From the Institute of Information Systems
of the University of Lübeck
Director: Univ.-Prof. Dr. rer. nat. habil. Ralf Möller

Hybrid Architecture for Hardware-accelerated Query Processing in Semantic Web Databases based on Runtime Reconfigurable FPGAs

Dissertation for Fulfillment of Requirements for the Doctoral Degree
of the University of Lübeck
— from the Department of Computer Sciences —
Submitted by Stefan Werner from Karl-Marx-Stadt, now Chemnitz.

Lübeck, October 2016

First referee:	PD Dr. rer. nat. habil. Sven Groppe
Second referee:	Prof. Dr. rer. nat. Stefan Fischer
Date of oral examination:	February 6th, 2017

Approved for printing. Lübeck, February 16th, 2017

Abstract

Nowadays it is a fact that more and more (unstructured) data is generated, stored and analyzed in several areas of science and industry – motivated by political and security reasons (surveillance, intelligence agencies), economical (advertisement, social media) or medical matters. Besides a flood of machine-generated data due to technological advances, e.g., in ubiquitous internet-of-thing products and increasing accuracy of sensors, a large amount of data is produced by humans in various forms. In order to enable machines to automatically analyze (possibly not well or completely defined) data the idea of the *Semantic Web* was created. Besides suitable data structures, optimized hardware is necessary to store and process this vast amount of data. Whereas persistently storing these massive data is hassle-free, the processing and analysis within a reasonable time frame becomes more and more difficult. In order to cope with these problems, in the last decades intensive work was done to optimize database software and data structures. Furthermore, technological advances enabled shrinking feature size to increase clock frequency and thus the overall performance. However, nowadays these approaches are reaching their limits (power wall) and in the last years the trend evolved to multi/many-core systems in order to increase performance. Additionally, these systems are not assembled with homogeneous cores, but rather are composed by heterogeneous and specialized cores which compute a specific task efficiently. The main issue of such systems is that these specialized cores can not be used in applications showing a huge variety in processing. Widely available *Field Programmable Gate Arrays* (FPGAs) with the capability of (partial) runtime reconfiguration are able to close the gap between the flexibility of general-purpose CPUs and the performance of specialized hardware accelerators.

In this work, a fully integrated hardware-accelerated query engine for large-scale datasets in the context of Semantic Web databases is presented. The proposed architecture allows the user to retrieve specific information from Semantic Web datasets by writing a query which is automatically adapted and executed on a runtime reconfigurable FPGA. As queries are typically unknown at design time a static approach is not feasible and not flexible to cover a wide range of queries at system runtime. Our proposed approach dynamically generates an optimized hardware accelerator in terms of an FPGA configuration for each individual query and transparently retrieves the query result to be displayed to the user. The benefits and limitations are evaluated on large-scale synthetic datasets with up to 260 million records as well as the widely known *Billion Triples Challenge* with over one billion records.

Kurzfassung

Es ist bekannt, dass in der heutigen Zeit immer mehr (unstrukturierte) Daten in verschiedensten Bereichen der Wissenschaft und Wirtschaft erzeugt, gespeichert und analysiert werden – (sicherheits-) politisch motiviert (Überwachung, Sicherheitsdienste), getrieben durch wirtschaftliche Interessen (Werbung, soziale Netzwerke) oder aus medizinischem Nutzen. Neben der Flut an Maschinen-generierten Daten verursacht durch technologische Fortschritte, z.B. allgegenwärtige Geräte des Internets der Dinge und höhere Genauigkeit von Sensoren, wird ein Großteil der Informationen in zahlreichen verschiedenen Formaten durch den Menschen produziert. Damit auch Maschinen diese möglicherweise unvollständigen Daten verarbeiten können, entstand die Idee des *Semantic Web*. Neben angepassten Datenstrukturen werden jedoch auch optimierte Rechensysteme benötigt, um die riesigen Datenmengen speichern und verarbeiten zu können. Während das Speichern dieser Datenmengen keine Herausforderung darstellt, so wird das Verarbeiten und Analysieren in akzeptablen Zeitfenstern mehr und mehr problematisch. Um diesen Anforderungen gerecht zu werden, wurde in den letzten Jahrzehnten intensiv an der Optimierung von Datenbanksystemen und den darunterliegenden Datenstrukturen geforscht. Des Weiteren erlaubten technologische Fortschritte die stetige Verkleinerung von elektronischen Schaltkreisen und damit die Erhöhung der Rechenfrequenz und -leistung. Jedoch stießen diese Methoden an ihre physikalischen Grenzen und trieben die Einführung von Mehrkernprozessoren voran, um dennoch Leistungsverbesserung erzielen zu können. Zudem setzen sich diese Systeme häufig nicht mehr nur aus gleichartigen Rechenkernen zusammen. Stattdessen basieren diese auf verschiedensten Spezialprozessoren, um damit bestimmte Aufgaben mit sehr hoher Effizienz lösen zu können. Jedoch lassen sich Spezialprozessoren typischerweise nur in eingeschränkten Aufgabenfeldern einsetzen und sind damit nicht für Anwendungen mit hohen Flexibilitätsanforderungen sinnvoll einsetzbar. Die wachsende Verfügbarkeit und Beliebtheit von *Field Programmable Gate Arrays* (FPGAs), mit der Möglichkeit der partiellen Rekonfiguration zur Laufzeit, stellen einen interessanten Lösungsansatz dar um die Flexibilität von typischen Prozessoren mit der Leistungsfähigkeit von Spezialprozessoren zu vereinen.

In dieser Arbeit wird ein vollständig integrierter Anfragebeschleuniger im Kontext von Semantic Web Datenbanken vorgestellt. Die entwickelte Architektur erlaubt es dem Benutzer eine Anfrage an Datensätze zu stellen, wobei die gegebene Anfrage zur Laufzeit auf einen FPGA abgebildet und anschließend beschleunigt ausgeführt wird. Typischerweise sind die Anfragen vorab nicht bekannt, so dass ein statischer Ansatz nicht flexibel genug ist, um ein breites Spektrum an Anfragen sinnvoll abbilden zu können. Daher erzeugt der hiermit vorgestellte Ansatz dynamisch zur Systemlaufzeit einen anfragespezifischen Beschleuniger und ermittelt transparent

für den Benutzer das Ergebnis der Anfrage. Im Rahmen dieser Arbeit werden die Vorteile und Einschränkungen, anhand von synthetischen Datensätzen mit bis zu 260 Millionen Einträgen sowie realen Datensätzen der *Billion Triples Challenge* mit mehr als einer Milliarde Einträgen, aufgezeigt.

Acknowledgments

During my PhD studies I have met many people which supported me and influenced my work. First of all, I would like to thank Prof. Volker Linnemann for giving me the opportunity to enter scientific research at the Institute for Information Systems and providing support after his leave. In addition, I would like to thank Prof. Ralf Möller for supporting me and giving me the freedom to finish this thesis.

All these years my advisor Dr. Sven Groppe has been motivating me to question my results and further improve them. I am certainly grateful for his help to finish my thesis. Additionally, I would like to thank Prof. Thilo Pionteck for introducing me into to the world of reconfigurable computing and collaborating in several publications. I thank the exam committee Prof. Stefan Fischer and Prof. Erik Maehle for reviewing my work and chairing my exam.

Furthermore, I would like to thank Dennis Heinrich for working together and exchanging experience in the field of FPGA development; Nils Fußgänger for maintaining the technical infrastructure and sharing knowledge in several technical topics. A special thanks goes to Angela König for selflessly supporting me in basically any situation.

I deeply thank my mother Gisela, my father Dietmar and my brother Frank for bringing me up without worries, fulfilling all my wishes but still taught me to be humble and providing me all freedom to make my own choices – always knowing to have their support.

Lastly, I greatly thank my fiancée Irina for mentally strengthening me everyday, keeping doubts out of my head and showing the pleasures of life not only during the period of writing this thesis.

Lübeck, February 2017 Stefan Werner

Contents

1 Introduction

1.1 Motivation

While economies at the beginning of the 20$^{\text{th}}$ century were characterized by industrialization, nowadays more and more business models rely on information exchange and analysis. The creation of the *World Wide Web* (WWW) – in which users became not only consumers, but also content producers – can be seen as the initial spark of this progression.

Furthermore, devices get smaller and cheaper which in turn increases the user basis. Nowadays there are more than 1 billion websites accessible [1]. Finding valuable information in this vast amount of data can not be manually done by humans. Therefore, search engines have been developed to cope with the information flood. The phrase *Just google it* has become ordinary. Amit Singhal[1] reported in October 2015 more than 100 billion searches a month [2]. Basically, these engines compare given keywords with plain text from a website. But the content is designed for humans to read, rather than for further processing by machines. Understanding the semantics of information provided by a website is rarely possible in an automated manner and thus results might consist of unrelated information. The *Semantic Web* (SW) intends to extend the current web providing a well-defined meaning for any kind of information by enriching existing data with metadata and semantically linking data and data sources. There is a growing number of applications that benefit from SW technologies and the datasets are becoming increasingly large (millions to billions [3]).

Additionally, not only humans are producing and consuming data. Any kind of sensors or even sensor networks are deployed to monitor various indicators with increasing precision. A fast query execution on this vast amount of data is essential

[1] Senior Vice President, Google Fellow, Google Inc., responsible for Google Search

to develop high performance SW applications. The increasing amount and complexity of data demand for higher performance of the underlying algorithms and hardware to analyze and query data in a reasonable time frame. Database systems are typically optimized for traditional hardware systems which are based on the commonly used *von-Neumann* architecture [4]. As a consequence, general-purpose *Central Processing Units* (CPUs) suffer from architectural limitations (e.g., *memory wall*). Additionally, technological limits do not allow continuous performance improvements by simply increasing the clock rate due to the *power wall*. Thus, re-thinking the universal design using general-purpose CPUs is required. Rather than using one general processor, multi- and many-core architecture have been developed. These can be homogenous architectures but more and more architectures base on heterogeneous designs. Typically, besides general-purpose CPUs, these consist of hardware accelerators which have been developed to efficiently fulfill a specific application task. Well-known examples are network and graphics processors. However, usually they can not be utilized in application domains showing a huge variety in processing such as query processing in SW databases. With the aid of *Field-Programmable Gate Arrays* (FPGAs) the gap between the flexibility of general-purpose CPUs and the performance of specialized hardware can be closed.

1.2 Scientific Contribution and Organization of this Work

This work investigates the utilization of runtime reconfigurable hardware architectures in the scope of query processing on Semantic Web data. As a result a dynamically reconfigurable hybrid query engine based on *Field-Programmable Gate Arrays* (FPGAs) is provided. The hybrid architecture is shown in Figure 1.1. It allows the user to retrieve specific information from Semantic Web datasets by writing a query which is automatically transformed into an FPGA configuration and executed on the runtime reconfigurable FPGA. Each query operator is a specialized processing core to process a particular intermediate result as fast as possible. The composition of several specialized processing cores enables the hardware-accelerated query evaluation. The major results of this thesis have been published in peer-reviewed conference proceedings and journal papers [5, 6, 7, 8, 9].

The main contributions of this work are:

- We introduce an operator template which enables the transparent communication of FPGA-based query operators.

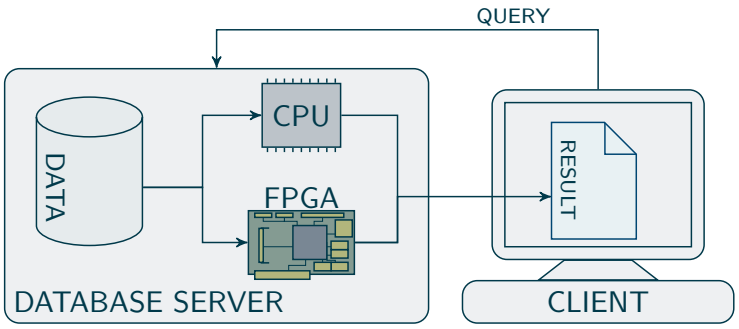


Figure 1.1 – Hybrid architecture - The client application sends a query to the hybrid database server which transparently determines the result using a hardware accelerator based on a *Field-Programmable Gate Array* (FPGA).

- We present and evaluate the FPGA-based implementations of various query operators, each realizing a dedicated hardware processing core.
- We present the first fully-integrated hybrid hardware-software system which utilizes a runtime reconfigurable FPGA to provide a query-specific hardware accelerator operating on large-scale Semantic Web datasets over one billion records. Compared to a highly optimized CPU-based software solution our hybrid query engine executes queries up to 32 times faster.
- We extend our architecture by the new concept of *Semi-static Operator Graphs* (SOGs) which deploy a general query structure on a runtime reconfigurable FPGA. Instead of exchanging the whole query structure this approach enables the exchange of single query operators in terms of *Reconfigurable Modules* (RMs). As a result the reconfiguration overhead can be significantly reduced.

Summarizing all, in this work we develop a hybrid database system which is able to automatically map and execute queries on a query-specific hardware accelerator.

In Chapter 2 the fundamentals and technical background of this work are provided. The motivation of the Semantic Web – one driving force of the data growth – is given, followed by a comprehensive overview of Semantic Web data formats and standards. On the example of our Semantic Web database LUPOSDATE we describe the typical query representation and query processing stages in software systems. As the technological basis of this work, we introduce the methodology of reconfigurable computing, specifically based on FPGAs. Besides the internal FPGA architecture, we present the typical development flow of FPGA-based hardware design. Concluding this chapter, we review recent work using modern hardware architectures in the field of database tasks.

In Chapter 3, we motivate the use of FPGAs in databases by providing a top-down view and outline potential benefits in query execution. Our final approach aims to automatically transform the internal query representation of our database system into a query-specific hardware accelerator by assembling dedicated processing cores. Each processing core executes one operator of the given query. Therefore, we introduce an operator template which defines a common interface for all operators. The common interface allows the transparent data flow between successive operators without needing the knowledge about their internal function. On the example of the join and filter operators, first performance insights are given. Furthermore, all supported operators of this work are introduced in this chapter.

With a set of operators, we show in Chapter 4 the automated composition of multiple operators to represent arbitrary queries. Therefore, the integration into the software system and the architecture of the hybrid query engine is described. The resources on the FPGA are divided into static components such as communication interfaces and one large *Reconfigurable Partition* (RP) which allows the exchange of the complete query structure. The benefits and limitations are evaluated on large-scale synthetic datasets with up to 260 million triples as well as a real-world dataset with over one billion triples.

In Chapter 5, we further improve our architecture by deploying a semi-static query structure on the FPGA. It consists of multiple small RPs rather than one large RP which allows the exchange of single query operators. This greatly reduces the overhead from minutes to milliseconds for exchanging queries on the FPGA during system runtime.

We recap this work and summarize the resulting benefits of our approach in Chapter 6. Subsequently, we outline numerous extensions and provide a broad range of ideas for future research based on this work.

2 Background

In this chapter we introduce the core technologies of this work. First the Semantic Web and its characteristics are described which further motivates the need of reconfigurable hardware platforms. Latter is presented in detail and discussed based on Field-Programmable Gate Arrays. In the remainder of this chapter, a comprehensive overview of recent research on modern hardware architectures in the field of database tasks is given.

2.1 Semantic Web

This section gives an introduction to the Semantic Web and its core technologies. Furthermore, we present the Semantic Web database LUPOSDATE which is extensively used as a software basis of this work.

2.1.1 Introduction

With the evolution of the *World Wide Web* (WWW) and its related increasing number of services the amount and availability of information has grown enormously. While in April 2006 more than 80 million websites have been reported [10], only ten years later in April 2016 this number marks more than one billion websites [11] (see Figure 2.1). But the content is typically designed for humans to read rather than for further processing by machines. The *web* characteristic comes from *dumb* links connecting one resource to another one without expressing their relation to each other [12, p. 29]. Additionally, understanding the semantics of information provided by a website is rarely possible in an automated manner.

On the other side, managing and finding valuable information in all these data was not achievable without algorithms implemented by the many search engines we use every day. Basically, users type some keywords related to some topic and

the search engine returns a list of websites by textually comparing one or more of the keywords with the website's content. If the keyword is ambiguous the list will contain unwanted results. Contrary, if the meaning of a keyword can be expressed using other words (synonyms), the list of results might be incomplete. Besides these problems, further visions of software agents taking advantage by bringing structure to meaningful websites, motivate the idea of the *Semantic Web* (SW) to establish a machine-readable web by adding methods for systematically analyzing data and inferring new information.

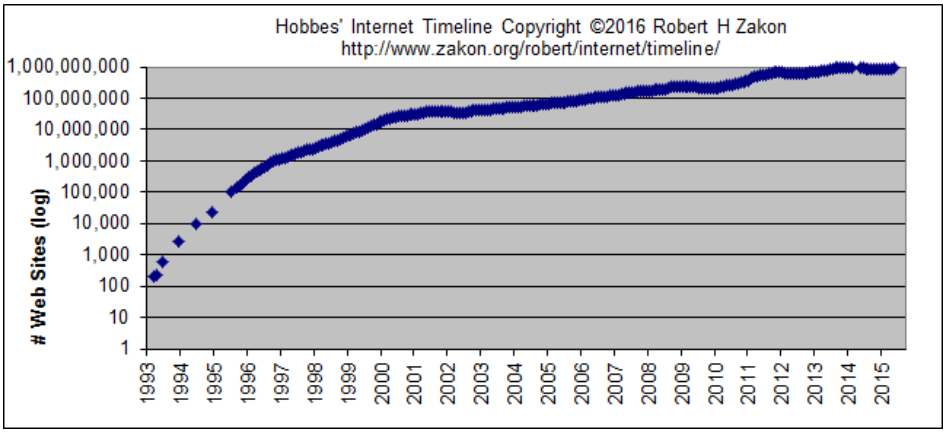


Figure 2.1 – Growth of the World Wide Web (*site* refers to the number of web servers, whereby one host may have multiple sites by using different domains or port numbers [1]).

Take the author's web page[1] shown in Figure 2.2 as a simple example. As a human it is easy to interpret the containing information and combine them to a comprehensive knowledge. The reader finds a name (simply inferred from the layout of the page and because it *looks* like a name) and a picture, and concludes that picture and name refer to the same person. Furthermore, the page shows an address, but a human would not consider it to be the home address of the author because it contains terms like *institute* and *Universität*, and *obviously* it is not a private website. In order to automatically retrieve these (for humans *obvious*) information by agents, powerful and complex algorithms are necessary. Due to the variety of websites in the WWW these algorithms are still error-prone because mostly they rely on a predefined and regular structure. In order to address these issues the SW does not intend to be an alternative web but to be an *extension of the current one, in which information is given well-defined meaning, better enabling computers and people to work in cooperation* [13]. Although some SW

[1] `http://www.ifis.uni-luebeck.de/index.php?id=werner&L=1`

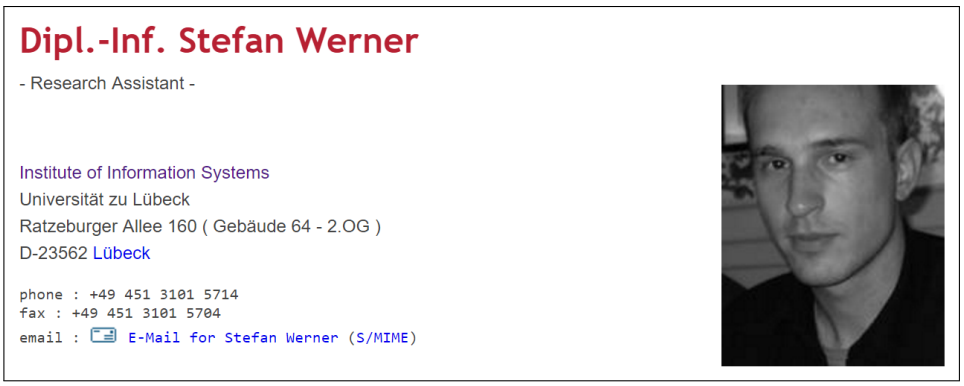


Figure 2.2 – Excerpt of the author's web page.

technologies benefit from *Artificial Intelligence* (AI) research it is not deemed as AI. Instead of *smarter machines* it establishes *smarter data* by describing resources and relations/links in a standard machine-readable format [12, p. 22].

Regarding the simple example, the website can be *semantically* enriched for machines by embedding additional information and relations between them. These can be of simple forms such as annotations expressing *Universität zu Lübeck is of type "university"* or more complex forms such as *name and picture belong to the same person*. Of course, these terms and relations have to be well defined and commonly used. The vastness of the WWW and its distributed growth requires the SW to provide methods to handle semantically duplicated and vague terms. Furthermore, new data sources might occur and others might disappear frequently, and thus information might be incomplete and inconsistent [14]. In order to cope with these challenges, the SW defines and combines several basic technologies.

2.1.2 Semantic Web Technologies

Although the Semantic Web is not directly recognizable to the end-user from outside, more and more SW technologies exert influence on web applications and their algorithms. In order to strengthen this trend, the *World Wide Web Consortium* (W3C) makes huge efforts to provide standardized languages and technologies to get closer to the full vision. As a result, several standards have been released and other proposals are currently work in progress. Figure 2.3 outlines the SW stack. The bottom layers provide the basis of the SW consisting of *Universal Character Set* (UCS) [15], *Internationalized Resource Identifiers* (IRIs) [16] and *Extensible Markup Language* (XML) [17]. The UCS specifies different encoding schemes for written form of all languages of the world and additional symbols.

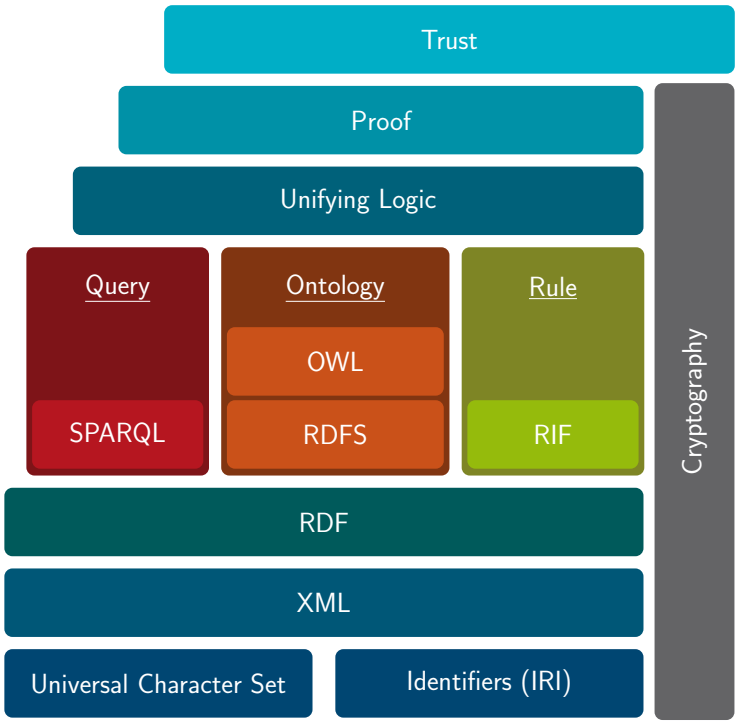


Figure 2.3 – Semantic Web stack (adapted from [12, p. 23]).

IRIs are basically an extension of *Universal Resource Identifiers* (URIs) but allow the use of almost all the UCS characters. An IRI consists of an string of characters to identify an abstract or physical resource. XML defines a set of rules for encoding human- and machine-readable documents. The *Resource Description Framework* (RDF) [18] allows to express and interconnect statements in form of triples which are used as the basic data format. The *SPARQL Protocol And RDF Query Language* (SPARQL) [19] provides a query language to retrieve information from RDF datasets. Taxonomies and ontologies can be expressed using *RDF Schema* (RDFS) [20] respectively *Web Ontology Language* (OWL) [21]. Latter allows to express complex properties and relations between resources. The *Rule Interchange Format* (RIF) [22] enables the exchange of different rule languages between existing rule systems. These frameworks and technologies will be further refined in the following sections. However, the remaining layers consisting of *unifying logic*, *proof* and *trust* as well as *cryptography*, that spans all layers, are not yet standardized.

2.1.2.1 Resource Description Framework

The *Resource Description Framework* (RDF) is an abstract framework for representing information in the web [18]. It is used as the basic format in the SW to express statements about web resources. In general, RDF datasets consist of *triples*. Each triple consists of the components *subject*, *predicate* (or *property*) and *object* (or *value*), and is formally defined as $(s, p, o) \in (I \cup B) \times I \times (I \cup B \cup L)$, with the pairwise disjoint infinite sets I, B and L, where I is a set of *IRIs*, B is the set of *blank nodes* and L contains *literals*. Each IRI or literal *denotes* something in the world and is called a *resource* [23]. However, they do not necessarily point to an actual existing resource such as a file or web address but can be abstract concepts or physical things. A literal consists of a lexical form in terms of a Unicode string and a data type annotation defining the range of possible values (using a data type IRI). For plain-text strings in a natural language a language tag must be appended which implies the data type *language-tagged string*[2]. Simple literals without any data type or language tag are interpreted as strings[3]. Blank nodes are unnamed identities without an identifier, and thus do not identify specific resources. They can represent complex attributes or multi-component structures [23].

Listing 2.1 shows an example RDF dataset consisting of five prefix definitions (lines 1 to 5) and nine triples (line 6 to 14) using the *Notation 3* (N3) [24]. Prefixes are declared for simplicity and readability. For instance, the predicate rdfs:subClassOf is expanded to <http://www.w3.org/2000/01/rdf-schema#subClassOf> due to the previous prefix definition of rdfs. The example consists of five distinct subjects. Some subjects are are used in multiple triples (e.g., l:Journal1). Other subjects are used in other triples as object (e.g., v:Article).

Besides N3, several other serialization formats for RDF data are available such as N-Triples [25], Turtle [26], and RDF/XML [27]. The specification *Resource Description Framework in Attributes* (RDFa) [28] describes how to embed structured data in any markup language such as the *Hypertext Markup Language* (HTML) [29].

Conceptually, a set of RDF triples can be interpreted as an RDF graph, in which the subjects respectively objects with the same IRI are represented as unique nodes and the predicates serve as labeled directed edges. Figure 2.4 shows the resulting graph regarding the example dataset in Listing 2.1. Throughout this work RDF is used as the basis input data format. The query language to access RDF data is introduced in the next section.

[2] IRI: `http://www.w3.org/1999/02/22-rdf-syntax-ns#langString`
[3] IRI: `http://www.w3.org/2001/XMLSchema#string`

Listing 2.1 – Example RDF dataset.

```
@prefix rdf:   <http://www.w3.org/1999/02/22-rdf-syntax-ns#> .
@prefix rdfs:  <http://www.w3.org/2000/01/rdf-schema#> .
@prefix swrc:  <http://swrc.ontoware.org/ontology#> .
@prefix v:  <http://localhost/vocabulary/> .
@prefix l:  <http://localhost/instances/> .
v:Journal rdfs:subClassOf v:Document .
v:Article rdfs:subClassOf v:Document .
l:Journal1 rdf:type v:Journal .
l:Journal1 v:title "Popular Journal"^^xsd:string .
l:Article1 rdf:type v:Article .
l:Article1 v:abstract "English abstract"@en .
l:Article1 v:publishedIn l:Journal1 .
l:Article1 v:creator l:Famous_Author .
l:Famous_Author v:name "Stephen Hawking"^^xsd:string .
```

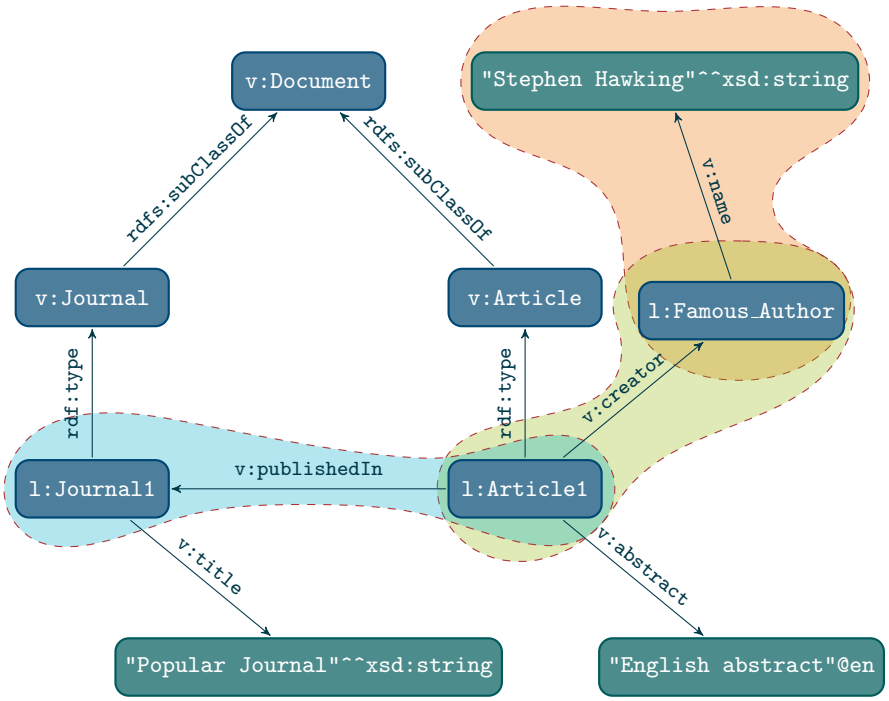


Figure 2.4 – RDF graph corresponding to Listing 2.1.

Listing 2.2 – SPARQL example query.

```
1 PREFIX v: <http://localhost/vocabulary/>
2 PREFIX l: <http://localhost/instances/>
3 SELECT ?aName WHERE {
4   ?author v:name         ?aName  .
5   ?doc    v:creator      ?author .
6   ?doc    v:publishedIn  l:Journal1
7 }
```

2.1.2.2 SPARQL Protocol And RDF Query Language

After standardized by the W3C, the *SPARQL Protocol And RDF Query Language* (SPARQL) in version 1.0 [30] became the most important query language for the SW. It enables access to the data by selecting specific parts of the possibly distributed RDF graph. Listing 2.2 shows an example query which requests all names of authors which published in the journal referenced by the IRI <http://localhost/instances/Journal1>.

Variables in a query are indicated by ? or $. The SELECT clause defines a projection list of variables to appear in the final result (i.e., the bound values of variable ?aName). Similar to RDF triples, the core component of SPARQL is a set of triple patterns (s,p,o). These are defined in the WHERE clause and constrain the input RDF data by specifying known RDF terms and replacing unknown terms by variables. Each triple pattern matches a subset of the RDF data when the terms are equal to the triple in the RDF data and bounds specific values to the variables of the pattern. The appearance of the same variable in several patterns requires to combine those intermediate results, which have the same values in common variables. This operation is known as (*natural*) *join* and frequently appears in SPARQL queries due to the linking structure of RDF. Regarding Listing 2.2 and the dataset shown in Listing 2.1 the result is determined as follows. The first triple pattern (line 4) matches one triple (surrounded orange in Figure 2.4) and returns bindings for the variables. The second triple pattern (line 5) matches one triple as well (green in Figure 2.4). As the first and the second pattern share the common variable ?author in the subject respectively object, a join between both intermediate results is implied. Additionally, the third triple pattern (line 6) shares the variable ?doc with the second pattern, which implies another join (blue in Figure 2.4). Besides the basic structure, consisting of SELECT-WHERE clause and triple patterns, SPARQL 1.0 and its recent version 1.1 [19] provide numerous other features. Besides SELECT other query forms using the keywords ASK, DESCRIBE

or CONSTRUCT are supported. ASK query returns whether a query pattern has a solution (*true*) or not (*false*). DESCRIBE queries return an RDF graph containing data about the resources. CONSTRUCT queries allows to construct and return a new RDF graph instead of only bound values for variables. Named graphs allow to use multiple RDF datasets (referred using an IRI) within one query. Besides triple patterns, the WHERE clause can consist of other expressions to further refine the intended result. The FILTER expression restricts the solution according to a given constraint. Each solution which evaluates to *false* (or an error) regarding the constraint is eliminated. The constraint can be of various logical conjunctive and disjunctive forms consisting of built-in functions (such as isIRI, bound, etc.), comparisons of typed literals and extensional functions named by an IRI. The filter expression (NOT) EXISTS tests whether a triple pattern matches the dataset or not. MINUS allows to remove solutions which fit its arguments. The UNION enables to specify pattern alternatives such that one of several graph patterns may match. One characteristic in the SW is that regularity and completeness can not be assumed in all RDF graphs. The OPTIONAL keyword enables to express patterns which add information to the results if there are any, but does not reject the results because some patterns do not match. The keywords DISTINCT and REDUCED determine whether duplicates are included in the result. The former strictly eliminates duplicates, the latter only permits them to be eliminated. The clause ORDER BY establishes an ascending or descending order of the solution regarding given variables. LIMIT defines an upper bound on the number of results returned. OFFSET controls where the solution starts from in the complete result. The combination of both is useful to select subsets of the result. Property paths enable to find a connection between graph nodes consisting of one or more predicates (properties). Federated queries are supported through the SERVICE keyword [31]. It allows to explicitly delegate certain sub-queries to another SPARQL endpoint. Besides querying data, SPARQL provides language constructs to update, create and remove RDF graphs [32]. The full grammar and reference of SPARQL 1.1 and its features can be found in [32, 33]. Throughout this work SELECT queries according SPARQL are mapped to an hardware accelerator in order to improve query performance. This enables a standardized access to the benefits of this work without learning a new language or new language constructs. The supported constructs are explained later in Chapter 3. A detailed description on how database systems evaluate queries is given later in Section 2.1.3.

2.1.2.3 Taxonomies and Ontologies

Taxonomies and ontologies (also called vocabularies) are used to improve knowledge about data by classifying terms and defining relationships between them ac-

cording to a specific domain. Additionally, relationships can be implicitly *inferred* based on the explicitly given relationships. In order to allow applications/machines to verify knowledge and reason knowledge (make implicit knowledge explicit) to finally *understand meaning*, the specific domain knowledge has to be formally expressed. Therefore, the W3C proposed the following different ontology languages. *RDF Schema* (RDFS) is a semantic extension of the RDF vocabulary. It provides an extensible type system for RDF by defining specific RDF terms [20]. The expressiveness covers definitions of classes (also called groups) such that resources (instances) can be assigned to classes which in turn are resources as well. Furthermore, classes can be described using RDF properties such as types or subclasses. The *Web Ontology Language* (OWL) [34] defines an additional vocabulary and formal semantics grouped together in one namespace owl. It provides three different fragments (*Lite*, *DL*, *Full*). The purpose of OWL *Lite* aims for the definition of taxonomies with simple constraints and restricts cardinalities to 0 and 1 – mainly intended to simplify tool implementation. OWL *DL* (Description Logic) maximizes the expressiveness by adding features like union or arbitrary cardinality, but still restricts for instance properties in order to preserve decidability (computable in finite time) and completeness. OWL *Full* enables maximum expressiveness without computational guarantees.

However, new requirements have been identified during OWL deployments which consequently resulted in the new revision OWL 2 [21]. It is backward compatible and adds new functionalities to increase expressiveness but also provides different subsets called *profiles* (*EL*, *QL*, *RL* [35]) which restrict modeling features to address specific use cases and simplify reasoning. A lower complexity allows conceptually simpler algorithms which in turn are easier to implement resulting in more and better tools [36]. OWL *EL*[4] is mainly designed for applications using ontologies consisting of a very large number of classes and properties, and allows classification and instance queries in polynomial time. Applications which mostly query data by reasoning on a large number of instance data should use OWL *QL* (Query Language). It covers the expressiveness of relatively lightweight ontologies and most of the expressiveness of *entity-relationship* and *UML* schemas [37]. Therefore, it is suitable as an ontological data access layer such that the unchanged data in a relational database can be queried through an ontology by rewriting a given query into an SQL query. OWL *RL* (Rule Language) provides scalable reasoning but retains a high level of expressiveness.

In practice there are several syntaxes available to describe and exchange OWL 2 ontologies, such as *RDF/XML*, *OWL/XML*, *Functional Syntax*, *Manchester Syntax* and *Turtle* [21].

[4] bases on EL family of description logics that provide only existential quantification [35]

A simple use case for using ontologies is the integration of data from different sources as different identifiers (*author*, *creator*, …) for the same concept may cause troubles combining these datasets. Adding an ontology which expresses the relations between terms (*author is the same as creator and vice versa*) can solve this problem easily and most important automatically. Among other things, ontologies are essential to provide agents with the capability to understand diverse information and enrich them to higher knowledge [38]. Ontologies realize the paradigm of the *Open World Assumption* which assumes everything not expressed in the database to be *unknown*, instead of considering it to be false [14, p. 16]. This is beneficial especially if we intentionally want to underspecify or at the moment we are not able to fully specify a domain. Ontologies do not have to be complex and already simple ontologies can have an impact, e.g., expressing *same as* relations regarding synonyms or RDF predicates (a:name,foaf:name) can improve search results. However, some ontologies cover a specific knowledge domain and are very complex. Thus, they are usually developed by domain experts [39, p. 42] but can be beneficially used in various applications.

Some popular ontologies are (without any claim to completeness): *Dublin Core* (dc) for describing generic metadata [40]; *Friend Of A Friend* (foaf) for describing social relationships between people focused on the virtual world [41]; *DOAP* for describing software projects [42]; *eClassOwl* (eBusiness ontology) for products and services [43]; *Music Ontology* to describe meta data related to the music industry [44]; *Open Biomedical Ontologies* including *Gene Ontology* and others such as *UniProt* [45, p. 307]. Various ontologies for different domains are collected and provided by the *Linked Open Vocabularies* (LOV) initiative[5].

This work does not provide support for hardware-accelerated ontology evaluation and thus is not further examined.

2.1.2.4 Rule Interchange Format (RIF)

The *Rule Interchange Format* (RIF)[22] was originally designed as a common standard for rules interchange in the SW between different existing rule systems. However, now it is seen as a complementation for ontologies as RIF rules can be used to express ontology inferences and conditions which may be complicated to describe in ontologies. The RIF-based rule inference follows the *Closed World Assumption* [46] which means that everything which is not derivable by a given set of RIF rules is considered to be false.

[5] `http://lov.okfn.org/dataset/lov`

RIF comes with two different dialects (i) *Basic Logic Dialect* (BLD) using logic rules to add new knowledge, and (ii) *Production Rule Dialect* (PRD) using production rules to change facts. Both extend the *Core Dialect* which uses the *Datatypes and Built-Ins* (DTB) specification describing a set of built-in datatypes, predicates and functions to be supported by all dialects. The additional dialect *Framework for Logic Dialects* (FLD) extends BLD and provides a formalism for specifying logic RIF dialects. New user-defined logic RIF dialects can be derived from FLD to cover new needs. This work does not provide support for hardware-accelerated rule execution and thus is not further examined. However, we will outline possible extensions of our approach including support of rule languages in Chapter 6.

2.1.2.5 Applications of the Semantic Web

Due to its nature, the SW is not visible for humans in the first place. The W3C provides a list of several reported prototypes (*use cases*) and deployed systems (*case studies*) using SW technologies [47]. In the following, we will shortly review some of them to show the variety of possible SW applications.

Especially in large corporations the integration and consolidation of disparate applications and different data sources is expensive in terms of time and resources. The automobile manufacture *Groupe Renault* established a repository with an OWL model storing repair, diagnostic operations and related concepts [48]. Instead of showing all possible diagnostic procedures in a manual, the web-based tool enables the technician to progressively discover repair information at different granularity levels and is instructed to perform tests and enter results. For information exchange and publishing RDF is used.

The broadcasting corporation BBC publishes large amounts of text, audio and videos online. In the past, they provided separate websites each focused on a broadcasting brand or a series of specific domains. Major issues regarding user experience resulted from the lack of cross linking and thus it was not possible to present contextual information (such as *which artist performed in the TV show*). *BBC Programmes* provides a uniform appearance using web identifiers and machine-readable formats (RDF/XML) for every BBC program [49]. Each site can be simply incorporated into other sites across different program genres. Similar *BBC Music* provides a web identifier for every artist and additionally gathers and aggregates informations from other public linked datasets (Wikipedia, MusicBrainz). In turn BBC editors directly contribute to these datasets.

AGFA Healthcare transformed clinical textual guides consisting of appropriateness criteria into a machine-readable medical knowledge base using RDF and OWL [50].

Recommendations for radiology orders with respect to the patient's condition can be generated by using SW rule engines. Furthermore, the reuse and linking of existing medical knowledge is possible and allows a standard description of clinical problems and patient conditions.

The Drug Ontology Project for Elsevier (DOPE) provides a single interface to access multiple sources in the life science domain [51]. Therefore, Elsevier's main life science thesaurus has been converted to RDFS, which afterwards has been used to index and describe several large data collections. The DOPE Browser allows to query and navigate through the collection, while hiding complexity such as multiple data sources.

The General Council of the Judiciary in Spain established a search system for young judges which provides and enhances the expert knowledge of over 400 judges from all over Spain with ontologies [52]. The search performs several steps such as topic detection, classification into subdomains, finding semantic similarities based on the question's legal meaning (defined in ontologies) rather than on keywords.

Typically, administrative authorities (cities, states, etc.) publish many documents (announcements, laws, etc.) every day. Citizens are supposed to read them but these documents are written in legal and administrative jargons, which causes a barrier between citizens and administration. A keyword based search fails on terms that are synonyms from a citizen's view but are clearly different for experts. Thus, the Principality of Asturias (Spain) provides an interface to express queries in their own words, which then are analyzed for the underlying concepts of terms to transform them into a common context [53]. The OWL-based ontologies include more than 10,000 concepts. Two thesauri consisting of the end-user and the expert vocabulary are linked to the concepts in the ontologies.

Besides others, the *Linking Open Data* (LOD) project [3] aims to publish new RDF datasets and to interconnect existing open RDF datasets. As a result the LOD cloud consisted of more than 30 billion triples and more than 500 million links in 2011[6] and hence often serves as an example of *Big Data*. One data source is the community-driven DBpedia [54] which extracts structured information (such as info boxes) from Wikipedia and provides a SPARQL endpoint to this dataset.

The catalog WorldCat [55] is the world's largest network of library content. It itemizes the collection of libraries all over the world. After searching a specific item, a webpage with embedded RDFa (which describes various properties of the bibliographic item) is returned. All these embedded information can be extracted and used in an automated fashion without the need of additionally providing access

[6] `http://lod-cloud.net/state/` (Version 0.3, 2011-09-19 / accessed: 2016-05-11)

methods such as web services or dedicated *Application Programming Interfaces* (APIs).

2.1.3 LUPOSDATE - A Semantic Web Database

The LUPOSDATE project [56] was originally initiated by Dr. Sven Groppe at the Institute of Information Systems and funded by the DFG[7]. After the funding period and still on-going, the LUPOSDATE system is continuously improved regarding performance and feature richness. It provides several query engines, all (except streaming engine) supporting full SPARQL 1.0 and SPARQL 1.1. The code is freely available as open source [57] and allows anybody the easy integration of SW technologies in any other application or the extension and contribution to LUPOSDATE itself. The developed architecture of this work is based on and integrated into the LUPOSDATE system. Therefore, in the following section the basic architecture LUPOSDATE shown in Figure 2.5 is introduced with focus on the main components (i) index construction and the underlying data structures as well as (ii) the processing stages in query evaluation.

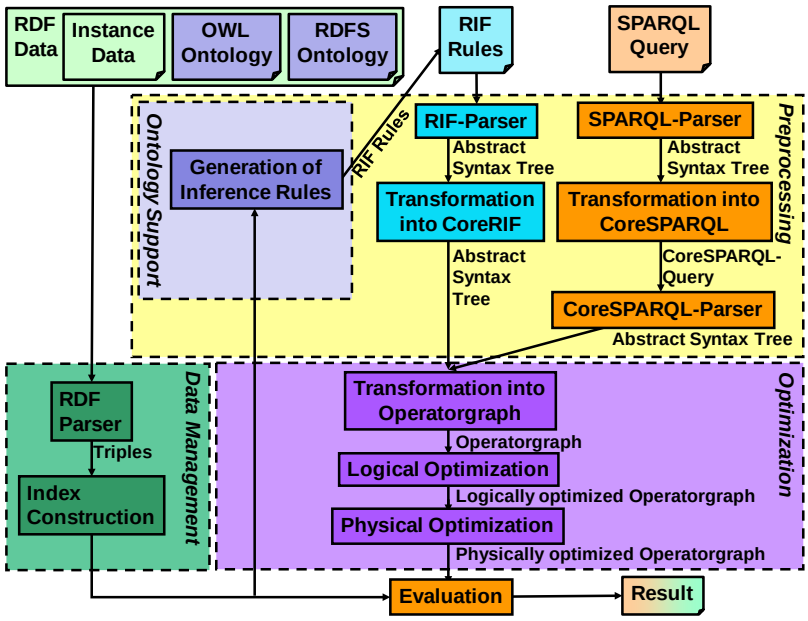


Figure 2.5 – Architecture of the Semantic Web database LUPOSDATE [58].

[7]Deutsche Forschungsgemeinschaft (German Research Foundation)

2.1.3.1 Index Generation

When it comes to manage large-scale datasets typically indices are used to access the stored data in an ordered and structured fashion. Over the past decades numerous indexing data structures have been presented. A general classification is to split them into hash-based and tree-based indices. In-memory optimized indices are usually based on hashing and perform very well on punctual requests. Due to the distributive nature of hash functions, hash-based indices do not support range or prefix searches. Additionally, although the size of available main memory is continuously growing, usually data and indices are persistently stored on disk-based devices such as *Hard Disk Drives* (HDDs). One drawback is their poor performance caused by the comparably long access time especially at random access patterns. On the other side HDDs are organized in relatively large blocks of data which allows them to store more data at one address. Regarding these properties, the B-tree [59] has been developed as an optimized data structure for disks and is widely used in database and file systems. Due to the block-based node structure, each node can store multiple pairs consisting of key and data. A variant, the B^+-tree [60], stores the actual data only at the lowest level (leaves). Thus, leaf nodes and internal nodes may have different formats. Latter can store more referential data (keys) which reduces the height of the tree. Additionally each leaf may have a pointer to the next leaf (*sequence set*) which, after finding the leaf, enables sequential access through the succeeding leaves. This is desirable especially for database systems.

With respect to RDF triples, SPARQL engines like *RDF3X* [61] and *Hexastore* [62] apply a simple but efficient scheme to provide the optimal (disk- and tree-based) index for a given triple pattern. Therefore, six *evaluation indices*, one for each of the possible collation orders (SPO, SOP, PSO, POS, OSP and OPS), are generated. The collation order states in which order the triple components s, p and o are considered as primary, secondary and tertiary keys in a global triple sequence. For instance, the collation order POS determines that triples are primarily sorted according to predicate (P), secondarily sorted according to the object (O), and tertiary sorted with respect to the subject (S). The POS index can efficiently retrieve triples which match to a triple pattern with given predicate only (object and subject are variables) or given predicate and object (only subject is a variable) by using the known components as a prefix key. In order to avoid an up to six times higher space consumption, the evaluation indices typically rather use integer identifiers (IDs) for strings than the actual component strings. This is beneficial in query evaluation as well because the main memory consumption is greatly reduced. Therefore, during import of RDF triples into LUPOSDATE all component strings are sorted in ascending order. Due to the amount of strings, they have to be sorted in several *runs* which are finally merged. When it comes to efficiently

store strings, especially those with common prefixes, usually *Patricia Tries* [63] are the first choice. Thus, they are used to generate the initial runs stored on external storage. In a final run the generated Patricia Tries are merged to retrieve the final sorted dataset [64]. Afterwards, an ascending integer ID is assigned to each containing string. The mappings between IDs and strings and vice versa are stored in additional *dictionary indices*. The index for translating strings to IDs is based on B^+-trees, which can be constructed easily due to the fact that the strings are already sorted. The opposite direction, IDs to strings, is implemented as HDD-based array realized using two files[8]. Obviously, the equality of two strings can be simply checked by comparing their IDs. If the IDs are the same then the underlying strings are the same as well. Furthermore, if the dictionary is not changed afterwards (by inserting a new string) or the lexical order is maintained (complete recreation of dictionary after insertion) then even relational comparisons like *smaller* and *greater* can be supported[9]. If latter can not be guaranteed and in all other cases (e.g., advanced string operations such as *substring* or regular expressions) a lookup in the dictionary to obtain the textual representation is mandatory. However, in a final run through the imported RDF data, each triple respectively its components are mapped to the related ID forming an *ID triple*. The ID triples are used to build the six evaluation indices [14]. Regarding the previous example in Listing 2.1, the corresponding dictionary is presented in Figure 2.6. Additionally, the resulting RDF triples consisting of the IDs and B^+-tree of the SPO index are shown. In this work the index maintenance and access is always served by the software system. However, Heinrich *et al.* [65] present a *hybrid* index which stores frequently accessed interior nodes within a hardware accelerator.

2.1.3.2 Query Evaluation

Query evaluation (or execution) is nothing new and lots of database research has investigated on optimizing query evaluation with respect to the shortest execution time. *Database Management Systems* (DBMSs) provide a *Query Language* (QL) to manipulate and retrieve data from one or more databases. After the user application submitted the query, it is parsed using a grammar defining the query language resulting in an *Abstract Syntax Tree* (AST). In an additional processing stage, LUPOSDATE transforms the given AST into a core fragment of SPARQL, called *CoreSPARQL* [66], which excludes redundant language constructs but still has the same expressiveness. Like in relational databases, the components of SPARQL queries are finally broken down to a set of nestable basic operators and are combined to an operator graph (or tree) representing the query. Note that the chosen

[8] One file consisting the ordered IDs and a references pointing to the string in other file [57].
[9] This is not implemented in LUPOSDATE since SPARQL 1.1 supports updates.

ID	component string
0	"Stephen Hawking"^^xsd:string
1	"Popular Journal"^^xsd:string
2	"English abstract"@en
3	l:Article1
4	l:Famous_Author
5	l:Journal1
6	rdf:type
7	rdfs:subClassOf
8	v:abstract
9	v:Article
10	v:creator
11	v:Document
12	v:Journal
13	v:name
14	v:publishedIn
15	v:title

```
11  7 10 .
 9  7 10 .
 5  6 11 .
 5 15  1 .
 4  6  9 .
 3  8  2 .
 3 14  5 .
 3 10  4 .
 4 13  0 .
```

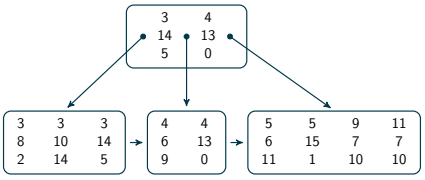


Figure 2.6 – Dictionary for component strings (*left*) and resulting ID triples (*right*) including possible B$^+$-tree (SPO order) regarding Listing 2.1 (without prefix substitution).

operators are only *logically* described and no actual implementation is given at this stage.

Each SPARQL operator (except the index scan operator) can be expressed as relational algebra. Definition 2.1 introduces the operations of the relational algebra and common extensions [67, 68]. It is worth to mention that the relational algebra is not turing-complete which allows easier optimization in order to provide an efficient access to large datasets. The relational algebra has a *set* semantic which implies that all (intermediate) results are without duplicates. But elimination of duplicates is not trivial and might consume much memory. Therefore, real query engines use the *multiset* (or *bag*) semantic due to the fact that it can be implemented more efficiently. However, typically they provide constructs to remove duplicates in a final processing step. This needs to be explicitly expressed in the query language.

In this work, we develop a runtime adaptive hardware accelerator to improve query performance of SPARQL queries supporting the following subset of the extended relational algebra operators: projection, selection/filter, join, union and limit/offset. The triple pattern scan is implemented as a hybrid operator such that it scans the triple indices in software and sends matching triples to the hardware accelerator. The final hybrid hardware-software architecture is able to fall-back into software-based query processing if the query consists of an unsupported operation and thus provides full SPARQL 1.1 support.

Definition 2.1 (Relational algebra and common extensions)
Given two relations $R(r_1, \ldots, r_m)$ and $S(s_1, \ldots, s_n)$ with r_i respectively s_j attributes of R respectively S. The relational algebra defines a set of operators on *sets of tuples* R and S, and thus no duplicate tuples are allowed. However, many real database languages and implementations follow multiset semantics. Therefore, we present the operators on R and S with respect to set ($\{\ldots\}$) semantics and on the multiset relations R_M and S_M with respect to multiset ($\{\ldots\}_M$) semantics. If the definition of a multiset operator is the same as the set operator then we denote it as *multiset equivalent*.

Projection returns set/multiset consisting of tuples only with the attributes $v_1, \ldots, v_k$ of R:

$$\pi_{v_1,\ldots,v_k}(R) := \{(t[v_1], \ldots, t[v_k]) \mid t \in R\} \text{ (multiset equivalent)}$$

Note: $t[v_i]$ represents the value in tuple t at attribute v_i.

Selection/Filter/Restriction returns set/multiset of tuples of R which hold a given propositional formula c:

$$\sigma_c(R) := \{t \mid t \in R \wedge c(t)\} \text{ with constraint } c \text{ (multiset equivalent)}$$

Note: The constraint c can be a simple variable/constant, a function call or a complex expression $r_i \theta r_j$ with r_i, r_j constants or attributes of R and $\theta \in \{=, \neq, <, \leq, \geq, >\}$. Furthermore, c can be of nested form $\neg c_1$ or $c_1 \wedge c_2$ or $c_1 \vee c_2$ with constraints c_1, c_2. SPARQL provides a rich set of built-in functions such as type checks, (NOT) EXISTS or regular expressions. For a full list of built-in functions see [33].

Cross Product returns set/multiset consisting of tuples of R concatenated with each tuple of S:

$$R \times S := \{(r_1, \ldots, r_m, s_1, \ldots, s_n) \mid (r_1, \ldots, r_m) \in R, (s_1, \ldots, s_n) \in S\}$$
(multiset equivalent)

Note: With $|T_M|$ cardinality of any multiset T_M then $|R_M \times S_M| = |R_M| \cdot |S_M|$. With $|T_M|_t$ cardinality of tuple t in T_M, then $|R_M \times S_M|_{rs} = |R_M|_r \cdot |S_M|_s$.

Union returns all tuples of R and S:

$$R \cup S := \{t \mid t \in R \vee t \in S\} \text{ (multiset equivalent)}$$

Note: With respect to union on multisets it holds $\forall t \in R_M \cup S_M : |R_M \cup S_M|_t = |R_M|_t + |S_M|_t$. In relational databases R and S must have the same scheme of attributes. In SPARQL union is implemented as *outer union* ($R \uplus S$), thus equal schemes are not required and unbound variables remain unbound in the result.

Difference returns set of all tuples of R which do not appear in S respectively multiset with as many removed duplicated tuples from R_M as appear in S_M:

$$\begin{aligned} \text{Set semantic:} \quad & R \backslash S := \{t \mid t \in R \wedge t \notin S\} \\ \text{Multiset semantic:} \quad & R_M \backslash S_M \subseteq R_M \wedge \\ & \forall t \in R_M \backslash S_M : |R_M \backslash S_M|_t = max(0, |R_M|_t - |S_M|_t) \end{aligned}$$

Note: In relational databases R and S must have the same scheme of attributes. This is not required in SPARQL.

The following operators can be expressed using the minimal set but are often defined explicitly as follows:

Intersection returns set/multiset of all tuples of R which also appear in S:

$$R \cap S := R \backslash (R \backslash S) \text{ (multiset equivalent)}$$

Note: R and S must have the same scheme of attributes. Due to the semantic of difference on multisets a tuple $t \in R_M$ and $t \in S_M$ will appear $min(|R_M|_t, |S_M|_t)$ times in $R \cap S$.

Theta-Join returns set/multiset of all tuples of R concatenated with all tuples of S which hold given a propositional formula c:

$$R \bowtie_c S := \sigma_c(R \times S) \text{ (multiset equivalent)}$$

Note: Join condition c is defined as selection condition (see selection). The *Equi-Join* restricts the condition to consist only of conjunctions of equality expressions. The *Natural Join* is based on the Equi-Join but implicitly checks equality only on commonly named attributes in both relations. One occurrence of each common attribute is eliminated from the result [68]. In SPARQL queries, the natural join is inferred for joining the results of different triple patterns with common variables (attributes).

The following operators can ***not*** *be expressed using the minimal set and therefore are often defined in QLs to enhance expressiveness:*

Extended projection enables renaming of attributes and binding aggregate values to attributes of the resulting relation:

$$\pi_{F_1,\ldots,F_k}(R) \text{ with either } F_i \in \{r_1,\ldots,r_m\} \ (r_j \text{ attribute of } R)$$
or F_i is an expression of the form $X \to Y$ with Y as new attribute and X is either an attribute of R (renaming), a constant value, an aggregation function or another complex arithmetic formula respectively string operation (multiset equivalent).

Note: Renaming is often necessary when other operations require a common scheme of the input relations such as union, difference and intersection. SPARQL supports renaming in SELECT expressions and BIND clauses. Additionally, SPARQL provides aggregate functions on associated attributes for counting the number of values (COUNT), returning the sum of values (SUM), returning the smallest respectively largest value (MIN/MAX), calculating the average of values (AVG), returning an arbitrary sample value (SAMPLE) and performing string concatenation across the values (GROUP_CONCAT) [33].

Distinct returns a set of a given multiset R_M:

$$\delta(R_M) := \{t \mid t \in R_M\}$$

Note: Duplicate elimination in real DBMS implementations is expensive because all tuples must be either sorted or partitioned.

Order-By returns a sorted *sequence* $R_{\langle\rangle}$ of R with respect to bound values of given attributes $v_1,\ldots,v_k$:

$$\tau_{v_1,\ldots,v_k}(R) := T = \langle t_1,\ldots,t_n\rangle \text{ with } v_i \in \{r_1,\ldots,r_m\} \wedge$$
$$\begin{aligned}\forall t_i \in T \text{ holds} \quad & t_i[v_1] < t_{i+1}[v_1] \vee (t_i[v_1] = t_{i+1}[v_1] \wedge \\ & t_i[v_2] < t_{i+1}[v_2] \vee (t_i[v_2] = t_{i+1}[v_2] \wedge (\ldots))) \\ & \wedge |R| = |R'| \wedge t_i \in R \text{ (multiset equivalent)}\end{aligned}$$

Note: Overall the semantic of sequences is equal to sets respectively multisets but sequences define a (not necessarily sorted) order on the containing elements. In real DBMS implementations the results of most operators are determined in an iterative and deterministic fashion and thus order is given implicitly. All previously described operators can process sequences as well.

Limit returns a sequence consisting of the first n tuples of sequence $R_{\langle\rangle}$:

$$limit_n(R_{\langle\rangle}) := \langle t_i \mid t_i \in R_{\langle\rangle} \wedge 1 \leq i \leq n \leq |R| \rangle \ (t_i \text{ is the } i\text{-}th \text{ tuple in } R)$$

Offset returns a sequence consisting of all except the first n tuples of $R_{\langle\rangle}$:

$$offset_n(R_{\langle\rangle}) := \langle t_i \mid t_i \in R \wedge n < i \leq |R| \rangle \ (t_i \text{ is the } i\text{-}th \text{ tuple in } R)$$

Note: Offset respectively Limit require a fixed order of tuples which is typically given implicitly in real DBMS implementations.

Group-By returns a sequence of tuples of set/multiset R partitioned into groups regarding to the bound values in one or more specified attributes [68]:

$$G_{F_1,\ldots,F_k}(R) \text{ with either } F_i \in \{r_1,\ldots,r_m\} \ (r_j \text{ attribute of } R)$$
or F_i is an expression of the form $X \to Y$ with Y as new attribute and X is either an attribute of R (renaming), a constant value, an aggregation function or another complex arithmetic formula respectively string operation (multiset equivalent).

Note: Additionally, SPARQL provides the HAVING expression which can be used to further restrict the tuples to be consolidated in a group [33].

Left Outer Join / Optional returns the join of R and S, and additionally contains tuples of input relation, which do not have any join partner in the other input relation:

$$R \mathbin{⟕} S := (R \backslash (R \ltimes S)) \uplus (R \bowtie S) \ (\textit{left outer join}) \text{ with}$$
$$R \ltimes S := \pi_{r_1,\ldots,r_m}(R \bowtie S) \ (\textit{semi join})$$
(multiset equivalent)

Note: Right outer join $R ⟖ S$ is realized by simply substituting R with S respectively S with R. *Full outer join* $R \mathbin{⟗} S := (R \backslash (R \ltimes S)) \uplus (R \bowtie S) \uplus (S \backslash (S \ltimes R))$.

With respect to SPARQL queries the following additional operator is defined:

Triple Pattern Scan returns triples satisfying a given pattern:

$$
\begin{aligned}
(e_1, e_2, e_3) := \{ \quad & E \mid (d_1, d_2, d_3) \in D \wedge \\
& E = ((x, v) \mid i \in \{1, 2, 3\} \wedge x = e_i \wedge e_i \in V \wedge v = d_i) \wedge \\
& ((\forall j \in \{1, 2, 3\} : (e_j \in V) \vee (e_j = d_i)) \wedge \\
& \qquad \forall (n, v_1) \in E : \forall (n, v_2) \in E : v_1 = v_2) \\
\} \quad & \text{with } D \text{ input graph containing all input triples}
\end{aligned}
$$

Note: The triple pattern scan returns always a set because the input graph D is defined to be a set of triples.

Regarding the previous example query (Listing 2.2 in Section 2.1.2) the logical operator graph shown in Figure 2.7 is generated. In this simple example the transformation of the query to a corresponding operator graph is obtained in a straight forward fashion. Each triple pattern results in one index scan operator with a collation order regarding the known terms in the triple pattern. As the first two index scans provide bindings for the common variable ?author, both intermediate results need to be joined. The calculated intermediate result consists of bindings for the variables ?author, ?aName and ?doc. The latter in turn is a common variable supplied by the remaining index scan and thus both intermediate results need to be joined regarding ?doc. The last step is projecting out the unwanted variables because in the query it is only asked for bindings of the variable ?aName. On the other hand, it is also possible to calculate the join of intermediate results provided by the second and third index scan, and afterwards join with the intermediate result of the first index scan. Typically for a given query there are numerous possible operator graphs which differ in processing *costs*, mostly in terms of time required to calculate the final result [69]. As a consequence database systems are equipped with optimizers which try to form an optimal operator graph which obtains the result in least amount of time. But already finding an optimal join order is a NP-hard problem [70, 71] and it must be avoided that the optimizer consumes too much time even before the processing of the query result has started. This goal is achieved by using best-effort optimizers which do not necessarily find the optimal solution but still significantly reduce the processing costs of complex queries. In the logical optimization, equivalence rules are applied to reduce the size of intermediate results as early as possible in the operator graph. This methodology saves memory space and processing costs at later possibly more complex operators. For instance, the early execution of a highly selective filter before a join can significantly impact the overall execution time as the join does not need to compare so many potential join partners any more. Other equivalence rules are early execution

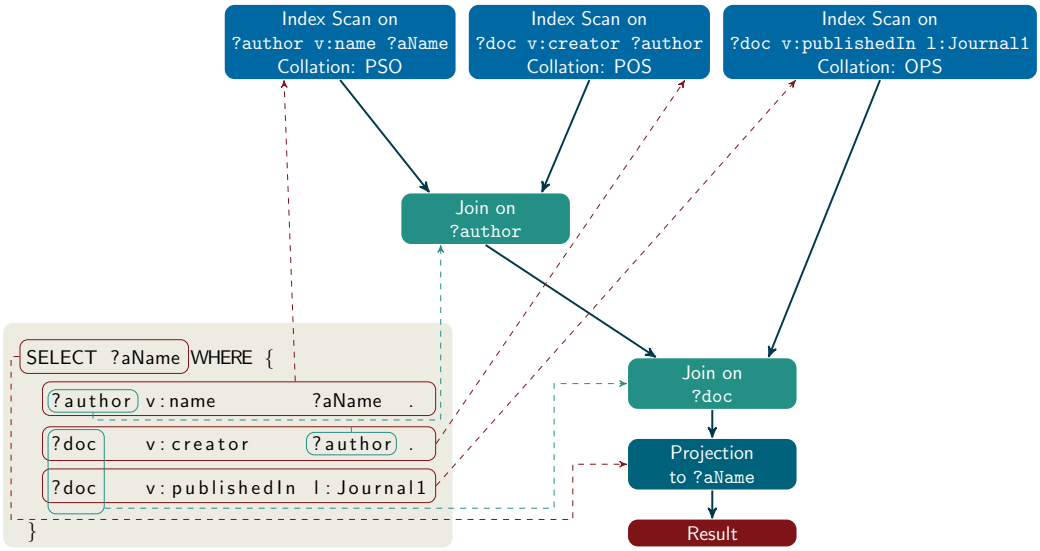


Figure 2.7 – Transformation of example query from Listing 2.2 into a logical operator graph.

of projection, combining sequences of projections and selections, associating cross product with selection to join, and identification of common subexpressions [72]. However, applying a rule does not necessarily lead to a better performance. Thus, heuristic methods apply certain rules which usually result in a better performance. With respect to finding a good join order, logical optimizers use cost models and estimations about the input size based on statistics (cost-based optimization). Afterwards, the physical optimization chooses for each logical operator a concrete implementation (physical operator) based on estimations about the structure (e.g. *is input sorted?*) and size of the intermediate results. Concrete implementations of physical operators especially join operators will be introduced in Chapter 3 of this work. The final operator graph consisting of physical operators is called *Query Execution Plan* (QEP).

Pipelining and Sideways Information Passing

As described before, each operator (except the index scans) gets as input the output of one or more preceding operators (intermediate results). In fact, many operators do not need the complete intermediate result of their predecessors to start processing and providing their own intermediate results. Instead of handing over completely *materialized* intermediate results, which possibly exceed main mem-

ory limits (with all its drawbacks[10]), a *pipeline* is established between operators providing only *some* intermediate results. This greatly reduces the main memory consumption in many cases during query evaluation. One widely used software paradigm to support the pipelined execution model is called *iterator concept* [73, 74] and requires only a minimal set of methods, namely open(), next() and close(). The method open() is used to start the computation. Afterwards each intermediate result can be requested one by one from predecessors using the corresponding next() method. The predecessors in turn might implicitly call open() and next() of their predecessors in order to calculate and provide an intermediate result. If all intermediate results are supplied or other conditions are reached to finish computation then the close() methods of all operators are called in a cascading manner which typically includes *cleaning* procedures such as releasing resources. However, some operators need to know the complete input data before starting the computation and tend to stall the whole processing pipeline, such as minus, sorting, grouping, and aggregation. Therefore, they are called *pipeline breaker*. Depending on the actual implementation some operators can be realized either as pipeline breaker or not, such as join and duplicate elimination. Pipeline breakers still support the iterator concept by first calculating the complete intermediate result internally and providing it one by one to the succeeding operator. Thus, the fact whether an operator is a pipeline breaker or not is completely hidden to the other operators which further strengthens modularization.

Another performance optimization is achieved by introducing an additional method next(v) which takes an argument v which is handed over to the preceding operator. The value v describes a lower limit which needs to be satisfied by the next provided intermediate result which implies that all potential previous intermediate results are not useful for the calling operator. Thus, processing steps are saved by *skipping* unnecessary intermediate results. Regarding index scans and their underlying B^+-trees this value can be used to jump across several leaves (which are read from slow HDD) by navigating[11] through only some internal nodes instead of traversing a large amount of leaves containing unwanted entries. This approach is called *Sideways Information Passing* (SIP) [75, 76] and can be highly beneficial if there are large gaps in otherwise unselective index scans.

[10] swapping to HDD
[11] prefix search with v

2.2 Reconfigurable Computing

In the last decades, computer technology has made enormous progress regarding different characteristics. On one side, the size of digital storage is still exponentially increasing - without any prospect of an end [77]. First available HDDs started with 3.5 MByte in the year 1956. Nowadays, 1 to 3 TByte is standard, but HDDs with up to 10 TByte are available [78]. On the other side, processing and analysis of this huge amount of data within a reasonable time frame becomes more and more difficult in traditional von-Neumann-based architectures using general-purpose CPUs. Besides application-specific data structures and optimized algorithms to increase performance, technological advances enabled shrinking feature size to increase clock frequency and thus the overall performance. However, nowadays these approaches reach limits to deal with known as the *power wall* [79, 80]. Although smaller feature size causes a lower energy consumption, effectively more transistors are placed in the same chip area. But a higher transistor density results in a higher power density, which directly increases the temperature emission. Either sophisticated (expensive) cooling solutions are used to counteract these high temperature or operating frequency is lowered, which in turn results in lower performance. In the last years, this contradiction led to the trend using multi/many-core[12] systems in order to increase overall system performance. Additionally, these systems are not assembled with uniform processing cores, but rather are composed by heterogeneous/asymmetric and specialized cores. Well-known representatives of specialized cores are *Graphics Processing Units* (GPUs) and *Digital Signal Processors* (DSPs). These are tailor-made *hardware accelerators* to solve a particular task very efficiently in terms of power consumption and performance. Besides GPUs and DSPs, for many other tasks so-called *Application-Specific Integrated Circuits* (ASICs) are developed. Typically, ASICs perform orders of magnitude better than solving the same task in software running on a general-purpose CPU. However, the main advantage of specialized cores computing one specific task efficiently, at the same time becomes a disadvantage as they can not be used in application domains showing a large variety on processing. Furthermore, architectural restrictions of the traditional von-Neumann architecture became apparent even before physical barriers have been reached. While CPU performance steadily increased for the last decades, the limited communication bandwidth for main memory access turned out to be a shortage known as the *von-Neumann bottleneck* [81] or *memory wall* [82, 83]. As a consequence CPUs spend much of their time *waiting* for the memory to response. Therefore, typically small but very fast memory (*cache*) is located within the CPU to store recently used instructions and data. Furthermore, different cache levels

[12] multicore: *small* number of *large* cores, many-core: *large* number of *small* cores

with growing size but slower speed are provided. However, this approach can only reduce but can not solve the performance degradation.

Reconfigurable hardware offers a solution to close the gap between flexibility and performance. Already in 1960, Estrin [84] sketched an architecture consisting of fixed structures and an inventory of high speed substructures, which can be dynamically connected to provide a problem oriented special purpose computer. In 1977 the *Ramming Machine* was presented as *a system, which, with no manual or mechanical interference, permits the building, changing, processing and destruction of real (not simulated !) digital hardware* [85]. Furthermore, the *data-driven* principles of *Xputer* were introduced in contrast to control-flow-driven von-Neumann architectures [86]. Nowadays *Reconfigurable Computing* involves architecture using reconfigurable devices providing dedicated application-specific hardware structures. With the wide availability of *Field-Programmable Gate Arrays* (FPGAs) and their increase of logic, reconfigurable computing became a vivid research field in numerous applications.

In this work we utilize an FPGA in the context of query evaluation. Therefore, the next section gives a brief overview regarding the internal architecture and provided resources of FPGAs. Afterwards, we describe how to obtain an application-specific configuration suitable for FPGAs. The chapter concludes with several application scenarios utilizing FPGAs.

2.2.1 Field-Programmable Gate Array

Just like ASICs, FPGAs implement functions in dedicated resources distributed on the chip in a possibly highly concurrent and spatial fashion. But instead of permanently binding particular resources to a specific operation, FPGAs allow reconfiguration of these resource allocations *in-the-field* (post fabrication). Typically, the FPGA device itself is mounted on an FPGA board, which provides several standardized physical interfaces to communicate with other devices such as a host systems, HDDs or other FPGAs. Numerous manufactures are available and are not necessarily producing both, FPGA dies and FPGA boards.

In the following sections, we will focus on FPGAs provided by the manufacturer Xilinx. Currently, Xilinx is market leader and in this work a Xilinx Virtex-6 FPGA [87] is used. However, the overall concept and structure of FPGAs are very similar and typically differs in used manufacturing technology, feature size, number of provided logic resources and dedicated hardware cores (ranging from DSPs up to full CPUs). The FPGA is organized in an array of *basic building blocks*. In Xilinx devices, these blocks are called *Configurable Logic Blocks* (CLBs) and

	XC6VLX75T	**XC6VHX380T**	XC6VHX565T
Slices Total	11,640	59,760	88,560
SLICEL	7,460	41,520	63,080
SLICEM	4,180	18,240	25,480
Number of 6-input LUTs	46,560	239,040	354,240
Number of FFs	93,120	478,080	708,480
Distributed RAM [Kb]	1,045	4,570	6,370
Shift Register [Kb]	~522	2,285	3,185

Table 2.1 – Available logic resources of three Virtex-6 devices [88].

are arranged in columns together with other dedicated resources (see subsequent sections). Regarding Virtex-6 FPGAs, each CLB consists of 2 slices [88]. A slice is assembled of four *Lookup Tables* (LUTs), three *Multiplexers* (MUXs), a carry chain and eight *Flip-Flops* (FFs). Each LUT can be used to implement one function with six input bits and one output bit. Furthermore, one LUT can be utilized to implement two functions, each with five input bits (using common logic inputs) and two separate output bits. The output of the LUTs can be stored in the FFs. Each LUT is realized using a very fast and tiny memory utilizing the input as address to select a single bit used as the output. Typically these memories are based on volatile *Static Random-Access Memory* (SRAM), but other technologies such as non-volatile flash or non-reversible *anti-fuse* are available. Regarding a 6-input LUT the memory stores 64 bits. Furthermore, slices are differentiated into *SLICEL* and *SLICEM*. Besides implementing only combinatorial functions (*SLICEL*), the LUTs of between 25 to 50% of all slices can be used as distributed 64-bit *Random-Access Memory* (RAM) or 32-bit shift registers (*SLICEM*). Table 2.1 outlines the available logic resources of three different devices of the Virtex-6 family ranging from the families low-end (XC6VLX75T) to high-end (XC6VHX565T) including the FPGA used in this work (XC6VHX380T). Figure 2.8 schematically shows a very small segment of the CLB array, which typically consists of hundreds of thousands CLBs. In order to achieve higher functions each CLB is connected to a hierarchical communication network consisting of *Programmable Interconnect Points* (PIPs), which enable the reconfiguration of communication links between CLBs. Furthermore, each slice provides dedicated carry logic to establish fast carry chains for vertical data propagation directly from one slice to another above. Carry chains can be cascaded to form wider add and subtract logic [88]. Within the CLB array, additional dedicated resources such as *Block RAMs* (BRAMs) or DSPs are column-wise located and will be introduced in the following sections.

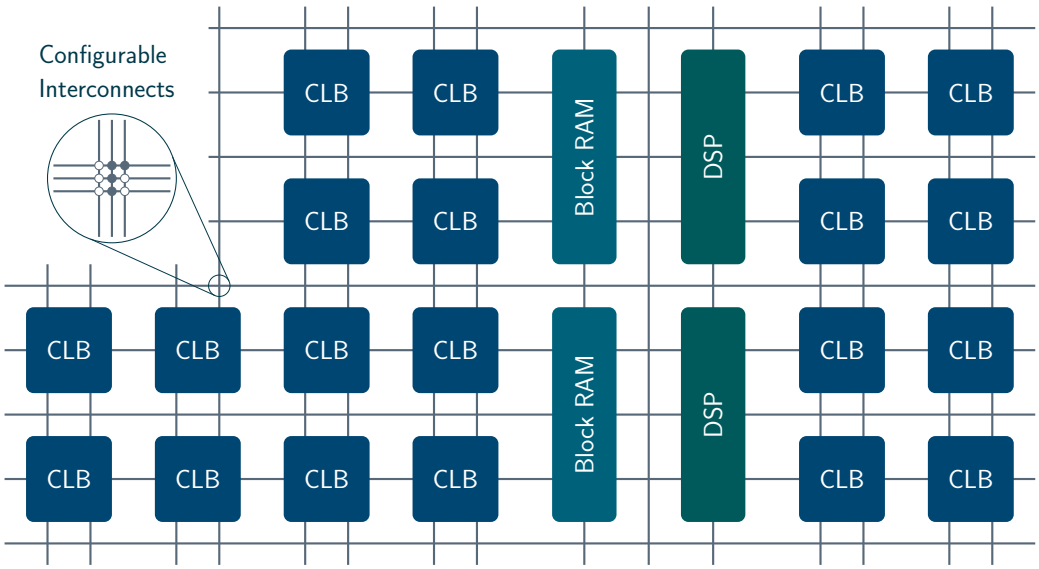


Figure 2.8 – Array of *Configurable Logic Blocks* (CLBs). Reconfigurable switch matrices allow flexible interconnection between CLBs. Additional resources such as *Block RAMs* (BRAMs) and *Digital Signal Processors* (DSPs) are located within the CLB mesh.

2.2.1.1 Memory Resources

As already mentioned some of the LUTs (in SLICEM) can be utilized as distributed memory in a very granular and localized fashion. Together with a very low access time (<1 ns), it opens up a wide range of applications as small data buffers (such as *First-In First-Out queues* (FIFOs) and shift registers) or small state machines. Additionally, on-chip *Block RAM* (BRAM) [89] is available, which is located in columns within the CLB array (see Figure 2.8). Depending on the device, multiple BRAMs are placed in columns in the whole FPGA die. In case of the Virtex-6 each block provides 36 KBit on-chip storage, but can be segmented into two independent 18 KBit BRAMs. Furthermore, they support true dual-port access by providing two completely independent access ports (each with own address and data signals in possibly different clock domains). Besides dedicated FIFO logic (to eliminate the need for additional CLB logic), an integrated cascade logic enables to logically combine BRAMs for higher storage needs using one common interface. This is supported by Xilinx' *Memory Interface Generator* (MIG) to generate various memory structures depending on the application's need. It also supports 64-bit *Error Correction Code* (ECC) to detect single or double-bit errors, and correct them in case of single bit errors. In summary, BRAM is ideal for mid-

sized buffering in terms of usage as local data storage, shift registers and FIFOs (due to integrated support by dedicated FIFO logic[13]). Both, distributed RAM and BRAM can be preloaded with the FPGA configuration. Furthermore, MIG allows to generate interfaces for various kinds of external memory, which are not located in the FPGA die, but on the FPGA board. Supported interfaces for Virtex-6 are DDR2 SDRAM, DDR3 SDRAM, RLDRAM II and QDRII+SRAM [90]. Thus, FPGA architectures provide a flexible memory hierarchy ranging from small but very fast storage up to comparably larger but slower memory with different possible granularities as outlined in Figure 2.9. Depending on the FPGA board, for each memory type there can be multiple modules. These modules can be utilized as one logical unit using only one controller implemented in the FPGA or using multiple controllers (one for each module) for concurrent access in the same or different tasks. This allows the hardware designer to create application-specific caching or pre-fetching strategies between different memory levels to successfully cascade access delays.

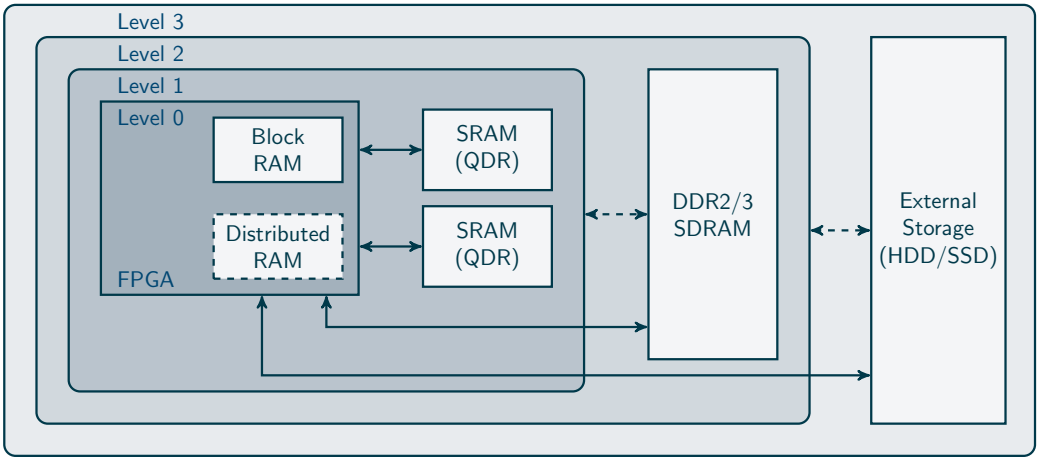


Figure 2.9 – Example of a memory hierarchy. Different memory types enable application-specific caching, pre-fetching and swapping strategies.

Besides BRAM, Figure 2.8 shows dedicated DSP elements, so-called *DSP48E1* tiles, each assembled of two DSP48E1 slices [91]. These highly optimized resources provide a variety of frequently required functions ranging from simple functions such as add and multiply, magnitude comparisons or bit-wise logic functions up to complex pattern detections and multi-precision arithmetics by cascading multiple DSP48E1 in a column. Deploying these DSP48E1 further reduces the general utilization of logic resources.

[13] one 18 KBit FIFO and one 18 KBit BRAM

2.2.1.2 Clocking Resources

Although FPGAs operate at a relatively low clock frequency and performance is mostly obtained due to their inherent parallelism and pipelining, clocking receives special attention. Contrary to running software on a higher clocked general-purpose CPU, which does not necessarily result in a better performance at the same scale, improving the achievable frequency of an FPGA design directly impacts the whole performance to the same extend. Thus, there are several dedicated global high speed signal paths used for clock routing only. They allow the distribution of clock signals all over the FPGA reaching every FFs with low skew. Skew is the maximum delay of the clock input of one FF to another one. Depending on the device size each Virtex-6 is divided into 6 to 18 clock regions [92]. Figure 2.10 provides a schematic view of the XC6VHX380T obtained from the software tool PlanAhead [93]. The die is divided into two columns spanning half of the device with each nine clock regions. Between the two clock region columns the *central column* is located. It contains 32 vertical spines of the global clock trees and nine *Clock Management Tiles* (CMTs)[14]. Each CMT provides two PPL-based[15] *Mixed-Mode Clock Managers* (MMCMs), which can take one input clock ranging from 10 to 800 MHz to generate multiple clocks with different frequencies and phase relationships. All clock regions have a height of 40 CLBs and are horizontally divided by a clock row (HROW) in their center. The HROW provides 12 horizontal spines of the global clock tree. Independently to the global clock tree each region provides six regional clock networks, which can be used to drive logic within the region as well as the region above and below. If no clock routing resources are left then regular routing resources are used to build clock trees with possibly noticeable clock skew. Due to its complexity the routing is handled by the implementation tools in order to guarantee the clock arrival at the *same* time. However, instantiation primitives are available if needed.

2.2.1.3 High Speed Input/Output Resources

Besides internal flexibility, FPGAs provide a highly customizable interface to communicate with the outside world supporting versatile *Input/Output* (IO) standards [94]. Therefore, it is equipped with 320 up to 1200 *Input/Output Blocks* (IOBs), each usable as either input or output pin. An IOB contains registers[16] and resources to translate internal voltage demand regarding a used IO standard [87]. This is highly beneficial as no additional external interface components on the

[14] one CMT per pair of left and right clock region
[15] PPL = Phase-Locked Loop
[16] supporting *Double Data Rate* (DDR) using two input, two output, two 3-state enable registers

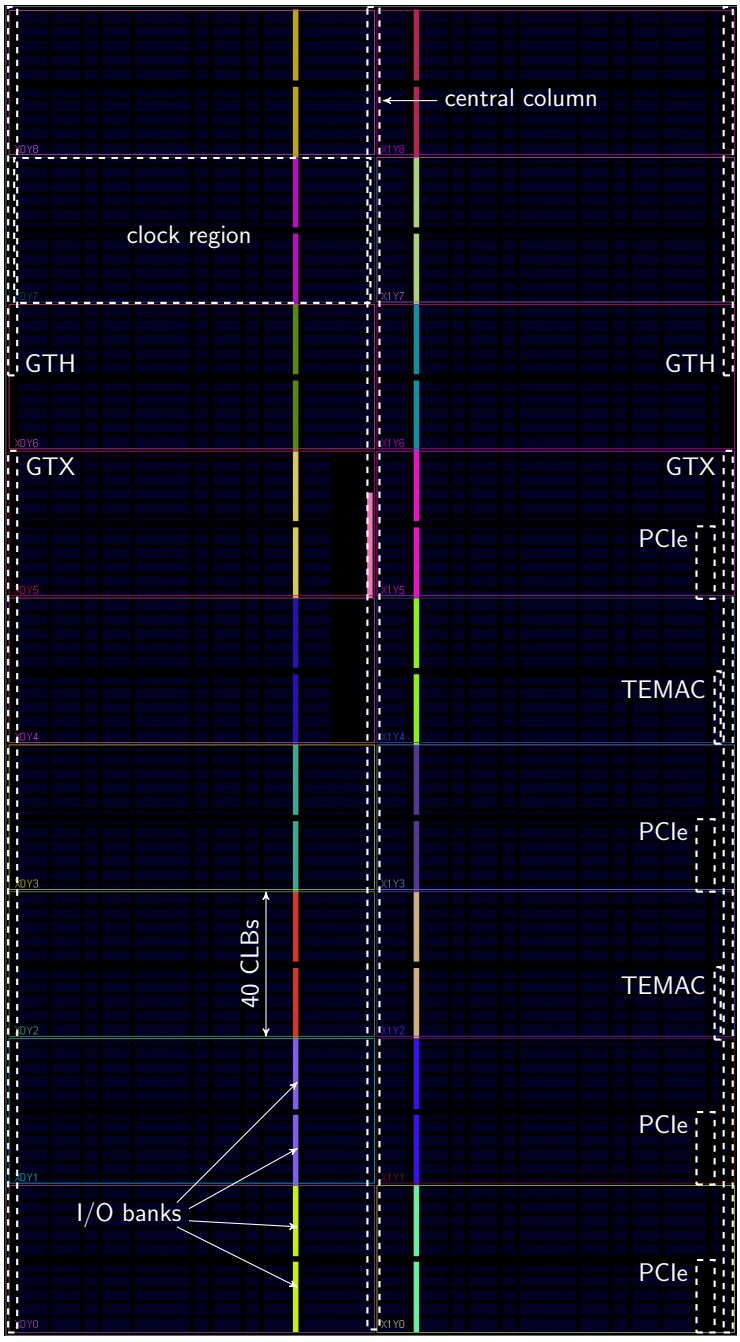


Figure 2.10 – Schematic view of the FPGA XC6VHX380T die obtained from PlanAhead [93].

board are needed for this purpose. Each 40 IOBs are grouped into an IO bank resulting in 8 up to 30 IO banks arranged in columns (see Figure 2.10). Note that some IO banking rules have to be followed regarding shared power pins for input respectively output when mixing different IO standards on IOBs in the same bank. When it comes to a specific FPGA board usually several pins are already assigned to dedicated components on the board such as buttons, LEDs, video or memory interfaces.

For high performance off-chip communications, between 8 to 72 *Gigabit Transceiver* (GTX) blocks provide a fast serial communication channel [95]. Each GTX combines independent transmitter and receiver using specialized on-chip circuitry to satisfy signal integrity requirements at high data rates (480 Mbit/s to 6.6 Gbit/s [87]). Additional GTH transceivers allow data rates of up to 10 Gbit/s [96]. Furthermore, up to four integrated interface blocks for PCIe designs are provided and can be utilized with the GTX [87, 97]. Multiple PCIe lanes can form a larger link (each up to 5.0 Gbit/s[17]) and be operated as *Endpoint* or *Root Port* to allow, e.g., host-to-FPGA and FPGA-to-FPGA communication [98]. The PCIe interface block covers the *physical*, *data link* and *transaction layer* [99]. However, the designer still has to take care of construction and interpretation of *Transaction Layer Packet* (TLP) packets in the FPGA fabric as well as the driver development on the possibly existing host. This itself is an error-prone task. Additionally, almost all Virtex-6 FPGAs provide 4 *Tri-Mode Ethernet MAC* (TEMAC) blocks implementing the *link layer* of the *Open Systems Interconnect* (OSI) protocol stack [87].

Besides these numerous interface options, some FPGAs include CPU hard cores such as ARM CPUs running at a much higher frequency than the equal implementation using FPGA resources [98]. Typically, they are used for tasks which are not beneficial for FPGA execution or management purposes like reconfiguration up to running whole operating systems.

2.2.2 Evaluation Platform

Throughout this work all FPGA related experiments are carried out on *Dini Group's* FPGA board *DNPCIe_10G_HXT_LL* [100] shown in Figure 2.11. The board is equipped with a *Xilinx* FPGA Virtex-6 XC6VHX380T-2FF1923. A resource overview of the FPGA is given in Table 2.1. Besides the FPGA, the board provides several memory interfaces such as two *Serial AT Attachment II* (SATA-II) ports for connection HDDs/*Solid State Drives* (SSDs), one DDR3 interface supporting up to 16 GByte (currently equipped with 4 GByte) and three independent

[17] PCIe base specification 2.0

Quad Data Rate II (QDRII+) SRAM memory channels (2x 144 Mbit, 1x 288 Mbit). Off-board communication is supported by three 10 *GbE* (Gigabit Ethernet) transceivers, one *Infiniband* channel, an 8-lane PCIe GEN-2[18] interface (16-lanes mechanical) and a *Universal Asynchronous Receiver Transmitter* (UART). However, not all provided components and interfaces are used in all performance evaluations and will be explicitly mentioned at corresponding locations in this work.

Figure 2.11 – FPGA board *DNPCIe_10G_HXT_LL* [100] used in this work.

The following section outlines how FPGA designers can create a configuration for an FPGA utilizing all these resources in an automated fashion.

2.2.3 Development Flow of FPGA Designs

In the previous sections we described the numerous kinds of available resources on the FPGA and the board. In the following sections the complete FPGA development flow will be presented. It consists of multiple stages assisted by vendor's implementation tools. The Xilinx *Integrated Synthesis Environment* (ISE)[19]

[18] theoretical bandwidth of 500 MByte/s per lane

[19] or Xilinx Vivado Design Suite for newer devices ≥ Ultrascale, Virtex-7, Kintex-7, Artix-7, and Zynq-7000

provides all necessary implementation tools for all translation steps and can either be used with a graphical user interface or script-based on the command line. Both extensively use the *Tool command language* (Tcl) [101, 102]. FPGA design is conceptually a top-down process, but implementation and verification follows a bottom-up fashion. This means after dividing the overall architecture into modules (or building blocks) and sub-modules, each can be implemented and verified easier. Afterwards, these lower-level modules are combined into a higher-level module and so on. The highest-level module is often referred as *top level module* (TOP) and typically consists only of instantiated modules and wires between them. Figure 2.12 presents the typical development stages of FPGA-based designs [103]. After an analysis and specification phase, the intended hardware design is *described* using a *Hardware Description Language* (HDL). The subsequent synthesis associates the described components with internal FPGA primitives, which afterwards are translated, mapped and placed to physical resources. During the routing process theses resources are connected. As a result a *bitstream* (or *bitfile*) is generated which is used as the FPGA configuration. The configuration is then loaded into the FPGA's configuration storage which basically determines the input-output relations of the LUTs and configures the interconnects. As a result an application-specific circuit is provided. During the whole process different simulation models assist the designer in testing and verifying the hardware design. In the following sections each implementation stage of the shown development flow and related properties are described.

2.2.3.1 Hardware Description Languages

Due to increasing capabilities, the amount of available transistors[20] and shorter development cycles, the hardware design can not be handled anymore by manually specifying complex functions at the gate level. Thus, each hardware design starts with an abstract module of the whole intended architecture with its corresponding input and output relations. In further steps, modules are divided into submodules, each with their appropriate input/output relations. In order to achieve a specific behavior of (sub-)modules, it has to instantiate other modules and/or needs to be equipped with a behavioral description. *Hardware Description Languages* (HDLs) enable hardware designers to provide *structural* and *behavioral* descriptions of hardware independently of the underlying technology [104]. Regarding ASIC and FPGA design, one commonly used HDL is *VHSIC Hardware Description Language* (VHDL)[21] [105]. Originally, VHDL was the result of the VHSIC initiative funded by the U.S. Department of Defense in the early 1980's.

[20] hundreds of million
[21] *Very High Speed Integrated Circuit* (VHSIC)

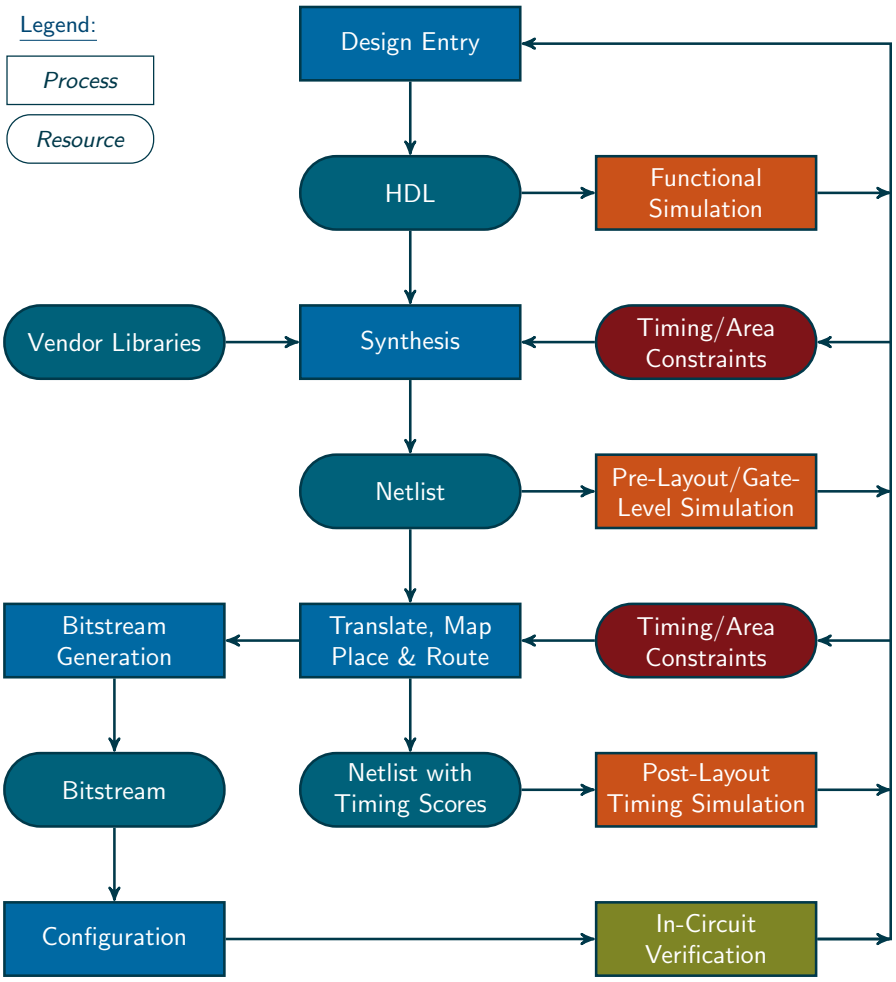


Figure 2.12 – FPGA development flow.

The main objective was to reduce efforts and costs caused by insufficient documentation and the usage of different tools, methods and languages from different suppliers, all resulting in numerous incompatibilities. The first standardized revision[22] in 1987 was developed by *Intermetrics*, *IBM* and *Texas Instruments*. In fact, firstly VHDL was created to describe *Integrated Circuits* (ICs) and to simulate these components [106]. This includes descriptions which might not have any counterpart in hardware to be mapped on. However, in 1988 *Synopsys* developed a *synthesis* tool which was able to *automatically* map a subset of the language constructs into a description at gate level. Further impulses were given by a growing

[22] IEEE 1076-1987

interest of *Computer Aided Engineering* (CAE) vendors, resulting in additional revisions and other related standards, with international involvement. The usage of a standardized HDL provides several advantages in many stages of the development cycle and can be summarized as follows [104, 107]:

- Implicit structural and behavioral documentation of complex systems by the HDL itself. Especially VHDL is a verbose and self-descriptive language.
- Simulation of behavioral description allows early verification of algorithms. No (error-prone) manual design and (performance-hungry) simulation on logic level is needed.
- Supports design on different abstraction levels. Translation from high-level to low-level description can be automatized.
- Simplified modularization supports parallel development of multiple designers on the same product.
- Simpler reuse of previously implemented and tested modules, which further reduces development time and costs.
- Widely used as an exchange format between different manufactures.
- Generally, HDL descriptions do not aim on a specific manufacturing technology and thus are portable to other technologies. However, this highly depends on the developer and the used language constructs to utilize resources.

Besides VHDL, the HDL Verilog [108] gained a wide prevalence and is mostly used in the USA. Basically, the expressiveness of both languages is comparable and most of the synthesis tools and simulators support both HDLs and even mixed designs. Since VHDL is exclusively used as a HDL in this work, we introduce the basic elements of VHDL in the following paragraphs. As described before each HDL-based design typically consists of structural and behavioral descriptions of modules.

The **entity** description defines the interface (input/output) of a hardware module without specifying the concrete behavior of the module. Listing 2.3 shows an example entity called myEntity. Each entity declaration consists of the components generic, port, type and constant. Generics are used to parametrize a module (e.g., used data width) and thus allow reuse of the same module with different parameters. Initial values are optional and can be overwritten during module instantiation but must be set before simulation respectively synthesis. Ports define the communication interface of the module and thus are sometimes referred as external signals. To each port a direction (in, out, inout, buffer) and a data type are assigned. The types std_logic and std_logic_vector are parts of the IEEE-1164 standard and correspond to one bit respectively a bit vector but provide a nine-valued logic for each bit. Choosing an appropriate data type and data width is crucial as each additional bit might cause increased logic utilization. In the type section user-defined types

Listing 2.3 – Entity example (VHDL).

```
1  entity myEntity is
2    Generic(
3      DATA_WIDTH : integer := 32
4    );
5    Port (
6      clk  :  in  std_logic;
7      sel  :  in  std_logic;
8      input1 :  in  std_logic_vector(DATA_WIDTH-1 downto 0);
9      input2 :  in  std_logic_vector(DATA_WIDTH-1 downto 0);
10     output1 :  out std_logic_vector(DATA_WIDTH-1 downto 0);
11     output2 :  out std_logic_vector(DATA_WIDTH-1 downto 0);
12   );
13   Type myType is Array (0 to 31) of integer;
14   Constant MAX_VALUE : integer := 255;
15 end myEntity;
```

can be declared, which in turn are assembled of basic data types. In the constant section constants can be defined to simplify code readability and refactoring as in usual software programming languages.

For a given entity the **architecture** description specifies the internal structure and behavior of a hardware module [109]. Each architecture is assigned to one entity, but each entity might have several architectures. This further assists the modularization and reuse of models as the internal structure and behavior of a module can be exchanged while the interface is kept. Listing 2.4 shows the example architecture myArch assigned to the entity myEntity. In the declaration head of each architecture (between the keywords architecture in line 1 and begin in line 5) additional internal definitions take place, e.g., signals (lines 3-4), constants and component declarations. Signals are the smallest unit carrying information regarding a specified type. The architecture body (between the keywords begin in line 5 and end in line 19) is composed of component instantiations including wires between them (structural description) and/or behavioral descriptions. Latter is differentiated into *concurrent statements* and *processes* (or *sequential statements*). Concurrent statements mostly describe selective assignments or combinatorial logic. For instance, the expression in line 7 in Listing 2.4 calculates the XOR of the input bit vectors input1 and input2, and assigns (indicated by <=) the result to the entities' output port output1. The concurrent statement in line 8 assigns the content of the internal signal temp1 to the output port output2 if the input port sel is set to '0', otherwise the content of the internal signal temp2 is assigned to output2.

Listing 2.4 – Architecture example (VHDL).

```
1  architecture myArch of myEntity is -- architecture head
2    -- declaration of signals,  constants,  types,  functions,  ...
3    signal  temp1 : std_logic_vector(DATA_WIDTH-1 downto 0);
4    signal  temp2 : std_logic_vector(DATA_WIDTH-1 downto 0);
5  begin -- architecture body
6    -- concurrent statements
7    output1 <= input1 xor input2;
8    output2 <= temp1 when (sel = '0') else temp2;
9    -- process  (sequential statement)
10   myProcess: process (clk) -- process name and sensitivity list
11     variable  sum : std_logic_vector(DATA_WIDTH-1 downto 0);
12   begin
13     if  rising_edge(clk)  then
14       sum := std_logic_vector(unsigned(input1) + unsigned(input2));
15       temp1 <= sum;
16       temp2 <= not input1;
17     end if;
18   end process;
19 end myArch;
```

Note that each change of the involved signals or the input *immediately*[23] effects at any time the content of the output signals in this example. Processes allow to describe behavior in a sequential manner (lines 10 to 18 in Listing 2.4). Typically processes are equipped with a *sensitivity list* consisting of signals, which *activate* the sequential execution when one of the signals changes[24]. After one iteration through the process the process is *suspended* until it is activated again. In the example, the process is always activated on a change of the clock signal clk. In the declaration part (line 11) additional process specific types, constants, *variables*, etc. can be defined. Variables (exclusively in processes) can have the same types as signals, but show a different behavior than signals in the sequential execution. The assignment of a value to a variable (indicated by :=) is directly available in subsequent statements of the current process iteration. Mostly variables are used to split and combine complex combinatorial calculations. Contrarily, an assignment to a signal changes the content of the signal at the end of the process. During the current process iteration the signal keeps its current value. Although the shown process is triggered at all signal changes (rising and falling) of clk, the actual calculation is done only on a rising edge in this example (line 13). The statement in line 14 sums up input1 and input2, and immediately stores the result in vari-

[23] with respect to a physical signal propagation time
[24] alternatively, there is a WAIT ON directive to model the same behavior

able sum. The stored value is assigned to signal temp1 in line 15. Note that this assignment does not change the content of temp1 immediately but at the end of the process. Each subsequent statement using temp1 in this process would work on the *old* content. In line 16 simply the inverted value of input1 is assigned to temp2. The values of temp1 respectively temp2 are used in the previously described statement in line 8. The actual resulting resources on the FPGA highly depend on the way how signals and variables are used. Although the assignment in line 14 seems to temporarily *store* the result in sum, in fact the implementation tools will directly assign the result to temp1, which in turn might not necessarily be a storing element but only a signal path. Typically, all signals assigned in clocked process are replaced with registers. If a variable is read before an assignment takes place in the current process iteration then for the variable a register is inferred. Thus, a variable stores the value between process iterations. However, variables have no direct equivalent to hardware and usually the same behavior can be expressed using signals exclusively. Processes, which are triggered by a clock signal, are called *synchronous* processes. In turn, processes, which do not depend on a clock signal, are called *asynchronous* processes. Besides clock signals, often reset signals are used to set modules respectively their internal signals to well defined states. Otherwise, the signals' contents can not be guaranteed. All existing concurrent statements, processes and instantiated components work concurrently and interact using the architectures signals. Figure 2.13 shows the corresponding block diagram described in Listing 2.4. On the left side all input signals are listed. The

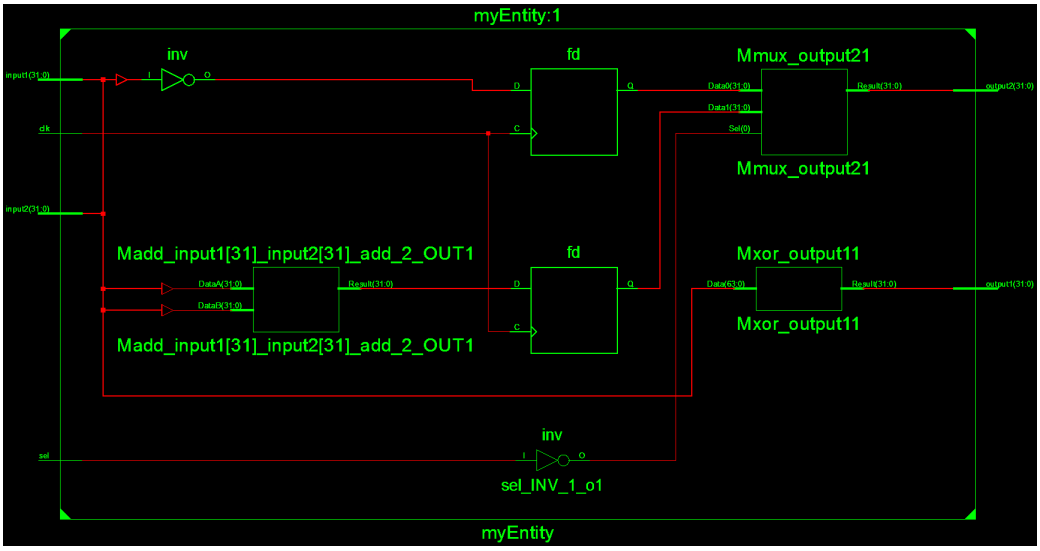


Figure 2.13 – Block diagram corresponding to Listing 2.4 at *Register Transfer Level* (RTL).

signal input1 is directed into an inverter and an adder. The result of the inverter is stored in a FF (fd). The second input of the adder is the input signal input2. The result of the adder is stored in another FF. After inverting the signal sel it is used as the input of a MUX which takes the intermediately stored values of the FFs and selects the corresponding output. Furthermore, both inputs input1 and input2 are processed by an XOR element. Thus, in this simple example the input is spread into different processing elements using dedicated signal paths and results are determined concurrently without interfering.

In Chapter 3 an *operator template* is described as an entity, which defines the common operator interface to be implemented by all query operators. This modularization allows a flexible interconnection of different operators without making assumptions about the preceding and succeeding operators. Furthermore, generics are used to parametrize different operators. Each operator's functionality is given in their architecture description.

Configurations are used to assign an architecture to an entity, as it is possible that for one entity multiple architectures exist [109]. If no configuration for an entity is given then the last translated architecture is used for this entity. Furthermore, parameters can be modified. Furthermore, **Packages** can be declared to further assist reuse within different projects or architectures [109]. Typically user-defined packages consist of new types, constants, functions and procedure declarations. Additionally, some predefined packages are provided, e.g., mathematical functions on standard data types. Besides newly written HDL, design reuse is assisted using so-called *Intellectual Property* (IP) cores. These cores are provided as either source code or netlists, which are mapped onto FPGA resources. Typically, FPGA vendors provide several cores for different applications such as memory interfaces or clock generators. Licensed and open source IP cores are available as well [110].

After the design is described in VHDL a long and processing intensive toolchain is applied on the description in order to get a suitable configuration file to configure the FPGA.

2.2.3.2 Constraints

Constraints are used to specify the design's performance standards in a *User Constraint File* (UCF) [107, 109]. It effects the FPGA toolchain in all development stages and impacts how the design is placed within the FPGA. Besides constraining timing specification (clock rate respectively signal propagation delay between storing elements), it is possible to manually specify mapping and block placement of components described in the HDL design. This is extensively used to assign ports

and signals to physical pins, which are connected to resources outside the FPGA die. A full list of constraints and attributes for Xilinx FPGAs is given in [111]. Over-constraining might result in an excessive replication in synthesis and failure of the whole translation process, but might also obtain better performance characteristics [107].

2.2.3.3 Synthesis

Synthesis is the first phase in which the HDL design is associated with internal FPGA primitives [112, 113]. Regarding Xilinx FPGAs the synthesis tool *Xilinx Synthesis Technology* (XST) [109] is provided in the Xilinx ISE Design Suite, but other (manufacturer independent) tools are available as well.

Before the actual synthesis, the HDL sources are parsed and syntax checks are applied. Afterwards, all found entities and modules are elaborated with their specified generics and the overall design hierarchy is recognized. In the following HDL synthesis, for each module/unit an element association takes place inferring basic elements (*primitives* and *macros*[25]) such as adders/subtractors, comparators, registers/BRAMs, multiplexers and so on[26]. During this process unconnected or unused ports and signals are detected. Additionally, *Finite State Machines* (FSMs) and RAM as well as clock and reset signals are detected. In the *advanced* synthesis, the previously detected basic macros are combined to larger macro blocks such as counters, pipelined multipliers or accumulation functions [109, p. 468]). Furthermore, for inferred RAM it is distinguished to implement it using LUTs or internal BRAM. In the *low level* synthesis the encoding for FSMs is set, a detection and removal of equivalent FFs/Latches are performed as well as optimizations of constant FF are applied. As a result a pre-optimized netlist using only technology-specific components of the targeted FPGA is provided. The process is summarized in reports about utilized basic elements, e.g. LUT, MUX, FF, BRAM, shift registers, clock and IO buffers and others such as MMCM or PCIe cores. Additionally, a first timing report on detected clock signals based on estimations associated with the targeted FPGA is given. At this and later stages the *critical timing path* becomes crucial to decide whether a performance constraint t_c is met or not. The critical timing path represents the longest propagation t_p between two storing elements. If $t_p > t_c$ then the constraint is not met. This can be solved by either weakening the constraint or shortening the path by introducing other storing elements in the path. Detection of clock signals is crucial as only then the dedicated clocking resources are used for routing these signals. However, the actual timing and thus the

[25] elements which are too complex to instantiate by just using the primitives [114, p. 3]
[26] a full list of all Virtex-6 primitives and macros is given in [114]

achievable clock frequency will be affected by later processes of the design flow. Depending on the report the developer can revise the HDL design in some aspects or adjust constraints at an early stage of the whole toolchain. The RTL view gives a hierarchical view of modules and submodules using generic symbols like adders or AND gates and is typically manufacturer independent. The technology view in turn shows the design using basic elements available on the targeted FPGA. Note that no board layout or interface signals are considered at this stage.

Synthesis tools typically infer resources regarding the chosen platform. Inference should be used to keep HDL code portable and maintainable. However, sometimes it is difficult to infer resources exactly as demanded. Therefore, a list of primitives is available to manually instantiate specific resources [114]. Furthermore, the code is not necessarily translated into what the designer would think. In fact the design tools heavily optimize based on synthesis constraints and tool options. The way of coding significantly impacts the design performance and thus requires serious design considerations at any time.

2.2.3.4 Translate, Map, Place & Route (PAR)

After the synthesis of one or more netlists, the *implementation phase* takes place. First the provided netlists are merged by the *translate* process. The succeeding *map* process fits the given design into a *Native Circuit Description* (NCD) targeting the physical resources (CLB, IOB, etc.) of the specified FPGA [103]. If enabled, the map process already places the design. Furthermore, unused logic is detected and removed. The *Place & Route* (PAR) tool maps the components on physical resources and logical connections to physically routing resources (switching matrices) with respect to timing constraints (*timing-driven*). If no timing constraint needs to be satisfied then a cost-based (*non timing-driven*) approach is applied. Cost tables are used to assign weighted values to relevant factors such as routing resources and connection lengths [115].

2.2.3.5 Bitstream Generation and Configuration

The *Bitstream Generator* (BitGen) produces a *configuration* (called *bitstream* or *bitfile*) from a given fully routed circuit. Using the *iMPACT* configuration tool [103], the binary data is then transferred into the configuration memory (LUTs, PIPs) of the FPGA and consequently defines the internal logic and interconnections. As the configuration memory is based on volatile SRAM the configuration gets lost on power shutdown. Thus, typically the FPGA board is equipped with a non-volatile

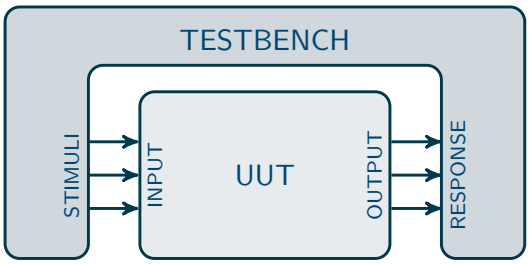


Figure 2.14 – Testbench verifies *Unit Under Test* (UUT) by providing input stimuli and evaluating the UUT's response.

flash memory holding the configuration. On startup the configuration is loaded from the flash into the FPGA.

Due to the enormous complexity of all involved tools each of them can be adjusted by several optimization flags with respect to timing, area and effort. Note that applying the whole toolchain on a HDL design takes at least several minutes for simple designs and up to hours or days for complex architectures.

2.2.3.6 Simulation

As shown in Figure 2.12 each development stage can be verified by simulating the design. Typically, a *testbench* module is used to simulate single modules or the whole design (referred as *Unit Under Test* (UUT)). The testbench itself is HDL code which instantiates components (given as code or netlist), initializes the design and generates the stimuli such as clock or input data (see Figure 2.14). Furthermore, usually the output is verified by comparing it with an expected result given by the designer (*golden vector* or self-checking testbench) [116]. *Functional* (or *behavioral*) simulation uses directly the HDL sources prior to the synthesis. Thus, no information about the implementation in terms of actual used physical resources are available and no assessments regarding timing can be made. However, due to the early stage in the whole development flow, the functional simulation should be frequently used for possibly all (sub-)modules to verify proper behavior. *Pre-Layout/Gate-Level* (or *Post-Synthesis*) simulation is used to verify the design based on the design's netlist after the synthesis. Although there is still no knowledge about the actual routing, the netlist contains information about the utilized hardware resources. Thus, the timing can be simulated with higher accuracy than in the functional simulation. *Post-Layout/Timing* (or *Post-P&R*) simulation uses the full knowledge about actual used resources, their location on the FPGA and the routing between them and thus provides the most accurate timing model. However,

writing a comprehensive testbench as well as executing the simulation at this stage is very time consuming. As a matter of fact, this simulation is least frequently used, because the design is verified in earlier stages and typically is adjusted to meet the constraints.

2.2.4 Dynamic Partial Reconfiguration

Initially, FPGAs have been used as *glue logic* or simple ASIC replacements for one given computation task [117]. Thus, slow configuration of the entire chip was sufficient. Nowadays, a single computation might be broken into multiple configurations, which need to be reconfigured during execution (*time-multiplexed hardware*). But some of design's components may be required during the whole process, e.g., communication interfaces. Furthermore, *Dynamic Partial Reconfiguration* (DPR) enables the modification of an operating FPGA design by loading a partial bitfile. Figure 2.15 shows the basic concept of DPR. The FPGA design is divided into *static logic* (or *top-level logic*) and one (or more) *Reconfigurable Partitions* (RPs) [118]. For each partition several *Reconfigurable Modules* (RMs) in terms of partial bitfiles may be provided (e.g., A1.bit, A2.bit, A3.bit in Figure 2.15). Although each module may provide another functionality, all of them must use common *partition pins* connecting the RP with the static logic. Therefore, *proxy logic* is automatically inserted by the implementation tools as fixed and known interface point [118].

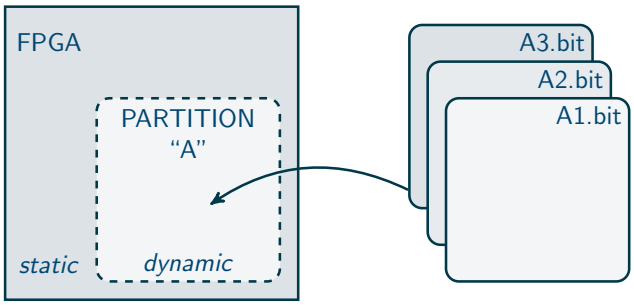


Figure 2.15 – Basic concept of *Dynamic Partial Reconfiguration* (DPR): Resources are divided into static logic and one or more *Reconfigurable Partitions* (RPs). Multiple configurations may be provided for each RP as *Reconfigurable Modules* (RMs) – adapted from [118].

The design flow slightly differs from the approach without DPR. Typically, separate projects for the static logic and each RM are created. In the static logic the RP is denoted as *black box*. Thus, the component is declared with its input and output relations but without specifying an underlying architecture. The netlists for the static logic and each RM are synthesized independently [118]. The following

partial reconfiguration design flow is graphically assisted by the application *PlanAhead* [93, 119]. For each RP an *Area Group Constraint* needs to be specified. The constraint assigns FPGA resources (slices, BRAM, etc.) to the RP using a global label. Thus, these resources are not available for static logic or any other RP. Additionally, note that optimizations cannot take place across partition boundaries and thus partitions can significantly effect timing, utilization and runtime [120]. In a first configuration (or *design run*) the implementation of the static logic, including placement and routing, is *promoted* (or exported). All subsequent design runs import the static logic from the promoted configuration and thus the static logic is identical in all design runs. In all design runs to each partition a RM is assigned or remains empty as a black box. The succeeding bitfile generation delivers a full configuration bitfile as well as partial bitfiles for each RM in the selected design run. Partial bitfiles of different design runs can be used in a mixed fashion as long the design runs share the common static logic. The partial bitfiles contain all configuration commands and data which is needed for the DPR [118, p. 93]. Besides an external configuration port, an *Internal Configuration Access Port* (ICAP) enables the configuration from within the static logic [121]. For instance, a minimal static design might consist only of the logic needed to provide an operational PCIe interface. After system startup, partial bitfiles can be transmitted through this interface, which then are configured into the reconfigurable partition(s) using the ICAP [122, 123]. Furthermore, a set of partial bitfiles might be stored in one of the FPGAs respectively the board's memory and can be loaded from there later. During partial reconfiguration static logic and all other reconfigurable partitions in the device continue to operate. Papadimitriou *et al.* [124] present a survey on performance and capabilities of partial reconfiguration. Koch *et al.* [125, 126] provide some use cases, which motivate the practical relevance of DPR in industrial systems.

In this work, DPR is used to divide the chip area into static logic, consisting of communication interfaces as well as the ICAP module, and one or more RPs. In Chapter 4 one comparably large RP is dynamically reconfigured with various RMs. Each RM is assembled using VHDL modules during system runtime with respect to a given query. In Chapter 5 we divide the chip area into multiple small RPs, each large enough to provide logic for one query operator at a time. Therefore, the static logic additionally consists of a semi-static structure of a general operator graph connecting the RPs.

2.2.5 Applications

Typically, FPGAs are used in application areas with a high demand on processing power and low energy consumption. In this section general application areas of FPGAs are outlined. Related work regarding usage of FPGAs in database tasks is presented in detail in Section 2.3.

First of all, the availability of HDLs and FPGAs opens doors for a broader range of learners to design and prototype hardware in a rapid and cost efficient fashion. Due to their reconfiguring capabilities they are more and more used as replacement for ASICs which can not be adapted after manufacturing to the rapidly changing algorithms and protocols [127]. Furthermore, the fast time to market cycle for new products makes them popular in many fields.

In the automotive industry several applications can be found. Nowadays cars are equipped with numerous sensors and actors possibly from different vendors providing different interfaces. FPGAs provide a high amount and variety of connectivity pins to attach these devices. Adjustable internal logic resources can provide several controllers to interconnect different subsystems. Besides being only a connector of different subsystems, the FPGA suits well in many applications with high requirements on information quality and real-time response to prevent serious accidents [128, pp. 149–159]. FPGA-based driver information and assistance solutions enable (without claim to completeness): real-time stereo vision and input image analytics (object and motion detection, traffic sign recognition, collision warning, pedestrian detection), surround view at high resolution and frame rate (including distortion correction, stitching).

In the rapidly evolving consumer electronics market continuously new features are added to many different device categories such as TVs, portable and networking devices [129, 130]. Manufactures are under competitive time pressure to handle continuous changes. Contrary to ASICs, FPGAs provide flexibility and low development costs to quickly adapt to market trends.

Further tasks applicable in many areas cover digital signal processing in real-time such as filter or fast Fourier transformation, compression of images, e.g. satellite pictures [131], audio or video (exchanging compression algorithm codec [128, pp. 336–341]).

Regarding *Wireless Sensor Networks* (WSN), Bellis *et al.* [132] propose sensor nodes providing a low power FPGA layer. Typically sensor nodes have limited processing capabilities and thus intensive processing tasks are performed *outside* the WSN. Therefore, collected data needs to be transferred via radio interfaces which obviously consume energy. But power consumption is a serious concern in

these deployments as battery storage of the nodes is limited and directly connected to the WSN lifetime. The FPGA is used to perform (digital signal) processing on the node to ultimately save overall power consumption.

In cryptography enormous performance benefits have been reported [133] and recently machine learning approaches, such as deep convolutional neural networks used for image classification [134, 135, 136, 137, 138], have been adapted to FPGAs.

All these and new FPGA applications will get a further boost as the CPU vendor *Intel* has bought the FPGA specialist *Altera* and presented a package including the server CPU *Xeon E5* and an FPGA [139]. Weisz *et al.* [140] study the impact of shared-memory CPU-FPGA systems on the example of large pointer-based data structures – more precisely linked lists. The authors conclude that traversing a single list with non-sequential node layout and a small payload size is least efficient on the FPGA due to memory latency. However, interleaving multiple linked list traversals overcomes these stalls and significantly enhances traversal performance.

2.3 Related Work

In the early days of databases in the late 1970's, first ideas occurred to increase the performance of database operations through the use of specialized hardware components. Leilich *et al.* [141] outlined a tailor-made search processor particularly to improve selection and restriction operations. The DIRECT architecture [142] considered multiple query processors dynamically assigned to a query to support intra-query and inter-query parallelism. Each query processor is equipped with an instruction set optimized to perform relational database operations efficiently. An interconnection matrix using cross-point switches enables the query processors to rapidly switch between shared memories. However, the centralized control for the parallel executions and the shared memory architecture limited the scalability. Although most projects appeared promising initially, almost none of them became commercially available [143]. As a result the *Gamma* database machine [144] employed a *shared-nothing* architecture which can be seen rather as a parallelized database than a hardware accelerator. Although not exploiting any parallelism, the *Intelligent Database Machine* (IDM) [145] appears to be the first widely used and most commercially successful product. However, at this time the technological capabilities were limited and thus it was simply easier to develop a database machine on paper than to assemble and evaluate a working prototype [146]. Thus, most performance gains were driven by speed enhancement in general-purpose CPUs and optimized data structures which address the characteristics of the underlying

hardware architecture [14, 73, 82, 83, 147, 148, 149, 150, 151]. Nowadays, since clock frequency growth reaches its limits and the availability of new hardware architectures, these and new ideas get a fresh impetus [80, 152]. In the following sections, we review recent work using modern hardware architectures in the field of database tasks, categorized into in-storage processing, general-purpose computing on GPUs and reconfigurable computing. Additionally, we discuss specifically related work in the remaining chapters and sections throughout this work.

2.3.1 In-storage Processing

While CPU performance steadily increased for the last decades, the latency of main-memory access turned out to be a bottleneck known as the *memory wall* [83, 148]. This imbalance exacerbates when it comes to persistent storages such as disk-based HDDs [153]. Although the latency is significantly reduced by flash-based SSDs, the disk bandwidth remains considerably low. With respect to databases, typically huge amounts of data are transferred to the CPU but often most of the data is irrelevant or is only a part of an aggregation. Already in 1998, Keeton [154] *et al.* introduced the concept of *intelligent disks* (IDISK) to offload data processing from the host's CPU to the low-power processors on disks. However, due to technological limitations it was never completely prototyped. Similarly, Riedel *et al.* [155, 156] as well as Acharya *et al.* [157] investigate the utilization of *active disks* from architectural and software-technical perspectives.

Nowadays, *Smart SSDs* provide an embedded processor, fast internal memory and multiple IO channels (16+) with faster interfaces resulting in a high performance on concurrent access patterns [158]. In order to estimate performance and energy consumption, Cho *et al.* [159] introduce methods to model the benefits of *intelligent SSDs*. The validation in a simulated environment assists the accuracy of the analytical model. Additionally, a table scan was evaluated on a prototype SSD. The scan includes the selection of 1 % of all records (1 GByte) and the projection of 4 bytes out of 150 bytes for each selected record. Overall a speedup of 2.3X and an energy improvement of 5.2X is reported. The table scan operation is not computing-intensive and thus the performance improvements are gained mostly by data transfer reduction. Comparably, Do *et al.* [160] motivate the use of smart SSDs in database applications. Therefore, they present a runtime framework to run user-defined applications in smart SSDs. The proposed system is evaluated using queries with selection, aggregation and range checks. Although significant speedups (>2.7X) are obtained in many cases, the performance degrades if selectivity is low and thus much data has to be processed by the low-performance processor. In these cases the regular (non-smart) SSD performs better. Seshadri *et al.* [161]

present the *Willow system*, a prototype which allows to implement application-specific data access utilizing the internal SSD processing resources. In fact, the prototype is implemented on an FPGA-based prototyping system. It consists of four FPGAs, each hosting two *storage processing units* which are equipped with 8 GByte DDR2 RAM (64 GByte total). However, only IO operators are investigated reporting latency and bandwidth improvements.

2.3.2 General Purpose Computing on Graphics Processing Units

Graphics Processing Units (GPUs) are specialized hardware accelerators to create and manipulate images stored in memory (frame buffer) to be shown on a display. Their internal structure is highly parallel which makes them interesting also for other fields than image processing. The methodology of performing general computations on GPUs is referred as *General Purpose Computing on Graphics Processing Units* (GPGPU). In order to efficiently utilize the huge amount of simple processing unit, the general computation and its data need to be migrated into a *graphical form*. Furthermore, GPUs follow a stream-computing model and thus are restricted regarding random access writes.

Geodatabases

Sun *et al.* [162] show how to turn these characteristics into an hardware accelerator for querying spatial databases (geodatabases) used in geographical information systems and computer-aided design (CAD) systems. Basically, first a set of candidate objects is determined in a filtering step using indexes and minimum bound rectangles. For each of these candidates the actual geometry is compared to the query geometry (*refinement*). Latter is computational intensive especially for complex geometries such as polygons. Therefore, the authors propose techniques to accelerate spatial selection and joins. The authors achieve speedups of up to 5.9X but also report hardware limitations due to the stream-computing model.

Govindaraju *et al.* [163] give an insight into the graphics architectural pipeline consisting of various units, each designed to perform one specific operation efficiently. One of these units, the *pixel processing engine*, provides a small and simple set of test functions to be performed on *fragments*. The engines are programmable by providing a *fragment program*. The authors show how to perform predicate evaluations, semi-linear queries[27], boolean combinations of simple predicates, range

[27] commonly used in spatial database and geodatabases

queries and aggregations (*count*, *min*/*max*, *sum*, *avg*). In the performance benchmark TCP/IP data for monitoring traffic patterns is stored in floating point numbers. Each record consists of four attributes. In case of semi-linear queries the authors observed an order of magnitude speedup over the optimized CPU-based implementations running on a dual 2.8 GHz Xeon CPU. Although several of the presented algorithms are not able to use all capabilities of the GPU and the GPU uses a relatively low clock rate[28], speedups of 2X up to 4X are achieved. In some case the GPU-based approach is slower than the algorithms running on the CPU, e.g. due to missing integer arithmetic instructions, accumulations are cumbersome. In succeeding work, Govindaraju *et al.* [164] utilize a GPU as a co-processor for solving sorting tasks on billion-record files. They map a bitonic sorting network to GPU rasterization operations and use the GPU's programmable hardware and high bandwidth memory interface to minimize data transfers between the CPU and the GPU. In order to efficiently utilize the GPU resources the bytes of the keys are represented as 32-bit floating point 2D arrays. The evaluation shows that the overall sorting performance of a mid-range GPU is comparable to that of a algorithm running on an expensive CPU.

Relational Databases

Further enhancements in GPU architecture, resulting in hundreds of very fast processing units, and the CUDA [165] programming model increased performance and programmability of GPUs. He *et al.* provide extensive work on relational query co-processing using GPUs. In [166], they investigate GPU-based join execution. Although speedups of 2X up to 7X for joins are reported, the authors highlight that the speedup is far behind the higher clock rates and memory bandwidths of the GPU compared to the CPU. In [167], they extend their system to an in-memory relational query co-processing system and present a cost model for estimating processing time on the GPU. Additionally, the system is able to dynamically assign operations either to be processed on the CPU or GPU taking into account whether the input data is already located in the GPU's in-device memory or is in the main memory. In the evaluation using two queries from the TPC-H benchmark [168] the proposed system achieves speedups of 13.8X on small datasets and 3.5X on larger datasets which do not fit into main memory. The authors highlight that mostly the IO time degrades the performance. Furthermore, they outline that the data transfer between the GPU's in-device memory and the main memory can become a performance bottleneck for GPGPU applications. Due to the cost of data transfer, the processing of simple queries on the GPU can be much slower than

[28] GeForce FX 5900 Ultra with clock rate 450 MHz; nowadays GPUs with clock rates >1.7 GHz are available

the CPU-based algorithms. The significant impact of transferring data from the main memory to the GPU's device memory and vice versa is reduced by Fang *et al.* [169]. Therefore, nine lightweight and cascadable compression schemes on GPUs are proposed and assembled by a compression planner in order to find the optimal combination. As a result the data transfer overhead can be reduced by up to 90 % and the overall query performance is improved by up to 10X.

The concept of transactions enables databases to concurrently execute read and write access while ensuring the *ACID* properties[29]. Thus, fast transaction processing is a cornerstone in online transaction processing (OLTP) applications in which thousands of transactions can be issued in a short time frame. He *et al.* [170] propose the *bulk execution model* to group multiple transactions into a bulk to be executed on the GPU as one task. Due to the variety of possible commands in one transaction, the author's approach is limited to a static set of procedures. Furthermore, the approach is limited to databases fitting in the GPU's memory (4 GByte). However, the evaluation show achievements in terms of speedups ranging from 4X up to 10X compared to a high-end quad-core CPU. Although the price of the GPU is three times higher than the CPU, the proposed system achieves a higher throughput per dollar ratio than the CPU-based implementation.

Besides dedicated GPU's, there are processor designs which integrate the CPU and GPU in a single chip. As a drawback, the memory bandwidth of the integrated GPU is significantly lower than for a discrete GPU. On the other hand, the *last level cache* is shared between CPU and GPU, and thus can be used for efficient co-processing. He *et al.* [171] propose methods for *in-cache* query co-processing for these coupled CPU-GPU architectures. Basically, CPU-assisted prefetching hides the memory latency and minimizes the cache misses of the GPU. Furthermore, the architecture allows a variation of core assignments (CPU and GPU) to execute decompression or query evaluation tasks. In experiments the authors show that the overall performance of TPC-H [168] queries can be improved by 1.4X.

Cheng *et al.* [172] experimentally evaluate the power consumption and performance of embedded systems which are equipped with a GPU. While selection and aggregation are executed faster and show lower energy consumption on the CPU, more complex operations such as sort and hash joins benefit from the GPU. Especially, simple operations are mostly stalled by memory transfers to the GPU. In comparison to a workstation the embedded system achieves a 15 % higher energy efficiency if the dataset is small enough to fit into the memory of the embedded device. Plain execution times of the embedded systems in comparison to the workstation are not reported. The authors sketch a cluster setting consisting of multiple embedded devices in order to efficiently process larger datasets.

[29] atomicity, consistency, isolation, durability [14, p. 165]

Hardware-Oblivious Database Designs

Obviously, all these approaches require a wide knowledge about the underlying hardware. Therefore, Heimel *et al.* [151] present a hardware-oblivious database design which allows to implement operators in an abstract fashion without relying on a particular hardware architecture. The proposed framework is able to transparently run the implemented operators either on a multi-core CPU or a GPU. In most cases the approach shows a competitive (and in some cases superior) performance to hand-tuned parallel database operators. The authors mention that there were multiple cases for which their approach was outperformed. However, the approach shows the opportunities of such a system and might further drive the adoption and acceptance of specialized hardware such as GPUs or FPGAs. Breß [173] further motivates the need of hybrid query processing engines based on GPUs. Due to increasing optimization space (*execute operator on GPU or CPU*) the challenge is how to make scheduling decisions with respect to heterogeneous architectures. These scheduling algorithms have to consider that changing a processing unit might imply additional data transfers. Therefore, the authors propose an alternative database design which learns the cost models without detailed knowledge about the underlying hardware during query processing [174]. Additionally to the cost models, data locality and load conditions across (co-)processors are used in query optimization. The in-memory DBMS CoGaDB [175] shows the feasibility of this approach but reveals a data transfer bottleneck [175]. A comprehensive analysis on the overhead introduced by memory transfers is given by Gregg and Hazelwood [176].

Indexing

Recently, Shahvarani *et al.* [177] propose a heterogeneous computing platform to accelerate lookups in B^+-trees. Before the tree can be used it is preprocessed into two parts, one resides in the GPU's memory and the other part in the host's main memory. Assuming the system throughput is bound by the CPU, all inner nodes of the tree are processed by GPU, while the CPU is used only for searching in leaf nodes. In order to address arbitrary GPU-to-CPU computation power ratio, the authors present a load balancing scheme which utilizes the CPU to traverse upper nodes. If a certain level is reached then the inner node of the index is transferred to the GPU which resumes the search in the remaining levels. In the evaluation the authors report an average speedup of 2.4X and prove the feasibility of their load balancing methodology.

In this work, we use a reconfigurable FPGA which is attached via PCIe and thus in our approach the data transfer might become a performance bottleneck. However, in Chapter 4 we will analyze this circumstance and show that the PCIe interface is not the bottleneck of our hybrid architecture.

2.3.3 Reconfigurable Computing

Reconfigurable architectures provide the post-fabrication programmability of software and the spatial parallelism of hardware [178]. Due to the availability of FPGAs reconfigurable computing becomes more and more attractive and affordable. Typically, these devices obtain their performance advantages rather by inherent parallelism than by high clock frequencies. Thus, algorithms mostly can not be adapted in a straight-forward fashion to beneficially utilize FPGAs. In the following paragraphs we review recent work utilizing FPGAs in database applications.

In data mining applications, the Apriori [179] algorithm is a popular correlation-based method. As it is computationally expensive, Baker *et al.* [180] present a scalable implementation on a FPGA. The architecture consists of a cyclic pipeline. The initial items are fed into a *candidate generator*[30] which in turn forwards the candidates in a *pruning* unit which forwards its result to the *support calculation*. These resulting items are fed back to the head of the pipeline and the process is repeated until the final candidate set is found. Especially the last step, the support calculation is the most time consuming and data intensive part. But the sequential algorithm consists of two loops with no dependencies and thus is highly suitable for parallelization. In simulations on datasets containing 100,000 items the authors report speedups of at least 4X compared to software solutions running on a dual-core Xeon.

Shan *et al.* [181] present an FPGA approach supporting the parallel programming framework *MapReduce* [182]. MapReduce is divided into two phases. The *map* function processes each input tuple (*key*,*value*) and generates a set of *intermediate* tuples. After grouping the intermediate tuples with respect to the key, the *reduce* function is called for each group. The calculations in the map phase as well as in the reduce phase are independent from each other and thus both phases suit well for parallelization. The authors provide a framework to place various *mappers*/*reducers* on the FPGA and deploy *RankBoost* [183] as a case study on their architecture. With an increasing number of up to 146 mappers, the achievable speedup increases of up to 31.1X. Although a PCIe interface is instantiated these results are obtained in simulations only.

[30] one generation of candidates is built into the next generation

Mueller *et al.* [184] implement an FPGA-based median operator for data streams consisting of 32 bit values. In a first step the data over a count-based sliding window is sorted using sorting networks (*bitonic merge*, *odd-even merge*). As for the median operator a fully sorted data sequence is more than required, the presented sorting networks are further optimized in terms of resource consumption by removing some swap elements. The introduction of FIFO queues decouples the executions of the median operator and the CPU and avoids the need of explicit synchronization. However, processing larger amounts of data ($\geq$ 4 KByte) benefits from utilizing a *Direct Memory Access* (DMA) controller in the FPGA fabric. The instantiation of up to four median operators on a Virtex-II Pro allows the processing of multiple streams without interfering each other and thus significantly improving performance and energy consumption compared to CPU systems.

Koch and Torresen [185] analyze different hardware sorting architectures to develop scalable sorters for large datasets. As a result they propose a combination of a FIFO-based and a tree-based merge sorter. DPR is used in order to save almost half of the FPGA resources or to improve processing performance. The whole chip area can be used to execute the FIFO-based merge sorter which stores the intermediate result in external memory. After reconfiguring the FPGA with the tree-based merge sorter it reads the intermediate results from the same external memory. Another approach uses two memory channels and uses half of the resources for the FIFO-based merge sorter and the remaining resources for the tree-based merge sorter. During the FIFO-based merge sorter stores the intermediate result in external memory, the tree-based merge sorter concurrently reads those and writes its result in the second external memory. After a reconfiguration also the first half is used as tree-based merge sorter to enable larger problem sizes (up to 20 GByte). Due to a poor IO performance of the system, it is evaluated using *IO emulation modules* to test the sorter at full speed. In this case the authors report a sorting throughput of 1 GByte/s (speedup of 1.4X compared to a GPU-based approach). On top of this work, Casper *et al.* [186] build new designs that make use of a modern multi-FPGA prototyping system with a large amount of memory capacity and bandwidth (four large FPGAs each equipped with 24 GByte RAM at a line speed of 38.4 GByte/s). They explore accelerating in-memory database operations with focus on throughput and utilization of memory bandwidth during sorting rather than determining the performance of the design. The evaluation shows substantial improvements in both absolute throughput and utilization of memory bandwidth. Additionally, the presented system performs an equi-join after sorting of two tables by utilizing two FPGAs for sorting and one FPGA for merging (sort merge join). However, the architecture supports only one join operator and not arbitrary queries.

Heinrich *et al.* [65] propose a hybrid index structure which stores the higher levels of a B^+-tree including the root on an FPGA. The lower levels including the leaves

are located on the host system. As a result, the access on frequently entered higher levels is accelerated. The FPGA returns an entry point (address) from where the software system continues the search.

Additional related work will be reviewed separately in the following chapters.

3 Query Operators on Field-Programmable Gate Arrays

In this chapter first a top level view on the intended hybrid query engine is given. Afterwards, following the typically bottom-up design flow, an operator template is introduced as the basis for a flexible system. It enables the transparent composition of multiple operators using a common interface. On the example of the join and filter operator the benefits compared to a software-only solution are experimentally shown in micro benchmarks. Additionally, all implemented operators of this work are outlined.

3.1 Hardware Acceleration for LUPOSDATE

The approach used in this work to manage and query RDF data is based on the *Semantic Web* (SW) database system LUPOSDATE [56, 57]. Figure 3.1 recaps the functionalities and query processing phases of LUPOSDATE (see Figure 2.5 in Section 2.1.3 for a detailed introduction). After parsing the SPARQL query, redundant language constructs are eliminated to simplify the following processes. Afterwards, an operator graph is generated in order to apply logical and physical optimizations. The logical optimization reorganizes the operator graph into an equivalent operator graph. It generates the same output for any input as the original operator graph, but needs less execution time to evaluate the query. The physical optimization aims to obtain the operator graph with the best estimated execution times by choosing a concrete implementation (physical operator) for each logical operator resulting in the *Query Execution Plan* (QEP). The QEP is executed by using formerly generated indices to retrieve the query's result. In this work, we extend the execution model by an additional layer which dynamically maps the QEP onto FPGA resources in order to provide a query-specific hardware accelerator.

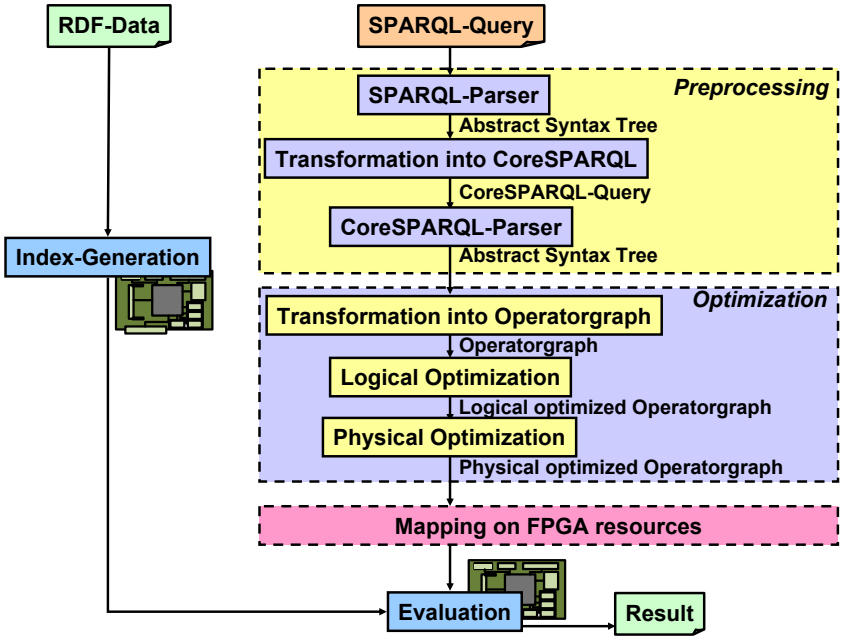


Figure 3.1 – Stages of query processing in the LUPOSDATE system. In this work, we introduce an additional stage which dynamically maps the query structure onto an FPGA to obtain a query-specific hardware accelerator.

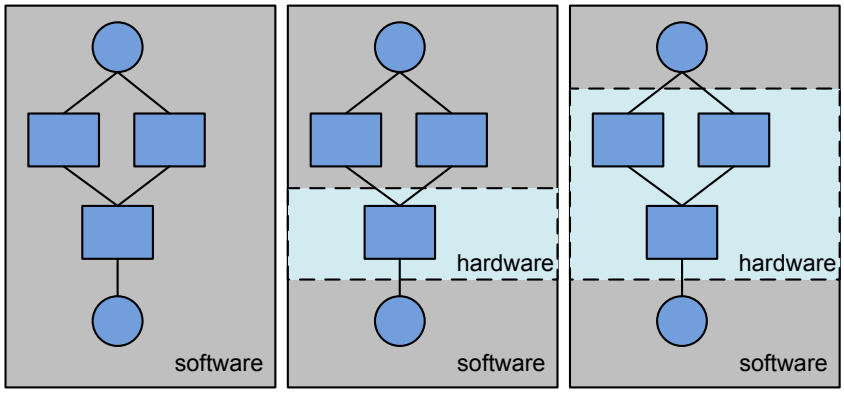
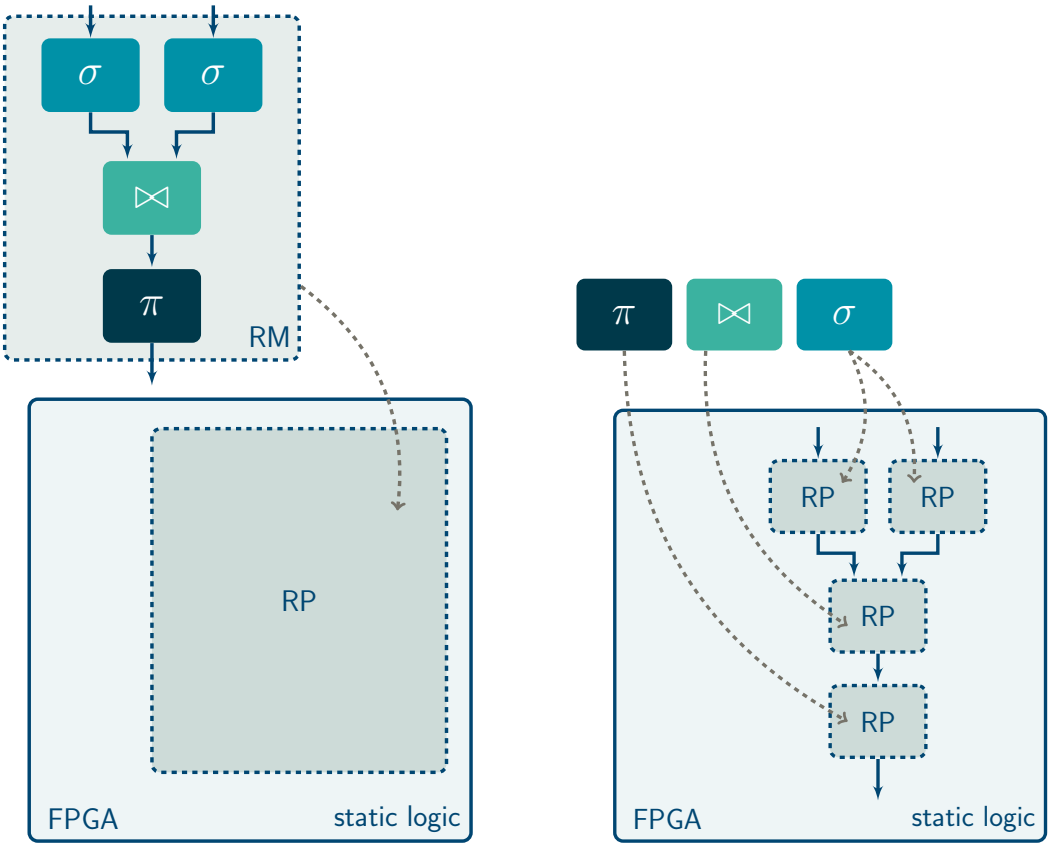


Figure 3.2 – Scopes of FPGA application in query execution. Blue rectangles indicate one processing task and lines between them represent data exchange.

Besides utilizing the FPGA in query execution, it can be beneficially used in other database tasks such as index generation (indicated by FPGA boards in Figure 3.1). Figure 3.2 shows different scopes of FPGA applications. It starts with an entire software system using no reconfigurable hardware. Then, it is followed by a hybrid system, which only accelerates one expensive operator. Other complex application tasks are still processed by the software system. Finally, it ends with a system where the software component simply acts as a controller providing input data and retrieving results from the reconfigurable hardware. *Dynamic Partial Reconfiguration* (DPR) provides the opportunity to use the full chip area and resources of the FPGA in the different phases.

DPR plays an important role in query execution because typically at deployment time of the database system the queries are not known in advance. Thus, the extended LUPOSDATE system has to generate a QEP for a given SPARQL query and maps it on the FPGA resources at runtime. In this work, we present two approaches. The first approach outlined in Figure 3.3(a) uses one large *Reconfigurable Partition* (RP) to exchange complete QEP including the interconnections between the operators. The second approach shown in Figure 3.3(b) separates the chip area into multiple small RPs, each large enough to take one operator at a time. The interconnections between the operators are fixed and thus the global structure of the QEP on the FPGA as well. However, we show in Chapter 5 how the structure can be modified to provide more flexibility and cover a wider range of queries. Furthermore, the communication costs to transfer data to the FPGA and back as well as reconfiguration time must be considered to determine a reasonable usage of the hardware acceleration. It is expected that not all queries benefit from hardware execution due to communication and reconfiguration overhead.

Figure 3.4 outlines the idea of a query pipeline in the proposed hardware/software system. After parsing and reconfiguring the incoming query A, the software component is able to parse and prepare the next incoming query B in parallel, while query A is still being processed. Similarly, query C can be configured for execution while both queries A and B are not yet finished. If one query is finished, the chip area can be released and used for another query. If multiple queries are incoming simultaneously, the partial reconfiguration time can be masked instead of waiting for previous queries to be completed. This allows masking the reconfiguration time because even the amount of time for small bitfiles cannot be neglected [124].



(a) one large RP **(b)** multiple small RPs

Figure 3.3 – Utilization of *Dynamic Partial Reconfiguration* (DPR) in query execution. (a) One large *Reconfigurable Partition* (RP) is reconfigured using a *Reconfigurable Module* (RM) representing a complete *Query Execution Plan* (QEP). (b) A predefined query structure consisting of multiple small RPs is deployed in the static logic. Each RP can take the RM of any operator.

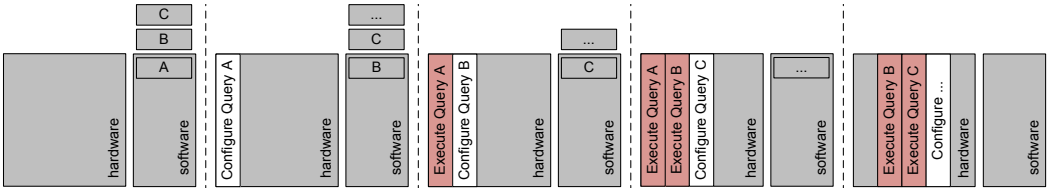


Figure 3.4 – Cascaded query processing. After one query is deployed on the FPGA the next is prepared, configured and executed. If one query is completely executed then the corresponding resources are freed.

Dimensions of Parallelism in Query Execution

Deploying multiple query operators on an FPGA provides true parallelism in various forms. Figure 3.5 outlines the described idea by using different operators in three QEP: A, B and C. Within one query, the operators are concurrently processing data deploying the following types of parallelism.

Horizontal Horizontal parallelism is provided in different levels of granularity. First, multiple queries can be executed in parallel (*inter-query parallelism* [73]). *Intra-query parallelism* is provided by the concurrent processing of intermediate results in two or more subtrees of a complex query tree. Regarding two operators on the same level of a (sub-)tree we refer as *horizontal inter-operator parallelism*. If the data can be partitioned and has less dependencies, then using several sub-units (composing one operator) can result in *horizontal intra-operator parallelism*.

Vertical While one operator is processing its recently consumed input, the preceding operator can already process newly arriving inputs of its predecessor (*inter-operator parallelism*). The natural data flow from the *top* to the *bottom* of the whole operator tree establishes an *operator pipeline* (*vertical intra-query parallelism*). Depending on the granularity of an operator also a micro pipeline inside the operator may be implemented (*vertical intra-operator parallelism*).

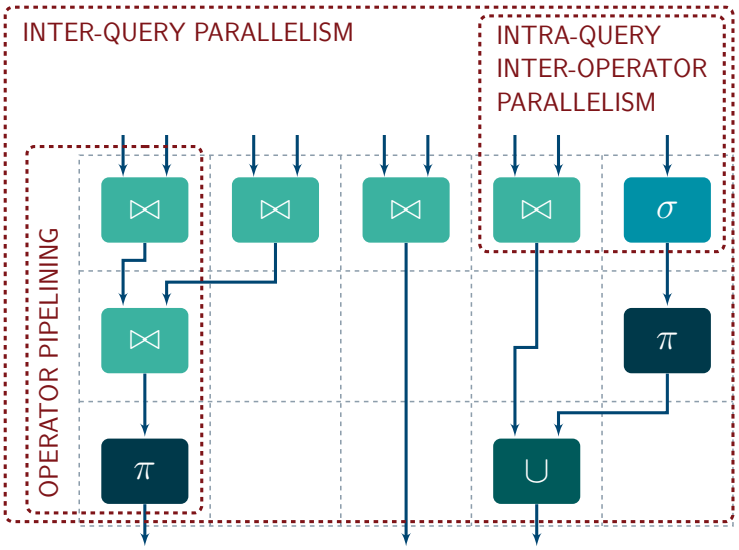


Figure 3.5 – Dimensions of parallelism in query execution.

3.2 Operator Template

In our hybrid architecture incoming queries are dynamically mapped onto FPGA resources following a building block concept. Therefore, each implemented query operator is expressed in VHDL using a minimalistic common interface specified as the *operator template* shown in Figure 3.6. It defines the input and output signals which need to be implemented by each operator. It is motivated by *Volcano* [74], a well-known scheme for query execution engines based on iterators (see Section 2.1.3). The scheme enables a high flexibility in orchestrating operators and a pervasive degree of parallelism. Operators can have up to two preceding operators. If an operator needs only one preceding operator (e.g., filter) then only the left or right input is used. The second input is simply not used by the operator. During synthesis of the design those unused signals are implicitly detected and removed accordingly. Furthermore, each operator has exactly one succeeding operator. The signals are grouped in such a way that the output of each operator can be used as an input for any other operator. Each group consists of four signals, namely data, valid, finished and read. The signal data provides the data to be processed, whereas the valid flag indicates the validity of this data. If an operator has read the current data it uses the backward channel read to indicate the preceding operator that the data was read. The preceding operator might provide more data by raising the valid flag again. If no more data will be present then the finished flag is raised.

In our hybrid query engine the data signal corresponds to the variables of the query to be processed. During query processing each variable is either bound or not bound to a specific value determining one (intermediate) result. Throughout this work, we refer to a set of partially bound variables as *bindings array* (see Definition 3.1). At the preprocessing stage all variables are enumerated and to each variable a position in the bindings array is appointed. This variable position remains unchanged during the entire query execution. Furthermore, the LUPOSDATE system can be configured to map the components of RDF triples to a unique numerical representation (see Section 2.1.3). The reversible bidirectional mapping is stored in a dictionary to enable the computation of specific results. Consequently, this approach leads to improvements in transfer speed and greatly reduces storage requirements. It is more convenient for FPGA to handle numeric values as opposed to unlimited strings.

Figure 3.7(a) shows a bindings array consisting of four variables. During the query evaluation the variables are bound to concrete values resulting in partially and fully bound bindings arrays (see Figure 3.7(b) and (c)). As the bindings array corresponds to the data signal it is directly mapped to a dedicated data path between the operators on the FPGA. In the software-based query execution in LUPOSDATE,

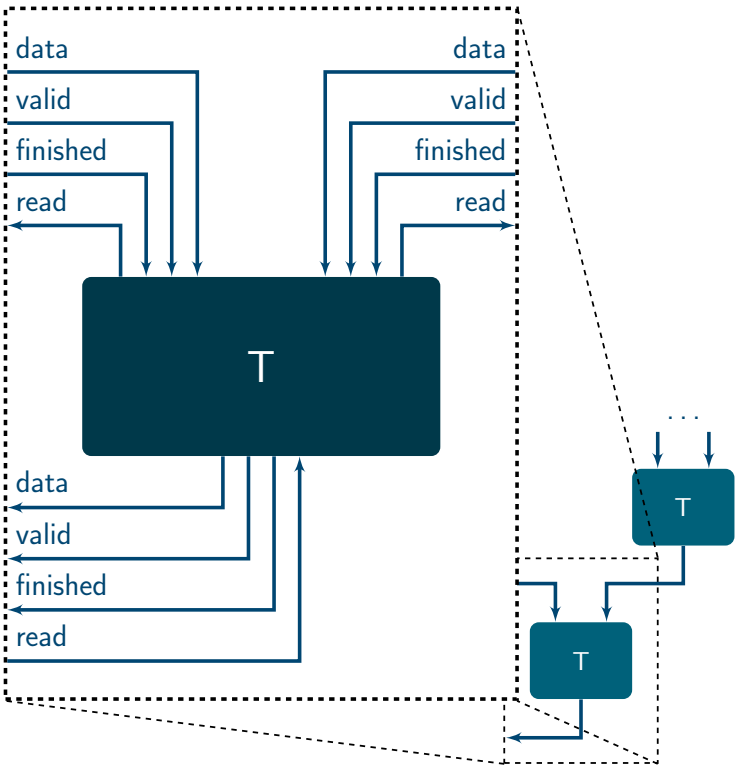


Figure 3.6 – The operator template defines the common interface of all implemented operators. Each group consists of (i) a vector data which corresponds to the bindings array, (ii) a valid flag which indicates the validity of data, (iii) a finished flag which indicates the end of data, and (iv) a backward flag read which notifies the proceeding operator that data was read.

bindings arrays consist of so-called *Lazy Literals*. The *lazy* characteristic comes from the fact that an instance only stores the literal's integer ID which results in a small memory footprint. However, if the materialized literal is required during query processing then the literal is looked up in the previously described dictionary using the integer ID.

Definition 3.1 (Binding and bindings array)
The pair $(?x, t)$ consisting of an variable $?x$ and RDF term t is called *binding* (t is bound to $?x$). A set of bindings bound on different variables $[(?a, t_a), (?b, t_b), \dots]$ is called *bindings array*. Each bindings array can be illustrated as a table with the variable names as column headers. Each row describes the bindings to the corresponding variable. It is not required that all variables in a bindings array are bound. Unbound variables are assigned to the value *null*.

(a) variable positions **(b)** partially bound **(c)** fully bound

Figure 3.7 – Each variable appearing in the has a dedicated position in the *bindings array* which remains unchanged during query execution. It is possible that not all variables are bound in the bindings array after being processed by an operator.

Listing 3.1 – The operator template defines the common interface of all implemented operators (VHDL).

```
1  entity Operator is
2   generic(
3     DATA_WIDTH  : integer := 64;
4     VALUE_WIDTH : integer := 32
5     --[... more operator specific  generics  ...]
6   );
7   port(
8     clk             : in  std_logic;
9     reset           : in  std_logic;
10
11    left_data       : in  std_logic_vector(DATA_WIDTH - 1 downto 0);
12    left_valid      : in  std_logic;
13    left_finished   : in  std_logic;
14    left_read       : out std_logic;
15
16    right_data      : in  std_logic_vector(DATA_WIDTH - 1 downto 0);
17    right_valid     : in  std_logic;
18    right_finished  : in  std_logic;
19    right_read      : out std_logic;
20
21    result_data     : out std_logic_vector(DATA_WIDTH - 1 downto 0);
22    result_valid    : out std_logic;
23    result_finished : out std_logic
24    result_read     : in  std_logic;
25 );
26 end entity Operator;
```

Due to the common interface arbitrary operators can be connected easily as each operator needs no knowledge about actual function of its preceding and succeeding operators. Listing 3.1 presents the entity corresponding to the operator template. The data width of the signals *_data is specified by the generic DATA_WIDTH. The DATA_WIDTH corresponds to the bit width of the bindings array to be processed by the operator. The bit width of a single variable in the bindings array

is set by the generic VALUE_WIDTH. It follows that the number of variables in the bindings array is determined by DATA_WIDTH/VALUE_WIDTH. Thus, the operator in Listing 3.1 processes bindings arrays consisting of $64/32 = 2$ variables. Remember that LUPOSDATE uses a dictionary to map component strings of RDF triple to integer IDs (see Section 2.1.3) and thus the signals *_data represent arrays consisting of integer IDs. Besides the signals for inter-operator communication, a clock signal and reset signal need to be connected. Typically, all instantiated operators operate at the same clock frequency. Issuing a reset will set the operator into an initial state. In the following sections, we will present the supported operators of our hybrid query engine.

3.3 Join Operator

In the following sections, we will focus on implementing the join operator which is one of the most crucial operators in query processing. Although semantically simple (see Definition 2.1), there are numerous algorithms for join processing. In the following sections we will adapt the algorithms *Nested Loop Join* (NLJ), *Merge Join* (MJ) and various forms of the *Hash Join* (HJ) to be executed on an FPGA. Parts and results of this section have been published by the author in [5, 7].

3.3.1 Join Algorithms

Figure 3.8 shows an example of a join operator with two incoming bindings arrays. In both bindings arrays, the variable ?b is bound to a value, which implies the usage of ?b as a join attribute. For all the remaining variables, a bound value appears in only one of the two bindings (?a, ?d in the left and ?c, ?e in the right

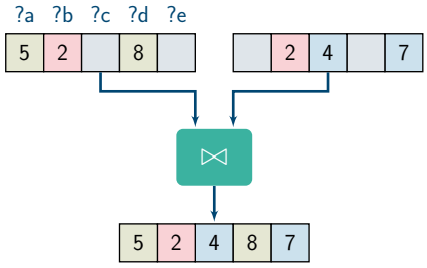


Figure 3.8 – Join of two bindings arrays. Common bound (intersecting) attributes form the join attributes (red). If the join condition is satisfied the bound values of both inputs are taken over into the resulting bindings array.

preceding operator). Consequently, the join is executed if the join attribute ?b in both bindings arrays accomplishes the join condition. In the following sections, the join condition evaluates the equivalence operation with respect to the join attributes (Equi-join, see Definition 2.1). The remaining bound values are inherited into the result.

Although the composition of a single result is simple, all join operators on the FPGA can take advantage by their inherent parallelism and the direct mapping of bindings arrays onto a data path. If a join is executed, the data signals of the two preceding operators simply need to be redirected to the data signal of the current operator's result interface. This is easily done by a multiplexer as shown in Listing 3.2. Each variable in the left incoming bindings array is verified to be valid (bound to a value) or not valid (unbound). The invalidity of a value is indicated by −1 but can be set in the global constant INVALID to any predefined value. As LUPOSDATE uses positive integer values for mapping between strings and IDs, the value −1 remains unused. If the left value is unbound then the content of the variable in the right bindings array is taken over into the result. Otherwise the left value is bound and thus is taken over into the result. The resulting schematic of Listing 3.2 is shown in Figure 3.9.

Listing 3.2 – Composition of a joined result (VHDL).

```
1 result_builder:  for i in 0 to ((DATA_WIDTH / VALUE_WIDTH)−1) generate
2 begin
3 result((VALUE_WIDTH*i+VALUE_WIDTH)−1 downto VALUE_WIDTH*i) <=
4   right_data((VALUE_WIDTH*i+VALUE_WIDTH)−1 downto VALUE_WIDTH*i)
5  when left_data((VALUE_WIDTH*i+VALUE_WIDTH)−1 downto VALUE_WIDTH*i) = INVALID
6  else left_data((VALUE_WIDTH*i+VALUE_WIDTH)−1 downto VALUE_WIDTH*i);
7 end generate result_builder;
```

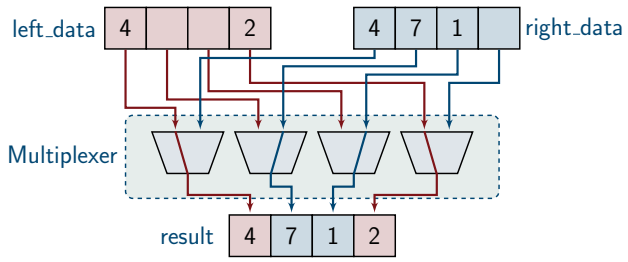


Figure 3.9 – Schematic of the structure expressed in Listing 3.2.

Contrary, a CPU-based software solution with an instruction set operating on a limited number of registers requires the sequential iteration through the two operands

Listing 3.3 – Sequential composition of joined result (C source code).

```
1 void join(int* left, int* right, int* result) {
2   int i;
3   for (i = 0; i < VARIABLES_PER_BINDING; i++) {
4     if (left[i]==INVALID_VALUE)
5       result[i]=right[i];
6     else
7       result[i]=left[i];
8 }}
```

to compose the result. While the implementation of the composition is a very simple loop as shown in Listing 3.3, obviously these steps require multiple clock cycles involving several register transfers. Especially when joins are frequent this simple task can have an impact on the performance compared to the FPGA-based composition.

Before presenting the concrete join algorithms, we introduce some commonly used definitions. The input of each join operator is defined as two sets of bound variables from preceding operators, where R and S denote the input provided by the left operator and the right operator, respectively. Variables which appear in both preceding operands are considered as a sequence of join attributes $J = (j_1, j_2, ..., j_k)$. For simplification we introduce the following definitions and functions before describing the particular join algorithms. Corresponding to the previously introduced operator template, we assume $r.data$ and $s.data$ ($r \in R$ and $s \in S$) to be the raw bindings arrays, and $r.valid$ and $s.valid$ indicate their validity. Definition 3.2 introduces the projection over the join attributes and thus returns only values for all attributes evaluated in the join condition.

Definition 3.2 (Projection over the join attributes)
Given $r.data$ being bound values of bindings array $r \in R$ and $j_i \in J$ join attributes. Then the projection over the join attributes is defined as

$$\pi_J(r.data) := (r.data[j_1], \ldots, r.data[j_k])$$

As mentioned before each join evaluates a join condition to decide whether two bindings arrays $r \in R$ and $s \in S$ are joinable or not. In the following sections we will focus on *Equi-Joins* and therefore introduce in Definition 3.3 the equality-relation according to the join attributes.

Definition 3.3 (Equality-relation according to the join attributes)
Given $r.data$, $s.data$ being bound values of bindings arrays $r \in R$, $s \in S$ and $j_i \in J$ join attributes. Then the equality-relation according to the join attributes is defined as

$$r \overset{J}{=} s := \begin{cases} true & if\ \forall j_i \in J : s.data[j_i] = r.data[j_i] \\ false & otherwise \end{cases}$$

Additionally, we define the smaller-relation with respect to the join attributes $j_i \in J$ in Definition 3.4.

Definition 3.4 (Smaller-relation according to the join attributes)
Given $r.data$, $s.data$ bound values of bindings arrays $r \in R$, $s \in S$ and $j_i \in J$ join attributes. Then the smaller-relation according to the join attributes is defined as

$$r <_J s := \begin{cases} true & if\ r.data[j_1] < s.data[j_1] \vee (r.data[j_1] = s.data[j_1] \wedge \\ & \quad (r.data[j_2] < s.data[j_2] \vee (r.data[j_2] = s.data[j_2] \wedge (\ldots)))) \\ false & otherwise \end{cases}$$

In the following sections we present the join algorithms which are implemented as FPGA-based operators.

3.3.1.1 Nested Loop Join

The *Nested Loop Join* (NLJ) in its simplest form contains a nested loop. The *outer loop* iterates through the bindings of the left operator while for each encountered bindings array the *inner loop* iterates through the bindings of the right operator. Because the defined interface only allows to read each intermediate result only once, the bindings arrays of the right operator have to be stored temporarily within the join operator. Algorithm 1 presents the basic processing steps. As long the right preceding operator provides valid data, it is stored into the operator's internal memory. Afterwards, as long the left preceding operator provides valid data, the retrieved left bindings array is compared with all bindings arrays stored in the internal memory with respect to the join attributes. If the join condition is satisfied then the resulting bindings array is provided on the operator's output. In Figure 3.10 two datasets with each four bindings (representing the join attribute) are joined using the NLJ. The first bindings array of the left input is compared to all bindings arrays of the right dataset. This results in one match (solid line) and 3 mismatches (dashed lines) for the first run through the right input. Afterwards,

Algorithm 1 Nested Loop Join (assuming that each bindings array can be read only once from preceding operators)

```
1:  while not right.finished do
2:   if right.valid then
3:    store right.data in memory
4:    read next right binding
5:   end if
6:  end while
7:  while not left.finished do
8:   if left.valid then
9:    for all b in memory do
10:    if left.data ⋈^J b.data then
11:     provide join(left.data, right.data)
12:    end if
13:   end for
14:   read next left binding
15:  end if
16: end while
```

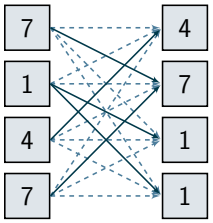


Figure 3.10 – Required comparisons of *Nested Loop Join* (NLJ). Solid lines indicate comparisons of joinable pairs. Dashed lines indicate comparisons of not joinable pairs.

the next bindings arrays of the left input is read and is compared to all bindings arrays of right input. Thus, the join evaluation using the NLJ causes multiple runs through the right dataset. Due to its simplicity the NLJ is suitable for very small, unsorted datasets but lacks of performance for large datasets due to the multiple iteration through the right dataset. Latter also requires sufficient storage to store the whole input of at least one preceding operator. Thus, this join algorithm is useful as a part of more complex algorithms. The straight-forward realization of the NLJ on the FPGA consists of two states. In the first state *READ_RIGHT* the NLJ consumes all the input bindings of the right preceding operator and stores them in the BRAM of the join core. If all right bindings are consumed, the join core switches into the second state *READ_LEFT*. Contrary to the first state, the join core does not consume all the left bindings at once. Instead it consumes one left binding and compares it with each previously stored right binding (inner loop) and provides the result in case of a match. Afterwards it requests the next left binding (outer loop). While iterating through the bindings in the memory the response time needs to be considered seriously (i.e. the duration from setting up the access address and receiving the actual data from the memory). The BRAM on the FPGA needs only one clock cycle. Another state to wait for the response of the memory might be needed while iterating through the inner loop. But as the right input is

sequentially stored, the implementation of the NLJ can avoid this waiting state by using both ports of the BRAM (dual-port). One port provides the next bindings array while the other port is used to pre-fetch the bindings array after next. Thus, on the FPGA the comparison of the join attributes of the two operands, the inheriting of the values to the result and the request of the next operand are processed in parallel within a single clock cycle. Contrary, the sequential execution on the CPU requires multiple clock cycles to solve these tasks.

The worst case time complexity is $O(|R| \cdot |S|)$ for all join algorithms as it might be possible that all bindings arrays of left input R are joinable with all bindings arrays of the right input S. In other words, in the worst case the join may degrade to a kind of cartesian product. However, in real-world cases the performance of different join algorithms differs significantly. As for each bindings array of the left input the join condition is evaluated considering all bindings arrays of the right input, the time complexity of the NLJ is always $\Omega(|R| \cdot |S|)$ [68]. Reading and storing one bindings array of the buffered input into the BRAM takes one cycle. Due to pre-fetching mechanism reading one bindings array from the other input and the buffer, evaluating the join condition and providing the joined result takes one cycle as well. In summary, all bindings arrays of S have to be stored in the buffer which afterwards is read for each bindings array in R. Thus, the NLJ requires at least $(|S| + |R| \cdot |S|)$ clock cycles assuming that the succeeding operator consumes the results at maximum speed.

3.3.1.2 Merge Join

The *Merge Join* (MJ) requires data that is sorted according to the join variables. Algorithm 2 presents the basic processing steps. It initially reads the bindings arrays of both preceding operators and checks if they are equal regarding the join attributes. Due to the sorting, all remaining bindings arrays have to be equal or larger than the current solution. If the join attributes are not equal, the next solution of the operator with the smaller value is requested. Otherwise, if the join condition is fulfilled, all bindings arrays with the same join attributes can be read to return the joined bindings arrays. In Figure 3.11 two sorted datasets are joined using the MJ. The first binding of the left input is compared to the bindings of the right input until the right input provides a binding which has a higher value regarding the join attribute. If this case occurs then the next bindings of the left input are compared in the same manner to the remaining bindings of the right input. As mentioned in Section 2.1.3, LUPOSDATE maintains six indices corresponding to the six collation orders of RDF triples which permits the frequent usage of the MJ. The FPGA implementation consists of two modules, *comparator*

Algorithm 2 MERGE JOIN (assuming that the input data is sorted accourding to the join attribute)

```
1:  while left.valid and right.valid do
2:   if left.data <_J right.data then
3:    read next left binding
4:   else
5:    if right.data <_J left.data then
6:     read next right binding
7:    else
8:     a ← left.data
9:     while right.valid and a =^J right.data do
10:     store right.data in memory
11:     read next right binding
12:    end while
13:    while left.valid and a =^J left.data do
14:     for all b in memory do
15:      provide join(left.data, b.data)
16:     end for
17:     read next left binding
18:    end while
19:   end if
20:  end if
21: end while
```

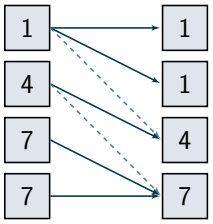


Figure 3.11 – Required comparisons of *Merge Join* (MJ). Solid lines indicate comparisons of joinable pairs. Dashed lines indicate comparisons of not joinable pairs.

and *merger*. Due to the knowledge about the order, the *comparator* consumes left and right bindings from both preceding operators until a match is found. Then the *merger* gets filled with the matching bindings and all following bindings with the same join attribute and starts providing the joined result to the succeeding operator. During this merging process the *comparator* already searches for the next matching pairs. When the *merger* finishes the current set the *comparator* can provide new matching bindings faster. This improves the performance especially if there are rows of matching pairs followed by long rows of non-matching pairs. Due to the knowledge about the order, the FPGA implementation is able to generate results while searching for the next matching join pairs.

Assuming the input data to be sorted, the MJ has a time complexity of $O(|R| \cdot |S|)$ respectively $\Omega(|R| + |S|)$ [68]. The comparison of two bindings arrays including temporarily storing them if they match is executed within one clock cycle. Each

subsequent bindings array with the same value in the join attributes is stored within one clock cycle. Afterwards, reading of the buffer and providing one joined result takes one clock cycle. During this cycle already the next bindings array is loaded from the buffer. Concurrently, the left and right inputs are consumed until a pair of bindings arrays satisfying the join condition is found. Thus, during joining the buffers, a great amount of bindings arrays might have been discarded already because they do not satisfy the join condition. In best case the left and right input already match after joining the buffers is done and can be stored in the buffers straightaway. Assuming that the left and right input contain no joinable bindings arrays then the presented MJ operator finishes the processing within $|R|+|S|$ clock cycles. If each bindings array of the left input has exactly one join partner in the right input then determining the join result takes $3(|R|+|S|)$ clock cycles. In worst case all bindings arrays of left and right input have to be joined. Then the MJ logically degrades to a NLJ. Temporarily storing the whole input of R and S takes $max(|R|, |S|)$ clock cycles. Then each bindings array in R has to be initially loaded from the buffer. At the same time the first bindings array of S is loaded from the buffer. All subsequent bindings arrays of S are loaded in a cascaded fashion. Thus, the MJ requires $(max(|R|, |S|) + 3|R| + |R| \cdot |S|)$ clock cycles in worst case.

3.3.1.3 Asymmetric Hash Join

The *Asymmetric Hash Join* (AHJ) is divided into two phases (see Algorithm 3). In the *build phase*, the bindings arrays of the *smaller* intermediate result are processed. Depending on a calculated hash value over the join attribute, they are stored in a hash table. In the following *probe phase* the bindings arrays of the larger intermediate result are probed against the previously constructed hash table. Therefore, for each bindings array of the larger result a lookup in the hash table regarding a given key (consisting of values of the join attributes) is executed. This returns all join partners of a given bindings array. Figure 3.12 shows the schematic of the AHJ. In the build phase the hash function h is applied on each bindings array of the left input and the bindings arrays are stored in the hash table according to the calculated hash value. In the probe phase the same hash function h is applied on each bindings array of the right input. The calculated hash value points to potential join partners (bindings arrays of the left input with the same hash value) in the hash table. However, depending on the hash function it is possible that two different keys map on the same hash value. Therefore, the hash table needs to compare the plain keys before returning the result. The AHJ is only applicable for equi-joins, and its performance is affected by the quality of the applied hash function. Typically, well known hash functions contain many additions, multiplications and modulo operations. Even though there are dedicated

Algorithm 3 Asymmetric Hash Join

```
1: while right.valid do
2:   key ← π_J(right.data)
3:   store tuple(key, right.data) in hash table H
4:   read next right binding
5: end while
6: while left.valid do
7:   key ← π_J(left.data)
8:   for all b in lookup(key, H) do
9:     provide join(left.data, b.data)
10:  end for
11:  read next left binding
12: end while
```

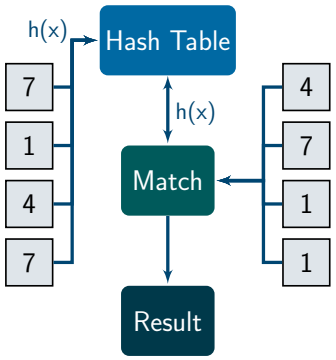


Figure 3.12 – Schematic view of *Asymmetric Hash Join* (AHJ).

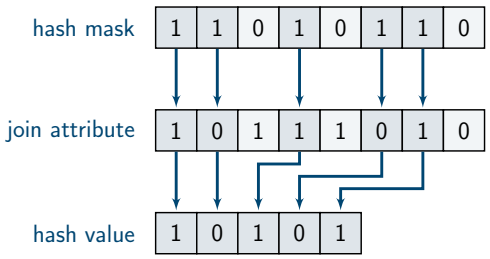


Figure 3.13 – The set bits in the hash mask denote the corresponding bits in the join attribute composing the hash value.

multipliers/accumulators available in advanced DSP48E1 slices [87], we avoid multiplications and additions. Therefore, this approach uses a *hash mask* with the same data width as the join attribute to describe which bits of the join attribute will contribute to the final hash value. As the hash mask is set at design time, choosing corresponding bits from the join attribute will result in simple wiring and will cause no additional clock cycles for computation. Ideally, the number of set bits in the hash mask is equal to the address width of the hash map. Figure 3.13 shows an example with a hash mask width and join attribute width of eight bits. In this example, the hash value width is five bits, which is equal to the number of set bits in the hash mask. Furthermore, implementing the hash table in BRAM enables a fast hash table lookup in one cycle. However, during the build phase no joined bindings arrays will be provided to the succeeding operator. Thus, the lower operators are stalled until the first intermediate result is generated in the probe phase. The time complexity of the build phase is $O(|R|)$ assuming R to be the smaller relation and calculating the hash value is possible in constant time

[68]. The time complexity of the probe phase is $O(|R| \cdot |S|)$ respectively $\Omega(|S|)$. Regarding our FPGA-based AHJ inserting one bindings array into the hash table consumes at least three clock cycles. The hash value is calculated instantaneously when the bindings array arrives at the input and the hash value is used as the address to retrieve the corresponding location in the BRAM in the first clock cycle. After waiting one clock cycle for the BRAM's response the returned data indicates either the address in the hash table is already used or is free. If latter then the bindings array is stored at the address consuming another clock cycle. If the location in the hash table holds already another bindings array then *open addressing with linear probing* is used [187, p. 526]. Linear probing simply increments the address by one until an empty location in the hash table is found to store the current bindings array. This becomes disadvantageous when the hash table gets filled or many values map onto the same location in the hash table but needs no additional memory overhead. The examination of whether one location is empty and loading the next address can be cascaded and thus taking two clock cycles. During the probe phase the lookup is analogous but after loading one bindings array from the hash table the join attributes need to be compared for equality. Assuming each bindings array of the left input has exactly one join partner in the right input and the location after each bindings array is free then determining the join result takes $(3|R|+4|S|)$ clock cycles (building hash table for R, two lookups – each consuming two clock cycles – in the hash table for each $s \in S$ for retrieving the join partner and checking for end of probing). However, the number of required probes depends on the load factor of the hash table. Knuth [187, p. 528] proves that the average number of probes for insertion or unsuccessful search is approximately 2.5 at a load factor of 50%, or 50.5 probes at a load factor of 90%. In worst case all bindings arrays of R map onto the same location in the hash table and thus building the hash table takes $\sum_{i=1}^{|R|}(3+2(i-1)) = |R|^2+2|R|$ clock cycles in total. Additionally, the probing phase requires $|S| \cdot 2|R|$ clock cycles (for each bindings array in S two cycles for each subsequent bindings array in the hash table).

3.3.1.4 Symmetric Hash Join

The *Symmetric Hash Join* (SHJ) uses two hash tables in parallel to decrease the latency until the first results are provided (see Algorithm 4). The two hash tables, HL and HR, store the bindings of the left and right operands, respectively (see Figure 3.14). If a valid left binding is incoming, the hash value according to the join attribute is calculated and the binding is stored in HL. Then, the probe phase is initiated to find a corresponding join partner in HR. Additionally, the right bindings will be stored in HR and checked against HL.

Algorithm 4 Symmetric Hash Join

```
1:  while not (left.finished and right.finished) do
2:   if left.valid then
3:    key ← π_J(left.data)
4:    store tuple(key, left.data) in hash table HL
5:    for all b in lookup(key, HR) do
6:     provide join(left.data, b.data)
7:    end for
8:    read next left binding
9:   end if
10:  if right.valid then
11:   key ← π_J(right.data)
12:   store tuple(key, right.data) in hash table HR
13:   for all b in lookup(key, HL) do
14:    provide join(right.data, b.data)
15:   end for
16:   read next right binding
17:  end if
18: end while
```

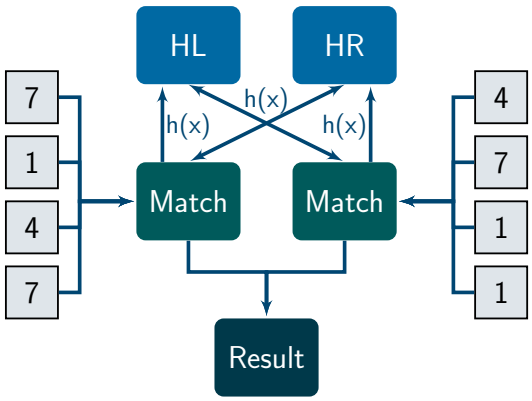


Figure 3.14 – Schematic view of *Symmetric Hash Join* (SHJ).

In the best case scenario, the first two bindings fulfill the join condition and can immediately be provided to the next operator. Besides the previously mentioned advantages, the FPGA implementation benefits from parallel calculating the hash values of the two operands and parallel searching for join partners in the corresponding hash tables. The mechanisms of probing and building are identical to the AHJ. However, during the probing in one hash table, it is blocked for inser-

tions. Assuming each bindings array of the left input has exactly one join partner in the right input and the location after each bindings array is free, then the insertion of one bindings array from the left respectively from the right input into the corresponding hash table takes three clock cycles. The subsequent probing in the other hash table takes another two plus two clock cycles. As the insertions into the hash table and the probe phase are executed concurrently, in total $7 \cdot max(|R|, |S|)$ clock cycles are required in this scenario. Note that even if both hash tables find a match only one joined result can be provided at a time. The other matching bindings array will be joined after the previously provided result is read. In worst case, each bindings array of R is joinable with each bindings array of S and bindings arrays of R and S are alternately arriving. Then building the hash tables takes $(|R|^2 + 2|R|) + (|S|^2 + 2|S|)$ clock cycles. Furthermore, we assume $|R| = |S|$ and the first arriving bindings array is from R. Then in total all probing steps take $\sum_{i=1}^{|R|}(3 + 2(i-1)) + \sum_{i=1}^{|S|}(3 + 2i) = |R|^2 + 2|R| + |S|^2 + 4|S|$ cycles (each bindings array $s_i \in S$ has to be probed with i bindings arrays from R).

3.3.1.5 Hash Join using separate chaining with linked list

The previously described approaches of the Hash Join use open addressing with linear probing as collision resolution. As well-known, while the hash table is filling up the performance of this approach decreases drastically. Thus, the *Hash Join using separate chaining with linked list* (HJL) is implemented. Accordingly to the AHJ, first a hash table of the smaller input is created by applying the hash function as mentioned before to get the address of the first binding with the same hash value. The general idea is depicted in Figure 3.15. First the hash function h is applied on the bindings array respectively the join attributes. The resulting value x is used to address the *hashmap*. It stores the *first* and the *last* address of bindings with the same hash value according to the join attribute. If no bindings array occurred before for a specific hash value then *first* and *last* are empty ($\perp$). In this case the next free memory location in the *bindings storage* is used to store the current bindings array. The next free memory location in the bindings storage is simply obtained by incrementing a counter starting at address zero. This address is then stored in the hashmap as *first* and *last* address. If another bindings array with the same hash value needs to be stored then the *first* address indicates that there was a bindings array with the same hash value before. Consequently, the current bindings array is stored in the next free location in the bindings storage and its address is written into the *last* address field in the hashmap as well as in an additional address field *successor* of the former last element. Hence, bindings arrays with the same hash value are linked through the *successor* address. The *first* address is used as the entry into the linked list structure. The corresponding

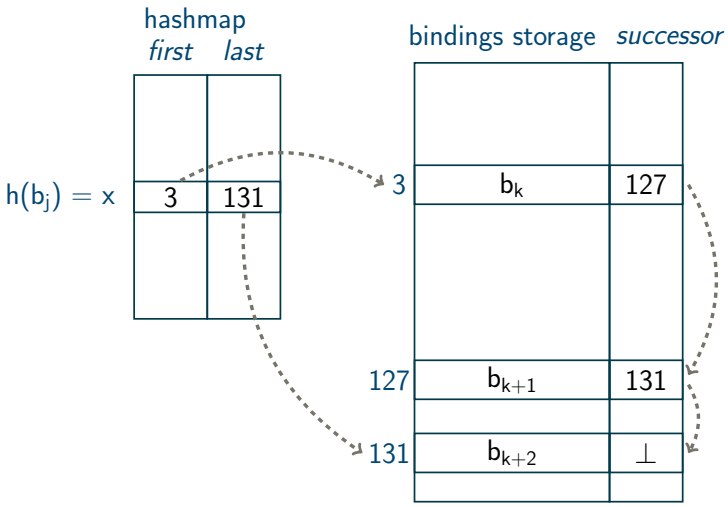


Figure 3.15 – Structure of the hashmap and bindings storage used in the *Hash Join using separate chaining with linked list* (HJL).

last address enables the easy and fast insertion. Drawback of this approach is the need of additional memory (currently realized using BRAM), but ensures that in the probe phase only bindings with the same hash value are compared, which is especially beneficial when the hash table has a high load factor. The overall time complexity of probe and build phase is comparable to the AHJ. However, because we explicitly store the address of the last element of each linked list, inserting a new bindings array at the end does not require to iterate through all elements as it is required at the AHJ using linear probing. Thus, calculating the hash value and retrieving the corresponding entry address of the linked list from the hash map takes three clock cycles. In the succeeding clock cycle the bindings array is written into the bindings storage, its address stored as the end of the linked list and as the successor of the former end of the linked list. In total the insertion of one bindings array into the hashmap takes four clock cycles in any case independently of previously inserted bindings arrays. During the probe phase, calculating the hash value and retrieving the corresponding entry address of the linked list from the hashmap takes three clock cycles. This returns the first and last address of the linked list and thus no potential join partner is available at this step. The retrieved entry address is used to retrieve the first bindings array from the linked list which takes another three clock cycles. Furthermore, this provides the address of the successor in the list. During provision of the current joined result the next successor can be retrieved from the list consuming three clock cycles. Loading the successors in a cascaded fashion can not be used due to the fact that the successor's address is just determined. In summary, the build phase takes $4|R|$ clock cycles

for all bindings arrays of R even if all bindings arrays of R map onto the same location in the hash table. Processing the join of one bindings array $s \in S$ takes $(3 + 3 \cdot L_{hash(s)})$ clock cycles with $L_{hash(s)}$ the length of the linked list containing the bindings arrays $r \in R$ with $hash(r) = hash(s)$. Assuming each bindings array of S has exactly one join partner in R then determining the join result takes $(4|R| + (3 + 3)|S|)$ clock cycles (building, one initial loading and one probing). In worst case the probing phase requires $|S| \cdot (3 + 3|R|)$ (for each bindings array in S three clock cycles for initially loading the entry in the hash table and three clock cycles for each bindings array from R).

3.3.2 Micro Benchmarks

In this section the hardware-accelerated and the software-based join execution are compared to give a first insight of potential performance improvements regarding FPGA-based query execution.

3.3.2.1 Evaluation Setup

In the following paragraphs we introduce the evaluation setups for the FPGA-based respectively CPU-based operators and describe the evaluation data and metrices.

FPGA All previously introduced join operators are described in VHDL to be evaluated on the FPGA platform (see Section 2.2.2). The FPGA platform operates in standalone mode and thus is not mounted into a host. Figure 3.16 shows the logical structure of the evaluation framework on the FPGA. Due to the previously introduced operator template (see Figure 3.6), the join operator has no knowledge about the internal implementation of its preceding operators (*Op X* and *Op Y*). The join operator uses the common interface to consume their outputs as its two inputs. Vice versa the preceding operator *Op Z* consumes the results produced by the join operator. To emulate the two preceding operators of the join operator, the data is transferred to the FPGA using the UART and stored in BRAM outside the join operator. The operators *Op X* and *Op Y* only provide these data. Thus, the communication costs are not considered in the following performance analysis but will be included in the evaluation of the complete system later (see Chapter 4). When the evaluation starts, the first bindings array is provided to the join operator. The evaluation ends when the join operator sets its finished signal, which means that the join operator has consumed all bindings of the preceding operators and will not provide more results.

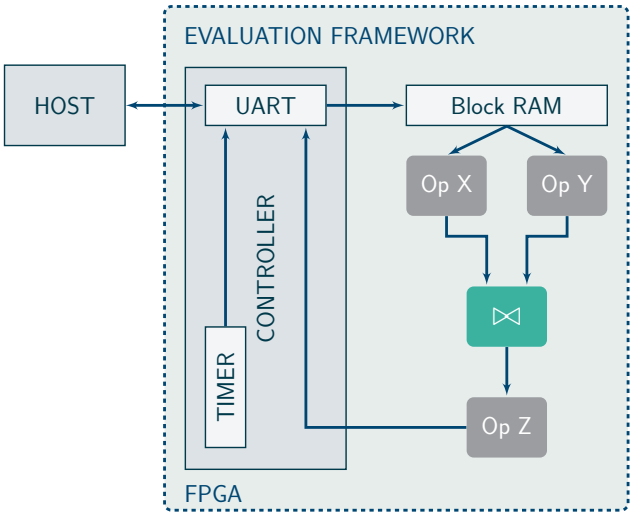


Figure 3.16 – Evaluation framework (connections between operators are simplified).

CPU-based implementation In order to obtain comparable results from a general-purpose CPU system (Intel Core 2 Duo T7500, 2.2 GHz, 4 MByte Cache, 3 GByte RAM, Windows 7 (32 Bit), compiler GCC 3.4.4), all join algorithms are implemented as an efficient software solution in the programming language C as well. The implementation in the programming language *C* follows the usual iterator concept [74] which enables pipelining in the whole operator graph. As the solutions are computed one by one a huge size of intermediate results is avoided. Furthermore, as the bindings arrays on the FPGA are simple arrays of integer, also the software-based approach operates on integer arrays. Although in a real-world database system complexer data structures are used to increase extensibility and maintainability of the software, we aim for maximum throughput of the software-based join execution in this first performance evaluation. In order to achieve the maximum performance on the general-purpose CPU system, the test data is read from the HDD into the main memory before starting the performance analysis. The time it takes to load the input from the hard drive is not included in this performance analysis. Each experimental run is executed 100 times to reduce deviation and fully utilize caching capabilities of the CPU.

Test data and performance metrics In the following micro benchmarks for both, the FPGA-based and software-based approach, the same datasets are used. Therefore, 100 datasets are randomly generated whereby each bindings array contains 4 variables, each 16 bits wide. Due to limited space of the BRAM, the

generated datasets with up to 1000 bindings arrays are relatively small. However, this is highly beneficial for the CPU-based execution as the datasets completely fit into the cache. In the following section, we analyze the execution time of all join operators on both platforms by varying input size and join probability. The latter is expressed by the *number of join attribute values* (JA). While in this evaluation the join is always determined regarding only one variable, the number of possible different bound values of this variable is set by JA. If JA is small then the probability of matching join pairs is high. For example, $|JA| = 1$ means that each bindings array has the same bound value regarding the join attribute and consequently each bindings array of the left input has to be joined with each bindings array of the right input. This corresponds to the cartesian product of both inputs. Thus, if $|R| = |S| = 1000$ and $|JA| = 1$ then the join operator generates one million joined bindings arrays. On the other hand, increasing JA results in a lower join probability. In the following evaluation and throughout this work we use the *Speedup* to indicate the performance enhancements of our FPGA-based approach compared to a software-based solution running on a general-purpose CPU. The speedup expresses the execution time ratio of both systems shown in Definition 3.5.

Definition 3.5 (Speedup)
$SP = \frac{T_{CPU}}{T_{FPGA}}$ with T_{CPU} execution time of the software-based solution running on a general-purpose CPU and T_{FPGA} execution time of the FPGA-based approach.

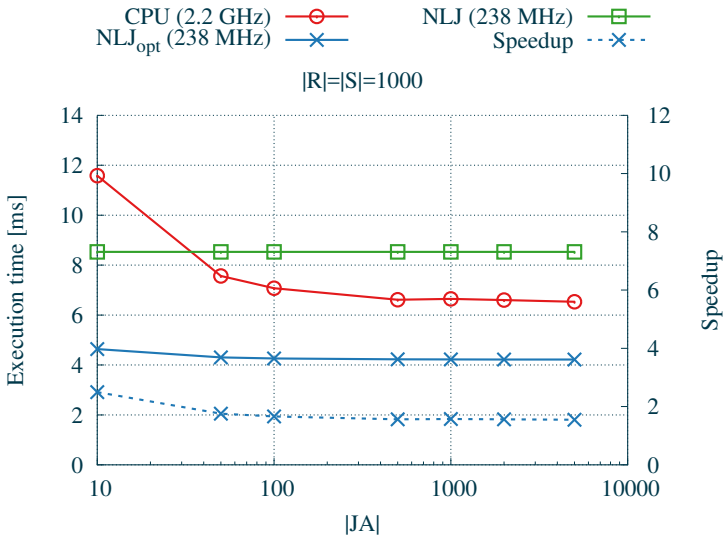
Thus, if $SP = 1$ then both systems provide the same performance. If $SP > 1$ then the FPGA-based execution is faster than the software-based execution. If $SP < 1$ the FPGA-based approach is slower than solving the task on the CPU. In addition to the scalar value we use X as a *unit*, e.g., 2X indicates a *two times faster* execution.

3.3.2.2 Comparison of Execution Times

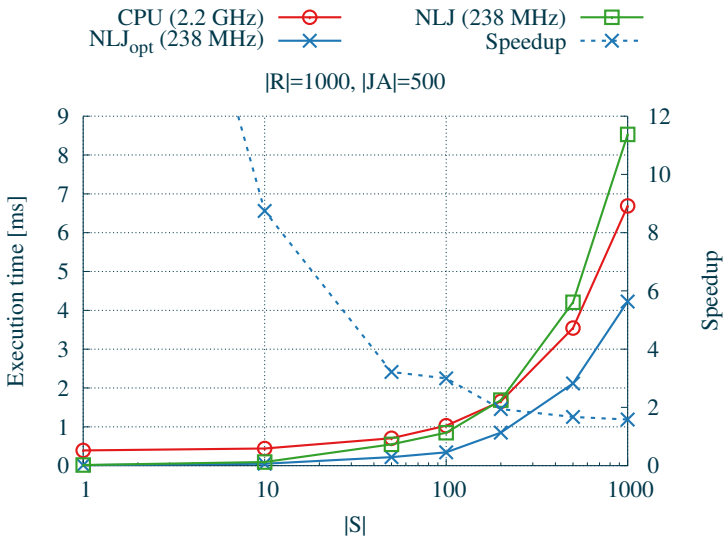
In the following paragraphs, the execution times of each join operator running on the FPGA and the software platform are compared.

Nested Loop Join (NLJ) Figure 3.17 illustrates the results of the NLJ. Besides the C implementation it shows the performance of two approaches for the FPGA running at a clock rate of 238 MHz. The first (unoptimized) approach (NLJ) is the straight forward implementation while the second approach (NLJ_{opt}) uses the previously described optimized memory access method to pre-fetch the next bindings array in advance from the internal buffer while evaluating the join condition

on the current bindings array (see Section 3.3.1). Figure 3.17(a) shows the impact of JA. If JA is small then many bindings arrays need to be joined causing a higher load on the CPU.



(a) Vary the number of potential join partners.



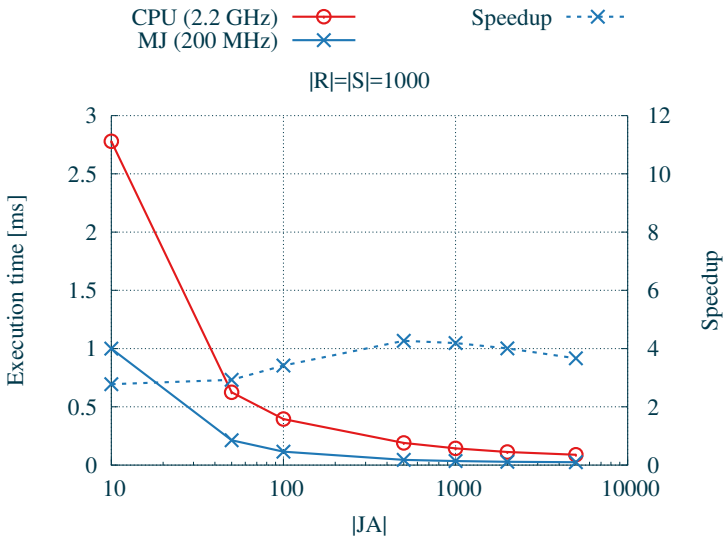
(b) Vary the input size of one operand.

Figure 3.17 – Execution times of *Nested Loop Join* (NLJ).

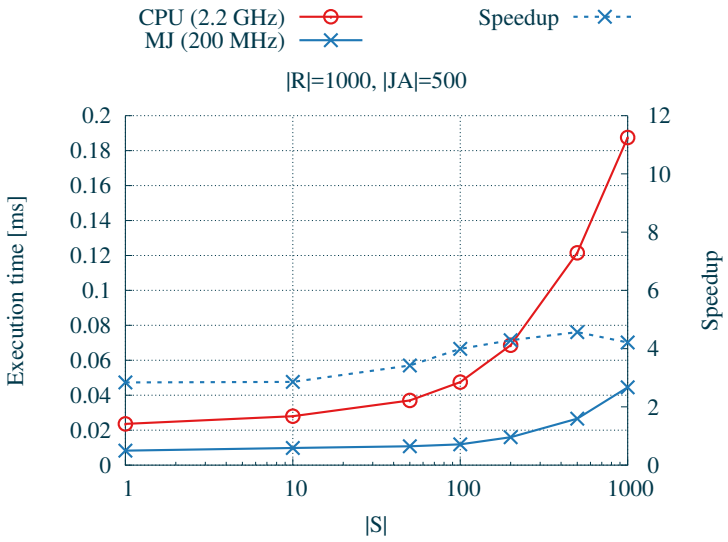
While the C implementation on the CPU sequentially iterates through each binding of the array in order to assemble the resulting bindings array, the FPGA can take advantage of building the results in parallel. This drawback of the CPU decreases when the number of JA increases and the number of joins decreases. Thus, the overall performance of the software solution increases while the performance of the unoptimized approach (NLJ) remains constant. The optimized NLJ (NLJ_{opt}) impressively shows how a relatively simple optimization in terms of a tailor-made pre-fetching significantly impacts the performance. The join probability has no significant impact on the NLJ_{opt} running on the FPGA and NLJ_{opt} achieves speedups of 1.8X to 3X. Figure 3.17(b) shows the execution times with a varied size of one input and a constant size of the other input and JA (|R|=1000, |JA|=500). With increasing |S| the execution times increase for the software-based and the FPGA-based approaches but shrinks the achievable speedup. At a very small input size the optimized FPGA approach significantly outperforms the software solution.

Merge Join (MJ) As mentioned before, the MJ requires the input data to be sorted with respect to the join attributes. Therefore, the data is sorted before processing it in the join operator and thus the time for sorting the data is not included in the total time. This assumption is reasonable based on the use of indices in the final intended system (see Section 2.1.3). Increasing JA in both approaches leads to a significantly decreasing execution time (Figure 3.18(a)). However, the FPGA-based MJ achieves speedups of 1.5X up to 4.2X compared to the software solution. Figure 3.18(b) shows the execution time depending on the input size |S| of one preceding operator. Obviously, increasing the input size results in a higher execution time but the hardware-accelerated MJ slightly increases the achievable speedup.

Asymmetric Hash Join (AHJ) In the following paragraphs, for all hash-based joins on the FPGA the hash mask HM_1=`'1111111100000000'` is used. That means only the 8 most significant bits of the 16 bit wide join attribute are considered to address the hash table. Later we will show the impact of the hash mask on the performance of the AHJ. In all test cases, the AHJ on the FPGA defeats the software solution by a speedup of at least 1.3X (see Figure 3.19(a)). If the frequency of joins is high (small JA), the gap between the software solution and the FPGA increases. If the size of one input is small and thus the hash table is sparsely filled the FPGA-based AHJ directly benefits from the very fast lookups in its local BRAM (see Figure 3.19(b)). The highest reached speedup is 5.7X for |S|=10.

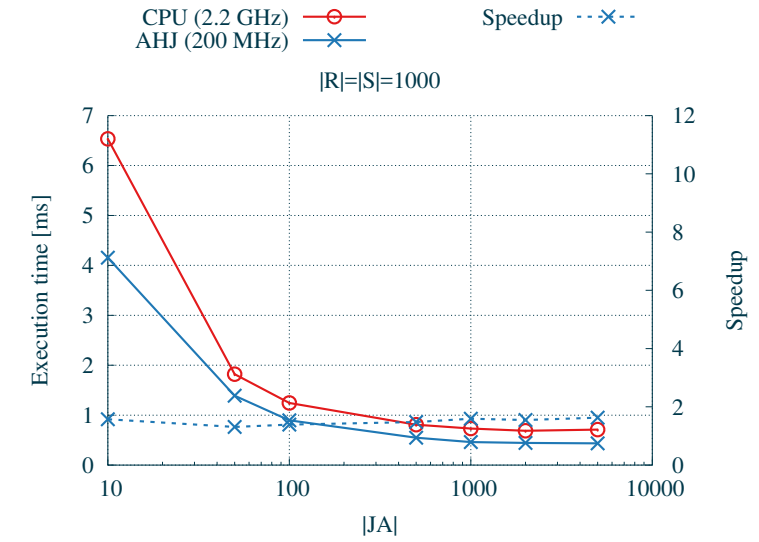


(a) Vary the number of potential join partners.

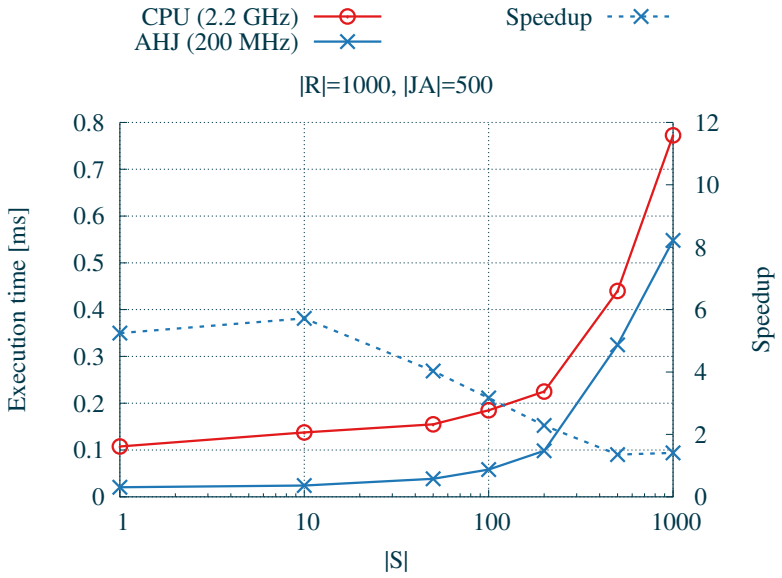


(b) Vary the input size of one operand.

Figure 3.18 – Execution times of *Merge Join* (MJ).



(a) Vary the number of potential join partners.



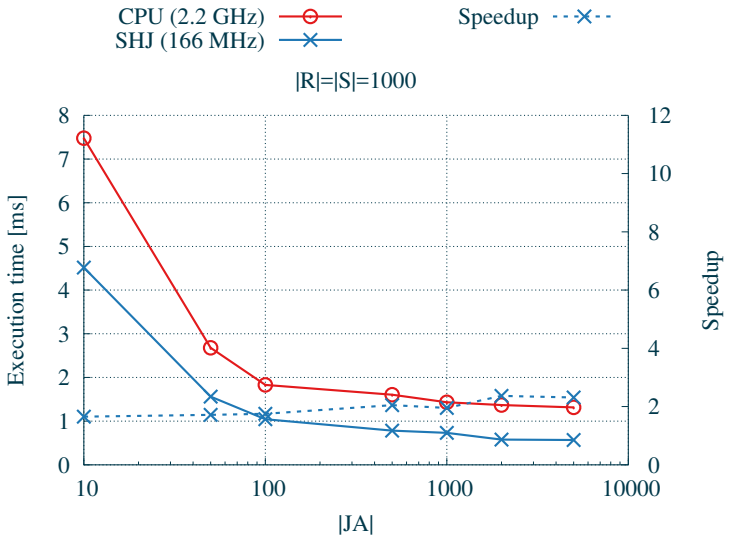
(b) Vary the input size of one operand.

Figure 3.19 – Execution times of *Asymmetric Hash Join* (AHJ).

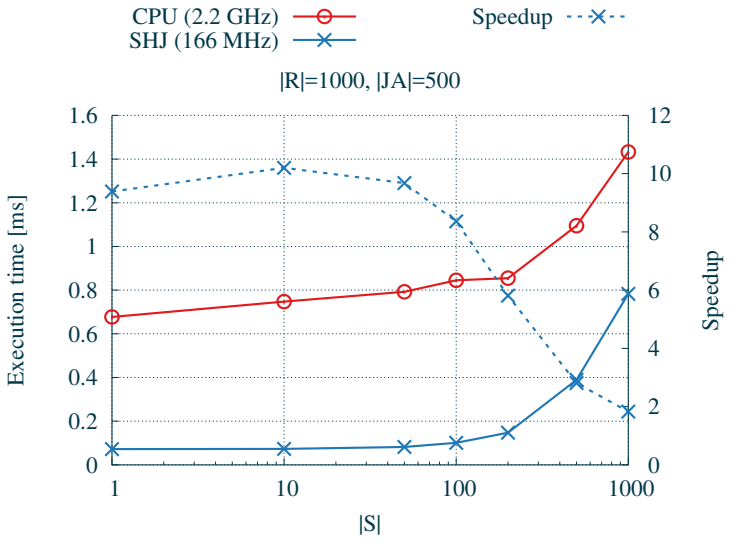
Symmetric Hash Join (SHJ) As mentioned before the SHJ uses two hash tables. Due to its higher circuit complexity, more logic resources are needed and the signal propagation delay increases. Thus, the SHJ can just be used at a clock rate of 167 MHz on the FPGA. However, the FPGA-based SHJ reaches a speedup between 1.3X (see Figure 3.20(a)) and 10.2X (see Figure 3.20(b)) compared to the software-based implementation.

Hash Join using separate chaining with linked list (HJL) In addition to the AHJ (using linear probing), the HJL was implemented and compared with the software solution. If JA is small and many bindings of the join attribute map on the same position in the hash table then the FPGA achieves a speedup of not less than 3X (see Figure 3.21(a)). Furthermore, if |JA| increases then the performance of the software as well as the hardware solution increases due to less collisions in the hash table. However, the FPGA-based HJL achieves higher performance gains and is able to increase the speedup up to 10.2X. Varying the input size (see Figure 3.21(b)) shows significantly that the performance drop of the C implementation is much more noticeable than for the FPGA. Overall the FPGA achieves a performance improvement between 8X and 9.4X in these test cases.

All join cores at normalized clock rate Figure 3.22 presents the execution times of all join algorithms executed on the FPGA running at a normalized clock rate of 100 MHz. As expected the MJ is the fastest join algorithm in all evaluated cases. If the input datasets are not sorted then the HJL is a wise choice. Usually the NLJ shows the worst performance. In order to choose a concrete algorithm for a logical operator in a specific query the LUPOSDATE system uses cost-based optimizations which take estimations about the cardinality of intermediate results into account [14]. According to these estimations, in the physical optimization a concrete join operator is chosen (see Section 2.1.3).

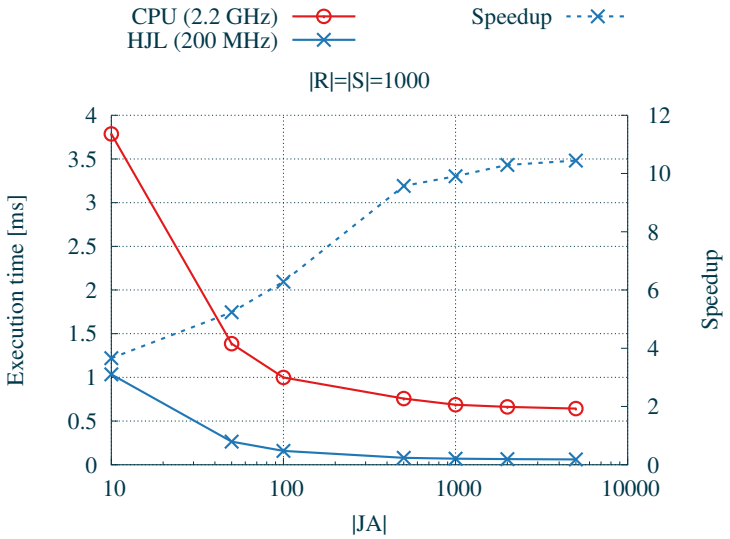


(a) Vary the number of potential join partners.

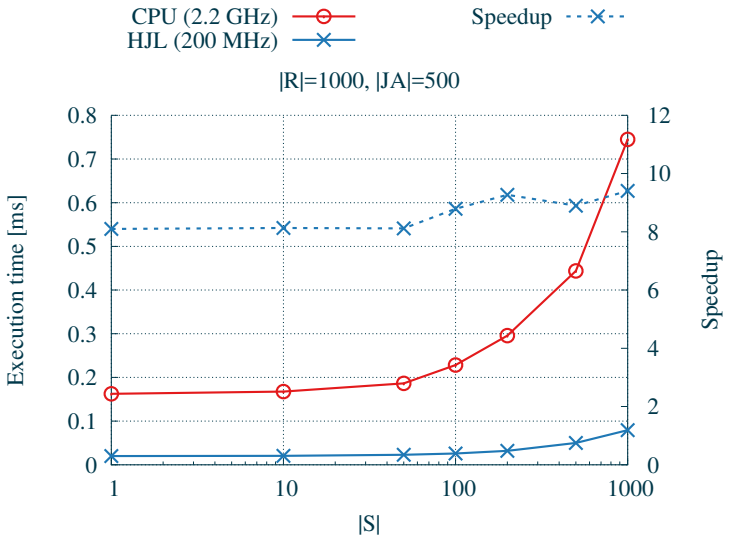


(b) Vary the input size of one operand.

Figure 3.20 – Execution times of *Symmetric Hash Join* (SHJ).

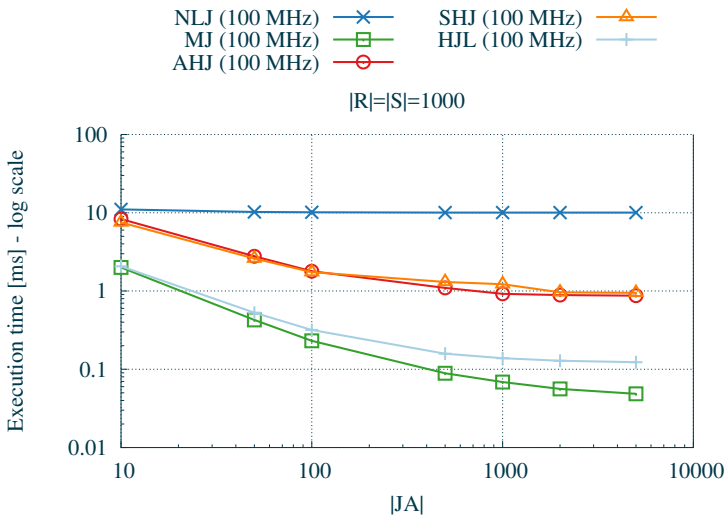


(a) Vary the number of potential join partners.

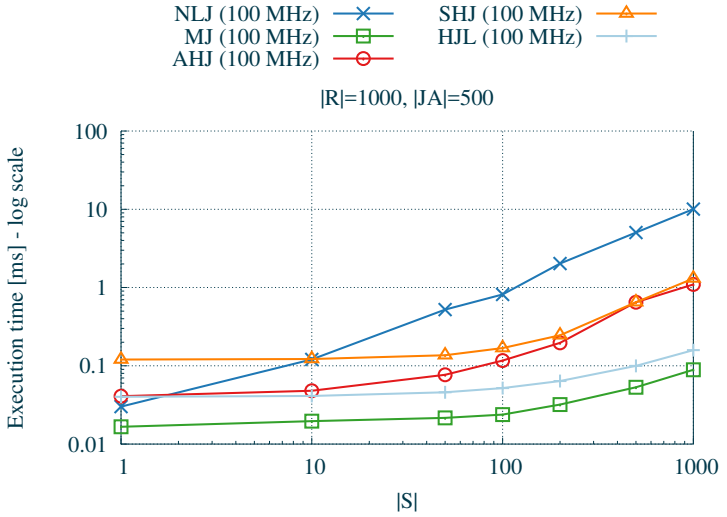


(b) Vary the input size of one operand.

Figure 3.21 – Execution times of *Hash Join using separate chaining with linked list* (HJL).



(a) Vary the number of potential join partners.



(b) Vary the input size of one operand.

Figure 3.22 – Execution times of all hardware-accelerated join operators running at a normalized clock rate of 100 MHz.

3.3.2.3 Impact of the Hash Mask

Certainly, the hash mask has an impact on the performance of hash-based joins. In the previous measurements the hash mask HM_1=`'1111111100000000'` was used. That means only the 8 most significant bits of the 16 bit wide join attribute are considered to address the hash table. Figure 3.23(a) as well as Figure 3.23(b) show the execution time of the AHJ using different hash masks. Like HM_1, the hash mask HM_2 uses 8 bit but the 4 most significant as well as the 4 least significant bit are set. Because of the uniform distribution of the data both hash masks show the same performance. Contrary, HM_3 considers only the 4 most significant bit and as expected the execution time increases significantly due to more occurring collisions in the hash table. HM_4 uses 12 bit and thus is able to further increase the performance compared to HM_1 and HM_2.

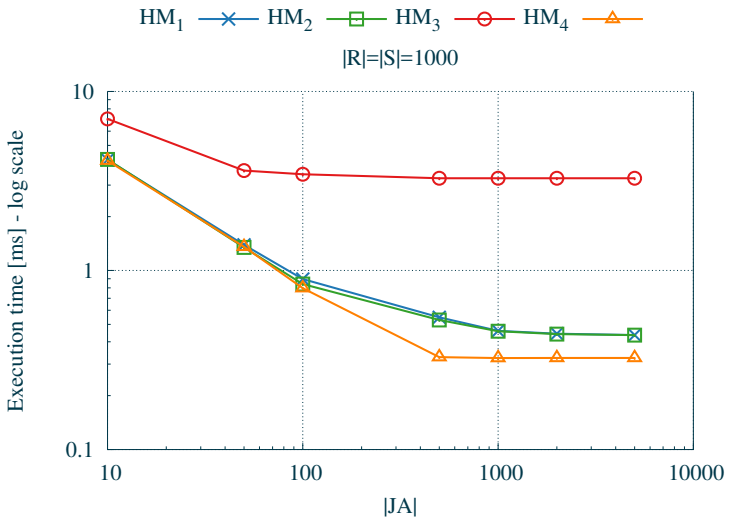
3.3.2.4 Device Utilization

Besides the single throughput of one join operator, the device utilization is an important fact. In join computation, many schemes are known to split and distribute the input dataset over multiple processing units. Therefore, it is interesting to know how many join operators could fit into the FPGA.

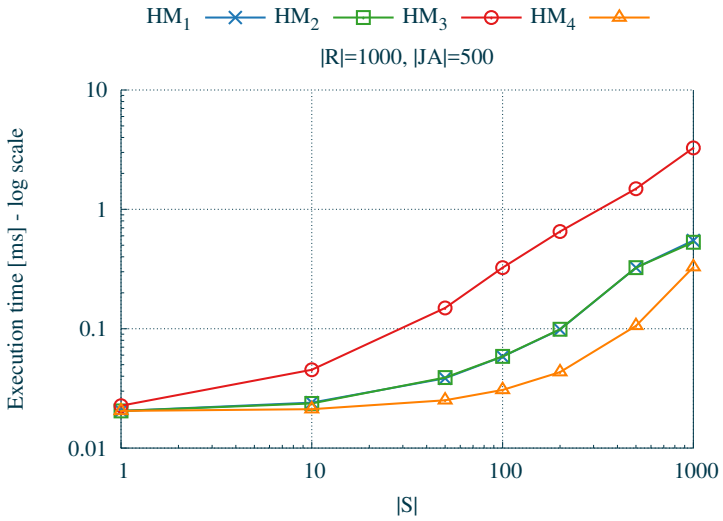
The provided resources of the used FPGA platform are given in Table 2.1 in Section 2.2.1. The numbers of utilized slices, LUTs and BRAMs of each join operator are summarized in Table 3.1. While the consumption of registers and LUTs is significantly low (maximum 1% of all slices), the available BRAM becomes a bottleneck. In that case, the BRAM suffices to implement 16 to 25 join cores, but a huge amount of logic resources remains unused.

Table 3.1 – Device utilization of join operators (total including the evaluation framework).

	Slices		LUTs		BRAM	
join	total	%	total	%	total	%
NLJ	519	0.87	1,362	0.57	45	5.9
MJ	489	0.82	989	0.41	30	3.9
AHJ	603	1.01	1,200	0.50	30	3.9
SHJ	642	1.07	1,509	0.63	45	5.9
HJL	582	0.97	1,090	0.46	45	5.9



(a) Vary the number of potential join partners.



(b) Vary the input size of one operand.

Figure 3.23 – Execution times of *Asymmetric Hash Join* (AHJ) with different hash masks.

3.3.3 Related Work

Teubner *et al.* [188] present a window-based stream join, called *Handshake join*. The approach lets the items of two data streams flow by in opposite directions to find join partners with each item they encounter. All items in a predefined window are considered in parallel to compute an intermediate result. Due to the window-based architecture and since the window size is limited by the chip area, this approach cannot be used to process joins of datasets larger than the relatively small windows. Following the same methodology using a pipeline of equal processing elements, Woods *et al.* [189, 190] present parallel data processing on FPGAs with *shifter lists*. They can be used to improve performance (up to 4X) in various applications such as *skyline operator* [191], *frequent item* [192], *n-closest pairs* and *k-means* [193].

Sukhwani *et al.* [194] investigate the use of FPGAs to accelerate data decompression and predicate evaluation in analytics queries. Halstead *et al.* [195] extend the approach by relational joins – in particular hash-based equi-join. The authors consider *star-schema* scenarios which are common in data warehousing. In such a scenario a central large *fact table* can be joined to multiple *dimension tables*, which are smaller tables containing attributes. Thus, all records of the dimension table are inserted into a hash table by applying a hash function on the corresponding join attributes (*build phase*). Afterwards, the fact table is streamed in and compared against the hash table (*probe phase*). The evaluation is executed on two tables, small enough to fit into the FPGA's internal BRAM. The execution in a cycle accurate simulator achieves a speedup of 11.3X compared to a commercial DBMS running on a 4.4 GHz multi-core CPU. However, the build phase is considered as one-time cost by the authors. Furthermore, the results on the FPGA are only simulated. Another flexible hash table design is proposed by István *et al.* [196, 197] but is not utilized in a database environment.

3.3.4 Summary of FPGA-based Join Operators

In this section, various join algorithms have been implemented as prototypes on an FPGA. The evaluation shows that the FPGA implementations are able to provide at least a competitive performance compared to general-purpose CPU with ten times higher clock rate. In most cases, a significant speedup of the FPGA prototype is achieved compared to the evaluated general-purpose CPU system. On the example of the NLJ it was also shown that a straight forward implementation on the FPGA does not guarantee performance improvements. But taking the characteristics of the FPGA into account, it can result in a great performance gain.

The software implementation, written in the programming language *C*, is highly optimized for maximum throughput rather than extensibility and maintainability of the software. Therefore, only primitive data types and no dynamic linking of method calls are used. However, in real-world database systems complexer data structures and concepts of object-oriented programming (e.g., inheritance) are used. Thus, these systems might not achieve the single join performance of the software used in this evaluation. Furthermore, each experimental run was executed 100 times to fully utilize caching capabilities of the CPU.

As previously described, the presented join algorithms differ in implementation complexity, resource consumption and as a consequence achieve different clock frequencies. Considering a complete operator graph the containing operators are integrated on the same FPGA device and are executed at the same frequency. After the feasibility of the hardware-accelerated join operator is shown in this section, the remaining operators need to be implemented to support complex SPARQL queries. Having an extended set of operators, the composition of multiple operators to a hardware-accelerated QEP will result in further improvements by taking advantages of the *intra-query* and *inter-operator* parallelism as well as *operator pipelining*. It is expected that the overall speedup will be more significant as the remaining operators are more suitable for pipelining than the join operator and more parallelism inside the FPGA can be taken into account.

3.4 Filter Operator

The filter expression restricts the solution space according to a given constraint and thus is essential for highly expressive queries. In the previous section, we implemented joins on the FPGA as one crucial element of the query execution. Due to the complexity and its memory consumption it is likely to execute joins later and reduce the amount of data in earlier processing steps. Typically, the query optimizer tries to push filter operations as close as possible to the data source in order to reduce the intermediate results and thus succeeding operators have to cope with less data.

Listing 3.4 shows a typical SPARQL query [30] utilizing a filter expression to return articles having more than five pages and published in February, March, May or April. The filter expression consists of boolean formulas which are checked for each intermediate result during query execution. If the boolean formula becomes true then the bindings of the variables remain in the intermediate result, otherwise they are discarded. As mentioned before the logical optimization tries to push filter expressions as close as possible to the data source. The early evaluation

Listing 3.4 – SPARQL example query with filter expression.

```
1 PREFIX swrc: <http://swrc.ontoware.org/ontology#>
2 SELECT ?article ?pages ?month WHERE {
3   ?article   swrc:month ?month .
4   ?article   swrc:pages ?pages .
5   FILTER ( ?pages > 5 && ?month > 1 && ?month < 6 )
6 }
```

of the filter expression reduces the size of intermediate results and consequently decreases the calculation costs for succeeding operators, such as joins. Due to their early involvement in the query evaluation an efficient and scalable filter operator is crucial for our hybrid query engine.

In the following sections, two approaches to implement the filter operator on an FPGA are presented. The previously described operator template supplies the input of two preceding operators to be processed by the current operator. The input bindings arrays are directly mapped to the data input signals as well as the resulting bindings array to the data output signal. However, the filter operator is a unary operator and consumes the actual data of only one preceding operator. Internally, it can just ignore the other input. This is feasible because there is actually no other preceding operator connected. In order to realize a higher flexibility, we use the second input interface for configuration data like the pattern or for setting the compare operation. The content and results of this section have been published by the author in [6].

3.4.1 Fully-Parallel Filter

The *Fully-Parallel Filter* presented in Figure 3.24 divides the bindings array w and the pattern p into its bindings w_i and p_i (*horizontal intra-operator parallelism*). The number i of bindings depends on the defined total width of the bindings array and the value width of a binding. Each pair of w_i and p_i is connected to a sub-filter SF_i. Each SF_i executes the actual comparison and decides whether the binding w_i fulfills the matching condition according to its corresponding pattern p_i. As this decision is binary, each sub-filter SF_i indicates a match by `'1'`, respectively a `'0'` in case of a mismatch. Finally, the matches of all sub-filters are mapped to a match vector m with m_i is the result of SF_i. In order to verify a global match the global filter operator compares the match vector with the match mask `'1..1'`. Obviously, choosing the match vector in this manner allows only conjugated conditions and is

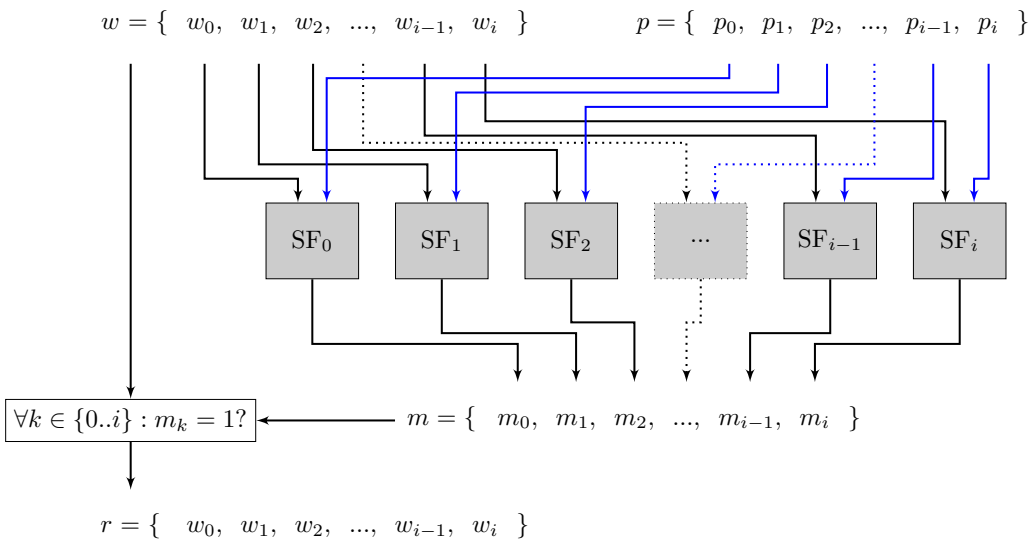


Figure 3.24 – Schematic of the Fully-Parallel Filter.

chosen for simplicity in this case. We will describe later why this approach does not affect the generality and allows any kind of condition with the same throughput.

An example execution of the Fully-Parallel Filter is shown in Figure 3.25. The bindings array contains the bindings of three variables and thus three sub-filters are used. Each sub-filter consumes a specific variable and the corresponding value of the pattern (see Figure 3.25(a)). The first incoming bindings array satisfies the whole condition (see Figure 3.25(b)). Thus, all sub-filters set their corresponding bit in the match mask to `'1'` and the bindings array is forwarded to the output. As the next bindings array does not match the condition in its first and third variable, the positions 1 and 3 in the match mask are set to `'0'` and consequently the bindings array is discarded (see Figure 3.25(c)). The last bindings array results in a global match as well and is forwarded to the output (see Figure 3.25(d)). Each sub-filter evaluates its condition within one clock cycle and thus evaluation of the complete condition requires only one clock cycle as well.

With increasing the total width of the bindings array, the number of sub-filters increases as well and the incoming bindings need to be distributed to the corresponding sub-filters. Thus, the distance in the FPGA between the port which receives the whole input bindings array and some of the sub-filters might become very long. This in turn might result in long signal paths which lead to longer clock cycles and thus limits the achievable clock rate of the whole design.

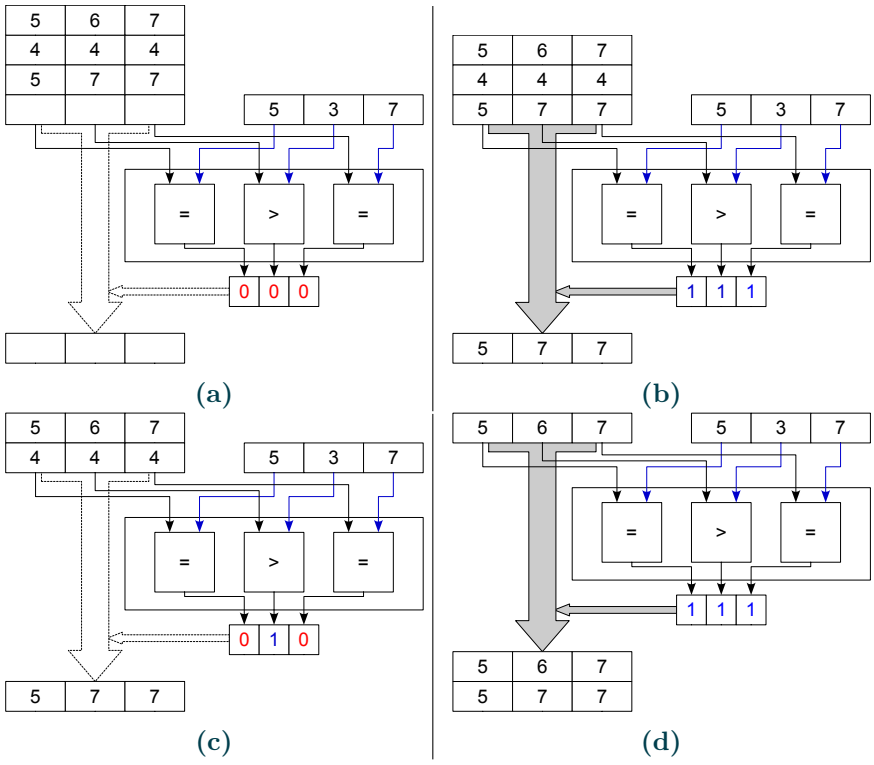


Figure 3.25 – Example execution of the Parallel Filter.

3.4.2 Pipelined Filter

The second approach of implementing an FPGA-based filter operator, the *Pipelined Filter* presented Figure 3.26, does not divide the input bindings array w, respectively the pattern p into several bindings and distributes them into the sub-filters. It applies the whole bindings array as well as the pattern to the head of pipelined sub-filter units (*vertical intra-operator parallelism*). Each SF_i is responsible for one specific binding in the data stream, and indicates a partial match by setting its output m_i. This output is used by the succeeding sub-filter and *iff* the partial match is set to `'1'` then the succeeding sub-filter will consume w and p to evaluate its own condition. Otherwise the whole bindings array will be dropped within the pipeline. Thus, if a bindings array reaches the end of the pipeline then all sub-filter expressions have been satisfied and it can be processed by succeeding operators. It is expected that the resource utilization is higher than for the Fully-Parallel Filter as the whole bindings array is forwarded from sub-filter to sub-filter which introduces additional registers to store the value. We can not shorten the bindings by the size of one binding with each pipeline stage because the overall filter

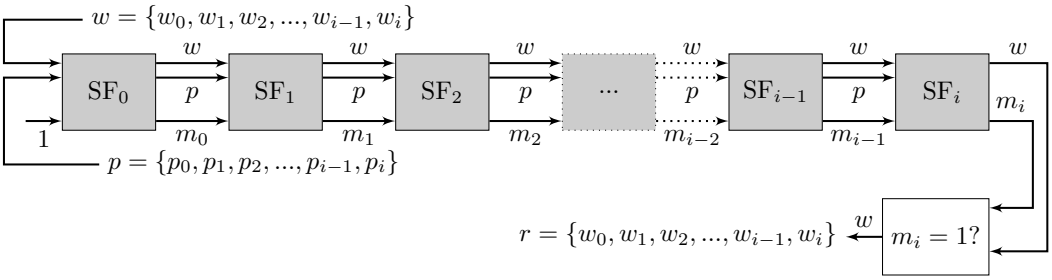


Figure 3.26 – Schematic of the Pipelined Filter.

already consumes the next bindings array to forward it to the head of the sub-filter pipeline. Otherwise in case of a match the complete bindings array would not be available anymore. However, even with a higher resource utilization we expect a better clock scalability with increasing the data width due to shorter signal paths.

An example data flow of the Pipelined Filter is shown in Figure 3.27. Each pipeline stage consists of one sub-filter and is responsible for one dedicated variable in the bindings array (see Figure 3.27(a)). The first incoming bindings array satisfies the first condition and thus is forwarded to the second pipeline stage. In the second stage it is checked for the next condition. Meanwhile, the next bindings array is applied in the first stage (see Figure 3.27(b)). As the second bindings array does not satisfy the first condition it is discarded and will not be evaluated in the subsequent stages. While the first bindings array is checked for the third condition, the first condition is checked for last bindings array (see Figure 3.27(c)). As the first as well as the third bindings array satisfy the whole filter expression they pass the whole pipeline and are forwarded to the output (see Figure 3.27(d)-(f)). Each sub-filter evaluates its condition within one clock cycle. Due to the pipeline structure no results will be return until the pipeline is filled. However, the Pipelined Filter can consume one bindings array per clock cycle and after the initial pipeline delay, one complete condition is evaluated each clock cycle.

3.4.3 General Filter Expressions

Obviously, the described approaches only implement conjunctive conditions. In case of the Fully-Parallel Filter the conjunction is introduced by the match vector as it is checked against the constant vector `'1..1'`. In order to realize disjunction we can define more constant vectors and connect the results of all vectors by logical OR; e.g., $C_0C_1C_2C_3 \vee C_4C_5C_6C_7$ can be evaluated by comparing the provided

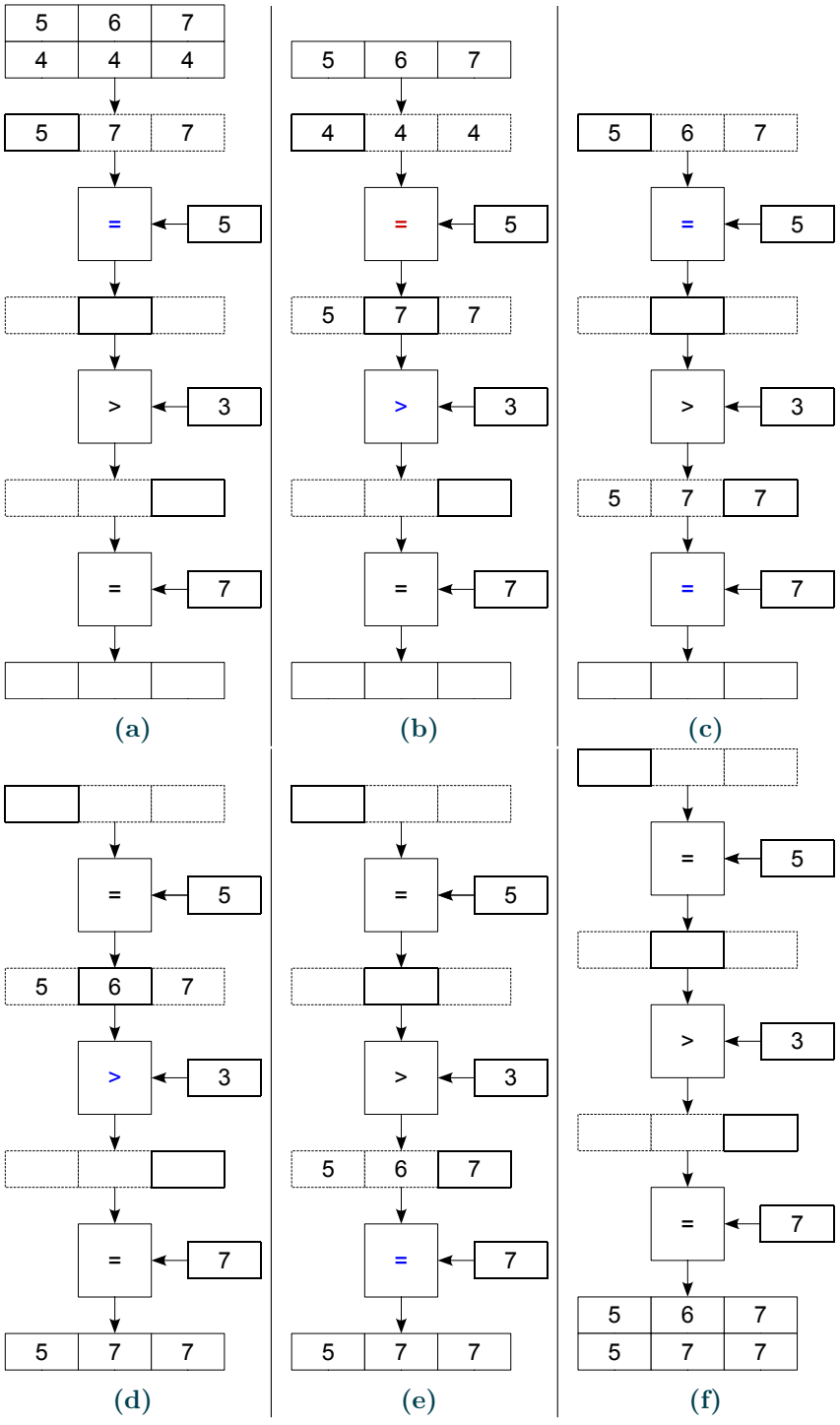


Figure 3.27 – Example execution of the Pipelined Filter.

match vector m with the match masks `'11110000'` respectively `'00001111'` and connect both results by logical OR. Due to the structure of the Pipelined Filter where each local match is only forwarded to the next sub-filter the implied conjunction can not be broken up. Introducing a match vector forwarded from sub-filter to sub-filter would cause more utilization overhead. Thus, taking the advantages of FPGAs into account we can configure multiple Pipelined Filter, each representing one conjunction, in parallel and finally evaluate their results. For an expression containing up to six disjunctions only one LUT is needed to evaluate the global result (see Section 2.2.1). This introduces a negligible signal propagation delay caused by the logic. However, more complex conditions consisting of various conjunctions and disjunctions need to be broken down into their atomic conditions and their single results have to be accumulated into the final result. Figure 3.28 shows an example for the complex condition $C_0 \vee (C_1 \vee (C_2 \wedge C_3))$ consisting of conditions C_0 to C_3. For each disjunction the bindings array is forwarded into two different branches. Thus, C_0 is evaluated in one branch and the remaining conditions are located in the other branch. As latter consists of another disjunction it is split into two branches again. Conjunctions are simply chained as described before. Note that the intermediate results of different disjunction branches have to be combined. Thus, it is necessary to ensure that a bindings array which satisfies all conditions on both branches is not duplicated in the result. Therefore, an introduced special union operator ($\uplus$) takes the results of two branches and compares their outcome. If both branches provide a result then only one is taken. However, due to the pipelining approach multiple bindings arrays might be within the pipeline at different stages. As a consequence the union operator might receive two different bindings arrays if the branches do not have the same depth. Therefore, empty pipeline stages ($\square$) are introduced in a branch if its depth is lower than the depth of the other branch. If both branches have the same depth then duplicated bindings arrays will arrive at the same time at the union operator which will forward only one. If a bindings array is discarded within a branch due to a failing condition then an *invalid* bindings array is forwarded to the union operator and detected by it.

3.4.4 Micro Benchmarks

With respect to comparability, the implementation for the general-purpose CPU written in the programming language *C* follows the iterator concept [74]. This is recommended especially for large datasets as intermediate results are not computed completely at once. This results in a smaller memory footprint for the operators but also causes some overhead. As described earlier the modular approach using the generic operator template introduced the signals read and valid to control the

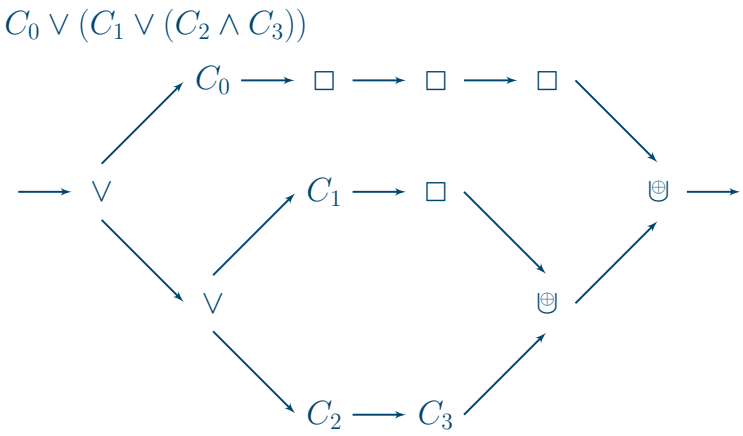


Figure 3.28 – Complex filter expressions can be broken down into multiple simple filter conditions. Each occurrence of a disjunction ($\vee$) results in two separate evaluation branches. Results of different branches need to be combined and introduced duplicates have to be removed (⊎). Additional pipeline stages ($\square$) are introduced to ensure the simultaneous arrival of potential duplicates at a union operator. Conjunctive condition ($\wedge$) are simply chained.

data flow. Consequently, both implementations for CPU as well as for the FPGA are affected and thus fairly comparable. We suppose that the variables needed for the filtering are located at the beginning of the bindings array. This can be done easily by connecting the circuit in this manner. Leftover data can be simply appended. We varied the following parameters to demonstrate the achievements under various conditions. The *match rate* (MR) describes the selectivity of the filter expression. Furthermore, we introduce the parameter *mismatch position* (MP) to model the fact that depending on the filter condition the filter operator can detect a mismatch without evaluating the whole expression, e.g., a complex conjunction is always false if the first subexpression is false. Thus, the MP determines after how many partial comparisons the filter operator can detect a mismatch, which allows it to skip the evaluation of the rest of the bindings array and go on with the next input. The label MP_i indicates a mismatch at position i. For instance, in case of MP_1, the filter can detect a potential mismatch within the first evaluation step. Of course, the filter still has to compare the whole bindings array in case of a match, as it does not know about the data composition. Vice versa, if the mismatch position is set to the last binding in the bindings array, then the filter will detect the global mismatch in the last evaluation step of the whole expression. Thus, the computational costs are almost[1] equal compared to a global match. The

[1] *almost* because in case of a global match some overhead for providing the result is introduced.

proposed architecture is generic and independent of the width of the bindings and bindings arrays. However, for evaluation purpose we consider in the first part of the experiments a conjunction of eight conditions and the mismatch position models the complexity of these conditions. Each data point represents the average of 10,000 single measurements respective to the parameters executed on a general-purpose CPU (Intel Core2 Quad Q9400, 2.66 GHz, 6 MByte cache, compiled with GCC 4.8.2). The data was preloaded into the main memory before measuring the time in order to ensure warm caches and to achieve the maximum performance on the CPU. The hardware-accelerated filter operators are described in VHDL. The FPGA runs at 200 MHz which is the default clock rate provided by the equipped clock generator (13 times lower than the CPU) – unless otherwise specified.

3.4.4.1 Throughput

By conception both approaches of the hardware-accelerated filter operators running at the same clock rate provide the same throughput. The difference in total processing time for the Pipelined Filter is negligible[2] and, thus, it achieves the same throughput as the Fully-Parallel Filter. Due to clarity we will reference in the next sections only to one of both hardware approaches. Later we will show the distinction between both hardware-accelerated approaches considering achievable clock rate and area consumption. Figure 3.29 shows the throughput of both FPGA-accelerated filter operators. Most noticeable is the fact that the FPGA performs independently of the data structure and shows for different MR and MP the same throughput. Figure 3.30 outlines the total throughput of the software-based and hardware-based filter operators. Contrarily to the hardware-accelerated filter operators, the performance of the software solution is heavily affected by the composition of the bindings arrays. If the match rate is low then the position of the mismatch has a high impact on the throughput, e.g., if MP_8 and MR=0 then the approach on the CPU has to verify all subexpressions in order to detect a mismatch in the last condition. Thus, the performance drops. For MP_1 and the same match rate this approach detects the mismatch in the first subexpression and consequently drops the bindings array immediately and requests the next bindings array to be evaluated. Increasing the match rate results in a lower sensitivity regarding MP of the software-based approach because a mismatch is detected in a late evaluation step. The computational effort to detect a match or a mismatch becomes the same. Figure 3.30 shows the corresponding speedup factors as dashed lines. At a match rate of 0% and MP=1 (the mismatch is detected in the first condition), the FPGA-based filter still achieves a speedup of 1.6X (60% faster). At higher match rates a significant higher speedup of up to 5.5X is reported.

[2] the Pipelined Filter introduces a delay of 8 clock cycles in order to fill the pipeline.

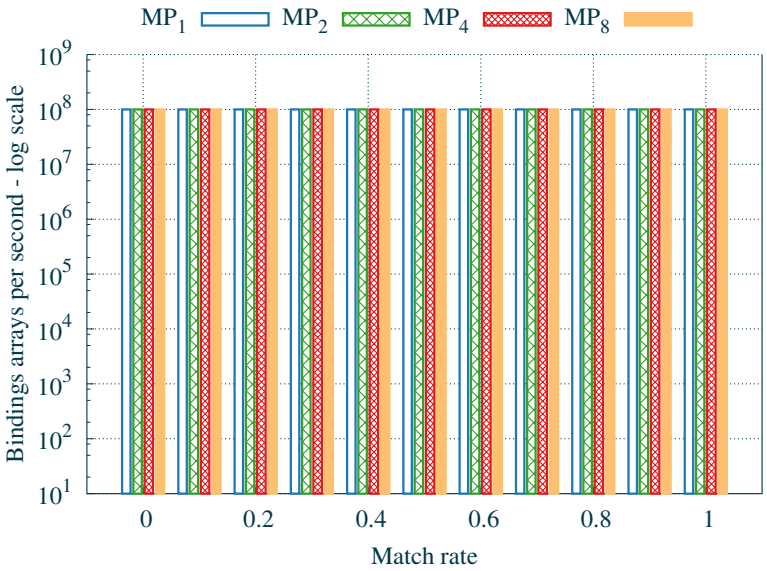


Figure 3.29 – Throughput of *Fully-Parallel* and *Pipelined Filter* executed on the FPGA. Both approaches achieve the same total throughput.

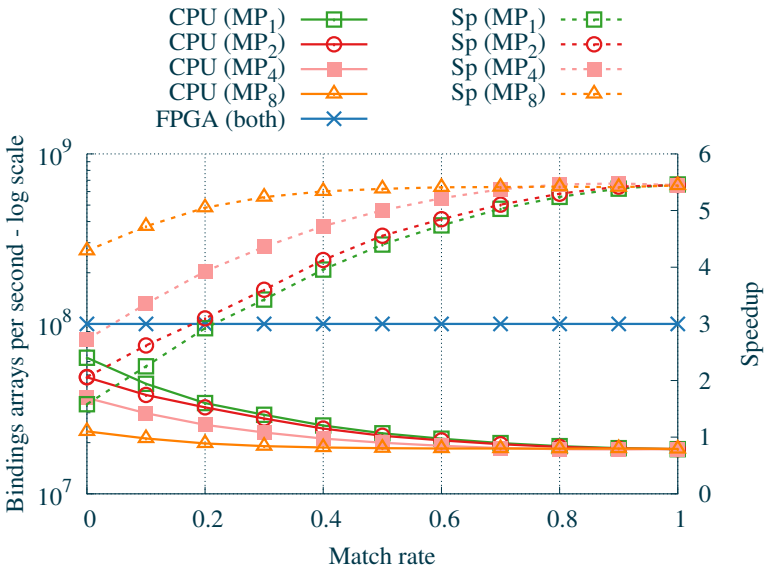
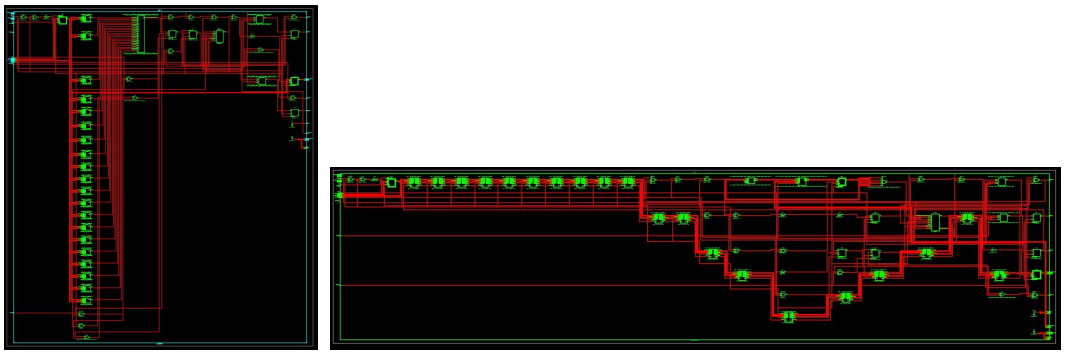


Figure 3.30 – Throughput of the software-based respectively hardware-based filter operator (solid lines) and resulting speedup *Sp* (dashed lines).



(a) Fully-Parallel Filter. **(b)** Pipelined Filter.

Figure 3.31 – *Register Transfer Level* (RTL) of filter operators.

3.4.4.2 Scalability

At the beginning of the evaluation section of the filter operator we mentioned that the throughput of both approaches in the FPGA is equal at the same clock rate. Due to the different design ideas and their complexity the resource requirements are different. Figure 3.31 shows both filter operators composed of 20 sub-filters as RTL. The Fully-Parallel Filter (see Figure 3.31(a)) consists of multiple parallel sub-filters while the sub-filters in the Pipelined Filter (see Figure 3.31(b)) are connected as a chain. This results in a trade off between resource consumption and timing requirements which effects the maximum achievable clock rate and thus the effective total performance. To obtain the maximum achievable clock rate we synthesized the whole design under varying the data width of the bindings array. We tightened the clock frequency by constraining it in the designs UCF file until PAR is not able to meet the timing requirements anymore. Furthermore, we varied the design goal and choose the highest achieved value for each filter type and parameter. Additionally, we noted the resource consumption in terms of utilized slices. Figure 3.32 presents the maximum achievable clock rate depending on the numbers of sub-filters respectively width of the bindings array. Each binding has a width of 32 bits. Furthermore, the number of utilized slices is shown. With increasing the amount of sub-filters the maximum frequency decreases for both approaches but the Pipelined Filter always reaches a higher clock rate (minimum 10% for 12 bindings per array) than the Fully-Parallel Filter. The Fully-Parallel Filter under 200 MHz for 16 or more sub-filters. As a drawback, the resource utilization of the Pipelined Filter is in general higher due to the additional needed registers between the sub-filter stages. However, in case of 20 bindings per array the Pipelined Filter achieves a 19% higher clock rate than the Fully-Parallel Filter while using 16% more slices. One advantage of the FPGA-implementation is the

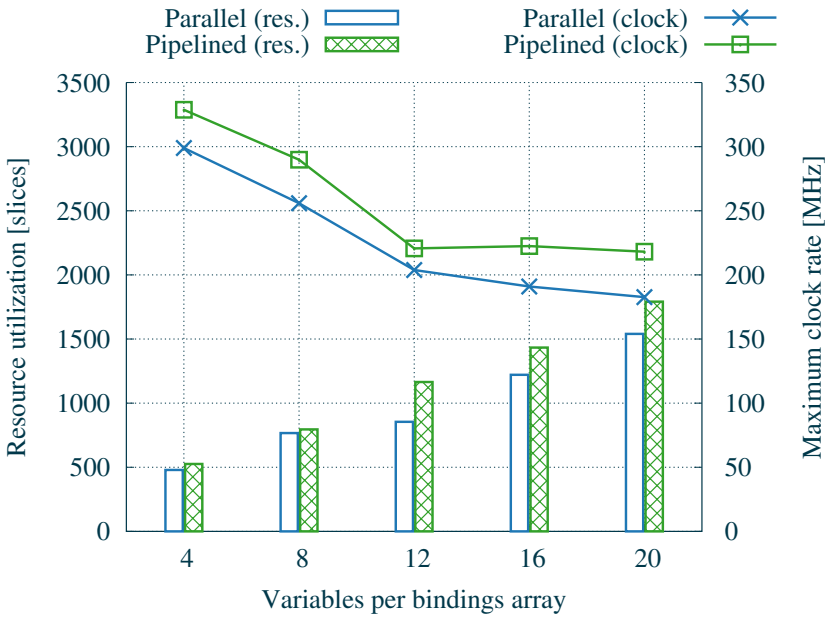


Figure 3.32 – Resource consumption (bars) and maximum achievable clock rate (lines) of both approaches (after *Place & Route* (PAR)). The Pipelined Filter has a higher resource utilization due to additional pipeline stages but achieves higher clock rates than the Fully-Parallel Filter.

fact that the throughput increases linearly with increasing the clock cycle. This results in an even higher throughput than reported in the section before. For example considering 8 bindings per bindings array, the Pipelined Filter achieves a clock rate of 289 MHz which directly leads to a total throughput of 144 million bindings arrays per second. The Parallel Filter achieves a clock rate of 256 MHz respectively a throughput of 128 million bindings arrays per second. In summary, the Pipelined Filter reaches a higher throughput at its maximum possible clock frequency.

3.4.5 Related Work

Woods *et al.* [198, 199] present *Ibex*, an intelligent storage engine for the relational database MySQL that supports off-loading of complex query operators using an FPGA. The FPGA is integrated into the data path between data source and host system. Data is read from an SSD attached to the FPGA and is directed

through a pipeline consisting of modules for tuple parsing, projection, selection and *Group-By*. The proposed system uses *runtime-parametrization* instead of DPR to adapt to a given query structure. This means that before fetching a table from the disk, the table schema (or catalog information) including projection flags is stored in local RAM. The predicates of the selection module are parameterizable (column, comparator, constant) as well but allow only the comparison of a column value with a constant. However, multiple predicates can be combined via boolean operations. The output signals of all base predicates together compose a lookup address in a truth table (BRAM) returning the result of the combined expression. The truth table is computed in software and loaded into the corresponding BRAM on the FPGA. More complex expressions, such as predicates on two columns, are evaluated in software. The Group-By operation is implemented using a hash table. A bit mask determines which parts of the record are used as *group key*. However, in order to avoid stalling[3] the hash table, duplicated tuples are *bypassed* to the host where they are processed in the software of the storage engine before passing on the tuples to MySQL's query engine. Even with a workload causing 90% bypass tuples, a speedup of 4.5X is achieved but also shows some performance inefficiencies of MySQL itself. In some cases the authors report speedups of up to 32X. Note that no indices are used. István [200] beneficially uses this system to generate statistics and histograms as a *side effect* of data movement in the data path. Typically, this is performed as a batch job, separately from query processing. The generation of histograms is computing intensive, but they are important for query planning. The proposed prototype calculates histograms faster and with similar or better accuracy than two commercial databases. However, the approach of an intelligent storage engines aims to utilize the whole FPGA chip area. In this work, we assemble multiple different operators to be reconfigured on the FPGA.

3.4.6 Summary of the FPGA-based Filter Operator

In this section, we presented two approaches to execute filter operators on an FPGA. The presented results show that both approaches obtain significant speedups in comparison to a general-purpose CPU system. While both approaches on the FPGA reach the same total throughput at the same clock rate, they slightly differ in scalability according to the width of the bindings array. Both implementations significantly defeat the software solution executed on a general-purpose CPU. At a 13 times lower clock rate the Fully-Parallel and the Pipelined Filter achieve a speedup of more than 5X. Both approaches can process one bindings array within two clock cycles, which results in a total throughput of 100M items per second at

[3] tuples with duplicate hash keys which are not from the same group require collision resolution

200 MHz with the potential of higher throughput through higher clock rates. The theoretical throughput of one bindings array per clock cycle is not reached due to the flow control introduced by the operator template. If an operator consumed the predecessor's output then it raises the corresponding read flag. The preceding operator notices the invalidity of its current output and needs at least one clock cycle to provide the next bindings array. Thus, one clock cycle delay is introduced for synchronization between two operators. Without this synchronization it can not be guaranteed that no data gets lost. This is critical especially if a lower operator can not consume intermediate results as fast as the upper operator produces them. If both, the current and succeeding operator, are able to handle incoming bindings arrays at a maximum speed of one bindings array per clock cycle then this can be realized by simply setting the read and valid flags to constant `'1'`. Consequently, the filter operator could double the total throughput to 200M bindings arrays per second. Additionally, an *almost overload* flag could be added. Succeeding, potentially slower operators could raise this signal after reaching a threshold. The preceding operators can slow down using the read/valid signals in order to avoid data loss. Furthermore, the filter expression is supposed to reduce the amount of data in an early step of the overall query evaluation. Depending on the selectivity of the expression the filter operator might consume data at a maximum speed of one bindings array per second but will not provide data at the same speed because many bindings arrays might not satisfy the filter expression and consequently will be dropped. In summary, the hereby presented filter operators satisfy the required efficiency and scalability necessary for the early involvement in our hybrid query engine.

3.5 Additional Operators

In the following section, we outline the remaining supported operators to increase expressiveness of our hybrid query engine. Contrary, to the previous two operators (join and filter), the implementation of the following operators is relatively simple. Thus, only the general functionality is outlined and no explicit performance evaluation is given. The concrete parametrization will be further explained later in Section 4.2.3. Additionally, not supported features of SPARQL are examined in this section as well.

3.5.1 RDF3XIndexScan

In the final hybrid architecture the evaluation indices in terms of B^+-trees are stored in the host's HDDs and maintained by the LUPOSDATE software system.

However, the evaluation indices are providing RDF triples respectively ID triples which are transferred to the FPGA platform. Each index scan in the software environment has as a counterpart a *RDF3XIndexScan* operator instantiated on the FPGA (see Figure 3.33). The RDF3XIndexScan receive triples and maps the triple components (s,p,o) into the corresponding ranges of the data signal which represent the currently processed bindings array structure. While receiving of triple components operates at the same clock rate as the interfacing core (e.g., PCIe), the provision of bindings arrays to succeeding operators runs at a common operator graph clock rate. Thus, the clock domain of the arbitrary communication interface and the query structure in the FPGA is completely decoupled. Assuming the communication interface provides triple data at maximum speed (one triple per clock cycle) the RDF3XIndexScan operator generates corresponding bindings array at the same rate.

3.5.2 Projection

The *Projection* carries out the SELECT clause of the SPARQL query. Thus, variables used during query evaluation but not required in the final result are discarded typically at the end of the QEP (see Figure 3.34). As the bindings array directly corresponds to wires on the FPGA the wires representing a variable to be discarded are simply not connected to the operator's output. Although this would allow an *instant* projection, the Projection temporarily stores the projected result within one clock cycle in an output register due to our conception of an operator pipeline.

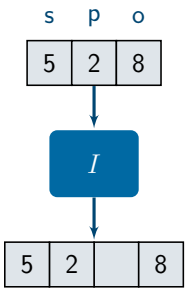


Figure 3.33 – RDF3XIndexScan operator recieves triples and maps the triple components (s,p,o) into the corresponding positions in the bindings array.

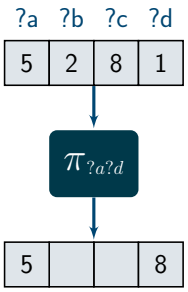
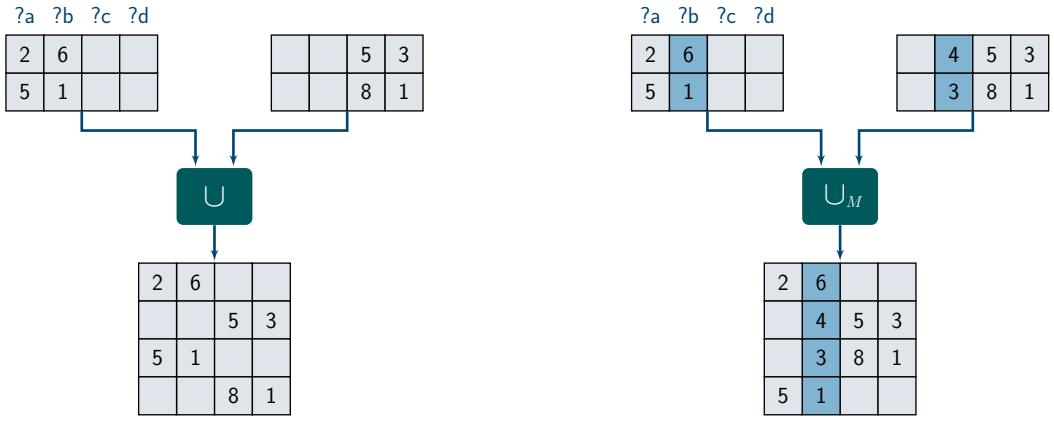


Figure 3.34 – Projection operator discards bound values of variables which do not appear in the final result.



(a) Simple Union. **(b)** Merge-Union.

Figure 3.35 – Union operators.

3.5.3 Union / Merge-Union

The *Union* operator builds the union of the results provided by its two predecessors (see Figure 3.35(a)). It is not necessary that the same variables are bound at this point in the query execution. The implementation is straight forward. If the left input is valid then it is forwarded to the output. Otherwise, if the right input is valid then it is forwarded to the output. If both preceding operators indicate to be finished then the Union sets its finished flag as well. The *Merge-Union* shown in Figure 3.35(b) is an extension and requires sorted input regarding common variables. It unifies the two inputs such that the result is still sorted. Therefore, it compares the bound values of the common variables and forwards the smaller input to the operator's output. If one of the preceding operators indicates to be finished then the remaining bindings arrays of the other operator are forwarded to the output. Both union operators take two inputs R and S but can provide only one result at a time to their output. Thus, they require at least $|R| + |S|$ clock cycles.

3.5.4 Limit and Offset

The *Limit(i)* operator is typically located directly at the end of the QEP and forwards up to i bindings arrays to its output. In Figure 3.36, the Limit operator is parametrized with two and thus only two of four bindings arrays remain in the result. If the limit is reached or its preceding operator indicates to be finished then the Limit operator sets its finished flag. In turn the *Offset(i)* operator skips the first

i resulting bindings arrays and simply passes the remaining bindings array to its successor. In Figure 3.37, the Offset operator is parametrized with two and thus the first two incoming bindings arrays are discarded from the result. Internally both functionalities are simple counting operations. In order to selects different subsets of the query result the Limit and Offset operators can be combined. Both operators can process one bindings array per clock cycle.

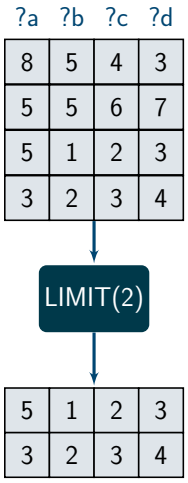


Figure 3.36 – Limit operator.

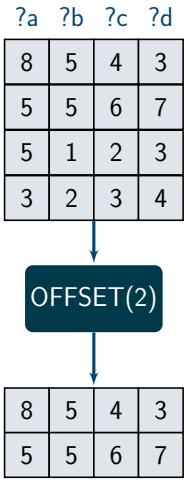


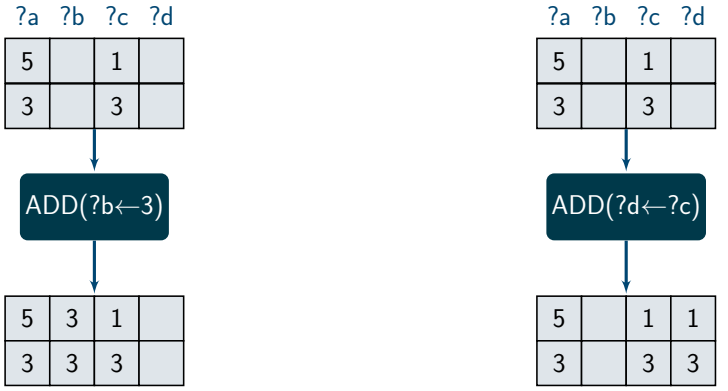
Figure 3.37 – Offset operator.

3.5.5 AddBinding / AddBindingFromOtherVar

The *AddBinding* respectively *AddBindingFromOtherVar* operators shown in Figure 3.38 might occur due to a logical optimization rule for constant respectively variable propagation [201]. E.g., if a simple filter expression checks a variable and a constant for equality then the constant can by pushed up in the operator graph replacing the variable with the constant value in the triple pattern of the corresponding index scan operators. Consequently, retrieved bindings arrays matching the modified triple pattern implies that the filter condition is already satisfied when provided by the index scan operator. Both operators can process one bindings array per clock cycle.

3.5.6 Unsupported Operators

At this stage, we support a subset of SPARQL 1.0 using the previously described operators. The Sorting and Distinct operators are not implemented so far. Both



(a) Add binding from constant value. **(b)** Add binding from another variable.

Figure 3.38 – AddBinding operators.

must temporarily store the whole intermediate result of their predecessors, and thus have enormous memory requirements which can not be satisfied by using only BRAM. Koch et al. [185] utilize the entire FPGA in their sorting architecture and thus is not applicable in our approach. However, extending our approach with additional memory interface such as DDR3 and with support for mass storage devices like SSDs we will be able to implement these operators in the future. The OPTIONAL operator (*left outer join*) can be derived from already implemented join operators. Furthermore, SPARQL tests such as *isIRI* and aggregation functions are not implemented, yet. Some redundant features like the SPARQL 1.1 paths (restricted to those without repetitions) can be partly covered by query rewriting.

3.6 Summary

In this chapter we have introduced the operator template and described the implementation of the join and filter operator. The evaluation shows possible speedups of up to 10X for the join operator respectively up to 5.5X for the filter operator. However, the evaluated datasets are small and communication costs are not considered yet in the performance evaluation because of the early prototype stage of the system. It might be questionable if the transfer of the data from the host system to the FPGA and back could devastate the achieved speedup. First of all, in the completely software-based query evaluation the data has to be loaded from the HDDs and all the data can not be provided to the CPU immediately as well. As the FPGA works in parallel to the host system, a hardware-accelerated QEP can begin with the evaluation right after the first bindings arrays have arrived. After

the initial latency needed to provide the first bindings arrays a steady flow of (i) sending new triples to the FPGA, followed by (ii) processing them on the FPGA through several pipelined and parallelized operators and finally (iii) sending the resulting bindings arrays back to the host system, can be established. Furthermore, in this chapter only single operators are evaluated on the FPGA which does not benefit from *inter-operator* parallelism. Consequently, the composition of multiple operators to a complete hardware-accelerated QEP will result in performance improvements by taking advantages of the *intra-query* and *inter-operator* parallelism as well as *operator pipelining*. However, when it comes to the execution of complete operator graphs we have to categorize the operators into *non-blocking* and *blocking* operators. Non-blocking operators are operators which can start the computation without receiving the whole input of the preceding operators and thus are highly suitable for *true* operator pipelining. Mainly those are operators which have a small or no memory footprint such as *Filter* or *(Merge-)Union*. Each of these operators can process one intermediate result in a constant time k. Consequently, a pipeline of several operators introduces an initial latency of $\#operators * k$ clock cycles. After the pipeline is filled each k clock cycles a new intermediate result is processed. Furthermore, also operators with medium/large memory footprint can support the pipelined execution. The proposed SHJ consumes continuously the input of both preceding operators and provides intermediate results before consuming the whole input. Also the MJ can be considered as a non-blocking operator. Because it retrieves sorted data, typically in real-world case-studies the amount of cached inputs of one preceding operator which have the same value at join attribute as the input of the other preceding operator is small. Additionally, the proposed implementation further improves pipelining capabilities because while joining matching pairs, the input of both preceding operators is consumed until a match is found and thus are already available after the cached data is joined. Then MJ can proceed faster with joining the next matching pairs. The LUPOSDATE system intensively uses indices and thus many of the join operations are executed using MJ. Blocking operators (Pipeline-Breaker) are operators which can not start the computation without receiving the whole input of at least one preceding operator. This class of operators contains all sorting operators as well as some join operators like AHJ or NLJ. Temporarily storing the whole input also implies a high demand of memory space. This amount of space is practically reduced by the LUPOSDATE architecture because of the previously described mapping of strings to integer IDs. Besides the small memory footprint of one variable in the bindings array, typically the number of variables in real-world queries is small as well and consequently the integrated memory (BRAM) of the FPGA can be used efficiently. Despite the true pipelining between operators, the FPGA gains some advantages regarding the way how to transfer intermediate results between two operators. While CPU-based systems have to store intermediate results *between* succeeding operators in additional

memory/registers, the QEP on the FPGA allows a natural flow of the bindings arrays between successive operators. Furthermore, the execution of the QEP in software is always effected by the operating system (e.g. scheduling of processes, moving of memory pages). Each operator on the FPGA performs the processing in dedicated hardware blocks and several processing cores do not interfere. In the following chapter, we will show how to automatically transform a given query into a configuration suitable for an FPGA.

4

Automated Composition and Execution of Hardware-accelerated Operator Graphs

In the previous chapter we introduced the operator template to assist the idea of composing hardware-accelerated *Query Execution Plans* (QEPs) consisting of arbitrary operators. As queries are typically unknown at system deployment time, a static approach is not feasible and not flexible to cover a wide range of queries at system runtime. Therefore, we introduce a runtime reconfigurable architecture based on an FPGA which is transparently integrated into the *Semantic Web* (SW) database LUPOSDATE. At system runtime, the proposed approach dynamically generates an optimized hardware accelerator in terms of an FPGA configuration for each individual query and transparently retrieves the query result to be displayed to the user. During hardware-accelerated execution the host supplies triple data to the FPGA and retrieves the results from the FPGA via PCIe interface. The benefits and limitations are evaluated on large-scale synthetic datasets with up to 260 million triples as well as the widely known *Billion Triples Challenge* (BTC) with over one billion triples [202]. Parts and results of this chapter have been published in [8, 9].

4.1 Hybrid Architecture

In this section we introduce our hybrid query engine in terms of processing tasks provided by the software system and an architectural overview of the runtime reconfigurable accelerator based on an FPGA.

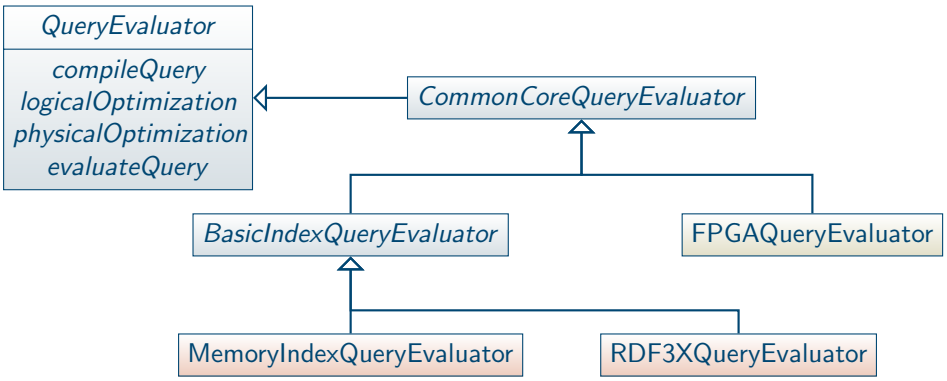


Figure 4.1 – Class hierarchy of query evaluators in LUPOSDATE (simplified).

4.1.1 Integration into LUPOSDATE

LUPOSDATE follows a modular approach to integrate new *query evaluators* and to exchange them during system runtime. In order to deploy a new query evaluator it has to be derived from the abstract class QueryEvaluator or one of its abstract subclasses, e.g., CommonCoreQueryEvaluator and BasicIndexQueryEvaluator (see Figure 4.1). Latter provide some initialization routines for often used setups and further simplify the development of new query evaluators. Besides compiling the query each evaluator provides methods for the logical and physical optimization as well as for the query evaluation. Among other query evaluators, LUPOSDATE provides the in-memory evaluator MemoryIndexQueryEvaluator and the RDF3XQueryEvaluator which operates on six evaluation indices as described in Section 2.1.3. In this work we integrate the FPGAQueryEvaluator into the LUPOSDATE framework to support a transparent query execution on the FPGA in order to increase query performance. It introduces another processing step after the physical optimization which maps the query structure onto FPGA resources. Afterwards the query is evaluated collaboratively by the host and the FPGA platform. Details are described in the following sections.

4.1.2 Hybrid Work Flow

Figure 4.2 shows the general processing flow of our hybrid query engine. First the user submits a query which is transformed into an operator graph by the LUPOSDATE system. On the operator graph several logical and physical optimizations are applied [14] resulting in the query's *Query Execution Plan* (QEP). Afterwards, the QEP is analyzed for unsupported operators, e.g., sorting operators. In case such an operator was found the query is evaluated completely by the

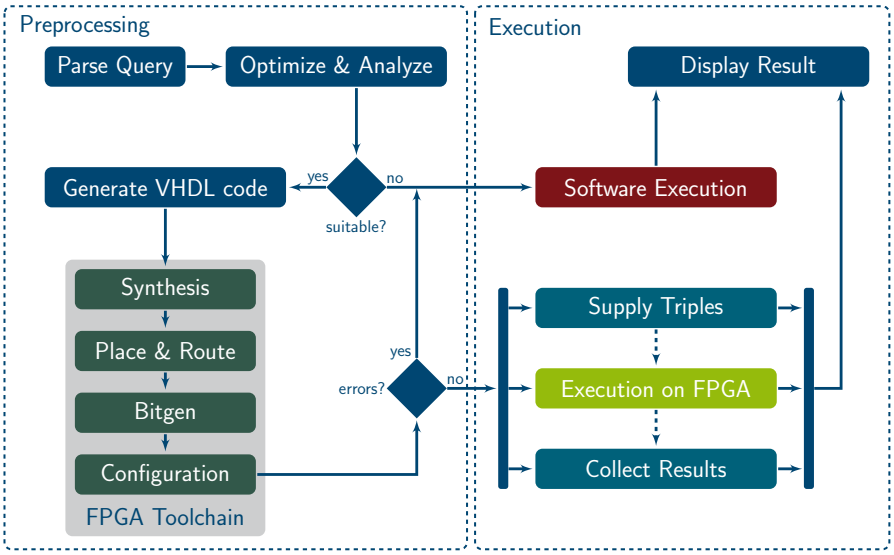


Figure 4.2 – Flow chart of the hybrid system.

software engine on the host system. If all operators are supported then our new extension traverses the operator graph and generates a VHDL file which represents the given query. A detailed description of this process is given later in Section 4.2. The full FPGA toolchain (*Xilinx Synthesis Technology* (XST), mapping, *Place & Route* (PAR)) is applied on the VHDL description of the circuit. The resulting bitfile is configured on the FPGA and the query is ready to be executed. Again, in case of any error (e.g., translation failed caused by lack of FPGA resources) the query is evaluated in software. This always preserves a running system. If no error occurred then the query is executed on the FPGA. In fact, during execution the host and FPGA work jointly. While the query operators are executed on the FPGA the host covers the following two tasks during query execution:

a) **Provision of input triples:** Depending on the triple patterns given in the query, a collation order for each index scan is chosen (see Section 2.1.3). The underlying data structure (B^+-tree) iteratively returns all ID triples satisfying the triple pattern. The ID triples are written in a buffer on the host and block-wisely passed on to the FPGA.

b) **Retrieval and post processing of results:** Concurrently, the host is requesting resulting bindings arrays from the FPGA. Usually this returns multiple results but not necessarily all at once. Thus, results are requested again until query execution is finished. In further post processing steps the result's low level representation (array of integer) is packed into higher data structures (literal and bindings objects) in order to return the result back to typical soft-

ware flow of LUPOSDATE and its modules for presenting the result to the user or submitting it to another application.

4.1.3 Hybrid Query Engine

Figure 4.3 shows the architecture of the hybrid query engine consisting of two parts: the host, which provides higher functions (such as user interface, query optimization, maintaining of evaluation indices, etc.), and the FPGA as an accelerator for query execution. The communication between both units is based on PCIe. The FPGA is divided into static logic and one *Reconfigurable Partition* (RP). The static logic contains modules which are independent of the actual query structure. Typically, those are modules for communication and memory interfaces. In this case it contains the PCIe *Endpoint* (EP), an *Internal Configuration Access Port* (ICAP) module and a managing module, the *Query Coordinator* (QC). The main task of the QC covers delivery of incoming triples to the corresponding index scan operators (RDF3XIndexScan) in the dynamic partition as well as retrieval and serialization of bindings arrays (forming the final result). The RP can be reconfigured at runtime with a bitfile representing any arbitrary QEP (limited by chip area). All QEPs contain index scan operators I_n with $n \in \{1..k\}$ with k fixed but adjustable at system deployment (currently $k = 8$). Each index scan represents a triple pattern (with possibly bound components) and maps the incoming triple components into the corresponding variable positions in the bindings array. Succeeding operators (such as Join, Filter, Union, etc.) consolidate and filter partly bound bindings arrays to combined final results. These results are returned to the QC in the static logic in order to transfer them to the host. On the host, the result is materialized in higher data structure for further processing such as displaying the result to the user or delivery to the calling application. The triples are supplied using *virtual streams* – one for each index scan operator in the QEP – between host and FPGA. Virtual streams logically divide the PCIe interface into multiple channels. In the FPGA and on the host, each stream provides a dedicated interface including dedicated buffer (BRAM respectively main memory). For incoming triple data, each stream is associated with one index scan operator. The result is always sent back on the first stream. Remaining streams can be used in the future to support processing of multiple queries on the FPGA at the same time. In the following section, we describe how to automatically obtain a configuration for the RP during system runtime.

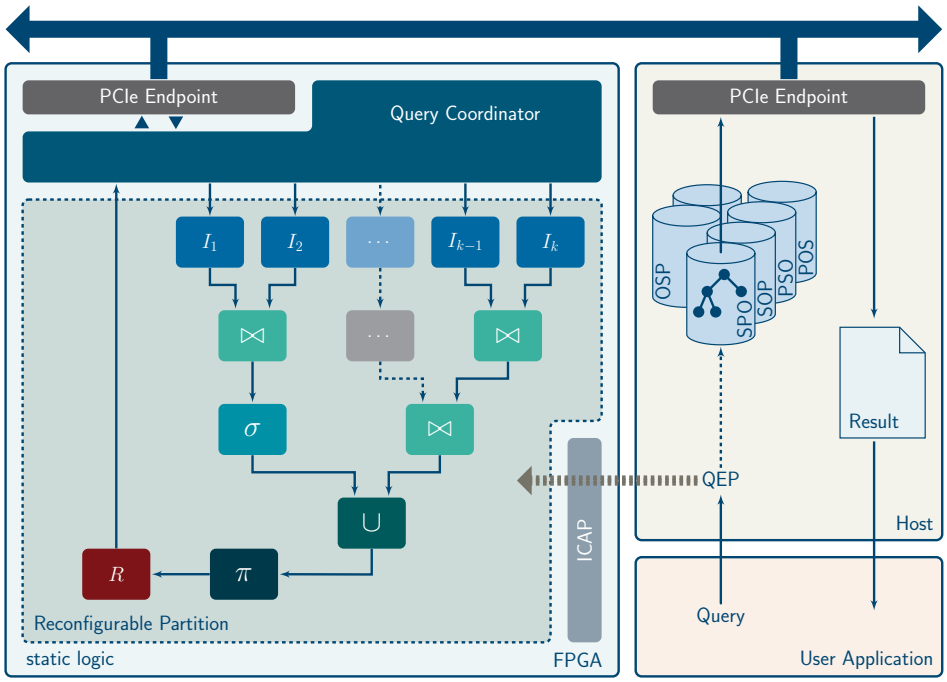


Figure 4.3 – Architectural overview of the integrated hybrid query execution engine. The FPGA's logic fabric is divided into the static logic and one large *Reconfigurable Partition* (RP). According to the *Query Execution Plan* (QEP) of a given query, a configuration is generated and configured via *Internal Configuration Access Port* (ICAP) onto the FPGA. Afterwards, triples from the indices located on the host are sent to the FPGA. The query results is determined within the FPGA and sent back to the host.

4.2 Automated Composition

The configuration of the RP is generated automatically for each query. Therefore, we use a VHDL template and the operators presented in Chapter 3 to compose a VHDL file which represents a given query. The template consists of a dynamic part and a static part (not to be confused with the static logic). The components of the template are summarized in Table 4.1. In the following section, a detailed description for each component of the template is given.

Table 4.1 – Components of the VHDL template to generate the *Reconfigurable Module* (RM) to be configured into the *Reconfigurable Partition* (RP).

Static signals & constants definition
Dynamic signals & constants definition
Dynamic signal assignment
Dynamic mapping of index scans
Static instantiations of entities
Dynamic instantiations of entities
Static *glue* logic

Listing 4.1 – Record type op_connection to connect two consecutive operators.

```
1 type op_connection is record
2  read_data : std_logic;
3  data      : std_logic_vector(DW-1 downto 0);
4  valid     : std_logic;
5  finished  : std_logic;
6 end record op_connection;
```

4.2.1 Static Components

Each query contains a result operator which is connected using the signal results of the record type op_connection shown in Listing 4.1. It is used as an interface between the dynamically generated QEP and the static code which takes the query results and serializes them into the PCIe engine. The record type op_connection is an essential part in the dynamic part of the VHDL template as well and will be explained in detail in Section 4.2.2. Additionally, the static instantiation of entities covers a reset generator, a cycle counter (for debugging purposes and to evaluate the raw FPGA performance) and signals to connect the logic in the RP with the QC in the static logic.

4.2.2 Dynamic Components

In the following sections, we describe how the query operators are dynamically instantiated and connected by composing VHDL code fragments based on the operator template. The resulting VHDL file describes the whole structure of the given query.

4.2.2.1 Operator Instantiation and Interconnects

The operator instantiation bases on the operator template introduced in Section 3.2. It defines the input and output signals which need to be implemented by each operator. Each operator can have up to two preceding and one succeeding operator. While traversing the QEP in LUPOSDATE, for each visited operator an ID X is assigned and the signals opXinput1, opXinput2 and opXoutput1 are defined (see lines 1 to 3 in Listing 4.2). Additionally, an entity of a specific operator with type OperatorType is instantiated (line 5). In the port map (lines 11 to 27) the internal operator signals are connected with the previously defined op_connection signals. The generic map (lines 6 to 10) is used to parameterize this particular operator. All operators have the generics for the data and value width in common which correspond to the width of the bindings arrays and variables. Therefore, two constants – BINDINGS_ARRAY_WIDTH and BINDINGS_WIDTH – are dynamically defined and thus assigned to all instantiated operators. Furthermore, each operator can have additional individual generics. These extensions will be described in Section 4.2.3.

The actual wiring of two consecutive operators is shown in Listing 4.3 and uses the previously assigned operator IDs to systematically connect two operators. As the output of operator X is used as (left) input of the operator Y this implies that X is the predecessor of Y (see lines 2 to 4). The read_data flag indicates operator X that operator Y has read the provided data and thus can provide the next data (see line 1).

4.2.2.2 Triple Input Mapping

The input mapping of triples to index scan operator is realized implicitly by assigning one individual virtual stream to each index scan operator within the QC. The synchronization between the PCIe EP at host and FPGA is implicitly covered by the PCIe core. The PCIe implementation is realized using the freely available *Xillybus* core [203] which allows to divide one physical interface into multiple logical (or virtual) stream interfaces. More details about the Xillybus core and a performance analysis is given in Appendix A. Using virtual streams is a great advantage if the triples of different index scans are not consumed at the same rate. If one index scan does not consume the triples as fast as they arrive then the buffers within the FPGA and the host memory get filled. This will finally block the writing method within the host application. Assuming that all triples for all index scans are sent using the *same* stream this might cause that the triples of a blocked index scan are blocking the commonly used stream. As a consequence, the other index scans

Listing 4.2 – Instantiation of operator X. Each operator is identified by its ID which is used to dynamically define corresponding signals.

```
signal opXinput1  : op_connection;
signal opXinput2  : op_connection;
signal opXoutput1 : op_connection;
[...]
operatorX : entity work.OperatorType(arch)
generic map(
  DATA_WIDTH  => BINDINGS_ARRAY_WIDTH,
  VALUE_WIDTH => BINDINGS_WIDTH,
  --[... more operator specific  generics  ...]
)
port map(
  [...]
  left_read       => opXinput1.read_data,
  left_data       => opXinput1.data,
  left_valid      => opXinput1.valid,
  left_finished   => opXinput1.finished,

  right_read      => opXinput2.read_data,
  right_data      => opXinput2.data,
  right_valid     => opXinput2.valid,
  right_finished  => opXinput2.finished,

  result_read     => opXoutput1.read_data,
  result_data     => opXoutput1.data,
  result_valid    => opXoutput1.valid,
  result_finished => opXoutput1.finished
);
```

Listing 4.3 – Connecting two operators identified by their ID's X and Y.

```
opXoutput1.read_data <= opYinput1.read_data;
opYinput1.data       <= opXoutput1.data;
opYinput1.valid      <= opXoutput1.valid;
opYinput1.finished   <= opXoutput1.finished;
```

receive no more triples and thus the whole query execution is blocked. Of course this can be avoided by an explicit and strict synchronization between host and FPGA, but causes additional delays and protocol overhead which degrades overall performance. Thus, we divide one physical stream into multiple virtual streams, each implicitly synchronized and not effecting other virtual streams. However, as all virtual streams share one physical interface consequently the bandwidth is shared as well. The available bandwidth for each virtual stream is not preassigned and thus if only one virtual stream is used to send data then this stream can utilize the full bandwidth.

As mentioned before, each QEP contains a result operator which interconnects the QEP in the RP to the QC located in the static logic. If the finished flag of the result operator is set to `'1'` then the query execution is finished. As a consequence the QC will close the result stream to the host which is detected on application level.

4.2.3 Parametrization of Operators

In the following sections, we present the parametrization of the operators previously introduced in Chapter 3 which are supported by our hybrid query engine. Besides common parameters such as number and data width of the variables, each operator is equipped with individual generics. While traversing the QEP in LUPOSDATE, the values to be assigned to the generic are calculated dynamically according to the current query.

4.2.3.1 RDF3XIndexScan

The RDF3XIndexScan is the link between the QC and the inner operators. Typically the RDF3XIndexScan provides data triples s, p, o but a bindings array can have less or more than three variables and also not all of the three triple components might be necessary to evaluate the query. Thus, the main objective of this operator is to receive triples from the QC and map their required components to a position in the bindings array. Therefore, the RDF3XIndexScan has three additional generics SUBJECT_POSITION, PREDICATE_POSITION and OBJECT_POSITION – each is a one-hot-coded bit vector indicating the position of the subject, predicate and object, respectively. Each vector consists of as much bits as there are variables in the bindings array. During synthesis these vectors are evaluated following a simple scheme: If bit x is set to `'1'` in the bit vector of one triple component then this triple component is connected to position x in the bindings array. Unbound variables are initialized with an *invalid* value.

4.2.3.2 Join

This operator joins the intermediate results of two preceding operators depending on one or more common join attributes. Similar to the RDF3XIndexScan, the position of the join attribute is determined by the one-hot-coded bit vector JOIN_VECTOR. As the structure of the bindings array is globally the same in the whole QEP, only one set bit is necessary. However, it is possible that a join on more than one common variable is executed. Although this is not yet supported by the proposed system it is possible to simply add additional bit vectors for secondary, tertiary, etc. orders. The bit vector is independent of the actual used join algorithm, e.g., Nested Loop Join or Merge Join.

4.2.3.3 Filter

In Section 3.4 we presented two approaches to implement the filter operator for SPARQL queries. Taking the optimizer of LUPOSDATE into account we are able to break down complex filter expressions into multiple simple filter operators of the scheme *VALUE COND VALUE*, with *COND* as the condition, e.g., equality, and *VALUE* either a constant or variable. Specifically, this means that conjunctions of filter expressions result in a chain of simple filter operators each evaluating only one relational condition involving one variable and a constant value respectively another variable. In case of disjunction the operator duplication takes place and thus multiple disjunctive conditions are evaluated by simple filter operators in concurrent branches of the operator graph. In turn the intermediate results of two (or more) branches simply need to be unified in a lower level of the QEP as described in Section 3.4.3. As a result each simple filter operator is equipped with the following generics. The generic FILTER_OP_TYPE describes the relational operation to be evaluated by the filter. Due to the mapping from strings to integer IDs our approach supports only *equal* and *unequal* comparators at the moment. However, if the dictionary (ID→string) would be available on the FPGA also other conditions such as *greater/smaller than* are possible. Furthermore, we have to distinguish between expressions comparing a variable with a constant value and comparing two variables of the bindings array. Therefore, the boolean FILTER_LEFT_IS_CONST is set to *true* if the left value is a constant. If so then the constant value is passed through the generic FILTER_LEFT_CONST_VALUE by setting the actual value to be compared. Contrary if the left value is not a constant then the one-hot-coded bit vector FILTER_LEFT_VAR_POS is considered. Like in previously described operators a set `'1'` bit in this vector corresponds to the position of the variable in the bindings array. The same scheme is applied for the right value of the filter expression as well by simply replacing the term LEFT with RIGHT in the generics.

4.2.3.4 Projection

The Projection carries out the SELECT clause of the SPARQL query. Therefore, it is equipped with the bit vector PROJECTION_VECTOR. It has as much bits as the bindings array has variables. If bit x is set to 1 bit in the bit vector then the corresponding variable at position x in the bindings array remains in the result. Otherwise the corresponding variable is discarded.

4.2.3.5 (Merge-) Union

The Union operator builds the union of the results provided by its two predecessors. It is not necessary that the same variables are bound at this point in the query execution. The Merge Union requires sorted input regarding common variables. It unifies the two inputs such that the result is still sorted. Therefore, it is equipped with the one-hot-coded bit vector UNION_VECTOR, to indicate the common variable such that the order can be preserved. Similar to the generics of the join additional bit vectors might be added to enable secondary, tertiary, etc. orders. The simple Union has no additional generics.

4.2.3.6 Limit and Offset

The Limit operator is typically located directly before the result operator and forwards a specific amount of resulting bindings arrays. After its limit is reached (by simply counting) or its preceding operator indicates to be finished, it rises its finished flag which propagates through the result operator to the QC. As a consequence the QC will close the result stream to the host which is then detected on application level. In turn, the Offset operator skips a specific number of the first resulting bindings arrays and simply passes the remaining bindings array to its successor. Combining Limit and Offset selects different subsets of the query result. Therefore, both operators are equipped with an integer generic LIMIT respectively OFFSET to set its corresponding value.

4.2.3.7 Unsupported Operators

As described in Section 3.5.6 some of the operators are not supported by our hybrid query engine, yet. If an unsupported operator is detected during query optimization the proposed system always falls back to the software-only execution covering full SPARQL 1.1. Additionally, as a next step the operator graph could be partly

located on the FPGA and on the host system. Latter executes the remaining not implemented operators.

4.3 Evaluation

This section describes the evaluation setup and analyzes the proposed architecture with respect to the query execution time and the resulting speedup compared to the software-based execution on a general-purpose CPU.

4.3.1 Evaluation Setup

The host system is a Dell Precision T3610 (Intel Xeon E5-1607 v2 3.0 GHz, 40 GByte DDR3-RAM) which is equipped with the FPGA board described in Section 2.2.2. As operation system Ubuntu 14.04 x86-64 (kernel 3.16.0-71-generic) is used. The evaluation data is located on the HDD of the host system and will be transferred to the FPGA during query execution via PCIe. The PCIe implementation is realized using the freely available *Xillybus* core [203]. Utilizing 4 PCIe lanes our architecture achieves throughputs of up to 430 MByte/s for writing data to the FPGA, and up to 225 MByte/s for reading data from the FPGA. The complete performance analysis of the PCIe interface is given in Appendix A.

However, we show in the next sections that the PCIe interface, although not even closely utilized at its specification, is not the bottleneck of our architecture. Before the performance analysis we have verified the soundness and completeness of the query results of the test queries which are introduced in the next sections.

4.3.2 SP²Bench SPARQL Performance Benchmark

As a first step in order to evaluate the presented approach systematically we use the *SP²Bench SPARQL Performance Benchmark* (SP²B) [204]. Besides example queries, it provides a data generator which is able to generate datasets with different triple cardinalities. The generated data itself is motivated by the project *Digital Bibliography & Library* (DBLP)[205] and thus is supposed to mirror key characteristics of real-world data.

For the following runtime analysis we use datasets with varying cardinalities starting at one million up to 262 million triple. Most of the SP²B queries use SPARQL features which are not fully supported by our hybrid query engine. These are the sorting and distinct operations as well as relational filter expressions requiring the

materialized string representation rather than the integer ID. Others are consisting only of a simple index scan and therefore are unlikely to benefit from our hybrid approach. However, we choose five SPARQL queries inspired by the SP^{2}B queries to show the feasibility of our approach. SP^{2}B-Q1 consists of one join and a simple filter expression. SP^{2}B-Q2 consists of two joins which can be executed independently. Both intermediate results are unified. SP^{2}B-Q3 consists of three joins. Two of them can be executed independently as well, while the third join combines the intermediate results of the previous two operators. The last join operates in a pipelined fashion concurrently to the other two joins. SP^{2}B-Q4 introduces an additional variable and thus another join, but the size of the result set is the same as for SP^{2}B-Q3. SP^{2}B-Q5 is a further extension with one variable/join more and a smaller result size. The complete test queries can be found in Appendix B. The size of the results depending on the input dataset size is shown in Figure 4.4 for each test query.

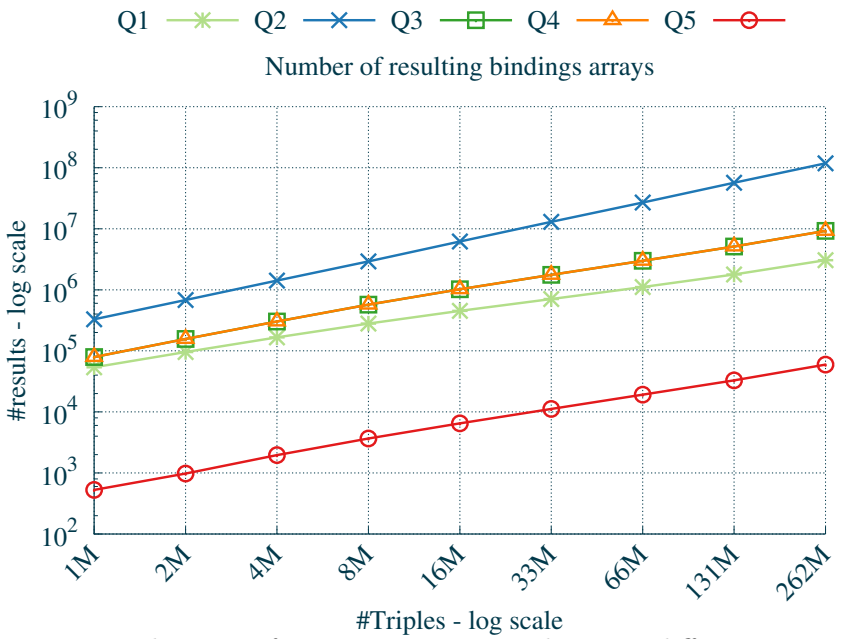


Figure 4.4 – Result size of test queries according to different sizes of the *SP^{2}Bench SPARQL Performance Benchmark* (SP^{2}B) dataset [206].

Table 4.2 gives an overview of the metrics used in the following performance evaluation. The execution time of the standard LUPOSDATE software system T_{CPU} is typically shown with a red line and each circle represents the average of 1,000

Table 4.2 – Performance metrics used in the evaluation of the software and hybrid system.

Label	Description
T_{CPU}	Query execution time using the standard LUPOSDATE software system running on a general-purpose CPU. No FPGA is involved.
$T_{FPGA\text{-}full}$	Query execution time of hybrid system including full iteration through the result on host.
$T_{FPGA\text{-}post}$	Query execution time of hybrid system including post processing of the result on host.
SP_{full}	Speedup T_{CPU} / $T_{FPGA\text{-}full}$
SP_{post}	Speedup T_{CPU} / $T_{FPGA\text{-}post}$

single executions with warm caches. Latter is achieved by a single execution of the identical query just before the actual performance evaluation. The execution times of the hybrid system are labeled with T_{FPGA} but covers the whole processing time of the hybrid system. This *includes* the communication costs to transfer triples from the host's HDD to the FPGA as well as to send back the bindings arrays forming the final result. Furthermore, we differentiate between $T_{FPGA\text{-}full}$ and $T_{FPGA\text{-}post}$. $T_{FPGA\text{-}full}$ includes the time of the whole evaluation on the hybrid system and a final iteration through the obtained result. $T_{FPGA\text{-}post}$ includes the previously described post processing steps on the host (see Section 4.1.2). Each of the both execution times are set in relation to the software-based execution times on the CPU resulting in the achieved speedup SP_{full} respectively SP_{post}.

Figure 4.5 to Figure 4.9 show the execution times of the test queries for different dataset sizes. Regarding the very simple query SP^{2}B-Q1, the software-only and the hybrid approach scale linearly to the dataset (see Figure 4.5). $T_{FPGA\text{-}full}$ grows slower on the hybrid system which results in an increasing speedup of up to 21X. However, post processing the result on the host causes a higher overhead with increasing result size shrinking the achievable speedup to 5X. As in both execution times ($T_{FPGA\text{-}full}$ and $T_{FPGA\text{-}post}$) the communication costs are included, the software turns out to be the bottleneck.

SP^{2}B-Q2 has the largest result set of all test queries. The speedup of the hybrid system is slightly increasing up to 5X with a growing dataset size (see Figure 4.6). Due to the enormous result size the post processing on the host further shrinks the speedup. However, our hybrid query engine is never slower than the CPU-based execution. Besides the enormous result size, the union operator causes the speedup degradation. Although this operator is very simple it tends to consume one intermediate result of one preceding operator and stalls the other preceding operator. In fact, it stalls a whole branch including a join in this particular query. SP^{2}B-Q3 contains three joins and has significantly less results than SP^{2}B-Q2, but more than SP^{2}B-Q1. Due to the previously described higher amount of concurrent

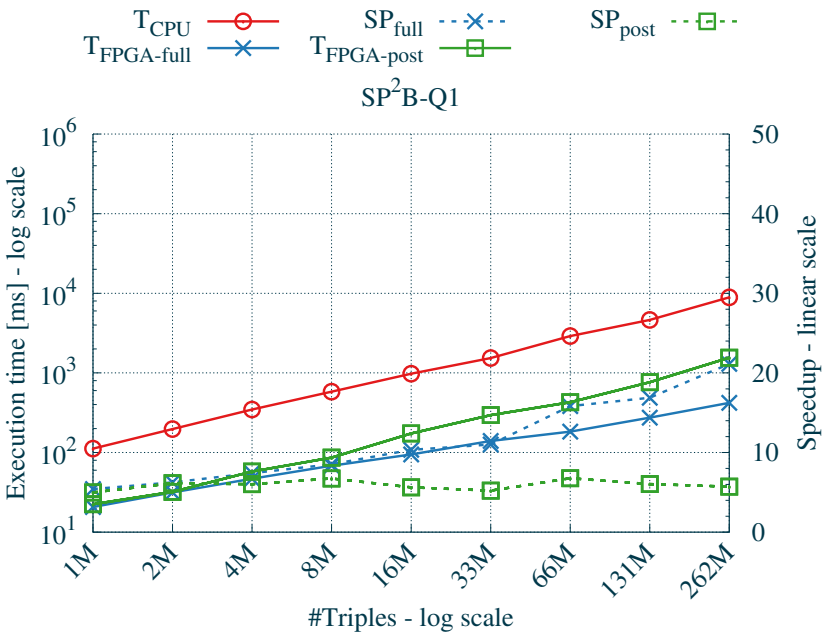


Figure 4.5 – Execution times of SP^{2}B-Q1 for different dataset sizes.

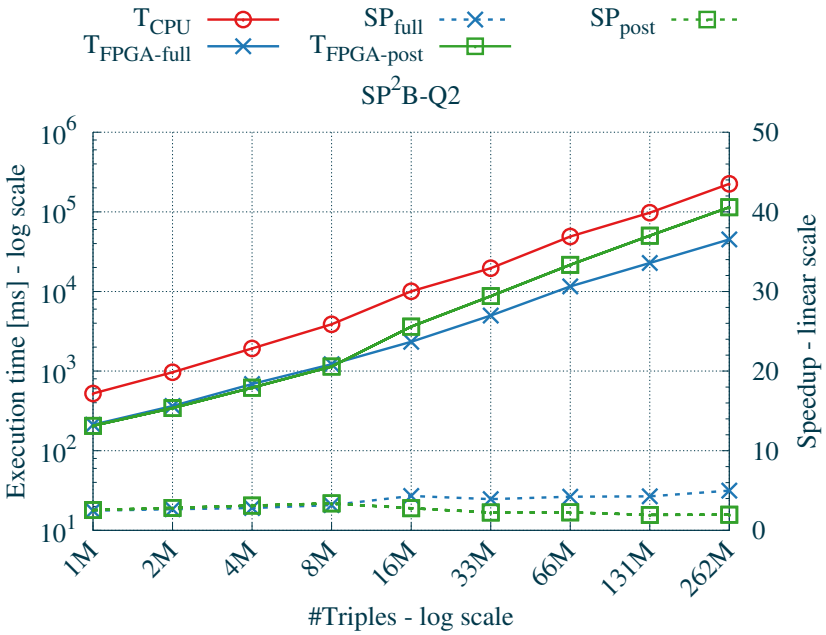


Figure 4.6 – Execution times of SP^{2}B-Q2 for different dataset sizes.

operators the hybrid system is able to show steady speedup improvements of up to 28X (see Figure 4.7). Again, post processing on the host shrinks the achievable speedup significantly down to a still significant speedup of 5X.

SP^{2}B-Q4 is an extension of SP^{2}B-Q3 by introducing a new variable and triple pattern resulting in another join. The number of resulting bindings array remains the same, but notice that regarding SP^{2}B-Q4 each bindings array contains one more variable causing a 25% higher bandwidth requirement and post processing overhead. Due to the additional join, the software-only execution needs more time to evaluate the query (see Figure 4.8). Contrary, the FPGA-accelerated execution shows almost no performance drop because the additional join is located in another branch of the QEP and is perfectly integrated into the operator pipeline. As a result the speedup increases up to 32X. However, at some point the speedup drops slightly but increases again. Contrary, the host system is not able to counterbalance this drop and post processing shrinks the speedup down to 5X.

SP^{2}B-Q5 further extends SP^{2}B-Q4 by another join respectively variable. This results in a significantly smaller results than the other queries. In Figure 4.9, it can be nicely seen that also this query suffers a speedup drop at 66 million triple but afterwards stabilizes and increases. As the result is relatively small the post processing on the host does not have a negative effect on the execution time.

It is worth to stress again the fact that the reported execution times and achieved speedups include the complete communication costs between host and FPGA which covers (i) reading triples at host side from HDD, (ii) sending triples to the FPGA using PCIe and (iii) on the other hand sending back the result from FPGA to the host and (iv) iterating through the result on the host.

4.3.3 Billion Triples Challenge

In this section, we evaluate our hybrid query engine using the read-world dataset from the *Billion Triples Challenge* (BTC) [202]. It contains crawled triple data from various sources, such as *DBpedia* and *freebase*, resulting in more than one billion distinct triples[1]. The dataset does not provide any reference queries. Therefore, we chose ten queries with different complexities regarding amount of operators, join distribution and result size. The test queries BTC-Q1 to BTC-Q10 can be found in Appendix B.

Figure 4.10 shows the resulting execution times for software-based execution ($\mathsf{T}_{\mathsf{CPU}}$) and the hybrid query engine ($\mathsf{T}_{\mathsf{FPGA\text{-}full}}$, $\mathsf{T}_{\mathsf{FPGA\text{-}post}}$) as well as the corresponding speedups

[1] 1,056,184,909 without duplicates

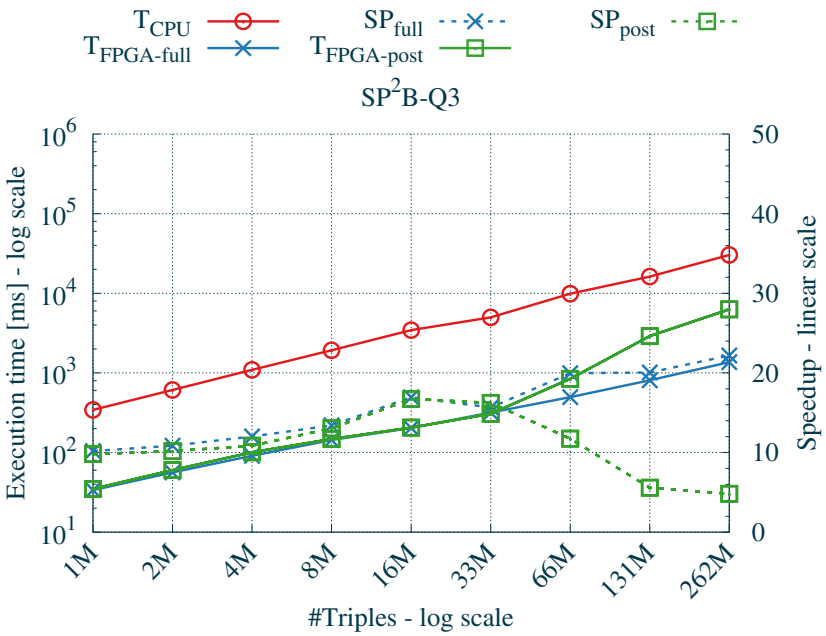


Figure 4.7 – Execution times of SP^{2}B-Q3 for different dataset sizes.

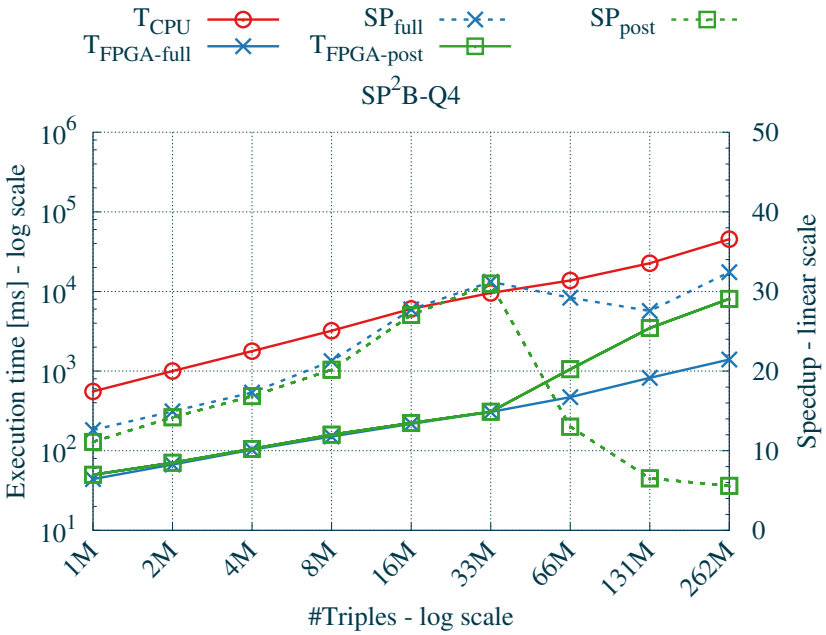


Figure 4.8 – Execution times of SP^{2}B-Q4 for different dataset sizes.

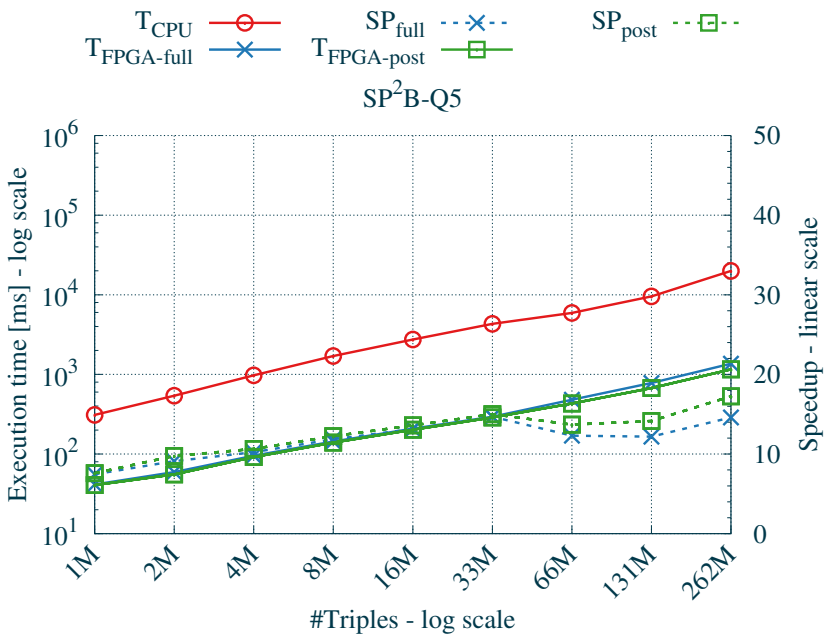


Figure 4.9 – Execution times of SP^{2}B-Q5 for different dataset sizes.

($\mathsf{SP}_{\mathsf{full}}$, $\mathsf{SP}_{\mathsf{post}}$). Again, all reported execution times include the communication costs between host and FPGA.

Although, BTC-Q1 is a very simple query, consisting of only one join, the hybrid system outperforms the software-only approach. BTC-Q2 is an extension of BTC-Q1 with an additional triple pattern and join, but the new pattern matches only a small number of triples. Thus, one join causes only low workload and the query execution does not benefit from the FPGA. However, the performance loss is almost not notable. In turn, BTC-Q3 has the same query structure as BTC-Q2 but the triple patterns match more triples which results in a higher utilization of the joins which has no effect on the hybrid system but on the software-only approach due to higher workload. BTC-Q4 and BTC-Q5 further extend BTC-Q3 by one respectively two additional triple patterns (with a small amount of matches) resulting in one respectively two additional joins. The latter causes higher workload on the software-only approach but has no impact on the hybrid system. BTC-Q6 has the same structure as BTC-Q5 but the distribution of triples between index scans is more homogenous. Furthermore, the involved joins generate less intermediate results which results in a significantly lower execution time than in the previous queries. BTC-Q7 extends BTC-Q6 by another triple pattern and join resulting in a ten times smaller result which is beneficial for the software-only approach but is still significantly slower than the hybrid system. BTC-Q8

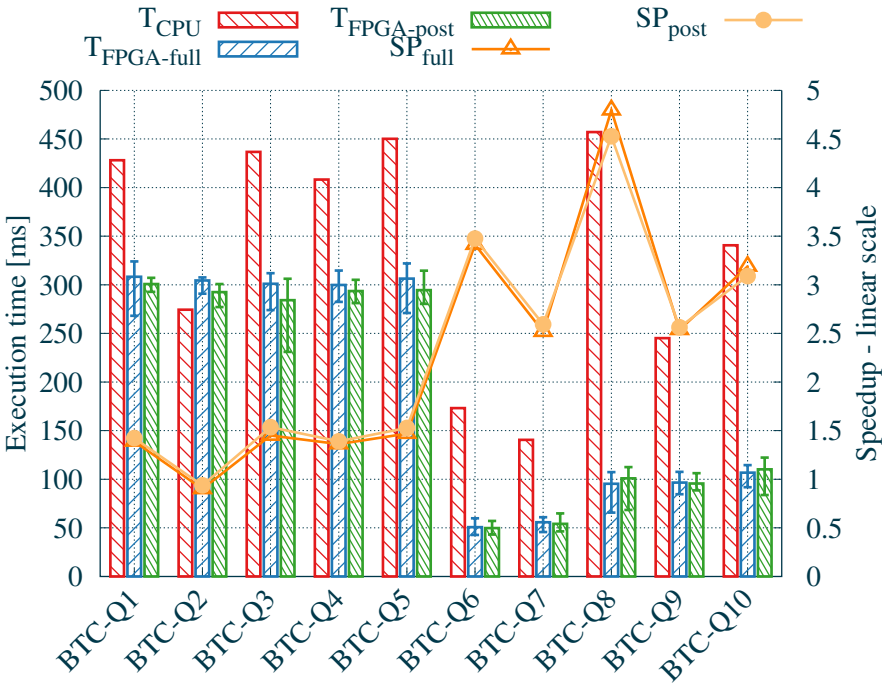


Figure 4.10 – Execution times of test queries on the BTC-2012 dataset [202]. Queries can be found in Appendix B.

results in a perfectly balanced operator graph consisting of four index scans and three joins which enables the hybrid system to make use of its inherent advantages. BTC-Q9 adds another triple pattern without any impact on the hybrid system but performance gain of software-only system. BTC-Q10 extends BTC-Q9 by adding two triple patterns resulting in two additional joins. Again, while the execution time of the hybrid system slightly changes, the performance of the software-only approach degrades due to the additional operators. In summary, it can be seen that the hybrid system is less sensible to the query structure but also the post processing on the host system is negligible.

4.4 Related Work

IBM's Netezza Data Appliance Architecture [207] uses FPGAs to reduce the amount of data to be transferred from (persistent) memory to the CPU by early execution of projection and restriction. It consists of several distributed computing nodes (*S-Blades*). Each node is equipped with multi-core CPUs, gigabytes of RAM and

FPGAs. So-called *FAST* engines are running in parallel inside the FPGAs and uncompress and filter data coming from the physical hard drives. Uncompressing data at wire speed on the FPGA increases the read throughput and thus reduces the drawback of hard disks. Filtering the data before reaching the CPU reduces the load on the CPU and thus increases performances. The FPGA is used as a co-processor to support the CPU but neither allows a modular composition of database operators nor a complete query execution on the FPGA.

Mueller *et al.* [208, 209, 210] propose the component library and compositional compiler *Glacier*. It takes a continuous query for data streams and generates a corresponding file written in the hardware description language VHDL. Due to the window-based processing on data streams, the approach is not suitable for large-scale datasets. Furthermore, no join operations are supported. After the time-consuming translation of the VHDL description to an FPGA configuration, the query can be configured on an FPGA in order to accelerate the evaluation on data streams. Consequently, this approach is not ideal for online processing or only suitable for a known query set. Note that the approach of this chapter suffers the same reconfiguration overhead. This remaining issue is addressed in Chapter 5.

Another synthesis framework that generates complex event processing engines for FPGAs from an SQL-based language is proposed by Takenaka *et al.* [211]. In fact, SQL primitives are translated into C source code which in turn is transformed into a circuit description using C-to-HDL compiler. The prototype allows window-based selection, matching and aggregation on multiple concurrent streams for a financial analysis application. Joins are not supported.

4.5 Summary

In this chapter we integrated a new query evaluator into LUPOSDATE which dynamically generates an FPGA-based accelerator for a given query. Therefore, the FPGA is divided into static logic and a large RP. The configuration file (RM) of the RP is automatically assembled by using a query template and a pool of operators. With respect to the query the operators are connected with each other and parametrized by operator specific generics. As all operators provide a common interface the presented framework can be easily extended by new operator implementations. In order to show the architecture's feasibility and benefits, we executed several queries on large-scale synthetic datasets generated by the SP^{2}B data generator and real-world datasets from the BTC. It was shown that our hybrid query engine is able to achieve speedups of up to 32X including all communication costs

between host and FPGA. In some cases our approach is not significantly faster than the CPU-based execution but always shows comparable performance.

However, one important aspect is not yet addressed in the performance evaluation. Due to complexity of the PAR process, which maps the query to actual resources on the FPGA, the RM generation takes between 20 to 30 minutes. Thus, although fully functional the system is not efficiently applicable in a real-world scenario with dynamically changing queries. A straightforward approach is the generation of RMs of frequently issued queries and reuse them during runtime. In fact, the detection and reuse of known queries is prototypically implemented in the proposed hybrid system. However, in a highly dynamic environment this approach is not applicable as any arbitrary query is possible. Therefore, we propose to divide the chip area into multiple small RPs rather than one large RP. Each RP is then able to take one (but arbitrary) operator. This remaining issue is addressed in the next chapter by introducing *Semi-static Operator Graphs* (SOGs).

5 Semi-static Operator Graphs

In the previous chapter, we have shown how to dynamically generate VHDL code to represent and execute a given query on an FPGA. It was shown that the query execution can highly benefit from our hybrid architecture. However, due to the fact that the whole FPGA toolchain was executed during system runtime the proposed architecture is not applicable in real-world scenarios with arbitrary queries. Thus, in this chapter we extend our architecture by *Semi-static Operator Graphs* (SOGs). SOGs deploy the structure of a general operator graph whereas each operator in the graph is a *Reconfigurable Partition* (RP). Each RP can take any arbitrary operator but only one operator at a time. Furthermore, the interconnections of preceding and succeeding operators is not finally set and can be manipulated by reconfiguring different *Reconfigurable Modules* (RMs) into the RPs. As a result the time consuming generation of configurations can be moved into the deployment time of the system but still maintains a certain degree of flexibility regarding different query structures. An article containing parts and results of this chapter has been submitted for publication at the current *Special Issue of the MICPRO Journal* and is under review.

5.1 Extending the Hybrid Query Engine

The overall architecture remains the same as in the previous chapter. The host provides higher functions (such as user interface, query optimization, maintaining of evaluation indices, etc.), and the FPGA is utilized as a runtime adaptive accelerator for query execution. The communication between both units is based on PCIe. In the previous chapter, the FPGA chip area was divided into static logic and one large RP. The RP provided logic resources for a whole operator graph with an arbitrary number of operators, only limited by the chip area. However, the configuration for the RP was generated during query time. In this chapter, *Dynamic Partial Reconfiguration* (DPR) is used at operator level rather than at

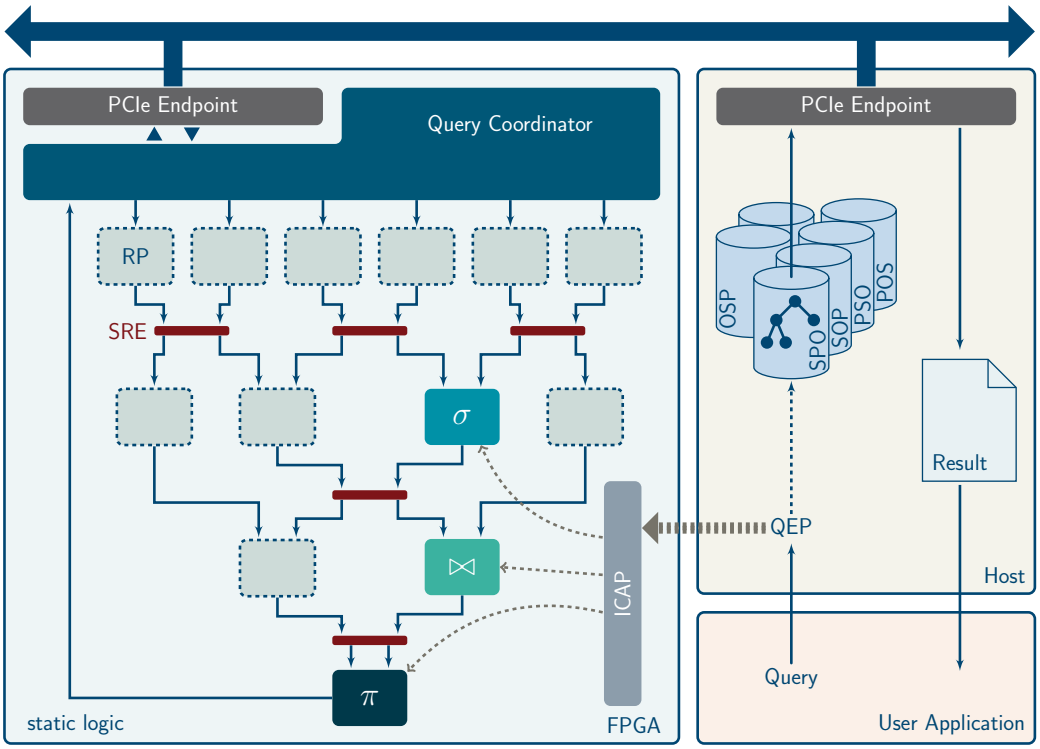


Figure 5.1 – Architectural overview of the integrated hybrid query execution engine using *Semi-static Operator Graph* (SOG). The *Query Execution Plan* (QEP) on the FPGA is assemble by choosing appropriate RMs and transferring them into *Reconfigurable Partition* (RP) via the *Internal Configuration Access Port* (ICAP).

QEP level. Figure 5.1 shows the modified hybrid architecture. Rather than providing one large RP, the FPGA area is divided into static logic and multiple small RPs, each taking any kind of operator. The static logic contains modules which are independent of the actual query structure. Typically, those are modules for communication and memory interfaces. In this case it contains the PCIe EP and the *Query Coordinator* (QC). Additionally, the interconnections between the operators are located in the static logic as well. But instead of directly connecting the output of one operator to the input of another operator, we introduce *Semi-static Routing Elements* (SREs) between operator levels. In Section 5.1.1 we explain how these SREs allow a flexible operator interconnection in the deployed operator graph. Utilizing DPR, the RMs of the required operators are transferred into the static logic via the PCIe interface and are configured into the RPs using the ICAP.

5.1.1 Semi-static Routing Element

Instead of connecting the output of one operator directly to the input of another operator, we instantiate SREs between them. Each SREs can take a specific number of operator outputs and redirects them to the same number of operator inputs. The number of supported operators can be set during deployment time of the system. Figure 5.2 presents an example with two inputs and two outputs. Dashed squares mark the RPs. The SREs are located in the static logic. The internal behavior of a SRE in terms of input-output-relations is effected by the external input succ_sel. In Figure 5.2, if succ_sel is set to `'0'` then the results of the preceding operators are routed straight downwards. Otherwise succ_sel is set to `'1'` and the results are routed crosswise. The actual value of succ_sel is set from within the RP and thus can be modified during system runtime by configuring different RMs in the corresponding RPs. SREs can be instantiated using more than two preceding and succeeding operators. Obviously, then the succ_sel inputs of the other operators must be considered, e.g., using three inputs/outputs implies six possible input-output relations which requires three bits for encoding. Internally, the SREs are simple *n-to-n* multiplexer (or *crossbars*). However it is not required that all inputs and outputs are connected to actual RPs and can be unconnected.

5.1.2 Modified Hybrid Work Flow

The fundamental change within the FPGA design using multiple small RPs instead of one large RP effects the general hybrid work flow as shown in Figure 5.3. In order to utilize the RPs in the FPGA fabric efficiently, it is divided into *deployment time* and *system runtime*.

5.1.2.1 Deployment time

Such as in traditional database systems, our hybrid system needs to be deployed before using it for query processing. Alongside the obligatory installation of the mandatory software and initial import of data, the static logic has to be deployed on the FPGA and the RMs need to be generated and stored in the repository. The semi-static structure of the QEP in the FPGA needs to be expressed explicitly in VHDL by the designer or one of the developed SOGs of this work can be used. Note that the static design contains no actual operator but RPs instead. After the SOG is chosen, several processing steps of the FPGA toolchain have to be applied. Therefore, we provide scripts using Python [212] and Tcl [102] to automatically execute the following tasks during deployment time:

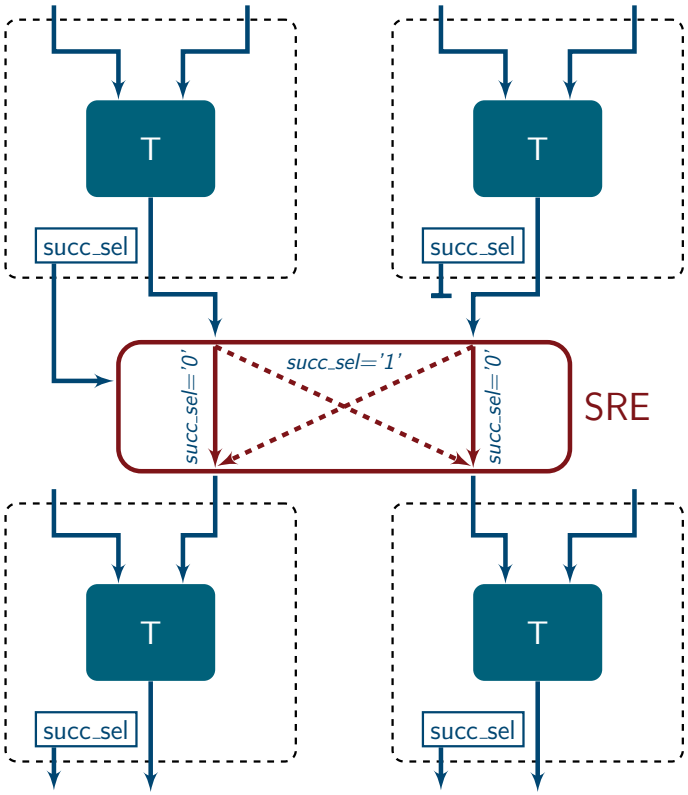


Figure 5.2 – *Semi-static Routing Elements* (SREs) are located in the static logic connecting multiple Reconfigurable Partitions (RPs, dashed squares). By transferring a *Reconfigurable Module* (RM) into the RP, the value of succ_sel is set which determines the final connection between two succeeding operators.

a) Synthesis of static logic including the SOG. At this stage an upper bound determining the number of supported variables must be given. The netlist is imported into a PlanAhead project [93].
b) Obtaining netlists for all operators. In fact, for each operator multiple netlists are generated with all possible permutations of their corresponding generics and possible values for succ_sel. Regarding Figure 5.2, there are at least two netlists for each operator type. One with succ_sel set to '0' and one with succ_sel set to '1'. As a consequence many netlists are generated. However, an alternative approach would have been to set these metadata during query execution by sending them just before the data stream. This would require additional operator logic to distinguish between metadata and actual triple data. Therefore, we decided to keep operators as simple as possible by the cost of additional deployment time.

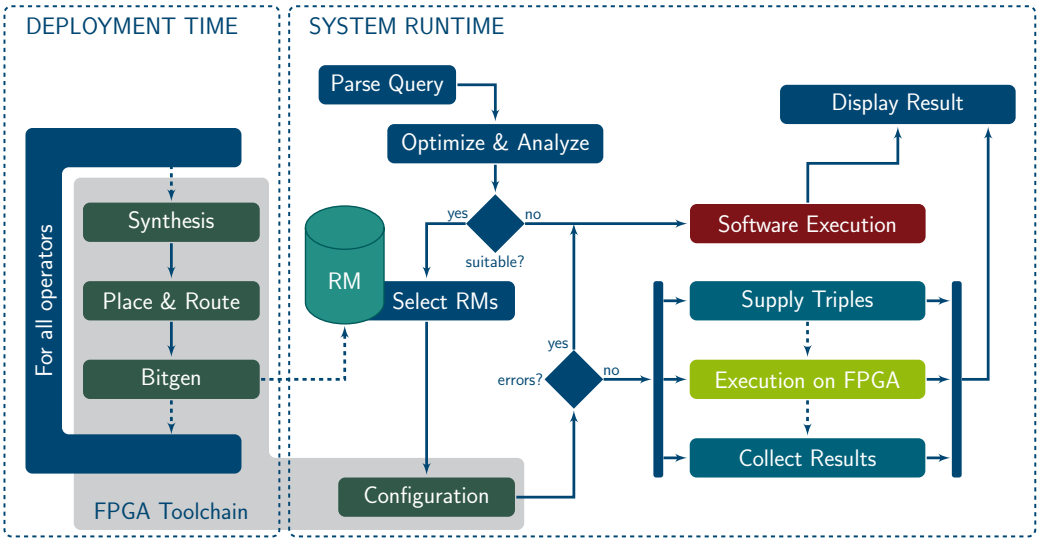


Figure 5.3 – Flow chart of the hybrid system using SOGs. The *Reconfigurable Modules* (RMs) are generated at system deployment time. During system runtime, the RMs are selected and configured into the RPs.

c) For each operator netlist one *run* is generated in the PlanAhead project importing the common static logic. Each run assigns its particular netlist to all RPs. An exception is the RDF3XIndexScan as it is always located in the *upper* levels of the SOG.
d) All generated runs are executed until BitGen. This generates the RMs of all operators for all RPs.
e) Finally, the static logic is configured onto the FPGA.

Due to many possible combinations of RPs, operator types, operator generics and values for succ_sel potentially many RMs are generated. As explained in Section 2.2.3 this process is very time consuming. However, with this approach we execute the time consuming FPGA toolchain during deployment time of the system and not during system runtime.

5.1.2.2 System Runtime

After the hybrid system has been deployed, the user submits a query which is transformed into an operator graph by the LUPOSDATE system. On the operator graph several logical and physical optimizations are applied (see Section 2.1.3). Afterwards the optimized operator graph is analyzed for unsupported operators. If all operators are supported then our new extension traverses the operator graph

and chooses appropriate RMs in terms of operator type and input for the SREs. The chosen RMs are configured via ICAP, setting the final structure of the operator graph on the FPGA.

Algorithm 5 shows the mapping algorithm for a given QEP and SOG. In a first processing step all possible tree structures are generated by encountering all possible assignments of succ_sel of all SREs. The SOG used in the subsequent evaluation consists of five SREs each with two inputs respectively outputs. Thus, the mapping algorithm generates 32 trees which can be handled in reasonable time. Due to the operator interface each operator in the QEP respectively each RP in the SOG can have up to two predecessors. If no predecessors is available then it is set to *null*. For each possible tree the MAP function is applied on the root of the QEP respectively the generated tree. The MAP function recursively assigns operators from the QEP to RPs in the tree. If a RP is not existent but an attempt to assign an operator is made or the RP exists but no operator is assigned (line 11) then *false* will be returned as this is not a valid assignment. If both are *null* – meaning that the previously assigned operator was an index scan mapped into the last RP of a path – then the mapping is valid (see line 14). Otherwise, the operator and the RP exist and the operator is assigned to the RP (line 17). Afterwards, the MAP function is applied on the left predecessor of the current operator respectively RP and vice versa on the right predecessors (line 18). If these mapping attempts in the subtrees are successful then a mapping for the tree rooted at the current operator is found. If one mapping attempt in the subtrees fails then a *pass-through* operator is assigned to the current RP (line 21). This is motivated by the fact that the branch in the SOG can be deeper than the branch in the QEP. Thus pass-through operators simply fill up the operator pipeline. If a pass-through operator was assigned, another attempt to assign the current operator to one of the current RP's predecessor is made (lines 22 and 25). If these attempts fail then no mapping of the current subtree can be found.

In case the QEP consists of unsupported operators or can not be mapped into the SOG, then the query is evaluated completely by the software engine on the host system. As described in Section 4.1.2, during the actual query execution host and FPGA work jointly. While the query engine on the FPGA determines the query results, the host system continuously supplies triples to the FPGA and retrieves results from the FPGA via PCIe.

Algorithm 5 SOG-MAPPING (Returns a mapping of a given QEP into a SOG)

```
1: procedure SOG-MAPPING(QEP, SOG)
2:  T ← Enumeration of all possible trees by permutating all SREs in SOG
3:  for all t in T do
4:   if MAP(QEP.root, t.root) then
5:    return mapping found
6:   end if
7:  end for
8:  return no mapping found
9: end procedure
10: function MAP(Operator op, ReconfigurablePartition rp) : boolean
11:  if (rp = null and not (op = null)) or (not(rp = null) and op = null) then
12:   return false
13:  end if
14:  if (rp = null and op = null) then
15:   return true
16:  end if
17:  rp.assign(op)
18:  if MAP(op.left, rp.left) and MAP(op.right, rp.right) then
19:   return true
20:  else
21:   rp.assign(pass-through)
22:   if MAP(op, rp.left) then
23:    return true
24:   end if
25:   if MAP(op, rp.right) then
26:    return true
27:   end if
28:  end if
29:  return false
30: end function
```

5.2 Evaluation

In this section we describe the evaluation setup and analyze the newly introduced concept of SOGs with respect to the reconfiguration time. Furthermore, we set the results in relation to the query execution times of the approach presented in Chapter 4.

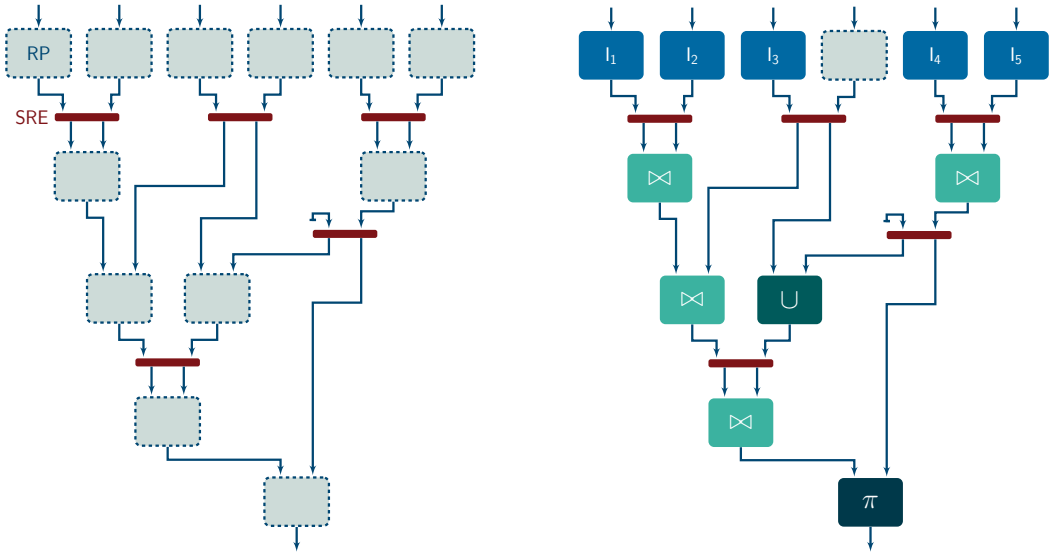
5.2.1 Evaluation Setup

The host system is a Dell Precision T3610 (Intel Xeon E5-1607 v2 3.0 GHz, 40 GByte DDR3-RAM) which is equipped with the FPGA board described in Section 2.2.2. As operation system Ubuntu 14.04 x64 (kernel 3.16.0-71-generic) is used. The ICAP module operates at 100 MHz and configuration data width is 32 bit. Thus, theoretical maximum configuration speed is 3.2 GBit/s (381 MByte/s). The compressed[1] size of the RMs varies from 199 KByte till 375 KByte. Thus, when supplying the RMs at full speed to the ICAP then the reconfiguration of one operator theoretically requires between 0.5 and 1 ms. However, the RMs are located at the host system and must be read from hard drive and transferred via PCIe into the static logic. Thus, the total reconfiguration overhead might still impact the performance of the hybrid query engine. Besides the methodology on exchanging QEPs, the evaluation setup remains the same as in Section 4.3.

5.2.2 Benchmarks

In this evaluation, we use the five SPARQL queries on the SP^{2}B dataset as introduced in Section 4.3.2. Each query is executed on three different datasets (66M, 131M and 262M triples) generated with the SP^{2}B data generator [206]. In order to support all five queries, the SOG shown in Figure 5.4(a) is described in VHDL and deployed on the FPGA. It consists of 12 RPs and supports queries with up to 8 variables. The generated Xillybus core provides eight virtual streams, of which 6 are used to supply the 6 index scan operators in the upper level. One stream is exclusively used to transfer the RMs to the ICAP module. One stream remains unused. Figure 5.4(b) shows the configuration of SP^{2}B-Q4 as an example. Note that the QEPs of some queries are smaller in height than the deployed SOG. Therefore, pass-through operator need to be configured into empty partitions to connect the configured smaller QEP with the last RP which is connected to the QC. We simply use the Union operator with only one predecessor to pass-through results.

[1] BitGen was used with flag *-g Compress* [103, p. 227]



(a) SOG without any configured *Reconfigurable Module* (RM).

(b) SP²B-Q4 configured into the SOG. Union acts as pass-through operator.

Figure 5.4 – Structure of the *Semi-static Operator Graph* (SOG) used in the evaluation.

Table 5.1 gives an overview of the utilized resources. Most of the resources in the static logic are consumed by the Xillybus core and corresponding buffers. Due to the heterogeneous FPGA architecture, the deployed RPs are not homogenous in shape and size but provide enough resources to take any of the operators. But, if necessary, we provide methods for efficiently identifying homogenous RPs [213]. In total all 12 RPs together cover less than half of the chip area.

Table 5.2 presents the measured reconfiguration times of the test queries and relates them to the query execution times. The query execution time using the standard LUPOSDATE system is labeled as T_{CPU}. The query execution time of the hybrid system T_{FPGA} denotes the pure execution time without reconfiguration overhead as reported in Section 4.3.2.

Table 5.1 – Resource utilization of deployed *Semi-static Operator Graph* (SOG) consisting of 12 *Reconfigurable Partitions* (RPs).

	Registers	LUTs	BRAM (36 KBit)
total available	478,080	239,040	768
static logic	3,518	4,858	18
RP size (each)	12,800 – 16,640	6,400 – 8,320	16 – 32

Table 5.2 – Execution times and reconfiguration overhead of five SPARQL queries on three different SP^{2}B datasets [206]. The reconfiguration time during system runtime is significantly reduced and slightly impacts the overall achievable speedup.

Query		Query execution			Reconfiguration		Total	
#Operators (+pass-through)	Triples [$\cdot 10^6$]	T_{CPU} [ms]	T_{FPGA} [ms]	SP w/o reconf.	nSOG [min:sec]	SOG [ms]	SOG [ms]	SP w/ reconf.
SP^{2}B-Q1	66	2,900	183	15.8			191.2	15.2
5 (+1)	131	4,623	273	16.9	21:43	8.2	281.2	16.4
	262	8,888	420	21.1			428.2	20.8
SP^{2}B-Q2	66	48,983	11,306	4.3			11,320.2	4.3
8 (+2)	131	97,465	22,297	4.4	23:39	14.2	22,311.2	4.4
	262	225,068	45,102	5.0			45,116.2	5.0
SP^{2}B-Q3	66	9,897	495	20.0			509.0	19.4
8 (+2)	131	16,194	807	20.1	23:22	14.0	821.0	19.7
	262	30,311	1,366	22.2			1,380.0	22.0
SP^{2}B-Q4	66	13,756	470	29.3			485.7	28.3
10 (+1)	131	22,666	822	27.6	25:08	15.7	837.7	27.1
	262	45,304	1,397	32.4			1,412.7	32.1
SP^{2}B-Q5	66	5,913	480	12.3			497.1	11.9
12	131	9,524	781	12.2	30:29	17.1	798.1	11.9
	262	19,920	1,364	14.6			1,381.1	14.4

Note that using SOGs neither impacts the overall performance of the query operators nor increases the query execution time on the FPGA. Even if additional pass-through operators have to be inserted to match the SOG (indicated by (+X) in the first column in Table 5.2) the total throughput remains the same. Only an initial delay of two clock cycles might be caused by each pass-through operator in the operator pipeline. The evaluated architecture operates at 200 MHz and thus the delay of two times 5 ns per pass-through operator is negligible. Furthermore, it is worth to stress again the fact that these execution times include the whole communication between host and FPGA. The reconfiguration overhead of the previous approach without using SOG is denoted with nSOG. As the whole FPGA toolchain was applied on the generated VHDL code, obtaining a configuration took between 21 minutes for SP^{2}B-Q1 and 30 minutes for SP^{2}B-Q5. Although the FPGA-based query execution was significantly accelerated, the overhead for obtaining a query-specific configuration prevented an effective use of reconfiguration capabilities in real-world scenarios. By utilizing SOGs most of the reconfiguration overhead is moved to the deployment time of the system. Consequently, the plain reconfiguration times of the RMs assembling the test queries on the FPGA consume only 8.2 ms for SP^{2}B-Q1 up to 17.1 ms for SP^{2}B-Q5 instead of minutes (see column Reconfiguration in Table 5.2). In average the reconfiguration of one operator takes

1.4 ms which is significantly higher than our previous estimations. However, due to the asynchronous PCIe interface these numbers do not reflect the internal reconfiguration performance of the ICAP module. Furthermore, loading the RMs from the HDD is the major bottleneck. If the RMs are already located in the host's main memory then copying the reconfiguration data into the DMA memory is at least one order of magnitude faster. Overall this slightly impacts the total speedup (see columns SP w/o reconfig and SP w/ reconfig in Table 5.2). If the queries are executed on smaller datasets the relative overhead introduced by the reconfiguration is logically higher due to shorter query execution times. The most significant speedup reduction from 29.3X to 28.3X is noted for SP^{2}B-Q4 at a dataset size of 66 million triples. Regarding queries with high execution times such as SP^{2}B-Q2 the reconfiguration overhead is negligible. In summary all achieved speedups are still significant and prove the feasibility of our approach. The generation of all RMs for the shown SOG takes around 17 hours on the mentioned workstation but significantly reduces reconfiguration overhead during query time. The queries of the BTC benchmark are not evaluated explicitly as the plain query execution time is unaffected by this approach. The reconfiguration overhead of a single operator is query independent and thus does not deliver new insights.

5.3 Related Work

Dennl *et al.* [214, 215] present FPGA-based concepts for on-the-fly hardware acceleration of SQL queries in the relational database MySQL. Assuming a small data bus width, the data is split into several, properly sequenced parts (chunks). For handling of intermediate results, spare chunks are allocated in the data stream and each module is aware of where to insert its results. In order to build a pipeline of modules, the authors use a post-order traversal through the operator tree to preserve the dependencies of the operators. The chunks and the linearization of operators save chip area, but also decrease the overall performance. The authors mentioned the necessity of join processing, but they focused only on projection and restriction such as in Netezza [207]. Thus, complete queries cannot be executed on the FPGA. However, in order to evaluate more complex queries (including joins) additional views are created to represent partial results. The proposed hardware-software system consists of four RP and achieves promising speedup gains for in-memory tables. Becher *et al.* [216] extend this approach to an embedded low-energy system-on-chip platform and add more complex operators (e.g., merge join and sorting of small datasets). In [217] the authors introduce a hash-based join and slightly extend the query with some additional restrictions. For simple

queries including one join they achieve a comparable performance but higher energy efficiency than a standard x86-based system.

Polig *et al.* [218] use FPGAs for text analytics. The proposed extension of IBM's SystemT allows to partition a given text-analytics query into a supergraph executed in software and an subgraph that is compiled into a hardware netlist. The evaluation shows promising speedup of up to 16X for small documents with a size of 2 KByte. However, DPR is not utilized and the generation of the FPGA configuration is time consuming, and thus only applicable for known queries.

Teubner *et al.* [219, 220] present an FPGA-based XML filter. The XML document is streamed into the FPGA via a network interface. The remaining XML expressions are transmitted to a CPU-based host which applies an XQuery. Due to the reduced amount of (not needed) data, less data needs to be parsed on the host, and the processing and main-memory overheads are reduced as well. In order to cover a wide range of different queries, the authors propose to statically compile a *skeleton automaton* that can be configured at runtime to implement query-dependent state automata. Each skeleton segment consists of a single state and two parameterized transition conditions. These conditions are set using an additional configuration stream prior to the query execution and thus no DPR is used for reconfiguration.

Sadoghi and Najafi [221, 222, 223, 224] describe a re-programmable event stream processor based on an FPGA. The architecture consists of *online programmable blocks* (OPB) [225], which each supports a number of basic query operators (selection, projection, window-joins). Each OPB can be dynamically configured using a simple instruction set. Thus, no DPR is used. The authors present some performance characteristics but provide no direct comparison to an existing software solution. However, the chainable OPBs provide a promising basic framework for stream processing applications.

In summary, the hereby presented hybrid architecture utilizing SOGs enables to provide query-specific hardware accelerators covering a wide range of queries with an appropriate number of operators including join operations.

5.4 Summary

In this chapter, we introduced the concepts of *Semi-static Operator Graphs* (SOGs) on FPGAs to provide a flexible hardware accelerator for query execution in the context of SW databases. Instead of generating one FPGA configuration for a given query during system runtime, we deploy a general query structure with a certain degree of flexibility. A SOG consists of multiple RPs with a common interface.

During the deployment of the system for each RP a set of RMs in terms of partial bitfiles is generated and stored into a repository. At system runtime, our hybrid system chooses RMs regarding a given SPARQL query and configures them via ICAP into the RP which sets the final structure of the operator graph on the FPGA. As a result the time consuming generation of RMs is executed before system runtime and significantly reduces the reconfiguration during query execution. The presented hybrid query engine significantly outperforms the software-based execution on the CPU taking communication and reconfiguration costs into account.

6 Conclusion

In this work we investigate the use of hardware accelerators in the dynamic context of query evaluation in *Semantic Web* (SW) databases. Due to the variety of queries a static approach deploying a predefined query processor (with a fixed set of processing primitives covering the different query operators) is not feasible. Therefore, in this work reconfigurable hardware in terms of a *Field-Programmable Gate Array* (FPGA) is used as a co-processor assisting the SW database LUPOSDATE. While the software system covers tasks which are either not performance critical or too complex and not suitable for hardware acceleration, we introduce a new hybrid query engine which utilizes a reconfigurable FPGA to accelerate query evaluation.

As a first step to an extensible and flexible architecture, we defined the operator template. Each implemented query operator must use the minimalistic common interface enabling the transparent collaboration without making any assumptions or providing detailed knowledge about preceding and succeeding operators. The building block concept enables the simple composition of multiple operators to match a given query structure and provides a pervasive degree of parallelism. On the example of the join and filter operator multiple FPGA-based approaches have been presented and evaluated. As a result both operators are able to provide at least a competitive performance compared to general-purpose CPUs. In most cases, the FPGA prototype achieves a significant speedup. Regarding the filter operator, it was shown how different types of parallelism can be deployed within one operator (pipelining, intra-operator parallelism) but must be evaluated in terms of scalability and resource consumption.

With a set of operators on the FPGA, the LUPOSDATE system was extended by a new query evaluator utilizing the capabilities of the FPGA. Therefore, the FPGA is divided into static logic and a large *Reconfigurable Partition* (RP). The configuration file of the RP is automatically assembled by using a VHDL query template and the set of implemented operators. Each operator can be parametrized

by operator specific generics. After the automatic transformation into a configuration, the FPGA is reconfigured and the query result is collaboratively determined. The performance gains of the proposed hybrid architecture are shown by executing multiple queries on large-scale synthetic datasets as well as real-world data from the *Billion Triples Challenge* (BTC). Our hybrid query engine achieves speedups of up to 32X including all communication costs between host and FPGA. Although our approach is not significantly faster in all test cases, it is never slower than the CPU-based execution.

However, due to the fact that the whole FPGA toolchain is executed during system runtime the proposed architecture is not applicable in real-world scenarios with arbitrary queries. Consequently, we introduced the concept of *Semi-static Operator Graphs* (SOGs). A SOG describes the structure of a general operator graph whereas each operator in the graph is replaced by a black box in terms of a RP. Each RP can take any arbitrary operator but only one operator at a time. The interconnections between the operators are predefined but allow a certain degree of flexibility. The concrete sequence of operators can be effected by configuring different *Reconfigurable Modules* (RMs) into the RPs. As a result the time consuming generation of configurations can be moved into the deployment time of the system. Instead of applying the whole FPGA toolchain during query time, only the prepared RMs need to be chosen and reconfigured. This greatly reduces the reconfiguration overhead of QEP from minutes to milliseconds but still maintains a certain degree of flexibility regarding different query structures.

In summary, this work contributes a hybrid query engine on the basis of an FPGA which is transparently integrated into the SW database LUPOSDATE. It is able to automatically execute and accelerate the query evaluation. However, as each new architecture, our work has potential to be improved and extended. In the following section we will outline possible extensions.

Future Work – Opportunities and Limitations

In this section we outline some further ideas based on this work and provide research directions to be investigated in the future.

Operators and Sideways Information Passing

In the current prototype of the hybrid query engine not all operators are supported. In case of unsupported operators the query evaluation is completely executed on the CPU-based host system. Besides implementing the missing operators on the

FPGA, in some cases it is reasonable to break down the strict separation. Instead of mapping the whole QEP onto the FPGA, only some supported operators could be processed by the FPGA and the remaining operators would be executed in the software system. For instance, the Distinct and Order-By operator are always located at the end of the QEP and thus can be executed on the host while retrieving the results from the FPGA. However, it is still recommended to develop the Distinct operator for the FPGA-based query execution because it typically reduces the size of the result to be transferred to and processed by the host.

Additionally, all operators on the FPGA do not support *Sideways Information Passing* (SIP) at the moment. As described in Section 2.1.3, SIP enables an operator to prune irrelevant intermediate results of its direct or indirect preceding operator and therefore saves unnecessary processing steps. Logically this is done by passing information across operators. For instance, a merge join operator can hand over the value of the join attribute of one input as a lower bound to retrieve only intermediate result of the other input which are equal or larger than this bound as smaller values will not be joined in any case.

In order to implement this feature in our hybrid query engine, the operator template has to be extended. Basically, for each input/output relation an additional signal to pass the lower bound and a signal to indicate the activation of SIP must be implemented. Furthermore, as in the current approach the indices are located at the host system, the lower bound must be sent to the host system via PCIe to beneficially use SIP in the index scan operation.

Index Scan and Maintenance

In the current approach the LUPOSDATE system generates and manages the indices. During query execution the indices are scanned for triples matching the triple patterns which are transferred to the FPGA. The FPGA board used in this work is equipped with two SATA-II ports and thus allows to directly attach HDDs/SSDs to store the indices. Consequently, the communication overhead and the load on the host will be significantly reduced because only the result of a query needs to be sent to and processed by the host system after deploying the query. Furthermore, it can simplify the support of the previously described SIP feature.

The evaluation indices could be either constructed on the host system and then transferred onto the FPGA's HDDs, or an FPGA design is deployed which is capable to generate the evaluation indices with potential performance benefits. Regarding access operations first attempts are presented by Heinrich *et al.* [65]. Thus,

FPGA resources could be used in a time-multiplexed fashion for index generation and query evaluation.

Relocation in Semi-static Operator Graphs

Introducing SOGs greatly reduces the overall reconfiguration time of complete QEPs on the FPGA. In this work, we generate separate RMs for each RP and each operator type. The generation of all RMs for the SOG used in this work takes around 17 hours. Although this time is spent during deployment time and does not effect the system runtime, it is desirable to reduce this time. This is further motivated by the fact that the RMs of one operator for different RPs actually base on the same netlist. Thus, as their description is identical, it is desirable to use the same partial bitfile of one RM in different RPs without applying the whole FPGA toolchain again and again. Using the same RM in different RPs is referred as *relocation*. One requirement is a uniform structure of the RPs in terms of provided resources (slices, BRAM, DSP). But the increasing heterogeneity and complexity of modern FPGAs complicate the manual identification of equally sized and structured RPs. Backasch *et al.* [213] propose a method for efficiently identifying homogenous RPs. It allows to define a pattern which represents minimal resource requirements provided by each RP. The pattern is dynamically adjusted in order to meet the structural requirements of the heterogeneous device. As a result it automatically returns a placement consisting of the maximum number of possible RPs. After identification and deployment of identical RPs, the RM can be modified in such a way that it can be configured into different RPs. The relocation can be done without modifying the actual configuration data but updating the address and CRC[1] field of a configuration data frame corresponding to the FPGA's configuration memory. Ichinomiya *et al.* [226] present methods of relocation at runtime during reconfiguration of the device. Furthermore, in order to adjust operator specific properties (such as the position of join variables) it is necessary to store the operator's parameters in registers within the operator and modify their content by manipulating the configuration data according to the query.

Besides optimizing the RM generation process, deploying new SOG structures can be simplified. Currently, if a new SOG needs to be deployed then a hardware designer has to describe the structure in an HDL. However, this description contains only instantiations of SREs and declarations of black boxes for the operators as well as interconnections between those modules. Although this is relatively simple it requires knowledge about HDLs. Thus, a GUI-based tool could assist the construction of SOGs by providing abstract modules which can be assembled in a

[1] Cyclic Redundancy Code

building block fashion. Internally, the tool could generate the corresponding HDL code.

Query Scheduler and Optimizer

In the current prototype of the hybrid query engine, the FPGA characteristics are not taken into account by LUPOSDATE's query optimizer. However, it was shown that not all queries remarkably benefit from the hardware-accelerated execution and thus it is reasonable to develop an estimator which is able to predict the expected performance gains. These can depend on the amount of data and the complexity of the query. Additionally, with respect to the SOG, the optimizer might avoid reconfiguration of some operators by reusing already configured operators to further reduce reconfiguration time. Some of the evaluated queries share common parts in the query structure and thus the reconfiguration of all operators is not necessary. Therefore, our query optimizer must be extended to detect these common structures.

Multi-query Support

In this work the FPGA is used to accelerate one query at a time. However, typically a database system receives many queries within a short period. Thus, it is desirable to accelerate multiple queries on the FPGA (intra-query parallelism). In fact, the modular approach and using virtual streams between host and FPGA allows an easy extension. Besides executing queries concurrently, methods for detecting common expressions have been proposed known as *Multi-query optimization* [222, 227, 228]. Multi-query optimization aims to interleave common structure of different QEPs in order to avoid redundant calculations and share results. It is reasonable to further investigate these methods with respect to FPGA-based query execution in order to save logic resources but also to improve performance.

Support of Other Query and Rule Languages

In this work we support the query language SPARQL. However, the presented concepts can be extended to support other query languages such as SQL for relational databases. Furthermore, in Section 2.1.2 we introduced the *Rule Interchange Format* (RIF) which enables to express inference rules and complex conditions. Comparable to the approach of Baker *et al.* [180] (see Section 2.3.3) a cyclic pipeline or graph consisting of rule operators could be established to evaluate the rules and generate explicit facts which are expressed implicitly.

Alternative Architectures

Currently, our approach deploys one RP to take a complete operator graph respectively multiple RPs each taking one operator. In the latter approach the interconnections are slightly adjustable but still restricted and do not allow arbitrary communications between any operator. Instead of direct connections or bus systems so-called *Network on Chips* (NoCs) can be deployed for on-chip communication. In a NoC each processing module is equipped with a router. The routers are arranged in a grid which allows direct communication with neighboring routers but also indirect communication with any other router in the NoC. Backasch *et al.* [229] present a generic hardware design based on a NoC which allows the composition of application specific data paths at system runtime. Each RP has an assigned router. This approach enables a higher degree of flexibility in placing one or more operator graphs onto an FPGA. However, as the NoC introduces delays in the data propagation, which might reduce query performance, it is still desirable that succeeding operators in the operator graph are placed beside each other in the NoC. Besides a flexible operator placement, a NoC-based architecture would simplify the integration and utilization of other components such as dedicated memory modules. Regarding the hereby proposed query engine, the dictionary which implements the mapping between strings and IDs could be easily integrated into the architecture and accessed from all operators. This would significantly enrich the support of hardware-accelerated query execution.

Harizopoulos *et al.* [230, 231, 232] introduce a *staged* database design for multi-core CPUs. They present the implementation of a *QPipe* which consists of independent *micro-engines*. Each relational operator is promoted to one micro-engine but each micro-engine might execute its particular operation for several queries on the same, overlapping or different datasets (*one-operator, many-queries*). To each micro-engine one or more threads are assigned. As a result locality in both data and computation across different queries can be efficiently exploited. Regarding a multi-FPGA setup this methodology could be applied such that each FPGA provides the computational resources for one micro-engine. Due to the different complexity of the operators either smaller FPGAs can be used for simple micro-engines or multiple simple micro-engines are consolidated on one FPGA. Furthermore, using DPR additional micro-engines can be deployed depending on the current load of already deployed micro-engines.

Final Thoughts

Another major advantage of FPGAs, besides inherent parallelism, is the significantly lower energy consumption. Fowers *et al.* [233] compare the performance of sliding-window applications executed on FPGAs, GPUs and multicore-CPUs, and report orders of magnitude lower energy consumption for FPGA-based solutions while achieving significant speedups. Thus, the use of FPGAs in database systems can not only improve overall performance but also reduce the energy footprints of data centers – highly beneficial from an economical as well as an ecological perspective.

A

Performance of PCIe

The PCIe interface is realized using the freely available Xillybus core [203]. The FPGA application interacts with the Xillybus core using a standard FIFO interface. The host application performs plain IO operations on file descriptors corresponding to interfaces within the FPGA design. Although logically the communication is stream-based, internally *Direct Memory Access* (DMA) buffers are allocated in the host's memory space [234]. The handshake protocol handles writing and reading request by filling and handing over these DMA buffers. Throughout this work *asynchronous* streams are used due to their better performance for continuous data flows. Regarding the FPGA-to-host communication (upstream), the host's DMA buffers are filled whenever possible, e.g. stream is open, data is available and DMA buffers are not full. If the host application wants to read the data then the data is copied from the DMA buffer into a buffer provided by the application.

Regarding host-to-FPGA communication (downstream), the application's writing call copies the present data into the DMA buffers and returns immediately in case all the data can be stored in the DMA buffers. The actual transfer of the buffers from the host to the FPGA is mostly triggered by two conditions. Either the DMA buffer is completely filled or a timeout expires. Currently the timeout is set to 5 ms but is adjustable during core generation [235]. If a buffer is not filled then the actual data transfer will be initiated after the timeout which results in a lower total throughput due to higher overhead compared to actual payload. Hence, with a high amount of data the internal buffers are utilized intensively and the data transactions are more efficient.

The used FPGA board is equipped with an 8-lane PCIe GEN-2 interface. The theoretical possible bandwidth of this interface is noted with 4 GByte/s. However, for the used *Xilinx* Virtex-6 a data rate of 400 MByte/s using only 4 PCIe lanes is reported [236]. Furthermore, the developers of Xillybus report a reduction to 200 MByte/s (due to overhead of the data link layer and TLP packet headers). Additionally, they point out that often the processing of the data on application

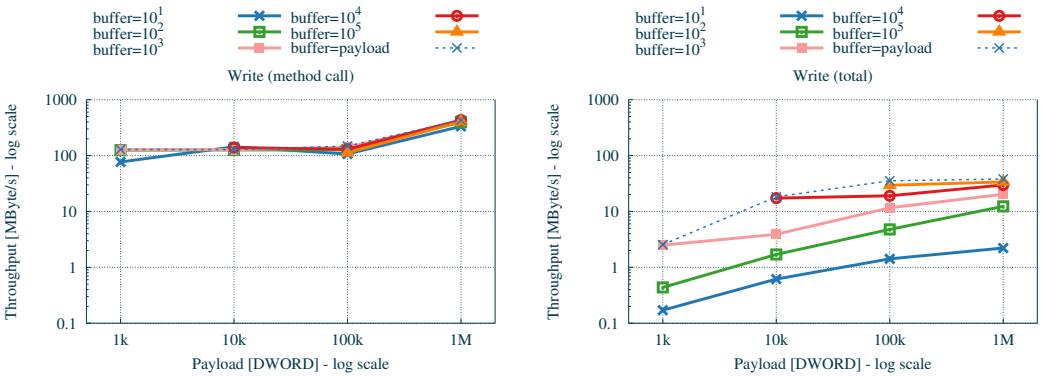
level turns out to be the real bottleneck [237]. Therefore, we evaluated the Xillybus core in our environment with different buffer sizes at application level. In this evaluation one downstream and one upstream is used. However, during core generation the number and characteristics of the streams can be adjusted to the applications need [235]. In our setting each stream utilizes 16 DMA buffers with each 128 KByte (2 MByte total). The interface width is 32 bit. The host system is a Dell Precision T3610 (Intel Xeon E5-1607 v2 3.0 GHz, 40 GByte DDR3-RAM) which is equipped with the FPGA board described in Section 2.2.2. As operation system Ubuntu 14.04 x86-64 (kernel 3.16.0-71-generic) is used.

A.1 Downstream Throughput

As mentioned before, a writing call from the application to transfer data to the FPGA simply copies the given data into the host's DMA buffers. Thus, a return of the method call does not imply that the data was already transferred into the FPGA design. In order to evaluate the downstream throughput we transfer datasets with varying sizes of 1,000 up to 1 million DWORDs (32 bit). In the FPGA design the data is simply discarded. Furthermore, we vary the buffer size at application level (not DMA buffers) from 10 up to 100,000 DWORDs and additionally a buffer which perfectly fits the length of the data payload. Figure A.1(a) shows the performance of the writing method call. No significant differences are achieved by the different buffer sizes. In turn, the total amount of data heavily impacts the throughput. With increasing size the throughput increases from 80 MByte/s up to 430 MByte/s. However, this scenario considers only the plain method call to the DMA engine. Figure A.1(b) takes into account that the buffer at application level has to be filled by the application. Therefore, we simply iterate through the buffer once, fill it with incrementing values and then call the writing method. As it can be seen in Figure A.1(b) this significantly slows down the throughput. This supports the statement of the Xillybus developers [237]. Due to additional overhead using multiple small buffers the usage of larger buffers enhances performance.

A.2 Upstream Throughput

As mentioned before, a reading call from the application simply copies data from the DMA buffers into the provided application buffer. For this evaluation data is continuously generated and send by the FPGA. Again, datasets with varying sizes of 1,000 up to 1 million DWORDs (32 bit) are transferred and read. Figure A.2(a) shows the performance of the reading method call. Note that the method does not

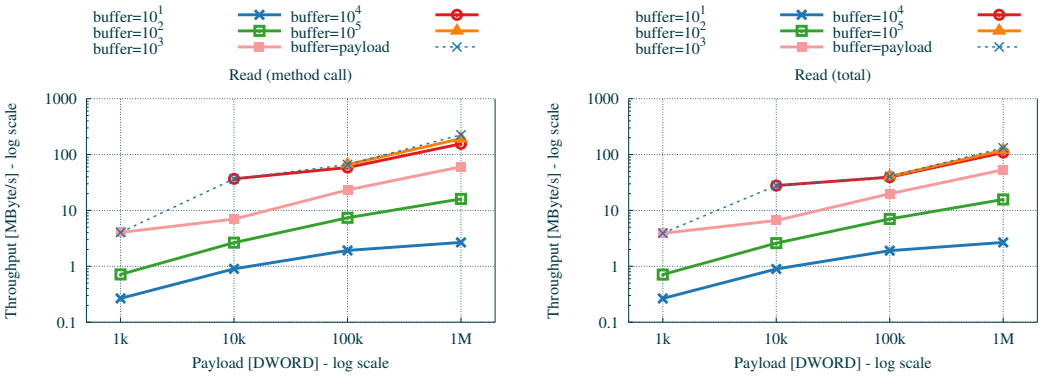


(a) Plain method call until return.

(b) Iteration through input buffer and method call until return.

Figure A.1 – Throughput of host-to-FPGA (downstream) communication.

return before the application buffer is filled in this scenario. In total the upstream throughput is significantly worse with 0.3 MByte/s up to 230 MByte/s compared to the previously reported downstream. It can be seen that the size of the application buffer has a high impact until a certain buffer size (10^4 DWORDs). Figure A.2(b) provides the results of calling the read method and subsequent iteration through the result. As the throughput of the plain read method is already slow the overhead on the application level is noticeable but not as serious as for the downstream.



(a) Plain method call until return.

(b) Method call until return and iteration through output buffer.

Figure A.2 – Throughput of FPGA-to-host (upstream) communication.

B Test Queries

B.1 Commonly used prefixes

The following prefix definitions are used in the test queries:

Prefix definitions

```
PREFIX bench: <http://localhost/vocabulary/bench/>
PREFIX dbo: <http://dbpedia.org/ontology/>
PREFIX dbp: <http://dbpedia.org/property/>
PREFIX dc: <http://purl.org/dc/elements/1.1/>
PREFIX dcterms: <http://purl.org/dc/terms/>
PREFIX foaf: <http://xmlns.com/foaf/0.1/>
PREFIX rdf: <http://www.w3.org/1999/02/22-rdf-syntax-ns#>
PREFIX rdfs: <http://www.w3.org/2000/01/rdf-schema#>
PREFIX swrc: <http://swrc.ontoware.org/ontology#>
```

B.2 Queries on SP²B dataset

The following test queries are used in Section 4.3.2:

SP²B-Q1

```
#Get all articles with property swrc:pages.
SELECT ?article WHERE {
  ?article rdf:type bench:Article .
  ?article ?property ?value .
  FILTER (?property = swrc:pages) .
}
```

SP²B-Q2 (equal to query *q9* of SP²B [206])

```
#Get incoming and outcoming properties of persons.
SELECT ?predicate WHERE {
  { ?person  rdf:type     foaf:Person  .
    ?subject  ?predicate  ?person
  } UNION {
    ?person  rdf:type     foaf:Person  .
    ?person  ?predicate  ?object
  }
}
```

SP²B-Q3

```
#Get all articles  with title,  number of pages and creator.
SELECT ?article ?title ?pages ?creator WHERE {
  ?article  rdf:type     bench:Article  .
  ?article  dc:title     ?title  .
  ?article  swrc:pages ?pages .
  ?article  dc:creator  ?creator  .
}
```

SP²B-Q4

```
#Get all articles  with titles,  number of pages, the creator  and the journal.
SELECT ?article ?title ?pages ?creator  ?journal  WHERE {
  ?article  rdf:type       bench:Article  .
  ?article  dc:title       ?title  .
  ?article  swrc:pages     ?pages .
  ?article  dc:creator     ?creator  .
  ?article  swrc:journal  ?journal  .
}
```

SP²B-Q5

```
#Get all articles  with titles,  number of pages, the creator,  journal  and month when published.
SELECT ?article ?title ?pages ?creator  ?journal  ?month WHERE {
  ?article  rdf:type       bench:Article  .
  ?article  dc:title       ?title  .
  ?article  swrc:pages     ?pages .
  ?article  dc:creator     ?creator  .
  ?article  swrc:journal  ?journal  .
  ?article  swrc:month     ?month .
}
```

B.3 Queries on BTC-2012 dataset

The following test queries are used in Section 4.3.3:

BTC-Q1

```
SELECT * WHERE {
  ?book1 dbp:author ?author .
  ?book1 dbp:name   ?title .
}
```

BTC-Q2

```
SELECT * WHERE {
  ?book1 dbp:author  ?author .
  ?book1 dbp:name    ?title .
  ?book1 dbp:pubDate ?date .
}
```

BTC-Q3

```
SELECT * WHERE {
  ?book1 dbp:author  ?author .
  ?book1 dbp:name    ?title .
  ?book1 dbp:country ?country .
}
```

BTC-Q4

```
SELECT * WHERE {
  ?book1 dbp:author  ?author .
  ?book1 dbp:name    ?title .
  ?book1 dbp:country ?country .
  ?book1 dbp:pages   ?pages .
}
```

BTC-Q5

```
SELECT * WHERE {
  ?book1 dbp:author  ?author .
  ?book1 dbp:name    ?title  .
  ?book1 dbp:country ?country .
  ?book1 dbp:pages   ?pages .
  ?book1 rdf:type    dbo:Book .
}
```

BTC-Q6

```
SELECT * WHERE {
  ?book1 dbp:author  ?author .
  ?book1 dbo:isbn    ?isbn  .
  ?book1 dbp:country ?country .
  ?book1 dbp:pages   ?pages .
  ?book1 rdf:type    dbo:Book .
}
```

BTC-Q7

```
SELECT * WHERE {
  ?book1 dbp:author  ?author .
  ?book1 dbo:isbn    ?isbn  .
  ?book1 dbp:country ?country .
  ?book1 dbp:pages   ?pages .
  ?book1 rdf:type    dbo:Book .
  ?book1 dbp:pubDate ?date .
}
```

BTC-Q8

```
SELECT * WHERE {
  ?s dbp:countryofbirth  ?o1 .
  ?s dbo:birthDate       ?o3 .
  ?s rdf:type            dbo:Athlete .
  ?s dbp:fullname        ?o5 .
}
```

BTC-Q9

```
SELECT * WHERE {
  ?s dbp:countryofbirth  ?o1 .
  ?s dbp:countryofdeath ?o2 .
  ?s dbo:birthDate       ?o3 .
  ?s dbo:deathDate       ?o4 .
  ?s rdf:type            dbo:Athlete .
}
```

BTC-Q10

```
SELECT * WHERE {
  ?s rdf:type            dbo:Athlete .
  ?s dbp:countryofbirth  ?o1 .
  ?s dbp:countryofdeath ?o2 .
  ?s dbo:birthDate       ?o3 .
  ?s dbo:deathDate       ?o4 .
  ?s dbp:fullname        ?o5 .
}
```

Acronym

AHJ	Asymmetric Hash Join
AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
AST	Abstract Syntax Tree
BitGen	Bitstream Generator
BRAM	Block RAM
BTC	Billion Triples Challenge
CAE	Computer Aided Engineering
CLB	Configurable Logic Block
CPU	Central Processing Unit
CMT	Clock Management Tile
DBMS	Database Management System
DDR	Double Data Rate
DMA	Direct Memory Access
DPR	Dynamic Partial Reconfiguration
DSP	Digital Signal Processor
EP	Endpoint
FF	Flip-Flop
FIFO	First-In First-Out queue
FPGA	Field-Programmable Gate Array
FSM	Finite State Machine
GPU	Graphics Processing Unit

GPGPU	General Purpose Computing on Graphics Processing Units
GTX	Gigabit Transceiver
HDD	Hard Disk Drive
HDL	Hardware Description Language
HJ	Hash Join
HJL	Hash Join using separate chaining with linked list
HTML	Hypertext Markup Language
IC	Integrated Circuit
ICAP	Internal Configuration Access Port
IEEE	Institute of Electrical and Electronics Engineers
IO	Input/Output
IOB	Input/Output Block
IP	Intellectual Property
IRI	Internationalized Resource Identifier
ISE	Integrated Synthesis Environment
LOD	Linking Open Data
LUPOSDATE	Logically and Physically Optimized Semantic Web Database Engine
LUT	Lookup Table
MIG	Memory Interface Generator
MJ	Merge Join
MMCM	Mixed-Mode Clock Manager
MUX	Multiplexer
N3	Notation 3
NCD	Native Circuit Description
NLJ	Nested Loop Join
NoC	Network on Chip
OSI	Open Systems Interconnect

OWL	Web Ontology Language
PAR	Place & Route
PCIe	Peripheral Component Interconnect Express
PIP	Programmable Interconnect Point
QC	Query Coordinator
QEP	Query Execution Plan
QL	Query Language
RAM	Random-Access Memory
RDF	Resource Description Framework
RDFa	Resource Description Framework in Attributes
RDFS	RDF Schema
RIF	Rule Interchange Format
RM	Reconfigurable Module
RP	Reconfigurable Partition
RTL	Register Transfer Level
SATA-II	Serial AT Attachment II
SHJ	Symmetric Hash Join
SIP	Sideways Information Passing
SOG	Semi-static Operator Graph
SP^{2}B	SP^{2}Bench SPARQL Performance Benchmark
SPARQL	SPARQL Protocol And RDF Query Language
SRAM	Static Random-Access Memory
SRE	Semi-static Routing Element
SSD	Solid State Drive
SW	Semantic Web
Tcl	Tool command language
TEMAC	Tri-Mode Ethernet MAC

TLP	Transaction Layer Packet
UART	Universal Asynchronous Receiver Transmitter
UCF	User Constraint File
UCS	Universal Character Set
URI	Universal Resource Identifier
UUT	Unit Under Test
W3C	World Wide Web Consortium
WWW	World Wide Web
VHDL	VHSIC Hardware Description Language
VHSIC	Very High Speed Integrated Circuit
XML	Extensible Markup Language
XST	Xilinx Synthesis Technology

References

[1] Robert H'obbes' Zakon. Hobbes' Internet Timeline (Version 23). `http://www.zakon.org/robert/internet/timeline/`, January 2016. Accessed: 2016-05-03.

[2] InternetLiveStats.com. Google Search Statistics. `http://www.internetlivestats.com/google-search-statistics/`. Accessed: 2016-05-12.

[3] Linked Open Data. Connect Distributed Data across the Web. `http://linkeddata.org/`. Accessed: 2016-06-08.

[4] John von Neumann. First Draft of a Report on the EDVAC. In *IEEE Annals of the History of Computing*, volume 15, 1993. Exact copy of the original typescript from 1945.

[5] Stefan Werner, Sven Groppe, Volker Linnemann, and Thilo Pionteck. Hardware-accelerated Join Processing in Large Semantic Web Databases with FPGAs. In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation (HPCS 2013)*, pages 131–138, Helsinki, Finland, July 2013. IEEE.

[6] Stefan Werner, Dennis Heinrich, Marc Stelzner, Sven Groppe, Rico Backasch, and Thilo Pionteck. Parallel and Pipelined Filter Operator for Hardware-Accelerated Operator Graphs in Semantic Web Databases. In *Proceedings of the 14th IEEE International Conference on Computer and Information Technology (CIT2014)*, pages 539–546, Xi'an, China, September 2014. IEEE.

[7] Stefan Werner, Dennis Heinrich, Marc Stelzner, Volker Linnemann, Thilo Pionteck, and Sven Groppe. Accelerated join evaluation in Semantic Web databases by using FPGAs. *Concurrency and Computation: Practice and Experience*, 28(7):2031–2051, May 2015.

[8] Stefan Werner, Dennis Heinrich, Jannik Piper, Sven Groppe, Rico Backasch, Christopher Blochwitz, and Thilo Pionteck. Automated Composition and Execution of Hardware-accelerated Operator Graphs. In *Proceedings of the 10th International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, June 2015. IEEE.

[9] Stefan Werner, Dennis Heinrich, Sven Groppe, Christopher Blochwitz, and Thilo Pionteck. Runtime Adaptive Hybrid Query Engine based on FPGAs. *Open Journal of Databases (OJDB)*, 3(1):21–41, 2016.

[10] Netcraft Ltd. April 2006 Web Server Survey. `http://news.netcraft.com/archives/2006/04/06/april_2006_web_server_survey.html`, April 2006. Accessed: 2016-05-03.

[11] Netcraft Ltd. April 2016 Web Server Survey. `http://news.netcraft.com/archives/2016/04/21/april-2016-web-server-survey.html`, April 2016. Accessed: 2016-05-03.

[12] Steve Bratt (W3C). Semantic Web, and Other W3C Technologies to Watch. `https://www.w3.org/2007/Talks/0130-sb-W3CTechSemWeb/0130-sb-W3CTechSemWeb.pdf`, January 2007. Accessed: 2016-05-03.

[13] Tim Berners-Lee, James Hendler, and Ora Lassila. The semantic web. *Scientific American*, 284(5):34–43, May 2001.

[14] Sven Groppe. *Data Management and Query Processing in Semantic Web Databases*. Springer Verlag, Heidelberg, 2011.

[15] International Organization for Standardization. ISO/IEC 10646:2014 - Universal Coded Character Set (UCS). `http://www.iso.org/iso/home/store/catalogue_ics/catalogue_detail_ics.htm?csnumber=63182`, September 2014. Accessed: 2016-05-17.

[16] M. Duerst and M. Suignard. RFC 3987 - Internationalized Resource Identifiers (IRIs). `https://tools.ietf.org/html/rfc3987`, January 2005. Accessed: 2016-05-17.

[17] World Wide Web Consortium (W3C). Extensible Markup Language (XML) 1.0 (Fifth Edition). `https://www.w3.org/TR/xml/`, November 2008. Accessed: 2016-05-17.

[18] World Wide Web Consortium (W3C). Resource Description Framework (RDF): Concepts and Abstract Syntax. `https://www.w3.org/TR/2004/REC-rdf-concepts-20040210/`, 2004. Accessed: 2016-10-01.

[19] The W3C SPARQL Working Group. SPARQL 1.1 Overview. `https://www.w3.org/TR/sparql11-overview/`, 2013. Accessed: 2016-08-15.

[20] World Wide Web Consortium (W3C). RDF Schema 1.1. `https://www.w3.org/TR/rdf-schema/`, February 2014. Accessed: 2016-08-15.

[21] W3C OWL Working Group. OWL 2 Web Ontology Language Document Overview (Second Edition). https://www.w3.org/TR/owl2-overview/, December 2012. Accessed: 2016-10-01.

[22] W3C Rule Interchange Format (RIF) Working Group. RIF Framework for Logic Dialects (Second Edition). https://www.w3.org/TR/2013/REC-rif-fld-20130205/, Februrary 2013. Accessed: 2016-08-15.

[23] World Wide Web Consortium (W3C). RDF 1.1 Concepts and Abstract Syntax. https://www.w3.org/TR/2014/REC-rdf11-concepts-20140225/, 2014. Accessed: 2016-08-15.

[24] World Wide Web Consortium (W3C). Notation3 (N3): A readable RDF syntax. https://www.w3.org/TeamSubmission/2011/SUBM-n3-20110328/, March 2011. Accessed: 2016-10-01.

[25] World Wide Web Consortium (W3C). RDF 1.1 N-Triples. https://www.w3.org/TR/2014/REC-n-triples-20140225/, February 2014. Accessed: 2016-08-15.

[26] World Wide Web Consortium (W3C). RDF 1.1 Turtle. https://www.w3.org/TR/2014/REC-turtle-20140225/, February 2014. Accessed: 2016-08-15.

[27] World Wide Web Consortium (W3C). RDF 1.1 XML Syntax. https://www.w3.org/TR/2014/REC-rdf-syntax-grammar-20140225/, February 2014. Accessed: 2016-08-15.

[28] World Wide Web Consortium (W3C). RDFa Core 1.1 - Third Edition. https://www.w3.org/TR/rdfa-core/, March 2015. Accessed: 2016-05-17.

[29] World Wide Web Consortium (W3C). HTML5 - A vocabulary and associated APIs for HTML and XHTML. https://www.w3.org/TR/html5/, October 2014. Accessed: 2016-05-17.

[30] World Wide Web Consortium (W3C). SPARQL Query Language for RDF. http://www.w3.org/TR/2008/REC-rdf-sparql-query-20080115, January 2008. Accessed: 2016-10-01.

[31] World Wide Web Consortium (W3C). SPARQL 1.1 Federated Query. https://www.w3.org/TR/sparql11-federated-query/, March 2013. Accessed: 2016-08-15.

[32] World Wide Web Consortium (W3C). SPARQL 1.1 Update. https://www.w3.org/TR/sparql11-update/, March 2013. Accessed: 2016-05-12.

[33] World Wide Web Consortium (W3C). SPARQL 1.1 Query Language. https://www.w3.org/TR/sparql11-query/, March 2013. Accessed: 2016-05-11.

[34] W3C Web Ontology Working Group. OWL Web Ontology Language Overview. https://www.w3.org/TR/2004/REC-owl-features-20040210/, February 2004. Accessed: 2016-10-01.

[35] W3C OWL Working Group. OWL 2 Web Ontology Language Profiles (Second Edition). https://www.w3.org/TR/2012/REC-owl2-profiles-20121211/, December 2012. Accessed: 2016-10-01.

[36] Markus Krötzsch. OWL 2 Profiles: An Introduction to Lightweight Ontology Languages. In Thomas Eiter and Thomas Krennwallner, editors, *Proceedings of the 8th Reasoning Web Summer School, Vienna, Austria, September 3–8 2012*, volume 7487 of *LNCS*, pages 112–183. Springer, 2012.

[37] World Wide Web Consortium (W3C). OWL 2 Web Ontology Language New Features and Rationale (Second Edition). https://www.w3.org/TR/2012/REC-owl2-new-features-20121211/, December 2012. Accessed: 2016-05-03.

[38] World Wide Web Consortium (W3C). OWL Web Ontology Language Use Cases and Requirements. https://www.w3.org/TR/webont-req/, February 2004. Accessed: 2016-10-01.

[39] Ivan Herman (W3C). Questions (and Answers) on the Semantic Web. https://www.w3.org/People/Ivan/CorePresentations/SW_QA/Slides.html, June 2007. Accessed: 2016-05-03.

[40] Dublin Core Metadata Initiative. Expressing Dublin Core metadata using the Resource Description Framework (RDF). http://dublincore.org/documents/2008/01/14/dc-rdf/, January 2008. Accessed: 2016-05-18.

[41] Dan Brickley and Libby Miller. FOAF Vocabulary Specification 0.99. http://xmlns.com/foaf/spec/20140114.html, January 2014. Accessed: 2016-05-18.

[42] Roger Meier and Edd Dumbill. Description of a Project (DOAP) vocabulary (on Github). https://github.com/edumbill/doap. Accessed: 2016-05-18.

[43] Martin Hepp and Andreas Radinger. eClassOWL - The Web Ontology for Products and Services (v 5.1.4). http://www.heppnetz.de/projects/eclassowl/, April 2010. Accessed: 2016-05-18.

[44] Yves Raimond, Samer Abdallah, Mark Sandler, and Frederick Giasson. The Music Ontology. In *Proceedings of the International Conference on Music Information Retrieval (ISMIR)*, 2007. Accessed: 2016-05-18.

[45] Ivan Herman (W3C). Tutorial on Semantic Web. `https://www.w3.org/People/Ivan/CorePresentations/SWTutorial/Slides.pdf`, April 2012. Accessed: 2016-05-04.

[46] Vladimir Lifschitz. Closed-world databases and circumscription. *Artif. Intell.*, 27(2):229–235, November 1985.

[47] World Wide Web Consortium (W3C). Semantic Web Case Studies and Use Cases. `http://www.w3.org/2001/sw/sweo/public/UseCases/`, June 2012. Accessed: 2016-05-12.

[48] François-Paul Servant. Semantic web technologies in technical automotive documentation. In *3rd OWL: Experiences and Directions Workshop (OWLED2007)*, 2007.

[49] Yves Raimond, Tom Scott, Silver Oliver, Patrick Sinclair, and Michael Smethurst. Use of Semantic Web technologies on the BBC Web Sites. In *Linking Enterprise Data*, pages 263–283. Springer US, 2010.

[50] Helen Chen and Jos de Roo. Using Semantic Web and Proof Technologies to Reduce Errors in Radiological Procedure Orders. `https://www.w3.org/2001/sw/sweo/public/UseCases/Agfa/`, February 2007. Accessed: 2016-05-17.

[51] Jeen Broekstra, Christiaan Fluit, Arjohn Kampman, Frank Van Harmelen, Heiner Stuckenschmidt, Ravinder Bhogal, A. Scerri, Anita De Waard, and Erik M. van Mulligen. The Drug Ontology Project for Elsevier - An RDF Architecture Enabling Thesaurus-Driven Data Integration. In *Proceedings of the WWW2004 Workshop on Application Design, Development and Implementation Issues in the Semantic Web*, January 2004.

[52] Pompeu Casanovas. Helping New Judges Answer Complex Legal Questions. `https://www.w3.org/2001/sw/sweo/public/UseCases/Judges/`, May 2007. Accessed: 2016-05-17.

[53] Diego Berrueta and Luis Polo. Enhancing Web Searches within the Principality of Asturias. `https://www.w3.org/2001/sw/sweo/public/UseCases/CTIC/`, August 2007. Accessed: 2016-05-17.

[54] Christian Bizer, Jens Lehmann, Georgi Kobilarov, Sören Auer, Christian Becker, Richard Cyganiak, and Sebastian Hellmann. Dbpedia - a crystallization point for the web of data. *Web Semant.*, 7(3):154–165, September 2009.

[55] OCLC Online Computer Library Center, Inc. OCLC WorldCat. `https://www.worldcat.org/`, 2016. Accessed: 2016-05-17.

[56] Jinghua Groppe, Sven Groppe, Andreas Schleifer, and Volker Linnemann. LuposDate: A Semantic Web Database System. In *Proceedings of the 18th ACM Conference on Information and Knowledge Management (ACM CIKM 2009)*, pages 2083–2084, Hong Kong, China, November 2 - 6 2009. ACM.

[57] S. Groppe. LUPOSDATE Open Source. `https://github.com/luposdate`, 2013. Accessed: 2016-08-15.

[58] Sven Groppe. LUPOSDATE Demonstration. `http://www.ifis.uni-luebeck.de/index.php?id=luposdate-demo`, October 2012. Accessed: 2016-10-01.

[59] R. Bayer and E. McCreight. Organization and maintenance of large ordered indices. In *Proceedings of the 1970 ACM SIGFIDET (Now SIGMOD) Workshop on Data Description, Access and Control*, SIGFIDET '70, pages 107–141, New York, NY, USA, 1970. ACM.

[60] Douglas Comer. Ubiquitous b-tree. *ACM Comput. Surv.*, 11(2):121–137, June 1979.

[61] Thomas Neumann and Gerhard Weikum. RDF3X: a RISCstyle Engine for RDF. In *Proceedings of the 34th International Conference on Very Large Data Bases (VLDB)*, pages 647–659, Auckland, New Zealand, 2008.

[62] Cathrin Weiss, Panagiotis Karras, and Abraham Bernstein. Hexastore: Sextuple indexing for semantic web data management. *Proc. VLDB Endow.*, 1(1):1008–1019, August 2008.

[63] Donald R. Morrison. Patricia—practical algorithm to retrieve information coded in alphanumeric. *J. ACM*, 15(4):514–534, October 1968.

[64] Sven Groppe, Dennis Heinrich, Stefan Werner, Christopher Blochwitz, and Thilo Pionteck. PatTrieSort - External String Sorting based on Patricia Tries. *Open Journal of Databases (OJDB)*, 2(1):36–50, 2015.

[65] Dennis Heinrich, Stefan Werner, Marc Stelzner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. Hybrid FPGA Approach for a B+ Tree in a Semantic Web Database System. In *Proceedings of the 10th International*

Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015), Bremen, Germany, June 2015. IEEE.

[66] Sven Groppe Jinghua Groppe and Jan Kolbaum. Optimization of SPARQL by Using coreSPARQL. In José Cordeiro and Joaquim Filipe, editors, *Proceedings of the 11th International Conference on Enterprise Information Systems, Volume DISI,(ICEIS 2009)*, pages 107–112, Milano, Italien, Mai 6 - 10 2009. INSTICC.

[67] E. F. Codd. A relational model of data for large shared data banks. *Commun. ACM*, 13(6):377–387, June 1970.

[68] Thomas M. Connolly and Carolyn E. Begg. *Database Systems: A Practical Approach to Design, Implementation, and Management (Global Edition).* Pearson Education Limited, 6th edition, 2015.

[69] Kiyoshi Ono and Guy M. Lohman. Measuring the complexity of join enumeration in query optimization. In *Proceedings of the 16th International Conference on Very Large Data Bases*, VLDB '90, pages 314–325, San Francisco, CA, USA, 1990. Morgan Kaufmann Publishers Inc.

[70] Toshihide Ibaraki and Tiko Kameda. On the optimal nesting order for computing n-relational joins. *ACM Trans. Database Syst.*, 9(3):482–502, September 1984.

[71] Donald Kossmann and Konrad Stocker. Iterative dynamic programming: A new class of query optimization algorithms. *ACM Trans. Database Syst.*, 25(1):43–82, March 2000.

[72] P. A. V. Hall. Optimization of Single Expressions in a Relational Data Base System. *IBM Journal of Research and Development*, 20(3):244 – 257, May 1976.

[73] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–169, June 1993.

[74] G. Graefe. Volcano - an extensible and parallel query evaluation system. *IEEE Trans. on Knowl. and Data Eng.*, 6(1):120–135, February 1994.

[75] Michael Steinbrunn, Klaus Peithner, Guido Moerkotte, and Alfons Kemper. Bypassing joins in disjunctive queries. In *Proceedings of the 21th International Conference on Very Large Data Bases*, VLDB '95, pages 228–238, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[76] Zachary G. Ives and Nicholas E. Taylor. Sideways information passing for push-style query processing. In *Proceedings of the 2008 IEEE 24th International Conference on Data Engineering*, ICDE '08, pages 774–783, Washington, DC, USA, 2008. IEEE Computer Society.

[77] Computer History Museum. Timeline of Computer History - Memory & Storage. `http://www.computerhistory.org/timeline/memory-storage/`, 2016. Accessed: 2016-08-31.

[78] HGST, Inc. Western digital corporation is now shipping world's first helium-filled 10tb pmr hdd to meet exponential growth in data, December 2015. `http://www.hgst.com/company/media-room/press-releases/western-digital-corporation-is-now-shipping-worlds-first-helium-filled-10TB-PMR-HDD-to-meet-exponential-growth-in-data`, Accessed: 2016-08-15.

[79] Avi Mendelson. How Many Cores is Too Many Cores? In *3rd HiPEAC Industrial Workshop on Compilers and Architectures*, Haifa, Israel, April 2007. IBM.

[80] Cor Meenderinck and Ben H. H. Juurlink. (When) Will CMPs Hit the Power Wall? In *Euro-Par 2008 Workshops - Parallel Processing, VHPC 2008, UNICORE 2008, HPPC 2008, SGS 2008, PROPER 2008, ROIA 2008, and DPA 2008, Las Palmas de Gran Canaria, Spain, August 25-26, 2008, Revised Selected Papers*, pages 184–193, 2008.

[81] John Backus. Can Programming Be Liberated from the Von Neumann Style?: A Functional Style and Its Algebra of Programs. *Commun. ACM*, 21(8):613–641, August 1978.

[82] Ambuj Shatdal, Chander Kant, and Jeffrey F. Naughton. Cache conscious algorithms for relational query processing. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 510–521, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[83] Stefan Manegold, Peter A. Boncz, and Martin L. Kersten. Optimizing database architecture for the new bottleneck: Memory access. *The VLDB Journal*, 9(3):231–246, December 2000.

[84] 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*, IRE-AIEE-ACM '60 (Western), pages 33–40, New York, NY, USA, 1960. ACM.

[85] Franz J. Rammig. A concept for the editing of hardware resulting in an automatic hardware-editor. In *Proceedings of the 14th Design Automation Conference*, DAC '77, pages 187–193, Piscataway, NJ, USA, 1977. IEEE Press.

[86] Reiner W. Hartenstein, Alexander G. Hirschbiel, and M. Weber. Xputers: An open family of non-von neumann architectures. In *Architektur Von Rechensystemen, Tagungsband, 11. ITG/GI-Fachtagung*, pages 45–58, Berlin, Germany, Germany, 1990. VDE-Verlag GmbH.

[87] Xilinx. Virtex-6 Family Overview. `http://www.xilinx.com/support/documentation/data_sheets/ds150.pdf`, January 2012. DS150 (v2.5) Accessed: 2016-03-03.

[88] Xilinx. Virtex-6 FPGA Configurable Logic Block. `http://www.xilinx.com/support/documentation/user_guides/ug364.pdf`, February 2012. UG364 (v1.2).

[89] Xilinx. Virtex-6 FPGA Memory Resources. `http://www.xilinx.com/support/documentation/user_guides/ug363.pdf`, February 2014.

[90] Xilinx. Virtex-6 FPGA Memory Interface Solutions. `http://www.xilinx.com/support/documentation/ip_documentation/mig/v3_92/ug406.pdf`, March 2013. UG406.

[91] Xilinx. Virtex-6 FPGA DSP48E1 Slice. `http://www.xilinx.com/support/documentation/user_guides/ug369.pdf`, February 2011. UG369 (v1.3).

[92] Xilinx. Virtex-6 FPGA Clocking Resources. `http://www.xilinx.com/support/documentation/user_guides/ug362.pdf`, January 2014. UG362 (v2.5).

[93] Xilinx. PlanAhead User Guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/PlanAhead_UserGuide.pdf`, October 2012. UG632 (v14.3).

[94] Xilinx. Virtex-6 FPGA SelectIO Resources. `http://www.xilinx.com/support/documentation/user_guides/ug361.pdf`, November 2014. UG361 (v1.6).

[95] Xilinx. Virtex-6 FPGA GTX Transceivers. `http://www.xilinx.com/support/documentation/user_guides/ug366.pdf`, July 2011. UG366 (v2.6).

[96] Xilinx. Virtex-6 FPGA GTH Transceivers. `http://www.xilinx.com/support/documentation/user_guides/ug371.pdf`, June 2011. UG371 (v2.2).

[97] Xilinx. LogiCORE IP Virtex-6 FPGA Integrated Block v2.5 for PCI Express. `http://www.xilinx.com/support/documentation/ip_documentation/v6_pcie/v2_5/ds800_v6_pcie.pdf`, January 2012. DS800.

[98] Doug Amos, Austin Lesea, and René Richter. *FPGA-based Prototyping Methodology Manual*. Synopsys Press, February 2011.

[99] Xilinx. Virtex-6 FPGA Integrated Block for PCI Express. `http://www.xilinx.com/support/documentation/ip_documentation/v6_pcie/v2_5/ug671_V6_IntBlock_PCIe.pdf`, January 2012. UG671.

[100] Dini Group. Dini Group DNPCIe_10G_HXT_LL. `http://www.dinigroup.com/web/DNPCIe_10G_HXT_LL.php`, December 2013. Accessed: 2016-08-01.

[101] John K. Ousterhout and Ken Jones. *Tcl and the Tk Toolkit*. Addison-Wesley Professional, 2nd edition, 2009.

[102] Clif Flynt. *Tcl/Tk : A Developer's Guide*. Morgan Kaufmann (Elsevier, Inc.), 3. edition, 2011.

[103] Xilinx. Command Line Tools User Guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/devref.pdf`, July 2012. UG628 (v14.2).

[104] Gunther Lehmann, Bernhard Wunder, and Manfred Selz. *Schaltungsdesign mit VHDL*. Franzis' Verlag, 1994.

[105] IEEE. *1076-2008 - IEEE Standard VHDL Language Reference Manual*, January 2009.

[106] Frank Kesel and Ruben Bartholomä. *Entwurf von digitalen Schaltungen und Systemen mit HDLs und FPGAs - Einführung mit VHDL und SystemC*. Oldenbourg, 2nd edition, 2009.

[107] Xilinx. Synthesis and Simulation Design Guide. `http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/sim.pdf`, December 2009. UG626 v11.4.

[108] IEEE. *1364-2001 - IEEE Standard Verilog Hardware Description Language*, 2001.

[109] Xilinx. XST User Guide for Virtex-6, Spartan-6, and 7 Series Devices. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/xst_v6s6.pdf, October 2012.

[110] ORSoC AB. Opencores. http://opencores.org/. Accessed: 2016-08-31.

[111] Xilinx. Constraints Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx11/cgd.pdf, December 2009. UG625 (v11.4).

[112] Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer. *Logic Synthesis*. McGraw-Hill, Inc., New York, NY, USA, 1994.

[113] Jie-Hong Roland Jiang and Srinivas Devadas. Logic Synthesis in a Nutshell . http://flolac.iis.sinica.edu.tw/flolac09/lib/exe/logic-synthesis.pdf, October 2008.

[114] Xilinx. Virtex-6 Libraries Guide for HDL Designs. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_4/virtex6_hdl.pdf, October 2012.

[115] Xilinx. Virtex-6 FPGA Routing Optimization Design Techniques. http://www.xilinx.com/support/documentation/white_papers/wp381_V6_Routing_Optimization.pdf, October 2010. WP381 (v1.0).

[116] Xilinx. Writing Efficient Testbenches. http://www.xilinx.com/support/documentation/application_notes/xapp199.pdf, May 2010. XAPP199 (v1.1).

[117] Scott Hauck and André Dehon. Reconfigurable computing hardware. In Scott Hauck and André Dehon, editors, *Reconfigurable Computing*, Systems on Silicon, pages 1–2. Morgan Kaufmann, Burlington, 2008.

[118] Xilinx. Partial Reconfiguration User Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/ug702.pdf, April 2013. UG702 (v14.5).

[119] Xilinx. Partial Reconfiguration Tutorial - PlanAhead Design Tool. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_1/PlanAhead_Tutorial_Partial_Reconfiguration.pdf, May 2012. UG743 (v14.1).

[120] Xilinx. Hierarchical Design Methodology Guide. http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_5/Hierarchical_Design_Methodology_Guide.pdf, April 2013. UG748 (v14.5).

[121] Xilinx. Virtex-6 FPGA Configuration. `http://www.xilinx.com/support/documentation/user_guides/ug360.pdf`, November 2015. UG360 (v3.9).

[122] Xilinx. Fast Configuration of PCI Express Technology through Partial Reconfiguration. `http://www.xilinx.com/support/documentation/application_notes/xapp883_Fast_Config_PCIe.pdf`, November 2010. XAPP883 (v1.0).

[123] Xilinx. Partial Reconfiguration of Xilinx FPGAs Using ISE Design Suite. `http://www.xilinx.com/support/documentation/white_papers/wp374_Partial_Reconfig_Xilinx_FPGAs.pdf`, May 2012. WP374 (v1.2).

[124] Kyprianos Papadimitriou, Apostolos Dollas, and Scott Hauck. Performance of partial reconfiguration in fpga systems: A survey and a cost model. *ACM Trans. Reconfigurable Technol. Syst.*, 4(4), 2011.

[125] Dirk Koch, Christian Beckhoff, and Jim Torresen. Zero logic overhead integration of partially reconfigurable modules. In *Proceedings of the 23rd Symposium on Integrated Circuits and System Design*, SBCCI '10, pages 103–108, New York, NY, USA, 2010. ACM.

[126] Dirk Koch, Jim Tørresen, Christian Beckhoff, Daniel Ziener, Christopher Dennl, Volker Breuer, Jürgen Teich, Michael Feilen, and Walter Stechele. Partial reconfiguration on fpgas in practice - tools and applications. In *ARCS 2012 Workshops, 28. Februar - 2. März 2012, München, Germany*, pages 297–319, 2012.

[127] Scott Hauck and André DeHon. Preface. In Scott Hauck and André Dehon, editors, *Reconfigurable Computing*, Systems on Silicon, pages xxiii – xxiv. Morgan Kaufmann, Burlington, 2008.

[128] Roger Woods, Katherine Compton, Christos Bouganis, and Pedro C. Diniz, editors. *Reconfigurable Computing: Architectures, Tools and Applications (4th International Workshop, ARC 2008 Proceedings)*. Lecture Notes in Computer Science. Springer-Verlag Berlin Heidelberg, March 2008.

[129] Altera Corporation. Consumer Applications. `https://www.altera.com/solutions/industry/consumer/overview.html`. Accessed: 2016-08-01.

[130] Xilinx Inc. Consumer Electronics - Smarter Vision: Intelligence for Smarter Consumer Systems. `http://www.xilinx.com/applications/consumer-electronics.html`. Accessed: 2016-08-01.

[131] Thomas W. Fry and Scott Hauck. Chapter 27 - SPIHT image compression. In Scott Hauck and André Dehon, editors, *Reconfigurable Computing*, Systems on Silicon, pages 565 – 590. Morgan Kaufmann, Burlington, 2008.

[132] Stephen J. Bellis, Kieran Delaney, John Barton, and Kafil M. Razeeb. Development of field programmable modular wireless sensor network nodes for ambient systems. In *In Computer Communications, Special Issue on Wireless Sensor Networks*, 2005.

[133] Xilinx. High Performance Computing Using FPGAs (WP375). `http://www.xilinx.com/support/documentation/white_papers/wp375_HPC_Using_FPGAs.pdf`, 2010. Accessed: 2016-03-03.

[134] Chen Zhang, Peng Li, Guangyu Sun, Yijin Guan, Bingjun Xiao, and Jason Cong. Optimizing FPGA-based accelerator design for deep convolutional neural networks. In *Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '15, pages 161–170, New York, NY, USA, 2015. ACM.

[135] Kalin Ovtcharov, Olatunji Ruwase, Joo-Young Kim, Jeremy Fowers, Karin Strauss, and Eric S. Chung. Accelerating deep convolutional neural networks using specialized hardware. In *White paper*. Microsoft Research, February 2015.

[136] *Toward Accelerating Deep Learning at Scale Using Specialized Logic*. HOTCHIPS, August 2015.

[137] Naveen Suda, Vikas Chandra, Ganesh Dasika, Abinash Mohanty, Yufei Ma, Sarma Vrudhula, Jae-sun Seo, and Yu Cao. Throughput-optimized opencl-based FPGA accelerator for large-scale convolutional neural networks. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 16–25, New York, NY, USA, 2016. ACM.

[138] Jiantao Qiu, Jie Wang, Song Yao, Kaiyuan Guo, Boxun Li, Erjin Zhou, Jincheng Yu, Tianqi Tang, Ningyi Xu, Sen Song, Yu Wang, and Huazhong Yang. Going deeper with embedded FPGA platform for convolutional neural network. In *Proceedings of the 2016 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*, FPGA '16, pages 26–35, New York, NY, USA, 2016. ACM.

[139] P. K. Gupta (Intel). Intel Xeon + FPGA Platform for the Data Center. In *FPL'15 Workshop on Reconfigurable Computing for the Masses*, September 2015.

[140] Gabriel Weisz, Joseph Melber, Yu Wang, Kermin Fleming, Eriko Nurvitadhi, and James C. Hoe. A study of pointer-chasing performance on shared-memory processor-FPGA systems. In *Proceedings of the 2016 ACM/SIGDA*

International Symposium on Field-Programmable Gate Arrays, FPGA '16, pages 264–273, New York, NY, USA, 2016. ACM.

[141] Hans-Otto Leilich, Günther Stiege, and Hans Christoph Zeidler. A search processor for data base management systems. In *Proceedings of the fourth international conference on Very Large Data Bases - Volume 4*, VLDB'1978, pages 280–287. VLDB Endowment, 1978.

[142] D.J. DeWitt. Direct - a multiprocessor organization for supporting relational database management systems. *Computers, IEEE Transactions on*, C-28(6):395–406, June 1979.

[143] Dina Bitton, David J. DeWitt, and Carolyn Turbyfill. Benchmarking database systems a systematic approach. In *Proceedings of the 9th International Conference on Very Large Data Bases*, VLDB '83, pages 8–19, San Francisco, CA, USA, 1983. Morgan Kaufmann Publishers Inc.

[144] D. J. Dewitt, S. Ghandeharizadeh, D. A. Schneider, A. Bricker, H. I. Hsiao, and R. Rasmussen. The gamma database machine project. *IEEE Trans. on Knowl. and Data Eng.*, 2(1):44–62, March 1990.

[145] Michael Ubell. The Intelligent Database Machine (IDM). In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*, Topics in Information Systems, chapter VII, pages 237–247. Springer Berlin Heidelberg, 1985.

[146] David J. DeWitt, Robert H. Gerber, Goetz Graefe, Michael L. Heytens, Krishna B. Kumar, and M. Muralikrishna. Gamma - a high performance dataflow database machine. In *Proceedings of the 12th International Conference on Very Large Data Bases*, VLDB '86, pages 228–237, San Francisco, CA, USA, 1986. Morgan Kaufmann Publishers Inc.

[147] Jun Rao and Kenneth A. Ross. Cache conscious indexing for decision-support in main memory. In *Proceedings of the 25th International Conference on Very Large Data Bases*, VLDB '99, pages 78–89, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc.

[148] Stefan Manegold, Peter Boncz, Martin Kersten, and Ieee Computer Society. Optimizing main-memory join on modern hardware. *IEEE Transactions on Knowledge and Data Eng*, 14:2002, 2002.

[149] Peter A. Boncz, Martin L. Kersten, and Stefan Manegold. Breaking the memory wall in monetdb. *Commun. ACM*, 51(12):77–85, December 2008.

[150] Georgios Giannikis, Gustavo Alonso, and Donald Kossmann. Shareddb: Killing one thousand queries with one stone. *Proc. VLDB Endow.*, 5(6):526–537, February 2012.

[151] Max Heimel, Michael Saecker, Holger Pirk, Stefan Manegold, and Volker Markl. Hardware-oblivious parallelism for in-memory column-stores. *Proc. VLDB Endow.*, 6(9):709–720, July 2013.

[152] Shekhar Borkar and Andrew A. Chien. The future of microprocessors. *Commun. ACM*, 54(5):67–77, May 2011.

[153] Sang-Won Lee, Bongki Moon, and Chanik Park. Advances in flash memory ssd technology for enterprise database applications. In *Proceedings of the 2009 ACM SIGMOD International Conference on Management of Data*, SIGMOD '09, pages 863–870, New York, NY, USA, 2009. ACM.

[154] Kimberly Keeton, David A. Patterson, and Joseph M. Hellerstein. A case for intelligent disks (idisks). *SIGMOD Rec.*, 27(3):42–52, September 1998.

[155] Erik Riedel, Garth A. Gibson, and Christos Faloutsos. Active storage for large-scale data mining and multimedia. In *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, pages 62–73, San Francisco, CA, USA, 1998. Morgan Kaufmann Publishers Inc.

[156] Erik Riedel, Christos Faloutsos, Garth A. Gibson, and David Nagle. Active disks for large-scale data processing. *Computer*, 34(6):68–74, June 2001.

[157] Anurag Acharya, Mustafa Uysal, and Joel Saltz. Active disks: Programming model, algorithms and evaluation. *SIGPLAN Not.*, 33(11):81–91, October 1998.

[158] Yangwook Kang, Yang suk Kee, Ethan L. Miller, and Chanik Park. Enabling cost-effective data processing with smart ssd. In *the 29th IEEE Symposium on Massive Storage Systems and Technologies (MSST 13)*, May 2013.

[159] Sangyeun Cho, Chanik Park, Hyunok Oh, Sungchan Kim, Youngmin Yi, and Gregory R. Ganger. Active disk meets flash: A case for intelligent ssds. In *Proceedings of the 27th International ACM Conference on International Conference on Supercomputing*, ICS '13, pages 91–102, New York, NY, USA, 2013. ACM.

[160] Jaeyoung Do, Yang-Suk Kee, Jignesh M. Patel, Chanik Park, Kwanghyun Park, and David J. DeWitt. Query processing on smart ssds: Opportunities and challenges. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1221–1230, New York, NY, USA, 2013. ACM.

[161] Sudharsan Seshadri, Mark Gahagan, Sundaram Bhaskaran, Trevor Bunker, Arup De, Yanqin Jin, Yang Liu, and Steven Swanson. Willow: A user-programmable ssd. In *Proceedings of the 11th USENIX Conference on Operating Systems Design and Implementation*, OSDI'14, pages 67–80, Berkeley, CA, USA, 2014. USENIX Association.

[162] Chengyu Sun, Divyakant Agrawal, and Amr El Abbadi. Hardware acceleration for spatial selections and joins. In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data*, SIGMOD '03, pages 455–466, New York, NY, USA, 2003. ACM.

[163] Naga K. Govindaraju, Brandon Lloyd, Wei Wang, Ming Lin, and Dinesh Manocha. Fast computation of database operations using graphics processors. In *Proceedings of the 2004 ACM SIGMOD International Conference on Management of Data*, SIGMOD '04, pages 215–226, New York, NY, USA, 2004. ACM.

[164] Naga Govindaraju, Jim Gray, Ritesh Kumar, and Dinesh Manocha. Gputerasort: High performance graphics co-processor sorting for large database management. In *Proceedings of the 2006 ACM SIGMOD International Conference on Management of Data*, SIGMOD '06, pages 325–336, New York, NY, USA, 2006. ACM.

[165] NVIDIA Corporation. Compute Unified Device Architecture (CUDA). `https://developer.nvidia.com/cuda-zone`. Accessed: 2016-07-01.

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

[167] Bingsheng He, Mian Lu, Ke Yang, Rui Fang, Naga K. Govindaraju, Qiong Luo, and Pedro V. Sander. Relational query coprocessing on graphics processors. *ACM Trans. Database Syst.*, 34(4):21:1–21:39, December 2009.

[168] Transaction Processing Performance Council. TPC-H Benchmark. `http://www.tpc.org/tpch/default.asp`. Accessed: 2016-07-01.

[169] Wenbin Fang, Bingsheng He, and Qiong Luo. Database compression on graphics processors. *Proc. VLDB Endow.*, 3(1-2):670–680, September 2010.

[170] Bingsheng He and Jeffrey Xu Yu. High-throughput transaction executions on graphics processors. *Proc. VLDB Endow.*, 4(5):314–325, February 2011.

[171] Jiong He, Shuhao Zhang, and Bingsheng He. In-cache query co-processing on coupled cpu-gpu architectures. *Proc. VLDB Endow.*, 8(4):329–340, December 2014.

[172] Xuntao Cheng, Bingsheng He, and Chiew Tong Lau. Energy-efficient query processing on embedded cpu-gpu architectures. In *Proceedings of the 11th International Workshop on Data Management on New Hardware*, DaMoN'15, pages 10:1–10:7, New York, NY, USA, 2015. ACM.

[173] Sebastian Breß and Gunter Saake. Why it is time for a hype: A hybrid query processing engine for efficient gpu coprocessing in dbms. *Proc. VLDB Endow.*, 6(12):1398–1403, August 2013.

[174] Sebastian Breß, Bastian Köcher, Max Heimel, Volker Markl, Michael Saecker, and Gunter Saake. Ocelot/hype: Optimized data processing on heterogeneous hardware. *Proc. VLDB Endow.*, 7(13):1609–1612, August 2014.

[175] Sebastian Breß. *Efficient Query Processing in Co-Processor-accelerated Databases*. PhD thesis, Otto-von-Guericke-Universität Magdeburg, 2015.

[176] Chris Gregg and Kim Hazelwood. Where is the data? why you cannot debate cpu vs. gpu performance without the answer. In *Proceedings of the IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS '11, pages 134–144, Washington, DC, USA, 2011. IEEE Computer Society.

[177] Amirhesam Shahvarani and Hans-Arno Jacobsen. A hybrid b+-tree as solution for in-memory indexing on CPU-GPU heterogeneous computing platforms. In *Proceedings of the 2016 International Conference on Management of Data*, SIGMOD '16, pages 1523–1538, New York, NY, USA, 2016. ACM.

[178] Steven A. Guccione. Chapter 3 - reconfigurable computing systems. In Scott Hauck and André Dehon, editors, *Reconfigurable Computing*, Systems on Silicon, pages 47–64. Morgan Kaufmann, Burlington, 2008.

[179] Rakesh Agrawal and Ramakrishnan Srikant. Fast algorithms for mining association rules in large databases. In *Proceedings of the 20th International Conference on Very Large Data Bases*, VLDB '94, pages 487–499, San Francisco, CA, USA, 1994. Morgan Kaufmann Publishers Inc.

[180] Zachary K. Baker and Viktor K. Prasanna. Efficient hardware data mining with the apriori algorithm on fpgas. In *Proceedings of the 13th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, FCCM '05, pages 3–12, Washington, DC, USA, 2005. IEEE Computer Society.

[181] Yi Shan, Bo Wang, Jing Yan, Yu Wang, Ningyi Xu, and Huazhong Yang. FPmr: Mapreduce framework on FPGA. In *Proceedings of the 18th Annual ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, FPGA '10, pages 93–102, New York, NY, USA, 2010. ACM.

[182] Jeffrey Dean and Sanjay Ghemawat. Mapreduce: Simplified data processing on large clusters. *Commun. ACM*, 51(1):107–113, January 2008.

[183] Yoav Freund, Raj Iyer, Robert E. Schapire, and Yoram Singer. An efficient boosting algorithm for combining preferences. *J. Mach. Learn. Res.*, 4:933–969, December 2003.

[184] Rene Mueller, Jens Teubner, and Gustavo Alonso. Data processing on fpgas. *Proc. VLDB Endow.*, 2:910–921, August 2009.

[185] Dirk Koch and Jim Torresen. Fpgasort: a high performance sorting architecture exploiting run-time reconfiguration on fpgas for large problem sorting. In *Proceedings of the 19th ACM/SIGDA international symposium on Field programmable gate arrays*, FPGA '11, pages 45–54, New York, NY, USA, 2011. ACM.

[186] Jared Casper and Kunle Olukotun. Hardware acceleration of database operations. In *Proceedings of the 2014 ACM/SIGDA International Symposium on Field-programmable Gate Arrays*, FPGA '14, pages 151–160, New York, NY, USA, 2014. ACM.

[187] Donald E. Knuth. *The art of computer programming: Sorting and searching*. The Art of Computer Programming. Addison-Wesley, 2nd edition, 1998.

[188] Jens Teubner and Rene Mueller. How Soccer Players Would do Stream Joins. In *Proceedings of the 2011 International Conference on Management of Data*, SIGMOD '11, pages 625–636, New York, NY, USA, 2011. ACM.

[189] Louis Woods, Gustavo Alonso, and Jens Teubner. Parallel computation of skyline queries. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:1–8, 2013.

[190] Louis Woods, Gustavo Alonso, and Jens Teubner. Parallelizing data processing on FPGAs with shifter lists. *ACM Trans. Reconfigurable Technol. Syst.*, 8(2):7:1–7:22, March 2015.

[191] Stephan Börzsönyi, Donald Kossmann, and Konrad Stocker. The skyline operator. In *Proc. 17th Int. Conf. on Data Engineering (ICDE)*, pages 421–430, Heidelberg, Germany, 2001.

[192] Jens Teubner, René Müller, and Gustavo Alonso. Frequent item computation on a chip. *IEEE Trans. Knowl. Data Eng.*, 23(8):1169–1181, 2011.

[193] P. Drineas, A. Frieze, R. Kannan, S. Vempala, and V. Vinay. Clustering large graphs via the singular value decomposition. *Mach. Learn.*, 56(1-3):9–33, June 2004.

[194] Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Balakrishna Iyer, Bernard Brezzo, Donna Dillenberger, and Sameh Asaad. Database analytics acceleration using fpgas. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, pages 411–420, New York, NY, USA, 2012. ACM.

[195] Robert J. Halstead, Bharat Sukhwani, Hong Min, Mathew Thoennes, Parijat Dube, Sameh Asaad, and Balakrishna Iyer. Accelerating join operation for relational databases with fpgas. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:17–20, 2013.

[196] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. A flexible hash table design for 10gbps key-value stores on fpgas. In *23rd International Conference on Field programmable Logic and Applications (FPL 2013)*, pages 1 – 8, New York, NY, USA, September 2013. IEEE.

[197] Zsolt István, Gustavo Alonso, Michaela Blott, and Kees Vissers. A hash table for line-rate data processing. *ACM Trans. Reconfigurable Technol. Syst.*, 8(2):13:1–13:15, March 2015.

[198] Louis Woods, Jens Teubner, and Gustavo Alonso. Less watts, more performance: An intelligent storage engine for data appliances. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, SIGMOD '13, pages 1073–1076, New York, NY, USA, 2013. ACM.

[199] Louis Woods, Zsolt István, and Gustavo Alonso. Ibex: An intelligent storage engine with support for advanced sql offloading. *Proc. VLDB Endow.*, 7(11):963–974, July 2014.

[200] Zsolt István, Louis Woods, and Gustavo Alonso. Histograms as a side effect of data movement for big data. In *Proceedings of the 2014 ACM SIGMOD International Conference on Management of Data*, SIGMOD '14, pages 1567–1578, Snowbird, UT, USA, 2014. ACM.

[201] Sven Groppe. Rule Documentation - Rule Constant Propagation. `http://www.ifis.uni-luebeck.de/~groppe/tutorial_demo/ruledoc/constantpropagationRule.html`. Accessed: 2016-08-15.

[202] Andreas Harth. Billion Triples Challenge dataset. `http://km.aifb.kit.edu/projects/btc-2012/`, 2012. Accessed: 2016-08-15.

[203] Xillybus Ltd. Xillybus - IP core product brief. `http://xillybus.com/downloads/xillybus_product_brief.pdf`, January 2016. Accessed: 2016-08-15 (v1.8).

[204] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench. `http://dbis.informatik.uni-freiburg.de/index.php?project=SP2B/download.php`, January 2009. Accessed: 2016-10-01.

[205] DBLP. Computer Science Bibliography. `http://dblp.uni-trier.de`. Accessed: 2016-08-31.

[206] Michael Schmidt, Thomas Hornung, Georg Lausen, and Christoph Pinkel. SP2Bench: A SPARQL Performance Benchmark. In *Proceedings of the 25th International Conference on Data Engineering, ICDE 2009, March 29 2009 - April 2 2009, Shanghai, China*, pages 222–233, 2009.

[207] IBM Corp. The Netezza Data Appliance Architecture: A Platform for High Performance Data Warehousing and Analytics, 2011.

[208] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires: A Query Compiler for FPGAs. *Proc. VLDB Endow.*, 2:229–240, August 2009.

[209] Rene Mueller and Jens Teubner. Fpga: what's in it for a database? In *Proceedings of the 35th SIGMOD international conference on Management of data*, SIGMOD '09, pages 999–1004, New York, NY, USA, 2009. ACM.

[210] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *Proceedings of the 2010 International Conference on Management of Data*, SIGMOD '10, pages 1159–1162, New York, NY, USA, 2010. ACM.

[211] Takashi Takenaka, Masamichi Takagi, and Hiroaki Inoue. A scalable complex event processing framework for combination of sql-based continuous queries and c/c++ functions. In *22nd International Conference on Field Programmable Logic and Applications (FPL)*, pages 237 – 242, Oslo, Norway, 2012.

[212] Python Software Foundation. Python. `https://www.python.org/`. Accessed: 2016-08-15.

[213] Rico Backasch, Gerald Hempel, Sven Groppe, Stefan Werner, and Thilo Pionteck. Identifying Homogenous Reconfigurable Regions in Heterogeneous FPGAs for Module Relocation. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2014.

[214] Christopher Dennl, Daniel Ziener, and Jürgen Teich. On-the-fly Composition of FPGA-Based SQL Query Accelerators Using a Partially Reconfigurable Module Library. *20th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, 20:45–52, 2012.

[215] Christopher Dennl, Daniel Ziener, and Jurgen Teich. Acceleration of SQL Restrictions and Aggregations Through FPGA-Based Dynamic Partial Reconfiguration. In *Proceedings of the 2013 IEEE 21st Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM '13, pages 25–28, Washington, DC, USA, 2013. IEEE Computer Society.

[216] Andreas Becher, Florian Bauer, Daniel Ziener, and Jürgen Teich. Energy-Aware SQL Query Acceleration through FPGA-Based Dynamic Partial Reconfiguration. In *Proceedings of the 24th International Conference on Field Programmable Logic and Applications (FPL 2014)*, pages 662–669. IEEE, 2014.

[217] Daniel Ziener, Florian Bauer, Andreas Becher, Christopher Dennl, Klaus Meyer-Wegener, Ute Schürfeld, Jürgen Teich, Jörg-Stephan Vogt, and Helmut Weber. Fpga-based dynamically reconfigurable sql query processing. *ACM Trans. Reconfigurable Technol. Syst.*, 9(4):25:1–25:24, August 2016.

[218] Raphael Polig, Kubilay Atasu, Laura Chiticariu, Christoph Hagleitner, H. Peter Hofstee, Frederick R. Reiss, Huaiyu Zhu, and Eva Sitaridi. Giving Text Analytics a Boost. *IEEE Micro*, 34(4):6–14, 2014.

[219] Jens Teubner, Louis Woods, and Chongling Nie. Skeleton automata for fpgas: Reconfiguring without reconstructing. In *Proc. ACM SIGMOD'12*, pages 229–240, 2012.

[220] Jens Teubner, Louis Woods, and Chongling Nie. Xlynx—an FPGA-based XML filter for hybrid xquery processing. *ACM Trans. Database Syst.*, 38(4):23:1–23:39, December 2013.

[221] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient event processing through reconfigurable hardware for algorithmic trading. *Proc. VLDB Endow.*, 3(1-2):1525–1528, September 2010.

[222] Mohammad Sadoghi, Rija Javed, Naif Tarafdar, Harsh Singh, Rohan Palaniappan, and Hans-Arno Jacobsen. Multi-query Stream Processing on FPGAs. In Anastasios Kementsietsidis and Marcos Antonio Vaz Salles, editors, *ICDE*, pages 1229–1232. IEEE Computer Society, 2012.

[223] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Flexible Query Processor on FPGAs. *Proc. VLDB Endow.*, 6(12):1310–1313, August 2013.

[224] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. The FQP Vision: Flexible Query Processing on a Reconfigurable Computing Fabric. *SIGMOD Rec.*, 44(2):5–10, August 2015.

[225] Mohammadreza Najafi, Mohammad Sadoghi, and Hans-Arno Jacobsen. Configurable hardware-based streaming architecture using Online Programmable-Blocks. In *31st IEEE International Conference on Data Engineering, ICDE 2015, Seoul, South Korea, April 13-17, 2015*, pages 819–830, 2015.

[226] Yoshihiro Ichinomiya, Motoki Amagasaki, Masahiro Iida, Morihiro Kuga, and Toshinori Sueyoshi. *A Bitstream Relocation Technique to Improve Flexibility of Partial Reconfiguration*, pages 139–152. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[227] Timos K. Sellis. Multiple-query optimization. *ACM Trans. Database Syst.*, 13(1):23–52, March 1988.

[228] Nilesh N. Dalvi, Sumit K. Sanghai, Prasan Roy, and S. Sudarshan. Pipelining in multi-query optimization. In *Proceedings of the Twentieth ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, PODS '01, pages 59–70, New York, NY, USA, 2001. ACM.

[229] Rico Backasch, Gerald Hempel, Thilo Pionteck, Sven Groppe, and Stefan Werner. An Architectural Template for Composing Application Specific Datapaths at Runtime. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2015.

[230] Stavros Harizopoulos and Anastassia Ailamaki. A case for staged database systems. In *CIDR*, 2003.

[231] Stavros Harizopoulos and Anastassia Ailamaki. Stageddb: Designing database servers for modern hardware. *IEEE Data Eng. Bull.*, 28(2):11–16, 2005.

[232] Stavros Harizopoulos, Vladislav Shkapenyuk, and Anastassia Ailamaki. Qpipe: A simultaneously pipelined relational query engine. In *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data*, SIGMOD '05, pages 383–394, New York, NY, USA, 2005. ACM.

[233] Jeremy Fowers, Greg Brown, Patrick Cooke, and Greg Stitt. A performance and energy comparison of fpgas, gpus, and multicores for sliding-window applications. In *Proceedings of the ACM/SIGDA international symposium on Field Programmable Gate Arrays*, FPGA '12, pages 47–56, New York, NY, USA, 2012. ACM.

[234] Xillybus Ltd. Xillybus host application programming guide for linux. `http://xillybus.com/downloads/doc/xillybus_host_programming_guide_linux.pdf`. Accessed: 2016-08-15 (Version 2.2).

[235] Xillybus Ltd. The custom ip core factory. `http://xillybus.com/custom-ip-factory`. Accessed: 2016-08-15.

[236] Xillybus Ltd. Download Xillybus for PCIe. `http://xillybus.com/pcie-download`. Accessed: 2016-08-15.

[237] Xillybus Ltd. Xillybus' data bandwidth. `http://xillybus.com/doc/xillybus-bandwidth`. Accessed: 2016-08-15.

Lists

List of Figures

List of Listings

List of Tables

Curriculum Vitae

Stefan Werner

Persönliches

Geburtstag	27. August 1984
Geburtsort	Karl-Marx-Stadt, jetzt Chemnitz

Beruflicher Werdegang

04/2011 – 10/2016	Wissenschaftlicher Mitarbeiter Institut für Informationssysteme *Universität zu Lübeck*

Ausbildung

10/2005 – 03/2011	Diplomstudium der Informatik (Dipl.-Inf.) mit Nebenfach Medieninformatik *Universität zu Lübeck*
10/2004 – 06/2005	Grundwehrdienst *6./Jägerbataillon 371, Marienberg*
08/2001 – 07/2004	Gymnasium (Abitur) *Berufliches Schulzentrum für Technik II, Chemnitz*
08/1995 – 06/2001	Mittelschule (Realschulabschluss) *Yorckschule, Chemnitz*

List of Personal Publications

Stefan Werner, Dennis Heinrich, Sven Groppe, Christopher Blochwitz, and Thilo Pionteck. Runtime Adaptive Hybrid Query Engine based on FPGAs. *Open Journal of Databases (OJDB)*, 3(1):21–41, 2016.

Rico Backasch, Gerald Hempel, Thilo Pionteck, Sven Groppe, and **Stefan Werner**. An Architectural Template for Composing Application Specific Datapaths at Runtime. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2015.

Christopher Blochwitz, Jan Moritz Joseph, Thilo Pionteck, Rico Backasch, **Stefan Werner**, Dennis Heinrich, and Sven Groppe. An optimized Radix-Tree for hardware-accelerated index generation for Semantic Web Databases. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2015.

Dennis Heinrich, **Stefan Werner**, Marc Stelzner, Christopher Blochwitz, Thilo Pionteck, and Sven Groppe. Hybrid FPGA Approach for a B+ Tree in a Semantic Web Database System. In *Proceedings of the 10^{th} International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, June 2015. IEEE.

Stefan Werner, Dennis Heinrich, Jannik Piper, Sven Groppe, Rico Backasch, Christopher Blochwitz, and Thilo Pionteck. Automated Composition and Execution of Hardware-accelerated Operator Graphs. In *Proceedings of the 10^{th} International Symposium on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2015)*, Bremen, Germany, June 2015. IEEE.

Stefan Werner, Dennis Heinrich, Marc Stelzner, Volker Linnemann, Thilo Pionteck, and Sven Groppe. Accelerated join evaluation in Semantic Web databases by using FPGAs. *Concurrency and Computation: Practice and Experience*, 28(7):2031–2051, May 2015.

Sven Groppe, Dennis Heinrich, and **Stefan Werner**. Distributed Join Approaches for W3C-Conform SPARQL Endpoints. *Open Journal of Semantic Web (OJSW)*, 2(1):30–52, 2015.

Sven Groppe, Dennis Heinrich, **Stefan Werner**, Christopher Blochwitz, and Thilo Pionteck. PatTrieSort - External String Sorting based on Patricia Tries. *Open Journal of Databases (OJDB)*, 2(1):36–50, 2015.

Rico Backasch, Gerald Hempel, Sven Groppe, **Stefan Werner**, and Thilo Pionteck. Identifying Homogenous Reconfigurable Regions in Heterogeneous FPGAs for Module Relocation. In *International Conference on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, December 2014.

Stefan Werner, Dennis Heinrich, Marc Stelzner, Sven Groppe, Rico Backasch, and Thilo Pionteck. Parallel and Pipelined Filter Operator for Hardware-Accelerated Operator Graphs in Semantic Web Databases. In *Proceedings of the 14th IEEE International Conference on Computer and Information Technology (CIT2014)*, pages 539–546, Xi'an, China, September 2014. IEEE.

Sven Groppe, Johannes Blume, Dennis Heinrich, and **Stefan Werner**. A Self-Optimizing Cloud Computing System for Distributed Storage and Processing of Semantic Web Data. *Open Journal of Cloud Computing (OJCC)*, 1(2):1–14, 2014.

Sven Groppe, Thomas Kiencke, **Stefan Werner**, Dennis Heinrich, Marc Stelzner, and Le Gruenwald. P-LUPOSDATE: Using Precomputed Bloom Filters to Speed Up SPARQL Processing in the Cloud. *Open Journal of Semantic Web (OJSW)*, 1(2):25–55, 2014.

Stefan Werner, Sven Groppe, Volker Linnemann, and Thilo Pionteck. Hardware-accelerated Join Processing in Large Semantic Web Databases with FPGAs. In *Proceedings of the 2013 International Conference on High Performance Computing & Simulation (HPCS 2013)*, pages 131–138, Helsinki, Finland, July 2013. IEEE.

Sven Groppe, Björn Schütt, and **Stefan Werner**. Eliminating the XML Overhead in Embedded XML Languages. In *Proceedings of the 28th ACM Symposium on Applied Computing (ACM SAC 2013)*, Coimbra, Portugal, March 2013. ACM.

Stefan Werner, Christoph Reinke, Sven Groppe, and Volker Linnemann. Adaptive Service Migration in Wireless Sensor Networks. In *Proceedings of the 12th International Conference on Parallel and Distributed Computing, Applications and Technologies (PDCAT-11)*, Gwangju, Korea, October 2011. IEEE.

Sven Groppe, Jinghua Groppe, **Stefan Werner**, Matthias Samsel, Florian Kalis, Kristina Fell, Peter Kliesch, and Markus Nakhlah. Monitoring eBay Auctions by querying RDF Streams. In *Proceedings of the Sixth International Conference on Digital Information Management (ICDIM 2011)*, pages 223–228, Trinity College, The University of Melbourne, Australia, September 2011. CPS.

Christoph Reinke, Nils Hoeller, **Stefan Werner**, Sven Groppe, and Volker Linnemann. Analysis and Comparison of Concurrency Control Protocols for Wireless Sensor Networks. In *Proceedings of the 3rd International Workshop on Performance Control in Wireless Sensor Networks (PWSN 2011) in conjunction with the 7th IEEE International Conference on Distributed Computing in Sensor Systems (IEEE DCOSS '11)*, pages 1–6, Barcelona, Spain, June 2011. IEEE.

Christoph Reinke, Nils Hoeller, **Stefan Werner**, Sven Groppe, and Volker Linnemann. Consistent Service Migration in Wireless Sensor Networks. In *Proceedings of the 2011 International Conference on Wireless and Optical Communications (ICWOC 2011 - including ICIME 2011)*, pages 278–285, Zhengzhou, China, May 2011. IEEE.

Sven Groppe, Jinghua Groppe, **Stefan Werner**, Matthias Samsel, Florian Kalis, Kristina Fell, Peter Kliesch, and Markus Naklah. Using a Streaming SPARQL Evaluator for Monitoring eBay Auctions. *International Journal of Web Applications (IJWA)*, 3(4):166–178, 2011.