Snowfall: Hardware Stream Analysis Made Easy

Jens Teubner Louis Woods
ETH Zurich, Systems Group · Universitätstrasse 6 · 8092 Zurich, Switzerland
{jens.teubner | louis.woods}@inf.ethz.ch

1 Introduction

Field-programmable gate arrays (FPGAs) are chip devices that can be runtime-reconfigured to realize arbitrary processing tasks directly in hardware. Industrial products [Net, Xtr] as well as research prototypes [MTA09, MVB+09, SLS+10, TMA11] demonstrated how this capability can be exploited to build highly efficient processors for data warehousing, data mining, or stream analysis tasks.

On the flip side, the construction of dedicated hardware circuits requires considerable engineering efforts and skills that are often not available in application-focussed development teams. To bridge this gap, at ETH we have developed a set of tools that aid developers of high-performance stream processing solutions and enable agile hardware generation for changing application demands.

In this demonstration, we showcase *Snowfall*, a compiler tool for low-level stream analysis. Comparable to scanner generators for software-based systems (*e.g.*, `lex`/`flex`), *Snowfall* can be used to decode incoming data streams in hardware, react to low-level patterns in a stream, and perform initial input data analysis. *Snowfall* plays well together with *Glacier*, a query-to-hardware compiler that we described and demonstrated in [MTA09, MTA10]. A typical use case is to use *Snowfall* for input parsing and pre-processing, then perform SQL-style query processing on top with a hardware query plan obtained with the help of *Glacier*.

In the demo, we illustrate *Snowfall* based on a real-world use case with exceptionally high demands for throughput and latency. With the help of *Snowfall*, we perform *risk checking* for *financial trading* applications. *Snowfall* allows for a declarative description of the problem, yet will generate a hardware circuit that can process input streams in real time.

2 Field-Programmable Gate Arrays and State Machines

Field-programmable gate arrays (FPGAs) are programmable chip devices that can implement electronic circuits directly in hardware. They are programmed with a *hardware description language* such as VHDL or Verilog. Vendor-provided *synthesis tools* map circuit descriptions expressed in these languages to basic FPGA device primitives (*e.g.*, *lookup*

```
SOH              = 0x01;   # special value "SOH" (field delimiter)
FIXVersion       = "4.2";

# FIX Data Types
Length           = [0-9]+;
Qty              = [0-9]* ('.' [0-9]*)?;
String           = (any - SOH) *;

# FIX Fields
BeginString      = "8=FIX." FIXVersion SOH;
BodyLength       = "9=" Length SOH;
CheckSum         = "10=" (any - SOH){3} SOH;
AnyField         = [1-9] [0-9]{0,3} "=" (any - SOH)* SOH;

NewOrderSingleMessage =
  BeginString BodyLength "35=D" SOH   # msg type = NewOrderSingleMessage
  AnyField * :>> "110=" Qty SOH       # quantity of executed order
  AnyField * :>> "55=" String SOH     # symbol
  AnyField * :>> "54=1" SOH           # this order is a buy
  AnyField * :>> CheckSum @check_order;

main := NewOrderSingleMessage;
```

Figure 1: Excerpt from a parser specification to decode FIX order messages.

tables; *flip-flop registers*; or *Block RAMs*). They generate a *bit stream* that, when uploaded to the FPGA, instantiates these primitives and realizes the hardware circuit.

Probably the most important design technique for FPGA circuits is the use of *finite state machines*, be it to implement the control logic that complements the data flow-oriented circuit components; to communicate with external devices; or to interpret data streams or protocols. Finite state machines fit the available FPGA chip resource types well and can run at very high speeds.

Designing the proper state machine for a given application need, however, can be tedious and error-prone. Even for relatively simple tasks, the necessary state machine can quickly grow too large to be truly understood by a human developer. And once programmed successfully, state machines tend to be hard to document and maintain. The problem is exacerbated by the necessity to express the state machine in VHDL or Verilog—languages that typical application developers are rarely familiar with.

3 *Snowfall*

Snowfall, which is part of a tool set that we develop in the context of the *Avalanche* project at ETH Zurich, addresses both aspects of the problem. It provides a high-level abstraction to express state machines and associated semantic actions.[1] *Snowfall* optimizes these state machines and emits VHDL code that implements them efficiently in hardware.

[1] *Snowfall* is based on the Ragel state machine compiler `http://www.complang.org/ragel/`.

Figure 1 shows an excerpt of the *Snowfall* code that decodes FIX messages for online trading applications. The code describes the lexical structure of buy orders (message type 'NewOrderSingleMessage' in the FIX specification) and inspects the quantity and stock symbol fields (FIX fields 110 and 55).

From the code in Figure 1, *Snowfall* will build a hardware state machine (expressed in VHDL) that recognizes the specified FIX message type. Whenever (parts of) the state machine have successfully matched on the input data, it will trigger the execution of *action code* blocks. These contain user-defined VHDL code that can be used to process lexical elements in the input stream (`lex`/`flex` are used in a similar way in software-based systems).

Since in this demo description we are restricted by space limits, Figure 1 shows only one example of how action code can be embedded into a *Snowfall* language specification. The `@check_order` annotation after the `CheckSum` syntactical element specifies that the routine `check_order` should be invoked whenever a full `NewOrderSingleMessage` was successfully parsed.

A typical implementation of an action code like `check_order` will build an internal representation of a FIX order tuple (*e.g.*, of schema $\langle \mathsf{quantity}, \mathsf{symbol} \rangle$). The tuple is then forwarded on to further hardware logic that performs high-level analysis of the stream of FIX orders. One such analysis task could be an assessment of the *risk* associated with the orders made. For instance, we would like an *alert* to be raised whenever the order volume within a given time window exceeds a certain limit.

3.1 *Glacier*: A Query-to-Hardware Compiler

Higher-level stream analysis tasks are a good fit for *Glacier*, another part of our FPGA toolbox. *Glacier* is a SQL-to-hardware compiler. Given a query in an SQL dialect with streaming extensions, *Glacier* generates the VHDL description of a corresponding *hardware query plan*. The inner workings of *Glacier* are the subjects of [MTA09, MTA10].

To implement our risk analysis example, a *Glacier*-generated hardware plan consumes tuples that our FIX parser constructed in `check_orders` and performs *aggregation* and *windowing* on the tuple stream. For instance, the query

```
SELECT SUM(quantity) AS qsum
  FROM orders [SIZE 600 ADVANCE 60 TIME]
 GROUP BY symbol
```

aggregates all orders over a window of 10 minutes and reports the ordered quantities for each stock symbol every minute. A violation of risk limits could easily be detected from the aggregated output of this query; or a dedicated query could be written that only emits data in alert situations.

In summary, the combination of *Snowfall* and *Glacier* makes the development of stream processing solutions on FPGAs comparable to a typical software development work flow.

At the same time, generated solutions will run as bare-hardware implementations and thus benefit from the architectural advantages offered by the FPGA technology. In particular, the risk analysis example sketched here benefits from *network-speed processing*—no order will be missed even under peak load—and *real-time latency*—the system could react to risk violations within sub-microseconds time.

4 Demonstration Setup

The real value of our tool set results from the seamless interplay among our own tools, but also with commercial FPGA synthesis and simulation tools. To make this point, we will bring to Kaiserslautern not only *Snowfall*, but also a full FPGA design environment as well as FPGA hardware. We will show how a full example application can be developed, simulated, and debugged; and we will show how the resulting application can process a (synthetic) FIX message stream in real time.

Visitors of the demo will be invited to modify our code examples, write their own queries for *Glacier*, and inspect the generated hardware solution using commercial circuit visualization tools. The focus application for this demonstration, *Snowfall*, includes functionality to debug and visualize generated state machines. We will show and explain this functionality and illustrate how *Snowfall* eases the development of FPGA-based stream processing solutions.

Acknowledgements

This work was supported by SNF *Ambizione* grant number 126405 and by the Enterprise Computing Center (ECC) of ETH Zurich (`http://www.ecc.ethz.ch/`).

References

[MTA09] Rene Mueller, Jens Teubner, and Gustavo Alonso. Streams on Wires—A Query compiler for FPGAs. *Proceedings of the VLDB Endowment (PVLDB)*, 2(1), August 2009.

[MTA10] Rene Mueller, Jens Teubner, and Gustavo Alonso. Glacier: A Query-to-Hardware Compiler. In *Proc. of the 2010 ACM SIGMOD Conference on Management of Data*, Indianapolis, IN, USA, June 2010.

[MVB+09] Abhishek Mitra, Marcos R. Vieira, Petko Bakalov, Vassilis J. Tsotras, and Walid A. Najjar. Boosting XML Filtering Through a Scalable FPGA-Based Architecture. In *Int'l Conference on Innovative Data Research (CIDR)*, Asilomar, CA, January 2009.

[Net] Netezza Inc. `http://www.netezza.com/`.

[SLS+10] Mohammad Sadoghi, Martin Labrecque, Harsh Singh, Warren Shum, and Hans-Arno Jacobsen. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. *Proceedings of the VLDB Endowment (PVLDB)*, 3(2), September 2010.

[TMA11] Jens Teubner, Rene Mueller, and Gustavo Alonso. Frequent Item Computation on a Chip. *IEEE Transactions on on Knowledge and Data Engineering (TKDE)*, 2011.

[Xtr] XtremeData Inc. `http://www.xtremedata.com/`.