Complex Event Detection at Wire Speed 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 detection is an advanced form of data stream processing where the stream(s) are scrutinized to identify given event patterns. The challenge for many *complex event processing* (CEP) systems is to be able to evaluate event patterns on high-volume data streams while adhering to real-time constraints. To solve this problem, in this paper we present a hardware based complex event detection system implemented on *field-programmable gate arrays* (FPGAs). By inserting the FPGA directly into the data path between the network interface and the CPU, our solution can detect complex events at gigabit wire speed with constant and fully predictable latency, independently of network load, packet size or data distribution. This is a significant improvement over CPU based systems and an architectural approach that opens up interesting opportunities for hybrid stream engines that combine the flexibility of the CPU with the parallelism and processing power of FPGAs.

1. INTRODUCTION

An increasing number of applications in areas such as finance, network surveillance, supply chain management or healthcare are confronted with the need to process high-volume event streams in real-time [7]. Typically, the individual data items (tuples) in those streams only become meaningful when put into context with other items of the same stream. Therefore, *complex event processing* (CEP) has constantly been gaining momentum in both industry and academia [1] since it aims at inferring meaningful higher-level events (complex events) from a sequence of low-level events.

Existing complex event detection engines face the problem that the data items of the event stream first need to be brought to the CPU via main memory before the CPU can start processing them. For instance, when events are coming from the network they are typically wrapped in small UDP packets. Once the packet rate exceeds a certain threshold, CPU based systems are no longer able to sustain the network load and start dropping UDP packets [11].

In this paper, we propose to move the complex event detection engine as close as possible to the origin of the event stream (in this case to the network interface) so that the network-memory-CPU bottleneck is avoided.

We have implemented our solution using a *field-programmable gate array* (FPGA). FPGAs are chips that host *configurable logic* and provide a substrate to implement arbitrary (data processing) functionality in hardware. Our system decodes network packets directly on the FPGA and can handle any packet size equally well at the highest frequency that a gigabit Ethernet link allows. Besides benefiting from data proximity our system heavily exploits the inherent parallelism that FPGAs can offer. For example, separate hardware components can concurrently deal with Ethernet frame decoding, complex event detection, and stream partitioning. Like existing FPGA solutions for, *e.g.*, deep packet inspection [10] in network intrusion detection systems such as SNORT [13] and BRO [2], our solution uses *regular expressions* to define the complex events of interest.

Contributions. Our main contribution is a high-throughput complex event detection system, which is based on regular expression matching in hardware. It can be attached directly to the network interface and thereby fully operate at gigabit wire speed while guaranteeing constant latencies. The design of an FPGA based solution is non-trivial. The bottleneck in the FPGA is the real estate on the chip. Unlike software based solutions, we cannot use *deterministic* finite automata (DFA) because of the exponential *state explosion.* Instead, we need to use *non-deterministic* finite automata (NFA).

In the paper we show how complex event patterns can be expressed in a declarative query language (Section 2) (based on a recent standardization effort [17]) to extend SQL with pattern matching capabilities. In the process, we identify key differences between classical regular expressions and complex event patterns.

We give a detailed description of how to compile complex event patterns specified in the aforementioned language into actual hardware circuits and compare different implementation approaches (Section 4). Furthermore, we present a hardware solution for the partitioning of streams to support sub-stream pattern matching (Section 6).

The results in each section are backed by experiments on actual hardware using prototype implementations and our complete system is evaluated in Section 7. We conclude the paper with a survey of related work (Section 8) and a summary of the results in Section 9.

2. COMPLEX EVENT PATTERNS

In this section we further illustrate the idea behind complex event detection and introduce a query language for defining complex event patterns. Our language closely resembles parts of the `MATCH-RECOGNIZE` clause of the current ANSI draft for a SQL pattern matching extension [17]. Nevertheless, since our focus is on Boolean, regular expression-based complex event detection, we have derived a simplified and less verbose language from this draft.

2.1 Regular Expression Patterns

To demonstrate the query language and show how it can be used to define complex events, we take the New York marathon as an example. Assume the runners need to pass an electronic checkpoint in each of the five boroughs Staten Island (`A`), Brooklyn (`B`), Queens (`C`), the Bronx (`D`) and Manhattan (`E`). While there is nothing wrong with a runner passing any single of the checkpoints, an incorrect order of passing them may indicate cheating. The expression

`A (A|C|D|E)*C | A (A|B|D|E)*D | A (A|B|C|E)*E`,

for instance, could be used to describe the complex event where a runner reached one of the checkpoints `C`, `D`, or `E` (from start point `A`), but has not passed the respective predecessor `B`, `C`, or `D`.

The appeal of regular expressions comes from the fact that they provide sufficient expressiveness for most real-world use cases, yet they can be implemented as *finite state automata*, which can ensure the necessary (constant) space and latency guarantees. In this work we use the dialect shown in Table 1 to describe regular expressions over events that we denote with identifiers in capital letters.

`A`, `NOTA`, `B` ...	event
`.`	wildcard: any event
(r)	grouping: bypass default binding
r`*`	closure: zero or more repetitions of r
$r_1\ r_2$	sequence: r_1 followed by r_2
r_1 | r_2	choice: either r_1 or r_2

Table 1: Regular expression dialect. Quantifier `*` binds stronger than the sequence operator, which binds stronger than choice `|`.

Notice that many other commonly known regular expression operators are pure syntactic sugar, *e.g.*, r+ is just a short-hand for rr*. Hence, we do not consider them here any further.

2.2 Tuples, Predicates and Events

Regular expression engines for text typically operate over an alphabet of 8-bit characters, *i.e.*, at most 256 different characters. Stream processors, by contrast, react to events that may be triggered by large input tuples. In our hypothetical marathon scenario, readers at the five checkpoints might produce a tuple stream `marathon` of schema:

⟨time : timestamp, checkpoint : string,
runner : int, speed : float⟩

The size of the corresponding alphabet, the *value domain*, of such a stream can be enormous and explicit value enumeration clearly is not feasible. Value ranges that may appear inside a complex event pattern are thus described as *predicates* over input tuples. A predicate is a condition that each incoming tuple either satisfies or does not. A *basic event* in a complex event pattern is the equivalent to a satisfied predicate. For instance, our earlier pattern for cheating in a marathon might read:

```
PATTERN ( A NOTB* C | A NOTC* D | A NOTD* E )
DEFINE
   A    AS (checkpoint  = 'Staten Island')
   NOTB AS (checkpoint != 'Brooklyn')
   C    AS (checkpoint  = 'Queens')
   NOTC AS (checkpoint != 'Queens')
   D    AS (checkpoint  = 'Bronx')
   NOTD AS (checkpoint != 'Bronx')
   E    AS (checkpoint  = 'Manhattan')
```

This query consists of a `PATTERN` clause and a `DEFINE` clause. In the `PATTERN` clause the complex event is specified using regular expression operators and predicate identifiers. The predicates are defined in the subsequent `DEFINE` clause. Observe that the absence of checkpoint readings (previously expressed, *e.g.*, as `(A|C|D|E)*`) can be described in a more readable way by using negation (`NOT`x definitions above).

2.3 Stream Pattern Peculiarities

The predicates in our regular expressions are different from the characters in classical regular expressions where a character unambiguously defines some element of the value domain. A predicate can encompass a whole range of values and is thus rather comparable to a *character class*, *e.g.*, `[a-z]` in a classical regular expression. The fundamental difference is that predicates can be satisfied simultaneously by the same input tuple just as overlapping character classes may match the same input character. This has consequences for the finite state machine that implements the regular expression. We will further discuss this issue in Section 4.

In classical regular expressions the alphabet covers the entire value domain. Typically, this is not the case in complex event patterns since only the value ranges of interest are specified in the `DEFINE` clause. Therefore, care needs to be taken that all relevant events are specified as predicates even if they are not part of the regular expression itself. Consider, for example, the pattern `A B C`. If only the predicates `A`, `B` and `C` are specified then the sequence '`ABDC`' of checkpoint readings would match this expression, because the tuples generated from the reader at checkpoint `D` would go unnoticed since they are not captured by any predicate.

2.4 Stream Partitioning

Real-world streams often contain the interleaved union of a number of semantical sub-streams. In our running example, the `marathon` stream contains one sub-stream or *partition* for each participant in the race. When analyzing such streams, patterns become meaningful only *within* each partition. A `PARTITION` clause can be used to divide the `marathon` stream by runner id into multiple sub-streams:

```
PARTITION runner
PATTERN ( A NOTB* C | A NOTC* D | A NOTD* E )
DEFINE
   A    AS (checkpoint = 'Staten Island')
   ...
```

The partitioning attribute (`runner`) thereby determines to which sub-stream the current tuple belongs. A hardware implementation of a partition strategy is given in Section 6.

3. SYSTEM ARCHITECTURE

In this section we give a high-level overview of the complex event detection system that we have developed and of its key components. 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 performance. Figure 1 depicts the placement of the FPGA in the data path between network interface and CPU.

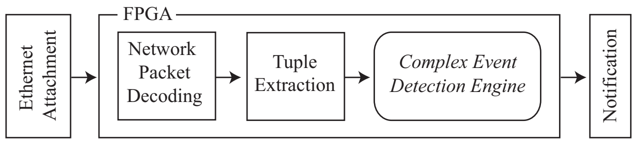


Figure 1: FPGA placed in Data Path

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

As soon as the first payload bytes arrive at the FPGA, the *Network Packet Decoding* component forwards them to the *Tuple Extraction* component, which has knowledge about the tuple schema. Note that a network packet can contain more than one tuple. The job of the *Tuple Extraction* component is to convert the payload bytes into tuples and forward them to the *Complex Event Detection Engine*, which is the actual heart of our system and is illustrated in more detail in Figure 2.

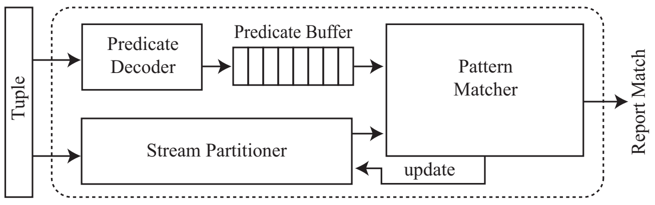


Figure 2: Complex Event Detection Engine

As illustrated in the figure, the *Complex Event Detection Engine* is made up of several sub-components. For each tuple the *Predicate Decoder* evaluates all defined predicates and returns a *predicate vector* that captures which predicates where satisfied by the tuple.

Concurrently, the *Stream Partitioner* retrieves the pattern matching state that corresponds to the sub-stream the current tuple belongs to and returns this information in the form of a *state vector*. As we will further explain in Section 6 the *Stream Partitioner* has longer latency than the *Predicate Decoder*. Therefore, *predicate vectors* need to be buffered by means of a simple FIFO buffer.

Finally, *predicate vector* and *state vector* are both fed to the *Pattern Matcher*. This component is responsible for the actual complex event detection. The *Pattern Matcher* updates the *state vector* and returns it to the *Stream Partitioner*. If a pattern was matched, *i.e.*, a complex event was detected, the end system needs to be informed, *e.g.*, per CPU interrupt.

4. PATTERN MATCHING WITH FPGAS

Pattern matching with regular expressions is a well studied problem. Finite automata are well suited to be implemented on FPGAs. However, the implementation is very different from that used in software systems given that the design constraints are completely different. For instance, since processing many active NFA states is not a bottleneck on a massively parallel platform such as an FPGA, the claim that DFAs are more efficient to execute than NFAs no longer holds. Thinking outside the box helps in the quest for good hardware designs. In this section we show how complex event patterns can be compiled to hardware circuits and we discuss the implications of doing so.

4.1 Finite State Automata

Two important types of finite automata are typically distinguished: *deterministic* finite automata (DFA) and *non-deterministic* finite automata (NFA). While both types provide equal expressiveness, DFAs differ substantially from NFAs in the way they process data. An essential property of DFAs is that at any given point in time only one state is active, *i.e.*, for each input symbol a single state needs to be processed. In contrast, an NFA can have multiple active states at the same time which all need to be processed when the next input symbol is read. Therefore, in software, DFAs tend to run much faster than NFAs which makes DFAs the method of choice in CPU based systems.

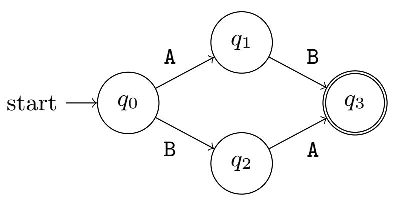
To compile a regular expression into a DFA, the expression is usually mapped to an NFA first, *e.g.*, using Thompson's algorithm [15]. Then the NFA is converted into an equivalent DFA using the *powerset construction* [8]. The powerset construction eliminates non-determinism by inserting new DFA states that incorporate all active NFA states at any one time. As a result DFAs are usually larger than NFAs which can be seen, for example, in the automaton for the expression

$$\texttt{(0|1)*1(0|1)}^{i} \qquad (\star)$$

where $\texttt{(0|1)}^{i}$ denotes an i-fold repetition of subexpression `(0|1)`. Whereas a non-deterministic automaton for this expression can be built with $1+(i+1)$ states, a corresponding DFA requires at least $1+(2^{i+1}-1)$ states. In general, a DFA may require up to 2^n states compared to an equivalent NFA with only n states [8]. The consequences of this phenomenon, known as *state explosion*, can be exceptionally severe for implementations in hardware where resources are more scarce, as we will show in Section 4.4.

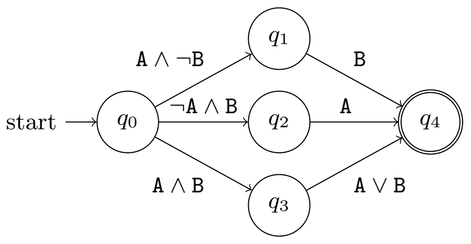
4.2 Overlapping Predicates

With classical regular expressions it depends solely on the regular expression whether NFA → DFA conversion leads to state explosion. Unfortunately, the use of predicates to specify event patterns additionally fosters state explosion in DFAs. This is because predicates can *overlap* and therefore a single tuple might satisfy more than one predicate at the same time. To exemplify, consider the regular expression `AB|BA` (which matches either "`AB`" or "`BA`"). If the predicates for `A` and `B` are mutually exclusive, the corresponding NFA and DFA both look the same:



(1)

For non-exclusive predicates, this automaton would violate the DFA property, since two transitions had to be followed for a tuple that satisfies A and B. To re-establish the DFA property, the overlap has to be made explicit by introducing a new state and additional transitions:



(2)

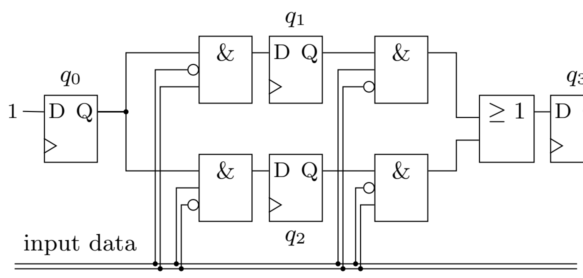
With the possibility of overlapping predicates, the 2^n blow-up in the number of DFA states becomes a realistic problem. Where k target states were sufficient in an NFA to support k independent predicates, $2^k - 1$ target states are needed in a DFA to cover all potential predicate overlaps. In addition, transition conditions will turn into k-way conjunctive predicates — with the corresponding high cost for evaluation.

4.3 NFAs in Hardware

When designing hardware applications on FPGAs area consumption is a key. Very large DFAs are not suitable to be implemented in hardware since they require too much chip space. This makes NFAs an interesting alternative. Even more so, because we can exploit the inherent parallelism of FPGAs to construct NFAs that run equally fast as their DFA counterparts.

An intuitive approach is to map each NFA state to a flip-flop register. Transitions can then be realized using connections in the interconnect network, and conditions turn into combinatorial logic. Effectively, the automaton is laid out in the 2-dimensional FPGA space.

To exemplify this implementation strategy, consider Automaton 1 in Section 4.2 and a 2-bit wide alphabet where $\mathtt{A} \equiv 01$ and $\mathtt{B} \equiv 10$. The corresponding hardware NFA uses four flip-flops to encode states q_0 through q_3:



Transitions between states are conditioned using four 3-way AND gates (we use the IEC symbols in the figure above). To this end, the 2-bit input data is replicated to each of the four gates (–○ indicates negated input). State q_3 has two incoming transitions, which has to be made explicit in terms of the OR gate to its left. In an FPGA, combinatorial logic gates are implemented using lookup tables.

This NFA design maps maps nicely to the architecture of FPGAs. An FPGA consists of a large number of so called *slices*. A slice embodies several LUT-flip-flop pairs (LUT = lookup table), *e.g.*, a Virtex-5 FPGA has four pairs per slice. The slices are connected with each other over an interconnect network. Each state of our NFA requires exactly one flip-flop and in most cases the lookup table of the LUT-flip-flop pair will be sufficient to implement the required combinatorial logic for that state.

4.4 Evaluation : State Explosion

In this section, we investigate the effects of *state explosion* on the FPGA. Circuits are usually specified in some hardware description language such as VHDL or Verilog. This code is then processed by a vendor-provided tool chain, which maps the circuit design to the slices on the FPGA. Only after this step one can say with certainty how much FPGA resources a design consumes.

To evaluate basic regular expressions on the FPGA we have developed a regular expression → VHDL compiler. The compiler generates a corresponding NFA and DFA representation in VHDL for a given regular expression. We then measured flip-flop, lookup table and slice consumption. As discussed in Section 4.1, the NFA for the expression

$$\texttt{(0|1)*1(0|1)}^{i} \qquad (\star)$$

will consist of $1 + (i + 1)$ states, while an equivalent DFA will need as many as $1 + (2^{i+1} - 1)$ states. As can be seen in Figure 3, this has immediate consequences on the consumption of slices in an FPGA-based implementation. An exponential number of slices is required to implement the DFA for increasing values of i.

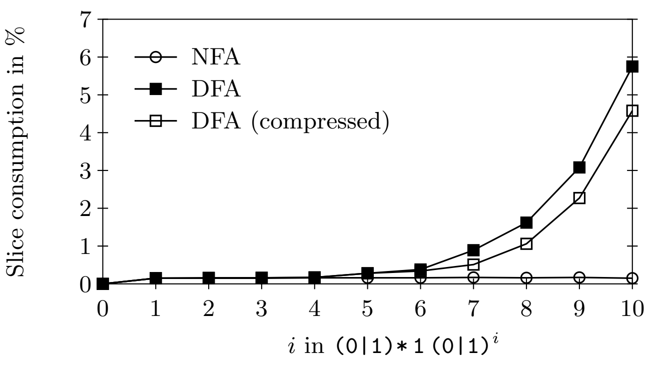


Figure 3: Effect of State Explosion

We have conducted experiments with NFAs and two different variants of DFAs, which are further explained below. For additional measurement results regarding flip-flop and lookup table consumption please refer to Appendix B.

NFA (–○–). Figure 3 clearly illustrates that in this example NFAs are by far superior to both DFA variants when it comes to FPGA resource consumption. The number of flip-flops and lookup tables (and effectively the number of slices) grows linearly with respect to i.

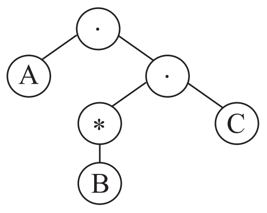
DFA (–■–). This variant of deterministic finite automaton was implemented in VHDL with a large *switch-case* statement. It turned out that the Xilinx *synthesizer* would use *one-hot* encoding for DFA state representation. Therefore, flip-flop consumption is proportional to the number of states of the DFA and is reflected in the exponential consumption of slices shown in Figure 3. Note that for this simple regular expression with i set to only 10 the DFA already uses up roughly 6% of the entire FPGA chip.

DFA Compressed (-⊟-). The exponential increase in flip-flop consumption can be reduced by *compressing* DFA states. Since we know that only one DFA state can be active at a time, we can compress the state representation into a single integer value that indexes the current state (k values can be indexed by an integer of size $\log_2 k$ bits). While this brings flip-flop consumption back to linear scaling, the DFA still experiences exponential growth in terms of lookup table consumption, because the transitions for all of the $1+(2^{i+1}-1)$ states still need to be implemented in logic. Thus, also for this DFA variant slice consumption is far from optimal.

4.5 From Patterns to Circuits

Converting a regular expression to an NFA is straightforward and can be achieved using, *e.g.*, Thompson's Algorithm [15] or the McNaughton-Yamada construction [9]. The concepts behind these algorithms can also be applied to generate NFAs directly in hardware, as was done in earlier work [16]. Here we focus on the process of compiling complex event patterns to NFA-based pattern matching circuits.

The first step for a compiler is to parse the complex event query and transform it into an *abstract syntax tree* (AST). This is an intermediate representation of the regular expression, better suited for further processing. For instance, the complex event pattern `AB*C` would translate to the abstract syntax tree depicted on the left. The leaves of this tree represent the *predicates* and the inner nodes correspond to regular expression operators (the sequence operator (·) is denoted with a dot). For every predicate in the regular expression we generate an NFA state — a hardware entity consisting of a single flip-flop and some combinatorial logic. These entities are then interconnected according to the inner nodes (regular expression operators) of the AST.



From the abstract syntax tree above our compiler generates the hardware NFA that is schematically depicted in Figure 4. The rounded boxes represent the entities that are generated for each leaf of the AST (`A`, `B` and `C`). These entities run fully in parallel, *e.g.*, each entity concurrently checks with the *Predicate Decoder* if the current tuple satisfied the appropriate predicate.

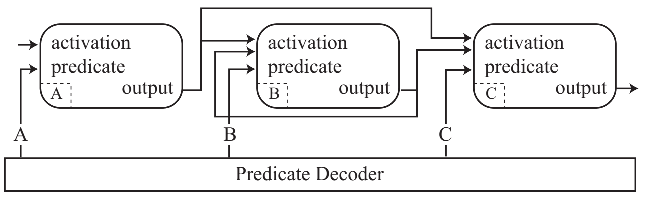


Figure 4: Hardwired NFA for `AB*C`

When an entity is *active* and the proper predicate is satisfied, the flip-flop of that entity is set to logic high, *i.e.*, the corresponding *output* wire will carry a `'1'`. An entity is either active per default or it can be activated by one of its predecessors. For example, entity `A` in Figure 4 is active per default because `AB*C` can match anywhere in the tuple stream. When predicate `A` is satisfied entity `B` and `C` will be activated by entity `A` since the output wire of entity `A` is connected to the activation port of entity `B` and `C`. In the next section, we explain how the compiler can decide which output wires need to be connected to which activation ports.

4.6 The Wiring Algorithm

The compiler can determine the proper *activation* wiring by traversing the abstract syntax tree and following a few basic rules depending on the regular expression operators encountered in the inner nodes of the tree. As the tree is traversed a set of *activation* signals is constantly being updated and when a leaf node is encountered the current set of signals is assigned to that node. Figure 5 illustrates this idea and each step is explained below.

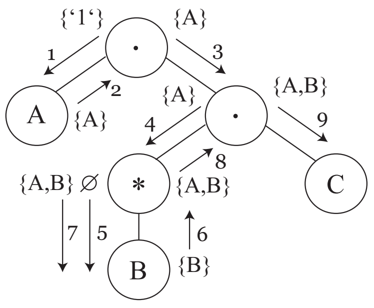


Figure 5: Wiring algorithm

Initially, the set of *activation* signals {'1'} has only one element. The starting point for the algorithm is the root of the tree, which in this case is a *sequence* node. (1) For this node the rule is to first process the left child and then the right one, *i.e.*, {'1'} is passed down to the left child. (2) Since this is a leaf node, the *activation* signals are applied. '1' in this case means default activation. Then the output signal of the entity that this leaf node represents is passed back to the parent. (3) The parent updates the *activation* set ({A}) and passes it down to its right child. (4) This node is again a *sequence* node and is processed analog to the previous one, *i.e.*, {A} is passed down to the *closure* node. (5) Since the sub-expression in a *closure* can activate itself, we first need to get the output signals from the sub-expression. For this purpose the empty set is passed down to the sub-tree. (6) The child of the *closure* node is a leaf node, but since the activation set is empty no signals are applied. Nevertheless, the output signal of this node {B} is returned to the parent. (7) At the *closure* node the two *activation* sets are merged to {A,B} and passed down to the child again where the activation signals are applied. (8) The set {A,B} is also returned to the parent of the *closure* node (9) and from there down to the parents right child.

5. PREDICATE DECODER

The *Predicate Decoder* is a separate component, consisting of pure combinatorial logic that takes a tuple as input and returns a *predicate vector* as output. The *predicate vector* has one bit for each predicate indicating whether it was satisfied by the current tuple. Based on this information the NFA can decide which transitions to take next. Having a separate component taking care of this task makes sense from an architectural point of view, but it can also reduce area consumption (see Appendix A).

6. STREAM PARTITIONER

In this section, we describe how the *Stream Partitioner* component works. As its name indicates, the stream partitioner divides a tuple stream into multiple sub-streams (partitions) based on some *partition identifier.* A partition identifier can be a single tuple attribute or a combination of several attributes. All tuples that have the same partition identifier belong to the same sub-stream. To do pattern matching on a partition basis, in principle, we need a separate NFA to process each partition individually. However, since every tuple belongs to exactly one partition and we need to process only one tuple at a time, it is sufficient to store the *state vector* of the NFA separately for every partition. The state vector contains a bit for every state of the NFA indicating whether that state is *active.* The NFA takes a state vector and a predicate vector as inputs and returns the updated state vector as output. The job of the *Stream Partitioner* is to find the state vector corresponding to a given partition identifier and forward it to the NFA. Thus, the *Stream Partitioner* provides similar functionality in hardware as a hash table does in software.

6.1 A Pipeline based Stream Partitioner

One can think of many different approaches to implement the desired functionality of the *Stream Partitioner* in hardware. Nevertheless, many designs that work well for a few partitions, will not scale to support a large number of partitions. In this paper, we propose to implement the *Stream Partitioner* as a pipeline. As was shown in [14], a pipeline exhibits very good scaling properties on FPGAs. In hardware, we need to avoid large fan-outs and long signal paths. In a pipeline, every pipeline element is connected only to its left and right neighbor. Therefore, the signal paths are very short and supporting more partitions does not increase fan-out. In our experiments, we were able to support 800 partitions using up 89% of FPGA chip space (see Figure 8).

6.2 The Pipeline Design

Our pipeline is a chain of *pipeline elements.* Every element besides the first and last one is wired with its left and right neighbor. A pipeline element can be *free* or associated with a partition. In the latter case, the pipeline element stores the *partition identifier* and the NFA *state vector* of the associated partition. When a new tuple arrives the partition identifier is extracted and inserted into the pipeline. It is then handed from one pipeline element to the next, once every clock cycle. The key idea is that pipeline elements can be *swapped* under certain conditions. Swapping means that two neighbors exchange their stored partition identifier and NFA state vector. For example, when an associated pipeline element matches a partition identifier passing by, that element is swapped towards the end of the pipeline. Thus, when the partition identifier has been propagated through the entire pipeline, the last element will be the one it is associated with. The NFA state vector can then be retrieved from that element and passed (together with the predicate vector) to the NFA which in return updates the state vector and stores it in the last pipeline element again.

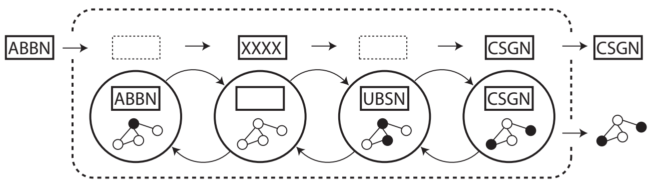


Figure 6: Stream Partitioner Pipeline

To exemplify, a pipeline is illustrated in Figure 6. The large circles represent pipeline elements with stored *partition identifier* and NFA *state vector.* The rectangle boxes above the circles are the partition identifiers that traverse the pipeline. Note that a new partition identifier can be injected into the pipeline only every other clock cycle. The reason is that within one clock period only every other pipeline element should trigger a swap. This is a requirement to avoid conflicting swap operations.

Pipeline elements are allocated dynamically. Initially the pipeline is empty, *i.e.*, all elements are *free.* The first free element encountered by a traversing partition identifier is swapped towards the end of the pipeline, as long as no associated pipeline element can be found. Hence, if the last pipeline element is free after the partition identifier has passed through the pipeline, then that element is allocated, *i.e.*, the partition identifier is stored there. If so, the NFA takes the *null vector* for the current state vector and stores the updated version in the last pipeline element as well. The swapping algorithm is displayed in Figure 7.

```
foreach partition identifier $x \in S$ do
    $i \leftarrow 1$;
    while $i < k$ do
        if $pe_i$.partitionidentifier $= x$ or $pe_i = free$ then
            swap $(pe_i, pe_{i+1})$;
        $i \leftarrow i + 1$;
    if $pe_k$.partitionidentifier $= x$ then // last element
        $pe_k$.statevec = NFA($pe_k$.statevec, predvec);
    else if $pe_k = free$ then
        $pe_k$.partitionidentifier $= x$;
        $pe_k$.statevec = NFA(nullvec, predvec);
    else
        // discard tuple
```

Figure 7: Pipeline Swapping Algorithm

When a partition identifier reaches the end of the pipeline the last pipeline element is in one of three states: (1) the pipeline element is associated with the partition identifier, (2) the pipeline element is free, (3) the pipeline element is associated with some other partition identifier. Notice that the third case can only occur when all pipeline elements are associated with other partitions, *i.e.*, the pipeline is full. In that case, our only option is to discard the current tuple. Therefore, we should try to avoid a full pipeline. We address this issue in the next section.

6.3 Temporary Pipeline Element Allocation

If we never free pipeline elements, no new partitions can ever be processed once the pipeline is full. In the worst case, tuples that belong to a very rare partition may occupy the pipeline, while at the same time tuples of a more frequent partition are lost. If we do not intend to detect overlapping regular expression matches, then a good point in time to free an associated pipeline element is when the NFA detects a match in the corresponding partition. Nevertheless, matches are more likely to occur with frequent partitions than with infrequent ones. Therefore, we might end up with a full pipeline, despite the *freeing*-strategy described above.

An entirely different approach is to restrict the amount of time a pipeline element can be associated with a given partition, allowing pattern matches to occur only within a certain window of time. We can achieve this by storing a timer in every freshly allocated pipeline element. When the timer runs down, the element is automatically freed. However, when pipeline elements are swapped the timer needs to be interchanged as well. A too fine granular timer could therefore lead to prohibitively high extra overhead in terms of FPGA resource consumption. We propose to use a lightweight timer that requires only four bits. A global *update-timer* signal determines when the timer value needs to be decremented. A four-bit timer can be decremented 16 times. That means we need to adjust the global update signal to fit our requirements, *e.g.*, if we want to allow a time window of one second, the timer needs to be updated roughly every 8,000,000 clock cycles of a 125MHz clock.

6.4 Evaluation of the Pipeline Approach

The number of supported, concurrent partitions directly translates to the number of pipeline elements, *i.e.*, the depth of the pipeline. Obviously, there is a limit to how many pipeline elements can be placed on a single FPGA chip. Concerning this matter, we have conducted experiments on a Virtex-5 FPGA from Xilinx (see Appendix C). It should be noted at this point that the next generation of this chip (Virtex-6) has significantly more resources to offer. In Figure 8 we show the percentage of FPGA resources consumed by a pipeline of varying depths. In this experiment, the partition identifier is a 16-bit attribute of a 128-bit tuple and the pattern we tested was `A(B|C*)D`, *i.e.*, the NFA *state vector* had four bits and the timer had four bits as well.

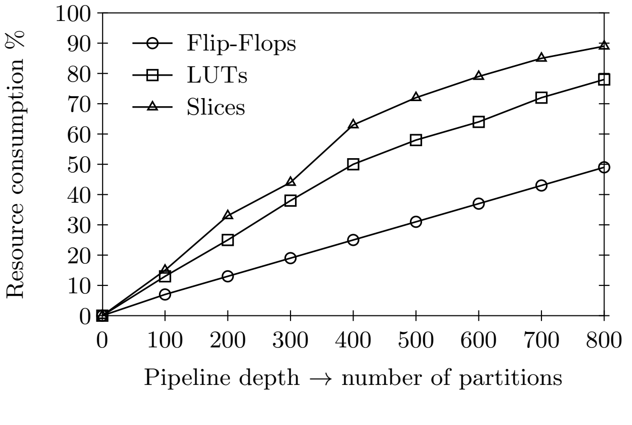


Figure 8: FF, LUT and Slice Consumption

From the graph it can be seen that we run out of lookup tables before we run out of flip-flops. Also, resource consumption increases linearly with respect to pipeline depth. The fact that 800 processing elements, occupying 89% of the available slices on our FPGA, did not lead to timing constraint violations or other problems, is a clear indication that our solutions exhibits excellent scaling.

7. SYSTEM EVALUATION

In this section we present the evaluation results of our complex event detection system as a whole. The implementation is on a Virtex-5 FPGA from Xilinx (see Appendix C). Besides verifying correctness of our system, the main goal of this implementation was to be able to perform throughput measurements and to check the maximal sustainable load with data arriving from a gigabit Ethernet link.

7.1 Experiment Setup

Tuples are transmitted to the FPGA over the network using UDP. Every UDP packet contains a fixed number of 128-bit wide tuples. To generate enough network load, we ran the tuple generator concurrently on three machines which we connected to the FPGA via a switch. We measured tuple and (Ethernet) frame throughput directly on the FPGA with additional circuitry developed especially for this purpose.

7.2 Throughput Measurements

If there was no network communication overhead then the theoretical upper bound of tuples that could hit our system on a gigabit link would be *7,812,500* tuples per second (1Gbit/s divided by 128 bit). To reduce communication overhead we can increase the size of the UDP packets so that more tuples fit into a single packet. In Figure 9 we measured tuple and (Ethernet) frame throughput of our complex event detection system with varying UDP packet sizes.

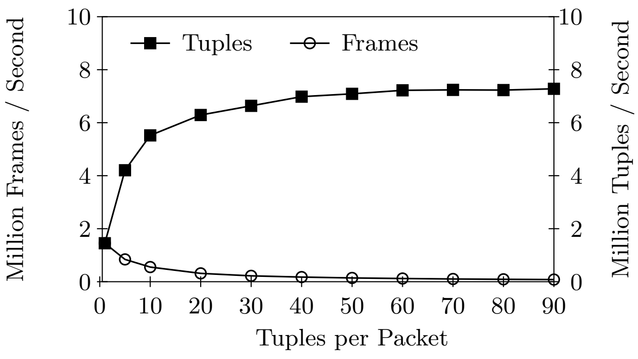


Figure 9: Experimental Results : Throughput

The communication overhead for each Ethernet frame includes 20 bytes for the IP header and 8 bytes for the UDP header next to the overhead for the frame itself (38 bytes), *i.e.*, the per frame overhead is 66 bytes (528 bits). With large frames, *e.g.*, of size 1,440 bytes containing 90 tuples ($528+90\times128$ bits), we were able to process up to *7,279,215* tuples per second—close to the theoretical upper limit stated above.

Knowing the frame overhead ($F_{overhead}$ = 528 bits), we can calculate the bandwidth utilization (B_{util}) using the following formula:

$$B_{util} = N_{frames/s} \times (F_{overhead} + N_{tpp} \times 128).$$

The number of frames per second ($N_{frames/s}$) is multiplied with the network communication overhead ($F_{overhead}$) and the UDP payload, which is the number of tuples per packet (N_{tpp}) times tuple size (128 bits). For the example above (90 tuples per packet) we measured that the FPGA processed approximately 80,880 frames per second which results in a bandwidth utilization of:

$$B_{util} = 80,880 \times (528 + 90 \times 128) = 974\ Mbit/s$$

Though this number is impressive, it needs to be said that it is typically not the large network packets that CPU based systems have trouble with. As was shown in [11], commodity systems struggle most with processing network data with high packet rates. Therefore, the more interesting results are the ones with small packets. The smallest packets in our experiments contained exactly one tuple. We measured that the FPGA processed 1,451,373 such packets per second, thus resulting in a bandwidth utilization of:

$$B_{util} = 1,451,373 \times (528 + 1 \times 128) = 952\ Mbit/s$$

This result demonstrates the true value of our work. It shows that in by-passing the network-memory-CPU bottleneck our system is able to detect complex event patterns even on network traffic with very high packet rates, something that is not feasible with CPU based solutions.

8. RELATED WORK

The importance of *complex event detection* is manifested in an ongoing effort to standardize an SQL extension with pattern matching support [17]. While this SQL extension has already been implemented in software systems [5, 4], to the best of our knowledge, we are the first to look at hardware based complex event detection.

In 1982, Floyd and Ullman [6] first studied implementing NFAs in hardware and showed that an n-state NFA requires at most $\mathcal{O}(n)$ circuit area. Later, Sidhu and Prasanna [12] presented an NFA based implementation of regular expressions on FPGAs. Yang *et al.* [16] have slightly modified the NFA architecture used in [12] to a more modular structure. Our NFAs are based on their work.

FPGAs were proposed to accelerate stream processing in [11], where a compiler was presented to transform continuous queries into logic circuits. The idea to use a pipeline of processing elements to implement our *Stream Partitioner* component was inspired by [14], where the *Space-Saving* algorithm was pipelined to solve the *frequent item problem.*

Finally, using an 8-to-256 character *pre-decoder* to share the character comparators among states of their NFAs and thus reducing hardware resources was suggested in [3]. Having a separate *Predicate Decoder* component in our *complex event detection system* is based on this idea.

9. SUMMARY

Complex event detection using CPU based systems suffer from severe limitations on the amount of data that can be brought to the CPU due to bottlenecks between the network, memory, and the CPU itself. In this paper, we propose to use FPGAs to circumvent this problem. By inserting the FPGA in the data path, *e.g.*, between network interface and CPU, we can detect *complex events* at gigabit wire speed. Our solution uses regular expressions implemented as finite automata to detect complex events. Given that FPGAs impose very different design constraints than software based solutions, the paper describes in detail the trade-offs of implementing non-deterministic finite automata in an FPGA. The experiments show that the resulting system is both efficient in terms of chip space requirements and can process event streams at very close to wire speed.

10. 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] BRO. `http://bro.ids.org/`.

[3] C. Clark and D. Schimmel. Scalable Pattern Matching for High Speed Networks. In *FCCM '04*, Washington, DC, USA, 2004.

[4] 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.

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

[6] R. Floyd and J. Ullman. The compilation of regular expressions into integrated circuits. *Symposium on Foundations of Computer Science*, 0:260–269, 1980.

[7] 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.

[8] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.

[9] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Transactions on Electronic Computers*, 9:39–47, 1960.

[10] A. Mitra, W. Najjar, and L. Bhuyan. Compiling PCRE to FPGA for accelerating SNORT IDS. In *ANCS'07*, New York, NY, USA, 2007.

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

[12] R. Sidhu and V. Prasanna. Fast Regular Expression Matching Using FPGAs. *Field-Programmable Custom Computing Machines, Annual IEEE Symposium on*, 0:227–238, 2001.

[13] SNORT. `http://www.snort.org`.

[14] J. Teubner, R. Müller, and G. Alonso. FPGA Acceleration for the Frequent Item Problem. In *ICDE'10*, Long Beach, CA, USA, 2010.

[15] K. Thompson. Programming Techniques: Regular expression search algorithm. *Commun. ACM*, 11(6):419–422, 1968.

[16] 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.

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

APPENDIX

A. PREDICATE DECODER

Decoding predicates in a separate component makes sense not only from an architectural point of view. It may also lead to a substantial reduction in chip space consumption, as we will show in this section.

Consider the regular expression `ABAB` and assume the predicates `A` and `B` define the ASCII characters 'A' and 'B', *i.e.*, `A` is satisfied when the seven wires encode the ASCII code '65'. Figure 10 illustrates the NFA that matches this regular expression without the support of a separate *Predicate Decoder*.

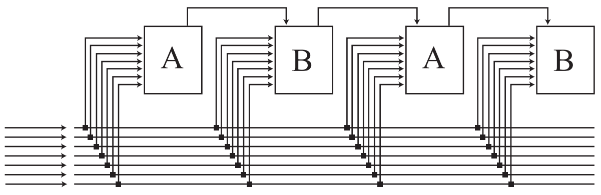


Figure 10: NFA for `ABAB`

The boxes in the figure represent states of the NFA. Notice that all seven wires are routed to every state and how it is redundantly checked whether an 'A' or a 'B' is matched. In Figure 11 the same NFA is depicted, however, with predicate decoding offloaded to a separate component.

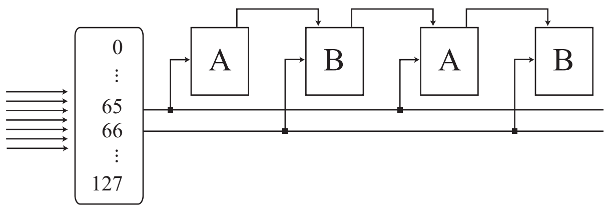


Figure 11: NFA for `ABAB` with *Predicate Decoder*

Now, at the NFA states it can be decided whether a predicate was satisfied by checking a single signal. This means less wires need to be routed, *i.e.*, the interconnect is used more efficiently. Also the logic for evaluating the predicates is no longer redundantly present on the chip which leads to a reduction in lookup table consumption.

The *Predicate Decoder*, in this case, converts 7-bit ASCII encoding into 128-bit *one-hot* encoding, *i.e.*, the *Predicate Decoder* is a simple *demultiplexer*. However, in a typical streaming application the predicates are more complex, *e.g.*, may contain Boolean operators and comparison operators. This makes the use of a separate *Predicate Decoder* even more compelling.

One could assume that the synthesizer detects redundant structures and optimizes them on its own. However, as our measurements show, this is not always the case. In Figure 12 we have compared a design with a separate *Predicate Decoder* against a design without.

Our measurements are based on the regular expression `(AB)`^{i}. The number of flip-flops used is negligible. We need one flip-flop per state in the NFA, *e.g.*, `(AB)`^{250} requires 500 flip-flops. Nevertheless, the lookup-table (LUT) consumption reaches critical levels with increasing i when no predicate decoder is used.

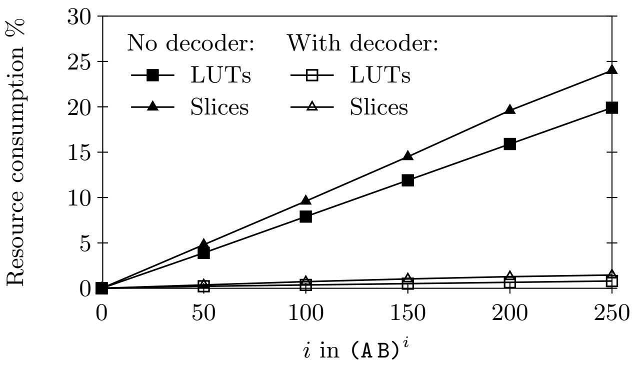


Figure 12: LUT and Slice Consumption

B. STATE EXPLOSION

In this section we present additional measurement results of our experiments concerning the *state explosion* in DFAs, which we discussed in Section 4.3. Figure 13 depicts the number of flip-flops (in percent) required by the respective finite automata types corresponding to the regular expression `(0|1)*1(0|1)`^{i}.

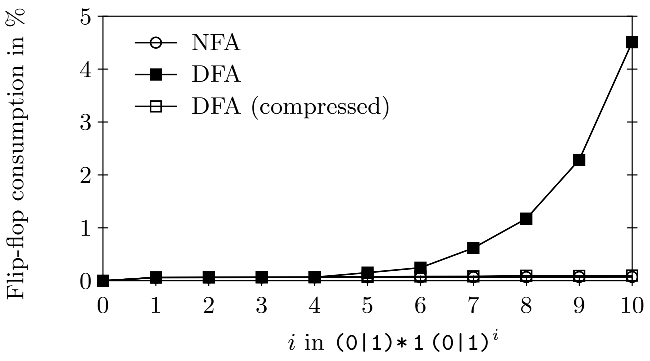


Figure 13: Flip-flop Consumption

The figure illustrates that by using a binary representation (DFA compressed) to store the active state of the DFA, *state explosion* does not affect flip-flop consumption. Thus, NFAs and DFAs with compressed states require a similar amount of filp-flops, whereas DFAs with *one-hot* encoded states exhibit exponential flip-flop consumption.

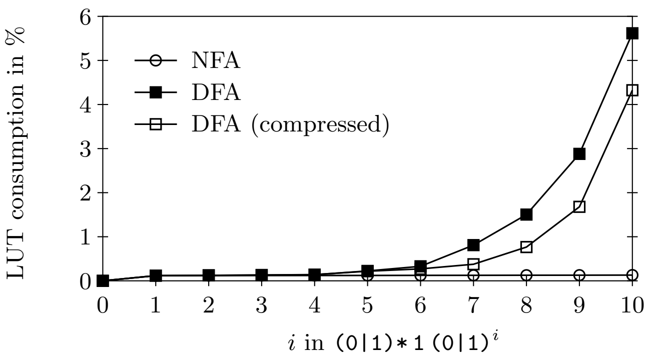


Figure 14: Lookup Table Consumption

In Figure 14 lookup table consumption is illustrated. In the NFA case the number of lookup tables consumed increases linearly with respect to i. While a DFA with compressed state representation requires less lookup tables than a DFA with *one-hot* encoded states, the number of lookup tables still grows exponentially with respect to i.

C. FPGA CHARACTERISTICS

In a nutshell, FPGAs are chip devices that host a pool of resources, which can be configured to implement user-specified logic circuits directly in hardware. Commonly available resource types include lookup tables (LUTs) to realize combinatorial logic, on-chip storage in terms of Block RAM (BRAM) and flip-flop registers, and a configurable interconnect network. All of our experiments were conducted on a Virtex-5 FPGA from Xilinx. Some selected characteristics are displayed in Table 2.

LUTs (6-to-1 lookup tables)	69,120
flip-flops (1-bit registers)	69,120
block RAM (total kbit)	5,328
block RAM (number of 36 kbit blocks)	148

Table 2: Resources available in a Virtex-5 FPGA from Xilinx (XC5VLX110T).