Less Watts, More Performance: An Intelligent Storage Engine for Data Appliances

Louis Woods   Gustavo Alonso
Systems Group, Dept. of Computer Science
ETH Zurich, Switzerland
{firstname.lastname}@inf.ethz.ch

Jens Teubner
DBIS Group, Dept. of Computer Science
TU Dortmund University, Germany
jens.teubner@cs.tu-dortmund.de

ABSTRACT

In this demonstration, we present *Ibex*, a novel storage engine featuring *hybrid*, FPGA-accelerated query processing. In *Ibex*, an FPGA is inserted along the path between the storage devices and the database engine. The FPGA acts as an *intelligent storage engine* supporting query off-loading from the query engine. Apart from significant performance improvements for many common SQL queries, the demo will show how *Ibex* reduces data movement, CPU usage, and overall energy consumption in database appliances.

Categories and Subject Descriptors

H.2 [**Database Management**]: Systems

Keywords

Storage, Intelligent, Data Appliance, Energy, FPGA, Ibex

1. INTRODUCTION

Modern data appliances such as IBM's Netezza [6] and Oracle's Exadata [10] are moving toward architectures with widespread query off-loading to different elements of the system. The goal is both to speed up query processing and to reduce data movement since the network bandwidth is still one of the dominant bottlenecks in cluster-based appliances. An important component in these systems is an *intelligent storage engine*, *i.e.*, a storage engine that already filters data before sending it to the query processor, by executing queries in part or whole on behalf of the processing nodes.

In this demonstration, we present *Ibex*, an intelligent storage engine for a data appliance that we are building at ETH Zurich [1] that is based on *field-programmable gate arrays* (FPGAs). FPGAs are massively parallel semiconductor devices that have attractive properties for data processing, which has been shown in a number of recent research papers [2, 7, 8, 11]. Unlike existing approaches, *Ibex* supports off-loading of more complex operators than simply selection and projection, *e.g.*, `GROUP BY` aggregation. As far as we are aware, this is the first demonstration of an intelligent storage engine at a database conference, and thus will serve to illustrate an important architectural database trend.

SIGMOD'13, June 22–27, 2013, New York, New York, USA.
Copyright 2013 ACM 978-1-4503-2037-5/13/06 ...$10.00.

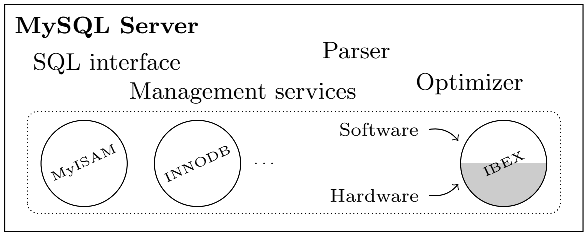


Figure 1: *Ibex* as a *pluggable* storage engine.

The demonstration will allow attendees to see how off-loading query processing to an FPGA-based accelerator *improves performance* and also *reduces energy consumption.* Query processing in relational databases may cause substantial CPU-activity [3]. As the low-power FPGA[1] is doing work for the CPU, the CPU can switch to a sleep state, consuming less power. Alternatively, the CPU has spare cycles for other work, and with extensive off-loading even a less powerful CPU would suffice for equivalent performance.

For the purposes of this demonstration, we chose MySQL as the database engine. MySQL 5.1 has introduced a *pluggable* storage engine architecture that makes it particularly easy to develop new storage engines (Figure 1). However, the ideas behind *Ibex* can be applied to many different engines from OLTP to OLAP and from row to column stores.

Integrating an FPGA into a DBMS requires profound knowledge of both the database and the FPGA domain. We have been working in this area for quite some time now, gathering many insights. As such, the work we demonstrate here combines many elements developed over the years, involving compiling query plans to circuits (*Glacier* [9]), techniques for runtime-reconfiguration of FPGAs (skeleton automata [12]), as well as low-level hardware components such as a SATA controller for FPGAs (Groundhog [13]).

Goals. The demonstration has two objectives. One is to discuss *performance improvements* and *energy savings* on a per-query basis. The fact that MySQL supports multiple storage engines makes live comparisons of *Ibex* with MyISAM and INNODB possible. The other is to provide attendees with a better understanding of intelligent storage engines, from a software as well as a hardware perspective.

[1]An FPGA consumes between one and two orders of magnitude less power than a server processor under full load.

2. THE SOFTWARE ARCHITECTURE

MySQL implements the *Volcano iterator model* [5]. To scan a table, MySQL repeatedly calls the storage engine method `rnd_next(uchar *buf)`, which fetches the next row of a table. Typically, query execution tasks such as *selection*, *aggregation*, and *projection* are performed in higher-level components above the storage engine. However, the MySQL storage engine interface already provides a mechanism to push selection down to the storage engine, which is used, for example, by the NDB storage engine designed for cluster environments.

MySQL ⟷ Storage Engine Interface. Besides selection we would like to to push projection and `GROUP BY` *aggregation* down to the *Ibex* storage engine. Therefore, we have extended the existing storage engine interface of MySQL with two additional methods, namely `proj_push(...)` and `grpby_push(...)`. The query optimizer can detect whether these methods are implemented by a given storage engine and optimize queries appropriately. On the other hand, a storage engine that implements these methods receives additional preprocessing information from MySQL.

Software Storage Engine ⟷ FPGA Interface. There are several ways to physically connect an FPGA to a host machine running a database. For our FPGA board, this can be done using PCI Express or Ethernet—for this demo we will use Ethernet because it allows us to take the FPGA board out of the PC, making the demo more ostensive.

We base our interface to the FPGA on MSR's SIRC (simple interface for reconfigurable computing [4]), which provides a uniform communication interface to the FPGA, hiding from the application the physical communication channel used and low-level protocol details. In essence, SIRC provides access to the input and output buffers on the FPGA, depicted in Figure 3, by means of a few C++ methods. Auxiliary methods to access a register file are used for control flow and parametrization of the hardware engine.

On top of this low-level communication interface to the FPGA, we built a higher-level class that enables accessing persistent storage via the FPGA. Apart from providing basic read/write block (a number of 512-byte sectors) methods, we also support scanning an entire table with projection, `GROUP BY` aggregation, and selection possibly pushed down to the FPGA, as depicted in Figure 2.

```
setProjection(columns);
setGroupByAgg(columns, aggtype, groupbykey);
setSelection(columns, predicates, tt);
scanTable(startLBA);
while(fetchResult()) { ... }
```

Figure 2: Full-table scan with multiple operators pushed down to the FPGA (tt = truth table).

Before scanning a table, we need to specify the computation that we want to push down to the FPGA. For projection, we simply indicate, which columns we want to retrieve. For `GROUP BY` aggregation, besides the aggregate columns we also need to specify the aggregation type (`COUNT`, `SUM`, etc.) for every column, as well as the `GROUP BY` key. Finally, for selection, we need to submit the columns that need to be evaluated against a `WHERE`-clause predicate together with the corresponding predicates for each column and a truth table (`tt`) that represents the Boolean concatenation of multiple predicates in the same `WHERE`-clause (see Section 3.1 for more details). The table scan is then initiated by calling `scanTable(startLBA)`, which submits the first *logical block address* (LBA) of the corresponding table to the FPGA. The results are then retrieved in multiple chunks by repeatedly calling `fetchResult()`.

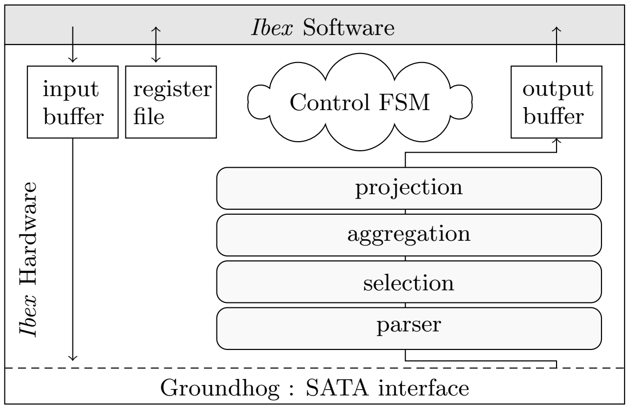


Figure 3: Query engine on the FPGA.

3. THE HARDWARE ARCHITECTURE

The high-level hardware architecture of *Ibex* is shown in Figure 3. Communication between the software side and the FPGA is realized via input and output buffers, as well as a register file for control flow and parametrization of the hardware query engine.

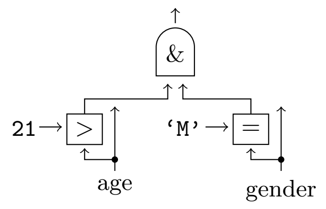
The demonstration focuses on read-only queries. Nevertheless, *Ibex* also supports updates bypassing the FPGA through Groundhog [13], an open-source SATA controller for FPGAs.

When a table is read, the data passes through a series of pipelined hardware components (*parser*, *selection*, *aggregation*, and *projection*), before it reaches the output buffer, from which it is sent to the software component. The *parser component* is the first in the pipeline. It consumes 16 bits per clock cycle of the raw byte stream delivered by Groundhog [13] and separates the columns of a row from each other by writing the respective values to different FIFO buffers. We only need to buffer the data for the relevant columns of a given query. The *selection component* concurrently evaluates (potentially) multiple predicates on a record and then computes the Boolean expression corresponding to the `WHERE`-clause. If the Boolean result is *false*, the respective record is discarded at this stage. Otherwise, the record is forwarded to the next stage. The *aggregation component* performs the necessary aggregations on the records that have passed the *selection component*, in the event of a `GROUP BY` query. The *projection component*, finally, materializes the projected columns and writes the projected result record to the output buffer for transmission to the *Ibex* software.

At the demonstration, the inner workings of all of these components will be presented in detail. Here, for lack of space, we will only discuss the *selection component* and the *aggregation component* in the two sections below.

3.1 Selection Component

Instead of hard-wiring predicate evaluation, we created a template circuit for every supported column type, which



Addr.	Value
00	0
01	0
10	0
11	1

Figure 4: Truth table for the evaluation of complex Boolean expressions versus hard-wiring.

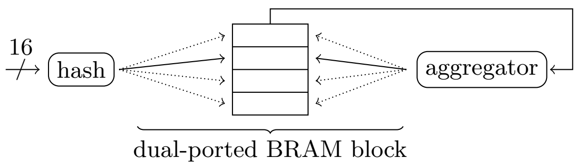


Figure 5: Dual-ported BRAM-based hash table.

can be parameterized at runtime—similar to the *skeleton automata* technique that we described in [12]. For instance, to evaluate `table.age > 21`, we simply need to store the value 21 as well as the desired comparator `>` ($\in$ {`=`,`<>`,`<`,`>`,`<=`,`>=`}) in local memory of the template circuit. String and regular expression matching on `VARCHAR` and `TEXT` columns can be performed with parameterizable finite-state machines, again using similar techniques as discussed in our previous work [12].

The problem that remains is combining multiple predicates into a single Boolean expression. Consider, for example, the following query:

```
SELECT * FROM table WHERE age > 21 AND gender = 'M'
```

Hard-wiring the Boolean `AND` operator, as illustrated on the left-hand side in Figure 4, is not an option since we need to change this wiring at runtime, *e.g.*, without having to reprogram the entire FPGA. A template-based approach, as we proposed for evaluating individual predicates, is also not feasible since the Boolean expressions in a `WHERE`-clause are unpredictable and can be arbitrarily complex.

Instead of wiring the output of the predicate evaluation circuits through series of gates we can combine those wires into a single address bus to an on-chip memory block, which then serves as a look-up table to evaluate a given Boolean expression—an example is displayed on the right-hand side in Figure 4 (here, for a single `AND` operator). Thus, before we run a query on the FPGA, we compute the truth table for the `WHERE`-clause in software and load that truth table into the dedicated on-chip memory on the FPGA.

3.2 Aggregation Component

Aggregation in `GROUP BY` queries quickly becomes a costly operation in existing relational databases. Using a hash-based scheme our aggregation component is able to compute multiple aggregates on the fly at line rate, as data is loaded from persistent storage. To achieve this, we use established techniques to leverage hardware parallelism. In particular, in *Ibex*, we aggressively use *pipeline parallelism* and *parallel collision resolution.*

Pipeline parallelism. Once a SATA read command has been issued, data needs to be processed at line rate, *i.e.*, every clock cycle the next 16 bits of data have to be consumed. This seems to be a problem because to process a key, we need to probe the hash table, update the aggregate, write it back to memory, and in the next clock cycle be ready to process the next key. Fortunately, on an FPGA all these steps can be pipelined, as illustrated in Figure 5. In addition, dual-ported BRAM blocks allow us to read and write data from/to the same BRAM block simultaneously. Thus, as we are probing one key, we can write back the aggregate for a different key.[2]

Parallel collision resolution. Since we are using hashing there exists the possibility of a collision, *i.e.*, two keys could be mapped to the same BRAM address. Typically collisions are resolved by finding an alternative address in the hash table using techniques such as *linear probing*, *double hashing*, etc. But such a strategy would make the time it takes to process a key unpredictable—a situation that should be avoided in FPGA designs.

However, if we construct our hash table from multiple parallel BRAM blocks, then we can proactively and in parallel look up an alternative address in each of the BRAM blocks and resolve collisions without delaying processing. While this technique significantly reduces the probability of a collision, n alternative addresses might still not be sufficient for each and every case. If a tuple causes a collision that cannot be resolved with n parallel lookups, that tuple is forwarded to the *Ibex* software component, where it can be taken care of later.

4. DEMONSTRATION SETUP

We bring a MySQL server installed on a high-end laptop, which communicates to an FPGA via Gigabit Ethernet using the SIRC framework [4] (see Section 2). The FPGA has a direct SATA link to an SSD, as illustrated in Figure 6. Additionally, we bring an electricity meter to measure the power consumption of the laptop during query execution.

We run the MySQL client on the laptop, and attendees will be able to execute either a set of predefined queries or type their own SQL queries at the MySQL prompt. A number of large tables using the MyISAM, INNODB and *Ibex* storage engine will be pre-installed. Tables that use *Ibex* will be stored on the SDD connected to the FPGA board, whereas tables using other storage engines are stored on an (equivalent) local SDD of the laptop. If time allows, visitors can even create and populate their own tables using *Ibex* or another storage engine.

4.1 Storage Engine Performance

Query execution times are reported directly by MySQL and displayed on the client terminal. Our test database will contain tables which are replicated among the different storage engines. Thus, the performance of different engines can be directly compared for a particular query. For many SQL queries, *Ibex* will significantly outperform the other storage engines in MySQL. To give an example, running the following simple `GROUP BY` aggregation query

```
SELECT id, count(id) FROM table GROUP BY id
```

on a one-gigabyte table with a single column of type `INT`, consisting of four different groups, was executed in **54.18 seconds** with MyISAM, in **179.27 seconds** with INNODB,

[2]Potential race conditions that could occur when two identical keys directly follow each other are avoided by an extra circuit that detects and handles this situation.

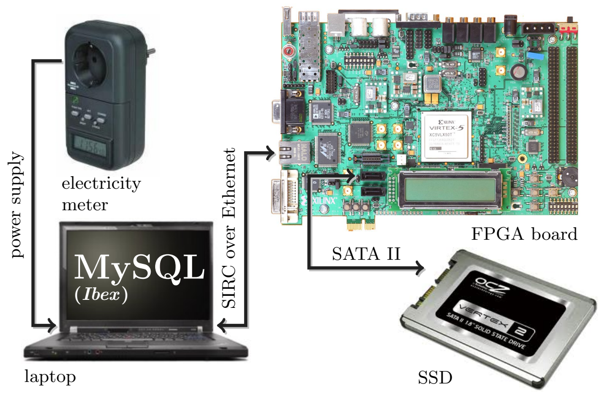


Figure 6: Hybrid MySQL Server Setup.

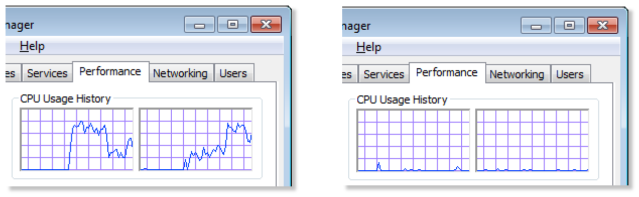


Figure 7: CPU usage during query execution: storage engine MyISAM (left) versus *Ibex* (right).

and in only **3.71 seconds** with *Ibex*. For 1 GB of data, 3.71 seconds correspond to a throughput of 270 MB/s, which is about the maximum sequential read speed on a SATA II SSD.

4.2 CPU Usage and Power Consumption

Besides performance improvements, we also demonstrate how our system improves energy consumption. At the demo, we will monitor CPU usage of the laptop running the MySQL server. Figure 7 displays two screenshots of the CPU usage monitor on our dual core laptop during execution of the same `GROUP BY` query used in the previous section.

While the CPU is involved quite significantly when the query is executed using the MyISAM storage engine, it is virtually idle with *Ibex* since almost the entire computation is pushed down to the FPGA.

The reduced CPU usage with *Ibex* directly translates to less power consumption of the host system, which is visible on our electricity meter that measures the wall power consumed by the laptop. During query execution with the MyISAM storage engine, our laptop consumes 45 watts of power, whereas with *Ibex* power consumption is only 31.5 watts. The FPGA itself running the hardware component of *Ibex* only consumes 2.8 watts of power.[3]

Since *Ibex* not only consumes less power but also executes queries faster than MyISAM, energy savings are even more significant. For the `GROUP BY` query above, the system roughly needs $E = 45 \times 54.18 =$ **2438 joules** of energy using MyISAM, whereas only $E = 31.5 \times 3.71 =$ **117 joules** are required with *Ibex* to execute the same query.

[3]FPGA power consumption was estimated using the Xilinx Power Analyzer tool.

4.3 Understanding the System

We have added a number of hooks to our system that allow visitors to better understand the inner workings of our system. The MySQL server can be run in *verbose* mode, such that for every query on tables using the *Ibex* storage engine the parts of the query that are pushed down to the FPGA are displayed on the server console. On the FPGA side, our circuits are equipped with monitoring circuitry that allow us to inspect and display data passing through the FPGA at runtime. Finally, there will also be the possibility to look at C++ source code of the *Ibex* storage engine, as well as the VHDL and Verilog code of its hardware counterpart.

Acknowledgements

This work was done while Jens Teubner was a postdoc at ETH Zurich, funded by the Swiss National Science foundation (*Ambizione* grant for project *Avalanche*). The work was further supported by the Enterprise Computing Center (ECC) of ETH Zurich (`http://www.ecc.ethz.ch/`).

5. REFERENCES

[1] Gustavo Alonso, Donald Kossmann, and Timothy Roscoe. SwissBox: An Architecture for Data Processing Appliances. In *CIDR'11*, Asilomar, CA, USA, 2011.

[2] Arvind Arasu et al. Orthogonal Security With Cipherbase. In *CIDR'13*, Asilomar, CA, USA, 2013.

[3] Peter Boncz, Marcin Zukowski, and Niels Nes. MonetDB/X100: Hyper-Pipelining Query Execution. In *CIDR'05*, Asilomar, CA, USA, 2005.

[4] Ken Eguro. SIRC: An Extensible Reconfigurable Computing Communication API. In *FCCM'10*, Charlotte, NC, USA, 2010.

[5] Goetz Graefe. Volcano - An Extensible and Parallel Query Evaluation System. *IEEE Transactions on Knowledge and Data Engineering*, 1994.

[6] IBM/Netezza. Whitepaper. `http://www.redbooks.ibm.com/abstracts/redp4725.html`.

[7] Dirk Koch and Jim Torresen. FPGASort: A High Performance Sorting Architecture Exploiting Run-time Reconfiguration on FPGAs for Large Problem Sorting. In *FPGA'11*, Monterey, CA, USA, 2011.

[8] Roger Moussalli et al. Massively Parallel XML Twig Filtering Using Dynamic Programming on FPGAs. In *ICDE'11*, Hannover, Germany, 2011.

[9] R. Müller, Jens Teubner, and Gustavo Alonso. Streams on Wires—A Query Compiler for FPGAs. In *VLDB'09*, Lyon, France, 2009.

[10] Oracle. Whitepaper. `http://www.oracle.com/technetwork/database/exadata/exadata-technical-whitepaper-134575.pdf`.

[11] Mohammad Sadoghi et al. Multi-query Stream Processing on FPGAs. In *ICDE'12*, Washington, DC, USA, 2012.

[12] Jens Teubner et al. Skeleton Automata for FPGAs: Reconfiguring without Reconstructing. In *SIGMOD'12*, Scottsdale, AZ, USA, 2012.

[13] Louis Woods and Ken Eguro. Groundhog—A Serial ATA Host Bus Adapter (HBA) for FPGAs. In *FCCM'12*, Toronto, Canada, 2012.