UNIVERSITY OF CALIFORNIA
RIVERSIDE

Hardware Acceleration of Database Applications

A Dissertation submitted in partial satisfaction
of the requirements for the degree of

Doctor of Philosophy

in

Computer Science

by

Roger Moussalli

March 2013

Acknowledgments

I am grateful to my advisor, Dr.Walid Najjar, whose invaluable guidance I will always cherish; to Dr.Mazen Saghir without whose help and trust, I would not have been here; and to Dr.Vassilis J. Tsotras, for all the kindness, help and advice.

ABSTRACT OF THE DISSERTATION

Hardware Acceleration of Database Applications

by

Roger Moussalli

Doctor of Philosophy, Graduate Program in Computer Science
University of California, Riverside, March 2013
Dr. Walid Najjar, Chairperson

General purpose computing platforms have generally been favored over customized computational setups, due to the simplified usability and significant reduction of development time. These general purpose machines make use of the Von-Neumann architectural model which suffers from the sequential aspect of computing and heavy reliance on memory offloading.

This dissertation proposes the use of hardware accelerators such as Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) as a substitute or co-processor to general purpose CPUs, with a focus on database applications. Here, large amounts of data are queried in a time-critical manner. This dissertation shows that using hardware platforms allows processing data in a streaming (single pass) and massively parallel manner, hence speeding up computation by several orders of magnitude when compared to general purpose CPUs. The complexity of programming these parallel platforms is abstracted from the developers, as hardware constructs are automatically generated from high-level application languages and/or specifications.

This dissertation explores the hardware acceleration of XML path and twig filtering, using novel dynamic programming algorithms. Publish-subscribe systems present the state of the art in information dissemination to multiple users. Current XML-based publish-subscribe systems provide users with considerable flexibility allowing the formulation of complex queries on the content as well as the (tree) structure of the streaming messages. Messages that contain one or more matches for a given user profile (query) are forwarded to the user.

This dissertation further studies FPGA-based architectures for processing expressive motion patterns on continuous spatio-temporal streams. Complex motion patterns are described as substantially flexible variable-enhanced regular expressions over a spatial alphabet that can be implicitly or explicitly anchored to the time domain. Using FPGAs, thousands of queries are matched in parallel. The challenges in handling several constructs of the assumed query language are explored, with a study on the tradeoffs between expressiveness, scalability and matching accuracy (eliminating false-positives).

Finally, the first parallel Golomb-Rice (GR) integer decompression FPGA-based architecture is detailed, allowing the decoding of unmodified GR streams at the deterministic rate of several bytes (multiple integers) per hardware cycle. Integer decompression is a first step in the querying of inverted indexes.

Contents

List of Figures

Chapter 1

Introduction

Due to their relative ease of use, general purpose processors are commonly favored at the heart of many computational platforms. These processors are deployed in environments with varying requirements, ranging from personal electronics, to game consoles and up to server-grade machines. General purpose CPUs follow the Von-Neumann model, and execute instructions sequentially. Furthermore, performance does not always linearly scale in multi-processor environments, mostly due to the challenges of data sharing across cores. As it is non-trivial for these CPUs to satisfy the increasing time-critical demands of several applications, they are often coupled with application- or domain-specific parallel accelerators, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), which strive given a certain class of instructions and memory access patterns.

Graphics Processing Units (GPUs) are emerging as computational platforms comprising of several hundreds of simple processors operating in a parallel fashion. While intended to be used solely for graphic applications, they are generally employed to accelerate solving general purpose problems of SIMD (Single Instruction Multiple

Data) type, thus referred to as General Purpose GPUs (GPGPUs). GPGPUs are used as co-processors to which the main CPU passes a stream of data; the GPGPU then processes the data with minimal memory footprint, and returns the processing results to the CPU. For additional details on the architecture and programming model of GPUs, see Appendix A.

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

1.1 Related Work

As traditional platforms are increasingly hitting limitations when processing high volumes of streaming data, researchers are investigating GPUs and FPGAs for database applications, for uses including enterprise data warehousing, business intelligence, predictive analytics and business continuity planning. Recent work has focused on the adoption of Field Programmable Gate Arrays (FPGAs) for the processing of large data streams [75, 14, 74, 63, 61, 78, 68, 85]. Netezza [62] employs FPGAs as part of their high-performance data warehouse appliances and advanced analytics solutions. [75] proposes the use of FPGAs to achieve realtime analytics on enterprise data. The FPGA-led performance boost up of compression/decompression has long been an active field of research, with the main focus on speeding up low-latency storage access [14, 74, 63]. The Glacier component library is presented in [61] which includes logic circuits of common operators such as selection, aggregation, and grouping for stream processing. [85] investigated the speedup of the frequent item problem using FPGAs, while in [78] FPGAs are utilized for complex event detection which uses regular expressions to represent events. Predicate-based filtering on FPGAs was investigated by [68] where user profiles are expressed as a conjunctive set of boolean filters.

Though FPGAs have shown to provide advantages over traditional architectures, they are limited by small resources and potentially expensive programming time. These two limitations have led researches to examine GPUs, since they offer flexibility of reprogramming.

GPUs have evolved to the point where many real-world applications are easily implemented and run significantly faster than on multi-core systems, thus, a large number of recent work has investigated GPUs for the acceleration of database applications[26, 32, 25, 4, 35, 45]. In [32] the authors utilized GPUs to accelerate relational joins, while in [25] GPUs are utilized for computing Fourier transforms. In [35] a CPU-GPU architecture is presented to accelerate tree-search, which was shown to have low latency and support online bulk updates to the tree. Recently, [4] proposed the utilization of GPUs to speed-up indexing by offloading list intersection and index compression operations to the GPU. [45] proposed a similarity join algorithm designed to exploit the parallelism and high data throughput on GPUs.

1.2 Contributions

This dissertation proposes the use of hardware accelerators such as Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) as a substitute or co-processor to general purpose CPUs, with a focus on database applications. Here, large amounts of data are queried in a time-critical manner. Specifically,

- This dissertation proposes the first stream-mode parallel approach for XML path filtering, which does not result in false positives. Wildcard and recursion (nesting) support is offered using this solution. The first implementation and study of FPGA- and GPU-based adaptations of the aforementioned solution are described. The FPGA approaches are shown to be up to 31X faster than software running on 12 CPU cores.

- A novel method for performing ordered/unordered holistic twig XML matching on FPGAs without any false positives is presented, based on the proposed path-style query solution. As is the case with the path queries FPGA setup, the achieved throughput is independent of the complexity of the user queries or the characteristics of the input XML stream. Furthermore, experimental comparison of different granularities of twig matching is presented, namely path-based (root-to-leaf) and pair-based (parent-child or ancestor-descendant). Speedups of over three orders of magnitude is achieved when compared to (single core) state-of-the-art CPU-based approaches.

- The first study on FPGA-based architectures processing expressive, variable-enhanced, motion patterns on spatio-temporal streams is offered. The challenges in handling several constructs of the assumed query language are explored, with a study on the tradeoffs between expressiveness, scalability and matching accuracy (eliminating false-positives). A performance study is presented, where FPGA setups are shown to outperform the current state-of-the-art CPU-based approaches.

- The first FPGA-based fully-parallel implementation of Golomb-Rice integer decompression engines is presented, where no assumptions and modifications are made on the compressed stream. Compression helps increasing the effective bandwidth from slower storage media. Hence, highly efficient decompression engines help hiding the latency of reconstructing the stored data. The presented decoder, capable of processing several bytes per cycle, is shown to outperform an efficient Golomb-Rice CPU-based implementation by up to 52X, while outperforming the high-performance PFOR technique by 4.7X.

The remainder of this dissertation is organized as follows: Sections 1.3, 1.4, and 1.5 respectively introduce each of the aforementioned applications. Chapter 2 describes the novel XML path filtering algorithm as implemented on FPGAs and GPUs. Chapter 3 goes over the non-trivial extension of the path matching algorithm supporting to the more complex twig queries, as applied to FPGAs. Chapter 4 goes over the FPGA querying of spatio-temporal streams. Chapter 5 details the novel FPGA-based no-stall parallel Golomb-Rice integer decompression architecture. Finally, conclusions are presented in Chapter 6.

1.3 XML Filtering

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 interests (also called *profiles*), and an infrastructure for matching subscriber interests with published messages and delivering *matched messages* to the interested subscriber (see Figure 1.1).

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 will 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 [88]. 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

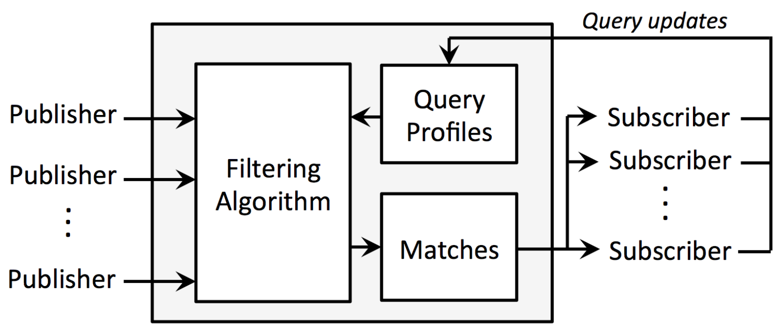


Figure 1.1: Architecture of a pub-sub system. Publishers feed a stream of messages into the system; subscribers post their profiles (queries); an infrastructure matches subscriber interests with published messages and delivers matched messages to the interested subscriber.

also on the structure of the messages. Note that the terms "profile" and "query" are used interchangeably.

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 the 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 [2, 16, 27, 40]. These memory-bound solutions, however, suffer from the Von Neumann bottleneck and are unable to handle large volume of input streams. On the other hand, FPGAs have been shown to be particularly suited for the stream processing of large amounts of data and do not suffer from the memory offloading problem faced by software implementations. Furthermore, GPUs as co-processors are also a favorable option for applications requiring massively parallel computations [32, 4, 35, 45], such that sequential computations are run on the CPU while the computationally-intensive part is accelerated by the highly parallel GPU architecture.

This dissertation examines how to exploit the parallelism found in XPath filtering. Using an incoming XML stream, parsing and matching with thousands of user profiles are performed simultaneously by matching engines. We show the benefits and tradeoffs 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, the proposed 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.

Chapter 2 goes over a novel stack-based dynamic programming filtering algorithm targeting path queries, as applied to FPGAs and GPUs, and whose throughput is independent of the complexity of the user queries or the characteristics of the input XML stream. 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. The filtering algorithm, as described in Chapter 2, appears in [59, 56]. Furthermore, the GPU mechanism described in that same chapter, appears in [56].

Chapter 3 describes non-trivial extensions to the path filtering algorithm to support unordered holistic twig matching on FPGAs without any false positives. Experimental comparison of different granularities of twig matching is presented, namely path-based (root-to-leaf) and pair-based (parent-child or ancestor-descendant). Comprehensive experiments are provided, comparing the throughput, area utilization and the accuracy of matching (percent of false positives) of the holistic, path-based and pair-based FPGA approaches. The proposed approach yields up to three orders of magnitude higher throughput when compared to state-of-the-art single core CPU-based filtering mechanisms. This work, as described in Chapter 3, appears in [57] © 2011 IEEE.

1.4 Querying Spatio-Temporal Databases

In recent years, the wide and increasing availability of collected information in the form of spatio-temporal data has lead to novel research advances in behavioral aspects of the monitored subjects. Using trajectory data harvested by devices such as GPS and cellular technologies, complex queries can be posed to detect specific motion patterns.

In this dissertation, we describe an FPGA-based setup allowing users to query spatio-temporal databases in a very powerful and intuitive way. As streams of trajectory data are harvested from devices, such as GPS and cellular devices, coordinates are then translated into semantic regions that partition the spatial domain; these regions can be grid regions representing areas of interests (e.g., neighborhoods, school districts, cities). Our work is based on complex pattern queries previously defined in [81, 82] to search for specific motion patterns in trajectories. A pattern query is specified as a combination of sequential spatio-temporal predicates, allowing the end user to search for specific parts of interests in trajectory databases. For example, the pattern query "*Find all taxis (trajectories) that first were in downtown Munich in the morning, later passed by Olympiapark around noon, and then were closest to the Munich airport*" provides a combination of temporal, range and Nearest-Neighbor (NN) predicates that have to be satisfied in the specific order. Essentially, flexible patterns cover that part of the query spectrum between the single spatio-temporal predicate queries, such as the range predicate that covers certain time instances of the trajectory life (e.g., "*Find all trajectories that passed by the Deutsches Museum area at 11pm*"), and similarity/clustering based queries, such as extracting similar movement patterns from a trajectories that cover the entire lifespan of the trajectory (e.g., "*Find all trajectories that are similar to a given query trajectory according to some similarity measure*").

Complex pattern queries can also have variable spatial predicates, and thus substantially enhancing the flexibility and expressive power of the framework. An example of a variable-enhanced query is "*Find all trajectories that started in a region X, then visited the downtown Munich, then at some later point visited X again*".

This work serves as a proof-of-concept on the performance benefits of evaluating complex motion pattern queries using FPGAs. Here we explore the challenges of supporting hundreds (up to thousands) of variable-enhanced flexible patterns on FPGAs in streaming (fully-pipelined) fashion, with a study on the tradeoffs between expressiveness, scalability and matching accuracy (eliminating false-positives). Using FPGAs all pattern query predicates are evaluated in parallel over sequential streams of trajectories, hence resulting in considerable speedup over CPU-based approaches. This also holds even when compared to CPU-based setups where the pre-processing of trajectories into inverted indices is performed beforehand. To the best of our knowledge, this work is the first detailing FPGA support for variable-enhanced flexible pattern queries.

1.5 Integer Decompression

The goal of data compression techniques is to reduce the storage space and/or increase the effective throughput from the data source (such as a storage medium). Other critical performance factors considered include code complexity and memory offloading requirements. Various compression techniques can be combined and are tailored to perform best within certain classes of applications, where assumptions on the data (format, range, occurrence, etc) hold. Examples of the latter are Run-Length Encoding (RLE) [23] as used by image compression (JPEG), and Lempel-Ziv-Welsch [84] (LZW) for text data.

Compression techniques can be mainly categorized as being lossy or lossless. Generally, lossy techniques result in a higher compression ratio, and/or a faster processing (compression/decompression) time. Lossy techniques are hence preferred when the original data does not have to be exactly retrieved from the compressed data, and differences with the original data are tolerable or non-noticeable (such is the case with audio, video, etc).

Moreover, compression techniques can be further classified as being bit-wise or byte-wise. Byte-wise (or byte-aligned, byte-granularity) approaches typically result in a lower compression ratio due to the coarser granularity, but offer a considerably higher compression/decompression throughput.

This dissertation focuses on the decompression of integers compressed using the lossless bit-wise Golomb-Rice [23, 83] (GR) entropy method. GR compression is designed to achieve high compression ratios on input streams with small integer ranges [51]; it is deployed in several applications, such as image compression [51, 36, 41, 79, 34, 42], audio compression [44, 22], as well as the compression of streams of inverted indices

[48, 89, 90, 72], and ECG signals [7, 12, 50]. Inverted indexes require very fast processing, and operate under low timing budgets as they are utilized in the querying of high-volume data, as in (web) search engines [93]; however, even though GR offers high compression ratios, other approaches are preferred due to the gap in decompression performance [90]. Similarly, with the augmented resolution standards on video processing and displays (Full-HD, Quad Full-HD), faster decompression is a must. Finally, the complex processing of the increasing amounts of ECG data can be further reduced using high-performance decoders, with decompression being a first step once data is received. In all of the aforementioned applications, inefficient decompression limits the input throughput to the computational pipelines.

We present a novel highly-parallel hardware core capable of decompressing streams of GR-coded integers at wire speed with constant throughput, operating on *raw unmodified* GR data. To the best of our knowledge, hardware and software (CPU-based) GR decoders assuming unmodified GR data operate bit-serially on the compressed stream, which highly bounds the achievable decompression speeds. On the other hand, modifications to the algorithm and assumptions on the compressed format allow the application of efficient optimizations [36, 90, 41], though the limiting assumptions cannot be generalized . The proposed no-stall hardware solution is shown to outperform state-of-the-art software and hardware approaches, and achieves up to 7.8 Gbps sustained decompression throughput while occupying 10% of the available resources on a Xilinx Virtex 6 LX240T, a mid- to low-size FPGA. This work, as described in Chapter 5, appears in [58] © 2013 IEEE.

Chapter 2

XML Path Filtering

2.1 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 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 will 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 [88]. 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. Note that the terms "profile" and "query" are used interchangeably.

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 the 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 [2, 16, 27, 40]. These memory-bound solutions, however, suffer from the Von Neumann bottleneck and are unable to handle large volume of input streams. On the other hand, FPGAs have been shown to be particularly suited for the stream processing of large amounts of data and do not suffer from the memory offloading problem faced by software implementations. Furthermore, GPUs as co-processors are also a favorable option for applications requiring massively parallel computations [32, 4, 35, 45], such that sequential computations are run on the CPU while the computationally-intensive part is accelerated by the highly parallel GPU architecture.

This dissertation examines how to exploit the parallelism found in XPath filtering. Using an incoming XML stream, parsing and matching with thousands of user profiles are performed simultaneously by matching engines. We show the benefits and tradeoffs 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, the proposed 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.

A novel stack-based dynamic programming filtering algorithm targeting path queries is presented in this chapter, as applied to FPGAs and GPUs, and whose throughput is independent of the complexity of the user queries or the characteristics of the input XML stream. 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.

2.2 Problem Definition

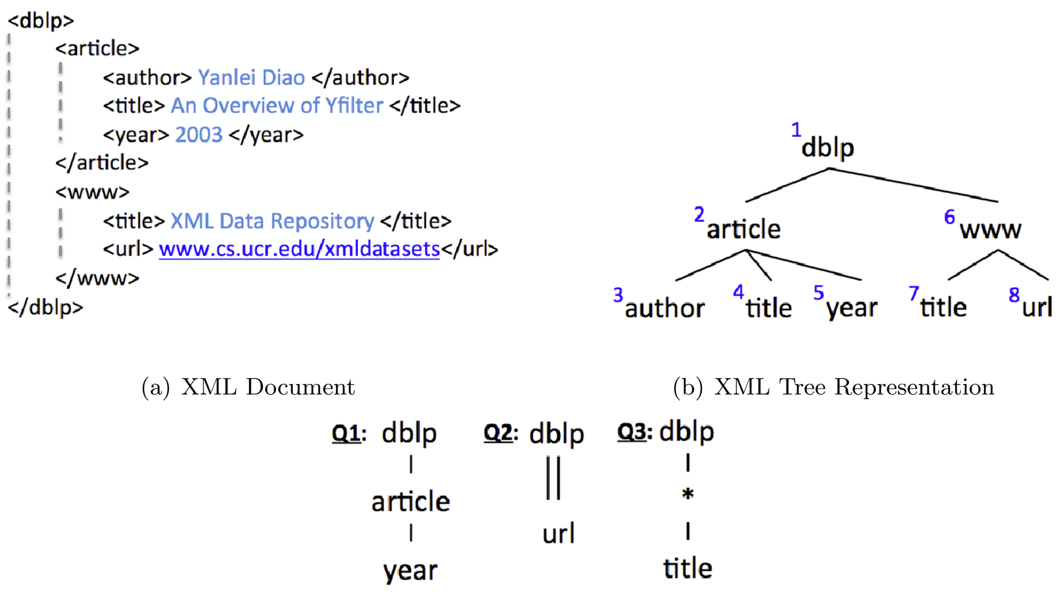
XML filtering is the core problem in a pub-sub system. Formally, given a collection of user profiles and a stream of XML documents, the objective of the filtering algorithm is to determine, for each document D, the set of profiles that have at least one match in D.

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 a ‘>’. Nodes in an XML document begin with a ‘start-tag’ (for example <author>) and end with a corresponding ‘end-tag’ (for example </author>). Figure 2.2(a) shows a small XML document example, while Figure 2.2 (b) shows the XML document’s tree representation. In this manuscript, we shall use the terms ‘tag’ and ‘node’ interchangeably. For simplicity, Figure 2.2(b) shows the tags/nodes (i.e. the structural relationship between nodes) in the XML document of Figure 2.2(a), but not the content (values). The values can be thought as special leaf nodes in the tree (not shown).

XPath [88] is a popular language for querying and selecting parts of an XML document. Here, 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 below:

$$\begin{array}{lll} Path & := & Step \mid Path\ \ Step \\ Step & := & Axis\ NodeTest \\ Axis & := & `/' \mid `//' \\ NodeTest & := & name \mid `*' \end{array}$$

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 ‘//’).



```
<dblp>
    <article>
        <author> Yanlei Diao </author>
        <title> An Overview of Yfilter </title>
        <year> 2003 </year>
    </article>
    <www>
        <title> XML Data Repository </title>
        <url> www.cs.ucr.edu/xmldatasets</url>
    </www>
</dblp>
```

(a) XML Document (b) XML Tree Representation

(c) Path Queries

Figure 2.1: Example XML Document in (a) textual and (b) tree representations. Sample XML Path Queries are displayed (c), querying the XML document.

Example path queries are shown in Figure 2.2 (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 2.2(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 2.2(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 [70], 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.

2.3 Related Work

2.3.1 Software Approaches to XML Filtering

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

Finite State Machine(FSM)-based approaches use a single or multiple machines to represent the user profiles [3, 16, 27, 31, 55]. An early work, XFilter [3], 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 [16] built upon the work of XFilter and proposed a Non-Deterministic Finite Automata (NFA) representation of user profiles (path expressions) which combines all profiles into a single machine, thus reducing the number of states needed to represent the set of user profiles. Whereas YFilter exploits prefix commonalities, the BUFF system builds the FSM in a bottom-up fashion to take advantage of suffix commonalities in profiles [55].

Several other FSM-based approaches were introduced that use different types of state-machines [27, 28, 64, 46]. In [27], the authors proposed a lazy Deterministic Finite Automata (DFA) which has a constant throughput with respect to the size of the query workload; however, lazy-DFA may suffer from state explosion depending on the number of nodes and level of recursion in the XML document, and the maximum depth of the XPath expressions. XPush [28] builds a single deterministic push down automaton using a lazy approach, while [46] builds a transducer, which employs a DFA with a set of buffers, and [64] employs a hierarchical organization of push down transducers with buffers.

Sequence-based approaches as in [40, 69] transform the XML document and user profiles into sequences and employ subsequence matching to determine which profiles have a match in the XML sequence. FiST [40] 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 [11, 9, 24]. XTrie [11] uses a trie-based data structure to index common sub-strings of XPath profiles, but it only supports the /child:: axis. AFilter [9] exploits both prefix and suffix commonalities in the set of XPath profiles. More recently, Gou and Chirkova [24] have proposed two stack-based stream-querying (and filtering) algorithms, LQ and EQ, which are based on lazy strategy and eager strategy, respectively.

2.3.2 Hardware-Accelerated Approaches to XML Processing

Previous works [15, 19, 47] 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 [19]. 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.

The work in [47] 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 [27]. 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 [53], 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 dissertation.

2.4 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 sub-path of length *J-1* has moreover matched. Below, 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 (paths of length 2, or simply pairs), and then extend it to general paths.

2.4.1 Pairs Matching

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

2.4.1.1 Parent/Child Relationships

Consider first using Finite State Machines (FSMs) as the base structure for the matching of XPaths [53]. Matching the pair $\{a/b\}$ is achieved through three states: as the XML document is streamed, the first state assumes the task of monitoring for an *open(a)* event; upon satisfying that condition, the second state is activated, until encountering a *close(a)* event. As long as the second state is activated, an *open(b)* event will result in activating the third state, indicating that the query matched in the document.

Such state machines fail however to successfully detect pairs in recursion-enabled XML documents, where nested tags are allowed. We overcome this limitation by introducing stacks.

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 of '*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 below, where a '1' indicates a match. A separate state for each query node at every tree level, rather than a single state per query, enables recursion (nesting) support.

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 2.2 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 tag of the {*a,b*} pair. Note that support for recursion is depicted under

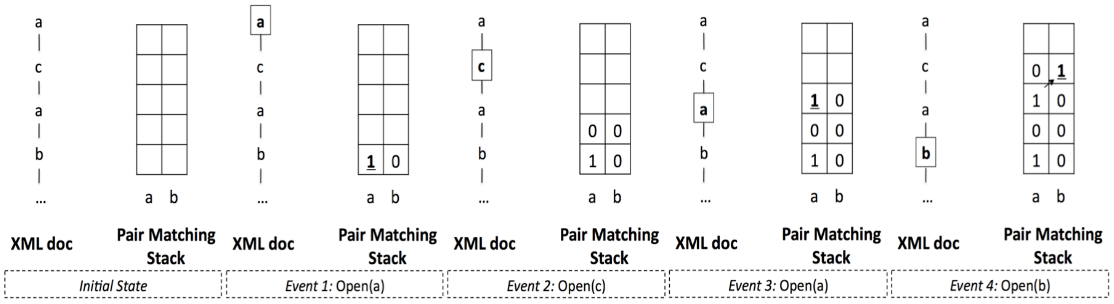


Figure 2.2: Overview of the matching of pair $\{a/b\}$. Each step refers to an *open(tag)* or *close(tag)* event, relative to the highlighted tag. A '1' in the '*b*' column indicates a match.

event 3, where each occurrence of the '*a*' tag in the XML document has a corresponding state (row) in the stack.

2.4.1.2 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, any *open(d)* event should result in a '1' being written to the second column. 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 stack at both columns can be updated simultaneously. Figure 2.3 shows the event-by-event matching of the pair $\{c//d\}$ in a sample XML document.

Figure 2.4(a) shows the tree representation of a sample XML document as it is processed. Figures (b) - (g) show several (two-node) queries and the top of the stack filtering mechanisms as it is being updated upon an *open 'b'* event. Using Figure 2.4, we visit some properties of path queries as applied to the stacks:

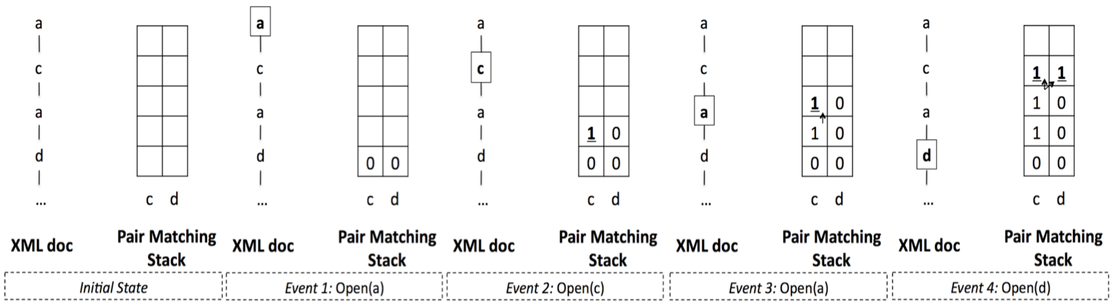


Figure 2.3: Overview of the matching of pair $\{c//d\}$. Each step refers to an *open(tag)* or *close(tag)* event, relative to the highlighted tag. A '1' in the '*d*' column indicates a match.

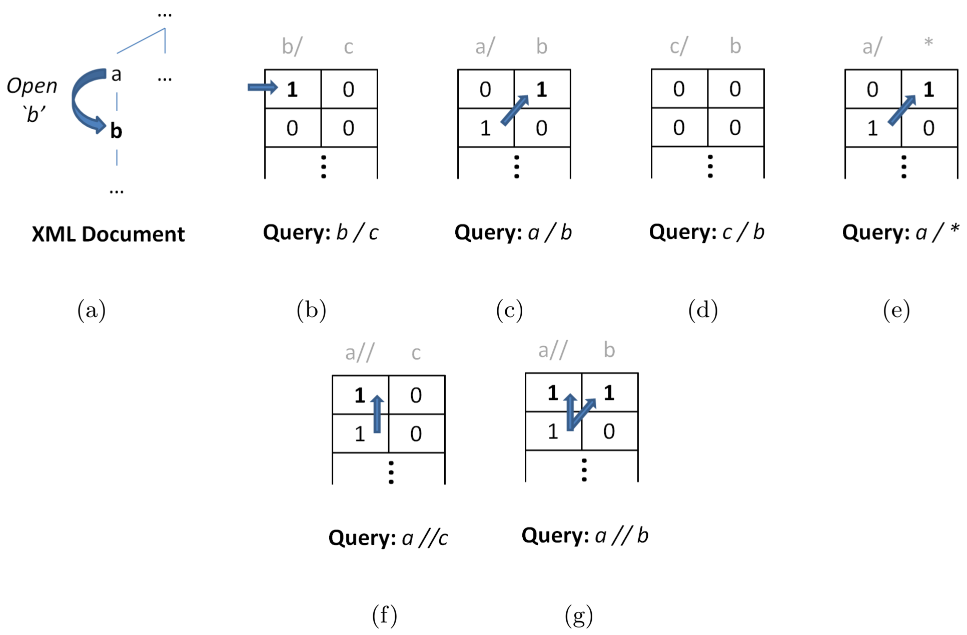


Figure 2.4: Sample XML document event (*Open 'b'*) shown in (a) alongside corresponding query stack updates at the respective top of the stacks (b) - (g). Query nodes and relations are portrayed in gray above the column they respectively map to.

- **path root nodes:** as these do not depend on a previous node, a ‘1’ is pushed onto their respective column of the stack (the first column) upon encountering a node with similar tag as part of the XML document (left column of Figure 2.4(b), vs. no effect in 2.4(d)).

- **parent-child relations:** these result in a *diagonally upwards* propagation of a ‘1’. In Figure 2.4(c), a ‘1’ previously stored in the root column propagates upwards to the right column upon a push, due to an *open ‘b’* event. Note that an open ‘b’ event does not suffice to store a ‘1’ in a ‘b’ column (see of Figure 2.4(d)).

- **ancestor-descendant relations:** these result in a *vertically upwards* propagation of a ‘1’ in columns to which query nodes followed by ‘//’ are mapped. Once a ‘1’ is pushed onto the column based on the two previously listed properties, and as long as it has not been popped, the ‘1’ is always pushed onto higher following tops of stack as the document is processed (see Figure 2.4(f)).

- **path leaf nodes:** A ‘1’ stored in the column to which a query leaf node is mapped, implies a successfully matched query. This occurs through a ‘1’ propagating upwards from the root column, through all middle columns, to the leaf. Figures 2.4(c), (e) and (g) exhibit matched query states.

- **wildcard nodes:** these imply a level of freedom where any document node is valid for a propagating ‘1’. See Figure 2.4(e).

Paths of any depth can be mapped to stacks, where each two columns are related using the above properties, as is detailed next.

2.4.2 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 2.4.1 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 equation applied to each cell $C_{top,j}$ on a push event, is given by:

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

where:

- *top* is the index of the new top of stack.
- $1 \leq top \leq$ maximum XML document depth.
- $1 \leq j \leq$ number of Path nodes.

Figure 2.5 shows an event-by-event overview of all the steps required for the matching of the XPath $\{a/c/a/c/b\}$.

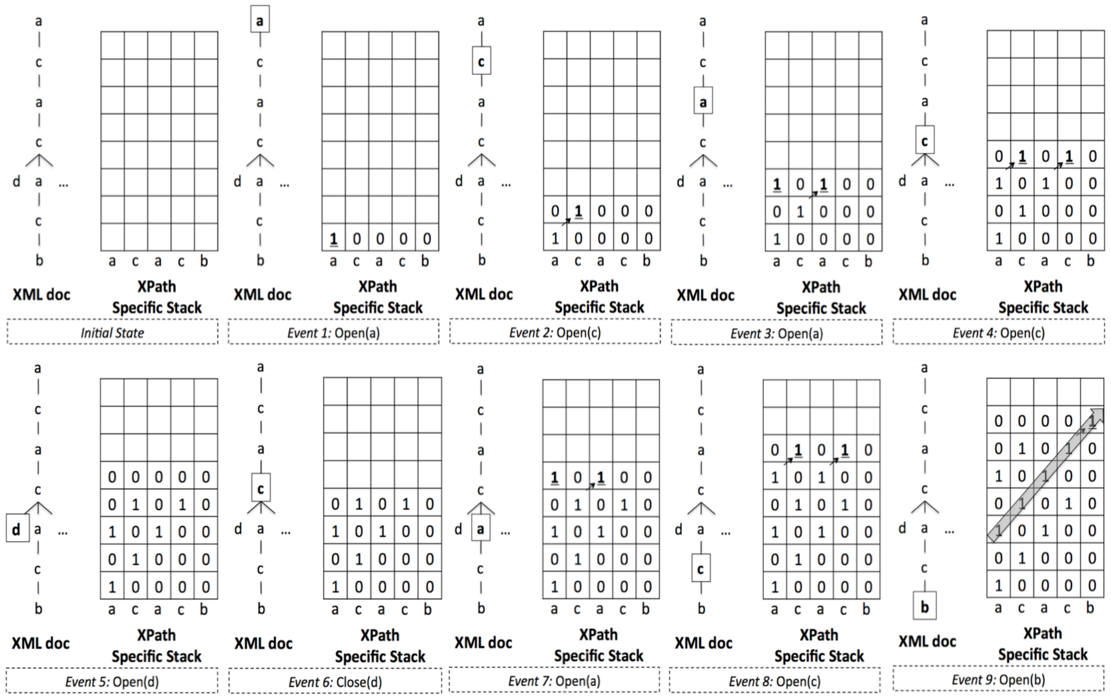


Figure 2.5: Overview of the matching of XPath $\{a/c/a/c/b\}$. Each step refers to an *open(tag)* or *close(tag)* event, relative to the highlighted tag. A '1' in the right-most column indicates a match.

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 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 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.

2.4.3 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 on 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 indicating matches to all descendants.

2.4.4 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** - where 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** - where 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 2.5 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.

2.4.5 Support for Predicate Expression Evaluation

The above 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 inter-column 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.

The work in this dissertation 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.

2.5 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 2.4 are explored; the first targeting setups where the query lifetime is considerably longer than that of the streamed XML document (Section 2.5.2); the second implementation targets queries that are updated regularly (Section 2.5.3). 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.

2.5.1 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 (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

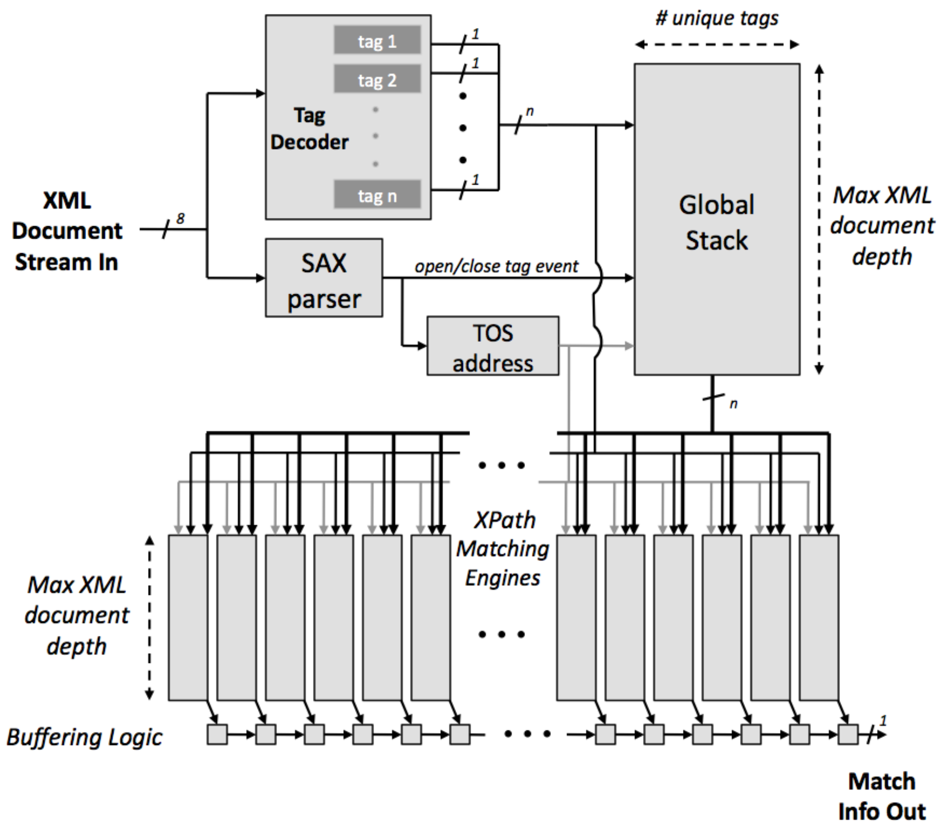


Figure 2.6: High-level FPGA-based system overview.

stream is complete, and all match states can then be reported. Figure 2.6 illustrates a high-level view of the system architecture.

Parsing is achieved using a hardware implementation of the *Simple API for XML* (SAX) Parser [70]. The SAX parser is an event-driven XML parser, ideal for streaming applications. Unlike other parsers (such as DOM [17]), 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.

With FPGAs being limited in hardware resources, a tag decoder is a desirable feature operating in conjunction with the SAX Parser. Since all query matching engines would need comparisons against respective tags, all engines executing in a parallel fashion, many redundant comparisons would take place across several engines, thus unnecessarily wasting resources. Decoders solve this issue by centralizing comparisons, and mapping decoded tags into single bit lines. All remaining comparisons are then translated into simple *AND* gates, hence, resulting in considerable savings in FPGA resources, since tags are shared across several queries. Our tag decoder is inspired from character decoding, the latter becoming conventional in pattern matching on FPGAs [13], [53], and which was shown to offer up to 83% of area savings in [53]. 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.

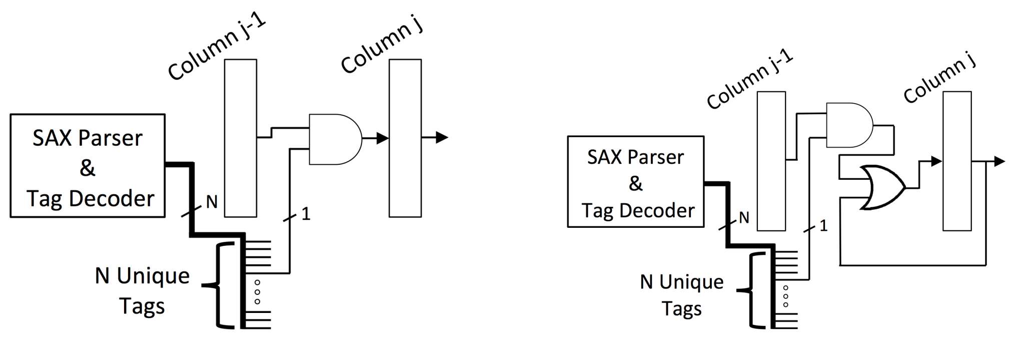
Figure 2.6 shows how a tag decoder would operate in parallel with a SAX Parser in order to generate *open* and *close* tag events, with a tag being a single bit line out of the possible n decoded ones. Note that only one of those bit lines is high at a given event, and all lines are cleared otherwise.

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 2.6, 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 2.5.2.2. The global stack is mapped to on-chip Block RAMs (BRAMs) [8], highly configurable hard-wired memory blocks that are embedded in most Xilinx FPGAs.

Finally, with up to tens of thousands of matching engines co-existing on chip, reporting matches becomes a more complicated issue, where mapping each match signal exclusively to an FPGA pin is not an option. Our previous approach [53] 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 2.6); 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



(a) (b)

Figure 2.7: Hardware logic connecting two PSS columns with the j^{th} and $j+1^{th}$ 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.

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.

2.5.2 Fully Customized FPGA Hardware

In this section, we describe the low-level implementation details of the Path Specific Stacks (PSS), the matching engines as described in Section 2.4.

2.5.2.1 Matching XPaths Using Path Specific Stacks

Stacks are implemented using Distributed Memory blocks, 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.

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 2.7(a) and 2.7(b). In case of a parent/child relation (Figure 2.7(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 $j-1^{th}$ column.

On the other hand, in case of an ancestor/descendant relation (Figure 2.7(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 $j-1^{th}$ 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 2.7(a) and 2.7(b). In the case of a parent/child relation with a wildcard as child (Figure 2.7(a)), the output of the $j-1^{th}$ column is connected to the input of j^{th} column, with no extra logic in between. In the case of an ancestor/descendant relation with a wildcard as descendant (Figure 2.7(b)), the output of the $j-1^{th}$ column is connected to the *OR* gate preceding the j^{th} column.

2.5.2.2 Applied Optimizations for PSS Reduced Resource Utilization

As described in Section 2.5.2.1, 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 2.5.

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. Instead, all that is of interest is whether the last predicate was matched throughout the document, reflecting if the query pattern appears in the streamed XML document.

The remaining two optimizations make use of the Global Stack, a structure shared by all matching engines (hence global), first introduced in Section 2.5.1 and Figure 2.6. 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.

The second optimization relates to *removing the column respective to the query root node.*

Using the Global Stack, the match state of the root node of each query is reflected in 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, and the current decoded tag. For instance, assuming a tag set of {A,B,C}, a TOS value of '010' indicates that B is the parent of the current active XML tag. A query starting with {B/A/...} would depend on the second TOS bit (respective to B) for the matching of node A.

The benefits of this optimization are effective when several queries make use of the Global Stack; this shared centralized structure merges and minimizes the information needed by all query root matching logic.

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 2.7(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 nodes 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 **non-conflicting** if they cannot be in a matched state at the same XML level. Non-conflicting 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.

Algorithm 1 details the path query node to stack column mapping algorithm, with column compaction enabled. The basic rules state that nodes followed by '//', and wildcards, map to their own restricted columns. Moreover, nodes having similar tags are not allowed to map to the same column. This algorithm returns the required stack width for a given query, and the mappings of the nodes to stack columns.

Algorithm 1: Twig Node to Push Stack Column Mapping

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

Algorithm 2: Mapper(tag T)

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

We show in Figures 2.8(a) and 2.8(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 2.5.

Looking at Figure 2.8(a); matching 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 2.8(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 *non-conflicting.* 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 are merged (*OR*'ed).

The *AND* gate respective to the 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 2.8(a) requires 7 LUTs (4 for logic, 3 for stack columns), while the implementation in Figure 2.8(b) requires 5 LUTs (3 for logic, 2 for stack columns). These results were generated through Synplify Pro 2010-09 (synthesis) and Xilinx ISE 14 (PAR). Although LUT sharing did not take effect here,

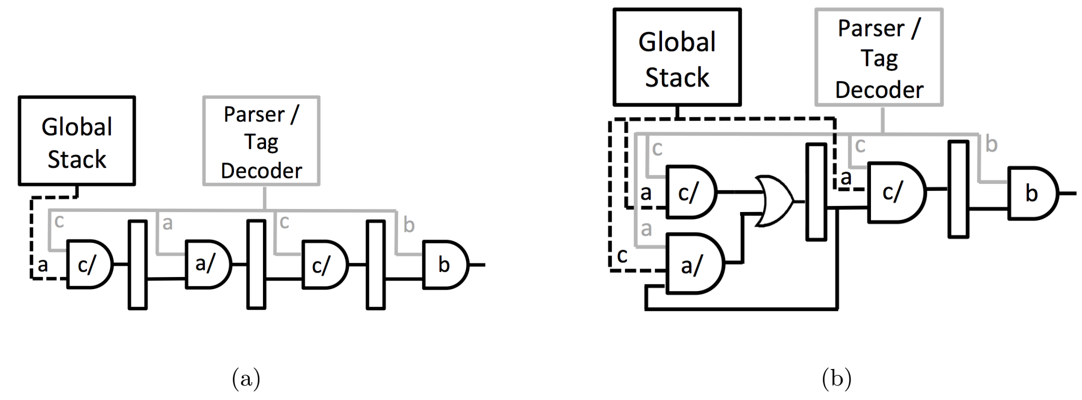


Figure 2.8: Hardware logic depicting the implementation of the filtering engine respective to query {a/c/a/c/b}, (a) without and (b) with the third stack optimization.

column optimizations would still result in area savings when LUT sharing is applied on XPath queries.

2.5.3 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 2.5.2, 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 bound 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.

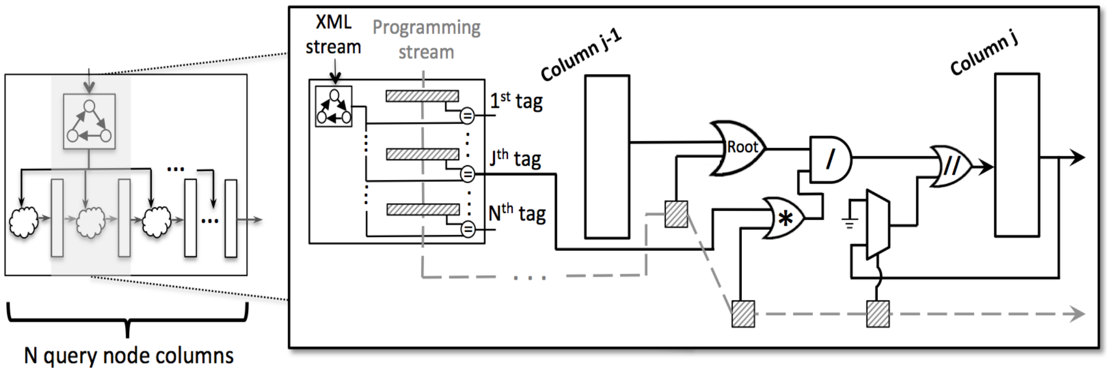


Figure 2.9: Programmable FPGA hardware overview, with emphasis on the column connections. The XML input passes through the parser and programmable tag decoder. Every decoded tag bit is connected to the logic preceding one column; every query node is mapped to a column. The connections between two columns can be programmed by writing to the (striped) Flip-Flops. A node can be a root node in the query (see **Root** above), can be a wildcards (see **'*'** above), can be followed by a parent/child relation or ancestor/descendant (see **'//'** above). The use of any of these is optional and node-dependent.

2.5.3.1 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 2.5.2.2, 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.

The programmable FPGA logic consists of a set of stack columns connected serially (see Figure 2.9). 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 2.9 shows the configurability of the connecting logic between two columns, and the support for:

- **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 left most 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 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 columns 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 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 in Chapter 3 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. Programmable twig query matching engines require 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. This discussion is however left out of this dissertation.

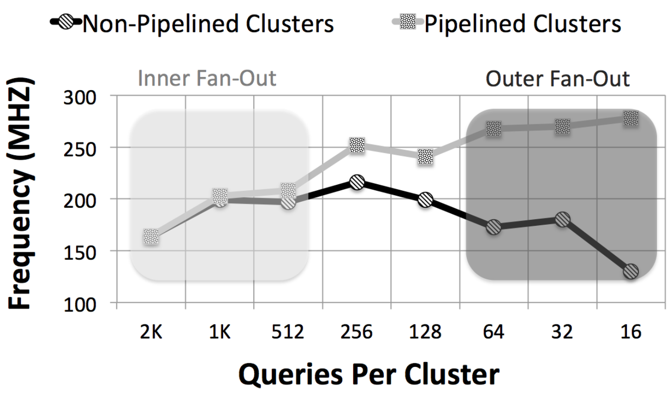


Figure 2.10: Design space exploration on 2K queries mapped to customized stacks on a Xilinx V6LX240T FPGA: as the cluster size is varied, the effect on performance is studied. The low frequency in the inner fan-out region is due to the many queries per cluster. Pipelining the input stream across clusters will fight over-clustering (effect visible through a maintained high frequency in the outer fan-out region).

2.5.4 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 (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 2.10 depicts a design space exploration to determine the adequate cluster size in order to achieve a good balance between fan-out within clusters (too many queries per cluster), and over-clustering (too few queries per cluster). Results are generate 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

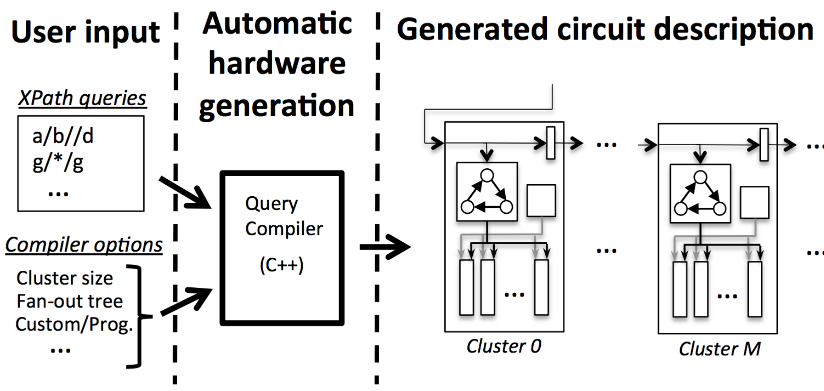


Figure 2.11: The process of generating a hardware circuit description consists of the user listing XPath queries, and setting compiler options as desired. Hardware is automatically generated using a developed query compiler.

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 over-clustering. Moreover, over-clustering increases resource utilization to replicate the parser and global stack, and it reduces the 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.

2.5.5 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 2.5.4), generating custom (Section 2.5.2) or programmable (Section 2.5.3) hardware, enabling column-level optimizations (Section 2.5.2.2), 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.

2.6 XPath Filtering on GPUs

With the filtering algorithm applied to an FPGA-based accelerator, the XML documents are directly streamed to the accelerator, and no external memories are required. Stacks are implemented using custom circuitry, where several optimizations are applied to reduce the width of stacks. Such a setup and optimizations are not directly applicable to GPUs. Here, the XML document has to be parsed before being sent to the GPU memory. Once the parsed XML document is in GPU memory, it is processed by software implementations of the PSS. Stacks are stored in shared memory, and we develop software kernels to implement different queries.

We investigate two mappings of the proposed parallel XPath filtering solution to GPUs. In the first, the query stack is holistically processed by a single *thread*. Whereas in the second, every *block* is associated with the task of updating a single column at a time. The most notable differences are as follows:

- There is no inter-thread communication (through the shared memory) in the first approach. This is not true in the second approach, as each column depends on the preceding one, and they are each processed by a different thread; here, upon every update, a look up in shared memory is required to read the value of at the top of the previous column.

- Commonalities across queries cannot be extracted when mapping each query to a thread (first approach). This will affect scalability, as optimizations can be applied to reduce the query set.

In the remainder of this section, we go over the implementation details of both approaches, and discuss applied optimizations.

2.6.1 Mapping Queries to Threads

As the XML document is processed, all columns in one path stack share the partial matching info stored in the top of stack. In our first approach, every query is processed holistically by a single thread. Even though no stack data is shared across queries/threads, the stack of every query in a block is mapped to the low-latency shared memory.

The kernel executed on every thread consists of a loop that computes, for every XML event, the update at every cell (of each column) in the top of stack. The GPU kernel executed per stack column is provided in Algorithm 3. Lines 4 through 14 are repeated per thread for all each columns in a stack (inner for-loop not shown). There is however one match state per query/thread (lines 17 through 19 are not needed when mapping queries to threads). Once all XML events are processed, the match state of the query is streamed back to the CPU.

With Streaming Processors (SPs) having hardware multithreading support, every SP can process at a time as many queries as the number of available hardware

Algorithm 3: GPU kernel per stack column

```
$current\_level \leftarrow 0$
$matched \leftarrow 0$
for all XML document events do
  if pop event then
    $current\_level$ - -
  else
    $current\_level$ ++
    if '1' propagates diagonally upwards $OR$
    vertically upwards then
      $stack[current\_column][current\_level] \leftarrow 1$
      $matched \leftarrow 1$
    else
      $stack[current\_column][current\_level] \leftarrow 0$
    end if
  end if
end for

if current column is a leaf column then
  $match\_state[column\_ID] \leftarrow matched$
end if
return
```

threads. However, the number of queries processed simultaneously in a block is bounded by the limitations in size of the shared memory; as every query makes use of a distinct stack, no more than a certain number of queries can be processed at once. That is true unless the stack is mapped to the global device memory. In that case, the stack limitations are lifted, but the performance at least halves.

2.6.2 Mapping Queries to Blocks

Another approach to query filtering on GPUs is to assign the processing of each stack column to a block, such that all the stack updates occur in a single Streaming Multiprocessor (SM), across threads. The stack is again mapped onto the shared memory.

2.6.2.1 Approach Overview

For every XML event, each thread updates the top of one column, and data is passed across threads through the shared memory. A synchronization event is required at each thread after every XML event, in order to have all the contents of the top of stack updated before proceeding to the next XML event.

Details of the GPU kernel executed by every thread, are provided in Algorithm 3. The main difference with the previous approach is that every thread will not loop for all query nodes (stack columns). Another difference is that a match state is kept for every query column, though only leaf columns report their match state to global memory (lines 17-19).

2.6.2.2 Common Prefix Optimization

Figure 2.12 depicts a sample set of queries to be filtered on the GPU. The stack column dependencies are shown for each query stack separately; these dependencies determine the propagating path (through columns) of a '1' within each stack. One observation is that Q0 and Q1 share the prefix *a/b/c/*, and thus the first three columns of their respective stacks behave identically. On the other hand, Q2 shares the prefix *a/b/* with both Q0 and Q1.

A tree representing all stacks merged at the common prefix nodes can be constructed, such that tree nodes represent stack columns and query nodes (see Figure 2.12). This tree will consist of the minimum number of stack columns required to match all queries. Queries Q0, Q1 and Q2 require at least 6 stack columns for filtering, in contrast to the initial 11 when not merged.

Nonetheless, this minimum might not be achieved in practice, since not all queries on the GPU can communicate through the localized shared memories; only queries evaluated in one block can share stack columns (unless the stacks are mapped to the high-latency global memory). Figure 2.12 depicts the mapping of query nodes with the applied optimization, and a block size of 5 (in practice, the block size can be up to several hundreds).

The fixed-sized block limitation implies that some of the minimal tree nodes will be mapped to more than a single block when the block is smaller than the tree; for instance, *a/b/* was computed in both blocks.

We look into the problem of mapping query nodes into blocks, while minimizing the number of required blocks. The minimal tree is taken as a reference, even though

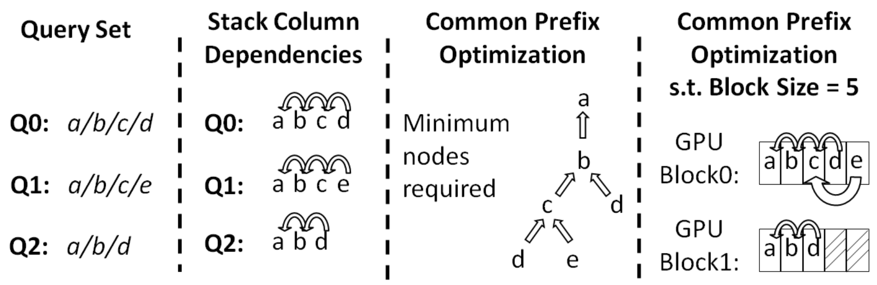


Figure 2.12: Sample query set shown initially, followed by the stack column dependencies for each query stack independently. A tree representation of all the queries as grouped by common prefixes is then portrayed. Finally, the query nodes are mapped to GPU blocks with a restriction on the block size.

this minimum cannot be achieved, as the assumed query set size is much larger than the maximum block size (tens of thousands of queries vs. block size at most 512).

The first approach employed executes the following steps:

1. Sort all queries (by tag ID)
2. Instantiate a new GPU block
    (a) Pop the query at the top of the sorted list
    (b) Find the block offset of the last common tag with the previously inserted query (if any)
    (c) Append the remaining query nodes to the block, linking the first suffix node to the node at the index as found in (b)
    (d) Repeat (a)-(c) until the popped query doesn't fit in the block
3. Repeat step 2 on the remaining queries

Nodes are linked using the 'previous column index' portion of the representation described in Figure 2.13(b). Stack columns of a single query are no longer contiguous in the shared memory, as columns are now shared across tags. The sorting step will insure that queries popped linearly will share prefix nodes.

We also looked into a variant of that approach, where, instead of linearly popping queries from the sorted list, we greedily pick the query sharing the longest common prefix with its neighbor, place it in the block, and fill the block with its neighboring queries. This approach runs much slower than the initial one (few seconds vs. tens of seconds), while providing minimal improvement. Therefore, for the remainder of this discussion, we focus on the initial placement technique.

One artifact of placing queries into blocks is fragmentation. Here, blocks exhibit empty slots where queries wouldn't otherwise fit. However, as will be shown later in our experimental evaluation and using the above approaches, the average wasted nodes per block was a little over a single node. Therefore, we deduce that the effect of fragmentation is minimal.

A larger block size naturally results in more commonalities exploited. However, the block size should be also set to maximize the occupancy on the GPU. Maximizing the occupancy can help to cover latency during global memory loads that are followed by a synchronization instruction. The occupancy is determined by the amount of shared memory and registers used by each thread block.

Note that the common prefix optimization cannot be applied to the first approach, where each SP holistically processes a query.

2.6.3 Streaming Processor Personalities

Every instance of the GPU kernel requires knowledge of the query node mapped to the stack column it processes. The CPU parses all queries and transmits to the GPU device memory a pre-processed form of all the queries as an array, where each entry represents a query node, and corresponds to a single kernel instance. Figure 2.13(b)

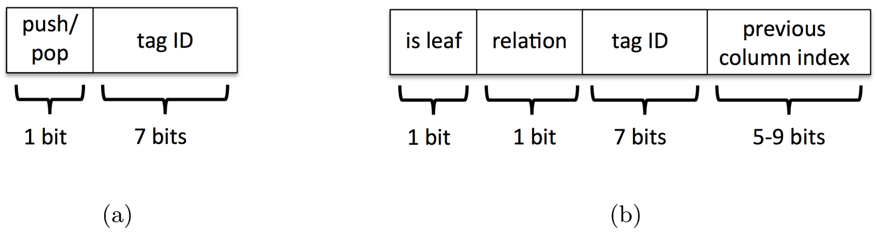


Figure 2.13: GPU Input format: (a) shows the storage format of every XML document event as streamed to the GPU, whereas (b) depicts the format of SP personalities, representing a query path node and its stack mapping.

represents a generic view of a query node representation, namely thread *personality*, on which the kernel instance functionality depends.

A personality consists of a single bit to indicate whether the node is a query leaf, one bit to indicate the relationship (parent, ancestor) between the node and the following one in the query, and a 7-bit representation of the tag ID. The previous column index entry refers to the index of the previous stack column in the block; 5-9 bits are required since GPU blocks can hold 32 to 512 kernel instances. We represent fields using the minimum number of bits and compact them into the smallest data types. This is in contrast to using one distinct variable per field of the personality. The 14-18 bit personality will be implemented as a 32-bit number, in contrast to several chars and integers. This would potentially reduce the storage requirement and memory transfers by half.

2.6.4 Efficient Use of the GPU Memory Hierarchy

The proposed GPU-based filtering algorithm only depends on the events and tags of XML documents, rather than content. In order to minimize transfer from CPU to GPU and utilized memory space, the CPU would compress the XML document events into an optimized representation that is then transferred to the GPU. Each compressed entry is 8-bit wide as shown in Figure 2.13(a), with one reserved bit indicating the event type (push/pop), and the remaining seven bits representing the corresponding tag ID. Every XML document node is translated into such an entry.

The pre-processed XML is transferred to the global device memory of the GPU, in order to be read by all threads. Since the XML document is read-only, small documents could fit into the cached constant memory. However, our experiments show that minimal speedup was achieved with such small documents mapped to the constant cache. We thus only utilize the global memory to map XML streams of all sizes.

Thread personalities are also transferred to the global memory of the GPU. However, since every thread will read its personality once, this is done initially, and the personality is stored in the thread registers.

As noted earlier, stacks are stored in the low-latency shared memory of every block. The maximum stack depth is set at compile time to allocate the required amount of shared memory.

With the completion of processing on the GPU, the match state of every query is written once to the global memory and streamed back to the CPU.

2.6.5 Supporting Batches of XML Documents

The previous discussion considered handling a single document. In real life pub-sub applications, we have witnessed that the incoming stream consists of batches of small documents. Supporting such batches represents an additional challenge for GPUs because of the decoupled nature of the CPU and GPU. The parsing of every XML document has to be performed on the CPU; the latter parses all the XML documents, and then streams to the GPU a processed representation of that document; the GPU

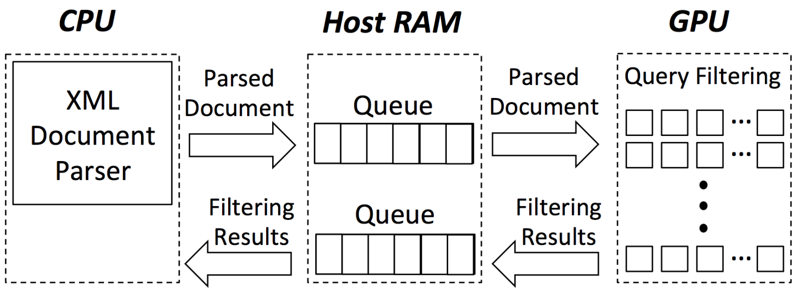


Figure 2.14: GPU filtering framework: a CPU thread parses every XML document and stores it in a format optimized for the GPU-based XML filter; a parsed XML document is stored in a queue on the host (CPU RAM); every parsed document is copied over to the GPU memory where query filtering is performed, and results are streamed back to the host (CPU) RAM.

then performs filtering for all user profiles on the received document. Once filtering is completed, filtering results are streamed back to the CPU which would be idle otherwise.

Figure 2.14 depicts the GPU filtering framework we deploy to handle document batches: a CPU thread parses every XML document and stores it in a format optimized for the GPU-based XML filter; a parsed XML document is stored in a queue on the host (CPU RAM); every parsed document is copied over to the GPU memory (not shown); query filtering is performed, and results are streamed back to the host (CPU) RAM.

This cycle is repeated for every XML document. Using this scheme, parsing and filtering are performed in parallel; however, filtering can only start after the document is parsed (and then copied to the GPU memory).

2.7 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 [16] and FiST [40].

For the performance experiments, we utilize the DBLP DTD provided by [80] to generate XML documents and user profiles. XML documents of maximum depth 16 and varying sizes were generated using the ToXGENE XML Generator [6]. 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 1100 MB; 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 [16].

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) 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. Here, a filtering setup with higher throughput is one less likely to drop packets, and able to process documents faster.

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 would reside on the CPU RAM. Since XML parsing is orthogonal to this work, parsing time is not included in performance measurements (though parsing is carried out on the FPGA, with no impact on throughput).

2.7.1 Experimental Evaluation of FPGA-Based Approaches

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

2.7.1.1 Setup and Platform

Our FPGA platform consists of a Pico M-501 board connected to an Intel Xeon processor via 8 lanes of PCI-e Gen. 2 [66]. 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 filtering engines communicate 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 our filtering engine is independent of the global clock. The PCIe interfaces incur an overhead of ≈8% of available FPGA resources.

The RAM on the FPGA board is not residing 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 (note that stacks are built using the FPGA logic).

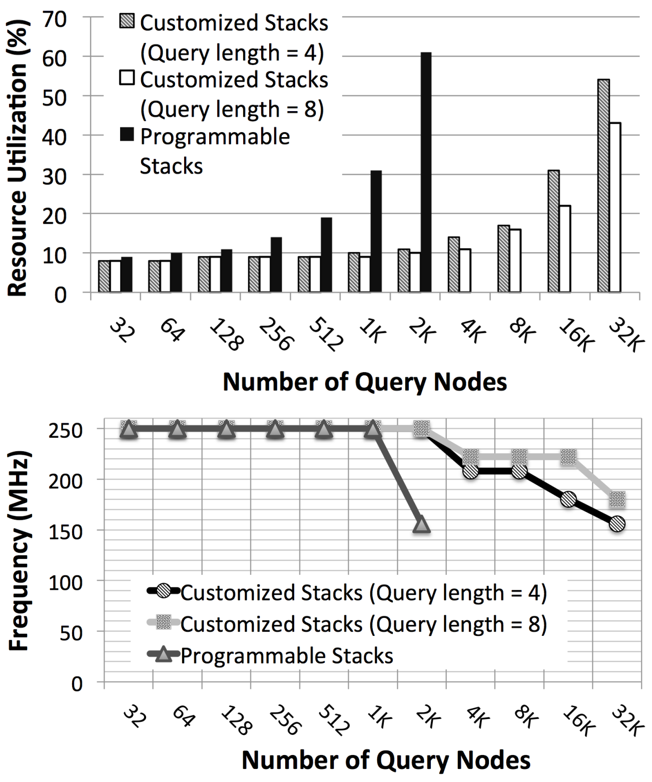


Figure 2.15: Resource utilization (a), and operational frequency (MHz) (b), of FPGA-based XML filtering approaches on a Xilinx Virtex 6 LX240T FPGA, as part of a PICO M-501 platform. Lower frequencies are due to the larger filtering circuits.

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.

2.7.1.2 Tradeoffs and Resource Utilization

Resource utilization is shown in Figure 2.15 (a), corresponding to the three implementations of the filtering algorithm on FPGAs, namely:

1. **Customized (Query length = 4)**: An implementation of the custom hardware approach described in Section 2.5.2, with PSS optimizations on, clusters of size 256 queries.

2. **Customized (Query length = 8)**: An implementation of the custom hardware approach described in Section 2.5.2, with PSS optimizations on, clusters of size 128 queries.

3. **Programmable**: An implementation of the programmable hardware approach described in Section 2.5.3, with a cluster size of 128 columns.

Note that the programmable implementation makes use of smaller clusters (size 128) than the customized counterpart. This smaller cluster size is preferable, since programmable clusters are bigger (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 2.15 (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).

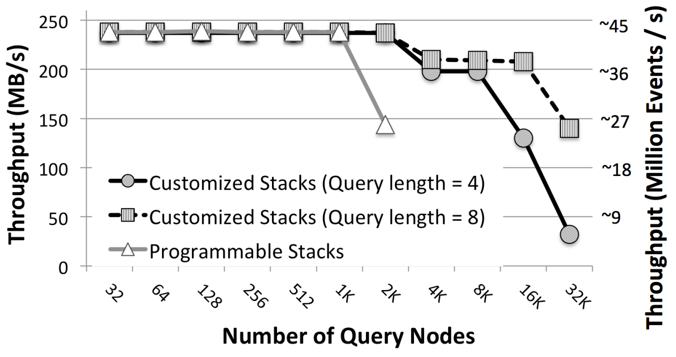
As expected, the custom hardware benefits from the reduced resource utilization, and that is not solely due to the PSS optimizations. Custom hardware uses on average 7 times less resources than a programmable approach (up to 12 times less). Note that we can further optimize the custom circuitry by making use of the common prefix optimization. This optimization can be combined with the stack optimizations presented in Section 2.5.2.2. The expected reduction in query nodes would be as studied in Section 2.7.2. We omit further exploration of this option here for brevity.

Doubling the query length requires on average 2 times more resources with the programmable implementations, and that is due to the doubling of the stack size (number of columns), and the resulting need for inter-column logic. Conversely, doubling the query length will 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 non-linear behavior in resource utilization while doubling the number of queries, is due to the heuristic-based nature of the tools. Moreover, in 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.

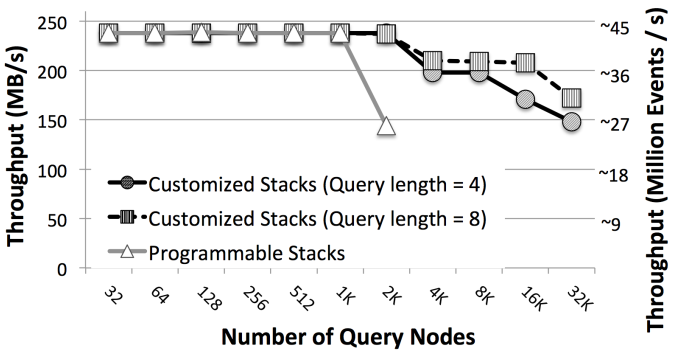
Though custom hardware approaches utilize considerably less resources than their programmable counterpart, this comes at the cost of *high reconfiguration time*, where updating queries in the custom hardware reconfiguration requires a new run through the synthesis, place and route tools, which could 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.

2.7.1.3 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.



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



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

Figure 2.16: 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 parser deployed on the FPGA is able to process a stream of up to 1 character (8 bits) per hardware cycle, generating events (push/pop) that are then forwarded to the stacks. The latter guarantee processing one event per cycle, as no memory external to the FPGA chip is used (also note that XML events are less frequent than document characters). As a result, the throughput of the filtering mechanisms is *deterministic*, 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.

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 2.15(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 2.16(a) and 2.16(b) depict the throughput of FPGA-based XML filtering approaches for a batch of 5K small ($\approx$220KB), and 500 medium ($\approx$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, 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 2.16(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 [19]. 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 dissertation. Using optimized parsers would help fight the lower operational frequency of certain circuits (Figure 2.15(b), >16K customized and 2K programmable nodes), and would help filter at higher bandwidth links when needed (10G ethernet).

2.7.2 Experimental Evaluation 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.

The remaining discussion in this section relates to the evaluation of the (node reduction) efficiency of the common prefix optimization, a study on the effect of the GPU block size, comparing the mapping of queries to GPU threads versus blocks, and end-to-end GPU throughput measurements.

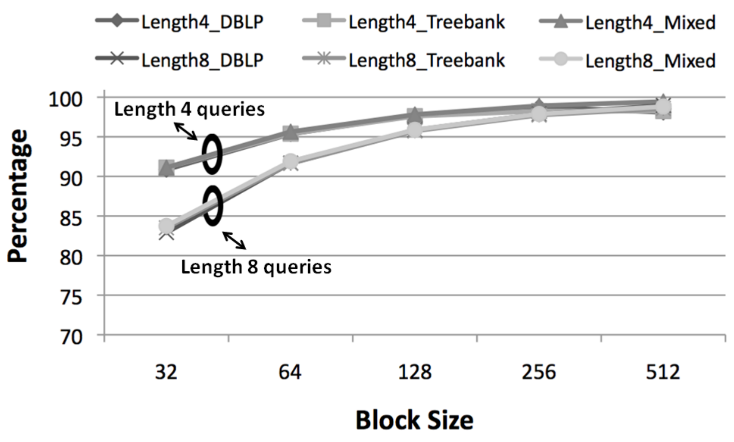


Figure 2.17: Percentage of reduced nodes vs. minimal tree, resulting from the common prefix optimization, with varying block sizes, on 32,000 queries. The mixed queries are generated from a set of four DTD's.

2.7.2.1 Common Prefix Optimization Evaluation

In this section, we evaluate the common prefix optimization, while mapping queries to blocks. Distinct queries are generated using the YFilter query generator [16], based on the following DTD's:

- DBLP, representing bibliographic information and tends to have more bushy trees [80].
- Treebank, representing tagged English sentences and tends to have deep recursive subtrees [80].
- SwissProt, a curated protein sequence database providing a high level of annotations, a minimal level of redundancy and high level of integration with other databases [80].
- XMark, as part of the XML Benchmark Project [87].

Several DTD's are used such as to ensure that the efficiency of the common prefix optimization is not biased towards a given class of XML documents.

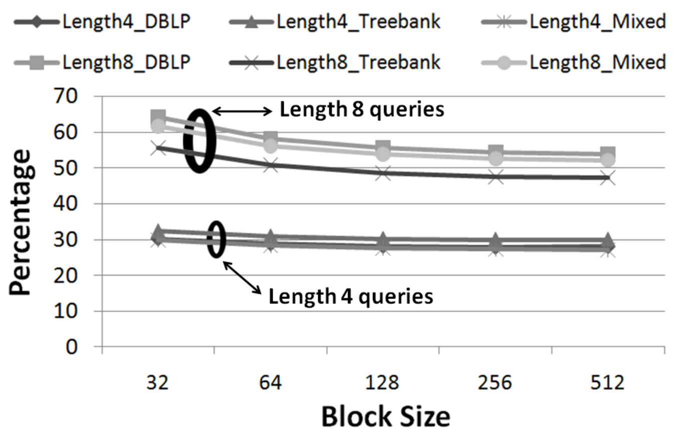


Figure 2.18: Percentage of resulting remaining nodes when applying the common prefix optimization, versus the original respective query sets (of size 32,000 queries each). The shown percentages includes empty GPU block nodes due to fragmentation.

We show in Figure 2.17 the percentage of reduced nodes compared to the minimal tree, resulting from the common prefix optimization, with varying block sizes. We perform our study on 32,000 queries of length 4 and length 8, while doubling the block size; queries are of class DBLP, Treebank and Mixed queries, the latter consisting of unrelated queries as generated by four DTD's (DBLP, Treebank, SwissProt, XMark). The purpose of the Mixed query set is to study the effect of the common prefix optimization on an un-biased set of queries exhibiting fewer overall commonalities.

As expected, larger block sizes provide more improvement, as there are fewer replicated nodes across blocks. Blocks of size 256 and 512 nodes result in an almost minimum number of nodes based on the minimum tree. On the other hand, length 4 and length 8 queries exhibit similar respective behavior, such that length 8 queries are always one step (block size) behind in terms of improvement; that is due to the fact that length 8 queries are likely to result in double the amount of query nodes.

Figure 2.18 shows the overall resulting remaining nodes from applying the common prefix optimization, as percentage of the original query set, with varying block sizes. The remaining nodes are nodes to be computed on the GPU, including the empty

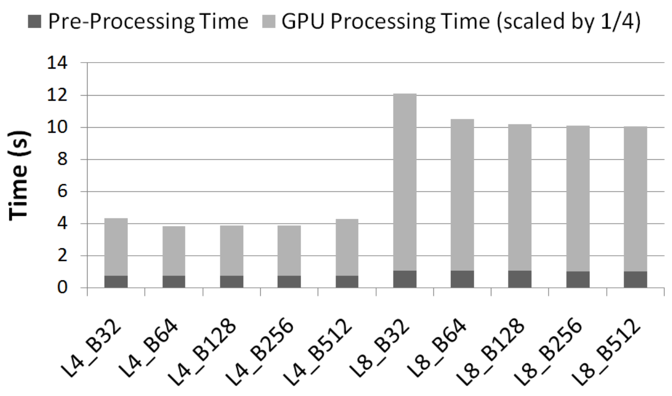


Figure 2.19: Total execution time filtering 32K queries over a 50MB DBLP XML document, highlighting pre-processing time on the CPU (with the common prefix optimization applied) , and GPU processing time (scaled by 1/4). The query set consists of length (L) 4 and 8 queries, while varying block sizes (B).

block nodes resulting from fragmentation. For all used query sets, length 4 queries result in the most reduction (mean of 71%). On the other hand, length 8 queries result in an average of 45% reduction, less than length 4 queries, due to the longer suffixes. This reduction will almost linearly affect the execution time, as discussed next.

2.7.2.2 Effect of the GPU Block Size

When applying the common prefix optimization, increased block sizes result in a further reduced query set, as seen earlier. However, due to the contention on the available computing resources and the limited amount of available shared memory per SM, larger block sizes can negatively affect performance when executing on the GPU.

We show in Figure 2.19 the total execution time required to filter 32K queries over a 50MB DBLP XML document, while highlighting pre-processing time on the CPU (with the common prefix optimization applied) , and the corresponding GPU processing time (scaled by 1/4 for presentation purposes). The query set consists of length 4 (L4) and length 8 (L8) queries, while varying the GPU block size (B32 . . . B512).

The pre-processing time depends solely on the query set, and requires an average of 750ms for queries of length 4, versus 1,040ms for length 8 queries, less than 4% on average of the overall execution time.

On one hand, while resource contention is less considerable, a block size of 32 results in the most processing time due to the least optimized query set. However, utilizing larger block sizes can negatively affect GPU processing time, even though the resulting query set is smaller. Block sizes of 64 and 256 result in the least processing time for queries of length 4 and 8, respectively.

For the remainder of this study, we set the block size to 128, as being a middle point, providing effective common prefix optimization (Figures 2.12, 2.18), and less resource contention than in larger block sizes.

2.7.2.3 Mapping Queries to GPU Threads versus Blocks

Figure 2.20 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 over-utilization of the GPU; more parallelism (queries) results in linearly more time (less speedup) due to the unavailability of processors.

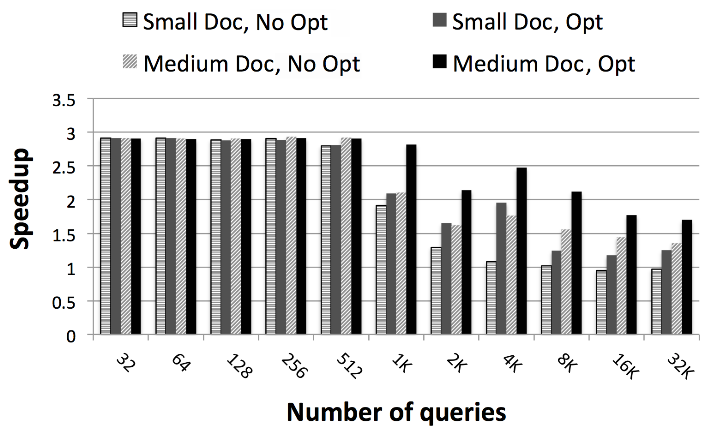


Figure 2.20: Speedup of mapping queries to GPU blocks versus mapping queries to GPU threads.

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 2.20, 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 less 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 study, we will only consider mapping queries to GPU blocks (one stack column per GPU kernel), with the common prefix optimization applied.

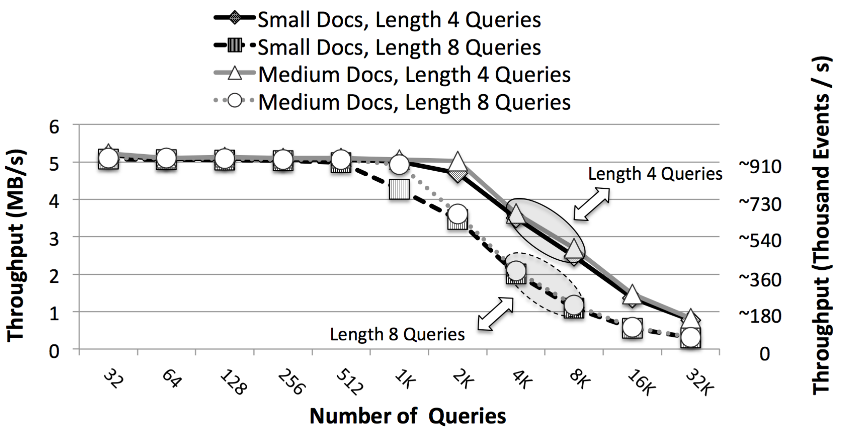


Figure 2.21: Throughput (MB/s) of the GPU-based approaches with queries mapped to blocks, for queries of length 4 and 8, with respect to a batch of 5K XML (small) documents of size 220KB each and 500 XML (medium) documents of size 2.2MB each.

2.7.2.4 Performance Evaluation

Figure 2.21 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 from 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 over-utilized. 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 a better performance prior to the breaking point. However, it would linearly help delaying the occurrence of the breaking point (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 2.21 (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

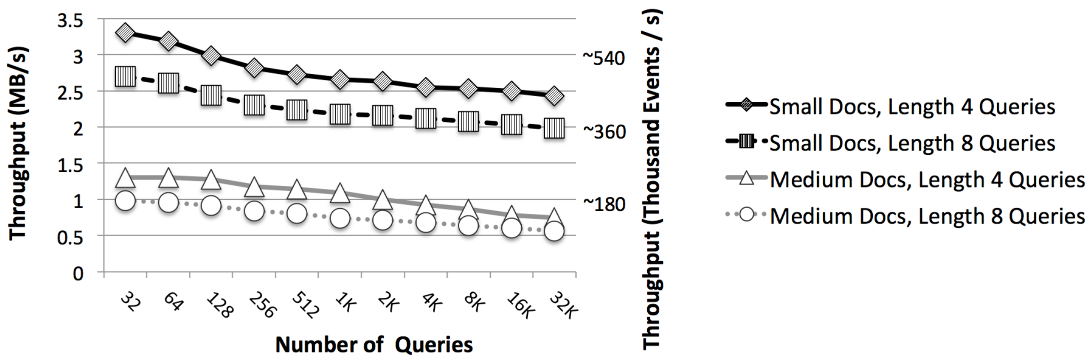


Figure 2.22: Throughput (MB/s) of the CPU-based approach (YFilter) for queries of length 4 and 8 with 15% probability of occurrence of '*' nodes and '//' relations, and varying documents sizes (namely 5K documents ≈220KB each, and 500 documents, ≈2.2MB each).

witnessed (average $< 1\%$). Increasing the percentage of occurrence has no effect, since '*' and '//' are supported by default for all stack columns.

2.7.3 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 (query length or percentage occurrence of '//' and '*') or XML document size is increased, incurring more than 7 GB 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 single CPU, as intended by design. These numbers give us a good understanding of the characteristics of the three used platforms (summarized in Table 1). We later show many (independent) instances of YFilter running on a multi-core, splitting the document set equally among all YFilter threads/instances (Figure 2.24). 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 2.22 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 2.22. 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.

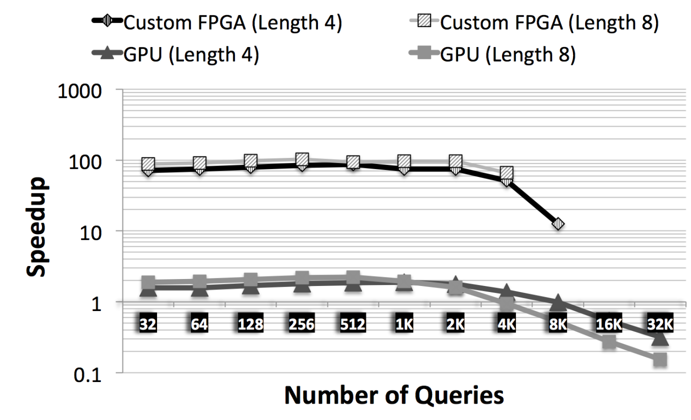
Finally, the effect on performance of varying percentages of '*' and '//' was studied by varying the percentage of occurrence of '*' and '//' from 5% to 25% (data is not shown for brevity). Increasing the percentage showed to constantly have a negative effect by deteriorating performance anywhere from 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.

2.7.4 Comparing FPGA, GPU and CPU-Based Filtering

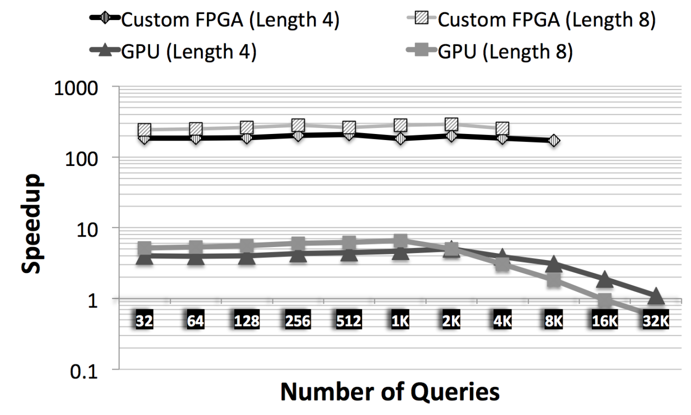
We proceed with the performance comparison of the customized FPGA circuit to the GPU-based filtering, and to a reference CPU-based approach (YFilter), while setting the percentage of occurrence of '*' and '//' to 15%.

Speedup (on a log scale) is shown in Figures 2.23(a), 2.23(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'.

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



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



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

Figure 2.23: 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'.

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 multi-cores 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 2.24 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

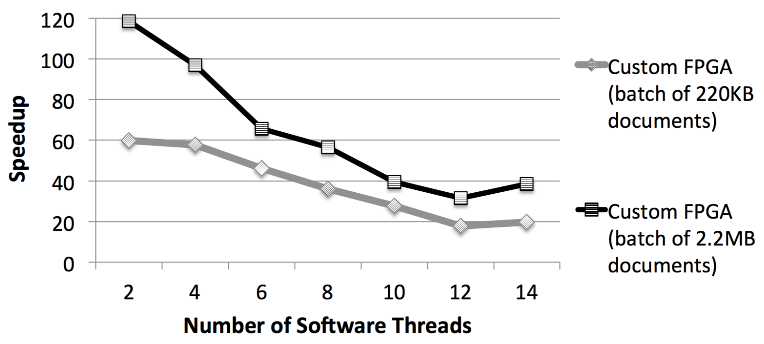


Figure 2.24: Speedup of the custom FPGA approach versus a multi-threaded version of YFilter, running on a 12-core machine. 2K queries of length 8 are assumed, with a 15% probability of occurrence of '*' and '//'. Batches of 5K 220KB documents and 500 2.2MB documents are used.

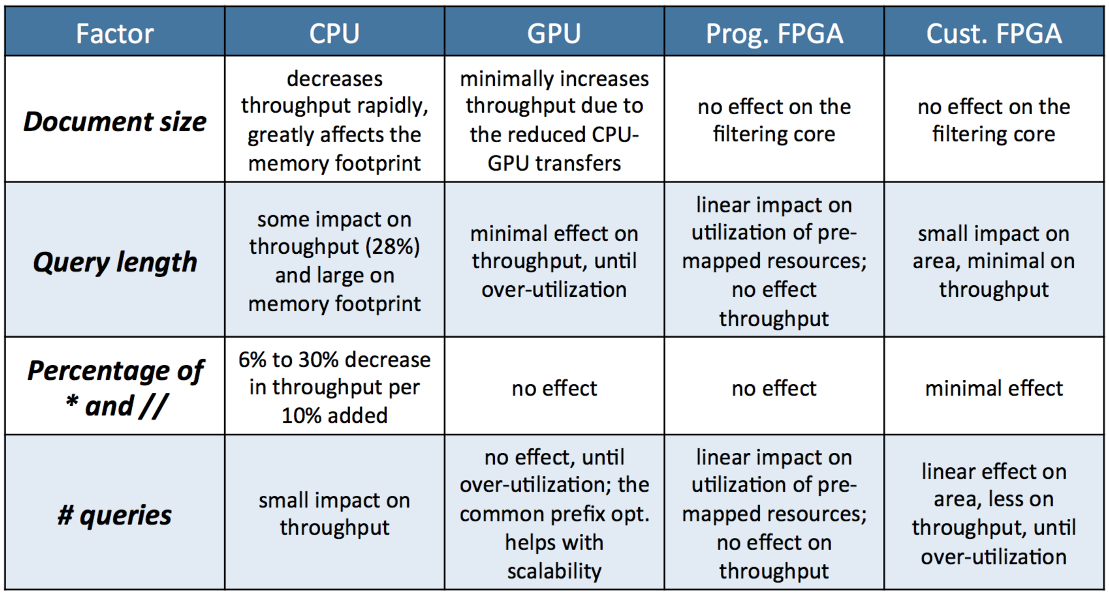
assumed, with a 15% probability of occurrence of '8' and '//'. Batches of 5K 220KB documents and 500 2.2MB documents are used.

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 Figure 2.25.

2.8 Conclusions

This work 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 providing 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-



Factor	CPU	GPU	Prog. FPGA	Cust. FPGA
Document size	decreases throughput rapidly, 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 on memory footprint	minimal effect on throughput, until over-utilization	linear impact on utilization of pre-mapped resources; no effect throughput	small impact on area, minimal on throughput
Percentage of * and //	6% to 30% decrease in throughput per 10% added	no effect	no effect	minimal effect
# 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

Figure 2.25: Effects of the factors studied on the different filtering platforms.

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 7 GB).

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 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.

Chapter 3

XML Twig Filtering

In Chapter 2, a stack-based approach to XPath processing was presented, allowing the use of '/' and '//' axises, as well as wildcard nodes in the query profile, and recursion (nesting) in the XML document. This method proved to consume less area on the FPGA and provide higher throughput, when compared to the implementation proposed in [53]. However, to process a complex twig structure, one common approach is to break a twig into into root-to-leaf paths, and an extra join step is required to join the results. The YFilter system takes such an approach by breaking twigs into absolute paths and inserting each path into the NFA to be matched. The match location for each path is recorded and utilized during the post-processing phase, where the Nested Path Filter is applied to join the paths. However, we are unable to adopt such an approach since the match location for each step in the XPath expression must be stored, and FPGAs have limited resources. In this chapter, we present a method which performs twig matching holistically on the FPGA by compiling each query profile into a hardware circuit. We also provide a dynamic programming formulation for the stack operations, as explained in Section 3.2. We compared this approach against other FPGA-based possible approaches, such as root-to-leaf path matching or parent-child/ancestor-descendant pair matching. These approaches require less area, however, introduce false positives. This comparison covers the full spectrum of granularity matching when considering XML filtering on FPGAs.

The FPGA setup used is as described in Section 2.5, such that the path query matching engines are replaced with the more complex twig query matching engines.

3.1 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 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 will 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 [88]. 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. Note that the terms "profile" and "query" are used interchangeably.

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 the 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 [2, 16, 27, 40]. These memory-bound solutions, however, suffer from the Von Neumann bottleneck and are unable to handle large volume of input streams. On the other hand, FPGAs have been shown to be particularly suited for the stream processing of large amounts of data and do not suffer from the memory offloading problem faced by software implementations. Furthermore, GPUs as co-processors are also a favorable option for applications requiring massively parallel computations [32, 4, 35, 45], such that sequential computations are run on the CPU while the computationally-intensive part is accelerated by the highly parallel GPU architecture.

This dissertation examines how to exploit the parallelism found in XPath filtering. Using an incoming XML stream, parsing and matching with thousands of user profiles are performed simultaneously by matching engines. We show the benefits and tradeoffs 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, the proposed 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.

This chapter presents a non-trivial extensions to the path filtering algorithm (Chapter 2) to support unordered holistic twig matching on FPGAs without any false positives. Experimental comparison of different granularities of twig matching is presented, namely path-based (root-to-leaf) and pair-based (parent-child or ancestor-descendant). Comprehensive experiments are provided, comparing the throughput, area utilization and the accuracy of matching (percent of false positives) of the holistic, path-based and pair-based FPGA approaches. The proposed approach yields up to three orders of magnitude higher throughput when compared to state-of-the-art single core CPU-based filtering mechanisms.

3.2 Holistic Twig Matching

In this section, we present a detailed overview of the proposed filtering mechanism.

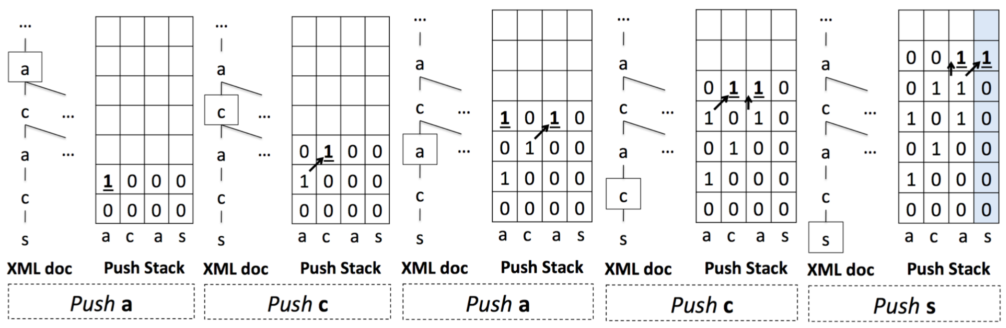


Figure 3.1: An event by event overview of the matching of path *a/c/a//s*. © 2011 IEEE.

3.2.1 Push Stacks for Path Matching

Matching for a twig consists of two parts working conjointly, namely, matching the root-to-leaf paths of the twig (Chapter 2), and appropriately joining the matched paths while reporting back to the root (Section 3.2.2).

3.2.1.1 Root-to-Leaf Path Matching

In order to successfully match a twig, the partial matching of every path from root to leaf belonging to that twig should be achieved. The matching of each root-to-leaf path requires what we refer to as a *push stack*. These are detailed in Section 2.4.

Figure 3.1 depicts an event-by-event example of the matching of the path *a/c/a//s*, as a sample XML document is being traversed. Notice how, from the 3rd to the 4th event, a '1' was allowed to propagate horizontally upwards in the 3rd column. An in-depth overview of push stacks is provided in Section 2.4

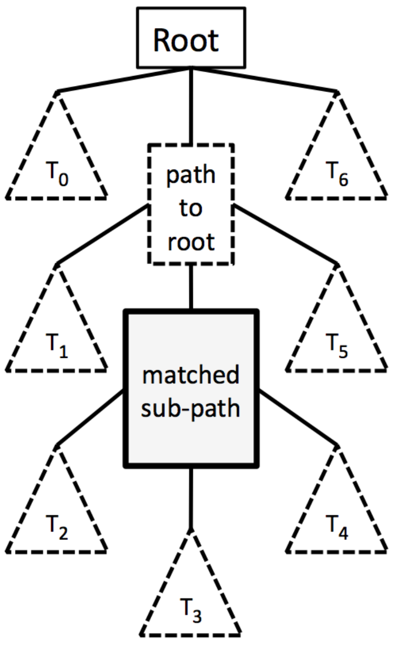


Figure 3.2: Generic view of any XML document with regards to any matched path. © 2011 IEEE.

3.2.1.2 Supporting Ancestor/Descendant Relationships

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

In Figure 3.2, sub-trees numbered according to the order by which they are encountered while streaming the XML doc. Moreover:

- The *Matched Sub-Path* can consist of 1 or more nodes.
- Any of *T0 . . . T6* could consist of zero or more nodes.
- The *Path To Root* can consist of 0 or more nodes.

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

- T0, T1, and T2 will not see a matched sub-path because by then the latter would have not matched; even after being in a match state, none of those subtrees are visited again.

  T3 is visited after the matched sub-path is matched, and the matched state of that sub-path is visible to this subtree, since push stacks report matches on pop, and T3 is reached after a series of push events.

- T4, T5, and T6 are visited after the matched sub-path is matched, and following a series of pop events. However, the matched state of the matched sub-path is also not visible here, as the leaf of the latter would be popped then, and the matched state of the sub-path with it.

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

3.2.1.3 Mapping Algorithm

The algorithm for mapping path nodes to push stack columns is covered in-depth in Section 2.5.2.2. Stack compaction optimizations are introduced to reduce to overall number of required stack columns. This is in contrast to the naive mapping, where each query node is coupled with a corresponding stack column.

With respect to twigs, nodes that are followed by both '/' and '//' in a twig require two push stack columns.

3.2.2 Pop Stacks for Joins

Recall that matching a twig consists of two parts working conjointly, namely the push stack , and the pop stack, as seen in Figure 3.4. Using the push stack, matching

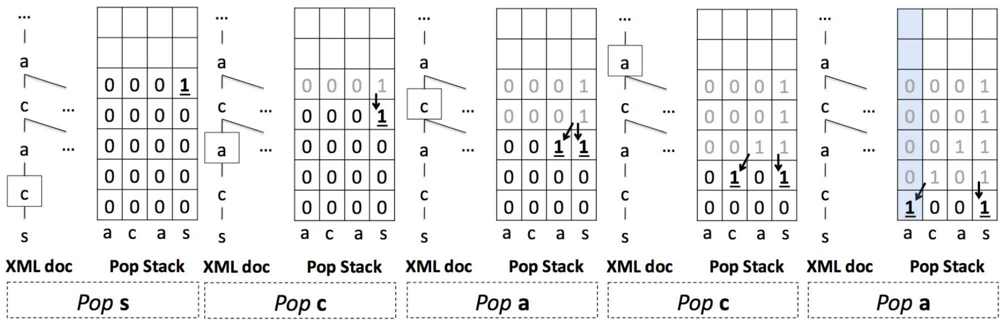


Figure 3.3: An event by event overview of the reporting of the matched state of path $a/c/a//s$. © 2011 IEEE.

any root-to-leaf path is achieved. However, matching all paths in a twig does not imply matching the twig, unless all paths match in the correct positions in order to form the twig at hand. The *pop stack* helps us achieve this task.

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

3.2.2.1 Leaf-to-Split Node Matched Path Reporting

Let us assume a simple twig of the form $a[b/c]/d/e$ that appears in a streamed XML document. Thus, (assuming) the path $a/b/c$ will be visited first, then each of c and b will be popped (in that order), thus rendering the twig root node a at the top of the stack again. At this point, there is no way to tell whether the twig will be found in the document, since the path $a/d/e$ would not have matched yet. However, when the leaf node c was encountered, it was noted in the push stack that the first path matched. This information has been lost when c was popped. A pop stack is needed here to report back to the twig root that the first path matched.

Figure 3.3 illustrates an event-by-event example of the reporting of the matched state of path $a/c/a//s$, following what was shown in Figure 3.1.

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

This is achieved through the use of dynamic programing, where the dynamic programming table is a stack, whose top of stack address is given by the address generator. Every stack column represents a path node, and every stack row represents a document node.

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

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

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

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

where:

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

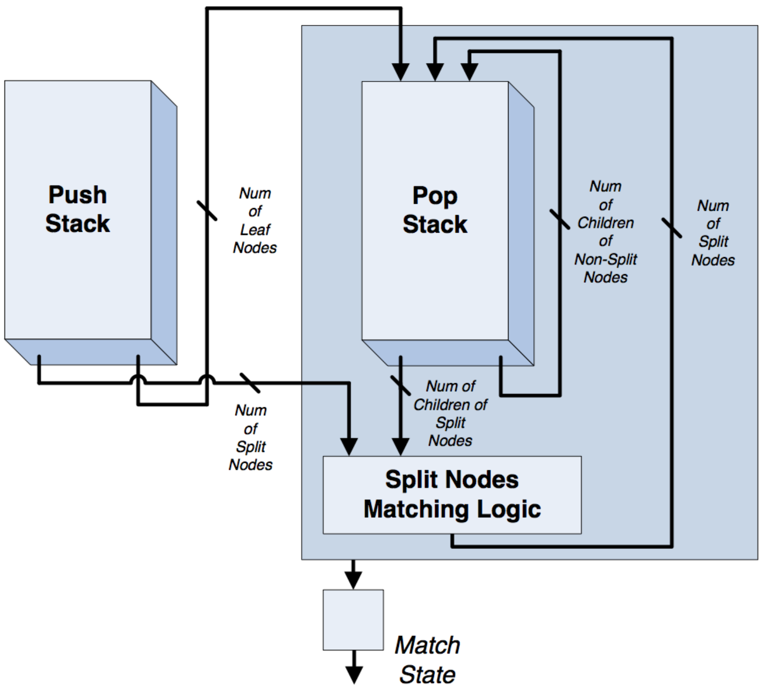
The recurrence equation encapsulates four main cases to report matches on a pop. If a node is a leaf in the twig (a), then a match is reported by propagating the corresponding output from the push stack. If a node is a split node (b), then a match can be reported only if all of its children/descendants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

Only if a node is preceded by '//' (d), then the reported match state is allowed to propagate vertically downwards. A '1' propagating diagonally left from a node preceded by '//' requires an extra check from the push stack, to ensure that the '1' is propagating to a valid path node.

Figure 3.4 illustrates a high-level view of the underlying matching mechanism when targeting the matching of a twig as a whole. Here, a single push stack is required, the width of which is defined by the mapping algorithm (introduced in 3.2.1.3), and a single pop stack, the width of which is the number of nodes in the twig. The split nodes matching logic consists of the AND-ing logic as noted in (b) as part of the recurrence equation. This logic further requires some cells from the push stack to help determining the exact position of the split node in the document.

3.2.2.2 Pop Stack Properties

Here are the properties of Pop Stacks:



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

Figure 3.4: Generic view of a holistic twig matching engine, using a push and a pop stacks. © 2011 IEEE.

- Pop stacks update both on push and pop events, s.t.:
  - On a push, always force writing a '0' (to reset to a new state).
  - On a pop, only a '1' can propagate downwards, but never a '0' (in order to not erase previous states).
- A '1' propagates diagonally from and to any column connecting a parent or ancestor to a child or descendant, respectively.
- Only in a '//' column, a '1' propagates vertically downwards, to indicate matches to all ancestors residing only on the path from root to the node mapped in that respective column (see Section 3.2.2.3 and the supporting Figure 3.2).

3.2.2.3 Supporting Ancestor/Descendant Relationships

Reporting the matching state of a sub-path preceded by a '//' should be visible to the ancestor of that sub-path. Referring to Figure 3.2, using pop stacks, reporting matched path rooted by a node preceded by '//' occurs as such:

- T0, T1, and T2 will not see a matched path because by then the latter would have not matched; even after being in a match state, none of those subtrees are visited again.

  T3 is visited after the matched path is matched; however, the matched state of that path is not visible to this subtree, since pop stacks report matches on pop, and T3 is reached after a series of push events.

- T4, T5, and T6 are visited after the matched path is matched, and following a series of pop events. However, the matched state of the matched path is also not visible here, as some push events are required to go into the tree nodes, and pop stack contents do not propagate on pushes.

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

3.2.2.4 Mapping Algorithm

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

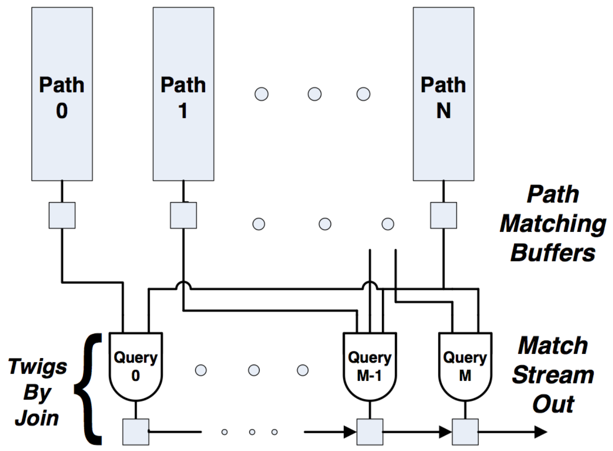


Figure 3.5: Generic view of any XML document with regards to any matched path. © 2011 IEEE.

3.3 Breaking Twigs into Paths and/or Pairs

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

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

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

Figure 3.5 provides an overview of the Path/Pair filtering approach. Twigs are split into several root-to-leaf paths or parent-child/ancestor-descendant pairs. Every path/pair matching engine is followed by a match state buffer. In case a path/pair match state is true, that state is held for the document's entirety. After the document is processed, a join step is required to verify all parts (paths or pairs) that represent a twig profile were matched. Thus, every twig requires a single AND gate in order to join all the partial paths/pairs that constitute it. In case of a single path/pair not matching, the twig's matching state is marked as false.

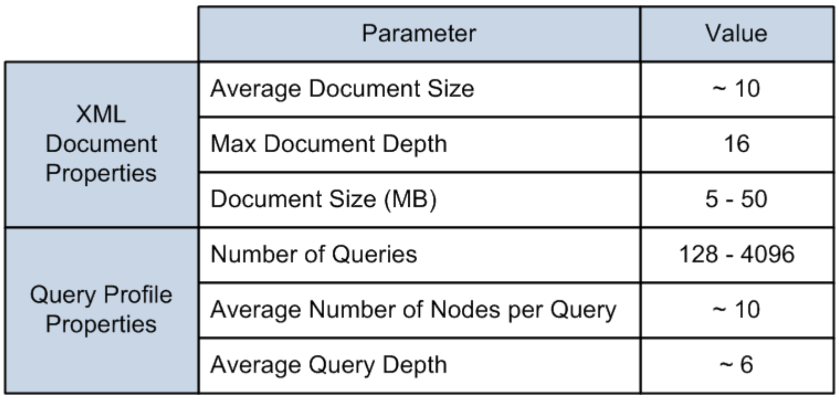
3.3.1 Advantages

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

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

3.3.2 Disadvantages

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



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

Figure 3.6: Experimental Parameters. © 2011 IEEE.

advantageous for applications where false positives are allowed or when the verification cost of the reduced profile set is small.

3.4 Filtering System Evaluation

In this section, we evaluate the proposed hardware architectures, and compare them to two of the state-of-the-art software counterparts, namely FiST [40] and YFilter [16]. For the experiments, we utilized the DBLP DTD provided by [80] to generate XML documents and user profiles. XML documents and query profiles were generated using the ToXGENE XML Generator [6] and YFilter query generator [16], respectively. Furthermore, in all datasets, we set the number of unique tags to 64, each consisting of two bytes. The experimental parameters are listed in 3.6. We make use of four datasets, namely, **Datasets 1 - 4**, where the probability of occurrence of '*' and '//' in the queries is set to 5%, 10%, 15% and 20%, respectively.

3.4.1 Hardware System Evaluation

Our hardware platform consists of a Xilinx Virtex 5 LX330 FPGA [86]. All the push and pop stacks are implemented using on-chip Distributed Memory (DMEM) blocks, available on Xilinx FPGAs. We provide a thorough evaluation of the five hardware approaches that have been presented so far in this chapter. These are:

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

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

We show in Figure 3.8 the resource utilization percentage on the target FPGA, while doubling the number of twigs. In Section 3.4.1, we make use of query Dataset 2, having 10% occurrence of '//' and '*'. Typically, the percentage of occurrence of '//' and '*' has minimal (negligible) effect on the FPGA-based approaches.

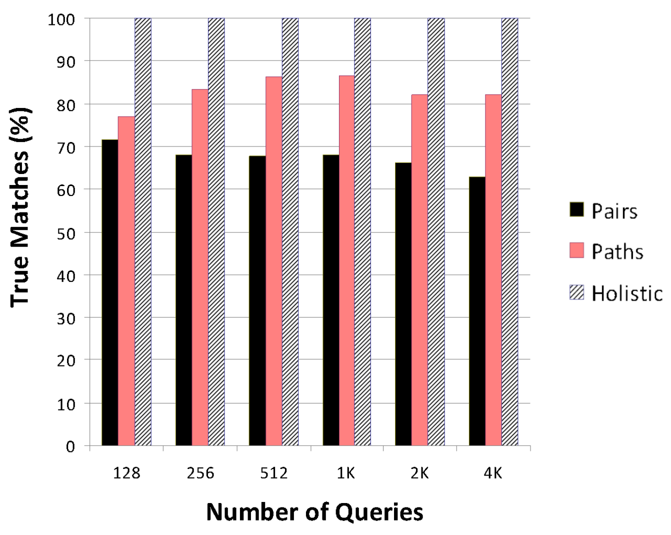


Figure 3.7: The percentage of true matches as reported from several hardware approaches. © 2011 IEEE.

The holistic approach is the least scalable, in contrast with the breaking of twigs into paths and pairs, where the commonalities across queries are exploited. Naturally, there are more common pairs than there are paths. However, looking at Figure 3.7, the percentage of false positives is largest when using pairs. The holistic approach, on the other hand, yields no false positives.

In Figure 3.9, while doubling the number of queries, we show the throughput of all proposed approaches, assuming a stream of one byte per cycle. As the FPGA utilization increases, the throughput decreases, as the task of placement and routing of components on the FPGA is hardened. For a given number of queries, the holistic approach exhibits the lowest throughput, being the more complex of all five. The pairs however, being the simplest, almost always demonstrates a higher throughput for a given query set.

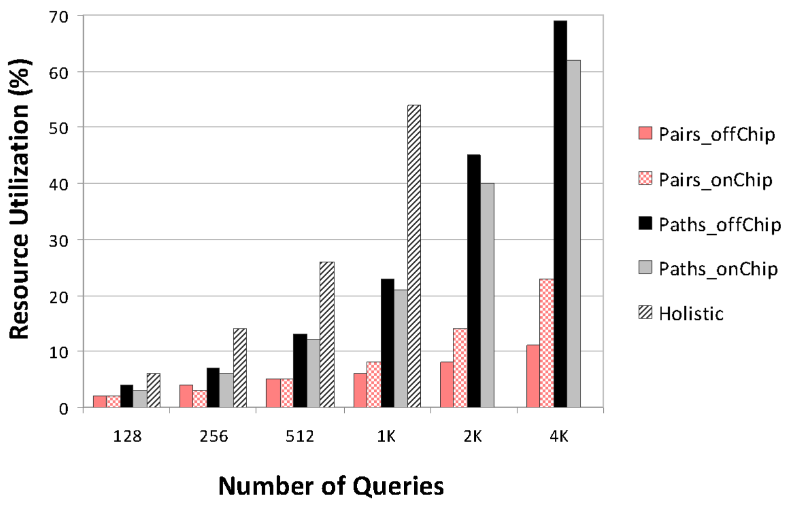


Figure 3.8: Resource utilization of the proposed hardware architectures on a V5LX330 FPGA, targeting Dataset 2 (10% occurrence of '//' and '*' in the queries). Note that the join step is not performed when using *off-chip* filtering methods. © 2011 IEEE.

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

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

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

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

We plot this metric in Figure 3.10. As the Holistic approach is mostly affected by high resource utilization across all query sets, the true work per unit area is the

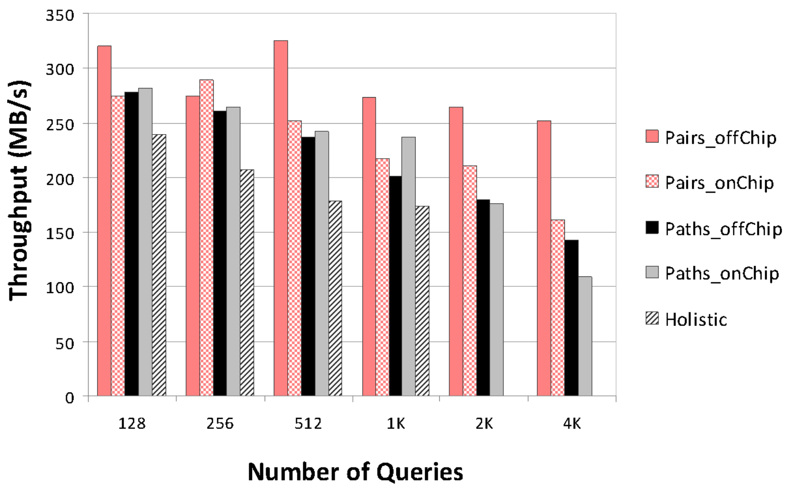


Figure 3.9: Throughput of the proposed hardware architectures on a V5LX330 FPGA, targeting Dataset 2 (10% occurrence of '//' and '*' in the queries). Note that the join step is not performed when using *off-chip* filtering methods. © 2011 IEEE.

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

3.4.2 Hardware/Software Performance Evaluation

Next, the proposed hardware approaches are compared against two state of the art software approaches, namely, YFilter [16] and FiST [40]. These approaches where chosen since they represent the two main techniques used for the XML filtering problem. The software approaches were run on a quad core 2.33GHz Intel Xeon machine with 8GB of RAM running Linux Red Hat 2.6. YFilter supports unordered query matching by

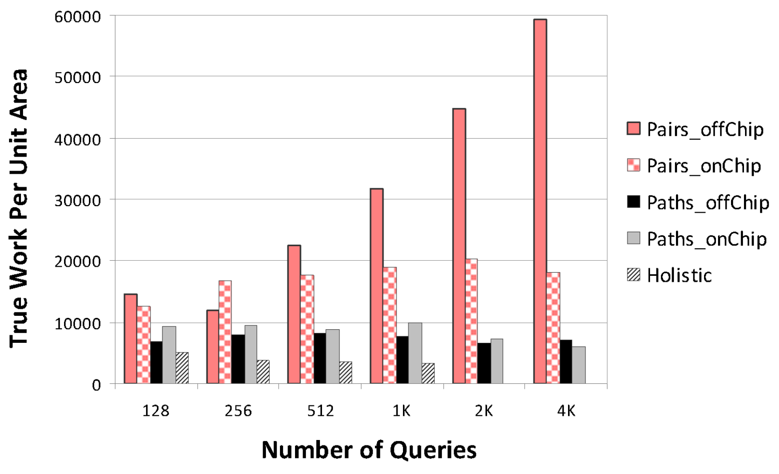


Figure 3.10: Efficiency of the proposed hardware architectures on a V5LX330 FPGA, targeting Dataset 2 (10% occurrence of '//' and '*' in the queries). Note that the join step is not performed when using *off-chip* filtering methods. © 2011 IEEE.

breaking twigs into root-to-leaf paths and building a unified NFA over the set of paths. After path matching, an additional join step is required to join the paths at the split nodes. FiST, on the other hand, only supports ordered query matching. The XML document and set of queries are transformed into their Prufer sequence representation and subsequence matching is performed to determine if a match exists. FiST also requires a post-processing phase to filter false positives.

The throughput of YFilter and FiST is shown in Figure 3.11. On average, YFilter achieves a throughput of 1.7, 1.2 and 0.3 MB/s for 5, 25 and 50 MB documents, respectively. FiST achieves a higher throughput of 3.88, 2.58, 1.40 MB/s for 5, 25, and 50MB documents, respectively, since it processes twigs in a holistic manner rather than processing individual paths. Although YFilter and FiST scale with increasing query workload, it is clear, however, that both approaches do not scale with increasing document size. In comparison, the holistic FPGA-based approach achieves an average of 200x speedup for 1K queries, and up to three orders of magnitude speedup.

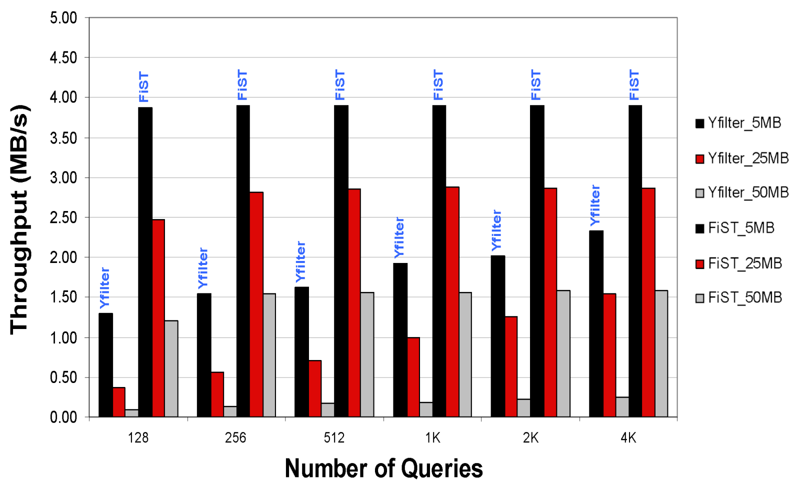


Figure 3.11: Throughput of FiST and YFilter when using 5, 25, and 50MB XML documents, and queries for dataset 2 (10% occurrence of '//' and '*' in the queries). © 2011 IEEE.

It should be noted that XML stream parsing is not the bottleneck for the software approaches. For the given experimental setup, using the Xerces Java parser, we were able to achieve a throughput rate of 23.1, 57.5, and 72.2 MB/sec for 5, 25 and 50 MB documents, respectively. Thus, the profile matching process is contributing to the low throughput of the software approaches, not the XML parsing. Whereas, for the FPGA-based approaches that we present, parsing is now the bottleneck, as the overall throughput is directly proportional to the number of bytes of XML that can be parsed per cycle. The query matching engines can process up to one SAX event per cycle; however, in practice, every event requires at least three bytes of XML ('<', '>', and a one-byte label). Nevertheless, the XML parsing problem is orthogonal to our current research, and other researchers have proposed complex FPGA-based parsers which are able achieve an average throughput of two bytes of XML per cycle (a peak 4 bytes per cycle) [19]. The proposed approaches in this dissertation are able to take advantage of all the effective bandwidth provided by these high-performance parsers.

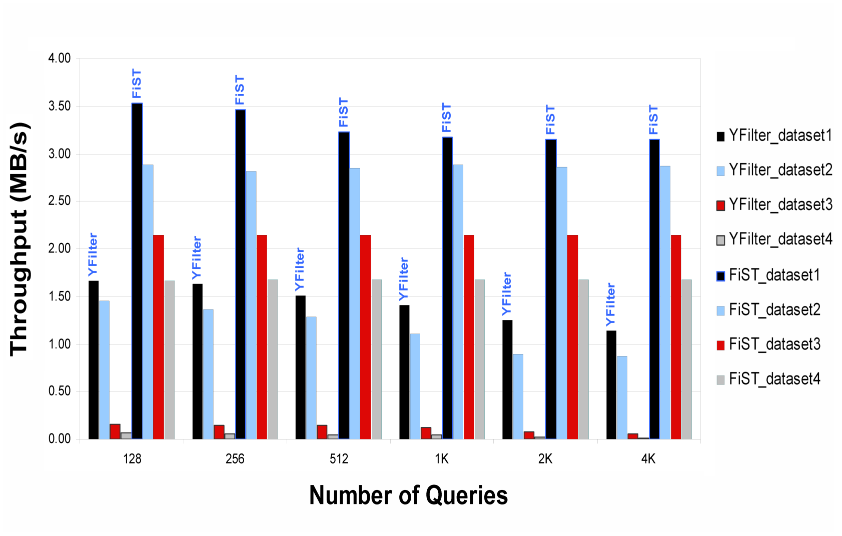


Figure 3.12: Throughput of FiST and YFilter for a 25MB XML document, while increasing the probability of occurrence of '//' and '*' in the queries. © 2011 IEEE.

In Figure 3.12, we show the effect of increasing the probability of occurrence of '//' and '*' in the query dataset, on software throughput, for a 25MB XML document. The performance of both YFilter and FiST highly depends on the complexity of the queries. As the occurrence of '//' and '*' reaches 15%, the software throughput degrades. As both YFilter and FiST are composed of two phases, query matching and verification/join phase, high occurrence of '//' and '*' lends to less selective queries and introduces more false positives in the query matching phase; thus more computation time is reuired for the verification/join phase. Furthermore, high occurrence of '//' and '*' degrades performance of the software approaches since expensive computations are performed to verify the specified pattern is satisfied. This performance degradation is not applicable to FPGA-based systems where the circuitry is the same for dealing with any tag, or any relation. Here, at 1K queries of dataset 4, the holistic FPGA-based approach yields an acceleration of 7000X and 2300X when compared to YFilter and FiST, respectively.

It should be noted that FiST only deals with ordered twigs, where the order of the subtrees under a split node is enforced. This can be seen as a set-back for applications where the extra flexibility level is desired. Nonetheless, while our current approach addresses unordered twig matching, enforcing order can be trivially achieved through the addition of more checks at the recurrence equation level. Specifically, the match state of the root of a subtree under a split node would not report diagonally left in a pop stack, unless all previous sibling subtrees would be indicated as matched. As it suffices to test the previous sibling alone, this check can be achieved almost for free in terms of resources, with no accompanying performance penalty.

3.5 Conclusions

In this work, we presented a novel FPGA-based architecture to address the XML filtering problem. Using custom stack generation, our architecture is the first providing full support for twig pattern matching, including parent-child ('/') and ancestor-descendant ('//') axes, wildcard nodes, and accounting for recursion in the XML document and queries. Hardware architectures do not suffer from the memory bottleneck problem (better known as the Von Neumann bottleneck), since they are highly suitable for stream processing; they would also not suffer from the limitations of sequential processing, as the proposed architecture would support thousands of twig matching engines operating in a parallel fashion. In addition to being able to match thousands all queries in parallel, through dynamic programming on FPGAs, we exploit parallelism by simultaneously matching for all nodes in the query.

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

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

Chapter 4

Querying Spatio-Temporal Databases

4.1 Introduction

Due to their relative ease of use, general purpose processors are commonly favored at the heart of many computational platforms. These processors are deployed in environments with varying requirements, ranging from personal electronics, to game consoles and up to server-grade machines. General purpose CPUs follow the Von-Neumann model, and execute instructions sequentially. Furthermore, performance does not always linearly scale in multi-processor environments, mostly due to the challenges of data sharing across cores. As it is non-trivial for these CPUs to satisfy the increasing time-critical demands of several applications, they are often coupled with application- or domain-specific parallel accelerators, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), which strive given a certain class of instructions and memory access patterns.

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

As traditional platforms are increasingly hitting limitations when processing large volumes of streaming data, researchers are investigating FPGAs for database applications. Recent work has focused on the adoption of FPGAs for data stream processing [61, 78, 68, 85]. The Glacier component library [61] proposed logic circuits of common operators, such as selection, aggregation, and grouping for stream processing. [78] investigated the speedup of the frequent item problem using FPGAs, while [85] proposed FPGAs for complex event detection using regular expressions. Predicate-based filtering on FPGAs was investigated by [68], where user profiles are expressed as a conjunctive set of boolean filters.

In this paper, we describe an FPGA-based setup allowing users to query spatio-temporal databases in a very powerful and intuitive way. Figure 4.1 depicts a generic overview of the various steps performed in spatio-temporal querying setups. Streams

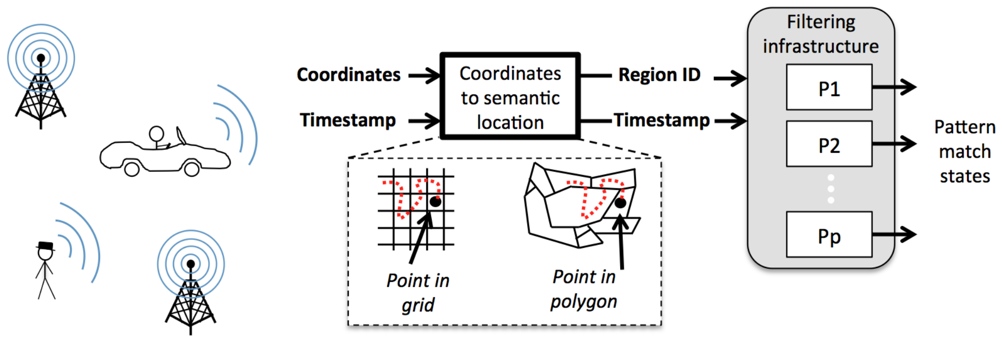


Figure 4.1: Generic overview of various steps performed in spatio-temporal querying setups.

of trajectory data are harvested from devices, such as GPS and cellular devices. Coordinates are then translated into semantic regions that partition the spatial domain; these regions can be grid regions representing areas of interests (e.g., neighborhoods, school districts, cities). Our work is based on complex pattern queries previously defined in [81, 82] to search for specific motion patterns in trajectories. A pattern query is specified as a combination of sequential spatio-temporal predicates, allowing the end user to search for specific parts of interests in trajectory databases. For example, the pattern query "*Find all taxis (trajectories) that first were in downtown Munich in the morning, later passed by Olympiapark around noon, and then were closest to the Munich airport*" provides a combination of temporal, range and Nearest-Neighbor (NN) predicates that have to be satisfied in the specific order. Essentially, flexible patterns cover that part of the query spectrum between the single spatio-temporal predicate queries, such as the range predicate that covers certain time instances of the trajectory life (e.g., "*Find all trajectories that passed by the Deutsches Museum area at 11pm*"), and similarity/clustering based queries, such as extracting similar movement patterns from a trajectories that cover the entire lifespan of the trajectory (e.g., "*Find all trajectories that are similar to a given query trajectory according to some similarity measure*").

Complex pattern queries can also have variable spatial predicates, and thus substantially enhancing the flexibility and expressive power of the framework. An example of a variable-enhanced query is "*Find all trajectories that started in a region X, then visited the downtown Munich, then at some later point visited X again*".

This work serves as a proof-of-concept on the performance benefits of evaluating complex motion pattern queries using FPGAs. Here we focus on the challenges of supporting hundreds (up to thousands) of variable-enhanced flexible patterns on FPGAs in streaming (fully-pipelined) fashion. Using FPGAs all pattern query predicates are evaluated in parallel over sequential streams of trajectories, hence resulting in considerable speedup over CPU-based approaches. This also holds even when compared to CPU-based setups where the pre-processing of trajectories into inverted indices is performed beforehand. To the best of our knowledge, this work is the first detailing FPGA support for variable-enhanced flexible pattern queries.

The remainder of this paper is organized as follows: related work is introduced in Section 4.2, and the query language is detailed in Section 4.3; Section 4.4 goes over the FPGA-based querying architecture; experimental evaluation is provided in Section 4.5, and conclusions appear in Section 4.6.

4.2 Related Work

Single predicate queries (Range and *NN* queries) for trajectory data have been well studied in the past (e.g., [65, 77]). To make the evaluation process more efficient, trajectories are approximated using Minimum Bounding Regions (MBR) to be then indexed using hierarchical spatiotemporal indexing structures [30, 76]. However, these solutions are efficient only to evaluate single predicate queries. For moving object data, patterns have been examined in the context of query language and modeling issues [20, 54, 5] as well as query evaluation algorithms [29, 18, 60].

The *FlexTrack* system [81, 82], which our work is based on, provides a more general and powerful query framework than previous approaches. In *FlexTrack*, queries can involve both fixed and *variable* regions as well as regular expression structures (e.g., repetitions, negations, optional structures) and explicit ordering of the predicates along the temporal axis. This system uses a hierarchical region alphabet, where the user has the ability to define queries with finer alphabet granularity (*zoom in*) for the portions of greater interest and higher granularity (*zoom out*) elsewhere. In order to efficiently evaluate flexible pattern queries, the *FlexTrack* system employs two lightweight index structures in the form of ordered lists that are stored in addition to the raw trajectory data. Given these index structures, two different algorithms for evaluating flexible pattern queries are provides: the Dynamic Programming Pattern (DPP), and the Index Join Pattern (IJP) algorithms. In the next section we describe the IJP algorithm in more details, since we use it to evaluate our proposed solution.

The use of hardware platforms for pattern matching has been recently proposed in the past. Evaluating regular expressions on FPGAs has been explored by many studies such as in [73, 52, 39, 85]. Most of these works focus on deep packet inspection and security as applications of interest. Using FPGAs, speedups of up to two orders of magnitude versus CPU-based approaches is achieved, as every data element in stream can be processed in a single hardware cycle.

The work in [59, 56, 57] presents a novel dynamic programming, push down automata approach for matching path and twig type patterns in the structure of XML documents, using FPGAs (path, twig) and GPUs (path). Using the massively parallel solution running on parallel platforms, up to three orders of magnitude speedup versus state-of-the-art CPU bases approaches was achieved.

In [73], the authors detail the NFA implementation of regular expressions on FPGAs. The authors in [52] propose generating hardware code from Perl Compatible Regular Expressions. The work in [39] focuses on the DFA implementation of regular expressions, while merging commonalities among multiple DFAs. [85] proposes the use of regular expressions for the representation of spatio-temporal queries. An FPGA implementation is detailed, allowing the sharing of query evaluation engines among several trajectories, with a minor impact on performance. The authors in [10] investigate the use of GPUs for the fast computation of proximity area views over streams of spatio-temporal data. Our work differs from the above from the perspective of the query language, described in Section 4.3. Specifically, an investigation of the FPGA-based support of variable-enhanced patterns is offered here.

4.3 The *FlexTrack* System

In this section we provide a briefly description of the query language syntax, as well as the key elements in the *FlexTrack* system.

4.3.1 Pattern Query Language

A trajectory T_{id} is defined as a list of locations collected for a specific moving object over an ordered sequence of timestamps, and is stored as a sequence of n pairs $\{(ls_1, ts_1), \ldots (ls_n, ts_n)\}$, where $ls_i \in \mathbb{R}^d$ is the object location recorded at timestamp ts_i ($ts_{i-1} < ts_i$). In the *FlexTrack*, the spatial domain is partitioned by a fixed set Σ_l of non-overlapping regions. Regions correspond to areas of interest (e.g. *school districts*, *airports*) and form the alphabet $\Sigma = \bigcup_l \Sigma_l = \{A, B, C, ...\}$ of the language.

In the *FlexTrack* query language, a spatio-temporal predicate $\mathcal{P}$ is defined by a triplet $\langle op, \mathcal{R}[, t]\rangle$, where $\mathcal{R}$ corresponds to a predefined spatial region in Σ or a *variable* in Γ ($\mathcal{R} \in \{\Sigma \cup \Gamma\}$), *op* describes the topological relationship (e.g. *meet*, *overlap*, *inside*) that the trajectory and the spatial region must satisfy over the (optional) time interval t ($t := (t_{from} : t_{to}) \mid t_s \mid t_r$). A predefined spatial region is explicitly specified by the user in the query predicate (e.g. "the downtown area of Munich'). In contrast, a *variable*, e.g. "$@x$", denotes an arbitrary region using the symbols in $\Gamma = \{@a, @b, ...\}$. Conceptually, *variables* work as placeholders for explicit spatial regions and can be bound to a specific region during the query evaluation.

In the *FlexTrack* language, a pattern query $\mathcal{Q} = (\mathcal{S}\ [\cup\ \mathcal{D}])$ is defined as a combination of a sequential pattern $\mathcal{S}$ and an optional set of constraints $\mathcal{D}$, where a trajectory matches $\mathcal{Q}$ if it satisfies both $\mathcal{S}$ and $\mathcal{D}$ parts. $\mathcal{D}$ part of $\mathcal{Q}$ allow us to describe general constraints. For instance, constrains can be distance-based constraints among the *variables* in $\mathcal{S}$ and the predefined regions (for more details, see [81]). And $\mathcal{S} := \mathcal{S}.\mathcal{S} \mid \mathcal{P} \mid !\mathcal{P} \mid \mathcal{P}^{\#} \mid ?^{+} \mid ?^{*}$ corresponds to a sequence of spatio-temporal predicates, while $\mathcal{D}$ represents a collection of constraints that may contain regions defined in $\mathcal{S}$. The wild-card "?" is also considered a variable, however it refers to any region without occurring multiple times within a $\mathcal{S}$.

The use of the same set of *variables* in describing both the topological predicates and the numerical conditions provides a very powerful language to query trajectories. To describe a query in *FlexTrack*, the user can use fixed regions for the parts of the trajectory where the behavior should satisfy known (strict) requirements, and *variables* for those sections where the exact behavior is not known but can be described by *variables* and the constraints between them.

In addition to the query language defined previously, we introduce the variable region set constraint. A region set constraint $\boldsymbol{V} \in \Sigma$ can be only applied to variable predicates, having the purpose of limiting the region values that a given variable can take. Region set constraints are optional, per variable, and are defined after the query pattern.

Consider the following query pattern and region set over *@x*, $A.B.@x.C.?+.@x$ $\{$*@x : A, D, E*$\}$. Here, @*x* is constrained by the regions {A,D,E}. In practice, a variable can be limited to the neighboring regions of the fixed query predicates. Other constraints can be set by the user, hence, limiting the number of matches of interest. From a performance perspective, the use of variable regions set constraints greatly simplifies hardware support for variable predicates separated by wildcards ?+/?*, as detailed in Section 4.4.

4.3.2 Pattern Query Evaluation

The *FlexTrack* system employs two lightweight index structures in the form of ordered lists that are stored in addition to the raw trajectory data. There is one *region-list* (*R-list*) per region and one *trajectory-list* (*T-list*) per trajectory. The *R-list* $\mathcal{L}_{\mathcal{I}}$ of a given region $\mathcal{I} \in \Sigma$ acts as an inverted index that contains all trajectories that passed by region $\mathcal{I}$. Each entry in $\mathcal{L}_{\mathcal{I}}$ contains a trajectory identifier T_{id}, the time interval (*ts-entry*:*ts-exit*] during which the trajectory was inside $\mathcal{I}$, and a pointer to the *T-list* of T_{id}. Entries in a *R-list* are ordered first by T_{id} and then by *ts-entry.*

In the *FlexTrack* system we use a uniform non-overlapping grid to partition the space and we overestimate the regions in Σ by approximating each one of them with the smallest collection of grid cells that completely encloses the region. Because of the overestimation, false positives may be generated from regions that do not completely fit the set of covering grid cells. They, however, can be removed with a verification step using the original trajectory data.

In order to fast prune trajectories that do not satisfy $\mathcal{S}$, the *FlexTrack* system uses the *T-list*, where each trajectory is approximated by the sequence of regions it visited in each level of the partitioning space. A record in the *T-list* of T_{id} contains the region and the time interval (*ts-entry*:*ts-exit*] during which this region was visited by T_{id}, ordered by *ts-entry.* In addition, entries in *T-list* maintain pointers to the *ts-entry* part in the original trajectory data. With these index structures, there are four different strategies for evaluating flexible pattern queries:

1. *Index Join Pattern* (*IJP*): this method is based on a merge join operation performed over the *R-lists* for every fixed predicate in $\mathcal{S}$. The *IJP* uses the *R-lists* for pruning and the *T-lists* for the *variable* binding;

2. *Dynamic Programming Pattern* (*DPP*): this method performs a subsequence matching between every predicate in $\mathcal{S}$ (including *variables*) and the trajectory approximations stored as the *T-lists.* The *DPP* uses mainly the *T-lists* for the subsequence matching and performs an intersection-based filtering with the *R-lists* to find candidate trajectories based on the fixed predicates in $\mathcal{S}$;

3. *Extended-KMP* (*E-KMP*): this method is similar to *DPP*, but uses the Knuth-Morris-Pratt algorithm [38] to find subsequence matches between the trajectory representations and the query pattern;

4. *Extended-NFA* (*E-NFA*): this is an NFA-based approach to deal with all predicates of our proposed language. This method also performs an intersection-based prun-

ing on the *R-lists* to fast prune trajectories that do not satisfy the fixed spatial predicates in S.

4.4 Proposed Hardware Solution

4.4.1 Compiling Queries to Hardware

In this work, pattern queries are evaluated in hardware on an FPGA device. As trajectories are compared against hundreds and potentially thousands of pattern queries, manually developing custom hardware code becomes an extremely tedious (and error prone) task. Unlike software querying platforms, where a single generic kernel (or set of) can be used for the evaluation of any query pattern, hardware is at an advantage when each query pattern is mapped to a customized circuit. Customized circuitry has the benefits of only utilizing the needed resources out of all (limited) on-chip resources. Furthermore, the throughput of the query evaluation engines is limited by the operational frequency (hardware clock) which can in-turn be optimized to maximize performance.

For this purpose, a software tool written in C++ was developed from scratch (more than 6,500 lines of code), taking as input a set of user-specified pattern queries, and automatically generating a customized Hardware Description Language (HDL) circuit description (see Figure 4.2). A set of compiler options can be specified, such as the degree of matching accuracy (reducing/eliminating false positives), and whether to make use of certain resource utilization (common prefix) and performance (clustering) optimizations.

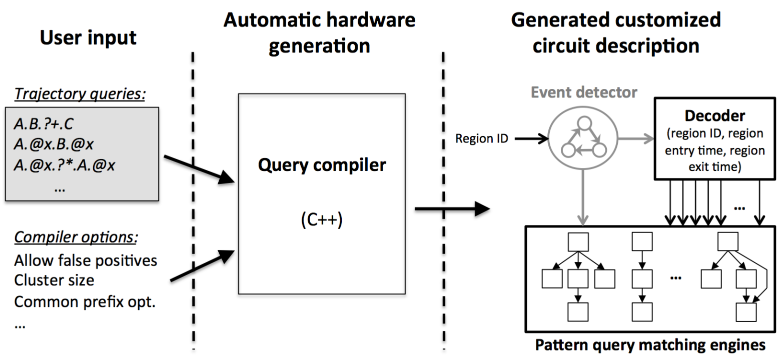


Figure 4.2: Query-to-hardware tool flow.

Utilizing a query compiler provides the flexibility of software (ease of expression of queries from a user perspective), and the performance of hardware platforms (higher throughput), while no compromises are introduced.

4.4.2 High Level Architecture Overview

As depicted in Figure 4.2, assuming an input stream of pairs ⟨ *location, timestamp* ⟩, the first step consists of translating the location onto semantic data; specifically, the region-IDs are of interest, using which the query patterns are expressed. The computational complexity of translating locations to regions depends on the nature of the map, and are discussed below:

- In the case of regions defined by a grid map, simple arithmetic operations are performed on the locations. These can be performed at wire speed (no stalling) on an FPGA.
- In the case of polygon-shaped regions, there are several well-defined point-in-polygon algorithms and respective hardware implementations in the literature (for instance, see [21, 33, 37, 71]). However, none of these can operate at wire speed when the number of polygons is large. Here, the locations of vertices are stored

off-chip in carefully designed data structures. The latter are traversed to locate the minimal set of polygons against which to test the presence of the locations.

As the design of an efficient location-to-region-ID block is orthogonal to pattern query matching, in this work, a grid map is assumed, and the location-to-region-ID conversion is abstracted away and computed offline. The input stream to the FPGA consists of ⟨ *region-ID, timestamp*⟩ pairs. A high level overview of the generated FPGA-based architecture is depicted at the right-hand side of Figure 4.2.

An event detector controller translates the ⟨ *region-ID, timestamp*⟩ pairs to ⟨ *region-ID, entry-timestamp, exit-timestamp* ⟩ tuples. The latter are then passed to decoders which transform the region-ID into a one-hot signal, and evaluate comparisons on entry and exit timestamps as needed by pattern queries. Making use of decoders greatly reduces resource utilization on the FPGA, as computations are centralized and redundancies are eliminated.

Next, a set of motion pattern query evaluation engines are deployed, providing performance benefits through the following two parallelization opportunities:

1. **Inter-pattern parallelism:** where the evaluation of all pattern queries is achieved in parallel. This parallelism is available due to the embarrassingly parallel nature of the pattern matching problem.

2. **Intra-pattern parallelism:** where the match states of all nodes within a pattern are evaluated in parallel.

The throughput of pattern query matching engines is limited to one event per cycle. Given the current assumed streaming mechanism, events are less frequent than region-IDs.

Lastly, once a trajectory is done being streamed into the FPGA, the match state of each pattern query is stored in a separate buffer. This in turn allows the match states to be streamed out of the FPGA from the buffer as a new trajectory is queried (streamed in), hence, exploiting one more parallelism opportunity.

A description of the hardware query matching engines follows. While the discussion focuses on predicate evaluation, timing constraints are evaluated in a similar manner in the region-ID decoder, and are hence left-out of the discussion for brevity.

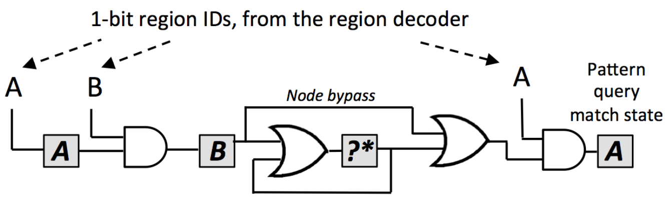
4.4.3 Evaluating Patterns with No Variables

We now describe the case of simple patterns with no variables. This approach is borrowed from the NFA-based regular expression evaluation as proposed in [52, 73]. Figure 4.3(a) depicts the matching engine respective to the pattern query *A.B.?*.A*, and Figure 4.3(b) details the matching steps of that query given a stream of region-ID events. Each query node is implemented as:

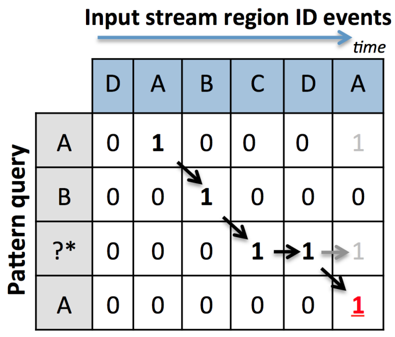
- A one-bit buffer (implemented using a flip-flop, depicted in grey), indicating whether the pattern has matched up to this node. All nodes are updated simultaneously, upon each region-ID event detected at the input stream.
- Logic preceding this buffer, to update the match state (buffer contents).

As each buffer indicates whether the pattern has matched up to that predicate, a query node can be in a matched state if and only if:

- All previous (non-wildstars) predicates up to itself have matched. Wildstars are an exception since they can be skipped by definition (zero or more). See the node


1-bit region IDs, from the region decoder
A
B
A
Node bypass
Pattern query match state
A
B
?*
A

(a)


Input stream region ID events
time
Pattern query

	D	A	B	C	D	A
A	0	**1**	0	0	0	1
B	0	0	**1**	0	0	0
?*	0	0	0	**1**	**1**	1
A	0	0	0	0	0	**1**

(b)

Figure 4.3: (a) query matching engines respective to the pattern query *A.B.?*.A*, and (b) an event-by-event overview of the matching of the query. All cells in a column are updated in parallel upon an event at the input stream. A '1' in a cell indicates that the query has matched up to that node; for a query to be marked as matched, a '1' should propagate from the first node (top row) to the last node (bottom row). Grey cell contents indicate *matched* states that did not contribute to the detected matched query state in red, but could contribute to later matches.

bypass in Figure 4.3(a). To perform this check, it suffices to check the match state of the first previous non-wildstar node.

- The current event (as noted by the region-ID decoder) relates to the region of that respective node. Wildcards are an exception, since by definition, they are not tied to a region-ID. Centralizing the comparisons and making use of a decoder helps considerably reducing the FPGA resource utilization respective to this inter-node logic (see the AND-gates in Figure 4.3(a)). This is in contrast to reading the multi-bit encoded region-ID and performing a comparison locally.

- It is a wildstar/wildplus, and it was in a match state at some point earlier. Wildstar and wildplus are **aggregation** nodes that, once matched, will hold that match state. See the OR-gate prior to the *?** node in Figure 4.3(a).

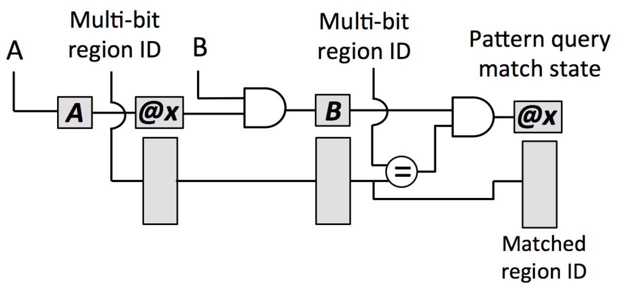
Looking closer at Figure 4.3(b), each cell reflects the match state of a query node. All cells in a column are updated in parallel upon an event at the input stream. A '1' in a cell indicates that the query has matched up to that node; for a query to be marked as matched, a '1' should propagate from the first node (top row) to the last node (bottom row). The '1' depicted in red in Figure 4.3(b) indicates that the query was detected in the input stream.

As wildstar (and wildplus) nodes act as aggregators, they hold a *matched* state once activated; hence, a '1' can propagate "horizontally" only at wildstar (and wildplus) nodes. Grey cell contents indicate *matched* states that did not contribute to the detected matched query state in red, but could contribute to later matches.

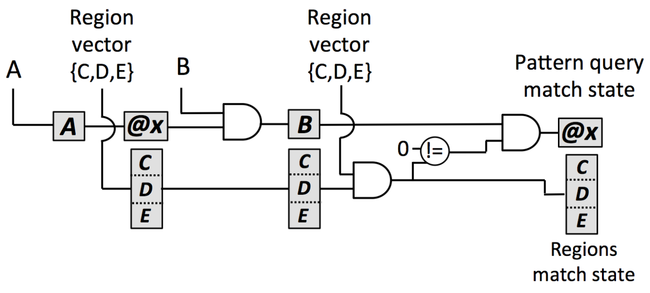
4.4.4 Evaluating Patterns with Variables and without Wildstar/Wildplus Predicates

Supporting variables in pattern query matching requires an added level of memory saving. The basic rule of variables is that *all instances of a given variable need to match the same region-ID for a variable to be in a match state.* When no aggregator nodes ($?^+$/?*) are used, the distance between these two region-IDs occurring is the number of nodes between the variable instances in the query.

One possible way for software systems to handle this would be to store, at each variable node (*in a matched state*), all the region-IDs encountered throughout the stream. A post-processing step would carefully intersect, for each variable, all stored region-IDs vectors. While that is a valid approach, storing region-IDs for each variable node of each pattern query is problematic as streams are longer. Furthermore, this is not needed unless aggregator nodes ($?^+$/$?^*$) occur in between variable occurrences; these cases are detailed in Sections 4.4.5 and 4.4.6. As FPGAs allow the deploying of custom matching engines for each pattern, matching pattern queries at streaming (no-stall) mode can be achieved here, with no post processing.



(a)



(b)

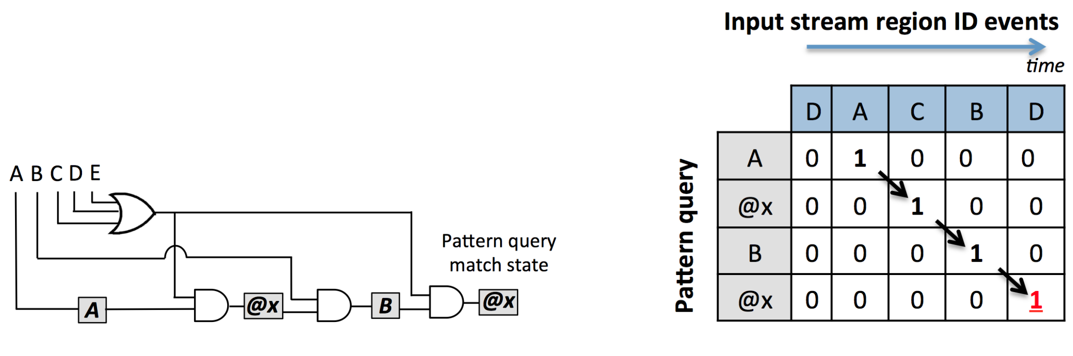
Figure 4.4: Query matching engines respective to the pattern query *A.@x.B.@x*, (a) witout and (b) with a region set constraint $\{C,D,E\}$ on *@x*. To handle variables in hardware, the first instance of a given variable in a query forwards, alongside the incoming match state, (a) the event detector's output encoded (multi-bit) region-ID, and (b) a one-hot signal consisting of bits respective to each region in the set of the variable. Every later instance of that variable in the query (here, the last query node) would match the event detector's ((a) encoded, and (b) multiple decoded) region-ID to the forwarded region-ID. If these match, then the region-ID is again forwarded, and the variable instance indicates a *matched* state.

To handle variables in hardware, the first instance of a given variable in a query would forward the event detector's output encoded (multi-bit) region-ID alongside the incoming match state (see the second node in Figure 4.4(a)). Some cycles later (depending on the location of variable instances in the pattern), every instance of that variable in the query would match the event detector's region-ID to the forwarded region-ID. If these match, then the region-ID is again forwarded, and the variable instance indicates a *matched* state. Stated in other terms, at a variable node (instance) in a query, a match state is indicated if the current region was encountered earlier (given a fixed implied distance), and all match state propagation checks in between were valid (implying the distance).

Note that an encoded region-ID is used since it is smaller in bit size than a decoded ID, and any region can potentially satisfy the query variable (variables are essentially a subset of wildcards). Also note that non-variable predicates buffer the forwarded region-ID, though no manipulation of the latter is required. Also note that one set of region-ID buffers is required per variable, starting from the first occurrence of that variable.

The same solution is applicable to pattern queries containing variables with region sets. Figure 4.4(b) shows the matching logic for the pattern *A.@x.B.@x* where *@x* is constrained by the regions $\{C,D,E\}$. Here, instead of storing the encoded region-ID in the variable buffers, the latter would hold, for each region in the set, a single bit. At the first occurrence of a variable, the buffer holds a one-hot vector, because input stream events are relative to one region only. Upon later instances of that variable, AND-ing the incoming region set buffer with specific bits of the region-ID decoder output will help indicating for which regions (if any) the pattern matches.

The above approach is similar to replicating the matching engine for each region in the variable region set constraint. For instance, the query in Figure 4.4(b) can be seen as three queries, namely *A.C.B.C, A.D.B.D* and *A.E.B.E.* However, the above approach offers much better scalability when multiple variables are used per pattern: replicating the pattern for each combination of variable regions would result in an exponential increase in resource utilization versus employing the aforementioned style of propagating buffers. Another advantage of the propagating region set variable buffers is detailed below, when dealing with wildstar/wildplus pattern predicates.



(a) (b)

Figure 4.5: (a) query matching engine respective to the pattern query *A.@x.B.@x* {*@x:C,D,E*}, such that the variable region set constraint is implemented as a “*relaxed*” OR. This relaxation helps save considerable hardware resources (compare to Figure 4.4(b)). (b) an event-by-event overview of the matching of the query resulting in a false positive, due to the OR-based implementation of the variable region set constraint.

An alternative “*relaxed*” implementation of the variable region set constraint is described next, with the goal of saving considerable hardware resources, though at the expense of introducing false positives. Instead of keeping a propagating buffer holding information on each region in the set, the match state can be updated if *any* of the regions in the set are decoded, using a simple OR-gate. Figure 4.5(a) depicts the gate-level implementation of the query *A.@x.B.@x* {*@x:C,D,E*}, such that the variable region set constraint is implemented as an OR. Thus, history keeping is minimized, as no exact region information is kept per variable. While this mechanism introduces false positives (as described in Figure 4.5(b)), the latter can be tolerable, depending on the application. Otherwise, a post-processing software step can be performed only on the patterns marked as matched by the FPGA hardware. This approach however helps fitting substantially more query engines on the FPGA, a benefit accentuated as the number of variables and the variable region sets' size increase.

4.4.5 Evaluating Patterns with a Single Variable and with Wildstar/Wildplus Predicates

The remainder of this discussion is applicable to both wildplus and wildstar query nodes.

As detailed earlier (Figure 4.3(a)), wildplus nodes act as **aggregator** nodes. When no variables are used, the only propagating information across nodes is a single bit value. In that case, a simple OR gate would suffice for aggregation (state saving).

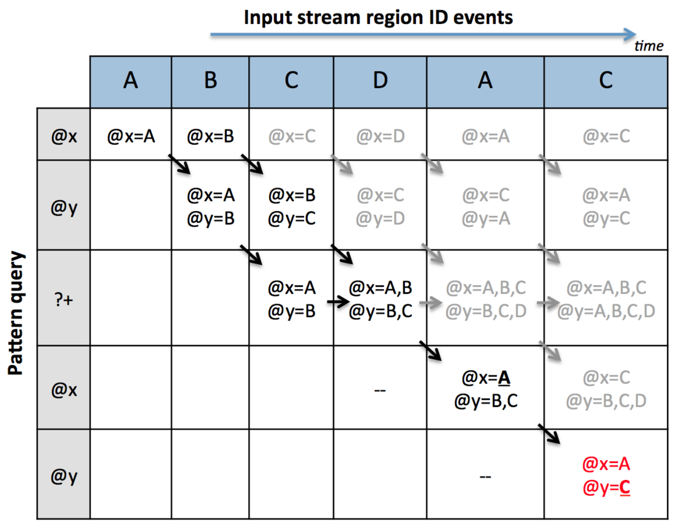
When a wildplus predicate is located in between two instances of a variable, all values of the region-ID buffer should be stored, and forwarded to the next stages (nodes). Keeping that history is required in order not to result in false negatives. However, due to performance and resource utilization constraints, storing all that history is not desired.

Using variable region set constraints, this limitation can be overcome by simply OR-ing the propagating buffer similarly to the match state buffer. This approach would store the information needed, and no history is lost. No false positives would result, and pattern evaluation is achieved at streaming mode.

4.4.6 Evaluating Patterns with Multiple Variables and with Wildstar/Wildplus Predicates

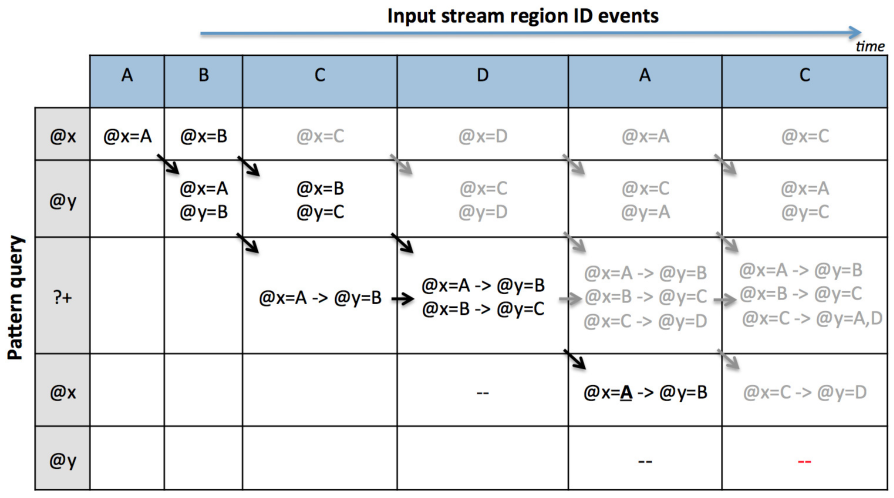
When more than one variable is used in a pattern, and with wildplus nodes in between instances of both these variables, the previous mechanism can lead to false-positives, as even more state should be saved than discussed earlier.

Figure 4.6(a) shows an event-by-event example of a pattern matching resulting in a false positive. Each cell in the grid holds the values stored inside each respective variable buffer. Buffers for the variable @x are used at each pattern node, whereas buffers for the variable @y span from the second pattern node (i.e. the first @y node), up to the last pattern node.



	A	B	C	D	A	C
@x	@x=A	@x=B	@x=C	@x=D	@x=A	@x=C
@y		@x=A @y=B	@x=B @y=C	@x=C @y=D	@x=C @y=A	@x=A @y=C
?+			@x=A @y=B	@x=A,B @y=B,C	@x=A,B,C @y=B,C,D	@x=A,B,C @y=A,B,C,D
@x				--	@x=**A** @y=B,C	@x=C @y=B,C,D
@y					--	@x=A @y=**C**

(a)



	A	B	C	D	A	C
@x	@x=A	@x=B	@x=C	@x=D	@x=A	@x=C
@y		@x=A @y=B	@x=B @y=C	@x=C @y=D	@x=C @y=A	@x=A @y=C
?+			@x=A -> @y=B	@x=A -> @y=B @x=B -> @y=C	@x=A -> @y=B @x=B -> @y=C @x=C -> @y=D	@x=A -> @y=B @x=B -> @y=C @x=C -> @y=A,D
@x				--	@x=**A** -> @y=B	@x=C -> @y=D
@y					--	--

(b)

Figure 4.6: Event-by-event matching of the pattern query *@x.@y.?*$^+$*.@x.@y* {*@x: A, B, C, D*} {*@y: A, B, C, D*}. The resulting match in (a) is a false positive; whereas enough state is saved in (b) at the aggregator node (?$^+$) to eliminate that false positive.

As described earlier, the wildplus node is the only node in the pattern query allowing horizontal propagation of match states. This is due to the nature of wildplus nodes which hold a *matched* state. As the variable buffers are OR-ed at that wildplus node, they will store the information of the union of all variable buffers encountered at that node. Looking at the $?^+$ row, notice that the variable buffers for both @x and @y hold an increasing number of regions. That level of stored information is not sufficient, as it will be shortly shown to result in a false positive.

Upon the D event, both variable buffers did not propagate to the second instance of @x. That is because the @x variable buffer does not reflect that the previous instance of @x held the value of D (yet). However, on the next event (A), the variable buffers propagated, and the @x variable buffer was masked with the event region. Hence, B was removed from the @x variable buffer. The @y variable buffer remains un-modified, since the @x node is not allowed to modify it.

Finally, at the last event (C), focusing at the second instance of @y (i.e. the last pattern predicate), a match is shown for *@x=A* and *@y=C*. While @x and @y did hold these values at some point, looking closer at the input stream, A and C were initially separated by B, though the query requires that the distance between @x and @y is 1 (back-to-back regions visited).

In order not to result in false positives, the level of history kept at the aggregator node has to be increased. Instead of only storing the union of all variable buffers, the information at the wildplus node should be the set of all variable buffers encountered. To reduce storage, that solution can be simplified such that, for each @x variable value, a list of all corresponding @y values are stored. Figure 4.6(b) depicts this solution. Focusing on the aggregator row, every value of @x is associated with a list of @y values. These can be deduced from the propagating variable buffers into the wildplus node. Note that @x=A is associated with @y=B. Therefore, the tuple *@x=A, @y=C* cannot result in a match, as is the case in Figure 4.6(a).

Nonetheless, implementing this solution in hardware is extremely costly in terms of resource utilization (and impact on the critical path/performance), especially with larger region sets and more variables per pattern. Furthermore, this solution does not scale with more variables, and does not hold with more aggregator nodes.

Another approach to eliminate false positives in such cases is a brute-force implementation of each query using all variable region-set combinations. For instance, the query *@x.@y.?$^+$.@x.@y* {*@x:A,B*} {*@y:C,D*} can be implemented as four simpler queries, namely:

- *A.C.?$^+$.A.C.*
- *A.D.?$^+$.A.D.*
- *B.C.?$^+$.B.C.*
- *B.D.?$^+$.B.D.*

This approach is encouraging when the number of variables and the size of the region sets is relatively small. Otherwise, the implied resource utilization explodes, even though each query is built using simple matching engines (no propagating variable buffers). Nonetheless, the common prefix optimization helps with the scalability.

A study on the resulting false positives versus resource utilization is performed in Section **??** to better evaluate the benefits of each approach. To recapitulate, when pattern queries make use of two or more variables, and with an aggregator node in between the occurrences of these variables, the proposed approaches are:

- Making use of propagating variable buffers. This approach results in the least false positives.
- Implementing region set constraints as an OR. The number of false positives here is a superset of the above case, and resource utilization is minimal. False positives are a superset, since the condition (OR check) to allow a match to propagate through a variable node is a superset of the first approach's variable node conditions (propagating buffers).
- A brute-force mapping of each query as the combination of all variable region-sets. This approach has no false positives, but does not scale well with more variables and larger region sets.

4.5 Experimental Evaluation

In this section we provide an extensive experimental evaluation of the proposed hardware architecture. The datasets used are first described, followed by the experimental setup. A thorough design space exploration on the proposed architecture is presented, alongside a study on matching accuracy. Performance measurements are then provided.

4.5.1 Dataset Description

In our experimental evaluation, we use four real trajectory datasets. The first two datasets are the Truck and Buses trajectorial data from [1]. Both datasets represent moving objects in the metropolitan area of Athens, Greece. The Truck dataset has 276 trajectories of 50 trucks where the longest trajectory timestamp is 13,540 time units. The Buses dataset has 145 trajectories of school buses with maximum timestamp 992.

The third dataset consists of GPS coordinates of 483 cabs operating in the San Francisco area [67] collected over a period of almost a month. The fourth dataset, Geolife [92, 91], was collected as GPS trajectory data in a period of over three years. The dataset contains 17,621 trajectories with a total distance of about 1.2 million kilometers and a total duration of more than 48,000 hours.

4.5.2 Setup

For simplicity of the experimental evaluation, we partition the spatial domain in uniform grid sizes. These grid cells become the alphabet for our queries. In order to generate relevant query patterns, we randomly sample and fragment the trajectories. The length and location of each fragment are randomly chosen. These fragments are then concatenated to create a query. We generated queries with different number of predicates, variables, and wildcards. The location of each variable and wildcards inside the query were randomly chosen.

Our FPGA platform consists of a Pico M-501 board connected to an Intel Xeon processor via 8 lanes of PCI-e Gen. 2 [66]. 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.

The hardware engines communicate 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 engine is independent of the global clock. The PCIe interfaces incur an overhead of $\approx$8% of available FPGA resources.

The RAM on the FPGA board is not residing 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.

Xilinx ISE 14 is used for synthesis and place-and-route. Default settings are set.

4.5.3 Design Space Exploration

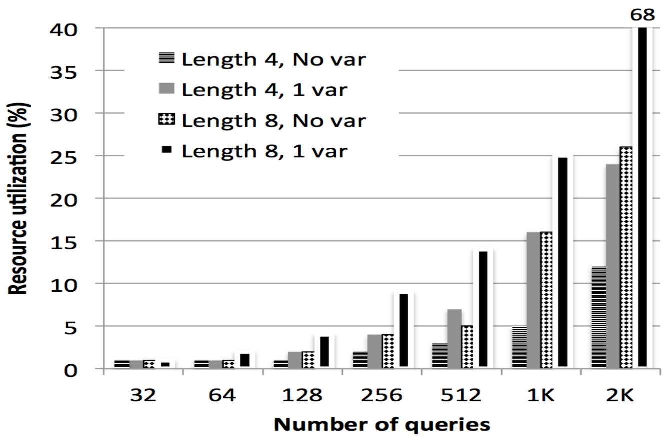
A study on the resource utilization and achievable performance (throughput) of the hardware engines follows.

Figure 4.7 shows (a) the resource utilization and (b) respective frequencies of the hardware engines, such that the number of queries is doubled (32, 64, 128 ... 2K), the query length is doubled (4, 8 nodes) and variables usage is varied: when used, one variable with a region set of 5 regions is assumed.

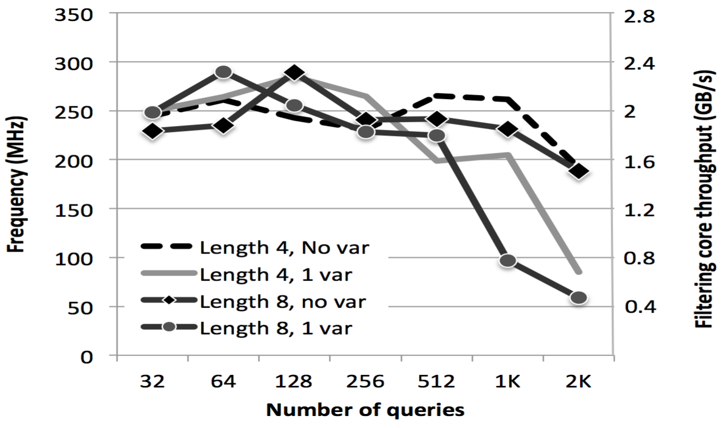
As the query compiler applies the common prefix optimization, and further resource sharing techniques are exercised by the synthesis/place-and-route tools, resource utilization does not double as the number of queries is doubled. Rather, a penalty of approximately 70% occurs.

Similarly, as the query length is doubled, an average increase of 80% in resources is found. However, adding one variable to each query results in doubling resource utilization on average. Note that the propagating buffer approach is employed for variable matching, and that these buffers propagate from the first occurrence of the variable, to the last.

Overall, up to several thousands of query matching engines can fit on the target Xilinx V6LX240T FPGA, a mid- to low-size FPGA.



(a)



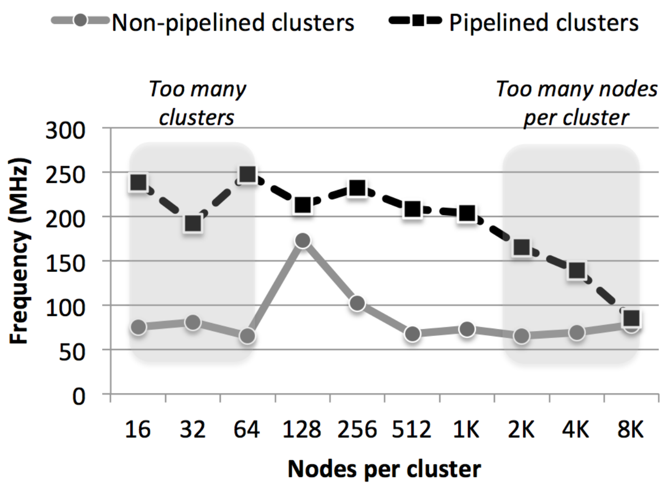
(b)

Figure 4.7: (a) resource utilization and (b) respective frequencies/throughput of the hardware engines, such that the number of queries is doubled, the query length is doubled, and variable usage is varied. When used, one variable of region set 5 regions is assumed. Results are shown for a Xilinx V6LX240T FPGA.

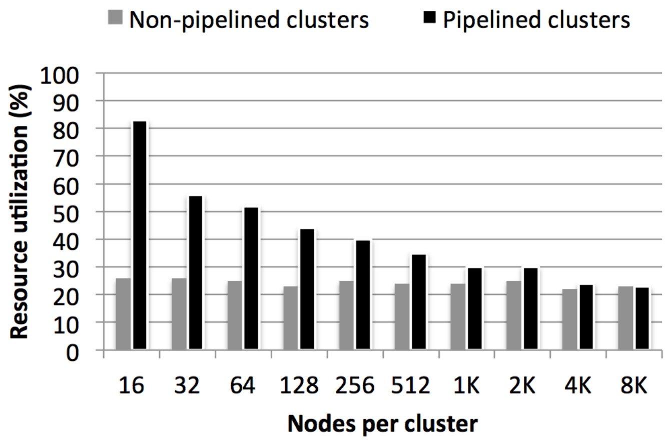
While these numbers address the scalability of the proposed matching engines, Figure 4.7(b) details the respective achievable performance in terms of:

- Operational frequency (MHz): measured as a function of the critical path, i.e. the longest wire connection of the FPGA circuit. This number is obtained post the place-and-route process of the FPGA tools.

- Throughput (GB/s): as the query matching engines process one ⟨ *region ID, timestamp* ⟩ pair per hardware cycle, the FPGA throughput can be deduced from the



(a)



(b)

Figure 4.8: (a) the achievable frequencies and (b) corresponding resource utilizations as a result of the clustering of 2K length 4 queries, with one variable (region set size = 5); results are shown for a Xilinx V6LX240T FPGA. We refer to a region ID decoder and its connected queries as a **cluster**. Two clustering approaches are considered; in the first, all clusters receive the input stream simultaneously (non-pipelined clusters); in the second, clusters are pipelined, such that each cluster forwards the input stream to the next.

circuit's operational frequency, given that the size of each input pair is 8 bytes (2 integers). Nonetheless, this computed throughput is respective to the FPGA circuitry, and might not reflect the end-to-end (CPU-FPGA and back) performance, which is platform dependent. End-to-end measurements are available further below.

As the number of queries increases, frequency/throughput is initially around the 250MHz/2GBs mark. Fluctuations are due to the heuristic-based nature of the FPGA tools, though generally a trend is deduced. As the number of queries becomes too large, frequency drops considerably for queries with variables. The drop is not as steep for queries with no variables; the reason being that queries with variables can be thought of as longer queries (due to the propagating buffers). This drop in frequency occurs because of the large fan-out from the region ID decoder to the many sinks, being the query nodes and propagating buffers.

Replicating the region ID decoder (and event detector) helps reducing fan-out, and will potentially eliminate it. Each region ID decoder is then connected to a set of queries. We refer to a region ID decoder and its connected queries as a **cluster**. Note that each query belongs to exactly one cluster. The query compiler is developed to take as input parameter the cluster size, as a function of query nodes.

Figure 4.8 shows (a) the achievable frequencies and (b) corresponding resource utilizations as a result of the clustering of 2K length 4 queries, with one variable (region set size = 5). Two clustering approaches are considered; in the first, all clusters receive the input stream simultaneously (non-pipelined clusters); in the second, clusters are pipelined, such that each cluster forwards the input stream to the next. Advantages (and disadvantages) of the latter are presented below.

Focusing on Figure 4.8(a), the number of nodes per cluster is doubled, starting from 16, and up to the maximum of 8K (due to the set of 2K queries of length 4 each). The frequency of non-pipelined clusters starts off as low, then exhibits an increase towards clusters of size 128 nodes, to then drop back to low levels. On the other hand, the frequency of pipelined clusters starts off as high with small clusters, then only gradually drops when clusters become too large.

For a fixed number of queries (here 2K), smaller clusters simply means more clusters. This translates into another performance problem, where a fan-out is created from the input stream to the clusters' region ID decoders. Since pipelined clusters forward the input stream to adjacent clusters, this issue does not apply, which can be seen in Figure 4.8(a) where the frequency starts off as high for pipelined clusters. However, when clusters become too large, both clustering approaches become inefficient. The frequency peak of non-pipelined clusters occurs outside of the "excessive load" regions (too many clusters, too many nodes per cluster), such that the synthesis/place-and-route FPGA tools were able to take advantage of the combination of the number of clusters and cluster size. This peak can only occur here.

The resulting area (%) due to clustering is shown Figure 4.8(b). With non-pipelined clusters, the area is almost constant, as the FPGA synthesis/place-and-route tools are able to detect shared resources across clusters. On the other hand, as each pipelined cluster receive its input in a different cycle from other clusters, the FPGA tools are not able to optimize across clusters. This leads to resource blow-out, when too many clusters are deployed. Nevertheless, as deduced from Figure 4.8(a) (and supported by Figure 4.7(b)), clusters need not hold less than 1K or even 512 nodes. This in turn limits the clustering penalty on resource utilization to a minimum, while still achieving good performance.

4.5.4 Query Engine Implementations and False Positives

As described in Sections 4.4.4 through 4.4.6, a query holding variables can be evaluated in one of three ways, namely (1) implementing the region set constraints as ORs (resulting in most false positives); (2) making use of propagating buffers (false

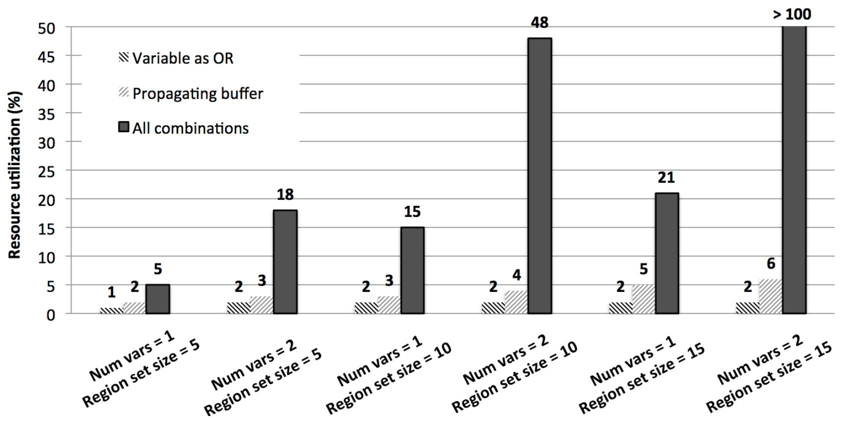


Figure 4.9: Scalability of the each of the following three implementations of 100 length 6 queries holding variabes: (1) **Variable as OR** implementing the region set constraints as ORs (resulting in most false positives); (2) **Propagating buffer** making use of propagating buffers (false positives arise only when using multiple variables alongside wildstar/wildplus nodes); (3) **All combinations** a brute-force mapping of each query as the combination of all variable region sets (no false positives). Results are shown for a target Xilinx V6LX240T FPGA.

positives arise only when using multiple variables alongside wildstar/wildplus nodes); (3) a brute-force mapping of each query as the combination of all variable region sets (no false positives). A study is presented next on the scalability of each approach as the number of variables and the region set size are increased.

Figure 4.9 illustrates the resource utilization of 100 length 6 queries holding variables, implemented in each of the aforementioned three approaches. The varied factors are the number of variables in each query patterns, and the respective region set size.

When implementing a *variable as OR*, each variable node is replaced with a simpler OR node. Thus, as expected (see Figure 4.9), increasing the number of variables has almost no effect on resource utilization. The same applies to increasing the region set size.

On the other hand, the *propagating buffer* technique starts off as utilizing slightly less than double the resources of the *variable as OR* approach. Furthermore,

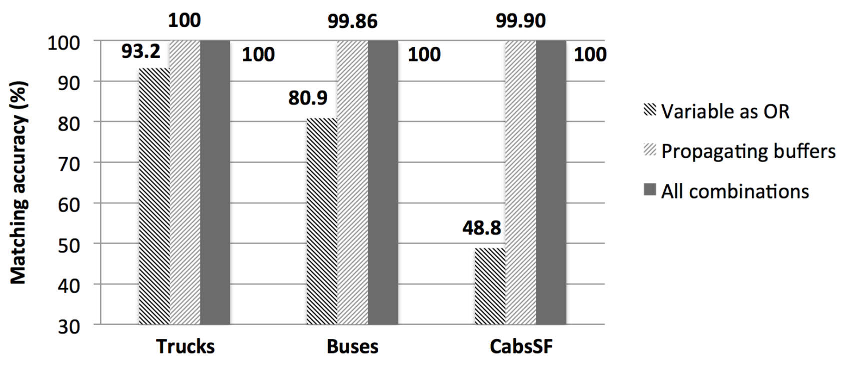


Figure 4.10: Matching accuracy (100-false positives %) for each implementation of 100 long queries, over three datasets, namely *Trucks*, *Buses* and *CabsSF*. Queries are synthetic, not biased, generated using our query generator tool. They contain two variables each, as well as one or more aggregator (?*/?+) nodes.

doubling the region set size results in a 50% area penalty. Doubling the number of variables per pattern query exhibits similar behavior.

Finally, when transforming a query into a set of queries based on *all combinations* of the region sets, resource utilization starts off as more than double that of the *propagating buffer* technique. Doubling the number of variables naturally has a steeper effect than doubling the region set size on resource utilization. Note that the common prefix optimization helps with the scalability this approach. Nonetheless, when using two variables of with 15 regions in the set, the resulting circuitry did not fit on the FPGA. Practically, it is best to make use of this approach for critical pattern queries where false positives are not tolerated.

A study on the resulting false positives of each of the three query engine implementations is provided in Figure 4.10, where the matching accuracy (100-false positives %) is recorded for each implementation of 100 long queries, over three datasets, namely *Trucks*, *Buses* and *CabsSF*. Queries are synthetic, not biased, generated using our query generator tool. They contain two variables each, as well as one or more aggregator

(?*/?+) nodes. Note that the *Propagating buffers* approach does not result in any false positives, unless multiple variables are used alongside aggregators.

The *All combinations* approach results in no false positives, by design. However, while the Variable as OR technique results in the most false positives (as expected), the matching accuracy varies from high (93.2%), to somewhat low (48.8%). On the other hand, matching accuracy is close to perfect ($> 99.8\%$) for the *Propagating buffers* implementation, even as false positives increase as a result of the *Variable as OR* implementation. No false positives were recorded on the *Trucks* dataset when making use of the emphpropagating buffers.

While the mileage of the *Variable as OR* implementation may vary, its scalability is key. Even when false positives are not tolerable, query matching engines can employ this technique, where the FPGA would be used as a pre-processing step with the goal of reducing the query set. The same applies for the *propagating buffers* implementation technique, where the query set would be reduced the most. Since the performance of CPU-based software approaches scales linearly with the number of pattern queries, reducing the query set has desirable advantages, especially that the time required for this pre-processing FPGA step is negligible.

4.5.5 Performance Evaluation

Figure 4.11 shows the end-to-end (CPU-RAM to FPGA and back) throughput of length 4 queries with 1 variable. The throughput of the FPGA filtering core is drawn in red. Throughput is lower from the FPGA filtering core for smaller trajectory files since steady state is not reached, and communication setup penalty is not hidden. For larger files, throughput is closer to the FPGA core's, while being limited by the physical

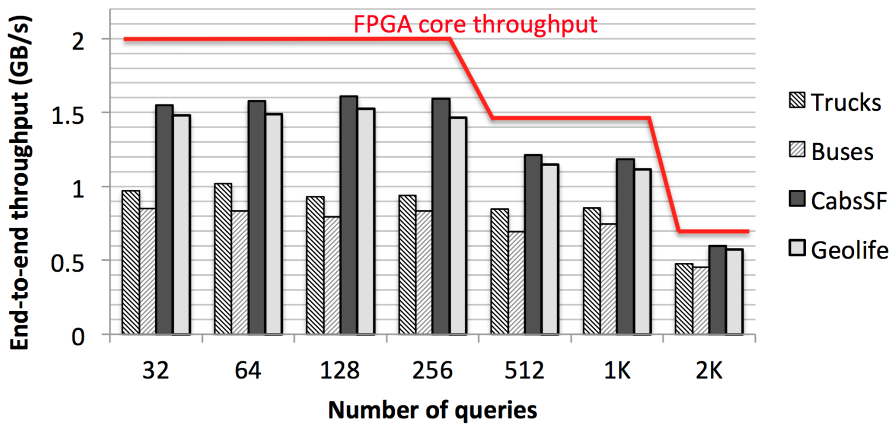


Figure 4.11: End-to-end (CPU-RAM to FPGA and back) throughput of length 4 queries with 1 variable. The throughput of the FPGA filtering core is drawn in red.

link's throughput. Furthermore, end-to-end performance includes the penalty of the drivers and buffer on both of the FPGA and CPU. Nonetheless, the throughput of the FPGA querying cores is independent of the trajectory file contents, as well as query structure (given a certain operational circuit frequency).

We ran through several issues with regards to the available *FlexTrack* code. Hence, software performance will be studied as part of our future work. Note that where the FPGA end-to-end execution time is in the milliseconds range, preliminary results show that software operates in the tens and hundreds of seconds range, and is greatly affected by the query structure and document contents. Thus, the presented FPGA setup results in considerable speedup (several orders of magnitude) and benefits with respect to immunity against query/trajectory characteristics.

4.6 Conclusions

The wide and increasing availability of collected data in the form of trajectory has lead to research advances in behavioral aspects of the monitored subjects. Using trajectory data harvested by devices, such as GPS and mobile devices, complex pattern queries can be posed to select trajectories based on specific events of interest. However, as the complexity of the posed queries increases, so do computational requirements, which are not easily met using traditional CPU-based software platforms.

In this work, the first proof-of-concept study on FPGA-based architectures for matching variable-enhanced complex patterns is presented, with a focus on stream-mode (single pass) filtering. A tool for automatically generating hardware constructs using a set of queries is presented, abstracting away ramifications of hardware code development and deployment. A thorough design space exploration of the hardware architectures shows that the presented solution offers good scalability, fitting thousands of query matching engines on a Xilinx V6LX240T FPGA, a mid- to low-size FPGA. Increasing the number of variables and wildcards is shown to have linear effect on the resulting circuit size, and negligible on performance. That is unlike CPU-based solutions, where performance is greatly affected from such query characteristics.

When handling queries with (a) no variables, (b) one variable, or (c) no wildcards with two or more variables, the proposed hardware architecture is able to process the trajectory data in a single pass. When two or more variables are used alongside wildcards, the proposed solution will result in false positives, though these are minimal in practice. Nonetheless, a no-false-positive solution is proposed, though being limited in scalability.

As part of our future work, we will be enhancing the proposed framework to allow online query updates. The deployed generic query engines would support ”any” query structure and node values. A stream of bits forwarded to the FPGA would program the connections between deployed query nodes. This approach should not be confused with Dynamic Partial Reconfiguration (DPR), where the bit configuration of the FPGA itself is updated.

Chapter 5

Golomb-Rice Integer Decompression

5.1 Introduction

The goal of data compression techniques is to reduce the storage space and/or increase the effective throughput from the data source (such as a storage medium). Other critical performance factors considered include code complexity and memory offloading requirements. Various compression techniques can be combined and are tailored to perform best within certain classes of applications, where assumptions on the data (format, range, occurrence, etc) hold. Examples of the latter are Run-Length Encoding (RLE) [23] as used by image compression (JPEG), and Lempel-Ziv-Welsch [84] (LZW) for text data.

Compression techniques can be mainly categorized as being lossy or lossless. Generally, lossy techniques result in a higher compression ratio, and/or a faster processing (compression/decompression) time. Lossy techniques are hence preferred when the original data does not have to be exactly retrieved from the compressed data, and differences with the original data are tolerable or non-noticeable (such is the case with audio, video, etc).

Moreover, compression techniques can be further classified as being bit-wise or byte-wise. Byte-wise (or byte-aligned, byte-granularity) approaches typically result in a lower compression ratio due to the coarser granularity, but offer a considerably higher compression/decompression throughput.

This chapter focuses on the decompression of integers compressed using the lossless bit-wise Golomb-Rice [23, 83] (GR) entropy method. GR compression is designed to achieve high compression ratios on input streams with small integer ranges [51]; it is deployed in several applications, such as image compression [51, 36, 41, 79, 34, 42], audio compression [44, 22], as well as the compression of streams of inverted indices [48, 89, 90, 72], and ECG signals [7, 12, 50]. Inverted indexes require very fast processing, and operate under low timing budgets as they are utilized in the querying of high-volume data, as in (web) search engines [93]; however, even though GR offers high compression ratios, other approaches are preferred due to the gap in decompression performance [90]. Similarly, with the augmented resolution standards on video processing and displays (Full-HD, Quad Full-HD), faster decompression is a must. Finally, the complex processing of the increasing amounts of ECG data can be further reduced using high-performance decoders, with decompression being a first step once data is received. In all of the aforementioned applications, inefficient decompression limits the input throughput to the computational pipelines.

We present a novel highly-parallel hardware core capable of decompressing streams of GR-coded integers at wire speed with constant throughput, operating on *raw unmodified* GR data. To the best of our knowledge, hardware and software (CPU-based) GR decoders assuming unmodified GR data operate bit-serially on the compressed stream, which highly bounds the achievable decompression speeds. On the other hand, modifications to the algorithm and assumptions on the compressed format allow the application of efficient optimizations [36, 90, 41], though the limiting assumptions cannot be generalized. The proposed no-stall hardware solution is shown to outperform state-of-the-art software and hardware approaches, and achieves up to 7.8 Gbps sustained decompression throughput while occupying 10% of the available resources on a Xilinx Virtex 6 LX240T, a mid- to low-size FPGA.

5.2 Golomb-Rice Compression Overview

5.2.1 Algorithm Description

Golomb-Rice, or simply GR, or Rice coding is a lossless bit-granularity integer compression approach, which performs best with datasets where the probability of occurrence of small numbers far exceeds that of large values. It is shown that for such input sets, GR coding has compression efficiency close to the more complex arithmetic coding, and comparable to Huffman coding, while no code tables are required, formerly a potential bottleneck in the hardware compression/decompression process [51].

In Golomb coding, given a divisor d, each input integer is encoded into two parts: a unary quotient q, and a binary remainder r. GR coding is a subset of Golomb coding, restricting divisors to powers of two. This implies that for a give d, the number of bits required to encode the remainder portion is fixed to $k = log_2(d)$ bits (otherwise variable with Golomb coding). This simple assumption/restriction has a practical negligible negative impact on compression ratio, and greatly simplifies the encoding/decoding process, by allowing the use of simple shift operations instead of the more complex division. Good choices of d (hence k) greatly affect the compression ratio, and d is generally picked as factor of the average of the input integer set [93, 90]; this discussion is however out of the scope of this dissertation.

Resulting from GR coding is the fact that integers smaller than d are encoded using $k + 1$ bits, being a single unary bit and the remainder bits; furthermore, the compression of (infrequent) large numbers can result in more bits than the original uncompressed number, due to the inefficient coding of the unary quotient.

5.2.2 Parallelism and Challenges

GR-coded streams offer great opportunity for parallel decompression, since integers can be decoded independently. However, finding the end of a compressed integer and the start of the next is non-trivial, and cannot be determined without knowledge of all prior stream contents.

Figure 5.1 shows a snapshot of a chunk of bits in a GR-encoded stream, with $k = 3$ (remainder) bits. Integers are coded as the unary quotient (variable number of 1's followed by a 0) followed by the binary remainder bits, from right to left.

The first 0 bit in Figure 5.1 starting from the right (underlined) could reflect the last unary quotient bit, which would result in the reconstruction of integers i and *i+1* as illustrated under the stream (note that unary values end with a 0 bit). On the other hand, this 0 bit could reflect the second remainder bit, leading to integers i and *i+1* as depicted above the stream.

In other words, a decoder processing an N-bit chunk of compressed data per cycle cannot assume independent chunks, since compressed integers are not contained within these chunks (integers span across chunks). Furthermore, in order to process a given N-bit chunk, the decoder has to process all previous compressed data in the stream.

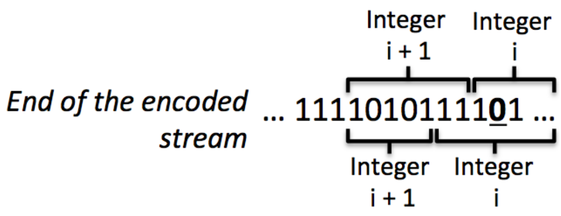


Figure 5.1: Snapshot of a chunk of bits in a GR-encoded stream. Assuming integers are coded as the unary quotient followed by the $k = 3$ bits binary remainder (from right to left), the above chunk can be decompressed in several ways (two of which are shown); the correct decoding cannot be determined without knowledge of all the prior contents of the encoded stream. © 2013 IEEE.

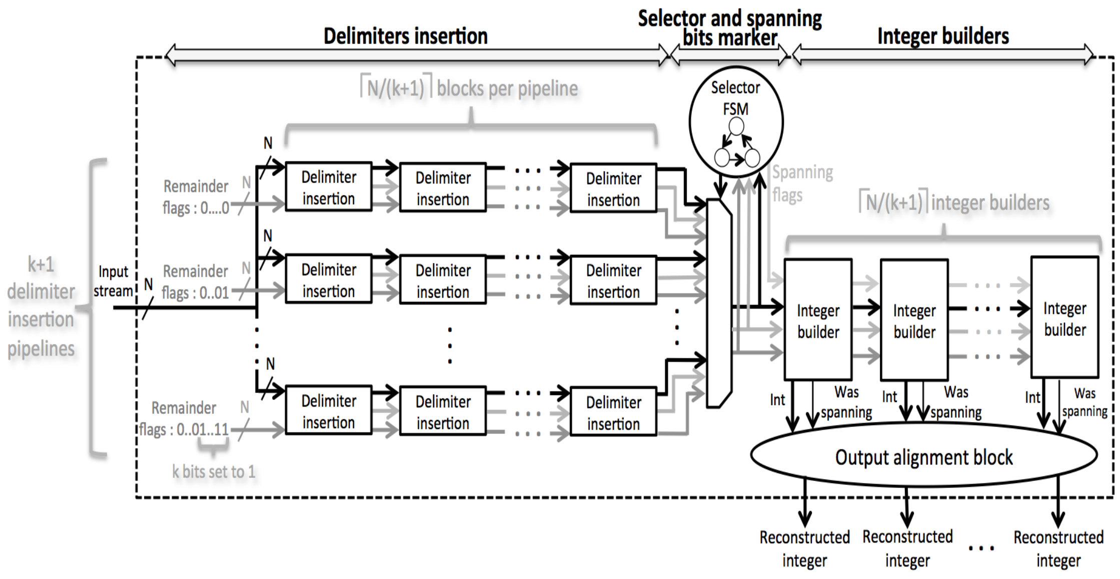


Figure 5.2: High-level overview of the no-stall GR decompression architecture, capable of sustaining a processing rate of N bits per cycle. © 2013 IEEE.

To the best of our knowledge, no (hardware and/or software) approach in the literature allows the processing of stream chunks in parallel. Section 5.4 details our proposed mechanism which overcomes this challenge.

5.3 Related work

The FPGA-led performance boost up of compression/decompression has long been an active field of research, with the main focus on speeding up low-latency storage access [14, 74, 63]. In this section, we focus on providing an overview of the application and implementation of Golomb-Rice coding in several fields.

Inverted indexes: [48] and [72] provide an overview of inverted index querying, as well as the description and performance of several compression techniques, such as variable length integers, Elias Gamma, Delta coding and Golomb-Rice. The authors in [89, 90] provide a thorough performance and compression ratio study of several compression approaches, as applied to the inverted index problem. A novel compression technique is presented with focus on performance, combining PForDelta with GR coding. The intuition is to partition input integers into blocks, such that the compressed output of each block contains all remainder bits first, followed by the unary quotients. This allows the fast retrieval of the fixed-length remainders through simple lookups. The extraction of the contiguous variable-length unary quotient values is also achieved through smart lookup functions. However, this approach is limited by the scalability of the lookup (limited further performance enhancements), as well as the modifications to the input format required. All the implementations of this work are open source, and will be used in the performance evaluation of our proposed hardware architecture.

Image/video: [51] proposes the use of GR coding to compress the Discrete Cosine Transform (DCT) coefficients found in JPEG-LS (lossless). The authors in [41, 42] propose novel frame-recompression algorithms targeting MPEG-2 and H.264 videos, respectively. The MPEG-2 FPGA-based decoder makes use several assumptions to implement a parallel GR decoder; for instance, integers are compressed into words of fixed size 21 bits, containing exactly 7 integers each; furthermore (importantly), the boundary of compressed integers is fixed within these 21 bits, and a (small) maximum unary size is assumed. While these assumptions hold in this case, they are not characteristics of GR streams. Similarly, [36] presents a GR-based novel color image FPGA

CODEC. A parallel decoder is presented, assuming modified compressed GR format, as well as small independent words containing a fixed number of compressed integers each. The authors in [79] describe the hardware implementation of a novel proposed compression codec targeting advanced-HD video, utilizing GR coding. FELICS, a lossless image compression format utilizing GR coding was introduced in [34].

Audio: [44] describes the use of GR coding in the the lossless audio MPEG-LS format; similarly, the Free Lossless Audio Codec (FLAC) [22] uses GR compression internally.

ECG signals: [7] and [12] detail the compression of DCT coefficients in Electrocardiography (ECG) signals using the lossless GR method. The work in [50] describes the FPGA implementation of a multi-bit per cycle GR compressor of ECG signals. Note that compression is orthogonal to decompression (the problem studied in this work). The described FPGA decompressor operates in a bit-serial manner.

Miscellaneous: [49] thoroughly studies the adaptive combination of Run-Length to Golomb-Rice coding; the intuition is that unary quotients resulting from GR consist of long streams of 1's, and can be efficiently compressed with run-length encoding. The authors in [43] propose the use of GR coding in conjunction with A/D converters. A detailed CMOS-level implementation is provided, showing the ease and advantages of integration of A/D converters to GR encoders.

5.4 Hardware Golomb-Rice Decompression

In this section, an overview of how parallel GR decompression can be achieved is provided, followed by an in-depth description of the proposed parallel hardware GR decompression engine.

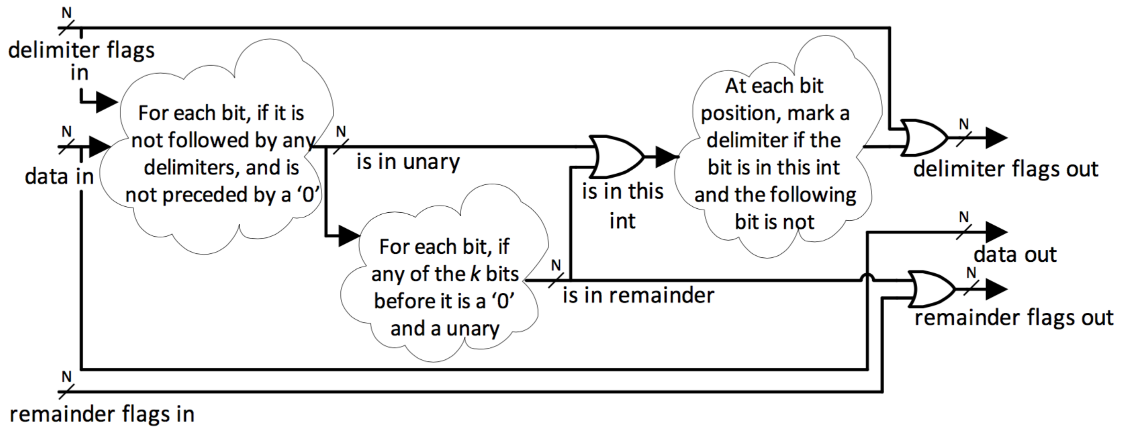

N
delimiter flags in
N
data in
For each bit, if it is not followed by any delimiters, and is not preceded by a '0'
N
is in unary
For each bit, if any of the k bits before it is a '0' and a unary
N
is in remainder
is in this int
At each bit position, mark a delimiter if the bit is in this int and the following bit is not
N
delimiter flags out
N
data out
N
remainder flags out
N
remainder flags in

Figure 5.3: Functionality and high-level implementation overview of a delimiter insertion block. Given a chunk of GR-encoded data, delimiter flags, and remainder flags, a delimiter insertion block is tasked with updating the input flags by marking the last bit (delimiter) and remainder bits of the next integer in the chunk, if any. © 2013 IEEE.

5.4.1 Parallel Extraction of Compressed Integers

As described in Section 5.2.2, a decoder cannot process an N-bit chunk of compressed data without any knowledge on prior contents of the compressed stream. Hence, processing an N-bit chunk should be delayed after all previous data is decoded. In reference to Figure 5.1, this would indicate the property (unary or remainder) of the highlighted 0 bit.

However, given that *k=3* (number of remainder bits), the highlighted 0 bit can only one of the following: (1) a unary bit, the last to be exact; (2) the first remainder bit; (3) the second remainder bit; and (4) finally, the third and last remainder bit. Note that option (2) is trivially dismissed since the previous bit is a 1, which cannot indicate the end of a unary (unaries end with a 0). However, for the sake of a generic example, any given (0 or 1) bit in a stream can have one of only *k+1* properties, as indicated earlier.

Assuming a decoder capable of processing a chunk of N GR-compressed bits in parallel: the i^{th} chunk of size N bits can be speculatively processed in *k+1* ways, where each would be an assumption on the property of the first bit in that chunk. Once the previous $i-1^{th}$ chunk is decoded, k computations are dismissed and one is committed. This will in turn allow the $i+1^{th}$ chunk to commit, and so on, for the remainder of the compressed stream.

Since GR coding mainly targets integers with small ranges, k is generally kept small; it is theoretically maximally less than the integer bit size (64 for double precision). Therefore, the *k+1* design space is limited. The choice of N depends on several physical constraints and performance requirements, as detailed in this section and Section 5.5.

Encoded integers potentially span across (two or more) chunks, either because their encoding starts towards the end of a chunk, and/or because the unary portion is long. This will require some data of consecutive chunks to be combined in order to reconstruct compressed integers.

5.4.2 No-Stall Architecture Overview

The remainder of this section details a parallel no-stall hardware GR decompression architecture based on the observations described above.

5.4.2.1 Delimiters Insertion

Figure 5.2 illustrates a high level overview of the proposed architecture, of which the first portion is delimiters insertion. As an N-bit chunk of compressed GR data is received, it is forwarded to *k+1* pipelines. Each of these pipelines holds an assumption on the input chunk, and receives masks (labeled remainder flags) indicating whether each chunk bit is a remainder or a unary. This mask naturally differs from one

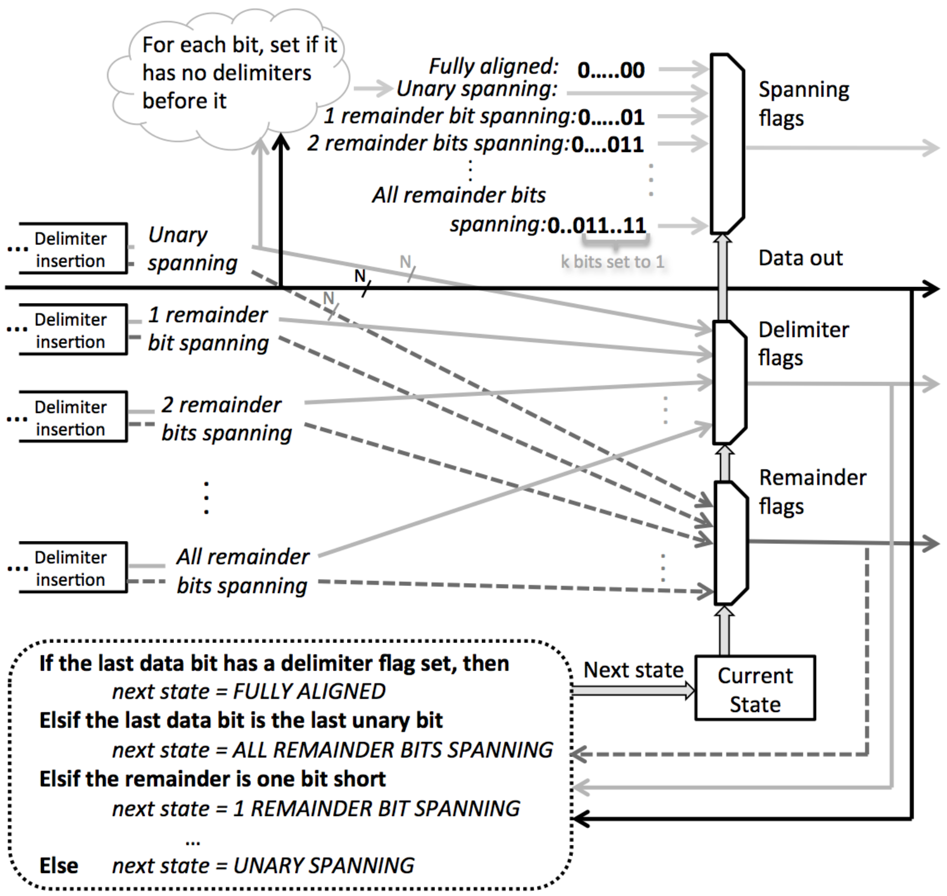

For each bit, set if it has no delimiters before it
Fully aligned: 0.....00
Unary spanning:
1 remainder bit spanning:0.....01
2 remainder bits spanning:0....011
All remainder bits spanning:0..011..11
k bits set to 1
Spanning flags
Data out
Delimiter flags
Remainder flags
Delimiter insertion
Unary spanning
Delimiter insertion
1 remainder bit spanning
Delimiter insertion
2 remainder bits spanning
Delimiter insertion
All remainder bits spanning
N
N
N
Next state
Current State
If the last data bit has a delimiter flag set, then
next state = FULLY ALIGNED
Elsif the last data bit is the last unary bit
next state = ALL REMAINDER BITS SPANNING
Elsif the remainder is one bit short
next state = 1 REMAINDER BIT SPANNING
...
Else next state = UNARY SPANNING

Figure 5.4: Functionality and high-level implementation overview of the *selector and spanning bits marker* stage. This stage is tasked with selecting the output of one of the delimiters insertion pipelines (left), based on knowledge of previously processed chunks (see the FSM transitions). All bits spanning into the current chunk are marked as such using the outputted *Spanning flag* bit vector (top). © 2013 IEEE.

pipeline to the next, as illustrated in Figure 5.2. The top pipeline assumes that the unary of the previous chunk spans into the received chunk (remainder flag is all 0's). The pipeline below it assumes that only one remainder bits spans (remainder flag of all 0's and a single 1); and so on until the bottom pipeline which assumes that all remainder bits span from the previous chunk (remainder flag of all 0's first, then k 1's). Given its respective assumption, each pipeline outputs the received data chunk, alongside a mask (flags) indicating all remainder bits in the received data chunk, as well as a mask indicating the delimiters (last bit) of each encoded integer in the chunk. These masks would allow the fast extraction of individual integers in a chunk, as detailed in later stages.

As depicted in Figure 5.2, each of the delimiters insertion pipeline consists of $\lceil \frac{N}{k+1} \rceil$ blocks, where the latter indicates the maximum number of encoded integers in a chunk (an encoded integer consists of at least k remainder bits and one unary bit). A ceiling notation is used to reflect the case of a compressed integer spanning into the next chunk, of which less than *k+1* bits are in this chunk.

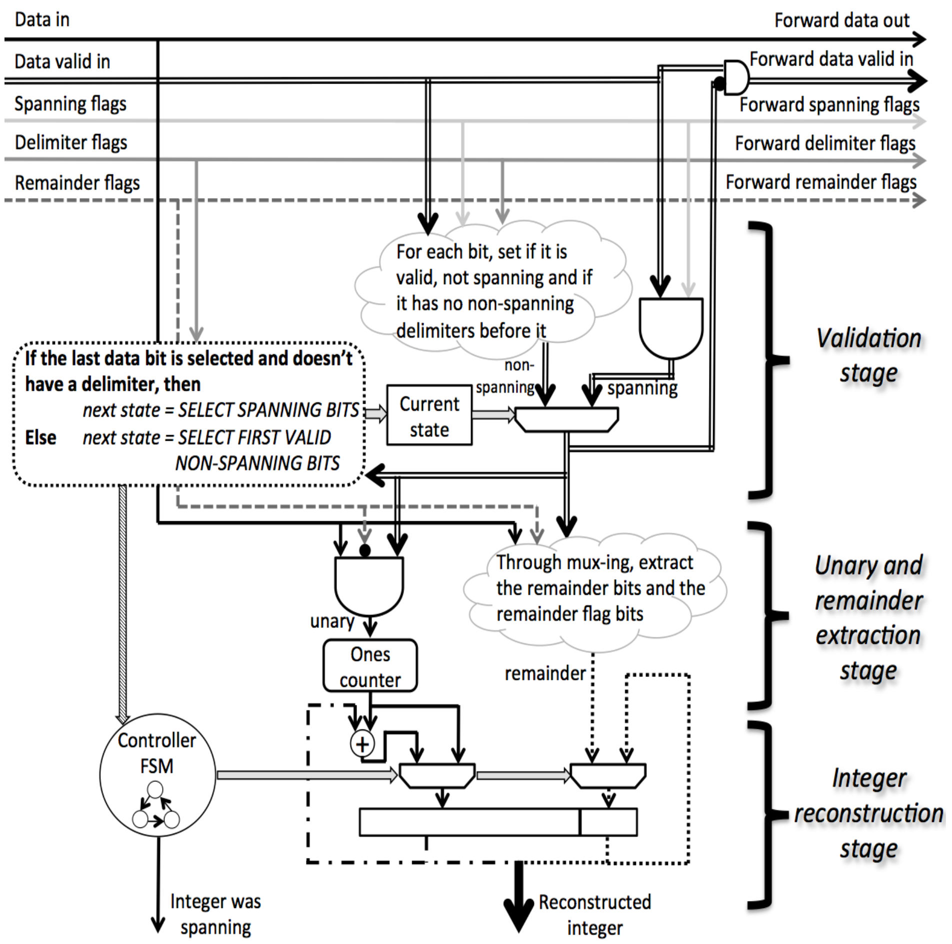
The functionality and high-level implementation overview of each block is shown in Figure 5.3. Given a chunk of GR-encoded data, delimiter flags, and remainder flags, a delimiter insertion block is tasked with updating the input flags by marking the last bit (delimiter) and remainder bits of the next integer in the chunk, if any. Having $\lceil \frac{N}{k+1} \rceil$ delimiter insertion blocks per pipeline guarantees that all delimiters and remainder bits in the input chunk will be marked.

Note that for the sake of delimiters insertion, the case of a chunk with no bits spanning into it is treated as a chunk with unary bits spanning into it (top-most delimiters insertion pipeline).

5.4.2.2 Selector (and Spanning Bits Marker) Stage

This is the next stage in the decompression pipeline, following the delimiters insertion. As the name indicates it, it is tasked with selecting the output of one of the delimiters insertion pipelines, based on knowledge of previously processed chunks.

The first chunk received by this stage is fully aligned, meaning that there are no bits spanning into it from the previous chunk. Therefore, the (flag vectors) output of the top-most delimiters insertion pipeline is selected, where the unary was assumed to



Data in
Forward data out
Data valid in
Forward data valid in
Spanning flags
Forward spanning flags
Delimiter flags
Forward delimiter flags
Remainder flags
Forward remainder flags
For each bit, set if it is valid, not spanning and if it has no non-spanning delimiters before it
If the last data bit is selected and doesn't have a delimiter, then next state = SELECT SPANNING BITS Else next state = SELECT FIRST VALID NON-SPANNING BITS
Current state
non-spanning
spanning
Validation stage
Through mux-ing, extract the remainder bits and the remainder flag bits
unary
Ones counter
remainder
Unary and remainder extraction stage
Controller FSM
Integer reconstruction stage
Integer was spanning
Reconstructed integer

Figure 5.5: Functionality and high-level implementation overview of the integer builder block. In the validation stage, one compressed integer from the input chunk is selected for reconstruction, then invalidated when passed to the following integer builder (if any). The unary quotient is converted to binary using a ones counter, whereas the remainder is simply multiplexed using the remainder flags. © 2013 IEEE.

be spanning. Then, by inspecting the last delimiter flag bit, the last data bit, and the last *k-1* remainder flag bits, the selector FSM can determine the state (hence multiplexer select value) respective to the next received chunk. This process is then repeated for every chunk received.

Figure 5.4 details the functionality and high-level implementation of the selector stage. The left hand side represents the output of each delimiters insertion pipeline, labeled with the initial assumption of every pipeline (unary spanning, one remainder bit spanning, etc). The output of the pipelines is multiplexed using the selector FSM.

The latter can be in one of $k+2$ states (fully aligned, unary spanning, one remainder bit spanning, two remainder bits spannings, etc), and the conditions for transitions across states are as shown in Figure 5.4.

The selector stage is further tasked with flagging spanning bits (bits in this chunk belonging to an integer starting in a previous chunk); the use of this flag will be clearer in the next decompression stages. Depending on the state of the current chunk, a set of spanning flags is chosen from, as shown in the top portion of Figure 5.4. In case the current chunk is fully aligned, then no bits span into it, and the spanning flag is set to all 0's. If remainder bits spans into the chunk, then depending on the number of remainder bits spanning, some of the least significant bits of the flag are set to 1. A constant flag exists for each of the aforementioned cases. On the other hand, when the (variable length) unary is spanning, the number of spanning bits is unknown, and has to be computed on the fly. As shown in Figure 5.4, a data bit is unary and spanning if there are no delimiters before it (which for every compressed data bit, is equivalent to looking for data bits with value 0 before it).

The output of the selector stage consists of a data chunk, delimiter, remainder and spanning flag vectors respective to each of the data bits. These signals are forwarded to a pipeline of integer builders.

5.4.2.3 Integer Builders

As shown in Figure 5.2 following the selector stage, a pipeline of $\lceil \frac{N}{k+1} \rceil$ integer builder blocks is deployed. The task of each integer builder is to reconstruct one of the integers in the chunk. Making use of $\lceil \frac{N}{k+1} \rceil$ blocks guarantees the handling of all potential integers in a chunk. Each integer builder selects one integer from the chunk for reconstruction, then invalidates it, and forwards the chunk with updated valid flags to the next integer builder. Invalidating an integer to be decoded ensures that no two integer builder process the same integer. Rules for choosing a compressed integer to decode are described below. Note that depending on the input stream, some integer builders are potentially idle in many cycles; that is because not all input chunks will contain bits of $\lceil \frac{N}{k+1} \rceil$ compressed integers.

Figure 5.5 details the implementation of an integer builder. The latter comprises of three main stages, namely the validation, unary and remainder extraction, and integer reconstruction stages.

In the validation stage, a compressed integer from the input chunk is selected, and its corresponding valid bit flags are cleared then forwarded alongside the data chunk to the next integer builder (if any). Initially, an integer builder picks the first non-spanning (valid) integer in the chunk. In case that integer is found to span (the last data bit is not delimited), then the integer builder will next select the spanning integer, in order to complete the reconstruction. This approach guarantees that no two integer builders will target the same compressed integer.

Once a compressed integer is selected (if spanning then through multiple cycles), the unary portion is converted back to binary through a one's counter. The remainder bits are selected using the remainder flag bits. The integer reconstruction stage handles the case of integers potentially spanning across multiple chunks.

The output of the integer builder block is a reconstructed integer (concatenated binary quotient and remainder), alongside a single bit flag indicating whether the compressed integer was spanning across two or more input chunks. This flag is used by the (next) output alignment block.

5.4.2.4 Output Alignment Block

The output alignment block adds buffers after every integer builder, such that the output of all integer builders in a given cycle reflects the processing of the same input data chunk. The number of buffers after an integer builder is simply the number of integer builders following it. Furthermore, reconstructed integers are re-ordered such that the integer spanning in the chunk is placed before others from that chunk. This is achieved using the *was-spanning* flag outputted with every reconstructed integer (Figures 5.2 and 5.5). Note that only one integer can be spanning in a given chunk. Also, even though the spanning integer is the first in a chunk, it is not necessarily processed by the first builder, since its processing could have started by a later builder with the previous chunk. Since some integer builders are idle in certain cycles, their output is disregarded by the output alignment block.

5.4.3 One-Integer Per Cycle Decoder Overview

Based on the no-stall architecture, a smaller one-integer per cycle architecture is presented as depicted in Figure 5.6. The main difference from the no-stall approach is the use of a single (modified) integer builder, as well as the FIFOs and respective controllers highlighted in dark grey. As a data chunk is received, the number of integers it contains is computed, and that many cycles are spent processing it, prior to moving onto the next chunk. Therefore, wire speed throughput is not maintained, and a FIFO (with corresponding controller FSM) is inserted between the (modified) integer builder and the delimiter insertion pipelines. A back pressure signal is propagated from that FIFO to the controller of another FIFO at the input of the decompression block, with the goal of avoiding dropping compressed data chunks.

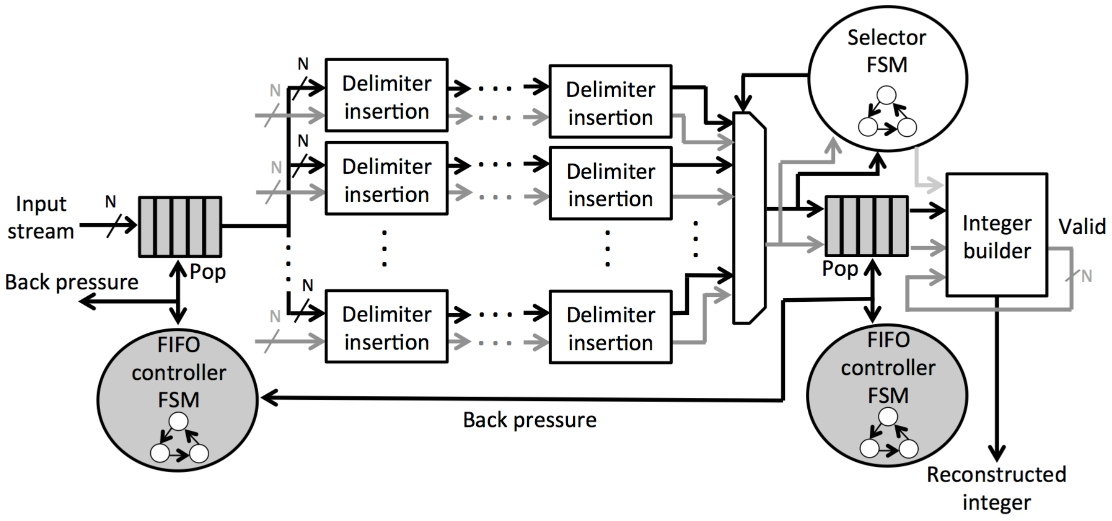
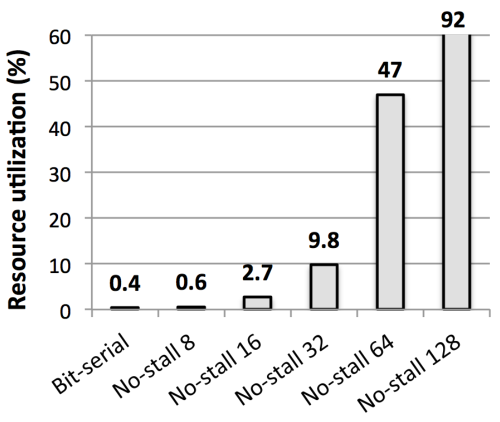


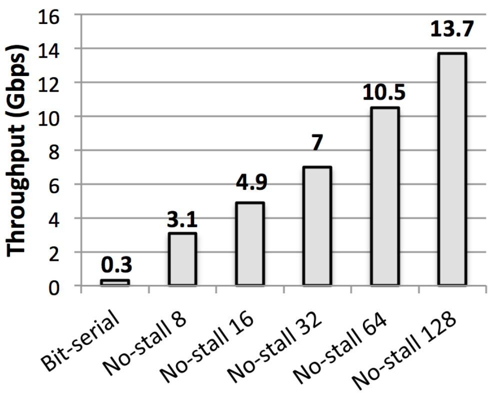
Figure 5.6: Overview of a decoder with a peak throughput of one integer per cycle. The main difference from the no-stall approach is the use of a single (modified) integer builder, as well as the FIFOs and respective controllers highlighted in dark grey. Back pressure is needed in between FIFOs to avoid dropping compressed data chunks. © 2013 IEEE.

This architecture is presented and studied as it requires less resources than a no-stall decoder accepting similar-sized input chunks; furthermore, it outperforms the bit-serial implementation (the only implementation in the literature with no assumptions on the compressed stream).

This *one-integer-per-cycle* architecture is mainly useful when the remainder size k is comparable to N. For instance, the one-integer-per-cycle decoder assuming chunks of size N uses less resources than a similar no-stall due to the fewer integer builders. The minimum throughput offered by the one-integer-per-cycle decoder is comparable to that of a no-stall accepting chunks of size k; and depending on the dataset, the one-integer-per-cycle could provide higher effective throughput than the latter.



(a)



(b)

Figure 5.7: Resource utilization (a) and throughput (b) of the hardware decoders are shown, targeting a Xilinx V6LX240T FPGA, with *k=3*. The naive bit-serial implementation is considered for comparison purposes. The no-stall decoder processing 32 bits per cycle occupies only 10% of the (mid- to low-sized) FPGA, and achieves a 7 Gbps throughput. © 2013 IEEE.

5.4.4 Decoder Generator

A (C++) tool has been developed to generate the HDL of the decompression pipeline, using certain parameter inputs. These include the input bit-width N (chunk size); the GR parameter k (number of remainder bits); whether to make use of a no-stall decoder, a single integer builder, or a bit-serial decoder (for testing purposes); the option to further pipeline certain stages; the option to deploy a multi-integer per cycle arbiter at the output, to match the bit-width of the interface of the block following the decompression pipeline (RAM, PCIe, computational core, etc); as well as other knobs useful to hardware designers. This (7000 lines of C++) tool was implemented from scratch.

5.5 Experimental Evaluation

In this section, an experimental evaluation of the proposed hardware GR decoder is carried out. A performance study versus state-of-the-art software decoders is further detailed.

5.5.1 Resource Utilization Study

The hardware decoders were tested on a Pico Computing M-501 board [66], connected to a host CPU via PCIe. The M-501 board includes a Xilinx Virtex 6 LX240T FPGA, which is assumed for the remainder of this study.

Figure 5.7(a) reports the resulting (post-place and route) resource utilization of the no-stall decoders, on the target V6LX240T FPGA. The Xilinx ISE v14.4 tools are used for synthesis/place and route, with the optimization goal set to speed (normal).

With the number of remainder bits *k=3*, each decoder is tailored for N, being the number of bits processed per cycle. A bit-serial decoder is included for comparison purposes.

The fully parallel no-stall architectures processing 8 and 16 bits per cycle occupy minimal FPGA resources ($< 3\%$). Generally, as N is doubled, the resulting decoder is around 4X larger, with the exception of No_stall-128. In the case of the latter, we suspect that the effort of the tools was higher for area, due to the size of the design (potentially not fitting). Furthermore, this 128-bit pipeline cannot be used on the target FPGA, though it fits; this is because any logic connecting the FPGA to peripheral devices (ethernet, DDR, PCIe, etc) would potentially require more than the remaining resources.

Figure 5.8 shows the resource utilization of a (32 bit) no-stall hardware decoder as the number of remainder bits k is varied. As k increases, the number of delimiter insertion pipelines (*k+1*) directly increases; conversely, the number of stages in each pipeline ($\lceil \frac{N}{k+1} \rceil$) directly decreases. Hence, the total number of delimiter insertion stages remains constant (equal to N) as k varies. On the other hand, as k increases, the number of integer builders ($\lceil \frac{N}{k+1} \rceil$) decreases, thus leading to a considerable drop in resource utilization (up to 40%). The effect of varying k on the operational frequency is marginal; as k increases, the critical paths in the delimiter insertion logic and integer builder increase (data omitted for brevity).

Resource utilization of the one-integer-per-cycle decoders (Section 5.4.3) is comparable to that of a no-stall decoder of the same bit-width, with large k assumed. For instance, a 32-bit one-integer-per-cycle decoder occupies around 6.5% of the FPGA logic, comparable to a 32-bit no-stall decoder with *k=21.* On the other hand, since a single integer builder block is used, and because varying k has no effect on the number of delimiter insertion stages, k has minimal impact on the overall resource utilization of the one-integer-per-cycle decoders. Data has been omitted due to space limitations.

5.5.2 Performance Evaluation

In this section, throughput is measured at the input of the studied decoders. In other words, it is measured as a function of the time required to process a compressed document, regardless of the rate at which uncompressed integers are generated at the output. The latter has been used (in addition to the former) as s metric in some studies such as in [90].

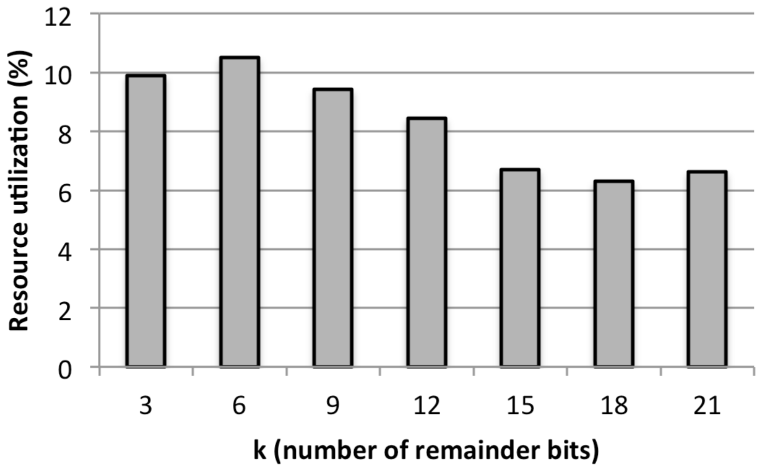


Figure 5.8: The resource utilization of a (32 bit) no-stall hardware decoder is studied as the number of remainder bits k is varied. Place and Route results are shown targeting a Xilinx V6LX240T FPGA. As k increases, the number of delimiter insertion pipelines directly increases, but the number of stages in each pipeline directly decreases. Hence, the total number of delimiter insertion stages remains constant as k varies. On the other hand, as k increases, the number of integer builders decreases, thus leading to a (considerable) drop in resource utilization. © 2013 IEEE.

The performance of the bit-serial and no-stall architecture hardware decompression cores is studied, as shown in Figure 5.7(b) , where k=*3*. Throughput is measured as a function of the operational frequency, and the number of bits processed per cycle; throughput does not increase linearly with the number of bits processed per cycle, due to the negative impact on the operational frequency of the decompression circuit.

The critical path of the no-stall decoders resides in the unary and remainder extraction stage of the integer builder. Specifically, the extraction of the remainder bits limits performance. Nonetheless, this block can be trivially pipelined further. The next long wire is found in the delimiter insertion stage; the latter can also be trivially pipelined, as it contains no control logic. Since the developed decoders achieve good performance, further pipelining is not applied here, due to the added penalty on resources.

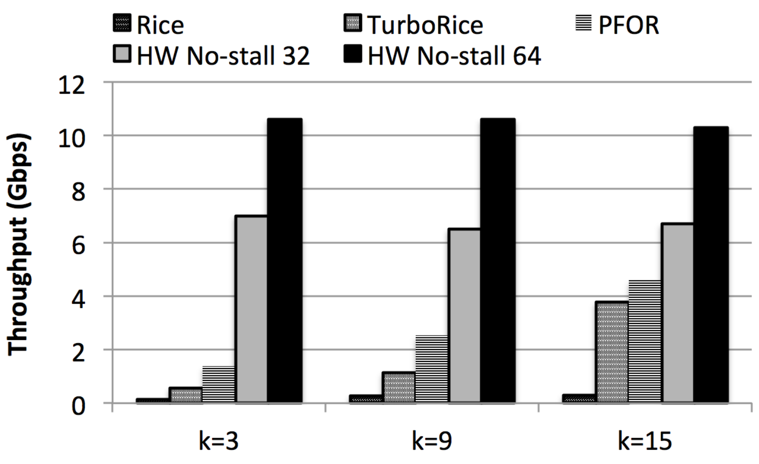


Figure 5.9: Throughput (Gbps) achieved by software and hardware decoders, as the number of remainder bits k is increased. The performance of two hardware decoders is reported here, namely (HW) No-stall 32 and No-stall 64, each processing 32 and 64 bits per hardware cycle, respectively. PFOR is considered as it has shown the best decompression performance in the literature [90]. TurboRice was introduced in [90] as a new approach combining the compression ratio of GR with the performance of PFOR. © 2013 IEEE.

Note that the performance of the one-integer-per-cycle decoders depends on the data set; here, the sustained throughput is bound by $k+1$ bits per cycle (the minimum compressed integer size) and N (the number of bits read per cycle).

We next compare the performance of the proposed decoders to state-of-the-art high performance CPU-based software decoders. We make use of the open source (C++) software decoders described in [90]. Three software decoders are considered, namely (1) Rice, a highly efficient implementation of the base GR coding; (2) TurboRice, a newly proposed approach in [90], combining the compression ratio benefits of GR coding, with the performance of the PFOR method; (3) PFOR, a compression mechanism that targets blocks of integers at a time (hence neither bit- nor byte-granularity). Other CPU-based approaches were studied (variable byte, S9, S16), and their performance was within the range or Rice, TurboRice, and PFOR; hence, only the latter are reported here.

All CPU-based approaches were ran on a CentOS 5 server with an Intel Xeon processor running at 2.53 GHz, with 8MB of L3 cache, and 36 GB of RAM. Synthetic datasets containing 500 million integers each were generated, while varying the range of the integers (hence k). A large set of integers is assumed in order to ensure that steady-state performance is measured. Throughput is measured as a function of the wall-clock time, such that the compressed and resulting uncompressed data reside in the CPU RAM. Moreover, throughput is measured as a function of the size of the compressed data, respective to each software approach.

Figure 5.9 shows the throughput (Gbps) achieved by software and hardware decoders, as the number of remainder bits k is increased. The performance of two hardware decoders is reported here, namely (HW) No-stall 32 and No-stall 64. As discussed earlier, varying k has marginal impact on the performance of the hardware decoders. On the other hand, CPU-based approaches perform better as k increases. As k is increased from 3 to 15, the throughput of the CPU approaches increases by an average of 4X. This is because the implementation of the CPU-based approaches processes encoded data 4 bytes at a time (one integer). As the remainder size increases, the number of (individual) unary bits to be processed per 4 bytes decreases. Nonetheless, data to be compressed by GR is generally small, and large remainder values are not assumed.

The proposed no-stall 32 and 64 architectures provide a higher throughput than PFOR (average of 3X and 4.7X speedup respectively), as well as TurboRice (average of 6.8X and 10.4X speedup respectively). Furthermore, the only software approach that operates on unmodified GR data is Rice, where the no-stall 32 and 64 architectures are respectively up to 52X and 79X faster, as well as respectively 34X and 51.5X faster on average.

5.6 Conclusions

A novel highly-parallel hardware core capable of decompressing streams of Golomb-Rice-coded integers at wire speed (no-stall) with constant throughput is presented, operating on *raw unmodified* GR data. To the best of our knowledge, hardware and software (CPU-based) GR decoders assuming unmodified GR data operate bit-serially on the compressed stream, which highly bounds the achievable decompression speeds. Hence, even though GR offers high compression ratios, other approaches are preferred due to the gap in decompression performance. The presented decoder, capable of processing several bytes per cycle, is shown to outperform an efficient GR CPU-based implementation by up to 52X, while utilizing 10% of resources available on a Xilinx V6LX240T FPGA. Furthermore, when operating on 64 bits per cycle, the presented decoder provides average speedups of 4.7X and 10.4X when respectively compared to a software implementation of the high-performance PFOR and TurboRice de/compression methods.

Chapter 6

Conclusions

Due to their relative ease of use, general purpose processors are commonly favored at the heart of many computational platforms. These processors are deployed in environments with varying requirements, ranging from personal electronics, to game consoles and up to server-grade machines. General purpose CPUs follow the Von-Neumann model, and execute instructions sequentially. Furthermore, performance does not always linearly scale in multi-processor environments, mostly due to the challenges of data sharing across cores. As it is non-trivial for these CPUs to satisfy the increasing time-critical demands of several applications, they are often coupled with application- or domain-specific parallel accelerators, such as Graphics Processing Units (GPUs) and Field Programmable Gate Arrays (FPGAs), which strive given a certain class of instructions and memory access patterns.

This dissertation proposes the use of hardware accelerators such as Field Programmable Gate Arrays (FPGAs) and Graphics Processing Units (GPUs) as a substitute or co-processor to general purpose CPUs, with a focus on database applications, where large amounts of data are queried in a time-critical manner. Specifically, the hardware acceleration of four applications is studied, namely:

- **XML Path Filtering:** This dissertation 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 providing support for complex XPath structural constructs, such as parent-child and ancestor descendant relations, whilst allowing wildcarding and recursion. A novel method for matching user profiles that supports dynamic query updates using a programmable FPGA is presented. 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 7 GB).

  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 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.

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

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

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

- **Querying Spatio-Temporal Databases:** The wide and increasing availability of collected data in the form of trajectory has lead to research advances in behavioral aspects of the monitored subjects. Using trajectory data harvested by devices, such as GPS and mobile devices, complex pattern queries can be posed to select trajectories based on specific events of interest. However, as the complexity of the posed queries increases, so do computational requirements, which are not easily met using traditional CPU-based software platforms.

In this work, the first proof-of-concept study on FPGA-based architectures for matching variable-enhanced complex patterns is presented, with a focus on stream-mode (single pass) filtering. A tool for automatically generating hardware constructs using a set of queries is presented, abstracting away ramifications of hardware code development and deployment. A thorough design space exploration of the hardware architectures shows that the presented solution offers good scalability, fitting thousands of query matching engines on a Xilinx V6LX240T FPGA, a mid- to low-size FPGA. Increasing the number of variables and wildcards is shown to have linear effect on the resulting circuit size, and negligible on performance. That is unlike CPU-based solutions, where performance is greatly affected from such query characteristics.

When handling queries with (a) no variables, (b) one variable, or (c) no wildcards with two or more variables, the proposed hardware architecture is able to process the trajectory data in a single pass. When two or more variables are used alongside wildcards, the proposed solution will result in false positives, though these are minimal in practice. Nonetheless, a no-false-positive solution is proposed, though being limited in scalability.

As part of our future work, we will be enhancing the proposed framework to allow online query updates. The deployed generic query engines would support "any" query structure and node values. A stream of bits forwarded to the FPGA would program the connections between deployed query nodes. This approach should not be confused with Dynamic Partial Reconfiguration (DPR), where the bit configuration of the FPGA itself is updated.

- **Golomb-Rice Integer Decompression:** A novel highly-parallel hardware core capable of decompressing streams of Golomb-Rice-coded integers at wire speed (no-stall) with constant throughput is presented, operating on *raw unmodified* GR data. To the best of our knowledge, hardware and software (CPU-based) GR decoders assuming unmodified GR data operate bit-serially on the compressed stream, which highly bounds the achievable decompression speeds. Hence, even though GR offers high compression ratios, other approaches are preferred due to the gap in decompression performance. The presented decoder, capable of processing several bytes per cycle, is shown to outperform an efficient GR CPU-based implementation by up to 52X, while utilizing 10% of resources available on a Xilinx V6LX240T FPGA. Furthermore, when operating on 64 bits per cycle, the presented decoder provides average speedups of 4.7X and 10.4X when respectively compared to a software implementation of the high-performance PFOR and TurboRice de/compression methods.

Bibliography

[1] Chorochronos. http://www.chorochronos.org/, 2013.

[2] Shurug Al-Khalifa, H.V. Jagadish, Nick Kodus, Jignesh Patel, Divesh Srivastava, and Yuqing Wu. Structural Joins: A Primitive for Efficient XML Query Pattern Matching. In *ICDE: Proceedings of the 18th International Conference on Data Engineering*, page 141, Washington, DC, USA, 2002. IEEE Computer Society.

[3] Mehmet Altinel and Michael J. Franklin. Efficient Filtering of XML Documents for Selective Dissemination of Information. In *VLDB: Proc. of the 26th Intl. Conf. on Very Large Data Bases*, pages 53–64, San Francisco, CA, USA, 2000. Morgan Kaufmann Publishers Inc.

[4] Naiyong K. Ao, Fan Zhang, Di Wu, Douglas Stones, Gang Wang, Xiaoguang Liu, Jing Liu, and Lin Sheng. Efficient Parallel Lists Intersection and Index Compression Algorithms using Graphics Processing Units. In *VLDB: Very Large Databases*, 2011.

[5] Mahmoud Attia Sakr and Ralf Hartmut Güting. Spatiotemporal pattern queries in secondo. In *Proc. of the Int'l Conf. on Advances in Spatial and Temporal Databases (SSTD)*, pages 422–426. Springer, 2009.

[6] Denilson Barbosa, Alberto Mendelzon, John Keenleyside, and Kelly Lyons. ToX-gene: a Template-Based Data Generator for XML. In *SIGMOD: Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, pages 616–616, New York, NY, USA, 2002. ACM.

[7] LV Batista, LC Carvalho, and EUK Melcher. Compression of ECG Signals based on Optimum Quantization Of Discrete Cosine Transform Coefficients and Golomb-Rice Coding. In *Engineering in Medicine and Biology Society, 2003. Proc. of the 25th Annual Int. Conf. of the IEEE*, volume 3, pages 2647–2650. IEEE, 2003.

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

[9] K. Selçuk Candan, Wang-Pin Hsiung, Songting Chen, Junichi Tatemura, and Di-vyakant Agrawal. AFilter: Adaptable XML Filtering with Prefix-Caching Suffix-Clustering. In *VLDB: Proceedings of the 32nd international conference on Very large data bases*, pages 559–570. VLDB Endowment, 2006.

[10] Jonathan Cazalas and Ratan Guha. GEDS: GPU Execution of Continuous Queries on Spatio-Temporal Data Streams. In *IEEE/IFIP 8th Int'l Conf. on Embedded and Ubiquitous Computing (EUC)*, pages 112–119, 2010.

[11] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi. Efficient Filtering of XML Documents with XPath Expressions. *The VLDB Journal*, 11(4):354–379, 2002.

[12] J. Chen, J. Ma, Y. Zhang, and X. Shi. ECG Compression based on Wavelet Transform and Golomb Coding. *Electronics Letters*, 42(6):322–324, 2006.

[13] C.R. Clark and D.E. Schimmel. Efficient Reconfigurable Logic Circuits for Matching Complex Network Intrusion Detection Patterns. In *13th Intl. Conf. on Field Programmable Logic and Applications*, pages 956–959. Springer, Lisbon, 2003.

[14] D.J. Craft. A Fast Hardware Data Compression Algorithm and Some Algorithmic Extensions. *IBM Journal of Research and Development*, 42(6):733–746, 1998.

[15] Zefu Dai, Nick Ni, and Jianwen Zhu. A 1 Cycle-Per-Byte XML Parsing Accelerator. In *FPGA: Proc. of the 18th Intl. Symposium on FPGAs*, pages 199–208, New York, NY, USA, 2010. ACM.

[16] Yanlei Diao, Mehmet Altinel, Michael J. Franklin, Hao Zhang, and Peter Fischer. Path Sharing and Predicate Evaluation for High-Performance XML Filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[17] W3.org on DOM. http://www.w3.org/DOM.

[18] Cédric du Mouza, Philippe Rigaux, and Michel Scholl. Efficient evaluation of parameterized pattern queries. In *Prov. of the ACM Int'l Conf. on Information and Knowledge Management (CIKM)*, pages 728–735, 2005.

[19] Fadi El-Hassan and Dan Ionescu. SCBXP: An Efficient Hardware-Based XML Parsing Technique. In *SPL: 5th Southern Conference on Programmable Logic*, pages 45–50. IEEE, April 2009.

[20] M. Erwig and M. Schneider. Spatio-temporal predicates. *IEEE TKDE*, pages 881–901, 2002.

[21] Joshua Fender and Jonathan Rose. A High-Speed Ray Tracing Engine Built on a Field-Programmable System. In *IEEE Int'l Conf. on Field-Programmable Technology (FPT)*, pages 188–195, 2003.

[22] Free Lossless Audio Codec. http://www.xiph.org/flac.

[23] S. W. Golomb. Run-Length Encodings. *Information Theory, IEEE Trans. on*, 12(3):399, 1966.

[24] Gang Gou and Rada Chirkova. Efficient Algorithms for Evaluating XPath Over Streams. In *SIGMOD: of the 2007 ACM SIGMOD international conference on Management of data*, pages 269–280, New York, NY, USA, 2007. ACM.

[25] Naga K. Govindaraju, Brandon Lloyd, Yuri Dotsenko, Burton Smith, and John Manferdelli. High Performance Discrete Fourier Transforms on Graphics Processors. In *Proc. of the 2008 ACM/IEEE Conf. on Supercomputing*, SC '08, pages 2:1–2:12, Piscataway, NJ, USA, 2008. IEEE Press.

[26] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast Computation of Database Operations Using Graphics Processors. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[27] Todd J. Green, Ashish Gupta, Gerome Miklau, Makoto Onizuka, and Dan Suciu. Processing XML Streams with Deterministic Automata and Stream Indexes. *ACM Trans. Database Syst.*, 29(4):752–788, 2004.

[28] Ashish Kumar Gupta and Dan Suciu. Stream Processing of XPath Queries with Predicates. In *SIGMOD: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 419–430, New York, NY, USA, 2003. ACM.

[29] Marios Hadjieleftheriou, George Kollios, Petko Bakalov, and Vassilis Tsotras. Complex spatio-temporal pattern queries. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, pages 877–888, 2005.

[30] Marios Hadjieleftheriou, George Kollios, Vassilis Tsotras, and Dimitrios Gunopulos. Indexing spatiotemporal archives. *VLDB J.*, pages 143–164, 2006.

[31] Bingsheng He, Qiong Luo, and Byron Choi. Cache-Conscious Automata for XML Filtering. *IEEE Trans. on Knowl. and Data Eng.*, 18(12):1629–1644, 2006.

[32] Bingsheng He, Ke Yang, Rui Fang, Mian Lu, Naga Govindaraju, Qiong Luo, and Pedro Sander. Relational Joins on Graphics Processors. In *Proc. of the 2008 ACM SIGMOD Intl. Conf. on Management of data*, SIGMOD '08, pages 511–524, New York, NY, USA, 2008. ACM.

[33] Paul S Heckbert. *Graphics Gems IV*, volume 4. Morgan Kaufmann, 1994.

[34] P.G. Howard and J.S. Vitter. Fast and Efficient Lossless Image Compression. In *Data Compression Conf., 1993. DCC'93.*, pages 351–360. IEEE, 1993.

[35] Changkyu Kim, Jatin Chhugani, Nadathur Satish, Eric Sedlar, Anthony Nguyen, Tim Kaldewey, Victor Lee, Scott Brandt, and Pradeep Dubey. FAST: Fast Architecture Sensitive Tree Search on Modern CPUs and GPUs. In *SIGMOD: Proceedings of the 2010 ACM SIGMOD international conference on Management of data*, 2010.

[36] H.S. Kim, J. Lee, H. Kim, S. Kang, and W.C. Park. A Lossless Color Image Compression Architecture Using a Parallel Golomb-Rice Hardware CODEC. *Circuits and Systems for Video Technology, IEEE Trans. on*, 21(11):1581–1587, 2011.

[37] Sung-Soo Kim, Seung-Woo Nam, and In-Ho Lee. Fast Ray-Triangle Intersection Computation Using Reconfigurable Hardware. *Computer Vision/Computer Graphics Collaboration Techniques*, pages 70–81, 2007.

[38] D. Knuth, J. Morris, and V. Pratt. Fast pattern matching in strings. *SIAM J. on Computing*, 1977.

[39] Sailesh Kumar, Sarang Dharmapurikar, Fang Yu, Patrick Crowley, and Jonathan Turner. Algorithms to Accelerate Multiple Regular Expressions Matching for Deep Packet Inspection. In *ACM SIGCOMM Computer Communication Review*, volume 36, pages 339–350, 2006.

[40] Joonho Kwon, Praveen Rao, Bongki Moon, and Sukho Lee. FiST: Scalable XML Document Filtering by Sequencing Twig Patterns. In *VLDB: Proc. of the 31st Intl. Conf. on Very Large Data Bases*, pages 217–228. VLDB Endowment, 2005.

[41] T.Y. Lee. A New Frame-Recompression Algorithm And Its Hardware Design For MPEG-2 Video Decoders. *Circuits and Systems for Video Technology, IEEE Trans. on*, 13(6):529–534, 2003.

[42] Y. Lee, C.E. Rhee, and H.J. Lee. A New Frame Recompression Algorithm Integrated with H.264 Video Compression. In *Circuits and Systems, 2007. ISCAS 2007. IEEE Int. Symp. on*, pages 1621–1624. IEEE, 2007.

[43] W.D. Leon-Salas, S. Balkir, K. Sayood, and M.W. Hoffman. An Analog-to-Digital Converter with Golomb-Rice Output Codes. *Circuits and Systems II: Express Briefs, IEEE Trans. on*, 53(4):278–282, 2006.

[44] T. Liebchen and Y.A. Reznik. MPEG-4 ALS: An Emerging Standard for Lossless Audio Coding. In *Data Compression Conf., 2004. Proc. DCC 2004*, pages 439–448. IEEE, 2004.

[45] M.D. Lieberman, J. Sankaranarayanan, and H. Samet. A Fast Similarity Join Algorithm Using Graphics Processing Units. In *Data Engineering, 2008. ICDE 2008. IEEE 24th Intl. Conf. on*, pages 1111–1120. IEEE, 2008.

[46] Bertram Ludäscher, Pratik Mukhopadhyay, and Yannis Papakonstantinou. A Transducer-Based XML Query Processor. In *VLDB: Proceedings of the 28th international conference on Very Large Data Bases*, pages 227–238. VLDB Endowment, 2002.

[47] J. V. Lunteren, T. Engbersen, J. Bostian, B. Carey, and C. Larsson. XML Accelerator Engine. In *1st Intl. Workshop on High Performance XML Processing*. Springer Berlin / Heidelberg, 2004.

[48] A.K. Mahapatra and S. Biswas. Inverted Indexes: Types and Techniques. *Int. Journal of Computer Science*, 8(1):384–392, 2011.

[49] H.S. Malvar. Adaptive Run-Length/Golomb-Rice Encoding of Quantized Generalized Gaussian Sources with Unknown Statistics. In *Data Compression Conf., 2006. DCC 2006. Proceedings*, pages 23–32. IEEE, 2006.

[50] M.M. Meira, J.A.G. de Lima, and L.V. Batista. An FPGA Implementation of a Lossless Electrocardiogram Compressor based on Prediction and Golomb-Rice Coding. In *Proc. V Workshop de Informática Médica*, 2005.

[51] N. Memon. Adaptive Coding of DCT Coefficients by Golomb-Rice Codes. In *Image Processing, 1998. ICIP 98. Proc. 1998 Int. Conf. on*, volume 1, pages 516–520. IEEE, 1998.

[52] Abhishek Mitra, Walid Najjar, and Laxmi Bhuyan. Compiling PCRE to FPGA for Accelerating SNORT IDS. In *Proc. of the ACM/IEEE Symp. on Architecture for Networking and Communications Systems (ANCS)*, pages 127–136, 2007.

[53] Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Walid Najjar, and Vassilis J. Tsotras. Boosting XML Filtering Through a Scalable FPGA-based Architecture. In *CIDR: 4th Conference on Innovative Data Systems Research*. ACM, 2009.

[54] Hoda Mokhtar, Jianwen Su, and Oscar Ibarra. On moving object queries. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, pages 188–198, 2002.

[55] Mirella M. Moro, Petko Bakalov, and Vassilis J. Tsotras. Early Profile Pruning on XML-Aware Publish-Subscribe Systems. In *VLDB: Proc. of the 33rd Intl. Conf. on Very Large Data Bases*, pages 866–877. VLDB Endowment, 2007.

[56] R. Moussalli, R. Halstead, M. Salloum, W. Najjar, and V.J. Tsotras. Efficient XML path filtering using GPUs. In *ADMS: Workshop on Accelerating Data Management Systems*, 2011.

[57] R. Moussalli, M. Salloum, W. Najjar, and V.J. Tsotras. Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In *ICDE: 2011 IEEE 27th International Conference on Data Engineering*. IEEE, 2011.

[58] Roger Moussalli, Walid Najjar, Xi Luo, and Amna Khan. A High Throughput No-Stall Golomb-Rice Hardware Decoder. In *Field-Programmable Custom Computing Machines (FCCM), 2013 IEEE 21st Annual Intl. Symp. on*. IEEE, 2013.

[59] Roger Moussalli, Mariam Salloum, Walid Najjar, and Vassilis Tsotras. Accelerating XML Query Matching Through Custom Stack Generation on FPGAs. In *HiPEAC: High Performance Embedded Architectures and Compilers*, pages 141–155. Springer Berlin / Heidelberg, 2010.

[60] Cédric Mouza and Philippe Rigaux. Mobility patterns. *Geoinformatica*, 9(4):297–319, 2005.

[61] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: a Query Compiler for FPGAs. *Proc. VLDB Endow.*, 2(1):229–240, 2009.

[62] Netezza. http://www.ibm.com/software/data/netezza/.

[63] J. Núñez, C. Feregrino, S. Jones, and S. Bateman. X-MatchPRO: A ProASIC-based 200 Mbytes/s full-duplex lossless data compressor. In *Field-Programmable Logic and Applications*, pages 613–617. Springer, 2001.

[64] Feng Peng and Sudarshan S. Chawathe. XPath Queries on Streaming Data. In *SIGMOD: Proceedings of the 2003 ACM SIGMOD international conference on Management of data*, pages 431–442, New York, NY, USA, 2003. ACM.

[65] D. Pfoser, C. Jensen, and Y. Theodoridis. Novel approaches in query processing for moving object trajectories. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, pages 395–406, 2000.

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

[67] Michal Piorkowski, Natasa Sarafijanovoc-Djukic, and Matthias Grossglauser. A Parsimonious Model of Mobile Partitioned Networks with Clustering. In *The First International Conference on COMmunication Systems and NETworkS (COMSNETS)*, January 2009.

[68] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Amo Jacobsen. Efficient Event Processing Through Reconfigurable Hardware for Algorithmic Trading. In *VLDB: Intl. Conf. on Very Large Data Bases (VLDB)*, 2010.

[69] Mariam Salloum and V.J. Tsotras. Efficient and Scalable Sequence-Based XML Filtering System. In *WebDB: Proc. of 12th Intl. Workshop on the Web and Databases.* ACM, 2009.

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

[71] Jörg Schmittler, Sven Woop, Daniel Wagner, Wolfgang J Paul, and Philipp Slusallek. Realtime Ray Tracing of Dynamic Scenes on an FPGA Chip. In *Proc. of the ACM SIGGRAPH/EUROGRAPHICS Conference on Graphics Hardware (HWWS)*, pages 95–106, 2004.

[72] F. Scholer, H.E. Williams, J. Yiannis, and J. Zobel. Compression of Inverted Indexes for Fast Query Evaluation. In *Proceedings of the 25th Annual Int. ACM SIGIR Conf. on Research and Development in Information Retrieval*, pages 222–229. ACM, 2002.

[73] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using fpgas. In *Proc. of the the Annual IEEE Symp. on Field-Programmable Custom Computing Machines (FCCM)*, pages 227–238, 2001.

[74] B. Sukhwani, B. Abali, B. Brezzo, and S. Asaad. High-Throughput, Lossless Data Compresion on FPGAs. In *Field-Programmable Custom Computing Machines (FCCM), 2011 IEEE 19th Annual Int. Symp. on*, pages 113–116. IEEE, 2011.

[75] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using FPGAs. In *Proceedings of the 21st Intl. Conf. on Parallel Architectures and Compilation Techniques*, pages 411–420. ACM, 2012.

[76] Y. Tao and D. Papadias. MV3R-Tree: A spatio-temporal access method for timestamp and interval queries. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, pages 431–440, 2001.

[77] Y. Tao, D. Papadias, and Q. Shen. Continuous nearest neighbor search. In *Proc. of the Intl. Conf. on Very Large Data Bases (VLDB)*, pages 287–298, 2002.

[78] Jens Teubner, René Müller, and Gustavo Alonso. FPGA Acceleration for the Frequent Item Problem. In *ICDE: 26th International Conference on Data Engineering Conference*, pages 669–680, 2010.

[79] T.H. Tsai and Y.H. Lee. A 6.4 Gbit/s Embedded Compression Codec for Memory-Efficient Applications on Advanced-HD Specification. *Circuits and Systems for Video Technology, IEEE Trans. on*, 20(10):1277–1291, 2010.

[80] University of Washington XML Repository. http://www.cs.washington.edu/research/xmldatasets.

[81] Marcos R. Vieira, Petko Bakalov, and Vassilis J. Tsotras. Querying Trajectories Using Flexible Patterns. In *Proc. of the 13th Int. Conf. on Extending Database Technology (EDBT)*, pages 406–417, 2010.

[82] Marcos R. Vieira, Petko Bakalov, and Vassilis J. Tsotras. FlexTrack: a System for Querying Flexible Patterns in Trajectory Databases. In *Proc. of the Int'l Conf. on Advances in Spatial and Temporal Databases (SSTD)*, pages 475–480. Springer, 2011.

[83] M.J. Weinberger, G. Seroussi, and G. Sapiro. LOCO-I: A Low Complexity, Context-Based, Lossless Image Compression Algorithm. In *Data Compression Conf., 1996. DCC'96. Proc.*, pages 140–149. IEEE, 1996.

[84] TA Welch. A Technique for High-Performance Data Compression. *Computers, IEEE Trans. on*, 17(6):8–19, 1984.

[85] Louis Woods, Jens Teubner, and Gustavo Alonso. Complex Event Detection at Wire Speed with FPGAs. In *VLDB: Proc. of the 2010 Very Large Data Bases (VLDB)*, 2010.

[86] XILINX DELIVERS 65nm VIRTEX-5 LX330. http://www.xilinx.com.

[87] The XML Benchmark Projcet. http://www.xml-benchmark.org.

[88] XML Path Language Version 1.0. http://www.w3.org/TR/xpath.

[89] H. Yan, S. Ding, and T. Suel. Inverted Index Compression and Query Processing with Optimized Document Ordering. In *Proc. of the 18th Int. Conf. on World Wide Web*, pages 401–410. ACM, 2009.

[90] J. Zhang, X. Long, and T. Suel. Performance of Compressed Inverted List Caching in Search Engines. In *Proc. of the 17th Int. Conf. on World Wide Web*, pages 387–396. ACM, 2008.

[91] Yu Zheng, Yukun Chen, Xing Xie, and Wei-Ying Ma. Geolife2. 0: a location-based social networking service. In *Mobile Data Management: Systems, Services and Middleware, 2009. MDM'09. Tenth International Conference on*, pages 357–358. IEEE, 2009.

[92] Yu Zheng, Xing Xie, and Wei-Ying Ma. Geolife: A collaborative social networking service among user, location and trajectory. *IEEE Data Engineering Bulletin*, 33(2):32–40, 2010.

[93] J. Zobel and A. Moffat. Inverted Files for Text Search Engines. *ACM Computing Surveys (CSUR)*, 38(2):6, 2006.

Appendix A

GPU Architectures and Programming Model

Graphics Processing Units (GPUs) are emerging as computational platforms comprising of several hundreds of simple processors operating in a parallel fashion. While intended to be used solely for graphic applications, they are generally employed to accelerate solving general purpose problems of SIMD (Single Instruction Multiple Data) type, thus referred to as General Purpose GPUs (GPGPUs).

GPGPUs are used as co-processors to which the main CPU passes a stream of data; the GPGPU then processes the data with minimal memory footprint, and returns the processing results to the CPU.

Figure A.1 shows a high level view of a generic GPU architecture; note that we are making use of the NVIDIA CUDA model and terminology. Streaming Processors (SPs) are simple processor cores that are clustered. Each cluster of SPs is referred to as a Streaming Multiprocessor (SM), such that all SPs within one SM execute instructions

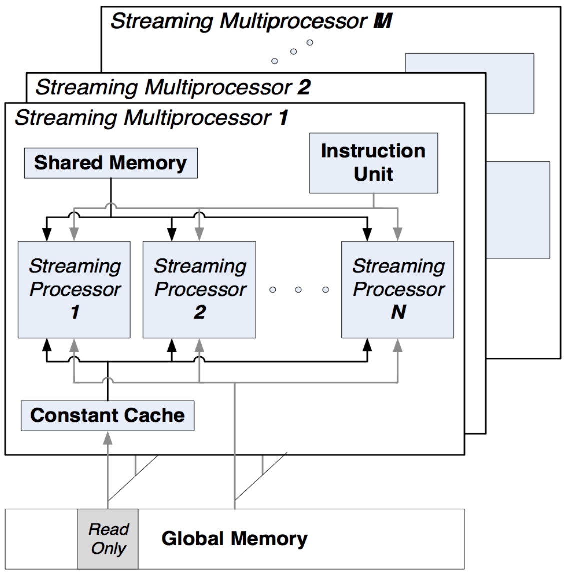


Figure A.1: High-level GPU architecture overview.

from the same memory block. When used with SIMD applications, all SPs on the GPU perform one common operation (at a time) on thousands of data elements.

Furthermore, all SPs within one SM communicate using a low latency shared memory structure. The SM also comprises of a *constant cache*, being a low-latency read-only memory, caching a (limited by size) read-only portion of the device global memory. The constant cache can be used for broadcast-type read operations, where all SPs require reading the same element from global memory. Finally, communication across SPs is achieved through the high latency global memory.

The programmer specifies the kernel that will be running on each of the SPs; however, when spawning the kernels onto the GPU, more instances of the kernel (threads) can be executed than the number of physical processing elements (SPs). The GPU manages switching threads on and off. Moreover, the number of physical cores is abstracted away from the programmer, and is only used at runtime.

Finally, the programmer specifies the number of instances of the kernel that are grouped to execute on a single SM. This group of kernels is referred to as the *block*. As the block size grows, the amount of shared memory available per block is reduced, and contention to computing resources increases; on the other hand, as the block size shrinks, the computational resources are under-utilized. The block size is determined per application basis, as to maximize occupancy (utilization).

Appendix B

Overview of FPGAs

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

In addition to the programmable elements, hard-wired components are added to FPGAs, such as embedded memory blocks and floating point cores. Hard-wired components operate at a higher clock frequency than their soft-logic counterparts, while

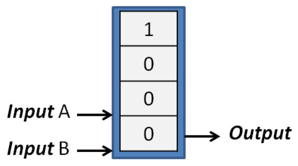


Figure B.1: Implementing a 2-input AND gate using a 2-input LUT.

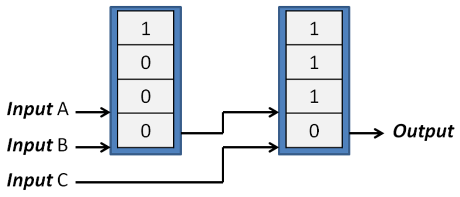


Figure B.2: Implementing *f(A,B,C)=(A AND B) OR C*, a 3-input boolean function, using two 2-input LUTs.

occupying a fraction of the resources. These are typically utilized by several classes of applications, and are simply bypassed if not used.

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

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