Master’s Thesis Nr. 57
Systems Group, Department of Computer Science, ETH Zurich

An FPGA-based Smart Database Storage Engine

by
Chongling Nie

Supervised by
Louis Woods
Prof. Dr. Gustavo Alonso

March 2012 – September 2012


Abstract
Systems Group, Department of Computer Science
ETH Zurich

Master Thesis

An FPGA-based Smart Database Storage Engine

by Chongling Nie

Ever increasing database sizes and query complexity presents an opportunity for hybrid, FPGA-based co-processor systems to significantly improve database performance. Although performance opportunities for FPGAs in fast data processing field are well-known, the design of a FPGA-based hybrid system still need for compromise between system flexibility and high FPGA-compilation cost. Elimination of this compilation cost would dramatically increase the efficiency of FPGA-based hybrid systems.

In this thesis, we implement an FPGA-based database storage engine that achieves this goal. Our FPGA-based database storage engine is able to accept dynamic selection and projection based filtering queries at runtime without extra FPGA-compilation cost. This is accomplished through two parts. First, we introduce a FPGA-readable data storage format, which allows FPGAs to recognize and project data on-the-fly. Second, we design a selection evaluation module based on the selection truth table, which gives our storage engine the capability of evaluating dynamic Boolean expressions at wire speed. Furthermore, we explore the FPGA-DB server interface, which is accomplished through implementing a data communication framework based on PCI Express. Finally, we show that the performance is able to gain through our FPGA-based co-processing database storage engine.
Acknowledgements

I would like to thank Louis Woods for his continuous support, valuable feedback, and for all guidances during the course of this thesis. I would also like to thank Dr. Jens Teubner and Ionut Subasu for interesting discussions and useful suggestions.

Furthermore, I would like to thank my mentor Prof. Gustavo Alonso for his invaluable support and guidance during my master studies at ETH Zurich.

Lastly, I am very thankful to my family and my friends for their encouragement and support.
# Contents

Abstract 

i

Acknowledgements iii

List of Figures vii

List of Tables ix

List of Listings xi

1 Introduction 1

1.1 Background and Motivation .............................................. 1
1.2 Problem Statement ....................................................... 2
1.3 Contribution .............................................................. 3
1.4 Outline of the Thesis ..................................................... 3

2 Fundamentals 5

2.1 FPGA ................................................................. 5
   2.1.1 FPGA Internals .................................................... 5
   2.1.2 Potential and Limitations ......................................... 7
2.2 Database Storage Engine ............................................... 7
2.3 System Overview ......................................................... 8

3 Storage Data Format Design 11

3.1 Data Format on Disks .................................................... 11
3.2 Plain Data Format ....................................................... 12
   3.2.1 Markup Operator Definition .................................... 13
   3.2.2 Database Table Schema Translation ............................. 13
   3.2.3 Data Content ....................................................... 15

4 Implementing Storage Engine in FPGAs 17

4.1 Data Collector .......................................................... 19
4.2 Data Parser ............................................................. 20
4.3 Data Projector .......................................................... 22
4.4 Selection Comparator .................................................... 23
4.5 Output Buffer .......................................................... 24
4.6 Data Selector ........................................................... 26
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>FPGA Architecture Overview</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Configurable Logic Block</td>
<td>6</td>
</tr>
<tr>
<td>2.3</td>
<td>MySQL’s Pluggable Storage Engine Architecture</td>
<td>8</td>
</tr>
<tr>
<td>2.4</td>
<td>Architecture Overview of the Traditional Database System</td>
<td>9</td>
</tr>
<tr>
<td>2.5</td>
<td>Architecture Overview of the Hybrid Database System</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Slotted Pages in Traditional DBMS</td>
<td>11</td>
</tr>
<tr>
<td>3.2</td>
<td>Example of Database Table Schema Translation</td>
<td>14</td>
</tr>
<tr>
<td>3.3</td>
<td>Data Content in the Plain Data Format</td>
<td>16</td>
</tr>
<tr>
<td>4.1</td>
<td>FPGA-based Storage Engine Architecture</td>
<td>18</td>
</tr>
<tr>
<td>4.2</td>
<td>Database Storage Engine Domain in SATA protocol layers</td>
<td>19</td>
</tr>
<tr>
<td>4.3</td>
<td>Data Collector</td>
<td>20</td>
</tr>
<tr>
<td>4.4</td>
<td>Data Parser</td>
<td>21</td>
</tr>
<tr>
<td>4.5</td>
<td>DFA Implementation of the Data Parser</td>
<td>21</td>
</tr>
<tr>
<td>4.6</td>
<td>NFA Implementation of the Data Parser</td>
<td>21</td>
</tr>
<tr>
<td>4.7</td>
<td>Data Projector</td>
<td>22</td>
</tr>
<tr>
<td>4.8</td>
<td>Selection Comparator</td>
<td>23</td>
</tr>
<tr>
<td>4.9</td>
<td>Data Output Buffer</td>
<td>25</td>
</tr>
<tr>
<td>4.10</td>
<td>Data Selector</td>
<td>26</td>
</tr>
<tr>
<td>4.11</td>
<td>Data Reader</td>
<td>27</td>
</tr>
<tr>
<td>4.12</td>
<td>Projection Vector</td>
<td>28</td>
</tr>
<tr>
<td>4.13</td>
<td>Selection Vector</td>
<td>29</td>
</tr>
<tr>
<td>4.14</td>
<td>Selection Conditions RAM</td>
<td>30</td>
</tr>
<tr>
<td>4.15</td>
<td>Selection Values RAM</td>
<td>31</td>
</tr>
<tr>
<td>4.16</td>
<td>Boolean Expressions and Selection Criterions as a Binary Tree</td>
<td>32</td>
</tr>
<tr>
<td>4.17</td>
<td>Boolean Expression Evaluation Circuits</td>
<td>32</td>
</tr>
<tr>
<td>4.18</td>
<td>Selection Truth Table</td>
<td>33</td>
</tr>
<tr>
<td>5.1</td>
<td>Data Content in Structured Data Format</td>
<td>37</td>
</tr>
<tr>
<td>5.2</td>
<td>Data Parser for Structured Data Format</td>
<td>38</td>
</tr>
<tr>
<td>6.1</td>
<td>Database Storage Engine in PCIe protocol layers</td>
<td>42</td>
</tr>
<tr>
<td>6.2</td>
<td>The General View of a TLP’s Structure</td>
<td>43</td>
</tr>
<tr>
<td>6.3</td>
<td>The First DW of a TLP Header</td>
<td>43</td>
</tr>
<tr>
<td>6.4</td>
<td>Example of Memory Write Request TLP</td>
<td>44</td>
</tr>
<tr>
<td>6.5</td>
<td>Example of Memory Read Request TLP</td>
<td>44</td>
</tr>
<tr>
<td>6.6</td>
<td>Example of Memory Read Completion TLP</td>
<td>45</td>
</tr>
<tr>
<td>6.7</td>
<td>PCIe Engine Architecture</td>
<td>46</td>
</tr>
</tbody>
</table>
List of Tables

3.1 Example Table Persons .............................................. 13
3.2 Definition of Markup Operators .................................. 13
3.3 Data Types and Sizes ................................................ 14

4.1 Processing Status Tags in Status RAM ............................ 25
4.2 Tag Definition of Comparison Operators ......................... 30

6.1 TLP Header Type and Format Field Encodings .................. 43

7.1 Storage Engine Resource Utilizations ............................. 50
7.2 PCIe Engine Resource Utilizations ................................. 50
7.3 Storage Engine Performance ........................................ 50
7.4 Evaluation Environment ............................................. 51
7.5 PCIe Engine Performance ........................................... 51
List of Listings

3.1 Schema of Example Table Persons .......................... 13
4.1 Pseudo Code for Data Parser .................................. 22
4.2 Pseudo Code for Data Projector ................................ 23
4.3 Pseudo Code for Selection Comparator ......................... 24
4.4 Pseudo Code for Data Selector ................................. 26
4.5 Pseudo Code for Data Reader ................................... 27
4.6 Example Query Q1 on Table Persons ............................ 28
5.1 Pseudo Code for Optimized Data Parser ....................... 38
Chapter 1

Introduction

Ever increasing database sizes and query complexity presents an opportunity for hybrid, FPGA-based co-processor systems to significantly improve database performance. Although performance opportunities for FPGAs in fast data processing field are well-known, the design of a FPGA-based hybrid system still need for compromise between system flexibility and high FPGA-compilation cost. Elimination of this compilation cost would dramatically increase the efficiency of FPGA-based hybrid systems. An FPGA-based “smart” database storage engine is the focus of this thesis. We present a implementation of FPGA-based smart database storage engine in a hybrid database system, which supports on-the-fly selection and projection based filtering and at-runtime reconfigurations with high throughput and low latency.

1.1 Background and Motivation

In recent years, researchers have suggested using programmable hardware, i.e. field-programmable gate arrays (FPGAs), to accelerate the database processing [1]. The idea of using dedicated hardware as a coprocessor to accelerate database tasks can be traced back to the end of 1970s, David J. DeWitt built a multiprocessor organization DIRECT, a so-called database machine based on a MIMD architecture for supporting DBMS [2]. However, because the performance improved unremarkable compared to systems using more powerful (higher clock-frequency) CPUs, and the costs to build dedicated database hardware were gigantic in that age; DIRECT project was discontinued at the end.

Unlike the challenge we met decades ago, the clock-frequency of general purpose CPUs cannot be increased easily any more because of the power wall\(^1\). The approach that

\(^1\)Power grows as \textit{Frequency}^3, thus the CPU power grows much faster than the performance gained by the higher frequency.
increasing the database performance by using higher frequency CPUs becomes infeasible today. Therefore, alternative solutions need to be explored.

Recently, in the database research field, a lot of new database architectures have been proposed. The trends can be separated into two groups:

- **new software architectures**: MapReduce\cite{3} and Hadoop Framework\cite{4}, NoSQL Databases (e.g. MongoDB\cite{5}), Column-store databases (e.g. MonetDB\cite{6}), Databases in Memories (e.g. SAP HANA\cite{7}, MemSQL\cite{8}), and so on.

- **new hardware architectures**: FPGA coprocessors acceleration\cite{9, 10}, Graphics processors acceleration\cite{11–13}, Multicore processors optimization and acceleration\cite{14}, Network processors acceleration\cite{15}.

As we can see in the *new hardware architectures* group from the above, people have already tried new hardware other than general purpose CPUs to improve the database performance. One of the most interesting tracks is the FPGA coprocessors acceleration. We try to go deep into this track to explore more possibilities in the hardware acceleration field.

### 1.2 Problem Statement

The main task of this thesis is to build an FPGA-based database storage engine that supports selection and projection based filtering. In particular, rather than only transferring data from the physical disk devices to the database server, the FPGA-based database storage engine get commands (selection and projection requirements) from the database server; then the FPGA-based storage engine executes as a disk controller as well as a data filter which is able to recognize and retrieve structures of data that is is transmitting. The interface between the DB server and the FPGA part of storage engine should be explored; the interface should allow the storage engine to push certain conditions of the query plan down to the FPGA which is used during selection and projection based filtering, and should be expandable for pushing down more functionalities in the future.

At the other end, an analysis how data is best organized on the disks should be concluded. The throughput of the FPGA-based storage engine and the latency introduced by the FPGA-based engine should be measured, other factors and relevant metrics, i.e. the FPGA resource consumptions, the circuits speed should be discussed.
1.3 Contribution

The main contribution of this thesis is the design of the FPGA-based smart database storage engine, which supports arbitrary selection and projection based filtering queries and at-runtime reconfigurations. The storage engine is able to accept dynamic selection and projection based filtering queries at runtime without any FPGA reconstruction processes. Two data storage formats we introduced in the thesis, the \textit{plain data format} and the \textit{structured data format}, give FPGAs the capability to recognize data it is transferring and to filter the data on-the-fly. Another achievement is that the design and implementation of a PCI Express engine on the FPGA. By using our PCIe engine in the FPGA co-processing system, we build a high throughput and low latency data channel between the FPGA and the host computer.

1.4 Outline of the Thesis

The rest of this thesis is organized as follows:

- **Chapter 2**: We provide a short introduction to the FPGA and the database storage engine. In particular, we show the FPGA internals, and analyze the MySQL’s storage engine architecture. Then we discuss the position of an FPGA-based database storage engine in the data path of a hybrid architecture system.

- **Chapter 3**: We give a short introduction about how the traditional database manages data on the disks, and we analyze that what the data format should be for an FPGA-based database storage engine. Then we define an FPGA-readable storage format – the plain data format.

- **Chapter 4**: We design a new storage engine system that has capabilities to handle arbitrary selection and projection based filtering queries without any FPGA reconstruction processes. We first describe the implementation of the storage engine by going through the data flow path. And then we show that how we do the reconfigurable filtering without any reconstruction processes.

- **Chapter 5**: We analyze the performance of the storage engine, then we optimize the storage data format and the data parser of the storage engine to get higher performance.

- **Chapter 6**: We propose using PCI Express as the interface between the FPGA and the DB server. First, we give a short introduction of the PCI Express protocol.
Then we show all details concerning the implementation of our PCIe engine on the FPGA.

- **Chapter 7**: We evaluate the performance of the storage engine and the performance of the PCIe engine; and discuss the circuits speed and the FPGA resource utilizations (i.e. RAMs, Flip-Flops, LUTs).

- Finally, the conclusion and potential future work are presented in **Chapter 8**.
Chapter 2

Fundamentals

This chapter provides a short introduction to FPGAs in Section 2.1, an overview of the database storage engine and MySQL Pluggable Storage Engine Architecture in Section 2.2, and an overview of the hybrid database system with an the FPGA-based storage engine in Section 2.3.

2.1 FPGA

An FPGA (field-programmable gate array) is a chip designed to be programmed and reprogrammed by customers after manufacturing. Customized designs can be made after the chip fabrication. FPGAs are the most powerful programmable logic devices today\(^1\). As we use Xilinx Virtex-5 Evaluation Board XUPV5\(^2\) and NetFPGA-10G\(^3\) to implement the hardware part of the database storage engine, the following of this section will explain FPGA Internals based on Xilinx FPGAs.

2.1.1 FPGA Internals

In essence, an FPGA, which is shown in Figure 2.1, is an integrated circuit that has a number of CLBs (Configurable Logic Block) as the basic building block. Compared with ASICs (Application Specific Integrated Circuits), the Boolean functions we design are not implemented by interconnecting dedicated logic gates (i.e. AND, OR, NOR, XOR gates) in FPGAs, but by configuring the LUTs (Lookup Tables) in CLBs.

\(^1\)There are other kinds of programmable chips, i.e. PLA, PLD, CPLD in programmable logic devices market.


\(^3\)NetFPGA is also based on Xilinx Virtex-5 FPGA chips, http://netfpga.org/
The LUT is basically a RAM-based N-input/1-output truth table, an arbitrary logic gate can be implemented by LUTs. As shown in Figure 2.2, there are 4 4-input/1-output LUTs in one CLB. Moreover, there are also D-flipflops at the end of each LUT output for implementing sequential circuits. By using HDL (hardware description language) to describe a circuit design, a bitstream file, which contains LUTs configuration information, is generated from the FPGA Design Tools; in a later stage, this bitstream file is loaded into FPGA chip (program stage), LUTs are configured according to the circuit design; then the FPGA is ready to work as the specified circuit.
2.1.2 Potential and Limitations

Potential

One of the most significant advantages of FPGAs is the re-programmable characteristic, it means that we are able to design dedicated “optimal” circuits for each application. Parallelism is also an advantage of the FPGA technology; resources (i.e. registers, LUTs) in the FPGA chip can be accessed in parallel, this characteristic can let us avoid the memory wall\(^4\) in a traditional von Neumann architecture, therefore FPGAs are perfect for applications like stream data processing\(^9\), frequent item detection\(^16\). Moreover, as FPGAs operate at a much lower clock speed than general purpose CPUs, it comes that FPGAs usually consume less power.

Limitations

Because resources, such as RAMs and LUTs, are limited in FPGA chips, the FPGA is not suitable for all applications. For example, machine learning and data mining which might require a large memory space is not suitable for FPGAs. Additionally, the design process of FPGAs is not the same of software development; extra efforts are needed, e.g. timing constraints analysis, post-place & route simulation. FPGA development is generally more time-consuming than software development.

2.2 Database Storage Engine

A database storage engine is one component of the DBMS, which is used to read, write, delete, update data from the database. Usually, the database storage engine is software running on DBMS servers. One DBMS can have several different database storage engines; each engine can have a different purpose, a different implementation; each engine has its own capability and limitation. By choosing a proper database storage engine under database sizes, application scenarios and other factors (e.g. high frequency queries), the performance of the database system can be improved. Moreover, some more advanced features like transaction can give the database application more functionalities.

MySQL’s Pluggable Storage Engine Architecture

As we can see from Figure 2.3, by providing a set of standard and consistent APIs between the storage engine and other parts of the DB server, the architecture of MySQL isolates all low-level details at the physical storage level from database users and DBAs. Moreover, as of MySQL 5.1, MySQL Server enables storage engines to be inserted into and unplugged

---

\(^4\)The bandwidth bottleneck between CPUs and main memories.
from a running MySQL server. This pluggable characteristic makes the MySQL Server more flexible and efficient to deal with different application scenarios. It also let our storage engine can be integrated into the MySQL Server easily.

Figure 2.3: MySQL’s Pluggable Storage Engine Architecture (Source: MySQL)

2.3 System Overview

In a traditional architecture, as we can see in Figure 2.4, all components of the DB system including the storage engine are integrated into the DB server(s); the database storage engine exploits the computing resources (i.e. CPUs, memory) of the DB server together with other components (i.e. the query execution, the query optimization) in a time sharing manner. The storage engine on the DB server is in charge of reading data from and writing data to disk(s). There are two potential performance bottlenecks of this architecture described in the following:

1. As the database storage engine is a software component which shares the computing resources with the query executor and optimizer; queries execution time can be affected by tasks of the storage engine like reading and writing data.

2. In Figure 2.4, the width of the grey arrow in the data path indicates the volume of the data. The large data volume can slow down certain volume-intensive operations.

However, in the hybrid architecture, as shown in Figure 2.5, the FPGA-based storage engine is inserted into the system’s data path, the FPGA is operating as a filter which
is located close to the data source. The tasks of the FPGA-based storage engine are the following:

- 1. Filters data as much as possible at the very beginning step according to queries.
- 2. Sends only essential data back to the DB server.

And the FPGA can also perform as a co-processor to offload part of query execution tasks from the general purpose CPUs. The general purpose CPUs perform other more complex operations in a later stage.
Chapter 3

Storage Data Format Design

3.1 Data Format on Disks

The storage engine is part of the hybrid architecture as shown in Figure 2.5. The DB server is no longer in charge of reading/writing data from/to persistent storage devices, this part has been moved to the FPGA. FPGAs must have the capability to directly access hard disks. As there is no file system, how to store and organize data directly on disk is a problem to solve, and the ability of the FPGA to understand the format of disks is also an important factor to consider when building the engine.

Traditionally, the DBMS manages space on disk via disk space manager. The disk space manager hides low level details of the hardware, and gives an abstraction of a piece of data – page. The software at higher levels of the DBMS are able to treat data as a collection of pages. The data of a table is stored in a slotted page with a slot directory, as shown in Figure 3.1.
The slot directory at the end of one page is used to handle variable length records (e.g. Varchar columns in a record). And the size of one page usually is a multiple of the size of a disk block, e.g. 16KB.

This page format fits well in traditional disk -- main memory -- CPU architectures. The DBMS has enough memory to read one page or many pages into the main memory, and the CPU can easily parse data in a page in a straightforward way. However, slotted pages are not suitable for FPGAs. First, the memory resources in FPGAs are not sufficient compared to the main memory in a workstation. Second, the circuits to parse slotted pages format are very complicated to implement, there are many challenges, such as pointer chasing on the bare metal etc.

As we can see from previous works [9, 16], FPGAs can handle and perform stream data very well. We believe that raw data in the stream mode is a good way for filtering in FPGAs. In the following subsection, we introduce the plain data format, a data format that supports stream processing and data storage on raw disks.

### 3.2 Plain Data Format

The plain data format is a light-weight markup structure that defines a set of rules for encoding a database table in this format. This is the first version data format we used in the thesis, however this format has a potential performance bottleneck that is discussed in Section 5.1, and we will refine it to an advanced version data format in Section 5.2.

There are three essential parts to describe a database table in the plain data format.

- **Markup operator definition**: the markup operator definition defines special characters as control symbols used in the plain data format.

- **Database table schema translation**: the database table schema translation is a definition table translated from the database table schema, which contains information about the number of columns in the table, data types of each column in the table, the size of each data type.

- **Data content**: the database table presented in the plain data format.

In the following of this section, the plain data format will be introduced with a simple example. First, we create a table `Persons` using the following schema in Listing 3.1. And then we insert several test tuples into the table `Persons`. The table `Persons` after insertions is shown in Table 3.1.
SCHEMA 1:
CREATE TABLE Persons
(
    P_Id int ,
    FirstName varchar(255),
    LastName varchar(255),
    Age int ,
    Sex char(1),
    Salary int
);

Listing 3.1: Schema of Example Table Persons

| +---------+-------------+----------+-----+-----+--------+
| | P_id | FirstName | LastName | Age | Sex | Salary |
| +---------+-------------+----------+-----+-----+--------+
| | 1000 | Jack | HILL | 26 | M | 6600 |
| | 1001 | Nate | SMITH | 28 | M | 7500 |
| | 1002 | Jenny | JONES | 25 | F | 7200 |
| | 1003 | Tom | TAYLOR | 24 | M | 6000 |
| | 1004 | Olivier | JACKSON | 30 | F | 8000 |
| +---------+-------------+----------+-----+-----+--------+

Table 3.1: Example Table Persons

3.2.1 Markup Operator Definition

To present the above Persons table in the plain data format, we use the markup operator definition that is shown in table 3.2. The operator can be any other characters and values with customized definitions. However, for the purpose of simplicity, common visible ascii characters are used in the example.

<table>
<thead>
<tr>
<th>Name</th>
<th>Character</th>
<th>Binary Value</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Record begin operator</td>
<td>+</td>
<td>0x2B</td>
<td>indicates the start of one record in the table</td>
</tr>
<tr>
<td>Record end operator</td>
<td>-</td>
<td>0x2D</td>
<td>indicates the end of one record in the table</td>
</tr>
<tr>
<td>Column separator</td>
<td></td>
<td>0x7C</td>
<td>indicates a separate between columns</td>
</tr>
</tbody>
</table>

Table 3.2: Definition of Markup Operators

3.2.2 Database Table Schema Translation

From the schema of the table Persons, we are able to get the database table schema translation that is shown as follows:
Figure 3.2: Example of Database Table Schema Translation

Figure 3.2 shows the translated database table schema in the binary format. The first two bytes are a number that shows how many columns are in this table; in our example it is 6. Then in the following 6 bytes, the schema translation contains information about data types and the size of each column. For example, the data type tag of column 5, which is column Salary in our example, is 0x1 (integer), as shown in the first half of the byte (4 bits); and the size of this column is 4 bytes, which is shown in the latter half of the byte (4 bits). Details of data types and sizes are shown in Table 3.3 as follows, this table is scalable to support more data types with customized definitions.

<table>
<thead>
<tr>
<th>Data type</th>
<th>Tag</th>
<th>Size</th>
<th>Mode</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Varchar</td>
<td>0x0</td>
<td>0</td>
<td>ASCII</td>
<td>Varchar’s length is variable, the column of Varchar only ends with the record end operator - or the column separator</td>
</tr>
<tr>
<td>Integer</td>
<td>0x1</td>
<td>4</td>
<td>Binary</td>
<td>Integer, a fixed size 4 bytes</td>
</tr>
<tr>
<td>Char</td>
<td>0x2</td>
<td>N</td>
<td>ASCII</td>
<td>Char, a fixed size N bytes from table schema</td>
</tr>
<tr>
<td>Long</td>
<td>0x3</td>
<td>8</td>
<td>Binary</td>
<td>Long, a fixed size 8 bytes</td>
</tr>
</tbody>
</table>

Table 3.3: Data Types and Sizes

The size of database table schema translation depends on the maximum number of columns $N_c$ in a table, it would consume $N_c/2 + 1$ bytes. For example, the schema translation of a table with 30 columns use 16 bytes. As the size is relatively small, it is reasonable to store this table in the distributed rams of the FPGA. And this ram is connected to the data parser, one component of our storage engine, which is shown in Chapter 4. We will present how the data parser processes the database table schema translation in Section 4.2.
3.2.3 Data Content

So far, we have defined the markup operators, and have gotten the database table schema translation. By applying the following rules, the example table Persons (Table 3.1) can be converted into the plain data format, as shown in Figure 3.3.

1. Each tuple starts with a record begin operator +.
2. Each tuple ends with a record end operator -. If the current column is the last column of this tuple, then append a - at the end of this column.
3. Each column in one tuple is separated by a column separator |. If the current column is not the last column of this tuple, then append a | before the start of the next column.
4. If the size of the current column’s data type is a fixed number \( N \), then reserve the following \( N \) bytes for this column.
5. If the size of the current column’s data type is unknown, e.g. Varchar, then fill this column byte by byte.
6. The table ends at a record end operator - without a further incoming record begin operator + at the next byte.
Chapter 3. Storage Data Format Design

<table>
<thead>
<tr>
<th>P_id</th>
<th>FirstName</th>
<th>LastName</th>
<th>Age</th>
<th>Sex</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Jack</td>
<td>HILL</td>
<td>26</td>
<td>M</td>
<td>6600</td>
</tr>
<tr>
<td>1001</td>
<td>Nate</td>
<td>SMITH</td>
<td>28</td>
<td>M</td>
<td>7500</td>
</tr>
</tbody>
</table>

Figure 3.3: Data Content in the Plain Data Format
Chapter 4

Implementing Storage Engine in FPGAs

There are some previous works that are able to filter data and execute queries for the DBMS on FPGAs [10, 17, 18]. In [10, 18], queries are pre-compiled to FPGAs\(^1\), the pre-compiled processes as known as Synthesis and Place-and-Route(PAR) are very time-consuming. It could cost several minutes to compile a FPGA design, sometimes even up to hours if implementing a “big” design on a “big” FPGA chip.

There is no problem if we use a dedicated circuit as a co-processing accelerator to accelerate one special query or some certain frequent queries. Because the co-processing accelerator is deployed into the system before running, once the deployment is done, the FPGA doesn’t need to be re-compiled again at run-time. However, in our design, the query requests can be changed at run-time, the pre-compilation cost in the above approach is unacceptable.

In [17], by using a partially reconfigurable module library, researchers gave a partial re-configuration based approach that which can possibly reduce and overcome the pre-compilation cost. However, they did not give the detailed reconfiguration times in the paper.

The goal of our system is a general purpose storage engine, which can accept arbitrary selection and projection based filtering. In order to overcome the pre-compilation cost in the above approaches, we design a new system that has capabilities to handle arbitrary selection and projection queries without any pre-compilation processes. The system consists of several basic blocks, the architecture is shown in Figure 4.1.

\(^1\)“pre-compiled”/“re-compiled” here means that for each query, a dedicated circuit either generated from compilers or from designers need to be loaded to FPGAs before executing the query.
The following of this chapter describes the implementation of the FPGA-based storage engine in detail. In the first part of the following sections (Section 4.1 to Section 4.7), we describe the implementation of the storage engine by going through the data flow path. Then in the second part of the following sections (Section 4.8), we show how we do the reconfigurable filtering without any pre-compilation processes. Here, we divide the data path into two separated paths from the data parser; this is because that the projection process could be much faster than the selection evaluation process. By dividing the data path into two separated and independent paths, we can handle these two processes in parallel, this reduces the latency of the entire system.
4.1 Data Collector

As there is no operating system and no file system running on the FPGA, all basic storage engine functions such as read and write disk sectors are directly sent to a HBA (Host Bus Adapter). In this thesis, we use Groundhog [19], an open-source SATA HBA (Host Bus Adapter) for FPGAs, to do data transfer between SATA-based hard disks and FPGAs. SATA is a computer bus interface for accessing mass storage devices. Groundhog, the SATA HBA, offers classical read and write commands of the SATA protocol. By exploiting special I/O transceivers (GTP/GTX) on the Xilinx FPGAs, Groundhog hides all technical details from command Layer to physical layer in the SATA protocol, as shown in Figure 4.2.

The data collector, as shown in Figure 4.3, is one module that we designed in the storage engine as a wrapper of the SATA HBA. The data collector further reduces the complexity of communications with SATA devices. Also, the storage engine can be reasonable modular with a focus on functionalities by using such a wrapper. The data collector exposes a simple data stream $\text{Din}, \text{DinVld} \rightarrow \text{Dout}, \text{DoutVld}$ interface to the storage engine, the storage engine can read data from or write data to SATA devices by setting the following signals of the wrapper:

- $\text{cmd}$: read or write command codes\(^2\).
- $\text{lba}$: logical block addressing, the address of disk where $\text{cmd}$ starts.
- $\text{sectorcnt}$: the number of sectors affected with this command.
- $\text{cmd_en}$: the valid signal of $\text{cmd}$, $\text{lba}$ and $\text{sectorcnt}$.

\(^2\)Documentation for Groundhog can be found here: http://groundhog.codeplex.com/
4.2 Data Parser

So far, we have defined the data layout on physical disk in Chapter 3, and we have the storage engine that is able to do basic read and write operations with SATA-based devices. Data parsing on FPGAs is the next task. Like most of data parsing and data filtering tasks, the parsing process of our plain data can be expressed using a finite-state machine (FSM). This approach has been proved to perform very good on FPGAs [16].

The data parser, as shown in Figure 4.4, is the second module in the storage engine’s data path right after the data collector, there are 4 main functions in the Data Parser:

- 1. By exploiting the database table schema translation (see Section 3.2.2), it parses each record into columns.
- 2. By checking the query information stored in the selection vector (see Section 4.8.3), it filters all data that is related to selection constraints for further processing.
- 3. It dispatches selection-related data to the selection comparator (see Section 4.4).
- 4. It forwards all parsed data to the data projector (see Section 4.3) to do further operations.

The data parser can be complied into either a deterministic finite-state machine (DFA), as shown in Figure 4.5 or a non-deterministic finite-state machine (NFA), as shown Figure 4.6. For the deterministic finite-state machine, there is only one single state can be active.
at any given time. This characteristic fits well in the software technology. However, for a non-deterministic finite-state machine, all states can be active in parallel in hardware. As the inherent parallelism characteristic of FPGAs, the NFA would be a better choice to implement the data parser on FPGAs. Also the NFA may use less states, which means less resources on the FPGA, to implement a FSM with the same functionalities. So we use the NFA approach to implement the data parser.

The algorithm in Listing 4.1 summarizes in pseudo code the behavior of the data parser.
WHILE data_vld = 1 DO
  IF data = '+' THEN
    column_cnt = 0;
    record_start = '1';
  ELSIF data = '|' THEN
    column_cnt = column_cnt + 1;
  ELSIF data = '-' THEN
    column_cnt = 0;
    record_end = '1';
  END IF;
  IF SelectionVector[column_cnt] = '1' THEN
    selection_values = data;
    selection_vld = '1';
  END IF;
END;

LISTING 4.1: Pseudo Code for Data Parser

4.3 Data Projector

The data projector, as shown in Figure 4.7, is a module connected right after the Data Parser in the data path. It projects columns of the records in a database table by using the projection vector (see Section 4.8.2) and the data parsing information from the data parser, and then sends all projected columns to the data output buffer.

The algorithm in Listing 4.2 summarizes in pseudo code the behavior of the data projector. The data projector only outputs the columns which have been set to 1 in the projection vector.
IF (Dout_en = 1) THEN
    data_rdy = 1;
    WHILE data_vld = 1 DO
        IF (ProjectionVector[column_cnt] = 1) THEN
            Dout = data[7:0];
            DoutVld = 1;
        ELSE
            Dout = 0;
            DoutVld = 0;
        END IF;
    END;
END IF;

Listing 4.2: Pseudo Code for Data Projector

4.4 Selection Comparator

The selection comparator is a module that evaluates all comparison operations of the selection criterions, as shown in Figure 4.8. The selection comparator gets all selection-constraints information from the selection vector, the selection conditions RAM and the selection Values RAM (see Section 4.8). It starts to evaluate each individual selection criterion from the start of one record, and it updates the evaluation result vector after one comparison is done. The selection comparator outputs the final result vector (Selection Evaluation Result Vector) at the end of each record. The algorithm in Listing 4.3 summarizes in pseudo code the behavior of the selection comparator.

The integer shift register is a 4 bytes register to store one integer column from the data parser. It need 4 clock cycles to get a integer for the comparison. The comparator in the module can be customized to support more comparison operators other than the standard
logic comparison operators. In the current stage, we only support integer comparisons in the selection criterions. However, it is possible to support more data types by adding corresponding logics in the selection comparator.

### 4.5 Output Buffer

There are two RAMs in the output buffer module, Status RAM and Record Buffer RAM. The projection process in the storage engine is straightforward, there is no extra latency clock cycles introduced by the projection process. However, the comparison process in the selection comparator could be more complex than the projection process. For example, the evaluation such as $1 \leq a < 2$ may introduce extra latency. Also after all comparison operations, we still need evaluate the final result by checking the selection truth table (see Section 4.8.5) in the data selector.

So the projection process could be much faster than the evaluation process of the selection criterions, the records after projection have to be stored into buffers before getting the evaluation result from the data selector. The Status RAM stores the processing status tag of each record after projection and the start address of each record in Record Buffer
RAM. The Record Buffer RAM stores the data content of projected columns of each record. The definition of the processing status tags is shown in Table 4.1.

<table>
<thead>
<tr>
<th>Tag (2 bits)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>the selection evaluation result is pending, waiting the final evaluation result from the data selector.</td>
</tr>
<tr>
<td>10</td>
<td>the selection evaluation result is true, the corresponding record should be sent to the DB server.</td>
</tr>
<tr>
<td>11</td>
<td>the selection evaluation result is false, the corresponding record should be discarded.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Tag (2 bits)</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>01</td>
<td>the selection evaluation result is pending, waiting the final evaluation result from the data selector.</td>
</tr>
<tr>
<td>10</td>
<td>the selection evaluation result is true, the corresponding record should be sent to the DB server.</td>
</tr>
<tr>
<td>11</td>
<td>the selection evaluation result is false, the corresponding record should be discarded.</td>
</tr>
</tbody>
</table>

**Table 4.1: Processing Status Tags in Status RAM**

For example, Figure 4.9 shows that Record 1 has passed the selection evaluation, and Record 2 and 3’s results are still pending. All data content of the projected records (Record 1, 2 and 3) are stored in the record buffer RAM.

![Figure 4.9: Data Output Buffer](image)

The size of the status RAM is relatively small, it could be implemented by a distributed RAM in the FPGA. The size of the record buffer RAM depends on the size of projected records, and the buffer depth required by the system. To get a reasonable capacity, it is implemented by block RAMs.
4.6 Data Selector

The data selector is one module to evaluate whether the record satisfies the selection criterions in \textbf{WHERE} clause, as shown in Figure 4.10. As shown in Section 4.8.5, the selection truth table is generated according to the selection criterions. The data selector checks every selection evaluation result with the selection truth table, and then modifies status tags in the output buffer according to the selection truth table’s output.

The algorithm in Listing 4.4 summarizes in pseudo code the behavior of the data selector.

```
1  status_tag = '00';
2  status_tag_vld = 0;
3  WHILE Eva_vld = 1 DO
4      IF SelectionTruthTable[Eva_result_vec] = 1 THEN
5          status_tag = '10'; \ true
6          status_tag_vld = 1;
7          status_tag_addr = status_tag_addr + 1;
8      ELSE
9          status_tag = '11'; \ false
10         status_tag_vld = 1;
11         status_tag_addr = status_tag_addr + 1;
12      END IF;
13  END;
```

LISTING 4.4: Pseudo Code for Data Selector

4.7 Data Reader

The data reader, as show in Figure 4.11, is the last module in the data path of the database storage engine. It reads data from the output buffer according to the processing tags in the status RAM of the output buffer. The data reader is also in charge of providing the
simple stream data interface to further modules, i.e. the Microsoft SIRC framework or the PCIe engine (see Chapter 6).

![Data Reader Diagram](image)

**Figure 4.11: Data Reader**

The algorithm in Listing 4.5 summarizes in pseudo code the behavior of the data reader.

```
WHILE status_tag_vld = 1 DO
  IF status_tag = '10' THEN // true
    IF record_addr != record_addr_next THEN
      Dout = RecordBufferRAM[record_addr];
      Dout_Vld = '1';
      record_addr = record_addr + 1;
    END IF;
  ELSIF status_tag = '11' THEN // false
    Dout <= 0;
    Dout_Vld <= '0';
    record_addr <= record_addr_next;
  ELSE
    // wait
  END IF;
END;
```

**Listing 4.5: Pseudo Code for Data Reader**

### 4.8 Reconfigurable Data Filtering

#### 4.8.1 Query Translation

The query translator is a software component running on the DB server. It is in charge of translating SQL queries (selection and projection related information) to an FPGA-readable format. In the following, we are going to explain how query translator works via an example query. The example query $Q_1$ is shown in Listing 4.6.
Q1:
SELECT FirstName, LastName, Age, Salary
FROM Persons
WHERE P_id < 1003 AND (Age < 26 OR Salary > 7500);

Listing 4.6: Example Query Q1 on Table Persons

The query translator extracts FPGA-readable parameters from the queries, and then re-configures\(^3\) the FPGA-based storage engine to do selection and projection based filtering. In the following subsections, we give more details on how the query translator converts the query Q1 as shown in Listing 4.6 to FPGA-readable parameters.

### 4.8.2 Projection Vector

The projection vector is expressed as a K-bit register on the FPGA. This register contains the index information of projected columns. K is a configurable parameter that presents the width of register, K should be kept larger than or equal to the number of columns of the queried table.

For example, in our example, the table Persons has 6 columns. We need to reserve at least 6 bits of the register for the projection vector. Here, we choose a 16-bit register as the projection vector. And according to the query in Listing 4.6, Columns FirstName, LastName, Age and Salary are 4 columns after projection. So we get the projection vector in Figure 4.12.

The bit indices of the projection vector correspond to indices of columns in the Persons table. The bit is set to 1 if the corresponding column is required to project in the query. Here, columns FirstName(1st\(^4\)), LastName(2nd), Age(3rd) and Salary(5th) are projected.

\(^3\)FPGA blurs the distinction between (re)compile and (re)configure the FPGAs. In this thesis, if we load our design to FPGAs, we refer to this as (re)compile; and if we only change parameter at runtime, we refer to this as (re)configure.

\(^4\)The index starts from 0.
Therefore, the bits 1, 2, 3 and 5 are set to 1, and all other bits keep 0. The discussion about the allocation method of the projection vector is shown in the next subsection together with the selection vector.

### 4.8.3 Selection Vector

Selection Vector is also a \( K \)-bit register on the FPGA. Similarly, this register contains the index information of columns which have been applied with selection constraints. \( K \) is a configurable parameter that presents the width of register, \( K \) should be kept larger than or equal to the number of columns of the queried table.

In our example, at least 6 bits need to be allocated as the selection vector. From the query \( Q1 \) as shown in Listing 4.6, columns \( P\_id \), \( Age \) and \( Salary \) need to fulfill a specified criterion if we extract this record. The converted selection vector is in Figure 4.13.

![Figure 4.13: Selection Vector](image)

Columns \( P\_id \) (0th), \( Age \) (3rd) and \( Salary \) (5th) are selected in the \texttt{WHERE} clause. The bits with value 1 in the selection vector present the indices of columns which have been applied with selection constraints in a straightforward way.

The size of the projection vector and the selection vector is not dynamically allocated per query basis at run-time. It need to be determined before the deployment of the system according the required capacity. If the storage engine need to handle the tables that could have up to 20 columns, then two 20 bits registers are needed for the selection and projection vector. As the size of these two vector are very small, we use registers on the FPGA to implement them, and it can easily scale up to even hundreds or thousands bits. However, the other blocks co-working with there two vector may not scale up correspondingly. The detailed discussion is in the following subsections.

### 4.8.4 Selection Conditions and Values RAMs

Unlike projection information that can be easily expressed by a single projection vector, selections are much more complex. Besides the selection vector that indicate indices of
columns, we need two more RAMs and a truth table (see Section 4.8.5) to store selection criterions.

1. Selection Conditions RAM

The selection conditions RAM is a $K \times 3$ bits RAM that contains comparison conditions of each selection criterion. $K$ is a configurable parameter that presents the depth of this RAM, and the width of the RAM is 3. $K$ should be kept equal to the width of Selection Vector.

In the following table 4.2, we define comparison operators with 3-bits width binary tags.

<table>
<thead>
<tr>
<th>Comparison Operator</th>
<th>Binary Tag</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>0x0</td>
</tr>
<tr>
<td>&lt;</td>
<td>0x1</td>
</tr>
<tr>
<td>&gt;</td>
<td>0x2</td>
</tr>
<tr>
<td>&lt;=</td>
<td>0x3</td>
</tr>
<tr>
<td>&gt;=</td>
<td>0x4</td>
</tr>
<tr>
<td>&lt;&gt;(!=)</td>
<td>0x5</td>
</tr>
</tbody>
</table>

**Table 4.2: Tag Definition of Comparison Operators**

For example, in our case, these 3 operators <, <, > in $P_{id} < 1003$, $Age < 26$ and $Salary > 7500$ are stored in the selection conditions RAM. The values in the RAM, as shown in Figure 4.14, are valid only if the corresponding bit in the selection vector is set to 1.

![Figure 4.14: Selection Conditions RAM](image)

The size of the selection conditions RAM need to be determined before the deployment too. It is also relatively small. For the storage engine is able to handle up to 20 columns, the size is $20 \times 3$, 60 bits. Both block RAMs and distributed RAMs are a reasonable choice, even registers could be a alternative option. However, if the storage engine need to support hundreds columns in a record, block RAMs should be the first choice.
2. Selection Values RAM

The selection values RAM is a $N \times 4 \times 8$ bits RAM that contains comparison values of each selection criterion. $N$ is a configurable parameter that presents the depth of this RAM, and the width of the RAM is 4 bytes = 4*8 bits (the size of the Integer). $N$ should be kept larger than or equal to the number of selection criterions in the WHERE clause.

In our example, those 3 values 1003, 26, 7500 in $P_id < 1003$, $Age < 26$ and $Salary > 7500$ are stored in the selection values RAM. The values in the RAM, as shown in Figure 4.15, are valid only if the address is smaller than the number of selection criterions.

\[
\begin{array}{|c|c|}
\hline
\text{Address} & \text{Values} \\
0x0 & 1003 \\
0x1 & 26 \\
0x2 & 7500 \\
\vdots & \vdots \\
0x0 & 0 \\
\hline
\end{array}
\]

FIGURE 4.15: Selection Values RAM

The size of the selection values RAM need to be determined before the deployment too. In the current stage, as we only support the integer comparisons in the comparator, the size of this RAM could be kept relatively small. However, if more data types are added in the future, the size of this RAM could expand rapidly. So the selection values RAM should be implemented by block RAMs.

4.8.5 Selection Truth Table

So far, we have converted all necessary information to evaluate Boolean expressions in the WHERE clause. The last problem is how to store Boolean expressions on the FPGA, and how to evaluate them.

Typically, Boolean expressions in the WHERE clause can be expressed as a rooted binary tree, as shown in Figure 4.16. All nodes in the binary tree are two-valued logic with a single output (true or false). External nodes, also known as leaf nodes, evaluate specified selection criterions. Internal nodes and the root node are logical connectives between two external nodes or between the output of an external node and the output of an internal node. The output of the root node shows the final evaluation result of the Boolean expressions in the WHERE clause.

5 Now, only Integer data type is supported as selection criterions
6 For adding the fixed length data types support, it is straightforward. However, for supporting string matching in the varchar columns, extra control logics and string parsers are needed.
The straightforward way to evaluate these Boolean expressions on FPGAs is to design corresponding circuits from the binary tree, as shown in Figure 4.17. All registers in this circuit are driven by the same clock, a fully pipelined structure is used to ensure the speed of circuit.

However, there are two weak points with this method. First, because of the pipelined structure, the latency of evaluation process depends on the depth of the rooted binary tree now, and it can be different with different Boolean expressions. In our example, the evaluation circuit introduces 3 clock cycles latency. In the worst case, the circuit will introduce up to $N-1$ clock cycles latency. $N$ is the number of external nodes in the tree. The main task of the storage engine is transferring data with high throughput and low latency. Introducing non-deterministic high latency goes against the basic concept.
Second, the structure of the tree is likely to change among queries. We cannot predict the structure of the tree and construct the corresponding circuit at runtime.

Therefore we introduce a RAM-based lookup table – the selection truth table to solve this problem. The size of the selection truth table is $2^N$ bits, and $N$ is a configurable parameter that presents the width of the RAM-based lookup table address bus. $N$ should be kept larger than or equal to the number of selection criterions in the `WHERE` clause of the query.

Recall the example query Q1 that we used, as shown in Listing 4.6; there are 3 selection criterions $P_{id} < 1003$, $Age < 26$ and $Salary > 7500$ in `WHERE` clause. So we get a 8 bits RAM-based lookup table with a 3 bits wide address bus, as shown in Figure 4.18. The address width is equal to the number of selection criterions. As all selection criterions are two-valued (true or false) logic, we use one bit in the address bus to show the evaluation result of one selection criterion, $1 = \text{true}$ and $0 = \text{false}$. In Figure 4.18, the bit’s values at the least significant(0th) column in the address bus show the evaluation result of $P_{id} < 1003$; the other bits at other columns are used in the same way.

For example, the address 0x5(3’b101) means that $P_{id} < 1003$ is true, $Age < 26$ is false, and $Salary > 7500$ is true. The value of this address 0x5(3’b101) in the RAM, which is $1 = \text{true}$, presents the final evaluation result of the Boolean expressions.
All values in the selection truth table are reconfigurable at runtime corresponding to the queries, and the cost of the Boolean expressions’ evaluation is reduced from $O(n)$ to 1 clock cycle. By exploiting the selection truth table, it makes the storage engine keep low latency and flexible.

The size of the selection truth table is one of the most important factors to affect the scalabilities of the storage engine, and it is also required before the deployment. For example, if we need to build a storage engine that supports up to 16 selection criterions in a single query, then this truth table will consume $2^{16}$ bits, 64k bits. That would take up 2 36k-bit block RAMs in the Virtex 5 FPGA. However, if we consider 22 selection criterions in a single query, the size would be 4M bits then, this size has already been reach or close to the RAM capacities of one single FPGA chip.

Because its size is $2^n$ bits, the truth table takes the most part of RAM resources used in the entire system, we should use block RAMs firstly on the FPGA to implement the selection truth table. If more selection criteria in one query need to be supported, on-board RAMs (SDR, DDR) other than on-chip RAMs should be considered. Anyhow, 16 ~ 20+ selection criterions in one single query has already met most of needs in the normal use cases.
Chapter 5

Optimizing the Storage Engine System

5.1 Motivation

As discussed in the previous chapter, the data parser processes one byte of the data in the plain data format every clock cycle. The theoretical maximum clock frequency of Xilinx Virtex 5 FPGAs is around 550 MHz [20]. However, in practice the maximum clock frequency of FPGA circuits is affected by many factors, e.g. the longest signal paths between two registers in the pipeline. Usually, the practical maximum clock frequency of Virtex5-based designs is around 180MHz with a fully pipelined structure [9]. As the throughput of the storage engine is proportional to the clock frequency of the FPGA, this means that the maximum throughput of our storage engine is limited to 180MB/s in the current stage.

From the SATA specification, we know that the theoretical maximum bandwidth of the 2nd generation SATA(SATA-II) is 300MB/s; and from the documentation [19] of the SATA HBA we used, the practical maximum read bandwidth of SATA-II based SSDs is around 260MB/s – 280MB/s. This throughput has already exceeded the maximum throughput we could get in the current storage engine.

Moreover, although the performance of our storage engine satisfies the 1Gbps Ethernet based the Microsoft SIRC framework (around 125MB/s), we plan to use the PCI Express protocol as the interface between the FPGA and the DB server (see Chapter 6) to make a tight-coupled system. The throughput of the PCI Express protocol can also exceed the throughput of our storage engine by an order of magnitude (see Appendix B). Therefore,
we need to optimize the plain data format and the data parser to improve the performance and get higher throughput.

5.2 Structured Data Format

In order to parse variable length data types such as Varchar, we used the column separator | to indicate the end of one column in the plain data format. The data parser needs to check whether the incoming byte is a column separator | or not every clock cycle. The check process on a byte-by-byte basis is slowing down the entire data flow.

The structured data format is 4-byte aligned. We abandoned the light-weight markup structure method used in the plain data format. There are no special characters as control symbols in the structured data format. We use a column counter and the database table schema translation to track and parse data. A database table in the structured data format consists of:

- **Database table schema translation**: the database table schema translation is a definition table translated from the database table schema, which contains information about the number of columns in the table, data types of each column in the table, the size of each data type. It consists of the same definitions as in the plain data format (Section 3.2).

- **Data content**: the database table presented in the structured data format.

- **Variable length data type header**: this is a 4-byte control header for the variable length columns which is located just before the data content of the variable length columns in data content. It is explained by an example in Figure 5.1.

The example table Persons (Table 3.1) in the structured data format is shown in Figure 5.1. The Varchar control header, which is marked read in Figure 5.1 consists of two variables, VC0 and VC1. The value of VC0 indicates how many 4-byte blocks the following column occupies.

In our case, SMITH is separated into SMIT and H, it requires two 4-byte blocks, therefore VC0 = 2. The value of VC1 shows how many bytes are used in the last 4-byte block. In our case SMITH, VC1 = 1, only H is stored in the last 4-byte block. By using Varchar control header and the database table schema translation, the data parser is able to handle variable length data types without the help of the a column separator |.
Chapter 5. Optimizing the Storage Engine System

<table>
<thead>
<tr>
<th>P_id</th>
<th>FirstName</th>
<th>LastName</th>
<th>Age</th>
<th>Sex</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1000</td>
<td>Jack</td>
<td>HILL</td>
<td>26</td>
<td>M</td>
<td>6600</td>
</tr>
<tr>
<td>1001</td>
<td>Nate</td>
<td>SMITH</td>
<td>28</td>
<td>M</td>
<td>7500</td>
</tr>
</tbody>
</table>


5.3 Data Parser for Structured Data Format

We change the design of the data parser to suit the structured data format. The new data parser module is shown in Figure 5.2. There are 3 main changes in the new data parser.

- 1. The new data parser has a wider data bus (32 bits) than the previous one (8bits).
- 2. A new 4-bit signal $D_{out, byteEn[3:0]}$ is introduced to do bytes enable control.
- 3. The new data parser abandons the Markup Operator Definition used in the plain format, and uses only the Database table schema translation to parse data.

The algorithm in Listing 5.1 summarizes in pseudo code the behavior of the new data parser.
Chapter 5. Optimizing the Storage Engine System

Figure 5.2: Data Parser for Structured Data Format

Listing 5.1: Pseudo Code for Optimized Data Parser

```plaintext
max_columns = DTST.column_number; // DTST: database table schema translation
column_cnt = 0;
WHILE data_vld = 1 DO
    // the codes for record_start, record_end is bypassed.
    DataType = DTST[column_cnt].DataType;
    Size = DTST[column_cnt].Size;
    IF DataType = 0x0 THEN // VARCHAR
        VC0 = data [31:16];
        VC1 = data [15:0];
        WHILE VC0 >= 0 DO
            VC0 = VC0 - 1;
            Dout = data [31:0];
            Dvld = '1';
            IF VC0 = 0 THEN
                Dout = data [31:0];
                Dvld = '1';
                Dout_byteEn according to VC1;
                END IF;
            END;
        ELSE // fixed length variables
            WHILE Size > 0 DO
                Size = Size - 4;
                Dout = data [31:0];
                Dvld = '1';
                END;
                column_cnt = column_cnt + 1;
            END IF;
            IF SelectionVector[column_cnt] = '1' THEN
                selection_values = data;
                selection_vld = '1';
            END IF;
        END;
    ELSE
        WHILE Size > 0 DO
            Size = Size - 4;
            Dout = data [31:0];
            Dvld = '1';
            END;
            column_cnt = column_cnt + 1;
        END IF;
        IF SelectionVector[column_cnt] = '1' THEN
            selection_values = data;
            selection_vld = '1';
        END IF;
    END;
```
5.4 Performance Analysis

Looking at the data content with the structured data format, the width of input data bus in the data parser can be expanded from 8bits to 32bits. The alignment-guaranteed characteristic makes it possible that every time the data parser is able to read out 4 bytes in one column. Ideally, the throughput of the storage engine will increase by 4 times (around 720MB/s). However, because of the additional logic that is used to parse variable length control headers (e.g. Varchar) have been added between pipeline registers. This increases the longest signal path of circuits, so the maximum clock frequency of the data parser is less than the frequency of the plain-format version parser (see Section 7.3). Moreover, more FPGAs resources are consumed as the wider data path and additional logics to handle variable length control header such as Varchar control header. The specific numbers about the clock frequencies and the FPGA resource consumptions are discussed in Chapter 7.

There is one open issue for the optimized data parser and the structured data format. we cannot do the char and varchar parsing with 4-byte per clock cycle speed in a selection criterion now. The problem of doing the char and varchar parsing with 4-byte per clock cycle speed on FPGAs is beyond the scope of the thesis. That is possible to get the one-byte-per-clock-cycle parsing ability by using additional logics. However, if the logics are parsing char and varchar on a byte-by-byte basis, that would slow down the data flow of the entire system.
Chapter 6

Using PCI Express

As discussed in the previous chapter, we plan to use PCI Express as the interface between the FPGA and the DB server. In this chapter, we first give a short introduction of PCI Express in Section 6.1; then we present more details of the PCI Express protocol at the transaction layer in Section 6.2. All details concerning the implementation of our PCIe engine on the FPGA are covered in Section 6.3.

6.1 Introduction to PCI Express

PCI express (PCIe) is a high-speed serial bus for attaching peripheral devices in a computer. PCIe is widely used in servers and workstations as a motherboard-level interconnect today.

Although it is a serial bus, it operates more like a network. Unlike the conventional buses that share physical connections with multiple sources, PCIe has a switch that controls several pairs of point-to-point connections, each peripheral device has one pair of dedicated point-to-point connections. Moreover, PCIe is a packet-based protocol, all read and write transactions on the PCIe bus are packetized. The PCIe switch controls the flows of all packets from different peripheral devices just like network switches.

The PCI express protocol consists of four layers, the physical layer, the data link layer, the transaction layer and the application layer, as shown in Figure 6.1. In the thesis, we exploit the Xilinx PCI express endpoint block IP core and Xilinx GTP/GTX transceivers to develop a PCIe-based data communication application between FPGAs and DB servers. The IP core and Xilinx transceivers hide all the complexity of the physical layer and the data link layer, and it also offers some operations for the transaction layer. Our PCIe
engine handles the rest part of the transaction layer and supplies a standard interface to the application layer.

![Diagram of Database Storage Engine in PCIe protocol layers]

We first want to give an overview of PCI Express with respect to an FPGA implementation. We explain how the FPGA makes a memory write and a memory read operation first. To keep the simplicity, we omit some details. When the FPGA needs to write data to the DB server, it generates a memory write TLP (Transaction Layer Packet) that contains the size of data, the destination address in the main memory and the payload data. And then the TLP is dispatched by the Xilinx IP core to the chipset (PCIe root complex) on the motherboard of the DB server. The chipset unpacks this TLP and writes the payload data into the main memory on the DB server. For a memory read operation, it is a bit more complex as it is a non-posted transaction. First, the FPGA generates a memory read TLP that contains the size of data that it requests, the requested data address in the main memory, the FPGA’s ID, a generated unique tag for this transaction. After the TLP is dispatched to the PCIe root complex, the chipset then generates the completion TLP(s) that contain the FPGA’s ID, the unique tag, and the payload data. The completion TLPs are sent back to the FPGA to finish the read transaction.

Before we present the implementation of our storage engine, there are some background about TLPs to be explained in detail in the following section.

### 6.2 Transaction Layer Packet

Recall that all read and write transactions on the PCIe bus are packetized at the transaction layer; we send TLPs to start read and write transactions, and we receive the completion TLPs to finish the read transactions. Figure 6.2 shows the general view of a TLP’s structure [21].
Chapter 6. Using PCI Express

3 or 4 DWs 0 or 1023 DWs

<table>
<thead>
<tr>
<th>Header</th>
<th>Data Payload(if applicable)</th>
</tr>
</thead>
</table>

Figure 6.2: The General View of a TLP’s Structure

The \textit{Fmt} and \textit{Type} fields (Figure 6.3) in the first DW of TLP header are two important parameters to determine the size of TLP header and the type of the TLP.

The \textit{Fmt} shows the size of the TLP, and it contains data payload after the TLP header; the \textit{Type} field shows the types of TLPs: memory read, memory write, completions, I/O read, I/O write, message, flow control, and so on. The following table 6.1 shows part of TLP header type and format field encodings. We choose 3 important types of the TLPs that we want to explain with examples in the following subsections.

<table>
<thead>
<tr>
<th>Fmt</th>
<th>Type</th>
<th>TLP</th>
</tr>
</thead>
<tbody>
<tr>
<td>00</td>
<td>0</td>
<td>3DW header, no data payload, Memory Read Request in 32-bit addressing mode</td>
</tr>
<tr>
<td>01</td>
<td>0</td>
<td>4DW header, no data payload, Memory Read Request in 64-bit addressing mode</td>
</tr>
<tr>
<td>10</td>
<td>0</td>
<td>3DW header, with data payload, Memory Write Request in 32-bit addressing mode</td>
</tr>
<tr>
<td>11</td>
<td>0</td>
<td>4DW header, with data payload, Memory Write Request in 64-bit addressing mode</td>
</tr>
<tr>
<td>10</td>
<td>1</td>
<td>3DW header, with data payload, Completion with the payload data</td>
</tr>
</tbody>
</table>

Table 6.1: TLP Header Type and Format Field Encodings

6.2.1 Memory Write Request TLP

Figure 6.4 shows that the requester(0x0000, the PCIe root complex that is the closest PCIe port to CPU) of the TLP writes one DW (double word) data 0x00ABCDEF into the physical address 0xFFFEDDC0 using the 32-bit addressing mode. \textit{Fmt} = 0x2 shows that it is a TLP with 3 DW header and data payload. \textit{Fmt} = 0x2 and \textit{Type} = 0x00 present that it is a memory write request TLP, a posted TLP.
The Length represents the size of the data payload. The R field means that those bits are reserved, and should be set as 0.

![Figure 6.4: Example of Memory Write Request TLP](image)

The LastBE and FirstBE, which are the byte enable fields, are rarely used. Usually FirstBE is kept to F, and LastBE is set to 0 if the Length is 0x001. Otherwise, LastBE is set to F too. The Tag is unused in the posted TLPs, it should be set to 0x00.

### 6.2.2 Memory Read Request TLP

Figure 6.5 shows a memory read request TLP. The requester 0x0000 makes a request that read one DW from the physical address 0xFFEEDDC0 using the 32-bit addressing mode. Fmt = 0x0 shows that it is a TLP with 3 DW header without data payload. Fmt = 0x0 and Type = 0x00 represent that it is a memory read request TLP in 32-bit addressing. The Length represents the size of data that this TLP requested in DWs.

![Figure 6.5: Example of Memory Read Request TLP](image)

The Tag is set by the requester. The Tag = 0x03 is used as a unique identifier for this request. This would be useful to locate the specific request between multiple requests. The completion TLP of this request should always copy this value into its Tag field. The requester can use this tag to track completion TLPs.
### 6.2.3 Memory Read Completion TLP

Figure 6.6 shows a completion TLP of a memory read request. The completer 0x0020 sends the requested read data back to the requester 0x0000. Fmt = 0x2 shows that it is a TLP with 3 DW header with data payload. Fmt = 0x2 and Type = 0x0A represent that it is a completion TLP with data payload.

![Figure 6.6: Example of Memory Read Completion TLP](image)

The Length = 0x001 represents the size of the data payload. And byte Count = 0x004 shows that the size of rest data in the rest of completion packets. The Lower Address is the 7 least significant bits of the address of the first byte in this completion’s data payload, this field is used to arrange data from multiple completion TLPs. The value in the Tag must be set as the same as its value in the read request TLP. In our example, it should be 0x03.

In the next section, we present that the design of our PCIe engine for the database storage engine system.

### 6.3 PCIe Engine Implementation

Our PCIe engine implementation consists of several basic blocks, the architecture is shown in Figure 6.7. This section describes the implementation of the PCIe engine in detail. First we introduce the Xilinx PCIe Endpoint Block IP Core in Section 6.3.1. Then we show each component in the PCIe Engine by going through the receiving (Rx) and transmitting (Tx) data paths (Section 6.3.2 to Section 6.3.6). The performance of our PCIe engine is shown in Chapter 7.
6.3.1 Xilinx PCIe Endpoint Block

The Xilinx PCIe endpoint block IP core is hard IP core built in the Virtex 5 series FGAs. It is compatible with the PCI Express base specification v1.1. There are 4 interfaces connected to the core, System (SYS) interface, PCI Express (PCI_EXP) interface, Configuration (CFG) interface and Transaction (TRN) interface. The SYS interface is composed of the clock and reset signals of the FPGA. And the PCI_EXP interface is connected to the Xilinx GTX/GTP transceivers – the physical layer connections.

The CFG and TRN are two important interfaces that are used by the FPGA applications. The CFG interface is used to configure the PCI configuration space. Parameters in the
Chapter 6. *Using PCI Express*  

PCI configuration space, such as the vendor ID, the device ID, the I/O range, the BAR (Base Address Register) size can be configured through the CFG interface. The TRN interface is used to receiving and transmitting data. We left out a lot of details in this section, all information about this IP core can be found in the Xilinx user guide [22].

### 6.3.2 Rx Engine

The Rx engine is one component connected to the PCIe IP core via the TRN interface. It is in charge of receiving all TLPs exposed by the IP core, and decoding the TLPs. It dispatches TLPs according to the *Type* and *Fmt* of the TLP.

If the TLP is a memory write request from the host computer, then the Rx engine sends the write request data payload to the customized register bank (see Section 6.3.3) or the FPGA applications according to the customized settings.

If it is a memory read request, then the Rx engine sends the read request to the completion TLP generator (see Section 6.3.4) via the register bank, all essential information such as the *Tag*, the requested size, the requested address and the requester ID are stored in the register bank for generating the corresponding completion TLPs in the future stage.

If it is a completion TLP, then the Rx engine extracts the data payload, then dispatches the payload to the register bank or the FPGA applications according to the settings; the engine also sends *Tag* information to the completion tags management module for tracking the completions.

### 6.3.3 Customized Register Bank

The customized register bank is one module that contains a collection of registers. It is used as a temporary data buffer to store the incoming memory write requests and completions. At the same time, some registers in the bank are also used to store the control values for other modules in the FPGA, such as the TLPs generators, the FPGA applications. The host computer could control such modules through modifying the values of certain registers.

### 6.3.4 TLP Generators

There are 3 TLP generators in our PCIe engine. These generators generate the corresponding well-formed TLPs according the specified registers in the customized register bank. The value of the specified registers can be set by either the FPGA applications or
the host computer. They raise requests to the TLPs arbiter (see Section 6.3.5) once there is a transmit-pending TLP, and then wait for the arbitration result from the TLPs arbiter. Once they get the transmit permission from the TLPs arbiter, they send well-formed TLPs out via the Tx engine.

### 6.3.5 TLPs Arbiter

The TLPs arbiter is used to arbitrate the order of the requests from the following 3 TLP generators, the DMA read TLP generator, the completion TLP generator and the DMA write TLP generator.

The strategy that in our PCIe engine is the following: the priority of the completion TLP generator > the priority of the DMA read TLP generator > the priority of the DMA write TLP generator. Because the completion TLPs must be sent back in a critical time slot (see the PCI Express base specification for details), otherwise a timeout error will automatically generated by the IP core. And the completion TLPs is usually a completion for the register bank read, the size of this kind of TLPs is very small (4DW), it can be sent out very fast. Then the read request TLPs are assigned with the 2nd priority. Also because the size of the read request TLPs without the data payload is small, it can be sent out very fast. Finally, the write request TLPs are assigned with the 3rd priority, because the size of the write request TLPs is relatively big compared to the other two kinds of TLPs as it has the data payload in the TLPs, it could occupy the Tx channel longer than the other two kinds of TLPs.

However, the order of the priorities of 3 types of TLPs can be changed according to the application scenarios. For example, if the DMA write throughput is a critical requirement compared to the DMA read throughput in a design, then the priority of the write request TLPs should be assigned higher than the priority of the read request TLPs.

### 6.3.6 Tx Engine

Finally, the Tx engine is in charge of forwarding the well-formed TLPs from the DMA write TLP generator, the DMA read TLP generator and the completion TLP generator to the Xilinx PCIe Endpoint Block IP core via the TRN interface.
Chapter 7

Measurements

We have conducted a number of performance measurements on both our storage engine and the PCIe engine. Besides the performance measurements, we also measured the FPGA resource utilizations of both two versions of the storage engine and the PCIe engine. We also got the maximum clock frequency of our design from the time summary of Xilinx ISE (Integrated Software Environment). In this chapter, we present the results of our measurements. In Section 7.1, we first give an overview of the measurement units related to our measurements. Then, in Section 7.2, we give the FPGA detailed resource utilizations of each engine. In Section 7.3, we show the performance of the storage engine. Finally, in Section 7.4, we present the performance of the PCIe engine.

7.1 Measurement Units

In the following of this chapter, we will be concerned with 4 measurement units: FFs (Flip-Flops), LUTs (Look-Up Tables), Slices, and Block RAMs. A Flip-Flop is a 1-bit register circuit in the Slices of the FPGA. The LUT is basically a RAM-based n-input/1-output truth table. The slices are the elementary programmable logic blocks in Xilinx FPGAs. For the Virtex-5 FPGAs, there are four Flip-Flops and four 6-input LUTs in one slice. A block RAM is a dedicated two-port memory instance in the FPGA. There are several block RAMs in one FPGA, the size of one block RAM is usually 18Kbit or 36Kbit. All measurement results we present in the following sections are evaluated on the NetFPGA-10G development board with a Xilinx Virtex 5 XC5VTX240T (Speed Grade: -2) FPGA.
7.2 Resource Utilizations

We measured the number of FFs, LUTs, occupied slices and BRAM blocks used for both the storage engine using the plain data format and the storage engine using the structured data format. The results of the resource utilizations are shown in Table 7.1.

<table>
<thead>
<tr>
<th>Module</th>
<th>FFs</th>
<th>LUTs</th>
<th>Occupied Slices</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain Version</td>
<td>1,255</td>
<td>2,401</td>
<td>1,091</td>
<td>2</td>
</tr>
<tr>
<td>Structured Version</td>
<td>1,486</td>
<td>2,679</td>
<td>1,173</td>
<td>2</td>
</tr>
</tbody>
</table>

FPGA XC5VTX240T

<table>
<thead>
<tr>
<th>Module</th>
<th>FFs (MB/s)</th>
<th>LUTs (MB/s)</th>
<th>Occupied Slices</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe engine</td>
<td>6,562</td>
<td>3,890</td>
<td>2,506</td>
<td>35</td>
</tr>
<tr>
<td>FPGA XC5VTX240T</td>
<td>149,760</td>
<td>149,760</td>
<td>37,440</td>
<td>324</td>
</tr>
</tbody>
</table>

Table 7.1: Storage Engine Resource Utilizations

From the above table, we can see that there are only slightly increases of FPGA resource consumptions in the structured version storage engine. It worth to pay such small part of resources to get higher throughput (see Section 7.3).

In the following table 7.2, we show the result of the resource utilizations of the PCIe engine including the Xilinx PCIe endpoint block IP core.

<table>
<thead>
<tr>
<th>Module</th>
<th>FFs</th>
<th>LUTs</th>
<th>Occupied Slices</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>PCIe engine</td>
<td>149,760</td>
<td>149,760</td>
<td>37,440</td>
<td>324</td>
</tr>
</tbody>
</table>

Table 7.2: PCIe Engine Resource Utilizations

7.3 Storage Engine Performance

We are able to gauge the maximum clock frequency (period) for our design from the Place-and-Route (PAR) report in the Xilinx ISE. By using the Best Case Achievable values in the timing analysis report, we can get the maximum clock frequency and the throughput of our design. The results of the maximum clock frequency and the throughput of our storage engine are shown in Table 7.3.

<table>
<thead>
<tr>
<th>Version</th>
<th>Max. clock frequency</th>
<th>Max. throughput</th>
</tr>
</thead>
<tbody>
<tr>
<td>Plain Version</td>
<td>189.39 MHz</td>
<td>189.39 MB/s</td>
</tr>
<tr>
<td>Structured Version</td>
<td>172.71 MHz</td>
<td>690.84 MB/s</td>
</tr>
</tbody>
</table>

Table 7.3: Storage Engine Performance

The structured data format version data parser is running on a slight lower clock frequency 172.71 MHz than the plain data format version data parser’s clock frequency 189.39 MHz.
Chapter 7. Measurements

The reason is that the additional logic which is used to parse the control header of variable length columns increased the longest data path between two pipeline registers. Therefore, the clock frequency slowed down to keep timing constraints.

The latency introduced by the storage engine itself depends on the data it is processing. It can be up to \( S_{\text{max}}/w \) clock cycles, \( S_{\text{max}} \) is the maximum size of one record in the database table. For the plain data format storage engine, \( w \) is 1. \( w \) is equal to 4 for the structured data format storage engine. And we present the latency of data communications between the FPGA and the host computer in the next section.

### 7.4 PCIe Engine Performance

In this section, we present the throughput and the latency of PCIe-based communications between the FPGA and the host computer. The evaluation environment is shown in the Table 7.4.

From the datasheet of Intel 6 Series Chipset/Intel C200 Series Chipset [23], we know that both Max Payload Size and Max Read Request Size the 6 Series/C200 chipset supported are 128 bytes. We tested 100 rounds of 4MB DMA read and DMA write operations. The throughput and latency of our PCIe engine based on the above evaluation environment are shown in Table 7.5. The DMA Write latency is tested by counting the time between sending the first command at the computer and getting the first piece of data under the user space of OS. The DMA Read latency is measured by counting the clock cycles of the FPGA between sending the first request on the FPGA and getting the first piece of data back on the FPGA.

<table>
<thead>
<tr>
<th>Processor</th>
<th>Intel(R) Core(TM) i7-2700K CPU @ 3.50GHz</th>
</tr>
</thead>
<tbody>
<tr>
<td>Chipset</td>
<td>Intel 6 Series Chipset/Intel C200 Series Chipset</td>
</tr>
<tr>
<td>Memory</td>
<td>7648 MiB</td>
</tr>
<tr>
<td>Storage</td>
<td>WD5000AAKX 500GB 7200 RPM 16MB Cache SATA 6.0Gb/s</td>
</tr>
<tr>
<td>Operating System</td>
<td>Ubuntu 12.04 (precise) 3.2.0-27-generic</td>
</tr>
<tr>
<td>FPGA</td>
<td>NetFPGA-10G Virtex 5 XC5VTX240T</td>
</tr>
</tbody>
</table>

Table 7.4: Evaluation Environment

<table>
<thead>
<tr>
<th>Operations</th>
<th>Max. throughput</th>
<th>Min. latency</th>
<th>Max. latency</th>
<th>Average latency</th>
</tr>
</thead>
<tbody>
<tr>
<td>DMA write(4MB)</td>
<td>1588.05MB/s</td>
<td>4.075 ( \mu )s</td>
<td>14.903 ( \mu )s</td>
<td>4.501 ( \mu )s</td>
</tr>
<tr>
<td>DMA read (4MB)</td>
<td>1105.22MB/s</td>
<td>0.868 ( \mu )s</td>
<td>1.224 ( \mu )s</td>
<td>0.909 ( \mu )s</td>
</tr>
</tbody>
</table>

Table 7.5: PCIe Engine Performance
Latency Discussion

Compared to the latency range of 1Gb Ethernet ip forwarding which is typically 50 $\mu$s to 125 $\mu$s [24], our PCIe engine decreased the latency by an order of magnitude, the PCIe-based interface is more suitable for a latency-intensive co-processing system.

Throughput Discussion

We can see that DMA write operations have a reasonable good performance\(^1\). However, the throughput of DMA read operations is unremarkable. We believe that there are 3 reasons. First, the inherent non-posted property of DMA read operations makes more overhead than posted operations (e.g. DMA write). Second, the Max Read Request Size of our system (Intel 6 Series Chipset) is too small to get better performance. Third, because our PCIe engine reads data from the host computer, which is the root complex in the PCIe architecture. The read completion boundary(RCB) is 64 bytes other than 128 bytes in such case, which generates more overhead at the transaction layer (see PCI Express base specification for more details).

\(^1\)For Max Payload Size = 128 bytes, the best possible achievable throughput is $86\% \times 2$GB/s $\approx 1760$ MB/s without considering the traffic overhead of ACK, NACK and flow control packets.
Chapter 8

Conclusion

8.1 Selection and Projection Based Filtering in FPGAs

The main goal of this thesis was to explore more possibilities in the FPGA coprocessors acceleration domain. In the first part of the thesis, we implemented an FPGA-based database storage engine with the selection and projection based filtering and the at-runtime reconfigurable abilities. There is no high FPGA-compilation cost of accepting arbitrary filtering queries dynamically.

8.1.1 Runtime Reconfigurable Filtering

To build an arbitrary selection and projection based filter in the storage engine, we discussed the design of the storage engine architecture from the ground up. The behaviors of all modules in the storage engine and how the framework of the storage engine works with the at-runtime reconfigurable ability are shown in Chapter 4.

8.1.2 Data Formats for the data processing on FPGAs

In order to let FPGAs have capabilities to process data on persistent devices, we defined an FPGA-readable plain storage format in Chapter 3. Then we discussed the throughput of the storage engine with the plain storage format. The one-byte-per-clock-cycle bottleneck in the plain storage format motivated us to explore alternative solutions. Therefore we introduced the structured data format in Chapter 5, and we optimized the storage engine system with a focus on performance. The performance of the new data parser with the structured data format increased up to 4 times.
8.1.3 Selection and Projection Based Filtering Performance

We showed that the throughput of our smart storage engine can achieve 4 bytes per clock cycle; the maximum clock frequency of the corresponding implementation on Xilinx Virtex 5 XC5VTX240T (Speed Grade: -2) is around 172.7 MHz, which means the maximum throughput is about 690 MB/s (172.7 MHz × 4 bytes). It is a remarkable throughput. With this, we believe that we have accomplished the main goal stated in Section 1.2.

To sum up the above 3 subsections, we can state that it is possible to build a high performance and low cost storage engine with flexible selection/projection based filtering capabilities on FPGAs.

8.2 Using PCI Express in the FPGA Co-processing System

In the latter part of the thesis, we developed a PCIe engine for the FPGA applications by exploiting the Xilinx Endpoint Block IP for PCI Express. We introduced PCI Express and showed the design of our PCIe engine in Chapter 6. And we also developed the dedicated drivers for testing the performance of our PCIe-based devices. The throughput and the latency of PCIe-based communications are shown in Chapter 7. Compared with ethernet-based solutions, the PCIe-based solution has serval advantages such as high throughput, high reliability, low latency, the DMA capability between FPGAs and the host computer. It is suitable as a interface between FPGAs and host computers in the FPGA co-processing system.

In the following section, we point out some aspects and directions requiring more work and further research.

8.3 Future Work

While the current storage engine system is already usable, there are still some aspects that can be improved.

- 1. In the current stage, all RAMs used in the storage engine are on-chip RAMs (Block RAMs, LUTs-based RAMs). As there is no sufficient RAM resources in a FPGA chip, on-board RAMs (SDR, DDR, SRAM) instead of on-chip RAMs can be considered for the scalability.
• 2. The support of more data types (i.e. DATE, DECIMAL) can be added into the selection comparator.

• 3. More customized comparison operators can be added into the selection comparator of the storage engine.

• 4. It would be an interesting direction that pushing down more functions such as aggregation, group by, into the FPGA-based storage engine.

• 5. A powerful and advanced space management unit on the FPGA.

• 6. Support column-store and column-oriented DBMS.
Appendix A

Glossary of Abbreviations

ASIC : Application Specific Integrated Circuit
CLB : Configurable Logic Block
CPLD : Complex Programmable Logic Device
CPU : Central Processing Unit
DBMS : Database Management System
FPGA : Field-Programmable Gate Array
FSM : Finite-State Machine
HBA : Host Bus Adapter
ISE : Integrated Software Environment (Xilinx software)
LBA : Logical Block Addressing
LUT : LookUp Table
MIMD : Multiple Instruction, Multiple Data
PCI : Peripheral Component Interconnect
PCIe : PCI Express
PLA : Programmable Logic Array
PLD : Programmable Logic Device
RAM : Random Access Memory
SATA : Serial ATA, Serial AT Attachment
SIRC : Simple Interface for Reconfigurable Computing
TLP : Transaction Layer Packet
VHDL : VHSIC Hardware Description Language
VHSIC : Very High Speed Integrated Circuit
## Appendix B

### Capacities of SATA and PCI Express

The table below shows the capacities of the SATA visions.

<table>
<thead>
<tr>
<th>SATA revision</th>
<th>Native capacity</th>
<th>Coding scheme</th>
<th>MUTR</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>1.5 Gbit/s</td>
<td>8b/10b</td>
<td>150 MB/s</td>
</tr>
<tr>
<td>2.0</td>
<td>3 Gbit/s</td>
<td>8b/10b</td>
<td>300 MB/s</td>
</tr>
<tr>
<td>3.0</td>
<td>6 Gbit/s</td>
<td>8b/10b</td>
<td>600 MB/s</td>
</tr>
</tbody>
</table>

The table below lists the capacities of the PCI Express visions in each direction.

<table>
<thead>
<tr>
<th>PCIe revision</th>
<th>Native capacity per lane</th>
<th>Coding scheme</th>
<th>MUTR X1</th>
<th>X16</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>2.5 Gbit/s</td>
<td>8b/10b</td>
<td>250 MB/s</td>
<td>4 GB/s</td>
</tr>
<tr>
<td>2.0</td>
<td>5 Gbit/s</td>
<td>8b/10b</td>
<td>500 MB/s</td>
<td>8 GB/s</td>
</tr>
<tr>
<td>3.0</td>
<td>8 Gbit/s</td>
<td>128b/130b</td>
<td>1 GB/s</td>
<td>16 GB/s</td>
</tr>
</tbody>
</table>

MUTR : Maximum Uncoded Transfer Rate.
Appendix B. Capacities of SATA and PCI Express
Bibliography


