Click Stream Pattern Matching with FPGAs

Louis Woods Jens Teubner Gustavo Alonso
Systems Group, Department of Computer Science
ETH Zurich, Switzerland
{louis.woods,jens.teubner,gustavo.alonso}@inf.ethz.ch

ABSTRACT

Complex event processing (CEP) is becoming an integral part of traditional stream processing. To cope with constantly growing data volumes and an increasing need for fast results, new technologies have to be explored. In this demonstration we present a hardware based *complex event detection* system implemented on a *field-programmable gate array* (FPGA). The FPGA is connected directly to the network and monitors incoming web server traffic detecting *click stream patterns* in real-time. As part of the demonstration, the process of transforming *complex event patterns* into hardware circuits will also be illustrated. This involves compiling click patterns to a hardware description language (VHDL) with our custom-built *query compiler*, as well as configuring the FPGA with the generated circuits.

1. BACKGROUND

Complex event detection is gaining significant importance in current stream processing systems [1, 2, 3, 4]. A *complex event* is a meaningful higher-level event derived from a number of low-level (basic) events. *Complex event patterns* are expressed with regular expressions [7], and multiple patterns typically need to be concurrently evaluated on one or several input streams. Commodity systems have problems with this task for various reasons. On the one hand, processing many patterns simultaneously over high-rate data streams will lead to a bottleneck on sequential systems. On the other hand, commodity systems have the problem that the data needs to be brought from the source of the stream, *e.g.*, the network interface, to the CPU via main memory before the CPU can process the data leading to another bottleneck [5]. As a solution to these problems, we propose to use *field-programmable gate arrays* (FPGAs) and to perform complex event detection directly in hardware. We thereby exploit the inherent parallelism of FPGAs to evaluate multiple complex event pattern concurrently, and by connecting the FPGA directly to the network we can avoid the *network-memory-bottleneck*.

2. DEMONSTRATION

The *complex events* that we will detect with the help of an FPGA correspond to *click stream patterns*, *i.e.*, the complex events really are sequences of page views of a given website. This use case has several interesting properties. First of all, the FPGA detects click patterns in *real-time*, at *wire speed*, and in an *non-invasive* manner, *e.g.*, the existing web application does not need to be altered, no server logs need to be processed, etc. Secondly, this use case allows us to demonstrate the usefulness of FPGAs for many different tasks. TCP packets will be decoded directly on the FPGA and their payloads will be scanned in order to detect specific HTTP requests. We use regular expressions in hardware to match HTTP requests. Therefore, multiple regular expressions have to be evaluated on every network packet in parallel. Finally, matching HTTP requests need to trigger events so that the complex event detection engine, which is also on the FPGA, can detect the *complex events.*

2.1 Stream Partitioning

When multiple users concurrently are browsing a website this generates a single interleaved stream of page views. However, it does not make sense to detect patterns on the accumulated click stream of all users. Therefore, the stream of page views needs to be partitioned on the FPGA so that click patterns can be detected on a per-user (per-client) basis. Our solution supports sub-stream pattern matching and we will show this feature in the demonstration.

2.2 Query Compilation

For the purpose of specifying complex events (click stream patterns), we present a declarative query language. This language, which we further describe in Section 3, is inspired by a recent standardization effort [7] to extend SQL with pattern matching capabilities. We have developed a compiler that translates queries specified in our query language to VHDL code. VHDL is a hardware description language that can be used to configure an FPGA. Conference attendees will be able to compile custom queries to VHDL and the generated state machines will be illustrated through a *state machine visualizer* built into the compiler.

2.3 Demo Setup

In this demonstration a small private network will be setup (Figure 1). We connect a web server with a number of laptops (clients) via a *managed* switch. We have configured the switch to mirror the port of the web server. The FPGA, programmed with the complex event detection engine, is connected to the *mirrored port*. This way, the FPGA can eavesdrop on the entire traffic between web server and clients. On the FPGA the Ethernet frames are decoded and potential TCP payloads are processed by the circuits of the pattern matching engine. The FPGA is pre-configured with click patterns of particular interest. When the FPGA detects a given click pattern, it will report this (complex) event on its LCD display. Visitors can use one of our laptops or connect their own laptop to the switch and see how the FPGA detects patterns that they generate by accessing the web server.

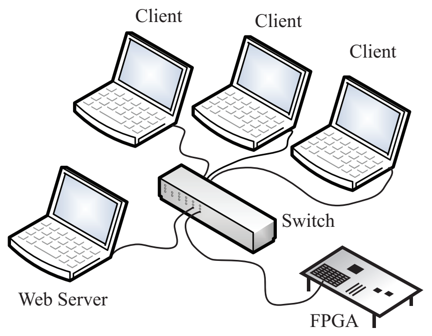

Figure 1: Demonstration Setup

Multiple complex event patterns are concurrently evaluated on the FPGA for every network packet sent to the web server. The patterns are detected on a per-client basis. That is, multiple visitors can generate page views on our web server simultaneously and patterns will be detected for each user individually. This is possible because the stream of clicks is partitioned by a special *Stream Partitioner* component (see Section 5).

3. CLICK STREAM PATTERN EXAMPLES

In this section we want to give some idea of the kinds of click stream patterns that can be detected by the circuits of our complex event detection system. As an example we will consider a multilevel web form such as are typically seen in online flight reservation systems. A schematic representation of the web form is depicted in Figure 2.

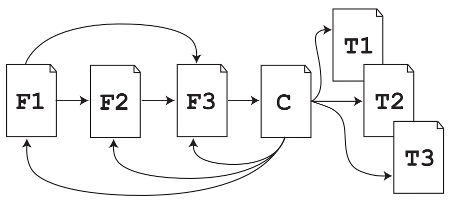

Figure 2: Multilevel Web Form

This form consists of three subforms, which are all on separate web pages. Depending on the options selected by the user at the first level (`F1`), the second level (`F2`) will be skipped. After the user has completed the third level (`F3`), a confirmation-page (`C`) is displayed. From the confirmation-page the user can go back to any of the previous levels and modify his data. Once the user clicks the confirm-button on the confirmation-page, one of three different thankyou-pages (`T1-T3`) will be displayed.

3.1 Detecting Page Clicks

A *complex event* in this application is a sequence of clicks corresponding to a given click pattern. The *basic* events are the individual page clicks. In this section we discuss how we can define the basic events of interest in the context of a declarative query language. We specify basic events with the help of *predicates*. These predicates can be declared in a special `DEFINE` clause of a complex event query. In this case a predicate is a specific condition that is evaluated for every incoming network packet payload. When the predicate is satisfied, this generates a basic event for the complex event detection engine. In Listing 1 a `DEFINE` clause is listed for the example web application illustrated in Figure 2.

```
1 DEFINE
2  F1 AS (Payload = /(GET|POST) \/form1\.html/)
3  F2 AS (Payload = /(GET|POST) \/form2\.html/)
4  F3 AS (Payload = /(GET|POST) \/form3\.html/)
5  C  AS (Payload = /(GET|POST) \/confirm\.html/)
6  T  AS (Payload = /(GET|POST) \/thankyou[1-3]\.html/)
7  O  AS (Payload = /(GET|POST) \/[^.]*\.html.*HTTP/)
```

Listing 1: Predicate Declarations

The predicates are labeled (`F1`, `F2`, etc.), so that they can be used in click pattern declarations. After the `AS` keyword follows the actual definition of the respective predicate. Here we specify predicates with the help of regular expressions. We use a Perl-like syntax, denoting regular expressions by enclosing them in two slashes.

We are interested in TCP payloads that start either with "GET" or "POST", according to the HTTP protocol. For example, clicking the submit-button on one of the subforms would trigger a POST request, whereas the links on the confirmation-page, taking the user back to one of the subforms, would produce GET requests.

Since it will be irrelevant in our patterns which one of the three thankyou-pages is displayed to the user, we can declare a single predicate representing all three thankyou-pages. On line six in Listing 1, we do exactly this with the help of a *character class* declaration (`[1-3]`).

The last predicate (`O`), on line seven in Listing 1, is supposed to catch any other GET or POST request issued by the user which is not covered by the five previously declared predicates. By appending `.*HTTP`[1] to the regular expression we ensure that this predicate is only satisfied when none of the previously defined predicates have been satisfied. This is because (in this case) click detection is terminated as soon as the first predicate is satisfied. Specifying this predicate is very important because the sequence `F1 F2 F3` has a different meaning than, say, the sequence `F1 F2 O F3`. If we had not specified predicate `O` these two sequences would be equivalent to the complex event detection engine because it would never 'know' about the extra click that happened between `F2` and `F3`.

3.2 Complex Event Patterns

With the predicates declared, it is now time to define some patterns over these predicates. Three example patterns are illustrated in Listing 2. The patterns are explained below.

[1] In HTTP version 1.0 and 1.1 GET and POST requests are followed by the version of the HTTP protocol, *i.e.*, "HTTP/1.0". Here we do not support HTTP 0.9, where this is not the case.

```
PATTERN pattern1 (F1 F2? F3 C T)
PATTERN pattern2 (F1 F2? F3 C ([F1-F3]+ C)+ T)
PATTERN pattern3 ([F1-C]+ O)
```

Listing 2: Pattern Declarations

pattern1 A user clicks through form one, two (optional) and three without interruption, *i.e.*, without browsing to a different page in between. On the confirmation-page the user clicks the confirm-button without going back to make any changes to his data.

pattern2 This time, the user at the confirmation page goes back at least one time to one of the subforms, before confirming. Notice how we use the range operator in the symbol class (`[F1-F3]`). Which predicates are included in the range depends on the order in which the predicates are declared, *i.e.*, in this case `[F1-F3]` is equivalent to `[F1 F2 F3]`.

pattern3 The last pattern detects users that start filling out the web form but then leave and request some other page (`O`) on this website.

4. PATTERN TO VHDL COMPILATION

As can be seen in the complex event queries that we have presented previously, we use regular expressions in two ways. First, character based regular expressions are used to detect certain HTTP requests on the bytes of the network packets. Second, regular expressions are used also for specifying the click stream patterns. On the FPGA, both regular expression types are implemented as finite automata. In hardware, as opposed to software, *deterministic* finite automata (DFA) are not more efficient than *non-deterministic* finite automata (NFA) [6]. In contrast, NFAs typically have more modest chip space requirements than DFAs.

Our query compiler can translate complex event patterns, specified in the query language illustrated above, to VHDL code. The VHDL code defines the necessary NFAs, a *Predicate Decoder* component and a *Stream Partitioner* component. At the demonstration, we will show how our compiler generates VHDL code from sample queries. Unfortunately, *synthesizing* the VHDL code, *placing and routing* the design, and *configuring* the FPGA on-site using a standard FPGA tool chain can take some time, *e.g.*, ten to fifteen minutes. Nevertheless, we can configure the FPGA with pre-generated bitstreams which is significantly faster and also demonstrates the process.

5. SYSTEM ARCHITECTURE

The aforementioned compiler takes a set of queries and generates a complex event detection system in hardware that can be run on an FPGA. In this section we give a high-level overview of the key components that this system consists of. As mentioned earlier, our system is connected directly to the Ethernet MAC component of the physical network interface so that we can achieve full wire speed throughput performance. Figure 3 depicts the placement of the FPGA in the data path. Typically this would be between network interface and CPU. For demonstration purposes, we will not actually notify a CPU but rather display pattern matches on the LCD display of the FPGA development board.

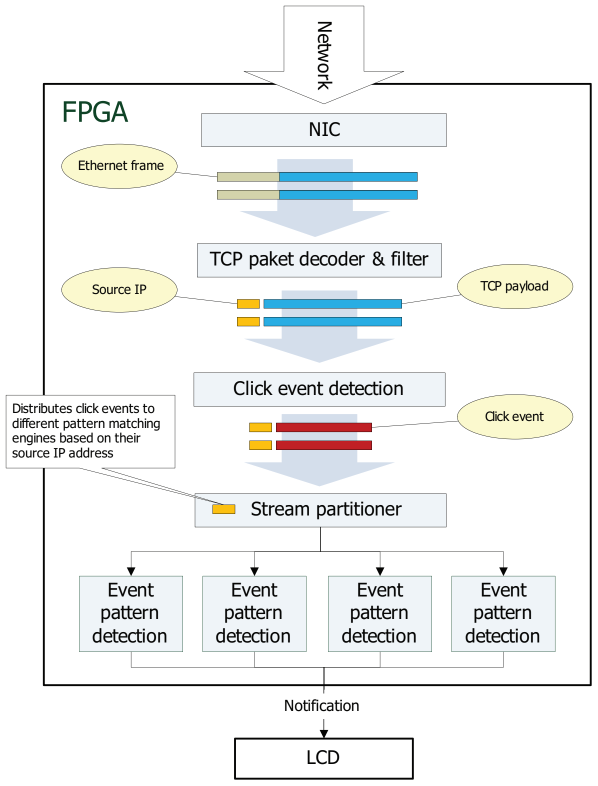

Figure 3: FPGA Placed in Data Path

On the FPGA we have implemented a *Network Packet Decoding* component that takes care of processing the raw Ethernet frames. Its main task is to properly unpack the payloads of the network packets, but it can also act as a filter by dropping packets, *e.g.*, based on an invalid IP address in the IP header or an invalid port in the TCP header.

The payload of the network packets addressed to our web server are forwarded to the next hardware component in our system, the *Click Event Detection* component. This component evaluates multiple regular expressions concurrently on the data of the payload. Each regular expression corresponds to a different kind of HTTP request.

When the *Click Event Detection* component matches an HTTP request of interest, it generates a (basic) event and forwards that event to the *Stream Partitioner* component. This component has to make sure that the event is processed by the appropriate finite state automaton. If we want to be able to detect sub-stream patterns, we need a separate finite state machine for every sub-stream. The *Stream Partitioner* determines the appropriate finite state machine and forwards the event to it.

This state machine then does the actual complex pattern detection and in case of a match the LCD component is informed and the match is reported on the LCD display. The LCD display has two display lines. On the top line the pattern that matched will be displayed and on the bottom line the IP address of the user that caused the pattern match. For illustration purposes, we will not allow overlapping matches of different patterns. However, in general overlapping patterns are not a problem since different patterns are evaluated by different hardware components.

6. SUMMARY

In this demonstration we show how an FPGA can be enabled to monitor data streams directly from the network in an *non-invasive* manner. Conference attendees can connect their own laptops or use one of our laptops to generate HTTP requests on a web server and see how their access patterns are recognized by the FPGA. Multiple users can evaluate the system concurrently and the patterns will be detected on a per-user basis. The matches will be reported on the LCD display together with the IP address of the user that caused the pattern match.

Furthermore, we present our custom-built query compiler showing how complex event patterns can be specified in a declarative query language and how these queries are translated to VHDL code. For interested visitors we can also synthesize custom queries and actually configure the FPGA with these queries so that visitors can learn the entire life cycle of an FPGA based query processor.

7. REFERENCES

[1] J. Agrawal, Y. Diao, D. Gyllstrom, and N. Immerman. Efficient pattern matching over event streams. In *SIGMOD '08*, New York, NY, USA, 2008.

[2] N. Dindar, B. Güç, P. Lau, A. Özaland M. Soner, and N. Tatbul. DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events (Demonstration). In *SIGMOD'09*, Providence, RI, USA, 2009.

[3] ESPER. `http://esper.codehaus.org/`.

[4] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: Complex Event Processing over Streams (Demo). In *CIDR'07*, Asilomar, CA, USA, 2007.

[5] R. Müller, J. Teubner, and G. Alonso. Streams on Wires - A Query Compiler for FPGAs. In *VLDB'09*, Lyon, France, 2009.

[6] Y. Yang, W. Jiang, and V. Prasanna. Compact architecture for high-throughput regular expression matching on FPGA. In *ANCS'08*, San Jose, California, USA, 2008.

[7] F. Zemke, A. Witkowski, M. Cherniack, and L. Colby. Pattern Matching in Sequences of Rows. In *Technical Report ANSI Standard Proposal*, 2007.