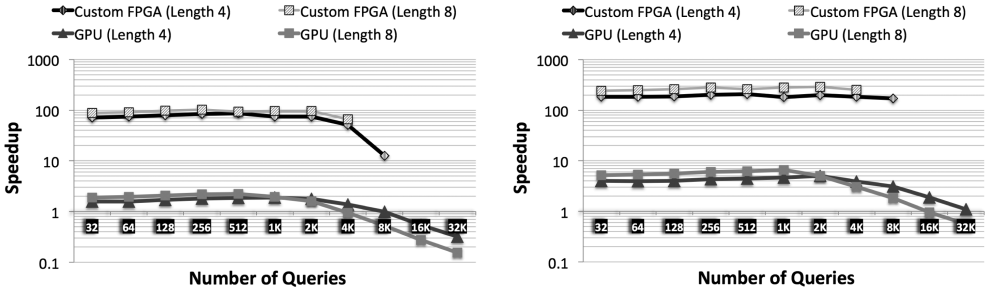
Three main observations can be made from Figure 13. First, the performance slowly deteriorates while doubling the number of queries, and unlike the GPU-based approaches, there is no breaking point; CPU-based approaches are able to scale very well with added queries due to the optimized data structures used. These cannot be mapped as-is on the GPU due to the nature of the memory hierarchy and the lack of parallelism offered by the CPU approaches (complexity vs. parallelism trade-off). Second, document size has a considerable effect on throughput: larger documents resulted in an average 2.8X slowdown. This is in contrast with FPGA and GPU filtering, where larger documents resulted in a higher throughput, since CPU-accelerator communication is minimized with larger documents. On the other hand, YFilter is negatively affected by larger documents, since larger data structures need to be maintained. Third, doubling the query length results in an average 28% slowdown. This is in contrast to the GPU accelerator, where throughput halves after the breaking point (and is not affected otherwise); similarly for the FPGA accelerators, doubling the query length has minimal effect on the operational frequency, until most of the FPGA resources are utilized.

Finally, the effect on performance of varying percentages of '*' and '//' was studied by varying the percentage of occurrence of '*' and '//' from 5% to 25% (data is not shown for brevity). Increasing the percentage showed to constantly have a negative effect by deteriorating performance anywhere from 6% to 30% by 10% more '*' and '//' in a given query set. This is due to the added level of freedom, hence complexity, to be supported.

6.4. Comparing FPGA-, GPU-, and CPU-Based Filtering

We proceed with the performance comparison of the customized FPGA circuit to the GPU-based filtering and to a reference CPU-based approach (YFilter) while setting the percentage of occurrence of '*' and '//' to 15%.

Speedup (on a log scale) is shown in Figures 14(a) and 14(b) for the customized FPGA- and GPU-based approaches over the CPU-based approach for batches of XML documents of size 220KB and 2.2MB, respectively. Slowdown is depicted for speedup values less than 1.



(a) Batch of 5K documents, ≈220KB each. (b) Batch of 500 documents, ≈2.2MB each.

Fig. 14. Speedup of the customized FPGA-based and the GPU-based approaches over a CPU-based approach for queries of length 4 and 8, with varying document sizes. Slowdown is depicted for speedup values less than 1.

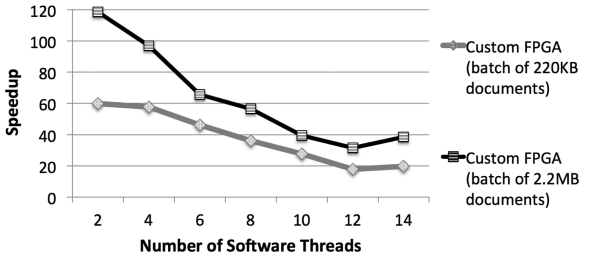


Fig. 15. Speedup of the custom FPGA approach versus a multithreaded version of YFilter running on a 12-core machine. 2K queries of length 8 are assumed, with a 15% probability of occurrence of '*' and '//'. Batches of 5K 220KB documents and 500 2.2MB documents are used.

Overall, using FPGAs provides up to 2.5 orders of magnitude speedup (average of 79X and 225X for batches of 220KB and 2.2MB documents, respectively). On the other hand, using GPUs provides up to 6.6X speedup (average of 2.7X) before the throughput saturation at the breakpoint. Prior to hitting the break point, speedup slightly increases gradually, since the GPU throughput is constant, whereas that of the CPU-based approaches is not. Moreover, speedup is higher for length 8 queries prior to the breaking point, which is reached faster with length 8 queries. Slowdown is witnessed beyond 8K queries for batches of 220KB documents, and at 32K queries of length 8 for batches of 2.2MB documents.

We (naively) extended YFilter to run on multicores by filtering every document using a separate thread. Hence, every thread is practically an instance of YFilter, and filtering for a single document is performed on a single core.

Figure 15 shows the speedup of the custom FPGA approach versus the multithreaded version of YFilter running on a 12-core machine. 2K queries of length 8 are assumed, with a 15% probability of occurrence of '8' and '//'. Batches of 5K 220KB documents and 500 2.2MB documents are used.

Making use of more CPU cores will almost linearly result in a higher performance from YFilter, until the number of threads exceeds the number of CPU cores. The custom FPGA approach is still 17X and 31X faster than YFilter, when all 12 cores are used, for batches of 200KB and 2MB documents, respectively.

In summary, the effects of the factors studied on the different filtering platforms are encapsulated in Table I.

Note that it is possible to map the stack-based approach onto CPUs while borrowing some of the advantages of hardware approaches, such as low memory footprint. Some

optimizations can be applied in order to only update the needed stack columns based on each XML events. However, the GPU implementation of our stack-based algorithm provides a clear insight on what to expect from software; instead, here all stacks are updated in parallel using low-latency memories, potentially providing higher performance than CPUs.

7. CONCLUSIONS

This article examines how to exploit the parallelism found in XPath filtering systems using accelerators. By converting XPath expressions into custom stacks, our architecture is the first to provide support for complex XPath structural constructs, such as parent-child and ancestor-descendant relations, whilst allowing wildcarding and recursion. We also present a novel method for matching user profiles that supports dynamic query updates using a programmable FPGA. This is in addition to the GPU-based filtering based on the presented filtering algorithm. An exhaustive performance evaluation of all accelerators is provided with comparison to state-of-the-art software approaches.

Using an incoming XML stream, thousands of user profiles are matched simultaneously with minimal memory footprint by stack-based matching engines. This is in contrast to conventional approaches bound by the sequential aspect of software computing, associated with a large memory footprint (over 7GB).

On average, using customized circuitry on FPGAs yields speedups of up to 2.5 orders of magnitude, whereas using GPUs provides up to 6.6X speedup, and in some cases, slowdown, versus software running on a single CPU core. The FPGA approaches are up to 31X faster than software running on a 12-CPU core. Finally, a novel approach for supporting on-the-fly query updates on the FPGA was presented, resulting in an average of 7X more resources than the custom FPGA approach.

REFERENCES

Shurug Al-Khalifa, H. V. Jagadish, Nick Kodus, Jignesh Patel, Divesh Srivastava, and Yuqing Wu. 2002. Structural Joins: A primitive for efficient XML query pattern matching. In *Proceedings of the 18th International Conference on Data Engineering*. IEEE Computer Society, 141.

Mehmet Altinel and Michael J. Franklin. 2000. Efficient filtering of XML documents for selective dissemination of information. In *Proceedings of the 26th International Conference on Very Large Data Bases*. Morgan Kaufmann Publishers Inc., San Francisco, CA, 53–64.

Naiyong K. Ao, Fan Zhang, Di Wu, Douglas Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Lin Sheng. 2011. Efficient parallel lists intersection and index compression algorithms using graphics processing units. *Proc. VLDB Endow.* 4, 8, 470–481.

Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. 2002. ToXgene: A template-based data generator for XML. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 616.

K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Junichi Tatemura, and Divyakant Agrawal. 2006. AFilter: Adaptable XML filtering with prefix-caching suffix-clustering. In *Proceedings of the 32nd International Conference on Very Large Data Bases*. VLDB Endowment, 559–570.

C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. 2002. Efficient filtering of XML documents with XPath expressions. *VLDB J.* 11, 4, 354–379.

Zefu Dai, Nick Ni, and Jianwen Zhu. 2010. A 1 cycle-per-byte XML parsing accelerator. In *Proceedings of the 18th International Symposium on FPGAs*. ACM, New York, NY, 199–208.

Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. 2003. Path sharing and predicate evaluation for high-performance XML filtering. *ACM Trans. Datab. Syst.* 28, 4, 467–516.

Fadi El-Hassan and Dan Ionescu. 2009. SCBXP: An efficient hardware-based XML parsing technique. In *Proceedings of the 5th Southern Conference on Programmable Logic*. IEEE, 45–50.

Gang Gou and Rada Chirkova. 2007. Efficient algorithms for evaluating XPath over streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 269–280.

Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. 2004. Processing XML streams with deterministic automata and stream indexes. *ACM Trans. Datab. Syst.* 29, 4, 752–788.

Ashish Kumar Gupta and Dan Suciu. 2003. Stream processing of XPath queries with predicates. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 419–430.

Bingsheng He, Qiong Luo, and Byron Choi. 2006. Cache-conscious automata for XML filtering. *IEEE Trans. Knowl. Data Eng.* 18, 12, 1629–1644.

Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. 2008. Relational joins on graphics processors. In *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'08)*. ACM, New York, NY, 511–524.

Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony Nguyen, Tim Kaldewey, Victor Lee, Scott Brandt, and Pradeep Dubey. 2010. FAST: Fast architecture sensitive tree search on modern CPUs and GPUs. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*.

Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. 2005. FiST: Scalable XML document filtering by sequencing twig patterns. In *Proceedings of the 31st International Conference on Very Large Data Bases*. VLDB Endowment, 217–228.

M. D. Lieberman, J. Sankaranarayanan, and H. Samet. 2008. A fast similarity join algorithm using graphics processing units. In *Proceedings of the IEEE 24th International Conference on Data Engineering (ICDE'08)*. IEEE, 1111–1120.

Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. 2002. A transducer-based XML query processor. In *Proceedings of the 28th International Conference on Very Large Data Bases*. VLDB Endowment, 227–238.

J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. 2004. XML accelerator engine. In *Proceedings of the 1st International Workshop on High Performance XML Processing*. Springer Berlin.

Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Walid Najjar, and Vassilis J. Tsotras. 2009. Boosting XML filtering through a scalable FPGA-based architecture. In *Proceedings of the 4th Conference on Innovative Data Systems Research*. ACM.

Mirella M. Moro, Petko Bakalov, and Vassilis J. Tsotras. 2007. Early profile pruning on XML-aware publish-subscribe systems. In *Proceedings of the 33rd International Conference on Very Large Data Bases*. VLDB Endowment, 866–877.

R. Moussalli, R. Halstead, M. Salloum, W. Najjar, and V. J. Tsotras. 2011a. Efficient XML path filtering using GPUs. In *Proceedings of the Workshop on Accelerating Data Management Systems (ADMS)*.

Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis Tsotras. 2010. Accelerating XML query matching through custom stack generation on FPGAs. In *Proceedings of the 5th International Conference on High Performance Embedded Architectures and Compilers*. Lecture Notes in Computer Science, vol. 5952. Springer, Berlin, 141–155.

R. Moussalli, M. Salloum, W. Najjar, and V. J. Tsotras. 2011b. Massively parallel XML twig filtering using dynamic programming on FPGAs. In *Proceedings of the IEEE 27th International Conference on Data Engineering (ICDE)*. IEEE.

Rene Mueller, Jens Teubner, and Gustavo Alonso. 2009. Streams on wires: A query compiler for FPGAs. *Proc. VLDB Endow.* 2, 1, 229–240.

Feng Peng and Sudarshan S. Chawathe. 2003. XPath queries on streaming data. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. ACM, New York, NY, 431–442.

Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Amo Jacobsen. 2010. Efficient event processing through reconfigurable hardware for algorithmic trading. In *Proceedings of the International Conference on Very Large Data Bases (VLDB)*.

Mariam Salloum and V. J. Tsotras. 2009. Efficient and scalable sequence-based XML filtering system. In *Proceedings of the 12th International Workshop on the Web and Databases (WebDB)*. ACM.

Pranav S. Vaidya, Jaehwan John Lee, Francis Bowen, Yingzi Du, Chandima H. Nadungodage, and Yuni Xia. 2010. Symbiote: A reconfigurable logic assisted data stream management system (RLADSMS). In *Proceedings of the International Conference on Management of Data*. ACM, New York, NY, 1147–1150.

Received November 2012; revised July 2013; accepted November 2013