Real-Time Pattern Matching with FPGAs

Louis Woods, Jens Teubner, Gustavo Alonso

Systems Group, Department of Computer Science, ETH Zurich
{firstname.lastname}@inf.ethz.ch

***Abstract*— We demonstrate a hardware implementation of a *complex event processor*, built on top of *field-programmable gate arrays* (FPGAs). Compared to CPU-based commodity systems, our solution shows distinctive advantages for stream monitoring tasks, *e.g.*, *wire-speed processing* and *predictable performance*.**

The demonstration is based on a query-to-hardware compiler for complex event patterns that we presented at VLDB 2010 [1]. By example of a *click stream monitoring* application, we illustrate the inner workings of our compiler and indicate how FPGAs can act as efficient and reliable processors for event streams.

I. Introduction

Today, an increasing need for low latency *complex event processing* is seen in various time-critical network stream monitoring tasks such as *real-time risk checking* in financial trading applications, *security auditing* in web service applications, or *intrusion detection* in Internet-based applications. All of these tasks come with high demands:

(a) A stream monitor must handle any input data at full *wire speed* in *real time*. It must be *robust* even under malicious conditions such as network flooding.
(b) The monitor must *not interfere* with any of the systems being monitored. Ideally, no existing system needs to be altered or invaded to perform stream monitoring.
(c) An easy-to-use and yet sufficiently expressive high-level query language should allow system administrators to configure the stream monitor to their needs. For instance, *regular expressions* are an elegant and familiar way to describe higher-level *complex events* over a number of *basic events*.

Together, these demands often hit the limits of software-based stream monitoring. It is known, for example, that high network packet rates (as they are common in financial trading applications) can quickly thrash the network stack of commodity systems [2]. A hard-wired hardware solution, on the other hand, is too rigid and does not provide the necessary programmability required by most of the applications mentioned above.

In earlier work [2], we showed that *field-programmable gate arrays* (FPGAs), user-programmable hardware chips, can offer significant advantages in similar problem scenarios, either as standalone devices or as part of a heterogeneous multi-core architecture. In this demonstration, we extend our earlier work and spotlight a fully functional stream monitor that detects *complex event patterns* in real time on a Gigabit network link.

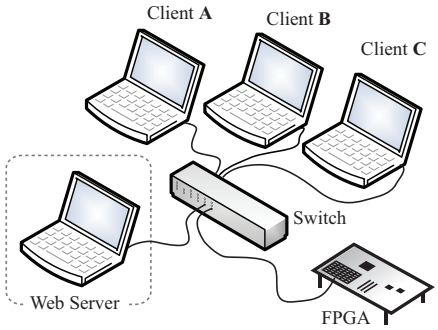

Fig. 1. Demonstration setup.

II. Demonstration

Our demonstration is based on the compiler that we presented at VLDB in [1]. The compiler translates declarative queries for *complex event patterns* into executable hardware circuits that can then be loaded onto the FPGA chip.

The compiler itself is a generic infrastructure for stream monitoring tasks. For this demonstration, we use it to implement a real-time *click stream monitoring system*, which analyzes the request stream of an Internet web server. The use case includes all necessary ingredients to demonstrate the inner workings of our compiler while allowing for active participation of demonstration visitors at the same time.

The demonstration setup is illustrated in Figure 1. We bring our own web server, installed on a laptop machine, and few additional laptops that act as clients to the web server—demonstration visitors can also connect their own laptops to our switch or access the server through WLAN. Our FPGA-based stream monitor is connected to the same network and can eavesdrop on the server traffic.[1] The FPGA device monitors the server traffic and detects multiple user-defined click stream patterns on a per-user basis concurrently.

A. Application Scenario

To illustrate complex event detection, we assume a multi-step web form as shown in Figure 2. The form consists of three sub-forms `F1`, `F2`, and `F3` and a final confirmation form `C`. Form `F2` is optional and can be skipped by the user. Once the user has confirmed his/her transaction, he/she will be directed to either of the three thank-you pages `T1` through `T3`. Such forms are commonly used, *e.g.*, for online shopping, flight

[1]Our network switch is configured to mirror the web server port to the FPGA for that purpose.

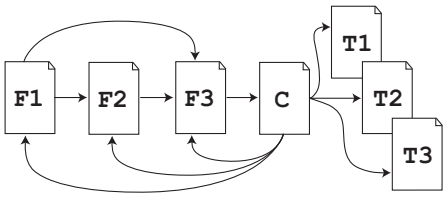

Fig. 2. Multi-step web form.

booking, database conference registration, to name but a few examples.

Here we are interested in detecting specific client behavior. In particular, we want to be informed whenever a *sequence* of *basic events* (page requests) matches a user-defined *event pattern*. If so, we raise a *complex event* with a higher-level meaning to the user who formulated the pattern.

B. Complex Event Patterns

As running examples, we consider the following three complex event patterns:

`direct-buy` Our ideal customer clicks through the sub-forms once and in proper order, then commits his/her transaction and sees any of the three thank-you pages.

`indirect-buy` Before confirming their order, some customers decide to make changes to their form entries and thus move back and forth between the `F␣` and `C` pages before finally placing their order. Frequent occurrences of this pattern may indicate problems in the design of the web form and its user interaction.

`aborted-buy` After visiting any of the `F␣` and `C` pages, some customers navigate away without placing an order (*i.e.*, they access some outside page `O` before reaching any of the thank-you pages). Again, such events may indicate design weaknesses of the web application.

All three event patterns need to be tracked concurrently on a per-user basis and without slowing down the web application itself. In Section III, we will show how these patterns can be described in a declarative manner.

C. Hardware Circuit Generation

In a nutshell, the main tasks of the FPGA-based stream monitor boil down to (1) identifying specific page requests, *i.e.*, basic events, (2) assigning those page requests to the appropriate sub-streams, and finally (3) detecting complex event patterns over the respective sub-streams, *i.e.*, on a per-user basis.

Tasks 1 and 3 rely heavily on *regular expression* pattern matching. In [1], we elaborated how regular expressions can be implemented on FPGAs using *finite state automata*. Most notably, *non-deterministic* automata (NFAs) operate as efficient as *deterministic* automata (DFAs) in FPGA hardware. However, the lower chip space requirements of non-deterministic automata are significant, thus making them the preferred design choice for FPGA-based implementations.

```
1  SELECT       S.src-ip
2  FROM         InputStream S
3  PARTITION BY S.src-ip
4  PATTERN direct-buy   (F1 F2? F3 C T)
5  PATTERN indirect-buy (F1 F2? F3 C ([F1-F3]+ C)+ T)
6  PATTERN aborted-buy  ([F1-C]+ O)
7  DEFINE
8   F1 AS (S.data=/(GET|POST) \/form1\.html/)
9   F2 AS (S.data=/(GET|POST) \/form2\.html/)
10  F3 AS (S.data=/(GET|POST) \/form3\.html/)
11  C  AS (S.data=/(GET|POST) \/confirm\.html/)
12  T  AS (S.data=/(GET|POST) \/thanks[1-3]\.html/)
13  O  AS (S.data=/(GET|POST) \/[^.]*\.html.*HTTP/)
```

Listing 1. Complex event pattern query.

We also support important functionality that goes beyond regular expression matching. The per-user tracking that our scenario is built on (Task 2) raises new challenges in FPGA designs whenever hard throughput guarantees have to be met. In [1], we showed how user tracking (or *partitioning* in the generic stream processing sense) can be realized in an FPGA in a scalable manner by means of a *pipelining* strategy.

While we refer to [1] for details on both, efficient regular expression matching and stream partitioning with FPGAs, our demonstration will illustrate the benefits of the design choices we have made. To this end, we let conference attendees compile their own complex event pattern queries into FPGA circuits. Our compiler is equipped with various hooks to inspect its inner workings. Demonstration visitors can also follow the full design flow to generate a hardware circuit, including gate-level views or FPGA chip space occupation.

III. Declarative Complex Event Queries

A major advancement of our work is that users can specify their event patterns of interest in a high-level, declarative language, yet can benefit from the speed and performance guarantees of query execution directly in hardware.

A. Query Language

The complex event query language of our system is inspired by an ongoing effort to extend SQL with pattern matching capabilities [3]. Listing 1 depicts the code necessary to detect all three event patterns that we motivated in Section II-B.

In this listing, lines 7–13 use a PCRE[2]-style regular expression syntax to detect basic events from the low-level network packet payload (`S.data`). The regular expressions will match HTTP requests (using either the GET or POST methods) for the three form pages `F1`–`F3`, the confirmation form `C`, or any of the three thank-you pages. Notice that we generate a single basic event `T` for requests to all three thank-you pages since their distinction is of no importance for the complex event patterns specified. The last basic event `O` (line 13) will catch any other request for an HTML file that is not covered by the five preceding regular expressions—and thus indicates a navigation to an outside page as needed to answer the complex event pattern `aborted-buy`.

[2]Perl Compatible Regular Expressions

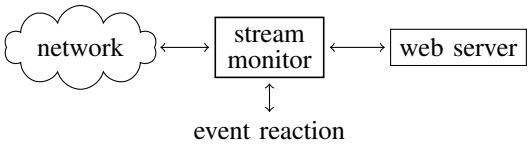

Fig. 3. High-level stream monitoring architecture.

Based on these six basic events, three high-level, complex events can be expressed as shown in lines 4–6. Again, we use familiar regular expression syntax, this time operating over the defined basic events rather than on raw packet payloads.

The `PARTITION BY` clause on line 3 requests user tracking based on source IP addresses, *i.e.*, the complex event must consist of basic events that were generated from the same source IP address.

Finally, the `SELECT` clause on line 1 specifies what we want to report when any of the three complex event patterns match—in this case, the source IP address of the user that caused the complex event (along with an implicit identifier of the corresponding pattern that matched).

B. Query Compilation

For the query in Listing 1, our compiler generates three different hardware components. First, the basic event detection specified by the `DEFINE` part of the query corresponds to the *predicate decoder* unit of [1], which explicitly materializes basic events for downstream processing. For each regular expression an individual finite state machine operates in parallel on the input data. Secondly, `PATTERN` clauses will be compiled into state automata as well, but they operate on basic events rather than on raw bytes. Finally, the `PARTITION BY` clause produces a *stream partitioner* component detailed in [1]. In brief, this component finds the current complex event pattern matching state of a given sub-stream based on the attributes defined in the `PARTITION BY` clause.

IV. Hardware Architecture

On a high level, a general architecture for *non-invasive*, *real-time* stream monitoring is illustrated in Figure 3. The monitor intercepts all traffic sent to/from the web server and reacts whenever a specified event is detected.

Since the FPGA device that we can bring to the conference site is equipped only with a single network port, we mimic network interception with help of a hardware switch and use a physical network setup as shown earlier in Figure 1.

For ease of demonstration, our system reacts to detected complex events simply by showing a message on a built-in LCD display on the FPGA board. Applied to a real-world scenario, our system could just as well send notifications to an outside system, alter network packets as they pass through the monitor (ensuring compliance with predefined rules), or drop packets from the network stream.

A. Inner Hardware Design

Figure 4 illustrates how complex event detection is performed inside the FPGA chip. Our custom-made logic is

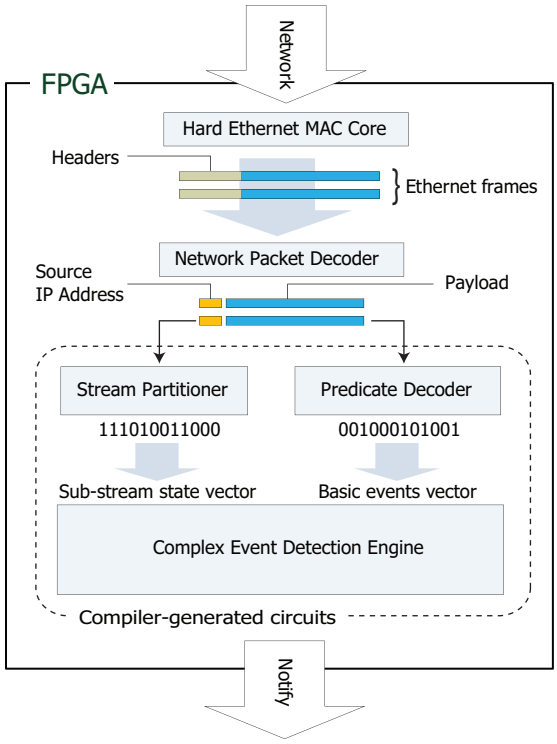

Fig. 4. FPGA-based pattern matching over network streams.

directly connected to the hard Ethernet MAC core of the physical network interface so that we can achieve full wire-speed throughput performance.

Once network packets enter our logic (top of Figure 4), they are routed into a *network packet decoding* component that takes care of processing the raw Ethernet frames. Its main task is to properly identify TCP packets and extract the source IP address and the payload from them. Packet decoding is implemented using a hard-coded state automaton.

The extracted payload is then forwarded to a *predicate decoding* unit. This unit contains a hardware implementation for each regular expression listed in the `DEFINE` part of the user query. They will all be run in parallel over the payload stream and a basic event will be generated for the first matching regular expression. In the web application scenario, a basic event corresponds to a "click" event by the web user.

The *stream partitioner* component is only instantiated if the query requests per-user tracking (via the `PARTITION BY` clause). The component is executed concurrently with the *predicate decoder* and separates the individual web user streams. For each web user a *state vector* is maintained to keep track of his/her click history. The *stream partitioner* selects the appropriate state vector (or allocates a new one if necessary) and forwards it to the final *complex event detection* stage.

A *complex event detection* unit (an NFA) is generated for every complex event defined by a `PATTERN` clause in the user query. Again, our system exploits the available hardware parallelism and evaluates all state automata concurrently. Upon a match, a complex event is raised—the event that is of actual

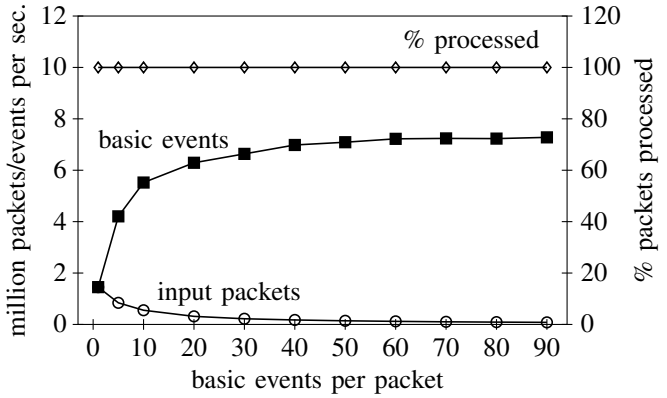

Fig. 5. Performance of FPGA stream monitor. We saturate the link with packets of varying size. Independent of the packet size, the FPGA will process 100 % of the input stream and never drop packets.

interest to the user. In our particular demonstration setup, we react to these complex events by showing a notification on the built-in LCD display indicating the pattern that was matched and listing the IP address of the user who caused the match.

B. Performance

Our earlier work [1] contains a detailed study on hardware-based complex event detection. Here we emphasize the advantages in the context of network-attached stream monitoring.

The hardware circuits generated by our compiler are designed to consume arbitrary input data at full wire speed. To verify this capability, we generated event streams where a varying number of basic events occur in each packet.[3] We saturated our 1 Gb/s network link with the generated stream and measured how much of it we could monitor with our FPGA device.

As can be seen in Figure 5, our device could monitor all input data at full wire speed, without any packet drops that in software can be caused by overload situations. This is particularly remarkable on the left end of the scale. Workloads that use many small-sized packets (as common, *e.g.*, in financial trading applications) are known to cause very high CPU load and often substantial latencies in software-based systems [4], [2], even though less payload data is actually sent over the wire (due to the increasing IP protocol overhead). Our system, by contrast, is fully robust to such workloads and is guaranteed to always run at full wire speed.

C. Related Work

Our system has obvious similarities to hardware-accelerated intrusion detection systems, such as [5], [6], [7]. But while these existing systems perform single packet inspection only, our design operates on logical streams and features explicit user tracking.

Software-based complex event processors (*e.g.*, [8], [9]) do support such functionality, but lack the integration (in-network processing) and performance (full wire-speed throughput) advantages of our hardware-accelerated device. For very complex analysis tasks (where a direct hardware implementation is no longer feasible) we could envision a hybrid setup where the FPGA acts as a pre-processor and filter to a back-end stream processor in software (in the spirit of [2]).

[3]For these experiments UDP packets were used.

V. Summary

In this demonstration we show a complete implementation of an FPGA-based *complex event processing system*. Our system monitors network streams in a *non-invasive* manner, allowing reliable stream analysis where speed is a concern, *e.g.*, in online trading, security auditing, or intrusion detection applications. On the usability side, our system features an expressive pattern description language allowing users to declaratively state high-level complex events of interest.

For illustration purposes, we showcase a scenario where demonstration visitors can actively participate: we monitor page requests sent to a web server that we provide and react to user-defined complex event patterns. Our expression compiler and the FPGA tool chain is equipped with numerous hooks to inspect the inner workings of our system, all the way from complex event pattern compilation down to the circuit-level design.

Acknowledgements

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

References

[1] L. Woods, J. Teubner, and G. Alonso, "Complex Event Detection at Wire Speed with FPGAs," *Proc. of the VLDB Endowment (PVLDB)*, vol. 3, no. 1, Sept. 2010.

[2] R. Mueller, J. Teubner, and G. Alonso, "Streams on Wires—A Query Compiler for FPGAs," *Proc. of the VLDB Endowment (PVLDB)*, vol. 2, no. 1, Aug. 2009.

[3] J. Melton, "SQL/RPR—Row Pattern Recognition with Application to Streaming Data Queries," INCITS Project Proposal H2-2008-027, http://www.softwareworkshop.com/h2/SQL-RPR-review-paper.pdf (retrieved July 2010).

[4] M. Dobrescu, N. Egi, K. Argyraki, B.-G. Chun, K. Fall, G. Iannaccone, A. Nies, M. Manesh, and S. Ratnasamy, "RouteBricks: Exploiting Parallelism to Scale Software Routers," in *Proc. of the 22nd ACM Symposium on Operating Systems Principles (SOSP)*, Big Sky, MT, USA, Oct. 2009.

[5] A. Mitra, W. Najjar, and L. Bhuyan, "Compiling PCRE to FPGA for Accelerating SNORT IDS," in *Proc. of the ACM/IEEE Symposium on Architecture for Networking and Communication Systems (ANCS)*, New York, NY, USA, Dec. 2007.

[6] R. Sidhu and V. Prasanna, "Fast Regular Expression Matching Using FPGAs," in *IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM)*, Rohnert Park, CA, USA, 2001.

[7] Y.-H. Yang, W. Jiang, and V. Prasanna, "Compact Architecture for High-Throughput Regular Expression Matching on FPGA," in *Proc. of the ACM/IEEE Symposium on Architecture for Networking and Communication Systems (ANCS)*, San Jose, CA, USA, Nov. 2008.

[8] N. Dindar, B. Güç, P. Lau, A. Ozal, M. Soner, and N. Tatbul, "DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events," in *Proc. of the ACM SIGMOD Conference on Management of Data*, Providence, RI, USA, 2009.

[9] D. Gyllstrom, E. Wu, H.-J. Chae, Y. Diao, P. Stahlberg, and G. Anderson, "SASE: Complex Event Processing over Streams," in *3rd Conference on Innovative Data Systems Research (CIDR)*, Asilomar, CA, USA, Jan. 2007.