Technische Universität München
Lehrstuhl für Integrierte Systeme

Text Analytics on Reconfigurable Platforms

Dipl.-Ing. Univ. Raphael Polig

Vollständiger Abdruck der von der Fakultät für Elektrotechnik und Informationstechnik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktor-Ingenieurs (Dr.-Ing.)

genehmigten Dissertation.

Vorsitzender: Univ.-Prof. Dr. Gerhard Rigoll
Prüfer der Dissertation:
1. apl. Prof. Dr. Walter Stechele
2. Univ.-Prof. Dr. Marco Platzner, Universität Paderborn

Die Dissertation wurde am 05.10.2015 bei der Technischen Universität München eingereicht und durch die Fakultät für Elektrotechnik und Informationstechnik am 22.11.2015 angenommen.

To all my family

Abstract

With the arrival of Big Data, analyzing large amounts of unstructured data is becoming increasingly popular. The way digital data is generated has been revolutionized by the use of mobile devices and social media platform where unstructured data is shared in many forms from digital video and images to plain text. But also scientific publications, medical records, patent applications or machine data logs are created and stored in form of digital data. The analysis of this type of data may reveal valuable insights to scientists, marketers, medical doctors and system administrators, which can help them in their decision making.

At the same time, the end of transistor scaling makes it difficult for future processor generations to provide sufficient computational power to perform such analyses. To cope with this situation special purpose processors or accelerators are entering the data-centers. In particular reconfigurable architectures, such as field-programmable gate arrays (FPGAs), have proven to provide significant performance gains at low power increase. While the low power aspect allows them to be integrated into existing systems, their ability to reconfigure provides a level of flexibility to adapt to new applications.

This dissertation explores how a query-based text analytics system can be mapped onto a reconfigurable architecture and exploit it to achieve significant speedups. For this purpose a hardware compiler has been developed to generate a custom architecture for a user defined query. The document text is scanned by a set of finite state-machines in a single pass character-by-character. The state-machines are able to report the start and end offsets of various patterns within the text and produce naturally ordered results by the end offset. These results are further processed by relational algebra operators which can leverage this natural ordering to compose a fast and efficient streaming architecture. Additional optimizations have been developed for the query compiler when targeting an FPGA. These optimizations can significantly reduce the amount of resources required by the generated hardware.

To avoid lengthy synthesis times for an FPGA device, this thesis also explores a programmable approach for relational operators which does not require synthesis nor reconfiguration of the device. This led to the design of a custom soft-core processor array that leverages insights learned from the hardware compilation architecture. At the heart of the soft-core is the concept of virtual streams, which allows multiple data flows to be consolidated onto a single core. The soft-core array is augmented with shared resources for pattern matching operations and token offset lookup.

To ensure the feasibility of the approach, the system integration of the accelerator into an enterprise server system is discussed. The hardware interface logic has been designed to minimize the latency for individual document processing, while maximizing the compiled query core's throughput by supporting multiple submitting software threads. The software interface library is designed accordingly and is made available to the high-level programming language Java.

This work demonstrates the capabilities of reconfigurable architectures for text-based information extraction systems by allowing large scale queries to be run on-line.

Zusammenfassung

Die Art und Weise, wie digitale Daten erzeugt werden, wurde durch die Nutzung mobiler Geräte und sozialer Medien revolutioniert. Diese Daten sind vorrangig unstrukturiert und liegen in Form von Bildern, Videos oder kurzen Text Abschnitten vor. Big Data Analysen versuchen aus diesen Daten konkrete Informationen zu gewinnen, um Wissenschaftlern, Marketing Experten, Ärzten und IT Kräften bei ihren Entscheidungen zu unterstützen. Dabei werden neben sozialen Medien, auch wissenschaftliche Arbeiten, Patentanmeldungen, Patienten-unterlagen und System Logs analysiert, was eine zunehmende Anforderung an die Rechenleistung von IT Systemen bedeutet.

Zur gleichen Zeit, erschwert das Ende der Transistor Skalierung es, künftigen Prozessor Generationen, genügend Rechenleistung für solche Analysen bereitzustellen. Um dieser Situation entgegenzuwirken, werden vermehrt anwendungs-spezifische Prozessoren oder Beschleuniger in Rechenzentren verbaut. Besonders rekonfigurierbare Architekturen, wie field-programmable gate arrays (FPGAs), erzielen merkliche Verbesserungen bei nur geringfügig höherer Leistungsaufnahme. Zum einen lässt der geringe Stromverbrauch es zu, solche Architekturen in bestehende Systeme zu integrieren, zum anderen erlaubt die Möglichkeit der Rekonfiguration sich an neue Anwendungen anzupassen.

Ziel dieser Dissertation ist die Untersuchung von rekonfigurierbaren Architekturen als Zielplatform für query-basierende Text Analyse Systeme, um eine signifikante Reduktion der Verarbeitungszeit zu erreichen. Dazu wurde ein Hardware Compiler entwickelt, der eine spezifische Architektur erzeugt, welche eine gegebene benutzer-definierte Query abbildet. Eine Reihe von Zustandsautomaten analysieren ein Dokument Buchstabe für Buchstabe in einem einzigen Durchgang und erkennen dabei verschiedene Text-Muster. Durch diese Art der Verarbeitung werden sortierte Ergebnise erzeugt, mit Start- und End-Punkten der Muster innerhalb des Dokumentes. Diese werden durch relationale Algebra weiter verarbeitet, welche diese natürliche Reihenfolge nutzen kann, um effizient in einer Datenfluss Architektur abgebildet zu werden. Für die Abbildung von Queries auf FPGAs wurden spezielle Optimierungen fuer den Compiler entwickelt. Diese können die benötigten Resourcen, und damit auf die Leistungsaufnahme, der erzeugten Hardware Architektur erheblich senken.

Um lange Synthese Zeiten fuer FPGA Syteme zu vermeiden, untersucht diese Arbeit auch einen programmierbaren Ansatz fuer relationale Algebra, welcher weder eine Synthese, noch eine Rekonfiguration des Chips benoetigt. Dies mündete in der En-

twicklung eines soft-core Prozessor Arrays, welches Erkenntnisse aus dem Hardware Compiler Ansatz weiterführt. Ein wichtiges Konzept dieser Architektur sind virtuelle Datenströme, welche es ermöglichen mehrere Ströme auf einem Prozessor zu verarbeiten.

Um den Nutzen des gesamten Ansatzes zu verifizieren, wurde der Beschleuniger in ein kommerzielles Server System integriert. Das entwickelte Hardware Interface minimiert die Verarbeitungslatenz fuer einzelne Dokumente, erlaubt aber mehreren software threads gleichzeitig Dokumente einzureichen. Dadurch wird der Verarbeitungsdurchsatz maximiert. Die Software Schnittstelle wurde dementsprechend entwickelt und steht für die Hochsprache Java zur Verfügung.

Diese Arbeit zeigt das Potential von rekonfigurierbaren Architekturen für Text-basierte Informationsexkration, durch das deutlich beschleunigte Ausführen repräsentativer Queries.

Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

Digital data is being generated in every aspect of our daily lives [69]. We create more than 2.5 billion gigabytes of data per day [12], from sensor data to online shopping transactions. A great part of this data is represented by un-structured and semi-structured text documents. Such as the 500 million Twitter messages per day [65], uncountable number of daily news entries around the world or fewer but more complex scientific research papers. All of this data may or may not contain valuable information for a different audience such as a medical scientist or a marketing expert [103]. Extracting the specific information from this text-based data is the task of text analytics.

Although many improvements have been applied to the underlying frameworks and algorithms, text analytics continues to be computationally very intensive. Also because the level of detail at which data is analyzed continues to increase to improve the quality of results achieved. Furthermore these frameworks benefit only little from new features in modern microprocessors such as wide single-instruction multiple-data units [60]. This leads to a performance gap between the ever faster growing amounts of data and the moderate performance enhancements of new processor generations.

Text analytics can be easily parallelized as extraction queries are run for each text document individually. Cloud computing offers a way to cope with the lack of single node performance by massively parallelizing the task on a large compute cluster. Thus each thread in a cluster can operate independently from each other and will be utilized as the amount of data to be analyzed can be scaled. But this comes with a drop of efficiency due to the increased management and communication overheads. To counter this problem system designers turn to heterogeneous platforms which include special purpose processors such as graphics processing units (GPUs), digital signal processors (DSPs) or field-programmable gate arrays (FPGAs). While such systems are pre-dominantly used in high-performance computing (HPC) clusters, they are finding their way into enterprise data-centers [83]. By off-loading selected compute intensive parts of an application to these accelerator units the performance and power efficiency of a single node

can be enhanced by orders of magnitude, depending on the application.

The execution units of GPUs and DSPs are geared towards numerical tasks where they can exploit a high degree of parallelism on many levels together with very high memory bandwidth. In scientific computing GPUs are often used to accelerate Fourier transforms, Monte Carlo simulations or image processing. But due to the high popularity and low cost of GPUs, several projects have successfully attempted to also implement string pattern matching operations on them such as exact string matching [68] or regular expressions [101]. These implementations achieve fairly high throughput rates by utilizing one thread per pattern on the GPU, at the cost of high power consumption. Also advanced regular expression features such as capturing groups are not available in these implementations as they can have a significant impact on performance on these architectures.

FPGAs are the most common form of a reconfigurable architecture. They can implement merely any digital logic which is defined using a hardware description language. This allows them to efficiently execute a wide range of applications such as computer vision [80], genomic sequencing [73] and database operations [75]. Also string-based tasks such as regular expression matching perform an order of magnitude better than on GPUs, at a fraction of the power costs. Recent work also included complex regular expression features [18] with no impact on performance.

A drawback of FPGAs are lengthy design and synthesis times, creating the hardware description and generating the configuration memory from it. Furthermore the configuration can only be generated by FPGA vendor tools. Because text analytics queries are static in many cases, these limitations may be negligible, but yet limit the use of FPGA acceleration. To cope with synthesis times, chip designers turn to coarse grained reconfigurable arrays (CGRAs) or many-core architectures. These types of architectures are tailored towards a specific range of applications and can achieve higher operating frequencies. Yet they provide different levels of programmability, by trading a level of parallelism.

To be able to use any type of accelerator, it needs to be integrated into a host system. FPGAs are often used as ingress accelerators, where they perform a pre-filtering of incoming data from disks or the network connection. To benefit from GPU acceleration, a single work item has to be large enough to amortize the communication costs to the accelerator and back. In the text analytics case the accelerator needs to be tightly integrated into the system with minimal communication overhead. This is because single documents often are pre-processed and the results are required immediately to continue execution.

This dissertation describes the use of FPGAs to accelerate the processing of text analytics queries. It implements a query compilation framework that allows offloading complete queries onto FPGAs. The framework consists of a set of hardware operator modules and a hardware compiler that generates the hardware description for a given query. The framework is tightly integrated into an enterprise server system and achieves up to 945 MB/s document processing rate which is 71 times higher than the original software product. To avoid long synthesis times, a novel programmable soft-core processor

array is presented which requires less area at equivalent average performance.

1.1 Thesis statement

FPGAs have proven to be highly efficient for sub-tasks used in query-based text analytics such as regular expression matching and relational algebra. But to fully exploit the potential of FPGAs for text analytics, a holistic approach is necessary that extends and combines these tasks in a single accelerator framework. The framework must consist of an architecture and a compiler to be able to offload contiguous query graph operations. Offloading only individual operations is not suitable due to the small work size of an individual document. Because text analytics is not a stand-alone application, the accelerator needs to be tightly integrated into a host system. The host provides the input data and immediately acts on the results produced by the accelerator.

This leads to the following statement:

> Field programmable gate arrays provide sufficient flexibility and independent parallelism, to efficiently execute query-based text analytics. While individual operations exist for this technology, only a holistic framework consisting of a compiler and an architecture, will be able to demonstrate the true end-to-end potential in terms of performance and power consumption.

To validate this thesis I proceed as follows: An essential first step is to support the complete set of pattern matching operators with features specific to text analytics. While existing architectures can be used for regular expressions, a new architecture needs to be explored for large-scale token-aware dictionary matching. Once these units are available a framework can be developed to map relational operations to an FPGA fabric. The framework consists of a set of hardware operator modules that are combined by a hardware compiler to implement a specified query. Although the hardware compilation approach can leverage all features of an FPGA architecture, it requires long synthesis times and FPGA vendor tools to be deployed. To avoid these requirements, I design a programmable micro-architecture, to allow fast and vendor-independent query compilation. The programmable architecture serves as an alternative or extension to dynamic queries. Furthermore this architecture can be implemented as an application-specific integrated circuit, which can increase the performance even further. As a last step, the accelerator is integrated into an enterprise server system to evaluate the impact of interface communication costs.

1.2 Summary and key contributions

This dissertation investigates how text analytics queries can be mapped efficiently to a reconfigurable logic-based accelerator and how it can be integrated into an enterprise system. The main goal is to achieve significant improvements in terms of performance and power efficiency compared to a high-end server node. The presented architectures

and framework enable to compile and run text analytics queries on a reconfigurable platform while little or no modification need to be made to existing applications to benefit from this technology. To summarize, the key contributions of this thesis are as follows:

Framework for compiling text analytics queries to FPGAs

I describe a first of a kind framework for compiling text analytics queries to hardware structures. A hardware operator library is presented that allows stream processing of relational operations. The compiler is able to fuse these hardware elements to generate parallel and deep pipelines to execute entire queries. The combination of pattern extraction units and relational algebra modules results in unprecedented document processing rates, at less overall power consumption compared to a state-of-the-art commercial software implementation. Furthermore the compiler applies novel optimizations that can be applied when running text analytics queries on an FPGA to reduce the area consumption. This enables execution of large complex queries and reduces the static power consumption of the accelerator.

The acceleration framework was tested with a set of representative queries from different application domains such as business analytics, social media analytics and log file analysis. When using an Altera Stratix IV FPGA the accelerator is able to execute these queries up to 79 times faster and 89 times more energy efficient than the original commercial software on a POWER7 CPU using 64 threads.

Programmable architecture for fast and vendor independent compilation

I design a tailored programmable microarchitecture to avoid lengthy query compilation times required by the hardware compilation approach. Also the need for FPGA vendor design software to compile a query is removed by using this architecture. It exploits the streaming nature of the relational operations found in text analytics queries and combines it with the sparsity of document processing. By introducing the concept of virtual streams, the architecture is able to consolidate multiple operators onto a single core, while exploiting parallelism on an array of cores. The core is designed to efficiently execute relational algebra operators that are augmented with operations specific to text analytics.

The implemented core achieves up to 200 MHz operating frequency on a Altera Stratix V FPGA and can process up to 128 virtual streams. A single core processes 16 virtual streams, it requires 23 % less logic resources and an order of magnitude less memory resources than the custom hardware modules. The required performance for a text analytics query was modeled to determine the required size of the processor array. This analysis showed that all evaluated queries can run with 25 cores or less, without impact on the document processing rate.

Application integration and evaluation

I integrate the presented accelerator framework into a high-end enterprise server system. The designed hardware-software interface allows multiple threads running on the host processor to submit jobs to the accelerator card simultaneously without a communication thread. A Java interface layer allows easy integration into existing application code and allows to perform end-to-end measurements to compare performance and power consumption.

The accelerator was integrated using the coherent accelerator processor interface available on the POWER8 CPU. It allows the accelerator to access the virtual memory space and communicate with a software process via shared memory. This allowed to design a low latency communication scheme which enables full bandwidth of the accelerator core using small work items, such as real-life documents. Deployed on an Altera Stratix V FPGA the integrated accelerator framework was able to achieve up to 945 MB/s document processing rate, end-to-end. This is up to 71 times faster than the multi-threaded software on the POWER8 CPU.

1.3 Outline of this thesis

This thesis is organized as follows:

Chapter 2 provides some background on the topic of text analytics and its importance. It will then introduce the SystemT text analytics application and its terminology.

Chapter 3 presents the hardware architectures and compilers developed for pattern detection operations. These architectures provide deterministic performance characteristics as well as advanced detection features.

Chapter 4 presents the relational operations and their hardware implementations. After introducing the individual operators, the hardware compiler is presented. It combines and fuses the operators to form execution pipelines that are described in Verilog.

Based on insights gained in chapter 4, chapter 5 will introduce a programmable microarchitecture to execute relational operators. A custom soft-core processor is presented and its use in a stream-based manycore mesh.

The overall system integration will be presented in chapter 6. The hardware-software interface is presented as well as the integration into a Java-based application.

Chapter 7 summarizes and concludes this dissertation, before giving an outlook on future work.

CHAPTER 2

Background

The goal of this chapter is to establish a common basis of understanding of text analytics and the difficulties it poses on general purpose compute systems. The task of text analytics is introduced and how it has developed since its first steps in the 1950's. Example applications will demonstrate the importance of text analytics in today's digital world.

After the task of text analytics has been introduced, a detailed overview is provided on query-based text analytics. As a representative application IBM's SystemT is presented, on which the accelerator framework in this thesis is based off, and important terminology and concepts are defined and explained.

Section 2.3 provides an overview on some reconfigurable architectures and discusses their trade-offs.

The task of regular expression matching is introduced in section 2.4. After a short introduction of the main concepts the core hardware architecture is presented which is being used in this work.

2.1 Text Analytics

One of the most recent and popular examples employing text analytics' technologies is IBM's Watson computer [47]. The system was built to compete in the American television game show *Jeopardy!* where questions posed in natural language need to be answered. During the game Watson used text analytics to *understand* the question that had been given. But before being able to retrieve the correct information from its memory to answer such a question, Watson required to *learn* and it did so by analyzing text.

The underlying framework Watson uses to analyze unstructured text data, is called Unstructured Information Management Architecture (UIMA) [46]. UIMA enables developers from different disciplines to create a single large scale analytics system from many different complex components. It defines interfaces that each component needs to implement, in order to communicate with each other. The components itself, perform the actual analysis on the unstructured data like language identification, tokenization or information extraction. These components are designed by natural language processing (NLP) experts and can be written in Java or C++. On the other side domain experts create UIMA pipelines that combine multiple of these components to extract relevant and correct information from unstructured data sources. This allows language processing techniques to independently advance, while being able to integrate them to a language understanding system [47].

It is the latest development in NLP, bringing together technologies from over 50 years of research and engineering. Some of the earliest examples of language processing can be considered the Georgetown experiment from 1957 [85], in which an IBM mainframe translated Russian sentences to English, or the famous computer program ELIZA [104] developed by Joseph Weizenbaum. All of these early systems were using a set of handcrafted rules to decompose a sentence and extract information from it. This approach is referred to as *knowledge engineering* and is still used in today's systems.

With the availability of increasing computational power in the 1980's and 1990's machine learning algorithms were introduced to natural language processing [49]. In contrast to the previous approach, systems based on machine learning try to create or learn rules to understand a document. These approaches are often based on statistical inference [98]. By analyzing large amounts of data and using statistical models that create rules with a certain probability. Often the input data to such systems is hand-annotated with the correct values for the system to learn. Such systems are more robust to unfamiliar input, which is very important when analyzing social media documents, where new words and short-forms are created daily.

Modern natural language processing frameworks such as UIMA, GATE [40], NLTK [25] or OpenNLP [23] often combine multiple approaches to achieve best results. Specifically the task of *information extraction* (IE) has received increased attention with the vast availability and fast growth of unstructured digital data over the last few years. Due to the wide popularity of social media platforms, internet users today create the same amount of data every week, as there was created from the beginning of the internet until 2003 [29]. Information extraction is a part of natural language processing that aims to transform unstructured or semi-structured data into structured form like a table [37]. Such kind of data structure may already contain the information a user was looking for or can be further processed by a machine to create it. Figure 2.1 illustrates the task of IE, creating a table containing the desired information, extracted from a news article on the left.

Information extraction is a combination of different techniques, which are applied to process a text document [37]. As a first step the document is scanned and split into parts. These parts can be e.g. entire sentences or single words. This step is referred to as *segmentation* or *tokenization*. The following step is called *classification* or *named*

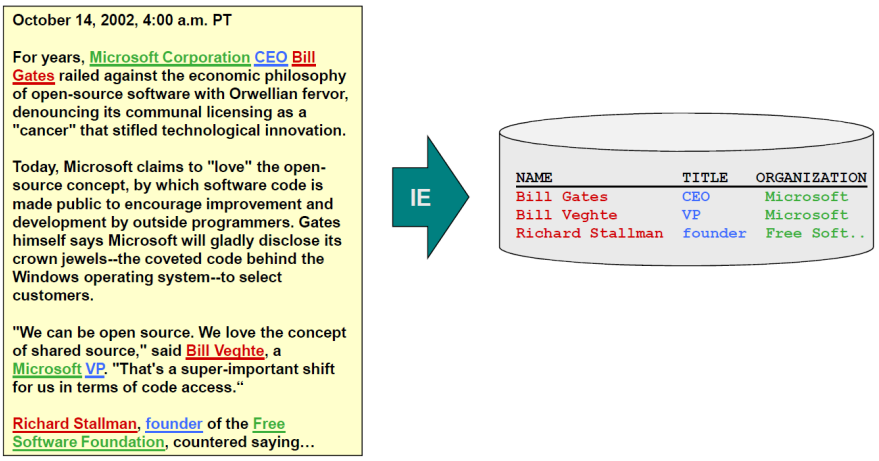


Figure 2.1: Information extraction transforms unstructured data into a structured form. Example by Cohen [37]

entity recognition (NER). It tries to identify words or a combination of multiple and assign them a category such as e.g. name of a country, name of an organization or telephone number. This is mainly done using pattern matching techniques such as regular expression matching and exact string matching. The same segment may belong to multiple categories or overlap with another segment as in this step the context is often neglected. After all possible entities have been detected, *relationship extraction* tries to identify the relation between them. This is often done by checking the distances between various entities and if they are contained within the same sentence. A similar task is *coreference resolution*, which tries to find text entities which refer to the same real-world entity.

In order to define the information that should be extracted from a document, a user needs to configure and combine these steps accordingly. In many cases users need to write their own software code while making use of libraries provided by natural language processing frameworks. With the shift towards language engineering it was important to make reuse of components and integrate them with the ability to relate them with each other. The Common Pattern Specification Language CPSL [17] developed in the late 1990's, became a very popular language for expressing rules on how certain components should behave and relate with each other. Similarly the Java Annotation Pattern Engine (JAPE) [41] defines a similar language to be used with the GATE framework.

These languages are mostly based on cascaded grammars defining a sequence of patterns expressed by regular expressions. Although these rules can be very powerful they often lack support for complex operations such as e.g. dictionary matching or the combination with character level regular expressions. Also because the input to such rules is a linear sequence of annotations, handling overlaps is difficult. Many of these issues have been addressed with advanced features of various frameworks but now they lack performance when analyzing data and achieving high quality results [66].

Because the available data on our planet is doubling every two years [97], performance and power efficiency are becoming an important factor when running such applications [64]. According to a study from the McKinsey Global Institute [69] these big data applications are becoming a major part of how businesses will work. And as the history of computing progresses from today's systems of engagement [74] towards cognitive computing [61], analyzing large amounts of data and especially text data will be a crucial task.

2.2 SystemT

This work is based on text analytics systems, which utilize rules to define the information which should be extracted from a set of documents called corpus. In particular algebraic rules, introduced by Clarke et al. [35] in 1995. A representative framework is SystemT [66], which is developed by a team at IBM Research - Almaden and is used in several IBM products such as IBM Notes or Infosphere BigInsights. It aims to overcome some of the difficulties that arise from using cascaded grammar-based approaches and their extensions. By simplifying the work-flow the interaction between various tasks becomes maintainable and can be expressed in a cleaner way. This also allows to decouple the way how rules are expressed in a language and how they will be executed by a system.

To achieve this SystemT uses a declarative rule language called *Annotation Query Language* (AQL), which is very similar to the Structured Query Language (SQL) [32] known from relational database applications. While keeping many relational operations from SQL like e.g. Select, Union or Join, AQL adds text-level features such as regular expressions and dictionary matching that operate on an entire text document or a segment of it. Such an expressive language allows users to define rules or queries in a modular and maintainable way, and are independent of the actual implementation of the operators. Figure 2.2 provides an example AQL query, extracting person names from a text document. Entire queries are often referred to as *extractors*.

SystemT is implemented in Java and consists of two main components as can be seen in Figure 2.3. On the left, there is the development environment in which a user creates and refines the AQL queries. Besides the user interface this environment contains the AQL compiler and an optimizer to create an execution plan for a given query. In a first step the AQL compiler translates the query into an annotation operator graph (AOG). The AOG is an acyclic dependency graph, where the nodes represent individual operators that work on the incoming data of their input edges. The optimizer then derives an execution plan for an AOG by applying transformations and using a cost-based optimization model. This can involve profiling the analysis of a set of reference documents to choose the best performing execution plan.

Once the user is satisfied with the results the developed query produces and the execution plan has been established, it can be deployed on the SystemT runtime. The runtime can be embedded into any Java application that requires text analytics capabilities. This can range from local email clients to large analytics applications. In the

```
create view Last as
        extract dictionary lastnames.dict on D.text
        as name from Document D;

create view First as
        extract dictionary firstnames.dict on D.text
        as name from Document D;

create view Caps as
        extract regex /[A-Z][a-z]*/ on D.text
        as name from Document D;

create view Person as
select S.name as name
from (
        ( select CombineSpans(F.name, C.name) as name
          from First F, Caps C
          where FollowsTok(F.name, C.name, 0, 0))
   union all
        ( select CombineSpans(F.name, L.name) as name
          from First F, Last L
          where FollowsTok(F.name, L.name, 0, 0))
   union all
        ( select *
          from First F )
) S
consolidate on name;
```

Figure 2.2: Example query written using the annotation query language (AQL)

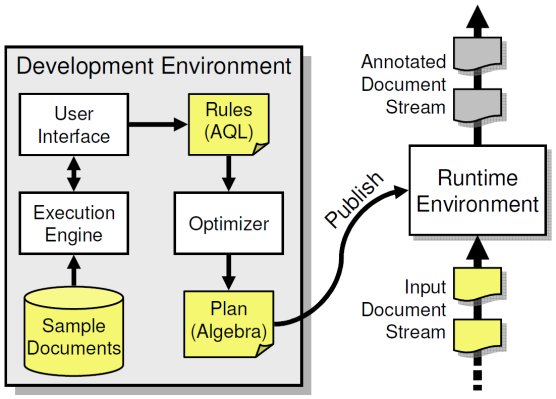


Figure 2.3: System architecture of SystemT [66]

latter case SystemT is often deployed on a large cluster of compute nodes using the Hadoop framework [11]. The SystemT runtime executes the query plan individually on each document it receives. The query plan is completely executed by a single thread over an entire document. Thus multiple threads are running independently from each other and exploit parallelism from the large number of individual documents. This type of large scale analytics is usually running continuously, processing online data, or for multiple hours on a given large set of documents.

Figure 2.4 represents the annotation operator graph for the AQL query shown in Figure 2.2. The two main categories of operators used are *extraction* operators and *relational* operators. Primarily the extraction operators consist of pattern matching steps such as regular expression matching or dictionary matching and are run over the entire document. This has the effect that often these types of operators are located at the very top or beginning of an AOG. These operators are creating a sequence of segments, which match the pattern they are looking for. Such segments are referred to as *Spans* and are described using a character-based start offset and an end offset. If an extraction operator follows a previous extraction operator, it will operate within the spans produced by its predecessor.

In most cases the extraction operators are followed by a usually larger number of relational operators. These operators mainly operate only on the offset values of the spans produced by parent nodes, creating a new set of spans. A prominent exception is the *Select* operation, which when using a regular expression in its conditional expression requires access to the actual document data of the span it currently operates on.

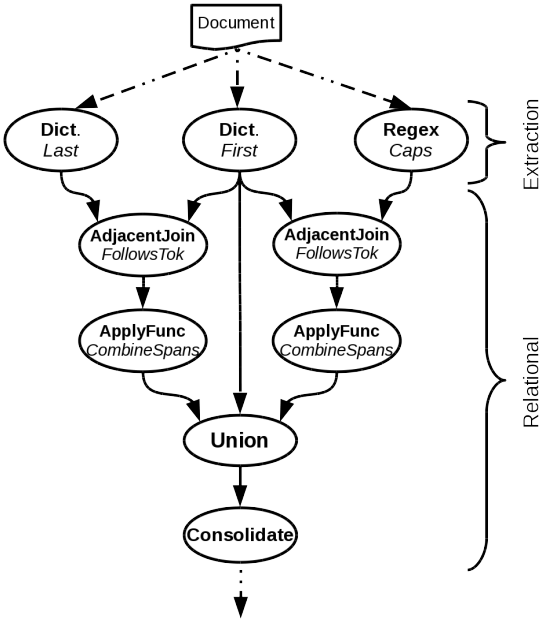


Figure 2.4: Example of an annotation operator graph (AOG)

2.3 Reconfigurable architectures

The main idea of this work is to create a custom datapath for parts of or an entire annotation operator graph to increase the document throughput rate. A custom datapath leverages the compute in space paradigm by carrying out many operations simultaneously in breadth as well as in depth. To enable the usage of a custom datapath the target platform requires to have at least one device with a reconfigurable architecture.

A reconfigurable architecture aims to perform a task just as fast as a dedicated piece of hardware. At the same time a reconfigurable architecture is able to adjust to perform another task just as fast at a different point in time. A general purpose processor coordinates the configuration of such a device with the appropriate data movement to and from it. Gerald Estrin was one of the first to propose such a computer architecture in 1960 [45]. But only with the advances in silicon technology and suitable electronic design automation (EDA) tools this idea came to live during the 1990's. At the same time these advancements pushed the development and performance of microprocessors with higher frequencies, larger caches, more complex instructions and lately increased number of cores per chip. Despite their efficiency, this put reconfigurable architectures into a niche due to their high costs and low productivity compared to general purpose processors.

With the end of frequency and multi-core scaling [44] the road to more performance for general purpose microprocessors became unclear. While optimizations to the architecture and compilers improve the overall performance of a system, the real gain in performance in recent years comes from heterogeneous systems [56]. Such systems include different types of compute units which are tailored towards a particular task such as massive parallel floating point operations. This already resembles Estrin's system architecture even so the compute units execute software code rather than being configured to perform a specific task.

Although system designers saw the potential of using reconfigurable architectures as an additional component in their systems, software application developers feared the loss of flexibility, longer design times and difficult debugging environment when using such components. This led to developments on both ends, leveraging languages and compilers as well as different reconfigurable architectures to simplify their usability in a heterogeneous system.

2.3.1 FPGAs

Field programmable gate arrays are the most widely deployed type of reconfigurable architecture. Initially used by the telecommunication industry, FPGAs today are used in many sectors such as automotive, healthcare or consumer devices [70]. The two main FPGA manufacturers are Altera and Xilinx, which control about 80% of today's multi-billion market [9].

Due to their fine granularity at Boolean logic level, FPGAs provide very high flexibility in what and how to implement a desired task. The basic idea of an FPGA is a set of configurable logic cells referred to as configurable logic block (CLB) (Xilinx) or adaptive logic module (ALM) (Altera) that can be arbitrarily interconnected via a programmable mesh of wires. A configurable logic cell is made up of a set of look-up tables (LUT), which can be configured to implement any desired logic function such as OR, AND or XOR. The LUTs are followed by a full adder (FA) structure, which can be used to create larger arithmetic or logic functions. The outputs of either LUTs or FAs can be routed to a flip-flop or register to create synchronous designs. Figure 2.5 illustrates this generic

architecture of a logic cell in an FPGA.

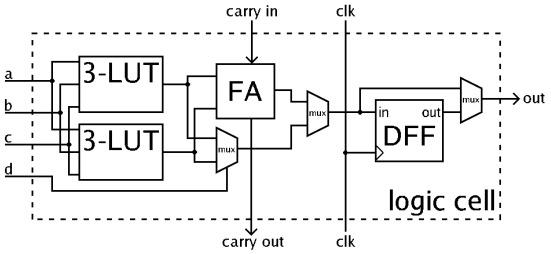


Figure 2.5: Example of a generic logic cell of an FPGA [10]

To enhance the efficiency and performance of FPGAs, modern architectures include special hard macros implementing a wide range of functions. Common blocks types are embedded memory, often referred to as BlockRAM, or digital signal processing (DSP) blocks, which consists of a multi-bit wide multiplier followed by an adder or accumulator stage. Furthermore complex input/output (I/O) blocks are added such as memory controllers for high throughput external memory access or PCI-Express controllers to provide a high-speed interface to a host processor. Especially for embedded systems some FPGA products include a full processor core such as an ARM A9 to perform less critical but more complex operations.

The mean to describe a task that should be executed on an FPGA is to write code in a hardware description language (HDL) such as VHDL or Verilog. Sophisticated electronic design automation tools are provided by chip vendors to compile such code and generate the contents of the configuration memory. The tools map the code to vendor-specific logic elements and perform placement and routing of these elements on the chip. This means that the design-flow for an FPGA application is closely related to designing hardware on a logic level. A designer needs to be aware e.g. of the limited amount and different types of available resources and the depth of logic between register stages to achieve timing closure. For hardware designers this is an ideal playground to experiment with different architectures but for a software programmer these are new unknown requirements.

To abstract these requirements from the programmers research and industry turned to high-level synthesis (HLS) [30]. In HLS the flow does not start from a hardware description but rather from an algorithm description in a programming language such as C. Although some restrictions apply this allowed software programmers to experiment with FPGAs in a way they are familiar with. A wide range of industry products adopted this flow and allowed different kind of inputs such as C, C++, SystemC or even Java and MatLab code.

With the increasing heterogeneity of computer systems the urge was strong to find a common way to program and communicate with different processing units. By the end of 2008 the first OpenCL technical specification was published by a consortium consisting of both hardware and software companies to tackle this challenge. OpenCL standard defines a hierarchical memory layout with different access permissions for different parts

of the code. While the code itself is very C-like, the programmer has to put some thought into how to partition the application into data-parallel and task-parallel pieces. This enabled hardware vendors to produce compilers that are able to compile such kernels to run on their devices. Initially CPU, GPU and DSP vendors provided such compilers and the required API library implementation to run it. But in 2013 Altera released the software development kit for OpenCL with their 13.0 tool suite [90].

Although all of these design technologies raised the interest of application designers, a last hurdle remains: long compilation times. Despite the use of incremental compilation and hard IP blocks the compile times of modern FPGAs can become easily multiple hours long.

2.3.2 Coarse grained reconfigurable and overlay architectures

While FPGAs can be configured at bit level, coarse grained reconfigurable arrays (CGRAs) and overlay architectures are configured at word level. These architectures consist of a sea of functional units (FUs) that can be interconnected using either statically configurable switchboxes, which create routing paths between FUs or by using a network type interconnect where packets are routed. The latter is often referred to as a network-on-chip (NoC). The coarse granularity of the individual components has two key advantages: On the one hand synthesis time to map a given algorithm to a CGRA is much smaller compared to the lengthy compilation times of FPGAs, on the other hand if designed as an ASIC the word-level functional units can achieve much higher frequencies as they can be optimized to their specific function. Such architectures are usually tailored to a specific set of applications such as multimedia or encryption tasks. The target application set determines the type and number of functional units as well as the level of configurability of the units and the interconnect network. Functional units can be word-wide adders and multipliers but also larger blocks such as complete FFT[1] IP blocks or programmable arithmetic-logic-units.

As an example figure 2.6 shows the architecture of the ADRES [72] CGRA template. It tightly integrates a very long instruction word (VLIW) processor with a reconfigurable mesh of reconfigurable cells (RCs). The RCs can be pre-configured with their functionality before generating the actual architecture. More architectures will be discussed in section 5.6.

While the term CGRA refers to an implementation as an ASIC, an overlay architecture is a design which is configured onto an FPGA and can be further configured to perform the desired task. Such designs may be also referred to as virtual or intermediate fabrics. The most notable difference of intermediate fabrics compared to CGRAs is that they provide the compiler the ability to select an optimized fabric for the required application. Using this two-level configuration the overhead consumed by the fabric can be minimized while maintaining short compilation times [95]. Besides the short compilation time another advantage of overlay fabrics is the ability to use readily available FPGA chips and decouple an end-user of the necessity to run the FPGA vendors synthesis tools to

[1] Fast Fourier Transform

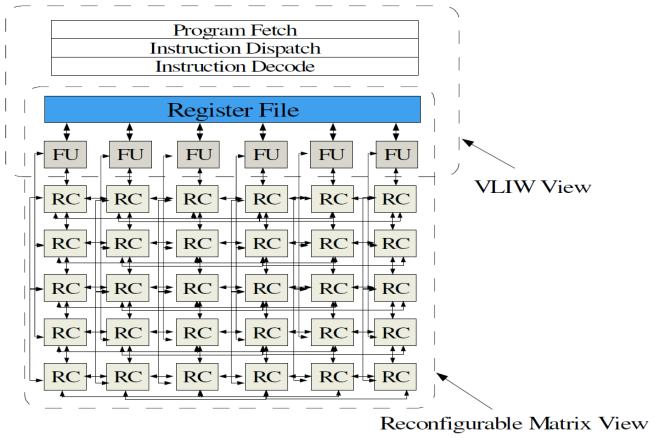


Figure 2.6: The ADRES CGRA core architecture [72]

program them.

2.3.3 Manycore processor arrays

Another type of coarse grained architecture is a manycore processor array. Instead of using static or configurable functional units (FUs), processor cores are used, which execute software kernels or even run entire operating systems. This makes the FUs more flexible and allows them to compute more complex kernels instead of single functional steps. This comes at the price of a higher area consumption of the FUs and a trade-off between computations in space versus time as instructions are carried out in sequence. But the ability to program the individual cores with common programming languages such as C, C++ or OpenCL has allowed such architecture to become commercially successful. Examples of such architectures are Intel's Xeon Phi [33], which is used as a co-processor card for high-performance computing (HPC) applications, or EZChip's (former Tilera) TILE processor [43] which targets high bandwidth networking applications. Also graphic processors (GPUs) may be considered a manycore architecture, such as the nVidia Kepler architecture [79], which is suitable for HPC and visual applications.

The interconnect of a manycore array is a key component, as it determines the communication performance of the core with each other and peripheral components such as memory and/or a system bus. Networks-on-Chip (NoC) are widely adopted as a communication scheme to connect a large number of functional units in a manycore system or a complex System-on-Chip (SoC) [27]. NoCs are categorized into circuit-switched and packet-switched networks. Circuit-switched networks create point-to-point connections at runtime between a sender and a receiver. Once the connection is established such networks provide a deterministic high bandwidth at a fair amount of power consumption [107]. Packet-switched networks can reduce the latency introduced by circuit-switched and also scale to an even higher number of cores. On the down-side packet-switched networks require the switch nodes to buffer the incoming packets before

forwarding them. Although the mechanisms are well-known from the classic networking domain these type of networks consume a high amount of power due to complex switch nodes and transmission of additional routing data with every data packet.

An alternative implementation are bus-based systems. For manycores the bus architecture is often designed as a ring connecting all or a subset of cores. Examples for such a design are Intel's Xeon Phi [33] or IBM's Cell Broadband engine [62]. The advantages of such architectures are the deterministic latency and performance they can provide as well as an adequate power consumption as little or no buffering is required and the routing complexity is low compared to packet-switched networks.

2.4 Regular expression matching

A regular expression is a search pattern consisting of a sequence of characters, where some of them have a special meaning and can be used to detect the defined pattern in a string. The formalism behind regular expressions is Kleene's theorem [63] which defines a regular language as a language that can be detected by a finite automaton. Thompson's construction algorithm [96] transforms a given regular expression into a non-deterministic finite automaton (NFA) by recursively splitting the expression into sub-expressions. A sub-expression at some point matches a defined rule which can be added to the automaton. To derive a deterministic finite automaton (DFA) from the constructed NFA, the powerset construction by Rabin and Scott [84] can be used. Running the finite automata over an input string will detect the pattern specified by the regular expression.

Pattern detection is a key component of text analytics and thus regular expressions are an important part of it. In SystemT regular expressions belong to the extraction type operators, the same as dictionary matching and are run over the entire document. The operator scans the document text and creates a span for every section of text matching the regular expression which is being evaluated.

The architectures used for regular expression matching were explored and designed by Atasu et al. [18, 20, 21]. Prior art hardware architectures for regular expression matching were only capable of identifying if and what regular expression matched a given string. These architectures targeted intrusion detection systems, where it was necessary to evaluate many patterns over a large number of network streams. A match was more of an exception and the reduced number of streams was then further evaluated in software to locate the match and determine an appropriate action. To locate the starting position of the match backtracking has to be performed which is computationally intensive and requires the input string to be scanned multiple times.

To avoid performing backtracking to find the start offset of a match, Atasu proposed the use of a network of state machines as seen in Fig. 2.7. It consists of multiple state machines and a shutdown logic. Every state machine implements the same DFA that has been constructed from the equivalent regular expression. On every starting transition a new state machine is started and captures the current character position as its start

offset. If two or more state machines reach an equivalent state the shutdown logic will compare the start offsets and disable all state machines but the one with the earliest offset. The comparison can also be omitted if during the activation of a new state machine the state (meaning whether or not a state machine is active) of the other state machines is captured. The size of the network can be computed during the conversion of the NFA representation to the DFA. The number of NFA states which are mapped to a single DFA state determines the number of required state machines to process the text document in a single pass [20].

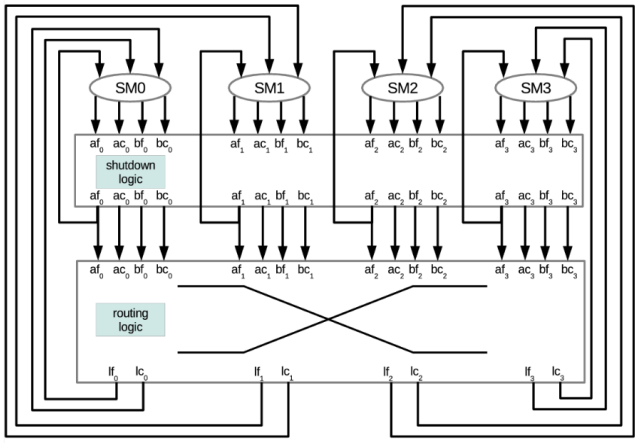


Figure 2.7: Network of state machines for regular expression matching [20]

An alternative architecture was proposed by Atasu in [18]. Instead of instantiating multiple state machines implementing the same DFA, this design employs a single NFA-based architecture based on Sidhu and Prasanna [89]. The start offset registers are attached to every state instead of every state machine. From multiple incoming transitions to the same state in the NFA the smallest start offset is selected and propagated. Also for this architecture it is possible to avoid the comparison logic by storing information about which state was active earlier. This optimized architecture requires up to three times less area resources and achieves up to 25% higher clock frequencies.

To implement the architecture a custom Verilog description is generated by a compiler for every regular expression individually. The compiled architecture can then be used as a module block by the hardware compiler presented in chapter 4.

2.5 Methodology

This dissertation aims to evaluate the feasibility and value of executing rule-based text analytics queries on reconfigurable architectures. To do so a representative text processing framework called SystemT is analyzed and extended to run on an FPGA platform.

The regular expression architecture presented in 2.4 serves as a starting point, as it

implements one fundamental pattern detection operator. To complete the set of extraction operators a scalable dictionary matching architecture will be developed. It needs to operate in parallel to the exiting regular expression architecture and support matching of multiple thousand string patterns.

Once the hardware architectures for the extraction operators are available, the relational algebra operators can be explored. Every operator requires an individual and configurable architecture but needs to be inter-operable with all operators. To compile complete queries onto an FPGA fabric, a compiler framework needs to be developed to generate and interconnect the operator modules. Such a framework will allow the evaluation and analysis of text analytics queries on FPGAs.

As an alternative, a programmable solution shall be presented that allows to change the query to be executed without reconfiguring the FPGA. The architecture will be implemented as a soft-fabric on the FPGA. Insights learned from the hardware compilation framework can be applied here and tailor the architecture.

To prove the value of the acceleration framework, it needs to be integrated into an enterprise system. Hardware and software interfaces need to be provided and evaluated, as they can add a substantial amount of communication overhead.

CHAPTER 3

Dictionary matching

Dictionary matching refers to the task of finding string patterns within a text document that belong to one or multiple collections of patterns, so called dictionaries. In natural language processing this step is also referred to as named entity recognition (NER) where a dictionary contains many patterns of a unique type such as city names, company names or actions (opens, starts, launches, ...). NER follows a tokenization step at which the document was split into non-overlapping segments called tokens. Tokenization is often done by splitting the text on white spaces, punctuation characters and special characters such as parentheses. Every created token is then matched against all required dictionaries to identify the potential meaning of it. Although many dictionary patterns consist only of a single token they also contain many other patterns which are a sequence of tokens, e.g. *New York*. The number of patterns in text analytics varies depending on the desired information to be discovered but ranges between a few hundred to a few 100,000 patterns.

Within SystemT the dictionary matching is an operator of the extraction class. Using AQL a user specifies each dictionary matching step individually by first creating a dictionary within AQL or from an external file and indicating whether the dictionary should be matched against the entire document or only a segment of it. The result is a list of spans indicating the positions within the text which matched the given dictionary. The compiler will group several dictionary matching operations into a single operator node if feasible for the execution plan. As software executes the operator nodes in sequence, it may decide to skip the execution of some operators if no results were produced by a previous operator. This can lead to a significant improvement in performance as entire sets of character-level operation may be skipped. On the other hand shared dictionary matching requires less passes over all tokens to identify their matches.

Despite these optimizations the processing time spent at dictionary matching is a significant amount of the overall runtime. SystemT is capable of generating an operator

type-level profile to inspect the performance bottlenecks of a text analytics query. The profiling results strongly depend on the query itself as well as the chosen set of documents the profiler runs with. Documents with few pattern matching results will perform better as many nodes in the AOG can be skipped, distorting the measurement. Thus the set of reference documents should be representative for the bulk of data to be processed.

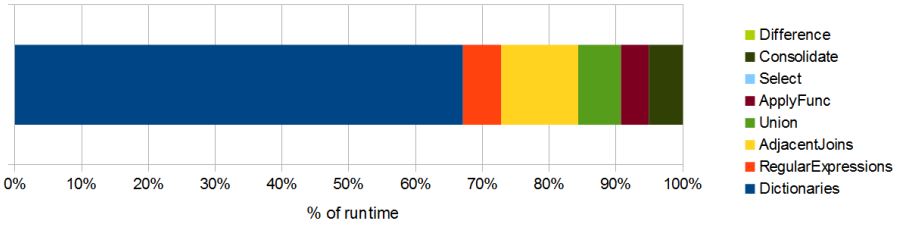


Figure 3.1: Profiling of a text analytics query dominated by Dictionaries

Figure 3.1 shows the profile of a text analytics query. The profile shows the relative processing time spent at each operator type. Together with regular expression matching, dictionary matching requires the most processing time in many queries ranging from 10 to 70 % of the overall execution time. Furthermore, because dictionary matching is part of the extraction operators which are required to run as a first set of operations, they determine the overall document processing rate. Reducing the time sent on this operator type is a first step to accelerate the overall document throughput.

The following sections will present and evaluate the hardware unit for dictionary matching. Section 3.1 will discuss the design requirements for the unit before section 3.2 will present the hardware architecture. Section 3.3 describes the compiler that is used to program the presented architecture. The design will be evaluated in section 3.4 before discussing related work in 3.5.

3.1 Design requirements

The design objectives for the dictionary matching unit are driven by the functional description of the operator as well as the existing implementation of the regular expression unit discussed in 2. The latter defines the way how input data shall be consumed by the pattern matching units as the text input can be shared among all of them. Because the regular expression unit consumes a single character every cycle the dictionary matching unit must follow this approach, though it could operate on entire tokens in a single cycle as presented by Agarwal et al. [14]. Although the token-based approach has a much higher throughput rate it limits the width of a single token to the one pre-configured in the hardware unit. Furthermore the overall processing rate would still be limited by the regular expression units thus no benefit can be gained by such an approach for the entire query processing architecture.

A functional requirement is the detection of patterns on pre-defined token boundaries. This means patterns may only be reported as a match if they start at the beginning of a token and match on the end of a token. This is similar to anchored regular expressions like e.g. ^abc$. The tokens are defined by software as tuples of two integers which the hardware unit must read to detect the individual tokens. Tokens may be single character wide and may follow each other immediately in the input text. The pattern detection requirement is extended by detecting sequences of tokens. The unit must be able to remember the occurrence of single token matches and report a match if a specified sequence appears.

When a pattern has been detected by the unit it must be reported on the output side. While many architectures are only required to signal that a match has occurred within a stream or file, this is not sufficient for text analytics as the subsequent operations require more information to function properly. Every match is required to be reported with the exact start- and end-offset location. The token-based operation may simplify this task but the appropriate offsets still need to be selected when it comes to multi-token patterns. In addition to the location information, a dictionary identifier needs to be reported. If multiple dictionaries contain the same pattern, every dictionary needs to be reported individually because the software wants the results grouped by dictionary in different memory locations to continue working with them. Also downstream relational operators in hardware require a valid tuple for each matching dictionary to function properly. These operators furthermore require their inputs to be sorted either by start or by end offset. Thus the dictionary matching architecture should support both output methods.

The aspect of scalability is neither negligible nor dominant. The architecture must support at least 5,000 patterns and should scale to as many as possible as dictionaries in text analytics can become arbitrarily large.

To summarize the requirements for the dictionary matching unit:

- Operate at a character-per-cycle rate to share the input data across multiple different pattern detection units
- Detect single token patterns only on pre-defined token boundaries
- Detect token sequences defined as multi token patterns
- Report all dictionaries containing the detected patterns with according offsets defining the position of the match
- Allow result reporting to be sorted either by start- or end-offset
- Support multiple large dictionaries with at least 5,000 patterns

3.2 Hardware architecture

The basic concept of the core architecture are two finite state machines (FSMs) connected in series. The first state machine is responsible for processing the input text document and detect single token wide patterns, while the second state machine is activated at every end of a token and reacts to the output of the first FSM to detect multi-token patterns. This type of architecture is similar to the concept of a decomposed automaton [78] which can be used to detect transition strings in regular expression patterns. This separates the problem into two parts, both having individual requirements and implementation options.

The input to the core are two data streams containing the document text data and the token definitions. The text data is stored as 8-bit ASCII characters and is streamed to the module one character per cycle. Although the ASCII encoding limits the number of the available characters it simplifies the design as every character is described in a single byte. Furthermore the UTF-8 standard includes ASCII as a subset, allowing the architecture to be extended to support UTF-8. The module uses an elastic interface with two handshaking signals valid and ready, where data is only accepted if both signals are asserted high. The token definitions are stored as tuples of two unsigned 32-bit integers and are passed to the core on a separate 64-bit wide bus using an elastic interface. Using 32-bit values simplifies the software integration as it standard datatype on many common platforms.

Figure 3.2 shows the overall dictionary matcher architecture. It is split into three sections: single token matching, multi token matching and result reporting. The single token matching part determines the processing rate of the text document and needs to synchronize the two input streams. It should have a deterministic operation and thus avoid backtracking or any other additional checks that need to be run when encountering a match. It is followed by the multi token section which operates on the results of the first stage. The multi token section is activated once per token and thus has a lower activity rate for natural language documents than the single token stage. Still the multi token part should finish within a single cycle or pipeline its operation to avoid stalling the input stage as a sequence of many single character wide tokens may appear e.g. in data logs. The last section handles the result reporting. It receives the matching results of the first two stages and generate the according outputs for them. As this stage may have to report multiple dictionaries for a single match the operation can take multiple cycles. The output of the core is composed of five integer values describing the dictionary identifier, the character-based start and end offsets and the token based start and end offsets. While the bitwidth of the character-based offsets remains at 32-bit the other values may be customized as they are used locally on the FPGA. If the dictionary id is required to be reported to software it will be padded to a 32-bit format. The result generation step will put the results on an elastic bus that holds the entire tuple to be communicated in a single cycle. It reacts to the ready signal controlled by the connected consumer.

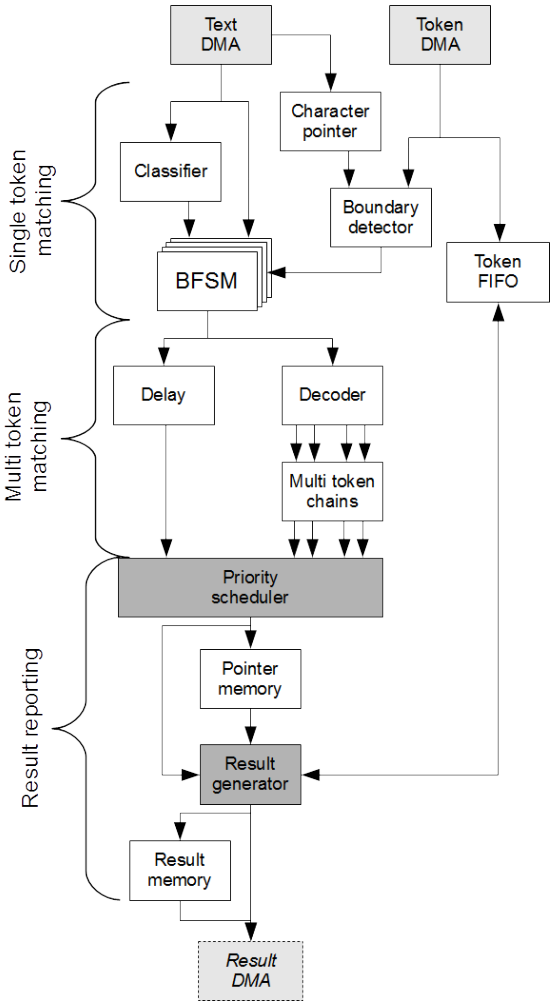


Figure 3.2: Dictionary matcher architecture

3.2.1 Single token matching

As a first step the single token matching (STM) part synchronizes the text document stream with the token offsets stream. To do so it uses an internal character pointer which represents the current position within the document. If both input streams assert their valid signals high, the character pointer is compared to the end offset of the token definition. If the pointer is less than or equal to it the character stream is accepted, while the token stream is advanced only if pointer is equal to the end offset. This keeps a constant stream of one character per cycle alive if both streams supply valid data fast enough. If the valid signal of either stream falls low, the other stream is stopped as well.

A boundary detection unit uses the character pointer and compares it with the token definitions to create start-of-token (sot) and end-of-token (eot) signals. These are single bit wide signals accompanying the character stream to indicate whether a character

is at the start or the end of a token or both for single wide tokens. These signals are important for the actual pattern matching unit and the activation of the multi token matching (MTM) part. Furthermore the eot signal enables a second counter for providing the current token id which is propagated with the token offsets to a buffer for use during the result reporting stage.

The pattern matching is performed by a finite-state machine (FSM) employing a deterministic finite automaton (DFA) constructed by a modified version of the Aho-Corasick algorithm [15] (AC). The AC-DFA contains a state for every prefix for every pattern in the dictionary where every transition is triggered by a single character. Every state is labeled whether it belongs to the dictionary or not. Additionally the AC-DFA contains fail transitions to other nodes that share the longest common prefix. This allows the automaton to evaluate additional transitions without backtracking.

The DFA is implemented using multiple Balanced Routing Table-based FSMs (BFSMs) by van Lunteren [99]. A BFSM is a programmable finite-state machine with transition rules stored in memory and uses a hashing function to efficiently distribute the rules in depth and width. A state is associated with a bitmask that is combined with the current input character to determine the next address to read from the rule memory. If the bitmask is all zero the address is directly determined by the 8 bit character value. To extend the address a table index or cluster index is added as the most significant bits. This cluster index is stored with every state node and determines in which rules cluster to operate. The transition rules are split into two groups: regular transition rules and default transition rules. The address generation for default transition rules does not rely on the current state and mask, but always uses the default state and a predefined mask.

An incoming character is passed to the classifier which assigns the character a 7-bit wide character class e.g. digit. On the FPGA this is implemented as a memory using the character as an address. The character is then used by the two address generation (AG) instances to produce an address for the transition as well as for the default rules. When the start-of-token signal is asserted high the remaining inputs to the transition rules address generator are set to default as the state-machine operates in an anchored mode. This means the state-machine can only leave the default state when the start-of-token signal is high, otherwise it will lock on the default state and shut down until the next start-of-token is asserted. This ensures that the FSM will only flag patterns that match with the beginning of a token.

Every entry in the transition rules memories consists of three rules with an increasing priority. At every cycle a complete rule line is read from both the transition rules and the default rules resulting in a total of six rules that are evaluated in parallel. The rules encode a test part and a result part [99] with an extended match information section that is variable in width depending on the number of patterns and dictionaries described in section 3.3. The test part of the rule defines what and how to test the input for. Whether to compare the character or the character class with the according test field, check for case sensitivity and/or test the current state or not. The tests are processed in parallel and then combined using a priority scheme where the first transition rule has the highest priority and the last default rules has the lowest. If none

of the rules match, the FSM will fall back to its default state and default mask.

If a rule is evaluated positively and contains a match flag, a pattern has been found. But the flag is combined using a logical AND with the end-of-token signal as only matches on the token boundaries are desired. Any match that occurs during the evaluation of a token is ignored. Once the end-of-flag has been asserted the FSM will shut down, continuing the process characters but waiting in its default state as the current characters are outside a token definition. If a match is valid, it will be signaled to the multi-token matching stage with the according match information encoded in the results part of the winning rule.

3.2.2 Multi token matching

The multi token matching (MTM) section of the architecture receives the match information determined by the single token matching (STM) stage. The MTM stage will always be enabled when the end-of-token signal is high regardless if there is a match or not. The stage will keep track of the matches that occurred over a sequence of single tokens and determine whether there was a multi token pattern matching a specific sequence.

To complement the resource usage of the single token matching stage the MTM stage is designed to utilize the logic and registers on the FPGA instead of memory blocks. The patterns are compiled to custom hardware blocks as described by Sidhu and Prasanna [89]. It implements a non-deterministic automaton (NFA) based on the Baeza-Yates or shift-and algorithm [22]. Every token sequence has its own register chain where the registers are interconnected using AND gates. While the output of a register drives one input of an AND gate, the second input is determined by the match information from the STM stage. This allows a bit flag to propagate through the register chain if the sequence of tokens is correct, otherwise the bit will fall low and not trigger the last output register high. As the chains operate on tokens the registers are clock-gated using the end-of-token signal.

The number of registers is equal to the number of tokens in a pattern. But as many patterns are searched for, shared prefixes can share the same resources. Also if different patterns belong to the same dictionary they can share the same resources by ORing the decoded match information or properly adjusting the decoding. Figure 3.3 shows an example circuit that detects the following three patterns: "multi token", "multi token match" and "multi token pattern". The patterns share the resources for their common prefix by adding a single OR to the last stage for the patterns of length three. The results are propagated to the result reporting stage.

3.2.3 Result reporting

After the successful detection of a dictionary element the results need to be produced. A single result is composed of a 32-bit dictionary identifier and the start and end offset position of the element found in the current document. As a single element may be

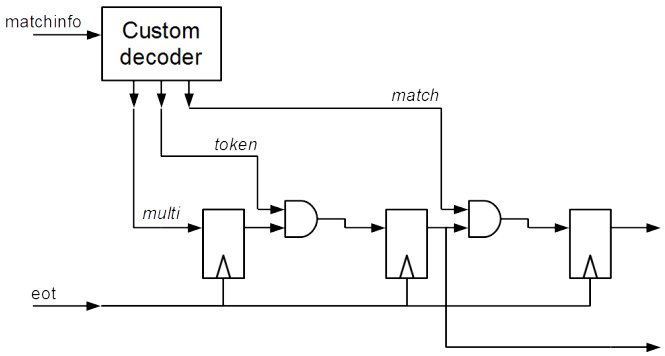


Figure 3.3: Multi token matching architecture

contained in multiple dictionaries, multiple results need to be produced. This implies that the amount of data generated by the dictionary matcher may exceed the size of the actual document processed. A cascaded result lookup is used to efficiently store the information in which dictionaries an element is contained in.

A match in the pattern detection engines results in an address derived from its matching state to the pointer memory. This memory contains a single entry for each match from the BFSM and the multi token chains. Each entry consists of an address to the result memory and a result length. The result memory contains the actual dictionary IDs, which are grouped by a particular element appearing in such a group. For instance if an element appears in dictionaries A, B and C then these form a continuous group in the result memory. A second element appearing in dictionaries D, E and F forms a separate continuous group. But for a further element appearing in B, C, D and E, the compiler will create a group overlapping the previously created ones, thus minimizing the necessary storage. Figure 3.4 illustrates this example.

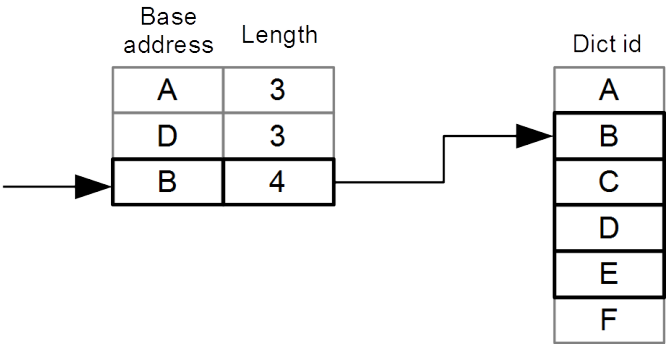


Figure 3.4: Cascaded result lookup

The result generator receives the base address and the result length and generates the addresses to the result memory. It also receives the number of tokens involved in a match and assembles the correct token offsets from the token FIFO and writes the

results to the output. This operation runs at full clock speed with no clock gating as it is independent of the end-of-token signal except for acknowledging the Token FIFO. Still the result generation may take more cycles and cause backpressure to the matching stages. To avoid the document processing from halting the single token matching stage is separated from the multi token matching stage by shallow FIFOs.

The output results are always sorted by their end offsets first and then by their start offsets. This is simply done by using a priority scheduler where patterns with more tokens have a higher priority. If a downstream requirement is to have the output sorted by start offset first, the priority scheduler inserts additional delays to shorter token patterns while keeping the scheduling mechanism. This way a single token match arrives later at the scheduler allowing the MTM stage to detect any longer patterns meanwhile. Finally the result generator needs to be aware of this setup as well, to correctly select the offsets from the Token FIFO.

3.3 Compiler

The compiler framework for the dictionary matcher architecture is composed of two parts. The outer part is written in Python and is responsible for processing the input files and generating the output files. Furthermore it generates the Verilog hardware description for the multi token matching architecture. The framework employs the regular expression compiler (RegX) by Rohrer et al. [86] to generate the initial transition rule memories for the BFSM engines. Figure 3.5 shows the basic steps of the compiler and their data flow.

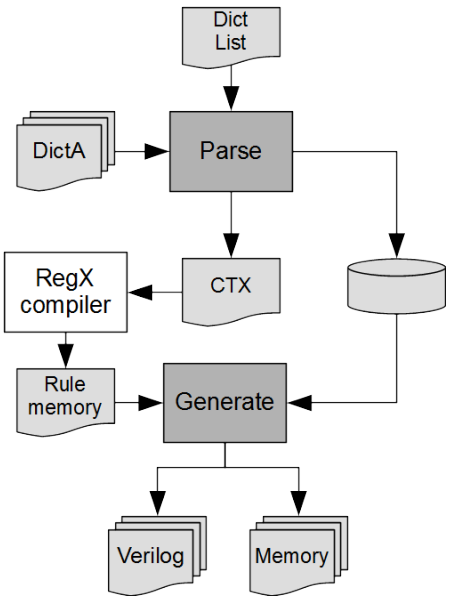


Figure 3.5: Dictionary compiler framework overview

The compilation process starts by reading a plain text file containing a list of paths to dictionaries to compile. Additionally each dictionary entry has an option flag to indicate whether or not the dictionary should be matched case sensitive. Every dictionary file contains one string pattern per line, where multi token patterns are split by whitespaces on the same line. The compiler will split the multi token patterns into single token patterns and consolidate them. Every single token pattern is assigned a unique id and the sequence of ids is stored for later use to generate the multi token matching architecture. It will then create the input file for the regular expression (RegX) compiler [86] and launch it.

The RegX compiler will compile the individual patterns by distributing them among multiple BFSMs and generate the according transition rule memories. Additionally it sets the value for the default mask of the BFSM and produces a results table providing information about which state on which BFSM belongs to which pattern id. These files are then read by the dictionary compiler again to generate the final output files. The BFSM rules are modified such that a multi token id is inserted on the appropriate matching rules. At the same time a custom decoder for the multi token chains is constructed as a Verilog file, which decodes the multi token id to a one-hot bus. This then allows the compilation of the multi token chain registers which are all compiled to single large Verilog file.

As a last step the result memory file is assembled. During the parsing step the compiler collected information about shared patterns and uses a simple heuristic to minimize the required memory space. It will start with the patterns with the most occurrences in different dictionaries as patterns that occur in only one dictionary can then simply point to an already existing entry if it exists. This will produce a longer sequence in the results memory where the dictionary ids are sorted in ascending order. The compiler will then iteratively add new id sequences by first sorting them and checking if they already exist in the results memory and create the corresponding entry in the pointer memory. If the sequence does not exist the compiler appends it to the end of the results memory. Although this may not lead to the optimal solution it provides an efficient solution in terms of processing time and result.

3.4 Evaluation

The dictionary matching architecture is evaluated under two aspects: performance and scalability. For both aspects the architecture is integrated with a POWER7 host using a predecessor of the Coherent Accelerator Processor Interface (CAPI) used in chapter 6. The accelerator card is directly attached to the processor bus (GX) and holds two FPGAs. One system FPGA that links the processor bus to an application FPGA which holds the actual accelerator logic. The application FPGA is an Altera Stratix IV GX530 FPGA running at 250 MHz and is connected to the system FPGA using a custom interface capable of 4 GB/s in both directions. The accelerator card can access the host's main memory via the processor bus. The application FPGA operates on virtual 64 bit addresses that are resolved to actual physical addresses by the system

FPGA in cooperation with a translation server running with the software application. Basic memory copy tests indicate a maximum bandwidth of around 3.4 GB/s for the system. Figure 3.6 illustrates the setup.

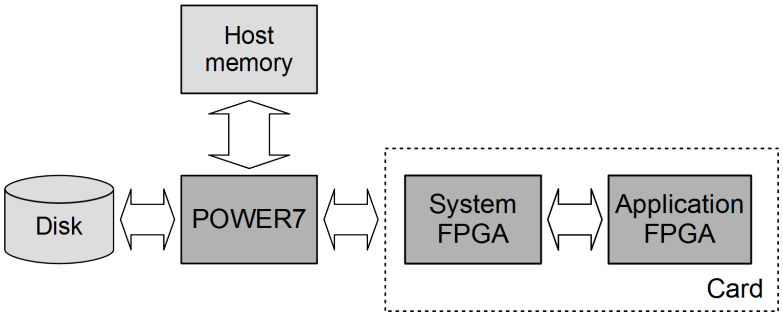


Figure 3.6: Evaluation setup for the dictionary matcher

The exerciser application used for performance evaluation is written in Java and the measured time is taken from the submission of the first document to the reception of the last results. The application will read a set of documents into the host memory and generate the token information to avoid any file I/O operations. To run for a sufficiently long time (>10 sec.) the exerciser will resubmit the documents to the accelerator card for a given number of times. This will adjust for any effects imposed by the operating system and the Java virtual machine. For the performance test a single thread is launched that submits a batch of documents to the hardware. Once half the number of documents are returned the thread will resubmit the batch of documents again to ensure the hardware queue is never empty.

The dictionaries have been taken from the Moby Word List [102] by Grady Ward. It is a collection of 16 dictionaries with different types such as names, places and other words. For the evaluation a subset with different properties has been selected and is summarized in table 3.1. The selection comprises the complete *NAMES* dictionaries of the Moby word list that represent basic sizes. Additionally two dictionaries have been constructed from the first 50,000 and 100,000 words of the *SINGLE* word list to stress the single token detection part of the architecture. To evaluate the multi-token matching part three dictionaries have been generated from the first 1,000 to 3,000 multi token words in the *COMPOUND* list.

3.4.1 Resource requirements

The resource requirements of the architecture determine the scalability of the design and the maximum size of the dictionaries that can be used. While the single token matching stage uses mainly memory resource to implement the transition rules of the AC-DFA the multi token matching stage relies on a large number of logic resources and registers to implement the NFA. These properties of the architecture complement each other to achieve a high utilization of the FPGA resources.

Dictionary name	# of words	# of characters	avg. char. / word	# of MT words
NAMES-M	3,897	27,518	7.06	1
NAMES-F	4,946	36,329	7.34	5
NAMES	21,986	158,275	7.2	7
SINGLE-50000	50,000	507,249	10.14	0
SINGLE-100000	100,000	1,034,613	10.34	0
COMPOUND-1000	1,000	27,338	27.34	1,000
COMPOUND-2000	2,000	42,294	21.25	2,000
COMPOUND-3000	3,000	57,575	19.19	2,000

Table 3.1: Dictionaries used for evaluation

Figure 3.7 shows the percentage of resources used on the Altera Stratix IV GX530 FPGA over the number of characters. For the single token matching stage it can be seen that the use of logic cells (ALUTs) stays below 10 % even for 800,000 characters. These were 32 instantiated BFSMs consuming about 70 % of all memory blocks (M9K) on the FPGA including the classifier and other memories in the architecture. Also a small number of output buffers used for the output channel communication is included in this numbers but stays the same for all evaluation runs.

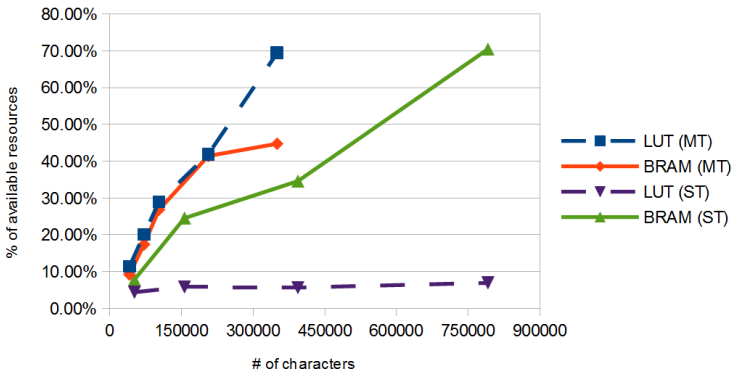


Figure 3.7: Resource consumption in percent vs. number of words

To formalize the design efficiency of the design two measures can be established. One is the logic cell efficiency which can be calculated by dividing the number of total required logic cells by the number of characters (Equation 3.1). And the memory efficiency which is the number of required memory bits per character (Equation 3.2).

$$eff_{lc} = \frac{LC_{total}}{chars} \tag{3.1}$$

$$eff_{mem} = \frac{bits_{total}}{chars} \tag{3.2}$$

On average the presented architecture achieves a logic efficiency of 0.35 LC/char and a memory efficiency of 25.5 bits/char. These values can be used to compare different types of architectures with each other. Section 3.5 will discuss related work and compare these numbers.

3.4.2 Performance

The primary performance measure is the character throughput of the architecture. As the compiler is limited to the ASCII character set the unit of characters per second is equivalent to bytes per second as every ASCII character has a size of exactly one byte. The throughput of the architecture indicates how fast a set of documents can be processed. The interface link to the dictionary matcher must also sustain the transfer of token definitions in parallel to the document characters but is not part of the reported throughput numbers.

The maximum peak processing rate can be calculated using equation 3.3. Every stream in the architecture consumes a single character per cycle thus the frequency determines the processing rate of a single stream. The throughput multiplies with every instance of an additional stream. While a single instance of a dictionary matcher is designed to process two character streams, the second one can be removed if i.e. there are insufficient logic resources two implement a second multi-token matching stage.

$$T_{max} = f * n_{streams} \tag{3.3}$$

The number of possible streams is limited by the available logic and memory resources. Limiting factors that may reduce the actual throughput are the saturation of the processor bus interface or transfer setup penalties when translating virtual addresses to physical ones. Both scenarios have a similar effect that the document and token streams have to wait for each other. Figure 3.8 shows the throughput for different number of streams over the individual document size.

The dependency from the document size can be explained by the virtual to physical address translation. For every document a new transfer has to be initiated and thus a new translation has to be requested. In the used CAPI predecessor system this translation happens via a software translation server performing the lookup and sending back the physical address using an MMIO write. This is causing a transfer setup penalty that is limiting the performance for small document sizes.

To improve the situation for small documents a batching mechanism has been introduced which groups four documents into one larger buffer before sending it to the FPGA. The effect can be seen in figure 3.8 between the two measurements for 2 streams. The batching mechanism helps the smaller documents (<1000 bytes) but also allows larger documents to reach the peak processing rate earlier.

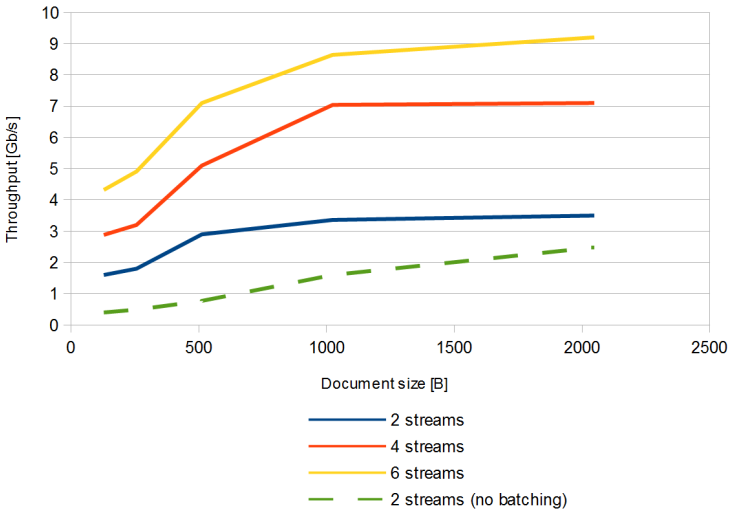


Figure 3.8: Throughput over document size for different number of streams

The maximum measure throughput was 9.2 Gb/s using six streams which is an equivalent of 1.15 giga characters per second. This is 76 % of the maximum theoretical peak processing rate and indicates a limitation on the interface side. A single instance of a dictionary matcher using two streams reached 3.5 Gb/s which is 87.5 % of the theoretical maximum.

The original Java-based software implementation reaches a maximum throughput of 51.44 Mb/s on an Intel XEON E5530 with 2.40 GHz using 16 threads. The software measurements use the same dictionaries and documents. This is an up to 178x improvement using the presented architecture.

3.5 Related work

String pattern and regular expression matching has been studied for a wide range of applications and on different platforms. FPGA implementations have been driven by the network intrusion detection (NID) community to identify malicious network streams. Such architectures do not have the requirement to report the exact positions of a match but rather flag a stream once a pattern has been detected. With the ever growing traffic in global networks other implementations for NID systems were done in ASIC or event custom processor technology. To compare the different architectures with each other two common metrics can be used:

- Memory efficiency in terms of how many bytes are required per character or pattern
- Aggregated throughput, which is the overall throughput of all streams through

the architecture. This may also be dependent on the number of patterns.

An alternative FPGA implementation for dictionary matching for text analytics is presented by Agarwal et al. [14]. It uses a collision free hashing architecture to determine if a token is contained within a dictionary. The size of the lookup table is determined by a predefined maximum length for a single token as this determines the hash size. This architecture implements its own tokenization circuitry, splitting the document text on whitespaces. Furthermore the design is not capable of multi token matches and multi dictionary reports. Despite the missing features the implementation achieves high throughput rates by processing every token in pipelined cycles.

Nakahara et al. [78] discuss an architecture based on a decomposed automaton for regular expression matching. The regular expression is converted to an NFA and the number of states is reduced by merging and concatenating transition strings creating a modular non-deterministic finite automaton with unbounded string transition (MNFAU). The transition strings are then detected by a DFA implemented using off-chip memory while the MNFAU is implemented in FPGA logic using and AND-shift architecture. The design consumes a single character per cycle and runs at 200 MHz, while being very efficient in terms of logic cells per character and memory per character.

Yang et al. [111] also take a combined approach to large scale string matching. To reduce the number of backward transitions in an Aho-Corasick DFA (AC-DFA) a binary search tree (BST) architecture is used to find a subset of strings and prefixes. Once the BST reaches an output state it can trigger a tail root node, which is a starting node of a AC-DFA that is implemented in a second part of the architecture. The architecture has a guaranteed worst-case performance and achieves high throughput rates for very large string sets.

A software implementation on the Cell Broadband Engine processor achieving good performance is presented by Scarpazza et al. [88]. By properly parallelizing the AC-DFA algorithm and carefully timing the memory transfers between the processing engines by using DMA primitives this software implementation achieves an aggregated throughput of 3.7 Gb/s for a set of 20,000 words.

Van Lunteren et al. [100] present a regular expression co-processor based on BFSM technology that was implemented as part of IBM's Power Edge of Network processor in 45nm SOI custom technology. The implementation focuses of regular expression matching for network intrusion detection. Two co-processor instances where implemented running at 2.3 GHz, achieving up to 40 Gb/s scan rates. To achieve high memory efficiency for various regular expressions a local result processor is used as a post-processor to the BFSM units. This allows to split patterns into smaller parts that can be stored more efficiently in memory. If a match occurs the co-processor is able to report the type of match and the end offset but then hands off to the main processor to further evaluate the stream.

Another network intrusion detection architecture is presented by Yu et al. [112] which is based on a ternary content addressable memory (TCAM). The TCAM can be used to directly find a pattern based on the input string. To increase the memory efficiency

the string patterns are split into prefix and suffix patterns to achieve a similar pattern length. The architecture can also handle correlated patterns and a subset of regular expressions at a scan rate of up to 2 Gb/s.

Other network detection intrusion detection systems are presented by Song et al. [93] as ASIC design, and FPGA designs by Bispo et al. [26] and Yang et al. [110]. Table 3.2 compares some architectures in terms of maximum reported performance and resource efficiency. It indicates that the presented design is well balanced in terms of resource utilization and achievable throughput, as it does not strongly outperform nor underperform in any category.

Method	Type	Gb/s	LC/char	bit/char
Sourdis 05 [94]	Dict	12.67	16.86	n.a.
Bispo 06 [26]	Regx	2.9	1.28	0
Yang 10 [110]	Dict	3.5	n.a.	8.4
Nakahara 11 [78]	Regx	1.6	0.25	21.4
Agarwal 13 [14]	Dict	17.8	0.4	21.2
BFSM+NFA	Regx	9.2	0.35	25.5

Table 3.2: Dictionaries used for evaluation

Figure 3.9 illustrates the performance of various architectures for string and regular expression matching over the number of patterns to be searched for. Although the architectures provide different levels of features, the figure shows the tradeoff between capacity and performance. The presented architecture [82] shows strong performance per pattern numbers getting close to the ASIC implementation by Song et al. [93].

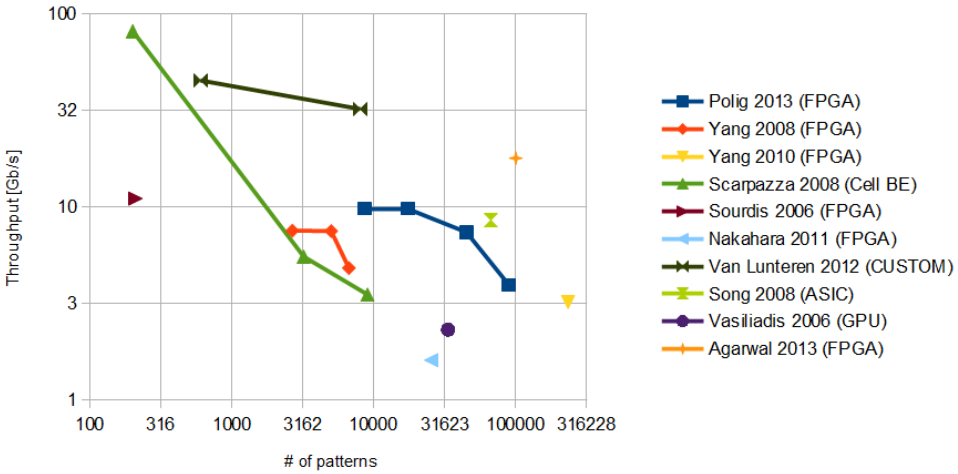


Figure 3.9: Dictionary matcher comparison

3.6 Summary

Named entity recognition (NER) is a key processing step in natural language processing. Individual string tokens are assigned one or multiple categories such as e.g. person or country. These categories refer to large lists of string patterns so called dictionaries. The patterns may only be matched on the token boundaries which are defined by a preceding tokenization step. Profiling of text analytics queries shows that NER is one of the most time consuming processing steps including regular expression and dictionary matching. Multiple hundreds of dictionaries may contain multiple thousands of patterns that are required to be found.

While previous work on string pattern matching has been focused on network intrusion detection systems (NIDS) where only the sole occurrence of a match is required to be reported, pattern matching in text analytics requires all match locations to be reported. Furthermore the sequence of pattern matches may generate additional matches so called multi token matches that also must be reported. If the same pattern appears in multiple dictionaries, every dictionary must be reported. This can be done by generating superset dictionaries or by the hardware architecture.

A dictionary matching design was presented to fulfill all the requirements of the text analytics processing step. It is composed of two matching stages where the first stage processes the document at a deterministic rate of one character per clock cycle and detects single token patterns. This stage is implemented as a programmable state-machine mainly utilizing the FPGA's block memory resources. The second stage implements a shift-and algorithm in custom logic to perform multi token matching. This stage complements the first stage in terms of area by utilizing mainly logic resources on the FPGA.

Evaluation has shown that despite the additional features of the architecture the design is capable of high performance large scale named entity recognition. The architecture was able to process 100,000 patterns at 3.2 Gb/s which is up to 178 times faster than the original software implementation.

CHAPTER 4

Relational operations

This chapter will discuss relational operations performed by text analytics queries and present a hardware library and compiler [81] to run such operations in FPGA logic, together with the extraction operators.

The relational model was introduced by Edgar Codd in 1970 [36]. It is a theoretical data-model based on a table-like data-structure called a relation, which consists of attributes organized as columns and individual data entries organized as rows called tuples. The attributes are defined by a header, and define the attribute's name and datatype. The relational model provides a declarative method to define the structure and retrieval of data, and does not define how it is implemented by a specific application. A common application is a relational database, which is organized according to this model. Most relational databases use the structured query language (SQL) to define and interact with the data. Figure 4.1 visualizes the concepts of the relational model.

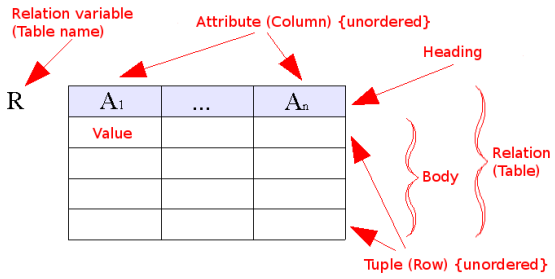


Figure 4.1: Concepts of the relational model [13]

To retrieve data from relations, queries can be formulated to define the desired data. Queries use relational operations to combine, filter and generate the requested result set from a number of relations. To select only a specific set of attributes from a relation

the *Project* operation is used. The restriction operation filters the rows that are not applicable to a given predicate, this operation is often also called *Select*. Multiple tables can be combined by generating the Cartesian product of two tables, where every tuple in one table is combined with every tuple from a second table. Usually the resulting product is filtered according to a given predicate. The combination of product generation and filtering is referred to as a *Join* operation. Additional operations accessing two or more tables are, *Difference* where all tuples from a first table are filtered if they appear in the second one, *Intersect* which generates a set of tuples that appear in both tables, and *Union* which creates a single table from multiple one that have the same structure.

In SystemT the initial tables are either created by the user and static for the entire query or generated by the extraction operators for each individual document. These tables contain attributes of type Span, but may also contain integer, float or string values. A particular sequence of attributes defining the header is called a *Schema* and is always associated with the output of an operator. SystemT provides relational operations with additional specific features such as pattern matching for the *Select* operation or special versions of *Join*. Also consolidation and grouping operations are provided to remove duplicate entries or count appearances.

The next section will define the design requirements for the implementation of the relational operators on the hardware before section 4.2 will describe the architecture and operation of the individual hardware modules. The compilation process is presented and discussed in section 4.3, before evaluating the design in section 4.4. Related work is outlined in section 4.5 and the chapter is summarized in 4.6.

4.1 Design objectives

As seen in chapter 2 the relational operators follow the extraction operators in an annotation operator graph (AOG). The extraction operators determine the rate at which documents are processed by the hardware and should not be stalled during their operation. As the output of the extraction engines is sorted by either start or end offset, many relational operations can be performed in a streaming fashion with some additional buffering, if necessary. This streaming type of processing should ensure that any tuple generated by the pattern detection units (PDUs) can be accepted by the following logic implementing the relational operators.

While the output tuples of the extraction operators always have the same schema of a single span type describing the position of a match, the schemas are altered by the relational operators. As the operators can be interconnected in any way, the hardware modules implementing the operators must be configurable to any arbitrary schema. The attributes used by the operator may also be located at any position within the schema, while the remainder of the schema is considered payload.

To process entire queries, a compiler framework must be provided to create and configure the hardware operators and interconnect them in the appropriate way to represent the annotation operator graph. The compiler must ensure the proper functionality of the

operators by adjusting settings of parameterizable modules or create custom modules, if necessary. Furthermore the compiler needs to check the order of the tuples on a graph edge to ensure the operator can process the stream in a single pass. Also the flow-control must be inspected by the compiler when a single operator has multiple consumers.

In order to process large queries or achieve higher processing rates, the compiler must apply optimization strategies to the annotator graph before creating the hardware description. Because of the sequential nature when processing a query in software, the runtime may skip entire parts of the graph when no results are produced at a given node. The hardware does not benefit from such a strategy and must leverage other insights to reduce resource consumption and increase frequency. Especially identifying unnecessary payload attributes or unused parts of an attribute, can have a significant impact on the area requirements of hardware operators.

Summarized the requirements for the relational operators framework are:

- Stream based execution of relational operations, to avoid stalling the extraction operators which provide the input results
- Operators must be configurable to arbitrary schemas, as they can be interconnected in any way
- Compiler framework that creates the individual operator modules and generates complete query graphs in hardware
- Optimization techniques to reduce the overall hardware resource consumption to allow complex queries to be mapped and reduce the static power consumption

4.2 Hardware modules

The basic functionality of every supported relational operator is implemented as a separate hardware module that can be configured by setting a range of parameters. Operators that are highly configurable such as the *Select* operator are compiled to custom HDL code. As the modules need to communicate with each other they all implement the same interface on their top-level. The interface is defined as an elastic interface, which allows the producer to indicate whether or not the data is *valid* and the receiver whether or not it is *ready*, to accept the data in the current clock cycle. This scheme also allows to insert register stages into both the valid and ready direction to achieve the desired target frequency.

The data is transported on a wide data-bus. The width of the bus is determined by the output schema of the driving operator. In order to decouple the design of the core modules and from the schema layout, a wrapper is generated for each operator instance which rearranges the attributes of the schema. Actively required attributes will be located on the most significant end of the data-bus while the remainder will be classified as payload for the actual module. On the output side of the module the wrapper will again rearrange the outputs to correspond to the correct schema of the

operator instance. Figure 4.2 shows an example of a wrapper module. The active column is moved to the left for the input and restored for the output of the module. Any newly generated columns get appended to the right.

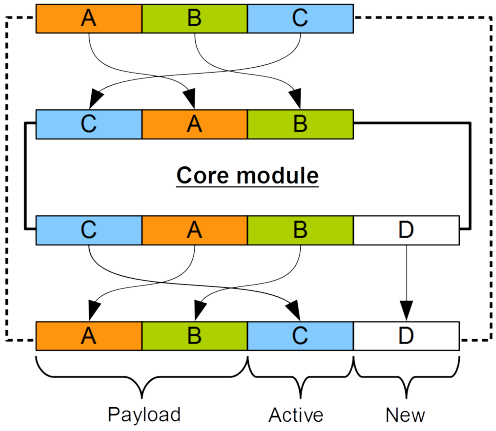


Figure 4.2: Wrapper module rearranging input and output schemas

The span data type, is represented as four integer values. The first two values are the character-based start and end offsets within a document. These values are 32 bits wide, each. The second pair are the token-based start and end offsets, and are 16 bits wide. These token ids are used internal to the hardware only, and simplify token-based range operations to avoid additional lookups.

An additional bit, which is treated the same as the data-bus, is the *end* signal. It indicates the end of a stream and must be asserted by each operator after it has produced the last tuple. The *end* flag is first generated by the extraction operators, after the last character of the document has been processed. Data on the data-bus is invalid when the *end* flag is asserted high.

4.2.1 Adjacent Join

The Adjacent Join operator is a specialized and heavily used Join type in text analytics queries. It calculates the product of two sets and uses a distance metric to filter the appropriate candidates. The distance metric is defined as the difference between the end token id of a span from set A and the start token id of a span from set B. If the distance lies within a user-specified range the two tuples from both sets are Joined and put into the resulting set. The distance may also specified as the distance between a character-based end and start-offset, in which case the operator is named Sort-Merge Join, which is a known type of Join algorithm.

In both cases the core hardware implementation is the same, taking two streams A, B as inputs and producing an output stream with the joined schema. This means the total data bus width remains the same for input and output. Figure 4.3 shows the operator

representation in the annotation operator graph on the left and an example on the right. The operator node has a secondary type node added to it describing the Join criterion in terms of distance and which set follows the other. The compiler will read the secondary node to correctly wire the wrapper module. Additionally the compiler sets the correct values for the minimum and maximum distance.

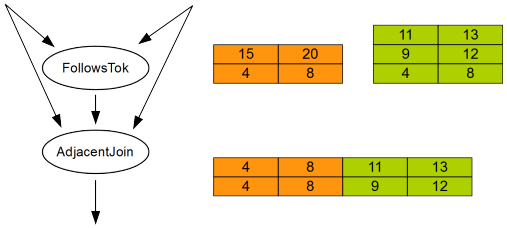


Figure 4.3: Adjacent Join operator node and example

The hardware operator benefits from the ordering of the tuples on the two input streams to process them in a streaming fashion. In the case where a tuple from stream B follows a tuple from stream A, stream A must be sorted by end offset while stream B must be sorted by start offset. Advancing stream A will always shorten the distance or keep it constant, while advancing stream B will only increase the distance. As multiple tuples from A may be joined by multiple tuples from stream B, the distance checks must be performed multiple times on the same tuples from one set. This requires at least one stream to be buffered again to a temporary area.

Figure 4.4 show the overall core architecture of the Adjacent Join module. Both incoming streams are buffered to individual FIFOs, as it cannot be foreseen when tuples arrive on the inputs. Once data is available in both FIFOs, a distance check is performed in a single clock cycle and determines the read and write operations on all buffers. If the calculated distance is within the user-defined range, the tuple is moved to the output and the valid signal is asserted high. In this case the tuple from stream B is also written to a temporary FIFO (T) as it may join the next tuple from A again. This process is repeated with a new tuple from stream B as long as the distance does not surpass the maximum defined distance. Once that happens, the first tuple from set A is discarded and the next one will be used. First all tuples from buffer T will be tested with the new tuple from A, and are run through the distance check. If the distance gets negative, the tuple is discarded from the temporary buffer as there will be no tuple in stream A with an earlier end offset. Once all tuples have been processed from the T FIFO, the process starts looking at tuples from stream B again.

This architecture is not suitable for generic Join as the temporary buffer has only a limited capacity to perform the product operation. In the event of a buffer overflow the first tuple of any FIFO is dropped to continue processing the document. In the general use-case this does not result in wrong results due to the linear processing of the text document. This means the distance of the first tuple in a full buffer to a first tuple of an empty buffer will be out of the valid range. But to inform the software of the event a special exception flag is set and it is left to the application whether to rerun the

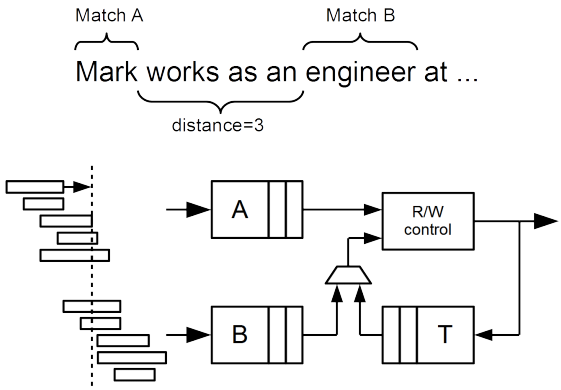


Figure 4.4: Architecture of the generic Adjacent Join module

document in software only, or ignore the exception.

4.2.2 Difference

The Difference operation takes two sets as input and removes any tuples from the first set if they appear also in the second set. The output schema is defined by the input schema of the first set (A). The hardware module does not require any parameters to be set by the compiler other than the width of the test input. The compiler will rewire the wrapper such that the attributes required for the equality test are fully sorted. The rewiring may switch start and end offset of a span depending on what the stream is sorted by first. Figure 4.5 illustrates the operator node and an example including wrapper rewiring.

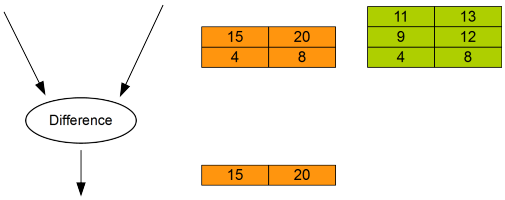


Figure 4.5: Difference operator node and example

The core architecture is similar to the Adjacent Join operator without the additional temporary buffer. The input streams are put into FIFOs as the streams need to wait for each other. The output from buffer A will always be presented as output data but it is not validated until a greater tuple arrives on stream B. Once valid data is available in both buffers a comparison operation is carried out in a single cycle to determine which buffer will be advanced and whether the output tuple is validated. Stream B is advanced as long as its tuples are smaller than the current tuple from buffer A. If the tuple from stream B is greater than the current tuple from A, the output valid signal is asserted

high and kept until the receiver accepts the tuple. Then stream A is advanced and the test begins again. If the tuples are equal, stream A is advanced as well but without validating the output.

As the Difference operator synchronizes two streams it may run into the same deadlock situation as the Adjacent Join, when a single buffer overflows. To resolve the situation a similar mechanism is used by dropping the first element of an overflowing buffer but if stream A overflows the elements are pushed to the output as it is likely that there is no equivalent tuple in set B. Again an exception flag is raised to allow the software application to react to such a scenario.

4.2.3 Union

The Union operator takes a number of sets with an equal schema and combines them to a single set. On the hardware this is equivalent to multiplexing multiple data-buses onto one. If the receiving operators do not require any special ordering of the output stream a round-robin arbiter is used to merge the streams. To increase the performance of the input acceptance rate a multi-staged implementation can be instantiated. The compiler can decide to set a stages parameter higher than one to do so. This might also be necessary for very wide unions of more than eight streams. Figure 4.6 depicts the operator node in the AOG and an example.

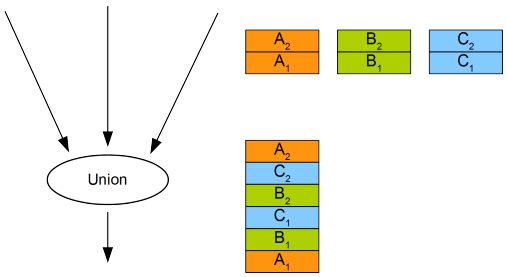


Figure 4.6: Union operator node and example

If the output is required to be sorted in a specific way a specialized version of the Union architecture is instantiated by the Compiler. This is a multi-staged version where at every stage a comparison is carried out for the required attribute and determines which tuples to advance first. This requires all streams to be valid before a decision can be made about the order of the output. Figure 4.7 outlines the architecture for this type of Union.

An alternative implementation with less resource consumption is to multiplex the streams in a round-robin fashion into one buffer and sort them before they are sent to the output. This type of architecture has a higher latency when producing outputs but it can accept incoming data without synchronizing the input streams.

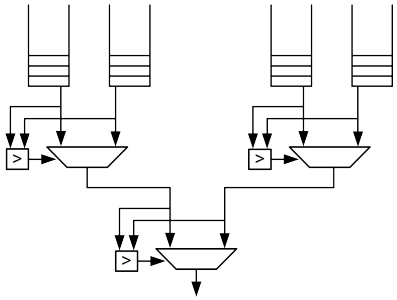


Figure 4.7: Architecture of a sorted Union

4.2.4 Select

To filter tuples from a single set based on an arbitrary condition the Select operation is used. For each tuple in a set it evaluates a condition that does not depend on any other tuple in the set. But the condition may operate on multiple attributes from the set's schema. To describe the condition or predicate that needs to be evaluated the Select node in the annotation operator graph is enhanced with a set of secondary nodes as shown in Figure 4.8. The compiler will take these nodes and transform them into appropriate Verilog code.

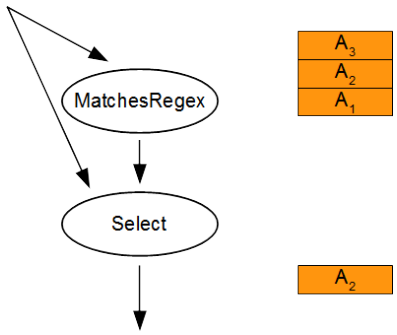


Figure 4.8: Select operator node and example

This means the condition evaluation core is an individually compiled hardware module. The top-level however is static that controls and schedules the stream flow and condition evaluation. Figure 4.9 illustrates the top-level design with the stream FIFO that is required to buffer the incoming tuples as the condition evaluation core may take several cycles before returning with a result. The read control unit is responsible for the communication between the buffer the evaluation core, as well as validating the output. In case of a full FIFO buffer the module will set its ready signal to low and not accept any input data until the current condition evaluation has finished.

Although the condition evaluation is a blocking operation and takes several clock cycles, it is sufficiently fast to keep the tuple buffer from overflowing. The compiler is given

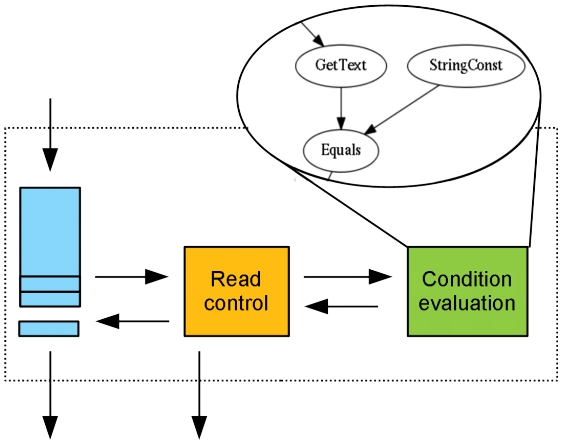


Figure 4.9: Top-level architecture of Select

additional profiling information about the rate at which tuples enter the operator. Typical numbers range from less than 0.1 to about 16 tuples for an average document size of 250 bytes. The compiler can then size the FIFO buffer accordingly to avoid excessive overflows.

The condition is described by the user via AQL statements. They can range from simple offset comparisons to evaluating multiple regular expressions over adjacent contexts of a span. These sub-conditions may be combined using Boolean logic to create the final condition flag. The user may specify to test the left or right context of a span, in terms of number of tokens to the left or right. In such a case the condition module will instantiate a token id buffer to perform a lookup of a token id to retrieve the respective character based offset. This way a new span is assembled that can be used for further processing. Figure 4.10 illustrates the architecture for the token buffer. The buffer will fill up until it is full and then drop the oldest token when a new one is written to it. If an access to a lost token is made an exception is raised but the oldest token is taken to continue processing. If an access to a token is made that is not yet in the buffer, the control logic will wait until it arrives.

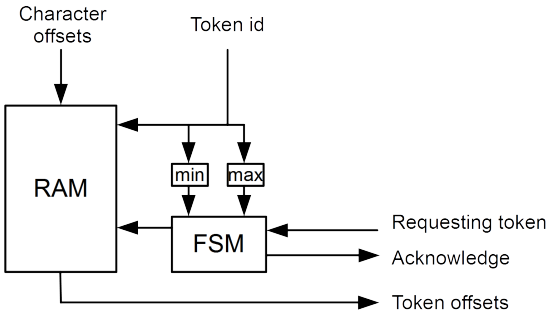


Figure 4.10: Architecture of the token buffer

If the condition requires to evaluate a regular expression or a string match of a span, the hardware module needs to access the actual document data as well. To do so a rolling buffer is instantiated that behaves equally as the token buffer by dropping the oldest character once a new character is written to the full buffer. This resembles a sliding window over the text document and provides sufficient range to retrieve document data as the incoming spans arrive in a linear order. The readout of a data is controlled by applying a span on the read control of the document buffer. The read control will then generate the necessary addresses to retrieve the entire span, allowing back pressure if necessary by the receiver.

4.2.5 Consolidate

Consolidation is another filter operation that is applied to a single set and handles duplicates or overlaps. This requires the operator to look at multiple tuples to determine whether the tuple in question is filtered out or not. To enable a streaming execution the hardware module leverages the fact that the tuples are sorted on the inputs. This allows to compare only two tuples and decide whether one of them can be eliminated. The operator can have different policy on which to consolidate the tuples. The most common cases are Exact Match and the Contained Within policies which will be described in more detail. Figure 4.11 shows the generic operator node in the annotation operator graph. The policy is annotated as an attribute to the node and read by the compiler to instantiate the proper hardware module.

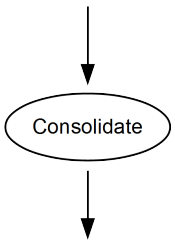


Figure 4.11: Generic Consolidate node

Exact Match

The Exact Match policy will only keep one tuple of multiple equivalent tuples in a set. Thus the hardware module will only validate the output data when consecutive spans are note equal or the end of stream has been reached. For the operator to work properly the data needs to be fully sorted either by start or end offset, the compiler will adjust the wiring within the wrapper to ensure that. The core architecture is made up of two register stages where the tuples are pushed through. The output valid signal is controlled by a wide comparator that will test the test field for equivalence. For a span only the character based offsets are compared as they are a superset of the token ids. If the two tuples are equal then the valid flag is not raised but the tuples are shifted by

one stage, therefore dropping the tuple currently held in the output stage. If the two tuples are not equivalent the valid signal is asserted high and the operator halts if the ready signal is low. Figure 4.12 illustrates the architecture.

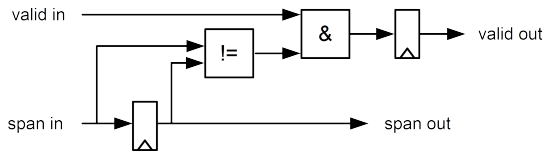


Figure 4.12: Architecture of an exact match predicate

Contained Within

To remove any spans that are contained within another span the Contained Within policy is used for the Consolidate operator. If for the span in question there is another span in the set with an equal or greater start offset and an equal or smaller end offset, the span is removed. For the hardware to operate properly the input stream needs to be sorted by start offset. The core architecture has two stages. The first stage captures the longest match for a unique start offset since the longest span contains all spans with the equivalent start offset. The resulting spans are passed to a second stage which removes all spans having an equal or smaller end as they are contained within an earlier span that had a smaller start offset.

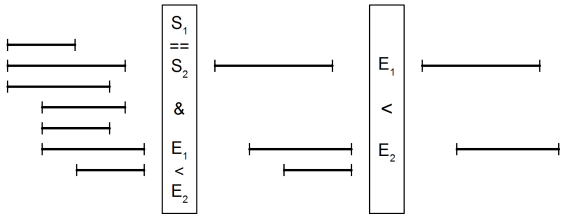


Figure 4.13: Two stage filtering for Contained Within consolidation

4.2.6 Apply Function

The Apply Function operator node runs a scalar function on every tuple of a set. Some of the functions are not suitable to be executed on the accelerator as they perform operations to setup auxiliary data. An example for such a case is the GetText operation which will add the string value to the schema, for a given span. This data is irrelevant for processing the query on the FPGA and often appears at the very end of an annotation operator graph to setup a specific output schema. This task remains in software and will not be pushed to the accelerator. The generic operator node is shown in Figure 4.14. The operator works on a single set and creates an additional attribute which is added to the output schema.

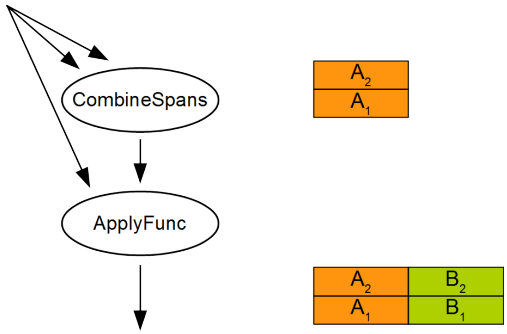


Figure 4.14: Apply Function node in the AOG with example tuples

The operator is enhance by secondary nodes that define the operations to be carried out on the tuples. The compiler will analyze the nodes and check if they are suitable for offloading. It will then generate a custom Verilog description for every Apply Function node accordingly. An important operation is the CombineSpans operation which will be discussed in detail.

Combine Spans

The Combine Spans operation is a secondary operation of the Apply Function operator which is carried out by itself. As the name suggests this operator combines two input spans and creates a new span by taking the smaller start offset and the greater end offset. In the general case the hardware module operates on every tuple of the stream in a single clock cycle using two comparators and four multiplexers. It does not require a special ordering of the tuples as every tuple is treated independently. The ready signal of the receiver is directly passed to the input to control the dataflow. The core architecture is outlined in Figure 4.15.

4.2.7 Project

A Project operation selects only a subset of attributes from the input schema and creates a new set. It may also retain every input attribute but reorder them. In software the attributes can be accessed by name but in hardware the sequence is important. It is realized by correctly wiring the Verilog top-level of the query graph without any additional modules and instances. The mapping between the input and output schema is defined as an attribute on the operator node in the annotation operator graph. It is described as a JSON list with pairs of attribute names, the first being the input attribute name and the second is the output name. Figure 4.16 shows the operator node with an example.

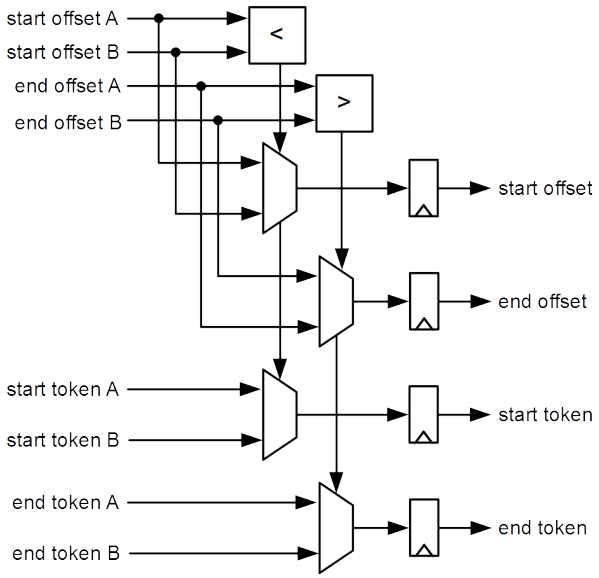


Figure 4.15: Architecture of the generic Combine Spans operation

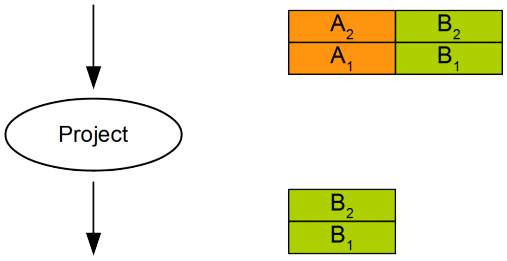


Figure 4.16: Project node in the AOG with example tuples

4.3 Compilation framework

The compilation process transforms a declarative AQL query into a synthesizable Verilog hardware description. It is responsible for creating the individual operator modules with their respective parameters, requirements and additional functional units. To perform these tasks a compilation framework has been designed that interacts with a number of different tools and component compilers and can be easily extended to support additional operators or tasks. Figure 4.17 shows an overview of the compilation framework and its components. The overall core framework is written in Python but employs other tools written in C++ and Java.

While the core framework handles graph level operations and transformations, the individual operators are defined as separate Python modules that can be loaded on demand. Every module defines the requirements for its operator, how the Verilog module is generated and how the module is instantiated and routed in the top-level graph module.

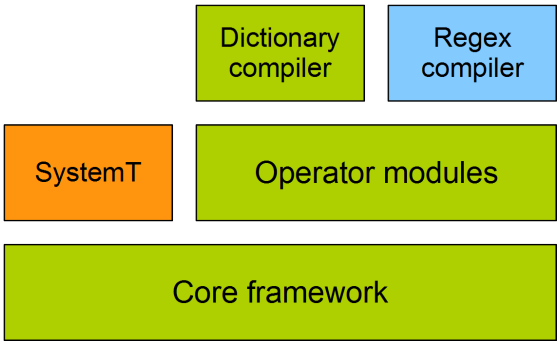


Figure 4.17: Overview of the compiler framework

Each module implements one operator class which extends either one of the base classes *Operator* or *Secondary*. The operator classes must implement at least the three methods *compile()*, *addToNetlist()* and *routeInst()* as these will be called by the core framework for each operator node in an annotation operator graph. The compile() method generates the HDL for a specific operator instance. As this process is very different for every operator the base classes only supply an empty method creating no HDL at all which can also be the case if no specific hardware module needs to be generated for the operator e.g. *Project*. Other operators such as RegularExpression or Dictionary will call external applications to generate the complete or parts of the hardware block. In this case the compile() method must setup the inputs to these applications appropriately and collect their output data.

The *addToNetlist()* and *routeInst()* methods define how a generated hardware module is instantiated and routed on the top-level graph module. Although a standard interface was defined some operators may require different or additional port definitions, as well as specific parameters that need to be set on the top-level instance. To have this flexibility these methods can be redefined by the operator classes. The methods provided by the base class can be used when the hardware module adheres to the standard interface definition and does not require any special processing. The module's name is stored as an attribute value on the object and needs to be set during initialization or operator compilation.

Furthermore the framework employs SystemT to compile an AQL query into an initial annotation operator graph (AOG) and acquire profiling information on the query.

4.3.1 Compilation

The compilation flow is multi-staged process to transform a text analytics query into a hardware description and eventually to an FPGA bitstream. This process requires multiple software tools and applications and is outlined in Figure 4.18 showing the steps involved and the data they produce.

The input to the compiler framework is a query written in AQL as well as a set of

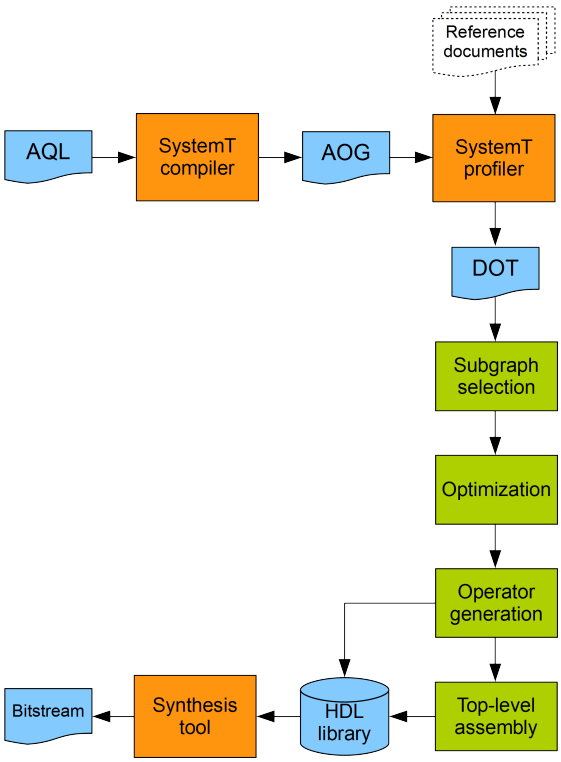


Figure 4.18: Processing steps of the compilation flow

reference documents the query should be run on. If the reference documents are omitted a set of standard documents is taken and the compiler will optimize the design more conservatively. The AQL query is transformed into an annotation operator graph (AOG) by calling the SystemT compiler, which generates a .aog file that can be used by the SystemT runtime to run the query on a document set. The AOG is run against the reference document set with an extended SystemT profiler that will capture statistics about the individual operator nodes such as average execution time and average number of tuples produced. This information is added to the operator nodes and dumped into a graph description using the GraphViz DOT language [51].

The root node is called *DocScan* and provides the document and token data. In the hardware description this results in a *SubgraphInput* module that synchronizes the two data streams of document characters and token definitions. This is the front-end circuitry of the dictionary matching architecture discussed in chapter 3 and is now shared across all extraction operators. As now multiple receivers may cause backpressure to this input node the flow control signals must be combined to invalidate the data if a single receiver asserts the ready signal low. This is done by *bumper* nodes and modules which the compiler will insert after any operator that has more than one receiver. Figure 4.19 depicts the logic of a bumper node.

After the initial *DocScan* node the actual operators follow and describe the data depen-

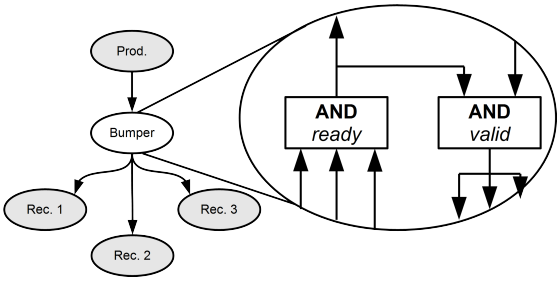


Figure 4.19: Inserted bumper node to synchronize multiple receivers

dency between the nodes. The last level of nodes are *Output* nodes. These nodes do not perform any operations but define the output set names and attributes. In a first step the compiler will select a single contiguous subgraph starting from the *DocScan* node. The subgraph contains all the operators that are supported by the framework, which in the ideal case are all nodes down to the *Output* nodes. If an operator is not supported by the framework, the compiler will add extra *Output* nodes that will produce the input tuples to the unsupported node which can then be executed in software. In many cases these are the last operators before the actual output nodes that transform a span into the actual string representation.

To finalize the subgraph selection step a SubgraphOutput node is added as the very last node in the graph.. This is a hardware module that will multiplex the individual outputs into a single 128-bit wide output stream using a round-robin arbitration scheme for a single tuple of each input edge. As the different output nodes may produce tuples with schemas of different width, a wider tuple takes multiple consecutive cycles when written to the output port. Furthermore the module will assign each output a 32-bit result id which prefixes the actual result schema. The integrating logic must decide whether the results are written to a single buffer in host memory or into multiple different ones.

The next step in the compilation process is to derive the ordering requirements of the operator inputs. This is done by visiting every node and query the ordering requirements for the input and output edges. This information is then collected for every edge in the graph, which are then checked for consistency. If an edge has contradicting ordering information the compiler will try to adjust the nodes by parameters appropriately. If that is not possible or multiple receivers have different requirements the compiler cannot change, it adds a sorting node to create a correctly sorted data stream. Once all requirements are satisfied, the compiler applies a set of optimizations which are discussed in section 4.3.2.

After the optimization phase the individual hardware modules are generated by calling the *compile()* method on all remaining primary nodes in the graph. The modules are then assembled to the top-level graph netlist according the graph description with the addToNetlist() and routeInstance() methods. The routeInstance() method will look at the input edges and interpret the index attributes annotated on these edges. If an operator produces multiple output views an index determines the view that is going to

the next operator. If an operator has multiple inputs the same applies and an index defines which input is meant by this edge. Secondary operators often only operate on one attribute of the entire input schema. Therefore an additional column index defines which attribute to pick. If multiple attributes are required, multiple edges are defined in the graph and routed accordingly. Every operator that additionally requires the document or token data gets these signals routed to them from the *SubgraphInput*. These signals are not allowed to create backpressure but honor the global valid signal after the input bumper node.

This concludes the HDL generation process and usually takes only a few seconds. If large dictionaries or complex regular expressions are used the compilation time may become multiple minutes long. Once all HDL files are in place the FPGA vendor tools are set up with the project files and the synthesis flow is executed. Depending on the complexity of the query the hardware design can become rather large, lengthening the synthesis time to multiple hours.

4.3.2 Optimizations

The annotation operator graph (AOG) generated by the SystemT compiler is optimized for sequential execution in software. On the hardware now all operators are running in parallel and require and allow a different set of optimizations to be applied. These optimizations primarily target the area requirements of the generated hardware design by inspecting dataflow constructs in the operator graph. Some constructs allow the compiler to fuse multiple operators and instantiate a special hardware module that covers all of the fused operators. Other optimizations remove operations that are required to satisfy the general case but can be omitted in the context they are instantiated. The following sections discuss these optimizations applied by the compiler.

Adjacent Join followed by CombineSpans

A common case is that a *CombineSpans* operation follows an *Adjacent Join* to create a single span containing both joined spans. A generic *CombineSpans* instantiates two comparators and four multiplexers to select the appropriate start and end offset fields. But for the case where the input to the CombineSpans operation is generated by an Adjacent Join the positions of the two spans are known, thus eliminating the need for a comparison. This removes the entire logic from the CombineSpans operation creating only wiring as shown in figure 4.20.

The removal of the comparison logic has further implications on the required data fields by the Adjacent Join module. Because the final span is constructed immediately one field of each input span is never required to generate the output. This reduces the width of all FIFO buffers in the Adjacent Join module. Depending whether the distance check is made with token ids or character offsets, the memory requirements of the module drop by 16 to 33 %. The input interface is kept the same to use the same top-level wiring methods but a number of bits will be dangling wires. These may trigger further

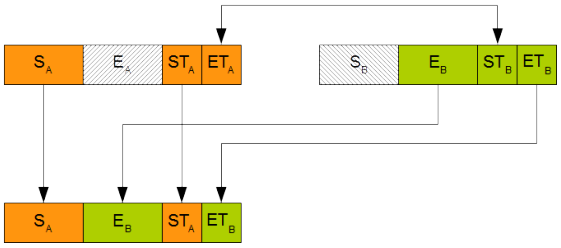


Figure 4.20: Required data fields for an Adjacent Join followed by a CombineSpans

optimizations applied by the synthesis tools as the logic to drive these ports can be removed.

To apply this optimization the compiler inspects the predicate of the *Adjacent Join* operation to determine which of the output spans follows the other one. It also inspects whether there are any Project operations in between the *Adjacent Join* and the *CombineSpans* operation that flip the position of the spans in the schema. Also the compiler ensures that only the generated output span by the *CombineSpans* operation propagates and the input spans are removed by following *Projects*. Once these aspects are determined the compiler sets a parameter on the top-level of the two module instances that direct the Verilog code to generate the appropriate hardware for this case.

Bi-directional Adjacent Join

An Adjacent Join checks the distance between to spans and joins them if they are within a maximum distance. This check only works to one side of the first input span i.e. to its right. Often queries implement this Join operation to both sides of the first input span checking the distance towards both directions and perform a Union on the results of these operations. Figure 4.21 shows such a situation in an annotation operator graph.

In this case the inputs A and B are joined twice with two different predicates. Another equal situation is to swap the inputs to the Adjacent Join and maintain the same predicate function. The compiler will detect both cases and replace it with a single bi-directional adjacent Join operator if the inputs arrive from dictionary matching units. Because in this case the maximum length of a span in terms of tokens can be predetermined, the read logic of an AdjacentJoin can be adjusted such that it only drops a span once the distance to the left can never be reduced again.

This optimization requires a more complex Adjacent Join module but allows the module to merge the buffering stage which is costly in terms of memory blocks. Also the Union stage can be removed which further reduces the amount of required FPGA resources.

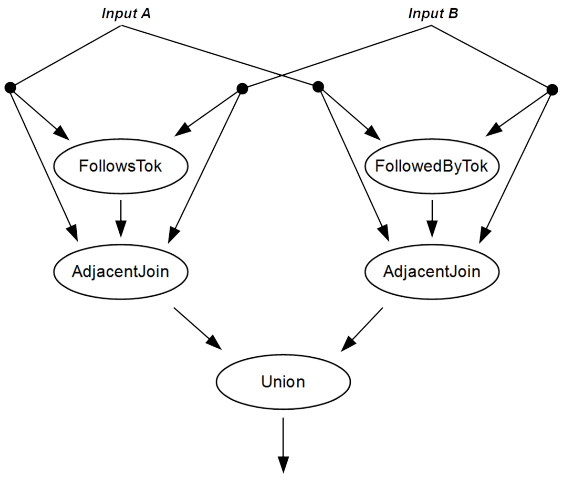


Figure 4.21: Original situation in an AOG that can be replaced by a bi-directional Join

Merge of equivalent subgraphs

After detagging a document with the Detag operator the query can select on which parts of the document it wants to operate. Often the same sub-query or subgraph are run on different parts of the document. Within the annotation operator graph these subgraphs are then replicated and their results are Union'ed. While for the software execution the operations must be replicated in hardware these subgraphs can be merged. By inserting a DocUnion operator after the Detag node the two document streams are multiplexed into a single document stream and fed to only one remaining instance of the subgraph. The DocUnion operator will create a valid stream of characters when one of the input streams from the Detag operator is valid. As the document is processed character by character only one stream can be valid at a time. Figure 4.22 illustrates the optimization and that the final Union operation can be removed.

Union of dictionary matches

In some cases keywords with similar meaning are stored in different dictionaries. In an AQL query the dictionaries get extracted separately and are then unioned. The AQL compiler does not combine these dictionaries but keeps the initial structure. Although this might be negligible for software the hardware implementation can benefit from merging the dictionaries into one large dictionary and remove the Union operator completely.

This can only be done if the individual dictionaries are not used elsewhere. The hardware compiler will check whether the output of a dictionary node does not go elsewhere and only then remove the Union operator. It will create a new dictionary and merge all required entries to compile the dictionary matcher appropriately. Figure 4.23 illustrates this optimization.

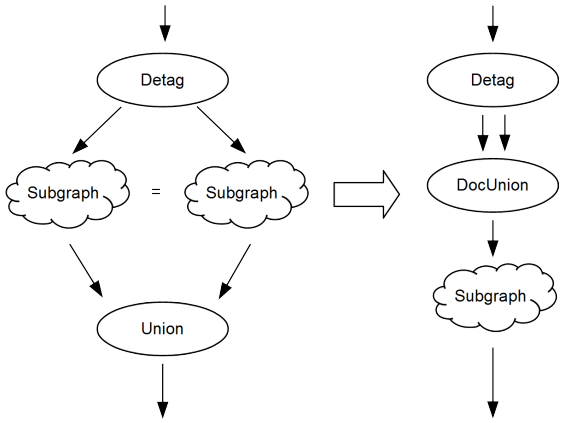


Figure 4.22: Equivalent subgraphs that are executed for different tagged parts can be merged

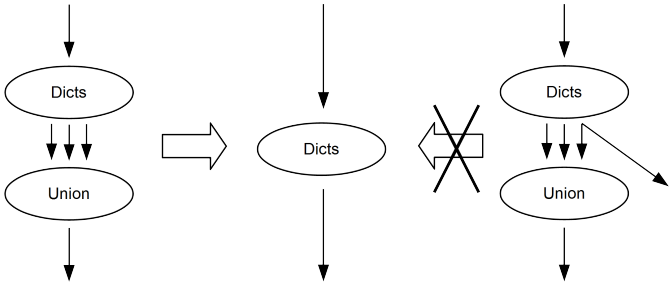


Figure 4.23: Union of dictionary matches can be removed

4.4 Evaluation

The main aspects for the evaluation are performance, scalability and energy efficiency. The primary driver for this work has been the limited performance of pure software-based text analytics software. The performance will be evaluated as the document throughput rate, which will be measured in number of characters per second. For ASCII encoding this is equivalent to bytes per second as a single character is always one byte. For UTF-8 encoding this is different as a single character can be made up of multiple bytes. The documents for this evaluation will only be ASCII encoded.

The scalability aspect reflects the ability of the hardware architecture to execute large complex queries. As every operator is instantiated as a separate module, large queries will occupy more resources on the FPGA and eventually more than there are available. The resource measurements will also evaluate the impact of the compiler optimizations.

With the increasing cost of energy [24, 64], energy efficiency is becoming a differentiating factor for a workload optimized system. Although the time to completion may always be shorter on the accelerated system, the overall energy consumption may be higher

than the one of a standard system. This part of the evaluation will measure the power consumption of different components within the system during idle time and runtime.

To evaluate the proposed framework, a set of representative queries with according reference documents has been chosen. The queries belong to different application domains such as log analytics (LA), business analytics (BA) and social media analytics (SMA). Queries and documents create different execution profiles to provide an overview of all evaluated aspects. Figure 4.24 shows the execution profiles of the various evaluation queries. It shows the relative execution time of the type of operator for a particular query run against the reference document set.

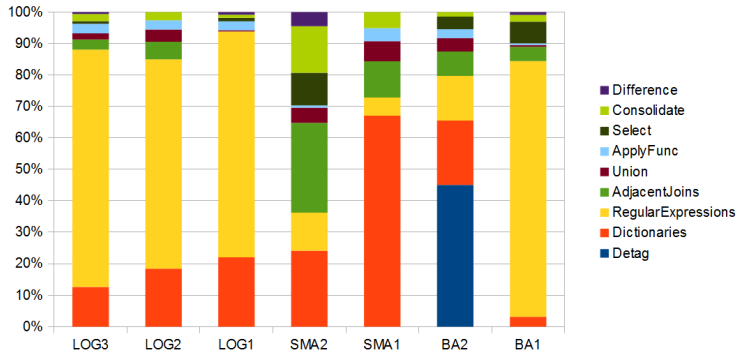


Figure 4.24: Execution profiles of the evaluation queries

While the log analytics queries (LOG) heavily rely on regular expression matching, the social media analytics queries (SMA) have a larger portion of relational algebra operations in them as well. The business analytics queries (BA) show very different behavior. BA1 relies heavily on regular expressions while BA2 spends a large amount of time on the detag operation preparing the document.

These numbers represent software profiling results. The time spent at relational operations appears to be low but the extraction operations (Regular Expressions, Dictionaries) have been accelerated by orders of magnitude in previous work [18] and the previous chapter 3. Thus the relational operations have to be accelerated to see an overall performance improvement.

The evaluation system is the POWER7-based enterprise server system introduced in 3.4. The POWER7 processor is capable of executing up to 64 threads in parallel using simultaneous multithreading (SMT) and runs at a frequency of 3.55 GHz. The FPGA accelerator card is directly attached to the processor host bus (GX bus) and is capable of 3.4 GB/s throughput. The compiled hardware query is embedded in a separate clock region to relax the timing requirements for the synthesis tools. All queries were compiled such that the hardware is capable of consuming four document streams in parallel.

4.4.1 Performance

To establish a performance reference the queries were run by the original software in a long-running setup to avoid any effects by the Java virtual machine. Each query was run against its two sets of documents for five times 30 seconds and the average was taken. One document set was the reference document set of each query while the second set was a collection of dummy documents data. The documents were all pre-loaded to main memory to avoid any limitation from the file system. Every query was also run with every possible number of software threads from one to 64 and the best value was selected. Figure 4.25 shows the performance results for all queries.

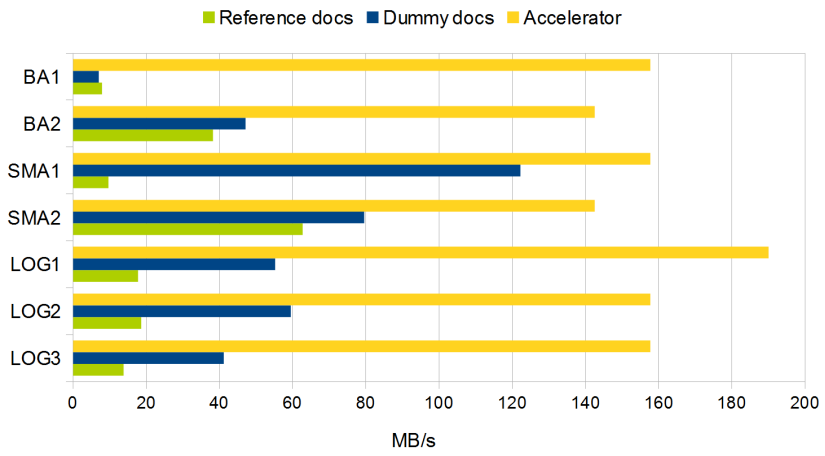


Figure 4.25: Maximum document throughput for multi-threaded software and a single hardware stream

The software performance is directly impacted by the type of document it scans. The performance of the dummy data documents is higher due to the limited number of pattern matches by the extraction operators. Hence there are no results for the relational operators to work on and the optimized execution plan allows the software to stop early. But also the query size or complexity has an impact on the software's throughput rate.

For the hardware measurements the documents were sent to the FPGA several times to also measure a 30 seconds window and count the number of characters processed. All documents were stored again in main memory and were then sent to the FPGA. The submission was done by a single software thread which was also running the translation server mechanism for virtual to real address translations. The compiled query was running at the maximum frequency with no negative slack. The hardware throughput results are summarized in table 4.1.

The performance of the hardware queries is primarily limited by the complexity of the query. The larger the queries are the more difficult is timing closure and thus the maximum achievable frequency which determines the maximum possible throughput rate.

System setup	LOG1	LOG2	LOG3	BA1	BA2	SMA1	SMA2
SW 1 thread	1.3	1.2	0.6	0.35	2.5	2.6	2.9
SW 64 threads	17.7	18.5	13.7	3.6	38.2	9.6	62.7
FPGA 4 streams	760.0	630.8	630.8	630.8	570.0	630.8	570.0
Improvement	42.8x	33.9x	45.7x	79.8x	14.9x	65.4x	9.1x

Table 4.1: Reference document throughput of the system in MB/s

Also the documents show an impact on the performance. But it is the size of the individual documents changing the throughput rather than the content of the documents. This is due to the higher transfer setup costs associated with smaller documents as discussed in section 3.4. To minimize the transfer setup costs the same batching mechanism was applied as in 3.4. In comparison the performance of the hardware queries is up to 79 times higher than of the pure software version running multiple threads on the host system.

4.4.2 Scalability

This section evaluates the resource consumption of the compiled hardware queries on the FPGA. This is important to understand the scalability of the approach and how complex the text analytics queries can be and still fit on the FPGA. The queries have all been compiled for four document streams with and without the optimizations discussed in 4.3.2. Figure 4.26 shows the percentage of the total available resources on the FPGA required by the compiled query for different types of resources.

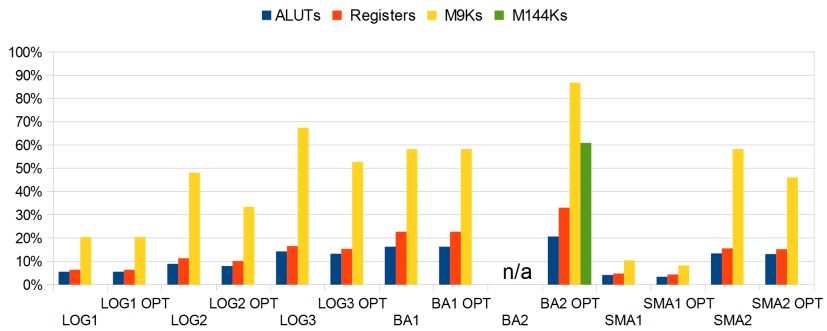


Figure 4.26: Query resource utilization on Stratix IV GX530

The numbers do not include the static amount of resources required for the communication logic with the system FPGA. This logic stays the same for all queries and uses 10 % of the Stratix IV GX530 FPGA's logic resources.

The dominating resource are the memory blocks (M9K and M144K). While the logic

resources range between 7 and 50 % the memory blocks sometimes require up to 90 % of all available resources. This is the case for query BA2 which is not able to be complied to the FPGA when not applying optimizations. This is due a detag operation followed by two large equivalent subgraphs. The remaining queries fit onto the FPGA without optimizations but benefit in a reduced area consumption.

4.4.3 Energy consumption

Another important aspect for compute systems within data-centers is the power or energy consumption. Predictions assume that the energy consumption of information and communication infrastructures will increase up to 30x by 2030 [48]. Although special purpose compute units offer better power efficiency for the task they were designed for they add a significant amount of static and idle power when they are not in-use. For this reason the energy evaluation is performed for the entire system and not the accelerator sub-system.

To measure the power consumption of a POWER-based host system, the EnergyScale power management system is used [71]. The system is equipped with various sensors and counters at critical locations such as the memory or I/O sub-system. These sensors can be read at maximum rate of 1 kHz during runtime via the system's service processor. For the measurements dynamic voltage and frequency scaling has been activated to balance performance and power consumption. The results are plotted in figure 4.27.

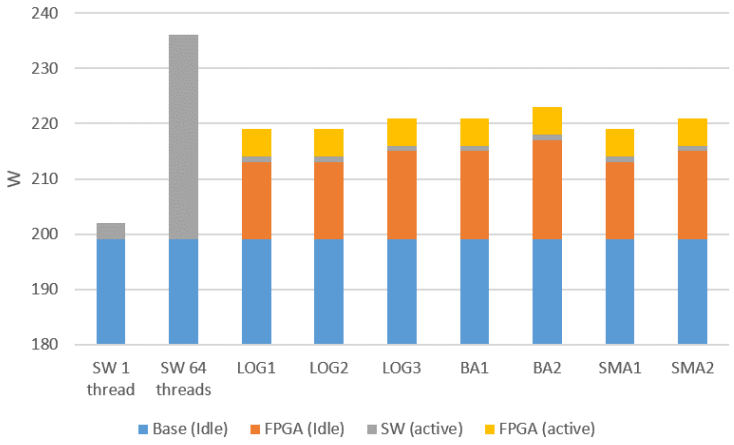


Figure 4.27: Overall system power consumption

The baseline system without the FPGA accelerator card attached consumes 199 W when idle. This amount of power is always consumed for all use cases with or without an accelerator. When a single thread is launched to execute a text analytics query additional 3 W are consumed by the system. The consumption ramps up to 37 W when using 64 threads on the processor. The additional power consumption is distributed among the processor modules and the memory sub-system. The power consumption is

not affected by the type of query or the document's content.

When the FPGA accelerator card is plugged into the system additional 14 to 18 W are consumed when the system is idle. This is the idle power consumption of the FPGA card mainly driven by static power which is different for each design which is loaded onto the FPGA. Larger text analytics queries result in a larger design on the FPGA area and thus activate more resources such as embedded memory blocks. If these blocks are not used the synthesis tool generates FPGA configurations which will power them down.

When processing documents on the FPGA additional 5 W are consumed by the FPGA which has been measured on the I/O sensor. The remaining system adds one more Watt for the total power consumption when running the text analytics query on the FPGA. This results in a lower power consumption compared to running the query with 64 threads on the POWER7 processors. But to correctly compare the required energy by the various system configurations the performance has to be taken into account by applying equation 4.1.

$$\epsilon = \frac{throughput}{P_{total}} \tag{4.1}$$

This results in a relative number of how many Joules are required per Megabyte of document data. Table 4.2 summarizes these values and indicates an up to 85x improvement in energy efficiency under full utilization. This is possible due to the both an improved throughput and better power efficiency of the accelerator design.

System setup	LOG1	LOG2	LOG3	BA1	BA2	SMA1	SMA2
SW 1 thread	0.006	0.006	0.003	0.002	0.013	0.013	0.015
SW 64 threads	0.07	0.08	0.06	0.03	0.16	0.04	0.26
FPGA	3.47	2.88	2.85	2.85	2.55	2.88	2.58
Improvement	46.1x	36.5x	48.8x	85.2x	15.8x	70.4x	9.7x

Table 4.2: Energy efficiency for various queries in J/MB

4.4.4 Utilization

To measure the efficiency of the hardware design its utilization was evaluated. Utilization can be defined using equation 4.2 as the amount of clock cycles the circuitry performs active operations over the total number of cycles required to complete a task.

$$\epsilon = \frac{t_{active}}{t_{total}} \tag{4.2}$$

The definition of an *active cycle* depends on the hardware architecture in question.

While for a microprocessor the number of NOP[1] instructions maybe considered inactive cycles, for a streaming architecture every cycle where there is valid data and no back-pressure may be considered an active cycle. We will use the latter to define the number of operations that are carried out during a cycle in a given compiled annotation operator graph. Equation 5.1 defines the activity of the architecture for a given time step t, as the sum of all input edges which have their *valid* and *ready* signals asserted high during that cycle. Input edges are considered because, for each input tuple the operator module will calculate a result, regardless whether it is sent to the output or not. An exception are the Project nodes as they are always implemented as wiring only their operation will not count towards the total activity value. To honor operators that require multiple cycles to compute their results e.g. *Select* an additional e_{ops} term is added. This term is updated using equation 4.4 in every cycle. If a tuple enters the operator during that cycle, the number of required cycles is added to it. If the term is not zero before adding new operations, a one gets subtracted to account for the operation. This distributes the number of operations properly over time.

$$ops(t) = \sum_{edges_{in}} e_{valid}(t) * e_{ready}(t) + e_{ops}(t) \tag{4.3}$$

$$e_{ops}(t) = e_{ops}(t-1) - 1 + (e_{valid}(t) * e_{ready}(t) * f_{ops}(t, tuple)) \tag{4.4}$$

These equations where implemented as functions to the Verilog top-level of a compiled annotation operator graph. During simulation the functions will print the current cycle and the number of operations active during that cycle. This data can then be plotted to visualize the activity and a utilization factor can be calculated by computing the sum of all operations and divide it by the number of cycles measured.

Figure 4.28 plots the activity for query LOG3 while processing four complete documents. A blue triangle indicates a non-zero activity from the extraction units while an orange dot is the activity from the relational operators in the compiled datapath. The high peaks indicate the end of a document as the end flag is propagating through all operators. While the first document produces a fairly large number of matches at the extraction units the number of matches during a single cycle rarely exceeds one. For the remaining documents the extraction operators produce less results. The relational operators are even less active for all documents but there are some transactions happening in all cases, sometimes at the very end of a document.

On average only in 5.5 % of the cycles an activity has been recorded at all across the measurements. When averaging the activity across all operators in the query graph the average utilization is less than 1 % of all cycles.

To verify these measurements on a profiling level the SystemT profiler has been extended to report the average number of tuples produced by each operator in the annotation operator graph for a given set of documents. When dividing this number by the average document size of the set, a utilization number for each operator can be derived as the

[1] No operation

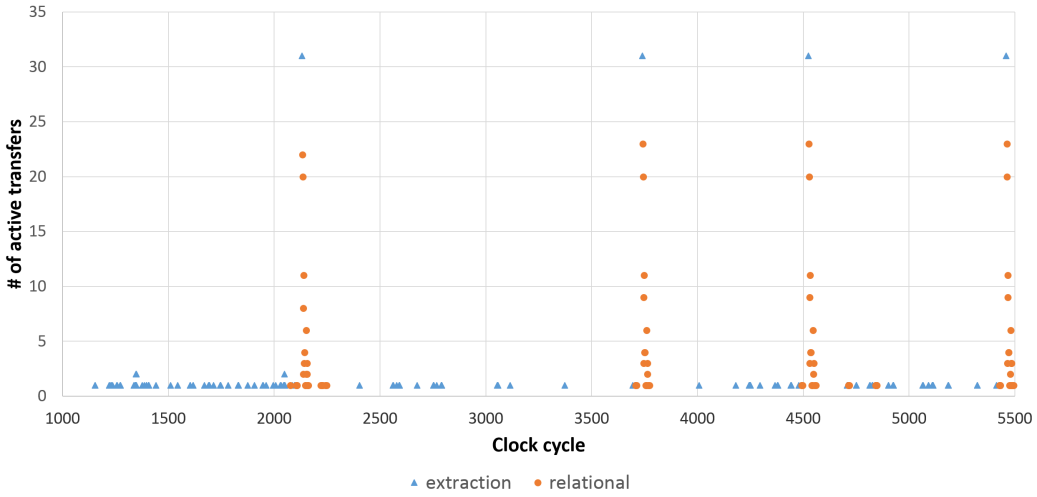


Figure 4.28: Total no. of active transfers in the datapath

document size roughly determines the execution time of the hardware pipeline. The obtained results match the hardware measurements and reveal an average maximum utilization of roughly 2 %. Figure 4.29 shows the utilization of the top 36 operators in query LOG3. Only three of 54 operators produce a tuple in more than 1 % of the execution time.

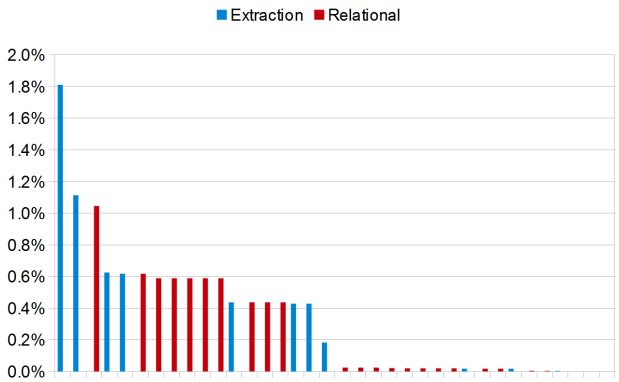


Figure 4.29: Utilization of the operators in query LOG3 from highest to lowest

Although the compiled query datapaths achieve high throughput rates and good scalability the utilization evaluation shows great potential to reduce the amount of required resources without sacrificing performance. Area resources may be time-shared among operators or document streams to achieve a higher utilization.

4.5 Related work

Using FPGAs to execute relational database queries has been studied by a few groups and was integrated into a commercial database appliance as well. A quantitative comparison is difficult as not only the applications vary from network stream applications to database application, but also the query language to be accelerated is ranging from SQL over Datalog to commercial languages. Also the presented approach combines pattern matching on unstructured data with the immediate execution of relational algebra operations on a hardware platform. On a qualitative level this section provides an overview of related work in the field of hardware-based query processing. Projects range from FPGAs and GPGPUs to a custom in-core architecture.

With the Glacier project Mueller et al. [76] present a query compiler for the StreamSQL language targeting streams of financial transactions. The transactions arrive as UDP packets via an Ethernet link that is directly attached to the FPGA. The queries allow a user to filter and aggregate incoming tuples to evaluate the number of transactions during a specified time window or construct the average value of the past five transactions. While the pure software implementation drops up to 60 % of the incoming packets the FPGA accelerator allows full processing at 100 million tuples per second.

Although the query language is capable of performing string comparisons the length of the string is predefined in the schema of the incoming tuples. Other than such string compares there are no text operations available. Also the use of *Join* was disallowed and was introduced as a separate component to the Glacier project [77]. Mueller evaluates various architectures to perform a windowed *Join* which is required when implementing a *Join* in a streaming context.

Work leveraging the use of dynamic partial reconfiguration for relational database queries has been done by Dennl et al. [42]. Similarly a library of components is used which implement basic operations such as Boolean and arithmetic functions. These target the restriction operation which is the condition statement in a Select operation. These blocks are pre-compiled to partial bitstreams which can be loaded to the re-configurable area of the FPGA. A datapath is assembled at runtime from these operators to represent the query statement. Thus no re-synthesis of the query is necessary and the query can run immediately once the datapath is configured. The performance is up to 6 times higher than of a pure software implementation but the library contains no complex string operations other than fixed length string comparisons.

The LINQits project by Chung et al. [34] aims to accelerate the language integrated query language (LINQ) which was developed by Microsoft and is part of the .NET framework. The LINQ language is similar to SQL and allows to easily query data structures such as arrays or enumerable classes. LINQits is a set of parameterizable hardware templates that implement functions available within LINQ such as Joins or GroupBy. During an ahead-of-time compilation step a user query is mapped to these hardware templates and a datapath is generated. This hardware datapath is then attached to an ARM core and the runtime system may decide to offload a query based on problem size or other input values. The speedup results range from 10x to 38x over an optimized

multi-threaded C code running on two ARM cores.

Also GPGPUs have been exploited for relational algebra query processing. Wu et al. [108] propose a set of kernel fusion optimization that can be applied to relational operators implemented on GPGPUs. They have extended a compilation and runtime framework based on the Datalog query language. Their optimizations aim to reduce the memory transfers required between the execution of multiple operators. By doing so a speedup by a factor of 2.89 for computation and 2.35 for PCIe transfers can be achieved over the original GPU-based implementation.

Lisa Wu et al. [109] present a in-core accelerator for the range partitioning step of relational operations. A large table is partitioned into multiple smaller tables where a specific key is always within a set of ranges. If range partitioning is performed prior to the actual relational operations the tables can be kept in a cache memory and a standard core can process the data faster. The in-core accelerator performs the task of range partitioning up to 7.8 times faster than a CPU and requires 6.9 % of the core's area. The data transfers to the accelerator are software controlled while the actual partitioning is performed in hardware.

4.6 Summary

To perform complete text analytics queries on an FPGA relational algebra operations have to be performed on the results produced by a pattern matching step. Due to the sequential nature of pattern matching the relational operators can benefit from a natural ordering of the input results. This can be exploited to implement the relational operators in a streaming fashion to achieve high performance at low area cost.

In this chapter a novel hardware compilation framework has been presented consisting of a hardware module library and a compiler. It builds on top of IBM's SystemT AQL compiler and allows to generate a full hardware description of a text analytics query defined in AQL. As a first of a kind it combines pattern detection units and relational algebra operations into a complex streaming datapath. Furthermore it supports string operations within filtering operators by buffering document and token data.

A set of optimizations applied by the compiler are specific to the hardware implementation of a query. Leveraging the spatial compute paradigm data types can be shortened and operations can be omitted as the location of the data already defines the result. By applying these optimizations the scalability for complex queries with more than 200 operators can be ensured.

The performance of the approach is up to 79 times higher than the original software implementation running on a large enterprise server with 64 threads. The accelerated system is capable of processing documents at 796 MB/s. The limiting factor of the throughput is the maximum achievable frequency and the number of parallel datapaths which are determined by the complexity of the query. The larger a query becomes the more area it consumes on the FPGA which reduces the frequency due to complex placement and routing constraints.

Also the energy consumption has been evaluated. Although the accelerator card adds to the system's overall power consumption, during active operation exercising the accelerator consumes less power than running the original software on all cores. This boosts the energy efficiency for text analytics queries by a factor of 85.

CHAPTER 5

Soft-core processor array

This chapter discusses the design and use of a soft-core processor array, called Turtle, to replace the custom compiled annotation operator graph (AOG) hardware presented in chapter 4. The motivation of using a processor array is, to be able to compile and execute different text analytics queries, without requiring re-synthesis and reconfiguration of the FPGA. This allows faster compilation times, enabling the use of dynamic queries and decouples the users of the system from FPGA vendor tools. Furthermore a context switch, meaning the change of a query on the hardware, can be performed faster on a soft-core instead of partially reconfiguring the FPGA.

5.1 Design objectives

The fact that the utilization of the individual operator modules is low, as discussed in section 4.4.4, indicates that the area resources can be shared in a time-sliced fashion among multiple operators or streams at the same performance.

There are multiple ways to implement time sharing. One option is to keep generic instances of the previously presented hardware operator modules and use them as functional units. A small control unit can orchestrate the memory transfers to and from the units, thereby implementing a given AOG. Depending on the memory and bus system this could strictly linearize the execution of the operators and perform very similar to software. In order to allow such functional units to operate in parallel, a high memory bandwidth would be required which is capable of accessing multiple regions in parallel.

Another option is to implement the operators as a software code on a programmable soft-core processor and create an array of cores to implement complete operator graphs. This allows every core to run any operator at any time, decoupling a specific operator from a specific functional unit in the system. The memory performance requirements

can then be relaxed by communicating results directly between individual cores. The load on the processor cores can be balanced by combining only an appropriate number of operators on a single core. To keep the area requirements for a single core low, complex operations should be externalized and shared among multiple cores. Such operations include the token ID to character pointer lookup and pattern matching within a specific region of the text document.

While the arithmetic and logic operations on the data itself should be kept at a word level, data movement operations such as loading from memory or core-to-core communication can be performed at the datatype level of a span, which is a quad-word. This should allow the data to be communicated efficiently while giving the core sufficient flexibility to implement any logic required by the operators. The communication between the cores should be based on streams as the previous chapter has shown the positive properties of streams in the text analytics context.

To summarize the requirements for the soft-core processor and array:

- Simple instruction set architecture capable of efficiently implementing relational algebra operators for text analytics
- Access to shared resources such as token definition data or text-based operations
- Efficient communication network with low overhead data identification

5.2 Microarchitecture

The soft-core microarchitecture as described here is tailored towards the Altera Stratix V [58] family of FPGAs. The main constraint arising from this target FPGA architecture are the distinct available sizes of the embedded memory blocks referred to as M20Ks. For Stratix V these blocks can be configured as follows (depth x width): 512 x 40b, 1024 x 20b, 2048 x 10b down to 16k x 1b. Adjusting the widths of instructions and data is crucial to achieve a high utilization of a single memory block, without leaving memory regions idle.

The top-level of the soft-core architecture is shown in figure 5.1. On the input side, the interfaces to the core consist of a load bus for configuring the core and data inputs from four directions, each being quad-word wide plus an additional index field for each bus to identify the virtual stream. The width of the index field is tailored to the shared-memory FIFO design discussed in section 5.2.2. The data output port is quad-word wide as well, includes a single index field and can signal valid data to four different consumers. Additionally each core has an external interface which will be discussed in section 5.2.5 to communicate with shared resources.

On the top-level, the core consists of nine main components. The program memory (PGM) holds and dispatches the instructions that are executed by the core. It can be written via the load bus interface and is 1k deep and 20 bits wide. The PGM contains the program counter (PC) which is automatically incrementing by one every clock cycle,

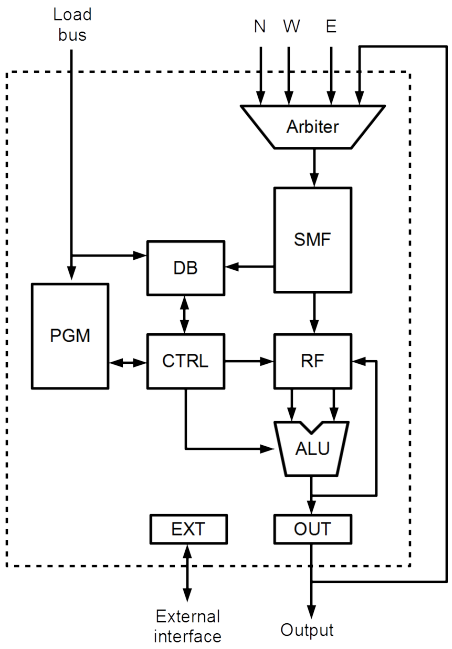


Figure 5.1: Turtle's top-level architecture

unless a stall is issued from the control logic. The PC can be saved and restored to and from a single stack register to implement *CALL* and *RET* instructions.

The data inputs are processed by a priority arbiter before being stored to the shared-memory FIFO (SMF). The connections to the arbiter will be discussed in section 5.3. The SMF is a key component of the architecture and allows to store multiple virtual FIFOs in a single large memory resource. The virtual FIFOs are accessed via the stream ID which is provided by the data inputs or an instruction accessing a FIFO. The SMF will be discussed in more detail in section 5.2.2.

If a virtual input stream is written to an empty FIFO the SMF activates the doorbell (DB) unit with the appropriate stream index. The doorbell consists of a short queue for the incoming stream indices and a look-up memory which holds the appropriate instruction address to start processing a particular stream. Once the control logic (CTRL) is ready to accept a new stream it accepts the start address, updates the program counter accordingly and starts processing the instruction flow. By accepting a new stream, the control logic pops the stream index from the doorbell's queue.

In order to perform operations with the arithmetic-logic unit (ALU) data needs to be loaded to the register file (RF) 5.2.3. The ALU can perform five basic operations: addition, subtraction, logical AND, OR and NOT in a single cycle using two words from the RF as inputs or one word from the RF and one as an immediate value stored in the instruction stream.

Once a result has been computed it can be pushed to the output registers forming a

complete quad-word. The instruction will then also assign a stream index to the output quad-word and a direction where it should be sent to. The output registers will signal the availability of new valid data by asserting a valid signal high until the appropriate receiver acknowledges the transfer with ready high signal. In the meantime the core can continue to operate on a new data tuple until it wants to push new data to the output registers. If a transfer is still ongoing and the core wants to initiate a new transfer the core will stall until the first transfer is completed.

5.2.1 Instruction Set Architecture

For the design of the instruction set an instruction word length of 16 bit has been chosen. This allows to store up to 1k instructions in a single M20K element and can still be extended to up to 20 bits for future use. The opcode width has been set to four bits allowing up to 16 instructions to be implemented. This may seem fairly small but is sufficient to implement the relational operations discussed in chapter 4. Table 5.1 gives an overview of the operators and their required ALU operations including external commands.

Operator	ADD	SUB	AND	OR	NOT	EXT
Joins	no	yes	yes	no	no	no
Difference	no	yes	yes	no	no	no
Consolidates	no	yes	yes	no	no	no
Unions	no	yes	yes	no	no	no
Apply Function	yes	yes	yes	yes	yes	yes
Select	yes	yes	yes	yes	yes	yes

Table 5.1: Required ALU operations by relational operator

Subtraction is often used as it implements the compare functionality. Most operators rely on conditional checks on the result of ALU operations whether a zero or an overflow has been signaled. Besides the ALU and jump operations, basic memory operations are required to load data from the SMF and push data to the output interface. The instructions are grouped into five types depending on the type of operation they carry out. The types are as follows:

- A - ALU operations
- J - Jump operations with or without condition
- M - Memory operations
- E - External operation
- D - Data word

ALU operations (format A) consist of the four bit wide opcode followed by a single type bit. The remaining fields are a three bit wide target register and two 4-bit source

registers. The type bit indicates whether the second operand is an immediate value stored in the instruction stream. If the type bit is set high then the two following 16-bit instructions form a 32-bit immediate value. The control logic will assemble the quad-word and pass it to the ALU accordingly.

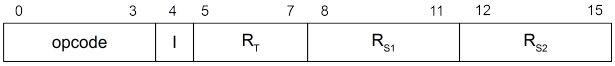


Figure 5.2: Instruction format A

There are five ALU operations that can be carried out ADD, SUB, AND, OR and NOT. The first four can be used with immediate values with the type bit set to 1. For the assembler these instructions are named ADDI, SUBI, ANDI and ORI.

Jump operations (format J) are based on the same opcode (1111) and are differentiated by a two bit condition type field. The remaining 10 bits represent an absolute address as the target address. The condition type field allows to implement four types of jumps. If the field is set to 00 the jump is an unconditional one, JMP. To test whether the zero flag has been raised or not, the branch instructions BZS (branch if zero set, 01) and BZC (branch if zero clear, 10) can be used. The carry flag can be tested by using BCS (branch carry set), which has type field value 11.

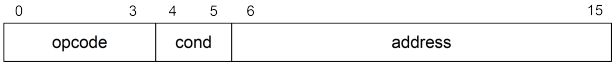


Figure 5.3: Instruction format J

Format F is also used by the CALL instruction which behaves equal to an unconditional jump but captures the current value of the program counter (PC) to perform a sub-routine. Once this routine ends the RET instruction will jump back to the stored PC value and continue. This allows to reuse program blocks for different operators and can reduce the total number of required instructions.

Memory operations (format M) target the shared-memory FIFO (see section 5.2.2) and the output interface. Although similar to classic load-store instructions these operations adhere to the stream-based concept. Following the four bit opcode a two bit wide register is indicated used either as a target or source register depending on the operation. Then a flag bit follows which indicates whether or not a load operation on an empty stream will result in a WAIT condition. The following field determines the sub-type of operation to be carried out and the remaining 7 bits represent the virtual stream index on which to operate.

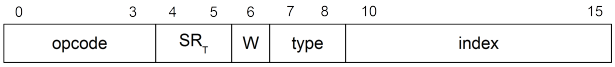


Figure 5.4: Instruction format M

External operations (formats E1 and E2) consist of two consecutive instruction words. The first instruction holds the opcode for an external command and two source registers. The second instruction carries the second pair of source registers as well as the three bit wide target register, the result will be written to. But the opcode field of the second instruction is used for the external processor to indicate the type of operation. This way the external functional unit gets four 32 bit values and an instruction opcode to compute a result which is written back to the target register.

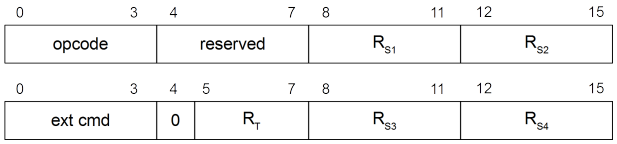


Figure 5.5: Instruction formats E1 and E2

5.2.2 Shared-memory FIFO

The shared-memory FIFO (SMF) is a key component in the Turtle architecture. It enables the efficient linear access to multiple virtual streams that can be stored in a single memory element. The architecture uses the concept of a singly-linked list where each data element stores a pointer to the next data element. A head and a tail pointer indicate the first and the last element of a list. By using multiple head and tail pointers multiple queues can be implemented. These pointers are indexed by the stream index which is required when writing to or reading from the SMF. An additional current pointer allows reads from the shared-memory FIFO without losing the read data. For some of the relational operations that is a key feature to avoid additional writes to a temporary queue, e.g. *AdjacentJoin*. Figure 5.6 shows the conceptual architecture and instructions for the SMF.

To keep track of free memory slots where data can be written to, a list of free pointers is kept in a bucket. After a reset the free bucket will output a linearly increasing address on request. Once all possible addresses have been issued the bucket operates as a FIFO. When a pointer becomes available again because of a POP or LDPOP operation, the pointer is written to the FIFO and can be issued to the next requester. If the FIFO is empty, then there are no slots available a request has to wait.

The main instructions that operate on the SMF are shown at the bottom of figure 5.6. The LD commands load the current element (light grey) into a specified register. The standard LD command advances the current pointer to the next element in the virtual queue while the LDS command leaves the current pointer untouched. The POP command loads the current element and drops the head of queue. At the same time the current pointer drops to the new head of queue. Similarly the LDPOP command drops the head of queue but leaves the current pointer untouched. To move the current pointer back to the head of queue the RST instruction can be used.

A high performance implementation of the SMF uses register files to implement the

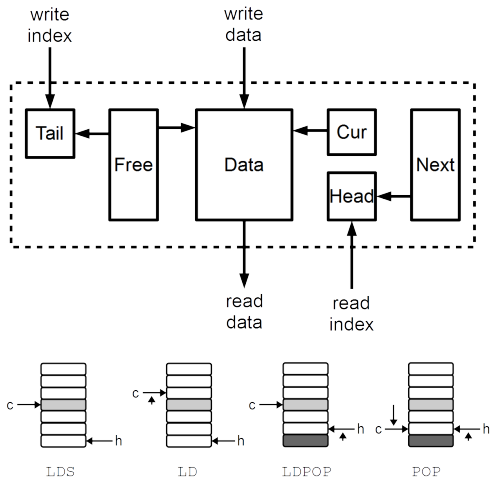


Figure 5.6: Architecture and instructions for the shared-memory FIFO

pointer tables (tail, head and current) while using embedded memory blocks for the data, next and free pointers. This implementation allows single cycles read and write operations that do not interfere with each other. But register files are expensive in terms of resource utilization on an FPGA. First because all storage bits are implemented as separate registers and second because multi-write and read ports require a lot of logic resources. An initial analysis showed that over 40 % of logic resources of a Turtle core were used by the shared-memory FIFO.

To reduce the required amount of logic resources the pointers (tail, head and current) as well as some status flags where consolidated onto a single M20K embedded RAM. By utilizing the byte-enable feature for the write operation the logic resource utilization was reduced by over 50 % at the resource penalty of one additional M20K. On the performance side the read operation now requires two cycles to complete while the write operation remains at a single cycle. This is acceptable as load instructions are rarely back to back or can be avoided.

5.2.3 Asymmetric register file

The Turtle core can perform ALU operations on data in the register file (RF). Although operations are performed on single 32-bit wide data words the basic data type used for the text analytics application is a span consisting of four data words. The core design has been chosen such that data movement operations (load/push) are always performed on spans while individual ALU operations are performed on single data words. This reduces the amount of required cycles to serialize and de-serialize a span.

As a result the register file has been designed asymmetric. It consists of 16 32-bit wide registers grouped into four groups of four registers each. These groups are also referred

to as *span registers* (SR). The first two span registers (SR0 and SR1) can only be written by load instructions with data from the SMF. Always a full span is loaded. The second and two span registers (SR2, SR3) can be written by the ALU and external commands. In this case every data word is written individually allowing the core to create and assemble new spans.

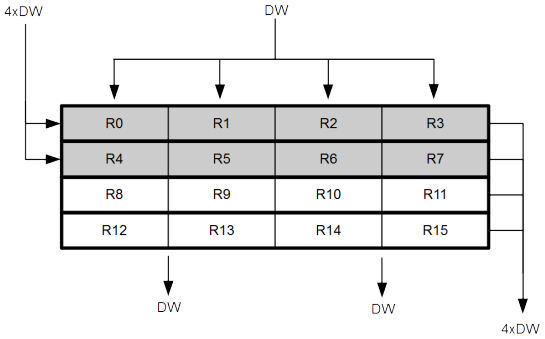


Figure 5.7: Register file with asymmetric read and write ports

On the read side there are two word-wide port to fetch two data elements for ALU processing. These ports can access every register in the register file. A third read port is used to implement the PUSH instruction. It can read all four span registers and reads an entire span at once.

5.2.4 Doorbell

The doorbell mechanism allows the core to fall into a *wait* mode when there is no data to be processed. Instead of polling for data from the SMF the core will stall and wait until it gets activated by a new incoming piece of data. Because every data is associated with a stream index, the index can be used to lookup the appropriate start address of the operator to be executed for it. This removes the burden of identifying the stream in software.

The doorbell logic is partially included in the shared-memory FIFO (SMF). When the SMF receives a write operation for an empty virtual stream it will raise a flag signal with the according stream index and present it to the doorbell module (DB). The stream index is stored into a FIFO deep enough to hold all virtual stream indices. Figure 5.8 shows the connections of the doorbell module.

On the output side of the FIFO the stream index is used as an address to a translation memory. This memory is configured with jump addresses that are generated during the assembly process. If the FIFO is not empty it signals it to the control logic of the core and presents the appropriate jump address for the first stream index. Once the core is in a wait state it will set the program counter to this new start address and continue operating. The jump is acknowledged and the stream index is removed from the FIFO.

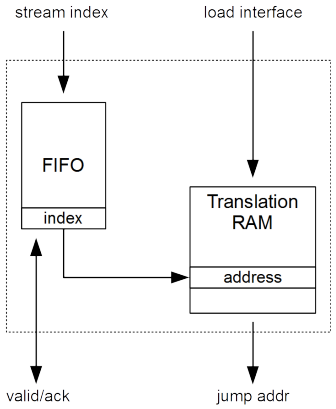


Figure 5.8: Doorbell architecture

5.2.5 External commands

For more complex operations such as regular expression matching or access to token definitions Turtle provides an interface to execute external commands. The interface consists of a control register and four data registers that present their data on the interface ports. When an external instruction is issued the control logic first sets the registers and then raises a valid flag to indicate the external resource to fetch the command. The valid pin remains high until the external resource acknowledges the completion of the operation and presents the result as a single dataword on the interface input. The result is written to a register in the RF which has been indicated by the external instruction.

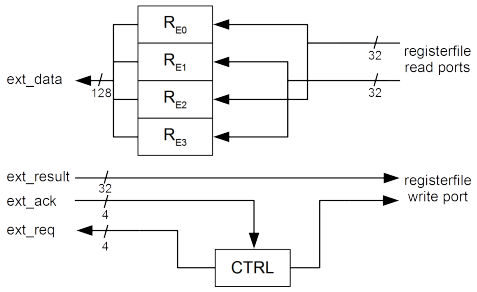


Figure 5.9: External commands architecture

The Turtle core stalls during the processing of an external command. Although this is inefficient in terms of utilization it simplifies the design as otherwise the context of the RF and the PC needs to be captured and restored. Assuming token lookups can be performed in a few cycles the main concern are regular expression checks as they have a longer processing time. In this case the partitioning of the operators across the core array (see section 5.3) is important.

5.3 Soft-core array

The Turtle core was designed to operate in a mesh of cores. Although a single core can execute multiple relational operators, it does not scale to the number and required performance for larger text analytics queries. Large queries often have a higher number of extraction operators, detecting patterns within the document. As a result, the number of input tuples to the mesh of cores increases and the mesh needs to be able to accept these tuples in order to avoid the pattern detection units (PDUs) from stalling. This is crucial, as the PDUs determine the document throughput performance.

On the other hand, large queries perform more relational operations to increase the quality of the extraction results. Although this means many more operations have to be carried out, many of these operations are restrictive, meaning there are less and less tuples propagating towards the end of an operator graph. As annotation operator graphs (AOGs) do not contain loops there is a clear propagation direction of tuples in an AOG.

These aspects can be taken into account when designing the overall system. As presented in section 5.2, the Turtle core has the ability to send and receive data streams to and from four directions. This allows to design a mesh of cores, where each core can communicate with its adjacent neighbors. But because of the directed nature of an AOG, the north connection has been rerouted to the sending core itself. As there are no loops, there is no need to go back. But by sending data to itself, a core can create its own temporary data or streams for another operator it runs.

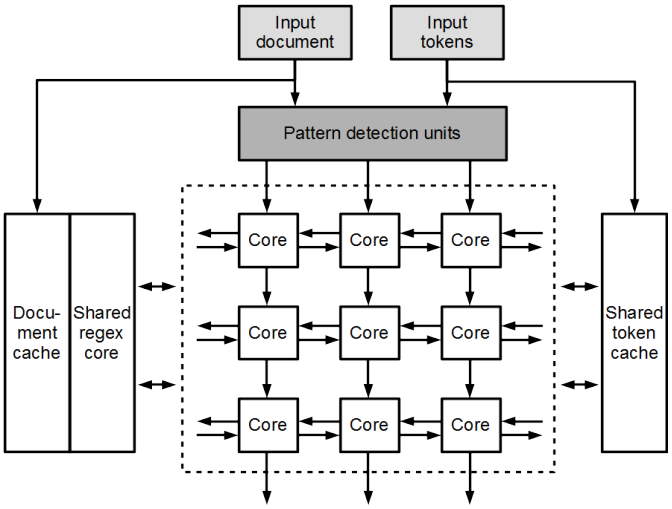


Figure 5.10: Turtle array architecture including shared units

Figure 5.10 depicts the overall system with the mesh of cores at the center. The west-most and east-most cores are interconnected as well, to have a consistent communication scheme for every core. The northern input to the top row of cores is fed by the pattern detection units, which operate on the input document and the associated token

definitions. The input data gets stored into rolling-buffer type caches discussed in sections 5.3.2 and 5.3.1. While the token cache can be accessed directly by a core the document cache is only used by the shared regular expression core. The last row of cores can create output data by sending data southwards with an associated stream index.

The inputs to every core arrive at a priority arbiter (see section 5.2. The highest priority is given to the input from the north as it is the primary source of data. Furthermore if a tuple is sent southwards its processing has been finished in a particular row. This is also true for the PDUs, which provide the initial tuples. These inputs should not stall and thus have the highest priority. Second priority is given to the self-loop because if a core sends data to itself it either requires it immediately or needs it to continue consuming other input streams. The last priorities are given to the horizontal connections in no particular order. The priority scheme is acceptable as it is unlikely for a core to produce a valid output at every clock cycle.

5.3.1 Shared token cache

The shared token cache (T-Cache) is used to buffer the token input stream for an associated document character stream. It provides a mechanism to access the token definitions required by the token-based *Context* functions used by some operators such as e.g. *Select*. The operator may want to build a new span by extending the original span three tokens to the left. While the calculation of the token index is a simple subtraction the character-based offset needs to be determined by a lookup.

As the pattern detection units create spans in order from the beginning of the document towards the end also the context lookups are likely to occur in such an order. Although some relational operations may have altered the spans this general concept still holds as seen in chapter 4. Also the range of the token-based context extension is in most cases less than 20 tokens. This allows the use of a rolling buffer where the character-based token definition is written in to FIFO-like memory. Once the FIFO is full the first element is dropped and the next definition is written in its place. To access the memory the LSBs of a given token index are used while the entire index is checked against two element counters to ensure the data is available in the cache. If the data is not available because it has not yet been received or it has been dropped the cache will fetch the required data from main memory. Figure 5.11 shows the overall architecture of the T-Cache.

The T-Cache can be accessed by a core using the two instructions $TOKS$ and $TOKE$. Each of the instructions requires a destination register and a single source register holding the token index. While $TOKS$ gets the start offset of a token $TOKE$ gets the end offset and stores it into the destination register. Both operations are blocking as all external commands (see section 5.2.5 until the result is available.

The T-Cache has the ability to store multiple token definition streams. This allows to program multiple operator graphs on the core array, which may operate on different documents.

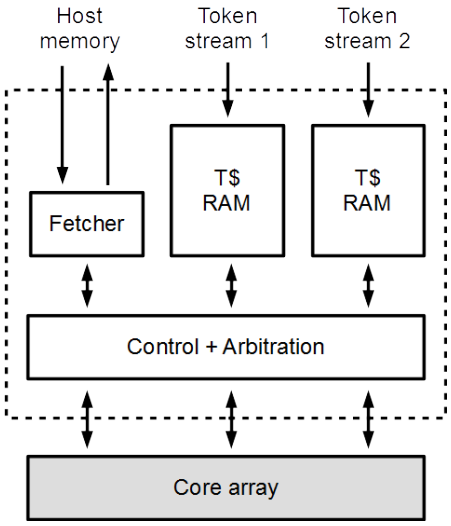


Figure 5.11: Shared token cache architecture

5.3.2 Shared regular expression unit

The condition check within a *Select* operator may require to check a regular expression against a span within the input document. This requires access to the actual document data and perform a character-based check, which cannot be executed very efficiently on the Turtle core. For this purpose a shared regular expression core (SREX) is added to the core array which can be accessed by multiple Turtle cores. The SREX has a document cache (D-Cache) attached to it that works similar to the shared token cache (5.3.1). But instead of reading a single character the D-Cache produces a character stream based on a given span either reading the document data from a buffer or main memory.

The SREX itself is implemented as a programmable state machine using the BaRT scheme (see chapter 3) called BFSM. It allows to program multiple regular expression patterns as a state machine which are checked at the same time when running a test. The SREX receives a span and a pattern id as inputs and checks whether the according pattern is contained within that section of the document. It can either check whether the pattern is contained within the span or if it matches on the boundaries of the span. The cores can submit requests to the SREX by issuing the instructions $CREX$ or $MREX$. Both instructions have four arguments: A destination register, a register containing the pattern id and two registers defining the character-based start and end offsets. While $CREX$ checks if the pattern is contained anywhere within the span $MREX$ requires the pattern to match on the boundaries of the span. As a result either a 0 or a 1 is written back to the destination register. While a zero means no match, a one indicates a positive result.

5.4 Programming

In order to program the presented architecture two simple assemblers have been developed. The main concept is to develop individual operators as Turtle assembly code (TASM) that may contain configurable parameters with default values. These operator templates are stored in an operator database for use by the array assembler. To instantiate an operator on a core in the processor array a second type of descriptive language (TSOA) is used that can overwrite the default values and associates the operator with one or multiple stream indices. This is required to create the contents of the doorbell translation RAM. At the end of the assembly process two binaries is produced for every core in the core array, one for the doorbell and one for the program code. Figure 5.12 illustrates the two main components as well as the operator library and the flow.

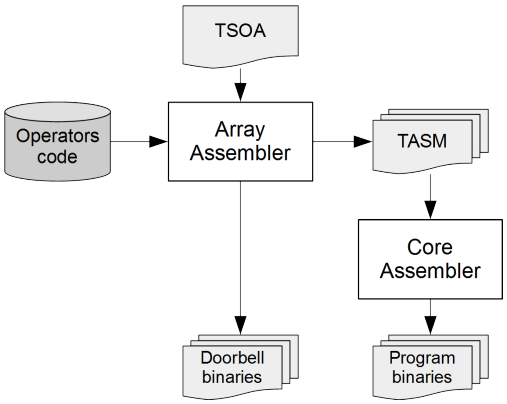


Figure 5.12: Assembler framework and flow

To create an operator a developer must write assembly code using the instruction set of the Turtle micro architecture. The code may contain labels that mark specific entry points of instruction blocks. These labels can then be used in any branch instructions such as e.g. JMP or BZC. Labels are defined by a descriptive name followed by a colon. To use the label in an instruction the colon prefixes the name of the label. The assembler resolves these labels and replaces them with the actual values. A reserved label is the $START$ label which will be used for creating the entry point in the doorbell to this operator.

Numerical values can be passed to the instructions as either decimal or hexadecimal values where the latter are prefixed with 0x. The $PUSH$ instruction may use the letters M, E, W and S to indicate the direction of the output where M is the self-loop. To use configurable values or directions variables can be used that are indicated with a dollar sign ($) followed by a number starting with 1. The variables can then be set by using the arguments passed to an operator instance in the array-level assembly code. The default value to a variable is assigned by the right-hand side of the equal sign separating variable name and default value. Below is an example of the assembly code for the consolidate on exact match operation.

```
START:
  LDW SR0 $1=0
  SUBI R8 R0 0xFFFFFFFF
  BZS :END
  LD SR1 $1=0
  SUB R8 R0 R4
  SUB R9 R1 R5
  OR R8 R8 R9
  BZS :OUT
END:
  PUSH SR0 $2=0 $3=S

OUT:
  POP $1=0
  JMP :START
```

Listing 5.1: Configurable assembly code for a consolidate on exact match operation

To program the core array an operator-level configuration can be used. It allows to instantiate multiple different operators on every core of the core array. First a core has to be defined by specifying the X:Y coordinates starting with 0:0 as the top-left core. Then a stream label needs to be defined which tells the assembler on which streams this operators will work on. This simplifies the process of creating the doorbell translation RAM contents. The label can be a multi-label if the same operator works on multiple stream ids which will create the same entry address for the defined streams. After the label the operator name is given followed by arguments setting the parameters in the code template.

```
0:0
S0,S1: AJCS 0 1 5 0 S
S2,S3: AJCS 2 3 1 1 S

1:0
S0,S1: AJCS 0 1 5 0 S
S2,S3: AJCS 2 3 5 1 S

0:1
S0,S1: DIFF 0 1 0 S
S2,S3: UNION2 2 3 4 M
S4: CONSCW 4 1 S

1:1
S0: COPY 0 2 W
S1: COPY 1 3 W
```

Listing 5.2: Core array assembly example

5.5 Evaluation

The evaluation of the soft-core array focuses on two aspects: scalability and performance. The scalability of the approach is determined by the resource consumption of the core array and its configuration. Although a single core can operate on multiple virtual streams now this adds several additional resources to it. Also the performance behavior is different from the previous 4 approach as the operators are implemented as small software sub-routines and take several clock cycles to finish producing a single result tuple.

5.5.1 Scalability

The resource consumption of the soft-core array is determined by the required area of a single core. As the core contains all the necessary arbitration and routing logic for four inputs the area requirements for an array of cores can be obtained by multiplying the area of a single core with the number of cores in the system. Synthesis results confirm this behavior for an array of up to 5x5 cores.

To evaluate the area overhead of a single core over the hardware library implementations the resources need to be normalized. While a single core can operate on multiple streams the library modules work on one, two or in case of a *Union* on multiple streams. A core with 16 virtual streams has been synthesized and multiple instances of an operator module have been synthesized to operate on 16 streams as well for a target frequency of 200 MHz. For example 8 *AdjacentJoin* modules or 16 *Consolidate* operators.

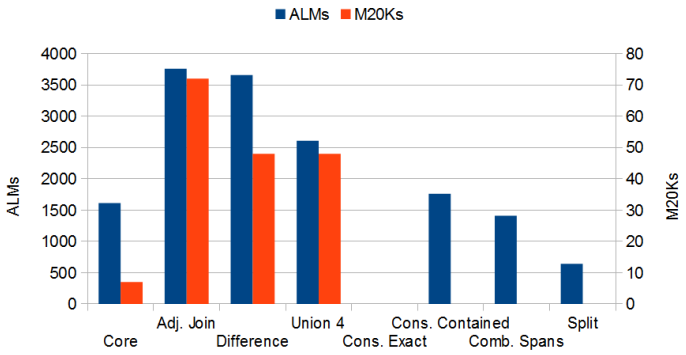


Figure 5.13: Normalized resource utilization of the core and different operator modules when servicing 16 input streams

Figure 5.13 shows the results of this comparison. The figure indicates that a single soft-core is more efficient in terms of area requirements than an equivalent number of operator instances. A significant difference can be seen for the operators with multiple input streams which require buffering and complex read logic to achieve the desired

behavior. The single stream modules have similar area requirements without the need for memory blocks.

The soft-core can be parameterized by the number of streams it supports. This is a trade-off between how many operators can be run on a single core and their performance. The more operators have to be executed on a core the slower is their individual execution time. The array may be built in an asymmetric fashion with a mix of smaller and larger cores. This can potentially optimize the utilization across the array.

To evaluate the impact of the number of virtual streams on the resource consumption again a normalized approach has been taken. Assuming 128 streams have to be processed this can be done by one single core with 128 virtual streams or four cores with 32 virtual streams each. The resource requirements by these setups can then be compared as shown in Figure 5.14.

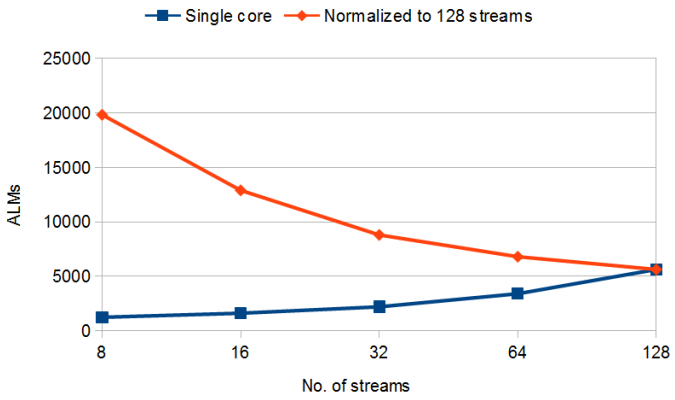


Figure 5.14: Resource utilization of a single core supporting different no. of streams as well as the normalized area consumption when supporting 128 streams

It shows that the resource requirements can be significantly reduced when using multiple virtual streams. As only the shared-memory FIFO is impacted by the number of virtual streams, the remaining logic does not need to be duplicated. The impact on the output register stage is an additional single bit to widen the stream index. If more than 128 virtual streams should be supported by the architecture the instruction width needs to be increased.

Although this evaluation neglects the performance degradation, it indicates the scalability of the approach by allowing more operators to be run on the same amount of area.

Resource	In streams	c/t (HW)	c/t (Core)
Core (16 vStreams)	16	n/a	n/a
Adjacent Join	2	1	12
Difference	2	1	13
Union of 4	4	1	4
Cons. Exact	1	1	12
Cons. Contained	1	1	12
Combine Spans	1	1	5
Split	1	1	10
Project	1	0	2

Table 5.2: Comparison of a single core to the custom hardware modules in terms of area and cycles/output tuple

5.5.2 Performance

To evaluate the performance impact of the programmable soft-core a theoretical evaluation has been performed to determine the number of required cores for a given queries. The hardware modules were designed to produce a new output tuple on every cycle if data is available. Although some operators are pipelined and have a short latency before the first tuple is produced the throughput is the relevant measure. The core performs its operations sequentially and thus requires a few cycles to produce every output tuple. The comparison is summarized in the last two columns of table 5.2. The cycles per tuple (c/t) count is one for all hardware modules while running as a single operator on a core the number of cycles per output tuple lies between 4 and 12. Some operators can be fused when running on a single core and do not require the total sum of cycles to produce a new tuple, i.e. a Combine Spans operation that follows an Adjacent Join adds only one additional cycle and reduces the amount of data to be communicated to the next core. The *Project* operation was implemented by custom wiring thus requiring 0 cycles. In terms of frequency hardware operators as well as the soft-core achieve the target of 200 MHz.

The evaluation queries from chapter 4.4 have been analyzed for their activity by using the software runtime and profiler. The queries were run against individual reference sets of documents to produce representative results. The profiler returns the average number of tuples each individual operator produces per document. As the pattern detection units used in the system consume the document one character per cycle, we can obtain an activity value per operator α_{op} by using equation 5.1. This value represents a relative rate at which a valid tuple is communicated via an edge of the query graph or a stream

in hardware.

$$\alpha_{op} = \frac{tuples_{op}}{size_{document}} \tag{5.1}$$

To calculate the maximum number of operators a single core can execute to sustain the average throughput we combine the activity rate of an operator with its processing time on a core from table 5.2. Equation 5.2 states that the sum of products of operator processing time and activity must not exceed 1, otherwise a core cannot keep up with the required activity rate at its output. Also the number of operators to be run on a single core may be limited by the available number of virtual streams.

$$ops_{core} \leq \frac{1}{\sum_{ops_{core}} (c/t)_{core_{op}} * \alpha_{op}} \tag{5.2}$$

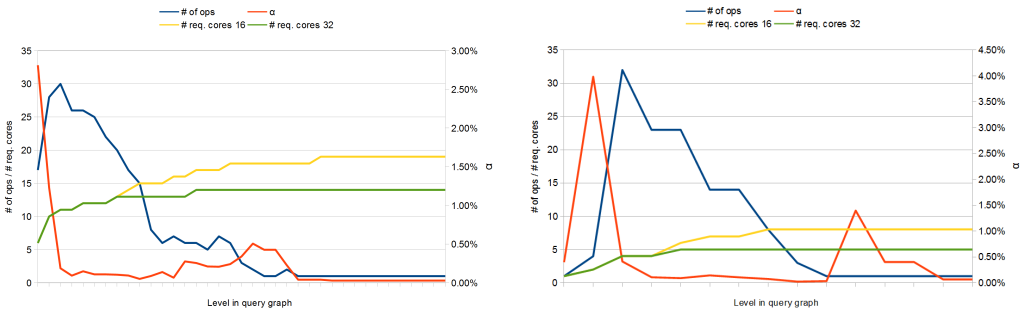
Using the profiling results this allows to estimate the required number of cores to execute a specific query on the array of cores. Fig. 5.15 shows the number of operator nodes on every level of a query graph together with the average operator activity rate α_{op}. While the activity rate drops rapidly on the first three levels, the number of operators decreases at a slower rate towards the center of the graph. The increase of α_{op} mid-way through the graph is due to *Union* operations that combine the outputs of multiple operators into a single stream. If the results of a Union operation do not require to be sorted this operation can be implemented on the array for free by assigning different results the same stream id.

Fig. 5.15 also shows the required number of cores to implement the given query using either 16 or 32 virtual streams per core. The number of cores is primarily driven by the number of operators, while only at the beginning the combination of activity and operators lead to a stronger increase. A higher number of virtual streams does not lead to a reduced number of overall cores while only using 8 streams requires a total of 38 cores for this query.

These numbers are considered estimations only as they do not take into account the possibility to fuse multiple operators when running on a single core which may lead to a lower number of required cores. But also it does not consider routing effects, as such when a core is only used as a bypass. In this case a core can only run a reduced number of operators, thus more core may be required. To evaluate these effects a compiler and mapper needs to be developed.

5.6 Related work

Coarse grained reconfigurable arrays (CGRAs) as well as manycore architectures have been studied for over 15 years. All architectures try to exploit the spatial-compute



(a) LOG3 (b) BA2

Figure 5.15: Profile of two queries showing the number of operators, the avg. operator activity rate and the required number of cores when using 16 or 32 virtual streams per core.

paradigm but vary in the programmability of the individual functional units. Introduced in 1980 by Kung et al. systolic arrays [67] where a first type of coarse grained architecture with fixed functional units and fixed routing. Although these special purpose architectures achieve high speeds they lack in programmability to allow other applications to utilize them.

The MorphoSys chip presented by Singh et al. [91] comprises of a RISC core and an attached array of reconfigurable cells (RCs). The RCs are the basic level of configuration and hold an ALU+multiplier unit with a context register which selects the input data and defines the operation to be executed in each cycle. The context register can be updated every cycle from the contents of a larger context memory. Video compression and encryption algorithms perform very efficiently on this architecture.

Goldstein et al. [54] presented PipeRench for accelerating multimedia streaming applications. The reconfigurable architecture consists of stripes which contain multiple processing elements (PEs) composed of an ALU and a pass registerfile. Multiple stripes can then be interconnected to create parallel pipelines with high computational density.

Mei et al. also target multimedia applications with their ADRES architecture [72]. It comprises of a very long instruction word (VLIW) which performs the load/store operations from and to the memory, and a reconfigurable array of reconfigurable cells which are tailored to perform data-flow kernels. Each of the cells contains a small local register file to perform local data operations.

Giefers et al. discuss a manycore architecture based on a mesh of PicoBlaze cores [52], where each core is managing the connected switch element. The architecture is integrated as a peripheral to a MicroBlaze core which controls the dataflow to the reconfigurable mesh. As a case study a mesh sorting algorithm is deployed on the architecture with significant improvements over single core execution.

Coole et al. [38] introduce intermediate fabrics as a translation layer between a user

netlist and the physical device. They are able to show a significant improvement in compilation time by using their technique for a range compute tasks. Through specialization of the intermediate fabrics the overhead of using an overlay architecture can be reduced by up to 45 %.

Capalija et al. present an coarse-grained overlay architecture for FPGAs [31]. Its functional units (FUs) are pre-synthesized single operations which can take multiple cycles to execute. These are interconnected by an elastic network compensating for execution and communication latency. This implementation achieves high frequencies of over 300 MHz and is tailored to the execution of data-flow graphs of numerical algorithms.

EGRA by Ansaloni et al. [16] is named expression grained. Similar to FPGAs which continue to include more complex hard IP macro blocks, EGRA uses a reconfigurable ALU cluster as a processing element. This allows it to perform more complex multi-staged computations in a single PE before passing the results on to the next PE.

Recent work of Hannig et al. [59] describes an invasive tightly-coupled processor array. The processing elements are similar to EGRA and use a very long instruction word (VLIW) but instead of executing in a staged fashion the PEs execute multiple functional units in parallel in a single cycle.

5.7 Summary

Over the past decade, FPGA research has presented many promising results for application speed-up. But even with the end of Dennard's scaling the integration of reconfigurable architectures into compute nodes still receives opposition. On one side application programmers do not want to wait multiple hours for their software to be synthesized and be limited to a particular FPGA architecture. Furthermore expensive EDA software needs to be licensed by the software developers. On the other hand datacenter administrators do not want FPGAs to be fully reconfigured due to the changing effects on the hardware setup.

To avoid both of these refutations, soft-layered architectures have been introduced, creating a portable abstraction layer. These architectures are often based on coarse grained reconfigurable arrays (CGRAs) or programmable manycore architectures. Compilation and reprogramming of such architectures can be very fast, while the interface logic remains static and does not change the hardware setup.

In this chapter a novel soft-core processor array has been introduced to efficiently execute relational algebra operations of text analytics queries. The architecture retains the benefits of the data streaming approach used in chapter 4 and replaces the hardware modules with a programmable soft-core. The core supports up to 128 virtual streams and provides sufficient performance, to achieve full document processing speeds by the pattern detection units. In order to map large queries, an array of cores has been introduced with a directed mesh interconnect and shared resources for pattern matching and token lookup.

The additional programmability of the core does not prevent the scalability of the approach. By using virtual streams, multiple operators can be consolidated onto a single core, resulting in a lower per operator area cost. On average a single core capable of 16 virtual streams, requires 23 % less logic resources than an equivalent number of custom hardware operator modules. This comes with a performance penalty, which should not exceed the average cycles per tuple per operator execution time. A model has been introduced, to estimate the required number of cores for a given text analytics query. All queries do not require more than 25 cores, to sustain the full document processing rates at the pattern detection units. Although this evaluation neglects the route-ability aspect when mapping a query to the processor array, it shows that a reasonable sized array can perform a wide number of text analytics queries.

CHAPTER 6

System integration

While the previous chapters focused on the functional core units to build the accelerator, this chapter discusses the integration of it into an enterprise server system. The integration is crucial to the system's overall performance as it links the relevant user software application to the accelerator core. It requires a balanced design to achieve a high utilization of the accelerator core without blocking the software side from continuing execution. Because the only valuable acceleration is the measurable acceleration from end to end.

Accelerators may be attached to a host system in many ways [106, 113]. The standard way of attaching an accelerator is via the system peripheral bus i.e. PCI Express. With the increasing transfer rates of the system bus this has become a viable option and is the standard communication link for most commercial GPGPU-, FPGA- and DSP-based accelerators. Although PCI-Express has a high bandwidth it still imposes a significant latency penalty. The software application needs to copy the data to the accelerator's on-card memory before it can be processed and eventually read back the results. Furthermore the driver software must copy the data to a location in main memory the peripheral device can access. To avoid these penalties other proposed architectures leverage in-memory computing [57, 113] where the processing hardware resides within the system's main memory. Although this paradigm allows high bandwidth at low latency, a standard programming model has yet to be developed. Another alternative are custom instructions within the processor core [19, 55]. This approach usually does not accelerate entire application tasks but rather smaller blocks of execution that reappear often in the standard instruction stream. Data transfers are provided by the same logic as for the processor core and allows access to the entire system's memory. To use the custom instructions support for them needs to be built into the compiler or dynamic recompilation takes place during execution time [28].

Another aspect of system integration is the ease of use for a software programmer. A software application is supposed to run on many systems either with or without special

purpose accelerators without any changes to the source code. If the hardware accelerator is not an essential part of the application the use of it must be as transparent as possible. Often this is achieved by replacing standard libraries with an accelerated versions. By dynamically linking these libraries they can be activated at runtime of the application without requiring a re-compilation.

This requires the hardware and driver software to provide binary compatibility on the data structures that are exchanged. If the driver software needs to perform too many memory transfers and translations the performance penalty imposed can become greater than the acceleration gained. Thus the thinner the driver layer is the fewer operations need to be performed by the host processor to interact with the accelerator. For an application that often switches between software and hardware execution a low latency interface is crucial.

6.1 Design objectives

The SystemT text analytics runtime is implemented as a Java package. Although it can be used stand-alone it is mainly used as a component within larger software applications such as IBM BigInsights or IBM InfoSphere Streams. These applications either provide users the ability to write their own AQL queries or use pre-defined queries developed by IBM internal domain experts. In either case these queries are run against large sets or fast moving set of documents such as e.g. large databases or social media streams.

While the document set is large the individual documents are relatively small. Social media documents are usually around 512 bytes in size, eMails are around a few kilobytes and larger patent applications or other publications may be up to a few megabytes. Following a document-per-thread execution model a thread within the SystemT runtime will finish processing a document before continuing with the next one. This leads to a latency requirement for the hardware accelerator as the host thread idles after submitting the document to the accelerator. To compensate for the idling threads additional ones can be launched to prepare and submit documents while the accelerator executes. This requires the accelerator to accept a larger number of work submissions without interrupting the current execution. To cope with the small document sizes on the hardware side data must be pre-fetched for the individual jobs. This closes the gap created by the transfer setup time and increases the overall document throughput rate while reducing the latency for individual jobs.

The following list summarizes the requirements for the system integration:

- Design and implementation of a communication logic that is capable of continuously supplying data to the text analytics core logic
- The communication logic must support multiple data streams that can operate independent from each other
- The communication logic must support a large number of software threads that may submit jobs at any time of operation

- Design and implementation of a software application programming interface (API)
- The API must support communication with multiple threads
- The API must be available for Java

The next section will discuss the hardware integration before the software API is presented in section 6.3. This chapter concludes with the evaluation of the interface and is summarized in 6.5.

6.2 Hardware integration

The hardware integration deals with the integration of the accelerator's core logic into a host system. It provides a communication logic to efficiently supply the core with all necessary data for continuous operation. This logic may provide access to different types of interfaces and/or memories such as e.g. PCI Express or local DDR3 memory. Furthermore the wrapper logic provides the host and ultimately the software with means to access and configure it.

For this work a POWER8-based system has been chosen as an enterprise-scale compute system. The POWER8 CPU is designed for large scale analytics workloads and provides a new interface for accelerators. The following sections 6.2.1 and 6.2.2 will introduce the host system before section 6.2.3 will present the communication logic which operates the text analytics core unit.

6.2.1 POWER8 host system

The host system used for this integration work is an IBM Power System S824 [87]. It is a 4U rack server based on two POWER8 dual-chip modules (DCM). Each of the DCMs has 6 active cores where each core is capable of executing 8 threads using simultaneous multi-threading (SMT) technology. This leads to a total of 96 threads that can be executed in parallel on the system and are visible to the operating system. The sockets in the system are connected to DDR3 memories via multiple memory hub chips [50]. These hub chips are located on each memory module and contain a level-4 cache with 16 MB of eDRAM. The maximum amount of memory is 2 TB while achieving a peak bandwidth of 192 GB/s for each socket.

In contrast to POWER7 the POWER8 processor has a direct PCI Express interface available. This reduces the system design complexity and the latency for PCIe communication. The system has twelve PCIe slots of which four can be used by the Coherent Accelerator Processor Interface (CAPI) 6.2.2.

Another key difference the previous generations of POWER processor is the endianness. POWER8 is designed to support both little-endian (LE) and big-endian modes without any particular performance difference. This enables the use of existing accelerator architectures such as GPUs and simplifies the process of porting software to the POWER

platform [92]. As an operating system the little endian version of Ubuntu 15.04 is used.

6.2.2 Coherent Accelerator Processor Interface

The Coherent Accelerator Processor Interface (CAPI) is a key feature of the POWER8 processor. It leverages the use of system-level accelerators by providing a high-bandwidth and low-latency interface built on top of the physical specification of PCIe. CAPI allows accelerators to operate in the same virtual address space as the processor cores allowing them to act as part of program execution [105]. This eliminates the need for device drivers and simplifies the software integration of the accelerator.

The enabling component for CAPI is the Coherent Accelerator Processor Proxy (CAPP) unit of the POWER8 processor chip. It is connected to the PCIe interface and acts as a representative core for the accelerator on the internal processor bus. It participates in the coherency protocols and maintains a directory of all cachelines held by the accelerator [105].

The accelerator itself uses the POWER Service Layer (PSL) to communicate with the processor's CAPP unit. The PSL is an IBM supplied IP block that can be implemented in different FPGA fabrics or in an ASIC. It contains a 256 kB cache for frequent data accesses and enables the accelerator to operate in the virtual address space. Furthermore it provides all memory operations as the cores itself such as e.g. reads, writes, reservations or locks. The actual accelerator is referred to as Accelerator Function Unit (AFU) and communicates with the PSL. Figure 6.1 shows the various units required for a CAPI system.

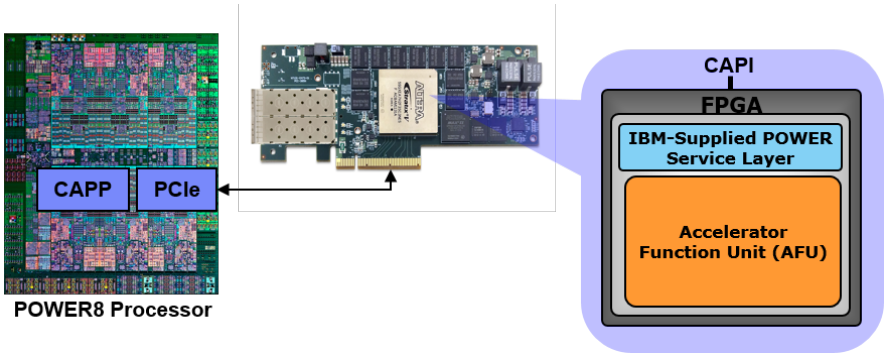


Figure 6.1: POWER8 with CAPP and attached CAPI accelerator

The simplest access method to the AFU is via a memory-mapped I/o (MMIO) interface that can read or write 32-bit or 64-bit words. The AFU must contain a configuration memory referred to as AFU descriptor which can be accessed via MMIO reads. This descriptor contains information for the hypervisor and operating system how the accelerator can be used and what resources it requires. For example it configures the minimum and maximum number of interrupts but also which programming model it supports. Two main categories of programming models are available, a dedicated model

and a shared model.

For the dedicated model the accelerator belongs to a single process running on the host side. The application can use the CAPI API *libcxl* to open, start and use an AFU. If an application tries to open an AFU which is in use by another process the function call will return with an error allowing the application to run in a non-accelerated mode. This model is suited for appliances where the entire system is optimized to perform a specific task such as i.e. a database server. Also application servers can employ this model as a single server provides services over the network. But if there are multiple different application types running on the system or the system is virtualized the dedicated model limits the use of the accelerator.

For sharing the accelerator across multiple processes two shared programming models are available. For both models the operating system and the hypervisor maintain a list of work elements that where requested by different processes. The AFU needs to be able to interrupt the execution of a specific work element, save the required context and continue with another work element. In the time shared model the hypervisor controls the time slicing. A request to stop execution is sent to the AFU which needs to react within a specified amount of time and pick up the new work element. Another time sliced model is the AFU directed mode. In this model the AFU controls the time slicing and services the queue of work elements.

Figure 6.2 shows in detail the various components used for a CAPI-based design. On the software (host) side the application code uses the libcxl API to control an AFU while data communication occurs via shared virtual memory. The CAPP unit communicates with the PSL on the FPGA over the PCIe interface and establishes coherency. The PSL at last communicates with the AFU via a set of five interfaces. A control interface to start, reset and stop the AFU from the host side. A MMIO interface to read and write small pieces of data. The command interface is driven by the AFU and allows it to issue commands such as reads, writes or locks on specified virtual 64-bit wide addresses. The commands are reordered to achieve the best performance and every command is acknowledged by the PSL via the response interface. Data transfers are performed via the buffer interface which provides separate read and write buses.

6.2.3 Text Analytics AFU

The Text Analytics AFU is the wrapper logic around a compiled annotation operator graph unit or a Turtle array. In both cases it is responsible to receive jobs from software threads, perform the necessary read and write operations and signal the completion of an individual job. The AFU was designed such that it can be parameterized to service a given number of document streams. This parameter needs to be set at synthesis time and is usually limited by the available resources depending on the core unit's size.

Figure 6.3 shows all units of the text analytics AFU top level. The control block reacts to commands on the PSL's control interface. For a reset command it generates a three cycle wide active high reset pulse that is fed to the entire AFU. It also controls the global clock enable signal across the AFU. Only once a start command has been issued

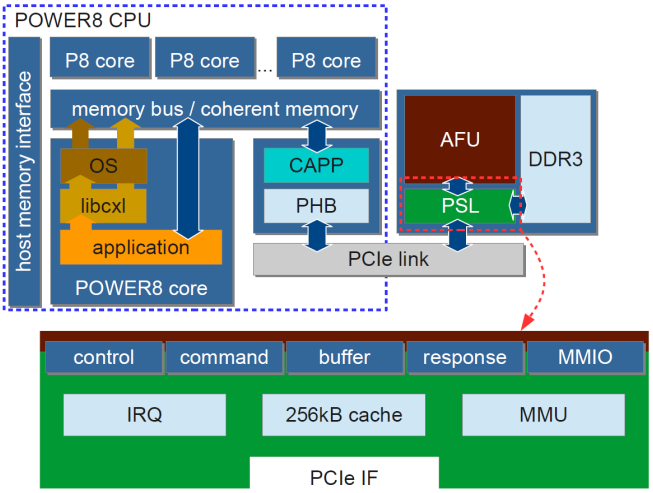


Figure 6.2: CAPI software and hardware stack [53]

by the PSL the AFU becomes active and reacts to any other interface. The control unit is also responsible for signaling the correct shutdown of the AFU. A shutdown can be triggered from the software by writing a 1 to the MMIO register at offset 0x0.

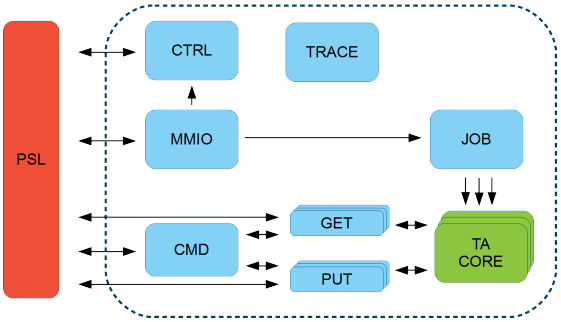


Figure 6.3: AFU top-level

The MMIO unit acknowledges the read and write operations on the PSL's MMIO interface. The unit also holds the AFU descriptor which is read during system boot time. The AFU is designed to operate as a dedicated accelerator for one host process only as this was the only programming model available at the time of implementation. The first 64-bit register at offset 0x0 is used as a control register. By writing defined values to it the software can control certain functions of the AFU such as e.g. shutdown. A read from this register return the states of all statemachines within the AFU. Offsets 1-3 are read-only registers and are used to read out the trace buffer for debugging purposes. The trace buffer stores the last 256 issued commands and received responses. Offset 4 is used as a write-only register and is used to submit a job pointer which is a 64-bit virtual address to a job struct (see 6.3). The job pointer gets immediately forwarded to

the job unit for further processing.

The cmd unit is the central unit for memory operations and interrupts. The cmd unit connect to the PSL's command and response interfaces and is responsible for maintaining the available credits the AFU has for submitting PSL commands. When an AFU start is issued by the control unit the cmd unit captures the maximum available credits indicated by the PSL. Every submitted command requires one credit which is returned when the command gets acknowledged via the response interface. The commands are identified by tags. The cmd unit keeps available tags in a bucket memory as the commands are serviced by PSL in a best performance order and may be returned reordered. The unit further acts as an arbiter for all client units connected to it and abstracts the PSL commands to a simple read, write or interrupt request signaling. When the cmd unit grants a request it indicates the used tag to the requesting client who is then responsible for all operations associated with this tag. Figure 6.4 shows the overall architecture of the cmd unit.

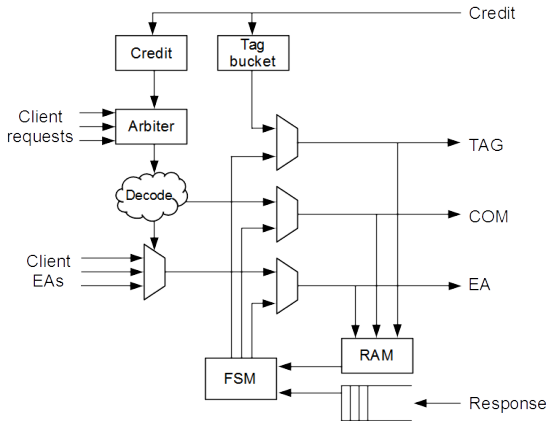


Figure 6.4: AFU command unit

As the memory operations are performed out-of-order the arriving data needs to be re-ordered before it can be passed to the core units. For this purpose a GET unit has been implemented that submits the memory read requests via the cmd unit and reacts to incoming data on the buffer interface. The GET unit receives in-order read request from the core logic, forwards them to the cmd unit and receives a tag for each request. The tag is decoded into a fence mask to check whether a response tag belongs to a particular GET unit instance. Furthermore an index is stored at the position of the tag value to indicate which memory request a tag services. When data arrives via the buffer interface a tag-to-index lookup is performed to determine the data's destination address in the data RAM. This can happen multiple times by the PSL so no further action is taken other than writing the data to the data RAM. The memory transaction is completed with a positive response on the response interface. The GET module performs a tag-to-index lookup for the response tag and sets a bit to high at the index position in the response mask. A read counter provides the read address for the data RAM and checks whether it has been set in the response mask. If so valid data is

indicated to the receiving core unit which needs to be ready to accept new data. Once data is transferred the index bit in the response mask is set back to zero.

A similar mechanism is implemented for the write operation by the PUT units. In this case the write data arrives in-order but is collected by the PSL out-of-order. The main reason to keep track of the ordering is the ability to know when a transfer has been completed. Only when all results have been transferred a job can be marked as complete and a signal can be sent to the software to continue execution. Both PUT and GET units operate on entire cachelines and do not perform any coalescing. The requests can be marked using an end flag. The units will then mark the last pieces of incoming data as end data or send a *done* pulse in case of PUT.

Jobs are submitted via MMIO writes and are forwarded to the job management unit (JOB). It receives a 64 bit virtual address pointer to a job structure. This job structure holds all necessary information to process a document through a text analytics pipeline and is set up by the software (6.3). The job management unit fetches the job structures and stores them in a FIFO queue. The core units indicate via a ready signal whether they can accept a new job. If a new job is available the job management unit will forward it to one of the core units using a round-robin arbitration. This prefetching scheme ensures that the actual struct data is available when a core unit becomes ready.

```
struct afu_ta_wed {
  unsigned char *doc; /* Document pointer */
  unsigned int *tok; /* Token pointer */
  unsigned int *sts; /* Status list pointer */
  unsigned int **bufs; /* Pointer to result buffers */
  unsigned int docSize; /* Document size in Bytes */
  unsigned int tokSize; /* Token size in Bytes */
  unsigned int bufsSize; /* Result buffer size in Bytes */
  unsigned int bufsCount; /* Number of result buffers */
};
```

Listing 6.1: Job struct for the text analytics AFU

The core units operate on the actual job structures received from the job management unit. A core unit is responsible for the read and write request generation towards the GET and PUT modules and for the serialization and de-serialization of data blocks. For the incoming data the job structure holds a pointer to the document and token buffers. Furthermore it contains the size values for both of these buffers in number of Bytes. With this information the core unit can generate all necessary read requests and produce a character and token stream that is sent to the compiled query or the processor array. Figure 6.5 illustrates the read transfers and input side of the core units.

On the output side of the query module the core unit collects the results data and stores it in the appropriate memory buffers in the host memory. For this the job structure contains a pointer to a list of result buffer pointers which all have the same initial size indicated by *bufsSize*. The number of result buffers is indicated by *bufsCount* and can be a maximum of 512 as this is the maximum number of entries of the lookup tables within the core unit. The result buffer pointers are preloaded into a lookup table while the result count table is reset to zero during this process. The query unit will produce

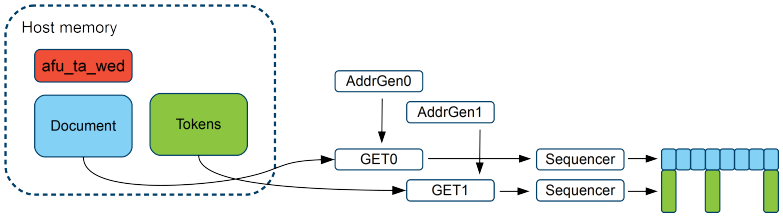


Figure 6.5: AFU core input side

outputs of 16 bytes together with a result index which is used as a key to the lookup tables. The data will be batched until a complete cacheline is available for a result index. Only then the data and the write request are submitted to the PUT unit.

Once the query unit indicates that is has finished processing the core unit will start fetching the next result buffer pointers. At the same time it scans the current lookup tables for remaining results that have not yet been written to host memory and submits them. Once all data is written to the host the result counts are reported back by writing them to the status (*sts) location with an offset of one. The first 32-bits in the status field are reserved for thread control mechanisms. A thread will poll this location to wait for a AFU_DONE value to continue operation. Figure 6.6 illustrates the write transfers and the output side of the core units.

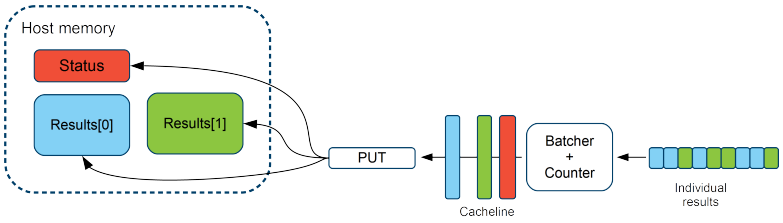


Figure 6.6: AFU core output side

6.3 Software integration

On the software side integration has to take place on two levels. On a low level there needs to be a programming interface to communicate with the hardware and setup the appropriate data structures for it. This also includes data structures and functions to support multiple threads. At a higher level language such as e.g. Java the integration work should provide a simple to use programming interface based around functions that abstract the lower level API. A user should not be bothered by implementation details but be exposed to the features on a functional level.

At a low level the core functionality is provided by a C library called libafu_ta. It wraps the basic libcxl calls and provides additional simplicity to deal with the text acceleration

AFU. To start interaction with the AFU the library provides the afu_ta_start() function which returns zero on success. This function opens the CXL device and attaches the AFU to the current process. It also maps the MMIO register space and saves the necessary pointers in a global struct. This allows all future libafu_ta calls to refer to this struct. To stop the AFU, close the device and free all data structures the equivalent stop() function must be called after all interaction with the accelerator has been finished.

To communicate with the AFU the library wraps MMIO read and write functions for 64 bit values. These are single instructions in the final program code followed by a synchronization instruction to ensure the operation has completed before continuing. These functions are used i.e. for the readout of the trace buffer contents or the submission of a pointer to a job structure. At a more functional level the library provides three functions to create, submit and wait for completion of a job. The flowchart in Figure 6.7 shows how to use these functions to sequentially submit a set of documents. After the AFU has been started a job memory structure can be created using the init_wed() function (WED refers to the term work element descriptor an is synonymous to a job structure). Once the job structure has been created and populated with data the job can be submitted and execution waits for its completion. After the AFU has finished working on the document the results can be used for further processing and the job structure can be freed. If all documents in the set have been processed the stop() function is called to shutdown the AFU and allow other processes to access it again.

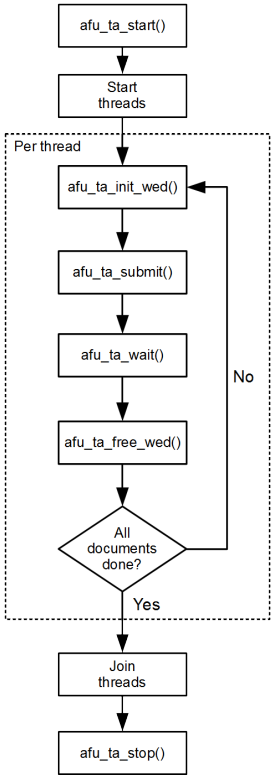


Figure 6.7: Flowchart to use the AFU by libafu_ta

6.3.1 Java interface

In order to use the AFU from within a Java application the libafu_ta function calls must be made available to be called from the Java virtual machine (JVM). For this purpose Java provides the Java Native Interface (JNI) [39]. It allows Java classes to define *native* methods that may be implemented in C, C++ or assembler. The methods declared as native are exported to a C header file which get automatically generated by the *javah* tool. This header file defines the exact naming and type definition for each function which corresponds to a single method in a Java class. These functions can then be implemented by the user.

In many cases it is sufficient to create a one to one mapping of the existing C library functions in a Java class and create wrapper JNI code to access and cast some Java datatypes. Also for the libafu_ta this is the first approach to make all functions available to a Java application. This way the previously described access to the AFU can be performed just equally in Java. Figure 6.8 illustrates the complete software stack provided to an application to interact with the text analytics AFU.

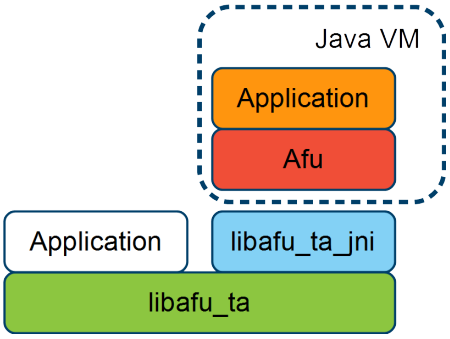


Figure 6.8: Library stack for the AFU

To simplify the submission of documents to the accelerator the Java class *Afu* provides a method *runSubgraph()* that takes a string and an array of token offsets and performs all necessary steps before returning. Once the accelerator has been started this method can be called by multiple threads for processing documents. Although the threads can submit jobs in parallel each thread will wait until the completion of its own document before continuing to submit a new one. This introduces a setup penalty for each thread that maybe hidden by launching more threads that prepare and submit documents in the meantime.

To avoid this penalty and avoid launching too many software threads an asynchronous interface is provided by the methods forkSubgraph() and joinSubgraph(). Instead of waiting for the completion of a job forkSubgraph() submits a job to the accelerator and returns immediately. This allows the software to continue operation, performing other computations or submitting a next document until it requires the results from the accelerator. At this point the program needs to call joinSubgraph() which will block until the accelerator is finished for a specific job indicated by an argument.

6.4 Evaluation

The system integration is evaluated in terms of performance impact as there are multiple communication interfaces that interact with each other. The hardware interface needs to be able to supply the compiled text analytics query core with a continuous stream of documents. This can be evaluated by creating a large batch of documents in system memory and submit all created jobs at once. This evaluation can take into account the document size which determines a single job size and the transfer setup costs.

A similar measurement can be set up to measure the impact of the multi-threaded software interface. Each thread prepares a job in system memory and submits it to the accelerator. The thread will wait for completion of the job before resubmitting the job again. The time between job completion and re-submission is software execution time and can be adjusted for the test application. The throughput performance of this test depends on the number of threads used and the time each thread waits before re-submitting a job.

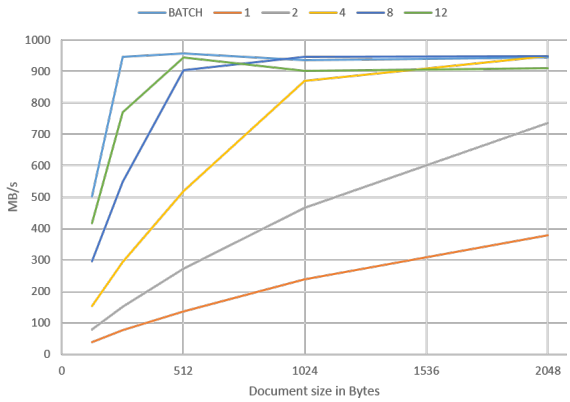


Figure 6.9: Interface throughput for different number of submitting threads and batch mode

Figure 6.9 shows the measured document throughput results in MB/s, both of these tests over the document size in bytes. The BATCH case marks the maximum possible throughput which is then limited by the AFU interface logic. At a document size of 128 B the performance is about 500 MB/s but immediately reaches its maximum of 950 MB/s with document sizes of 256 B and larger. This test has been performed with a test query, that is capable of running at 250 MHz. Thus the possible peak throughput is 1 GB/s.

When exercising the AFU with sequential call by the multi-threaded software interface the performance depends on the number of software threads running and the document size. While a single thread only achieves 40 MB/s with 128 B sized documents the throughput increases linearly with the number of threads close to the maximum of 500 MB/s limited by the hardware. Also the document size increases the throughput

linearly for all cases. By using 12 threads the full performance of the BATCH mode can be achieved with 512 B documents.

The last level interface is the Java Native Interface (JNI) connecting the Java virtual machine (JVM) to the accelerator. The interface has to copy the document and token offset data from the Java heap to native code memory before submitting a job. To test the performance of the JNI a similar multi-threaded test can be performed. A set of documents is prepared within the JVM for multiple threads. Each thread then calls a dummy version of the *runSubgraph()* method which will copy the according data to native code and return immediately after the data has been copied. This ensures only the impact by the JNI can be measured.

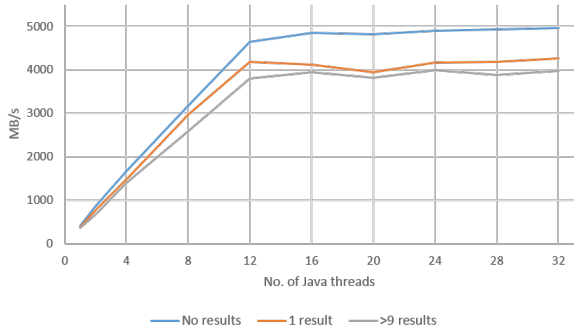


Figure 6.10: Java Native Interface throughput for different number of submitting threads

The measurement results are plotted in Figure 6.10 as the throughput of document characters per second over the number of Java threads. The throughput increases linearly from 1 to 12 threads where it reaches a peak of 4.6 GB/s. When increasing further the number of threads the throughput does not increase much anymore. The maximum observed document throughput rate is about 5 GB/s with no results returning to the JVM.

If the JNI has to return results back to the JVM the document throughput rate drops significantly for higher numbers of threads. Up to 1 GB/s less throughput rate can be achieved when returning 9 or more results. The number of results does not change the performance impact as much as returning results at all. This seems to be caused by the creation of a new integer array on the Java heap when returning results oppose to returning a Null when there are no results.

6.4.1 Final performance evaluation

For a final evaluation, the reference queries from chapter 4.4 have been recompiled for the Altera Stratix V FPGA, used by the POWER8-based system. Although this is a newer generation FPGA, the number of streams was not increased beyond four, because a larger portion of the FPGA is occupied by the CAPI PSL logic. On the POWER7

system, this logic was put onto the separate system FPGA, which reduced the interface logic on the actual application FPGA. Nevertheless, the placed and routed queries are all capable of achieving an operating frequency of 200 MHz, which is 20 to 33 % higher compared to the previous Stratix IV FPGA.

The software implementation was run in a scale-out test, utilizing up to 96 threads on the POWER8 server. The best result was selected, regardless of the number of threads. The documents where pre-loaded to Java memory to avoid any file I/O operations. The application then runs for 30 seconds and counts the number of characters processed. On average the POWER8 system shows a 29 % document throughput improvement over POWER7, using Java 8. Figure 6.11 summarizes the performance of the individual systems when processing the reference documents, and shows the throughput of a single FPGA processing stream on the respective system.

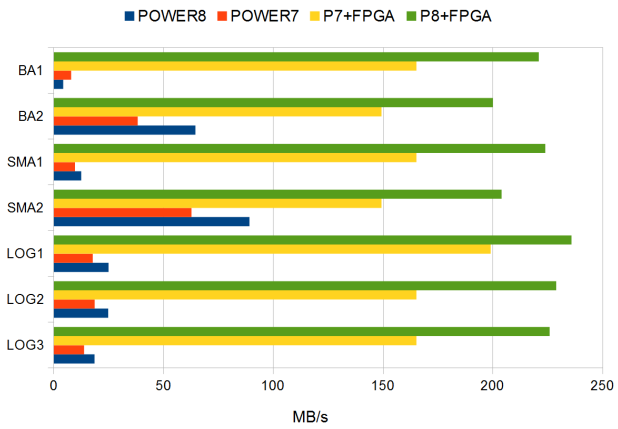


Figure 6.11: Performance of POWER7 and POWER8 systems and their respective single stream FPGA accelerator

It shows, that new generations of reconfigurable devices can keep up with the performance improvements of new processors. The accelerated system performs 36 times better in terms of throughput than the pure software implementation, excluding query BA1. BA1 is the only query which is running slower on the POWER8 system and can be executed up to 200 times faster on the FPGA.

6.5 Summary

Although the increased performance and energy efficiency of an accelerator can be high at an architectural level, the integration into a host system can reduce this benefit significantly. Efficient and reliable system integration is a key component to successfully deploy an accelerator in an enterprise environment. It comprises of a hardware communication logic that can operate the core logic to its full capacity, as well as software

components for low and high level languages.

This chapter presented all necessary components to efficiently deploy the text analytics accelerator within a POWER8-based enterprise system. The hardware integration leverages the novel coherent accelerator processor interface (CAPI) to operate in the virtual address space of the software process. The architecture was designed such that it supports multiple software threads submitting document processing jobs to it and shared memory completion notification. To avoid transfer setup penalties for small documents the communication logic implements a pre-fetching scheme while waiting for the query core to finish processing. Additional trace and debug registers simplify the error detection and debug.

On the software side two libraries are provided to communicate with the accelerator. One library provides the basic functionality to start and stop the accelerator as well as setup the appropriate data structures in memory which the accelerator can interpret. Also functions for job submission and wait for completion are available to exercise the accelerator from multiple threads. A second library enables the use of the accelerator from with the Java virtual machine. By using the Java Native Interface documents are copied to native code and processed on the accelerator before returning to the Java heap. This library also provides an asynchronous call scheme to achieve high document throughput with a fewer number of threads.

Measurements indicate that the impact of the integration are acceptably low reaching 95% of the query core's peak throughput rate. End-to-end a document processing rate of up to 943 MB/s has been measured, which is up to 71 times faster than the multi-threaded software implementation. This performance can be achieved by either running the accelerator in a batched mode where a set of documents is prepared and then sent off to the accelerator for processing. An alternative integration into a software application is a sequential call to the accelerator. This type of integration may cost less effort for the software developer and can still reach high performance by utilizing multiple threads.

Although the new generation POWER processor is up to 29 % faster when performing text analytics, the presented accelerator framework can maintain a significant performance lead. By using a commercial off the shelf FPGA card, this integration demonstrates the feasibility and impact of reconfigurable architectures for text-based information extraction.

CHAPTER 7

Conclusion

With the arrival of Big Data the task of information extraction has gained high interest. The ability to process large amounts of unstructured textual data and distill a piece of structured information from it, has become an important task in today's enterprise businesses. The task of text analytics poses challenges to modern processor architectures which limit the performance at which information can be extracted from a document corpus.

This dissertation presents an integrated accelerator framework for text analytics based on reconfigurable architectures. The framework augments an existing query-based text analytics application with a hardware compiler which allows queries to be offloaded to an FPGA-based co-processor. The hardware compiler the spatial compute paradigm on a reconfigurable architecture by creating a custom datapath for every query. Every datapath combines pattern detection tasks with relational algebra tasks which all operate in parallel to create a highly efficient streaming architecture for text analysis.

The evaluation of the framework demonstrates significant improvements in performance compared to the original software product running on a powerful enterprise server system. These measurements include end-to-end communication costs from the host's main memory and back. Also the power consumption of the accelerated system is lower than the pure software implementation when processing data. This suggests that the accelerator framework is a valuable addition to increase the performance for text analytics queries and free processor cycles for additional tasks.

Because hardware compilation may be considered a static approach this dissertation also presents a programmable architecture for efficiently executing relational algebra operations which are specific to text analytics. The architecture consists of an array of custom soft-core processors that leverage the streaming dataflow with an added level of programmability. The evaluation of the architectures shows that it is sufficiently fast to avoid the pattern detection step from stalling. Furthermore by consolidating multiple

relational operators onto the same chip area the architecture allows the execution of large complex queries.

The impact of text analytics acceleration on a reconfigurable platform has been demonstrated. It enables users to go from offline to online processing of large unstructured text data.

7.1 Outlook

While the presented work has been integrated into enterprise systems and has proven strong results in various demonstrators there remain open issues and questions. The pattern detection units in this work operate with ASCII characters which limits them to the English language. Enabling UTF-8 support is a key task to enable the accelerator for international use. As the ASCII character set is a subset of UTF-8 the extension to the pattern detection units may be trivial but has an impact on the offsets of the document. Because a UTF-8 character may consist of multiple bytes the character offset of a token does not point to the correct memory location anymore.

One research direction is the HW/SW partitioning of an annotation operator graph. For large queries it may be necessary or for custom functions to be executed on the host processor. Such graphs must be partitioned in an optimized fashion with communication and resource constraints in mind. The presented framework provides the necessary hardware compilation step to perform such research.

To fully utilize the presented programmable architecture a compiler needs to be developed. It needs to be able to partition the relational operators across the processor array thereby utilizing the profiling information available to avoid performance bottlenecks. Optimizations to the annotation operator graph should be explored to benefit from architectural features.

Another research direction is dynamic adaptation of compute resources on the programmable architecture. Depending on the load of a particular virtual stream it may be moved to another processor. Metrics and methods needs to be found to extend and evaluate such a system. The presented programmable architecture provides a possible baseline to be extended.

Acronyms

AFU	accelerator functional unit
ALU	arithmetic logic unit
AOG	annotation operator graph
ALM	adaptive logic module
API	application programming interface
ASCII	american standard code for information interchange
ASIC	application-specific integrated circuit
AQL	annotation query language
BaRT	balanced routing table
BFSM	BaRT-based finite state machine
CAM	content addressable memory
CAPI	coherent accelerator processor interface
CAPP	coherent accelerator processor proxy
CGRA	coarse-grained reconfigurable array
CMOS	complementary metal oxide semi-conductor
CPU	central processing unit
DMC	dual-chip module
DFA	deterministic finite automaton
DMA	direct memory access
DSP	digital signal processor
EDA	electronic design automation
FIFO	first-in first-out
FPGA	field programmable gate array
FU	functional unit
FSM	finite state machine
GPU	graphics processing unit
GPGPU	general-purpose computing on graphics processing unit
HDL	hardware description language
HLS	high-level synthesis
HPC	high-performance computing
IP	intellectual property
ISA	insturction set architecture

JNI	Java native interface
JSON	JavaScript object notation
JVM	Java virtual machine
LUT	look-up table
M20K	internal FPGA memory block
M9K	internal FPGA memory block
MMIO	memory mapped input/output
NER	named entity recognition
NFA	non-deterministic finite automaton
NIDS	network intrusion detection system
NLP	natural language processing
NoC	network on chip
PDU	pattern detection unit
PE	processing element
PSL	POWER service layer
RAM	random access memory
RISC	reduced instruction set computer
ROM	read-only memory
SIMD	single-instruction multiple-data
SMT	simultaneous multi-threading
SQL	structured query language
UTF	unicode transformation format
VLIW	very long instruction word
WED	work element descriptor

Author's Publications

[1] Raphael Polig, Heiner Giefers, and Walter Stechele. A Soft-Core Processor Array for Relational Operators. In *Application-Specific Systems, Architectures and Processors (ASAP), 2015 IEEE 26th International Conference on*, pages 59–66. IEEE, 2015.

[2] Raphael Polig, Kubilay Atasu, Heiner Giefers, and Laura Chiticariu. Compiling text analytics queries to FPGAs. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[3] Raphael Polig, Kubilay Atasu, Laura Chiticariu, Christoph Hagleitner, Peter H. Hofstee, Frederick R. Reiss, and Huaiyu Zhu. Hardware-accelerated text analytics. *IEEE Hotchips, 2014*, IEEE, 2014.

[4] Raphael Polig, Kubilay Atasu, Laura Chiticariu, Christoph Hagleitner, Peter H. Hofstee, Frederick R. Reiss, and Huaiyu Zhu. Giving text analytics a boost. In *IEEE Micro, 2014*, pages 6–14. IEEE, 2014.

[5] Raphael Polig, Kubilay Atasu, and Christoph Hagleitner. Token-based dictionary pattern matching for text analytics. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–6. IEEE, 2013.

[6] Kubilay Atasu, Raphael Polig, Christoph Hagleitner, and Frederick R Reiss. Hardware-accelerated regular expression matching for high-throughput text analytics. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–7. IEEE, 2013.

[7] Kubilay Atasu, Raphael Polig, Jonathan Rohrer, and Christoph Hagleitner. Exploring the design space of programmable regular expression matching accelerators. *Journal of Systems Architecture*, 59(10):1184–1196, 2013.

[8] Kanak Agarwal and Raphael Polig. A high-speed and large-scale dictionary matching engine for information extraction systems. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 59–66. IEEE, 2013.

Bibliography

[9] Altera and xilinx report: The battle continues. http://seekingalpha.com/article/85478-altera-and-xilinx-report-the-battle-continues. Accessed: 2015-01-21.

[10] FPGA cell example png. http://commons.wikimedia.org/wiki/File:FPGA_cell_example.png. Accessed: 2015-01-21.

[11] Hadoop. http://hadoop.apache.org/. Accessed: 2015-01-19.

[12] IBM: What is Big Data? Bringing Big Data to enterprise. http://www-01.ibm.com/software/data/bigdata/. Accessed: 2014-03-21.

[13] An illustration of Relational model concepts. http://en.wikipedia.org/wiki/File:Relational_model_concepts.png. Accessed: 2015-03-06.

[14] Kanak Agarwal and Raphael Polig. A high-speed and large-scale dictionary matching engine for information extraction systems. In *Application-Specific Systems, Architectures and Processors (ASAP), 2013 IEEE 24th International Conference on*, pages 59–66. IEEE, 2013.

[15] Alfred V Aho and Margaret J Corasick. Efficient string matching: an aid to bibliographic search. *Communications of the ACM*, 18(6):333–340, 1975.

[16] Giovanni Ansaloni, Paolo Bonzini, and Laura Pozzi. EGRA: A coarse grained reconfigurable architectural template. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 19(6):1062–1074, 2011.

[17] Douglas E Appelt and Boyan Onyshkevych. The common pattern specification language. In *Proceedings of a workshop on held at Baltimore, Maryland: October 13-15, 1998*, pages 23–30. Association for Computational Linguistics, 1998.

[18] Kubilay Atasu. Resource-efficient regular expression matching architecture for text analytics. In *Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference on*, pages 1–8. IEEE, 2014.

[19] Kubilay Atasu, Wayne Luk, Oskar Mencer, Can Özturan, and Günhan Dündar. Fish: Fast instruction synthesis for custom processors. *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, 20(1):52–65, 2012.

[20] Kubilay Atasu, Raphael Polig, Christoph Hagleitner, and Frederick R Reiss. Hardware-accelerated regular expression matching for high-throughput text analytics. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–7. IEEE, 2013.

[21] Kubilay Atasu, Raphael Polig, Jonathan Rohrer, and Christoph Hagleitner. Exploring the design space of programmable regular expression matching accelerators. *Journal of Systems Architecture*, 59(10):1184–1196, 2013.

[22] Ricardo Baeza-Yates and Gaston H Gonnet. A new approach to text searching. *Communications of the ACM*, 35(10):74–82, 1992.

[23] Jason Baldridge. The opennlp project. *URL: http://opennlp. apache. org/index. html,(accessed 2 February 2015)*, 2005.

[24] Luiz André Barroso and Urs Hölzle. The case for energy-proportional computing. *IEEE computer*, 40(12):33–37, 2007.

[25] Steven Bird. NLTK: the natural language toolkit. In *Proceedings of the COLING/ACL on Interactive presentation sessions*, pages 69–72. Association for Computational Linguistics, 2006.

[26] Joao Bispo, Ioannis Sourdis, Joao MP Cardoso, and Stamatis Vassiliadis. Regular expression matching for reconfigurable packet inspection. In *Field Programmable Technology, 2006. FPT 2006. IEEE International Conference on*, pages 119–126. IEEE, 2006.

[27] Tobias Bjerregaard and Shankar Mahadevan. A survey of research and practices of network-on-chip. *ACM Computing Surveys (CSUR)*, 38(1):1, 2006.

[28] Darrell Boggs, Gary Brown, Nathan Tuck, and K Venkatraman. Denver: Nvidia's first 64-bit arm processor. 2015.

[29] Erik Cambria and Bebo White. Jumping nlp curves: A review of natural language processing research. *IEEE Computational Intelligence Magazine*, 9(2):48–57, 2014.

[30] Andrew Canis, Jongsok Choi, Mark Aldham, Victor Zhang, Ahmed Kammoona, Jason H Anderson, Stephen Brown, and Tomasz Czajkowski. Legup: high-level synthesis for fpga-based processor/accelerator systems. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, pages 33–36. ACM, 2011.

[31] Davor Capalija and Tarek S Abdelrahman. A high-performance overlay architecture for pipelined execution of data flow graphs. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–8. IEEE, 2013.

[32] Donald D Chamberlin and Raymond F Boyce. Sequel: A structured english query language. In *Proceedings of the 1974 ACM SIGFIDET (now SIGMOD) workshop on Data description, access and control*, pages 249–264. ACM, 1974.

[33] George Chrysos. Intel® Xeon Phi Coprocessor - the architecture. *Intel Whitepaper*, 2014.

[34] Eric S Chung, John D Davis, and Jaewon Lee. Linqits: Big data on little clients. In *ACM SIGARCH Computer Architecture News*, volume 41, pages 261–272. ACM, 2013.

[35] Charles LA Clarke, Gordon V. Cormack, and Forbes J. Burkowski. An algebra for structured text search and a framework for its implementation. *The Computer Journal*, 38(1):43–56, 1995.

[36] Edgar F Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, 1970.

[37] W Cohen and Andrew McCallum. Information extraction from the world wide web. KDD, 2003.

[38] James Coole and Greg Stitt. Intermediate fabrics: Virtual architectures for circuit portability and fast placement and routing. In *Hardware/Software Codesign and System Synthesis (CODES+ ISSS), 2010 IEEE/ACM/IFIP International Conference on*, pages 13–22. IEEE, 2010.

[39] Oracle Corp. Java native interface specification. *URL: http://docs. oracle. com/-javase/8/docs/technotes/guides/jni/spec/jniTOC. html,(accessed 4 August 2015)*, 2015.

[40] Hamish Cunningham. GATE, a general architecture for text engineering. *Computers and the Humanities*, 36(2):223–254, 2002.

[41] Hamish Cunningham, Diana Maynard, and Valentin Tablan. Jape: a java annotation patterns engine. 1999.

[42] Christopher Dennl, Daniel Ziener, and Jürgen Teich. On-the-fly composition of FPGA-based SQL query accelerators using a partially reconfigurable module library. In *Field-Programmable Custom Computing Machines (FCCM), 2012 IEEE 20th Annual International Symposium on*, pages 45–52. IEEE, 2012.

[43] George Doud. Accelerating the data plane with the tile - mx manycore processor. *Linley Data Center Conference*, 2015.

[44] Hadi Esmaeilzadeh, Emily Blem, Renee St Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark silicon and the end of multicore scaling. In *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*, pages 365–376. IEEE, 2011.

[45] Gerald Estrin. Organization of computer systems: the fixed plus variable structure computer. In *Papers presented at the May 3-5, 1960, western joint IRE-AIEE-ACM computer conference*, pages 33–40. ACM, 1960.

[46] David Ferrucci and Adam Lally. UIMA: an architectural approach to unstructured information processing in the corporate research environment. *Natural Language Engineering*, 10(3-4):327–348, 2004.

[47] David A Ferrucci. Introduction to this is watson. *IBM Journal of Research and Development*, 56(3.4):1–1, 2012.

[48] Gerhard Fettweis and Ernesto Zimmermann. Ict energy consumption-trends and challenges. In *Proceedings of the 11th International Symposium on Wireless Personal Multimedia Communications*, volume 2, page 6, 2008.

[49] Dayne Freitag. Multistrategy learning for information extraction. In *ICML*, pages 161–169, 1998.

[50] Joshua Friedrich, Hung Le, William Starke, Jeff Stuechli, Balaram Sinharoy, Eric J Fluhr, Daniel Dreps, Victor Zyuban, Gregory Still, Christopher Gonzalez, et al. The POWER8 tm processor: Designed for big data, analytics, and cloud environments. In *IC Design & Technology (ICICDT), 2014 IEEE International Conference on*, pages 1–4. IEEE, 2014.

[51] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *SOFTWARE - PRACTICE AND EXPERIENCE*, 30(11):1203–1233, 2000.

[52] Heiner Giefers and Marco Platzner. A many-core implementation based on the reconfigurable mesh model. In *Field Programmable Logic and Applications, 2007. FPL 2007. International Conference on*, pages 41–46. IEEE, 2007.

[53] Heiner Giefers, Raphael Polig, and Christoph Hagleitner. Accelerating arithmetic kernels with coherent attached fpga coprocessors. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1072–1077. EDA Consortium, 2015.

[54] Seth Copen Goldstein, Herman Schmit, Mihai Budiu, Srihari Cadambi, Matthew Moe, and R Reed Taylor. Piperench: A reconfigurable architecture and compiler. *Computer*, 33(4):70–77, 2000.

[55] Diego González, Guillermo Botella, Carlos García, Manuel Prieto, and Francisco Tirado. Acceleration of block-matching algorithms using a custom instruction-based paradigm on a nios ii microprocessor. *EURASIP Journal on Advances in Signal Processing*, 2013(1):1–20, 2013.

[56] Michael Gschwind. The cell broadband engine: exploiting multiple levels of parallelism in a chip multiprocessor. *International Journal of Parallel Programming*, 35(3):233–262, 2007.

[57] Said Hamdioui, Lei Xie, Hoang Anh Du Nguyen, Mottaqiallah Taouil, Koen Bertels, Henk Corporaal, Hailong Jiao, Francky Catthoor, Dirk Wouters, Linn Eike, et al. Memristor based computation-in-memory architecture for data-intensive applications. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, pages 1718–1725. EDA Consortium, 2015.

[58] Stratix V Device Handbook. 14th ed., 2014.

[59] Frank Hannig, Vahid Lari, Srinivas Boppu, Alexandru Tanase, and Oliver Reiche. Invasive Tightly-Coupled Processor Arrays: A Domain-Specific Architecture/Compiler Co-Design Approach. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(4s):133, 2014.

[60] Nikos Hardavellas, Michael Ferdman, Babak Falsafi, and Anastasia Ailamaki. Toward dark silicon in servers. *IEEE Micro*, 31(EPFL-ARTICLE-168285):6–15, 2011.

[61] John Kelly III and Steve Hamm. *Smart Machines: IBM's Watson and the Era of Cognitive Computing*. Columbia University Press, 2013.

[62] Michael Kistler, Michael Perrone, and Fabrizio Petrini. Cell multiprocessor communication network: Built for speed. *IEEE micro*, 26(3):10–23, 2006.

[63] Stephen Cole Kleene. Representation of events in nerve nets and finite automata. Technical report, DTIC Document, 1951.

[64] Jonathan Koomey. Growth in data center electricity use 2005 to 2010. *A report by Analytical Press, completed at the request of The New York Times*, 2011.

[65] Raffi Krikorian. New tweets per second record, and how!, 2013.

[66] Rajasekar Krishnamurthy, Yunyao Li, Sriram Raghavan, Frederick Reiss, Shivakumar Vaithyanathan, and Huaiyu Zhu. SystemT: a system for declarative information extraction. *ACM SIGMOD Record*, 37(4):7–13, 2009.

[67] HT Kung and Philip L Lehman. Systolic (VLSI) arrays for relational database operations. In *Proceedings of the 1980 ACM SIGMOD international conference on Management of data*, pages 105–116. ACM, 1980.

[68] Cheng-Hung Lin, Sheng-Yu Tsai, Chen-Hsiung Liu, Shih-Chieh Chang, and J-M Shyu. Accelerating string matching using multi-threaded algorithm on gpu. In *Global Telecommunications Conference (GLOBECOM 2010), 2010 IEEE*, pages 1–5. IEEE, 2010.

[69] James Manyika, Michael Chui, Brad Brown, Jacques Bughin, Richard Dobbs, Charles Roxburgh, and Angela Hung Byers. Big data: The next frontier for innovation, competition, and productivity. Technical report, McKinsey Global Institute, May 2011.

[70] Clive Maxfield. *The design warrior's guide to FPGAs devices, tools and flows.* Newnes Elsevier, Boston, 2004.

[71] H. Y McCreary, M. A. Broyles, M.S. Floyd, A.J. Geissler, S. P. Hartman, F.L. Rawson, T.J. Rosedahl, J.C. Rubio, and M.S. Ware. EnergyScale for IBM POWER6 microprocessor-based systems. *IBM Journal of Research and Development*, 51(6):775–786, Nov 2007.

[72] Bingfeng Mei, Serge Vernalde, Diederik Verkest, Hugo De Man, and Rudy Lauwereins. Adres: An architecture with tightly coupled vliw processor and coarse-grained reconfigurable matrix. In *Field Programmable Logic and Application*, pages 61–70. Springer, 2003.

[73] Pingfan Meng, Matthew Jacobsen, Motoki Kimura, Vladimir Dergachev, Thomas Anantharaman, Michael Requa, and Ryan Kastner. Hardware accelerated novel optical de novo assembly for large-scale genomes. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[74] Geoffrey A Moore. *Crossing the Chasm: Marketing and Selling Technology Project*. HarperCollins e-Books, 2014.

[75] Rene Mueller and Jens Teubner. FPGA: what's in it for a database? In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of data*, pages 999–1004. ACM, 2009.

[76] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on wires: a query compiler for fpgas. *Proceedings of the VLDB Endowment*, 2(1):229–240, 2009.

[77] René Müller. *Data stream processing on embedded devices.* PhD thesis, ETH Zurich, 2010.

[78] Hiroki Nakahara, Tsutomu Sasao, and Munehiro Matsuura. A regular expression matching circuit based on a decomposed automaton. In *Reconfigurable Computing: Architectures, Tools and Applications*, pages 16–28. Springer, 2011.

[79] nVidia Corporation. Nvidia's next generation cuda compute architecture: Kepler gk110. *nVidia Whitepaper*, 2012.

[80] Matthias Pohl, Michael Schaeferling, and Gundolf Kiefer. An efficient FPGA-based hardware framework for natural feature extraction and related Computer Vision tasks. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–8. IEEE, 2014.

[81] Raphael Polig, Kubilay Atasu, Heiner Giefers, and Laura Chiticariu. Compiling text analytics queries to FPGAs. In *Field Programmable Logic and Applications (FPL), 2014 24th International Conference on*, pages 1–6. IEEE, 2014.

[82] Raphael Polig, Kubilay Atasu, and Christoph Hagleitner. Token-based dictionary pattern matching for text analytics. In *Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on*, pages 1–6. IEEE, 2013.

[83] Andrew Putnam, Adrian M Caulfield, Eric S Chung, Derek Chiou, Kypros Constantinides, John Demme, Hadi Esmaeilzadeh, Jeremy Fowers, Gopi Prashanth Gopal, Jan Gray, et al. A reconfigurable fabric for accelerating large-scale datacenter services. In *Computer Architecture (ISCA), 2014 ACM/IEEE 41st International Symposium on*, pages 13–24. IEEE, 2014.

[84] Michael O Rabin and Dana Scott. Finite automata and their decision problems. *IBM journal of research and development*, 3(2):114–125, 1959.

[85] IBM Press Release. 701 translator. http://www-03.ibm.com/ibm/history/exhibits/701/701_translator.html, 1954. Accessed: 2015-01-15.

[86] Jonathan Rohrer, Kubilay Atasu, Jan van Lunteren, and Christoph Hagleitner. Memory-efficient distribution of regular expressions for fast deep packet inspection. In *Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 147–154. ACM, 2009.

[87] IBM Power Systems S814, S824 Technical Overview, and Introduction. First edition, 2014.

[88] Daniele Paolo Scarpazza, Oreste Villa, and Fabrizio Petrini. High-speed string searching against large dictionaries on the Cell/BE processor. In *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*, pages 1–12. IEEE, 2008.

[89] Reetinder Sidhu and Viktor K Prasanna. Fast regular expression matching using FPGAs. In *Field-Programmable Custom Computing Machines, 2001. FCCM'01. The 9th Annual IEEE Symposium on*, pages 227–238. IEEE, 2001.

[90] Deshanand Singh. Implementing fpga design with the opencl standard. *Altera whitepaper*, 2011.

[91] Hartej Singh, Ming-Hau Lee, Guangming Lu, Fadi J Kurdahi, Nader Bagherzadeh, and Eliseu M Chaves Filho. Morphosys: an integrated reconfigurable system for data-parallel and computation-intensive applications. *Computers, IEEE Transactions on*, 49(5):465–481, 2000.

[92] B Sinharoy, JA Van Norstrand, RJ Eickemeyer, HQ Le, J Leenstra, DQ Nguyen, B Konigsburg, K Ward, MD Brown, JE Moreira, et al. Ibm power8 processor core microarchitecture. *IBM Journal of Research and Development*, 59(1):2–1, 2015.

[93] Tian Song, Wei Zhang, Dongsheng Wang, and Yibo Xue. A memory efficient multiple pattern matching architecture for network security. In *INFOCOM 2008. The 27th Conference on Computer Communications. IEEE*. IEEE, 2008.

[94] Ioannis Sourdis and Dionisios Pnevmatikatos. Fast, large-scale string match for a 10 Gbps FPGA-based NIDS. In *New Algorithms, Architectures and Applications for Reconfigurable Computing*, pages 195–207. Springer, 2005.

[95] Greg Stitt and James Coole. Intermediate fabrics: Virtual architectures for near-instant FPGA compilation. *Embedded Systems Letters, IEEE*, 3(3):81–84, 2011.

[96] Ken Thompson. Programming techniques: Regular expression search algorithm. *Communications of the ACM*, 11(6):419–422, 1968.

[97] Vernon Turner, David Reinsel, John F. Gantz, and Stephen Minton. The digital universe of opportunities: Rich data and the increasing value of the internet of things. Technical report, IDC, April 2014.

[98] Graham Upton and Ian Cook. *A Dictionary of Statistics 3e*. Oxford university press, 2014.

[99] Jan Van Lunteren et al. High-performance pattern-matching for intrusion detection. In *Infocom*, volume 6, pages 1–13. Citeseer, 2006.

[100] Jan van Lunteren and Alexis Guanella. Hardware-accelerated regular expression matching at multiple tens of gb/s. In *INFOCOM, 2012 Proceedings IEEE*, pages 1737–1745. IEEE, 2012.

[101] Giorgos Vasiliadis, Michalis Polychronakis, Spiros Antonatos, Evangelos P Markatos, and Sotiris Ioannidis. Regular expression matching on graphics hardware for intrusion detection. In *Recent Advances in Intrusion Detection*, pages 265–283. Springer, 2009.

[102] Grady Ward. Moby Word Lists. http://www.gutenberg.org/ebooks/3201. Accessed: 2015-03-3.

[103] Gerhard Weikum. From text to entities and from entities to insight: A perspective on unstructured big data. Presented at Microsoft Research Big Data Analytics Workshop 2013, Cambridge, UK, 2013.

[104] Joseph Weizenbaum. Eliza a computer program for the study of natural language communication between man and machine. *Communications of the ACM*, 9(1):36–45, 1966.

[105] Bruce Wile. Coherent Accelerator Processor Interface (CAPI) for POWER8 Systems. IBM White Paper, Sep 2014.

[106] Wayne Wolf. The future of multiprocessor systems-on-chips. In *Design Automation Conference, 2004. Proceedings. 41st*, pages 681–685. IEEE, 2004.

[107] Pascal T Wolkotte, Gerard JM Smit, Gerard K Rauwerda, and Lodewijk T Smit. An energy-efficient reconfigurable circuit-switched network-on-chip. In *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*, pages 155a–155a. IEEE, 2005.

[108] Haicheng Wu, Gregory Diamos, Srihari Cadambi, and Sudhakar Yalamanchili. Kernel weaver: Automatically fusing database primitives for efficient gpu computation. In *Proceedings of the 2012 45th Annual IEEE/ACM International Symposium on Microarchitecture*, pages 107–118. IEEE Computer Society, 2012.

[109] Lisa Wu, Raymond J Barker, Martha A Kim, and Kenneth A Ross. Navigating big data with high-throughput, energy-efficient data partitioning. *ACM SIGARCH Computer Architecture News*, 41(3):249–260, 2013.

[110] YE Yang, Hoang Le, and Viktor K Prasanna. High performance dictionary-based string matching for deep packet inspection. In *INFOCOM, 2010 Proceedings IEEE*, pages 1–5. IEEE, 2010.

[111] YE Yang and Viktor K Prasanna. Memory-efficient pipelined architecture for large-scale string matching. In *Field Programmable Custom Computing Machines, 2009. FCCM'09. 17th IEEE Symposium on*, pages 104–111. IEEE, 2009.

[112] Fang Yu, Randy H Katz, and Tirunellai V Lakshman. Gigabit rate packet pattern-matching using tcam. In *Network Protocols, 2004. ICNP 2004. Proceedings of the 12th IEEE International Conference on*, pages 174–183. IEEE, 2004.

[113] Qiuling Zhu, Bilal Akin, H Ekin Sumbul, Fazle Sadi, James C Hoe, Larry Pileggi, and Franz Franchetti. A 3d-stacked logic-in-memory accelerator for application-specific data intensive computing. In *3D Systems Integration Conference (3DIC), 2013 IEEE International*, pages 1–7. IEEE, 2013.