

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 [6]. Typically, the individual data items (tuples) in those streams only become meaningful when put into context with other items of the same stream.

Complex event processing (CEP) aims at inferring meaningful higher-level events (complex events) from a sequence of low-level events [1]. 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 arrive 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 [12].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Articles from this volume were presented at The 36th International Conference on Very Large Data Bases, September 13-17, 2010, Singapore.

Proceedings of the VLDB Endowment, Vol. 3, No. 1
Copyright 2010 VLDB Endowment 2150-8097/10/09... \$ 10.00.

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 as to avoid the network-memory-CPU bottleneck.

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.

Whereas *regular expressions* have been used in existing FPGA solutions [2, 10, 20] for, *e.g.*, deep packet inspection in network intrusion detection systems¹, our solution uses regular expressions to define the complex events of interest.

Contributions. Our main contribution is a high-throughput complex event detection system based on regular expression matching in hardware. It can be attached directly to the network interface and it operates 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) that guarantee a bound on the space required.

In the paper, we show how complex event patterns can be expressed in a declarative query language based on a recent standardization effort [21] to extend SQL with pattern matching capabilities (Section 2). 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 (Section 7). We conclude the paper with a survey of related work (Section 8) and a summary of the results (Section 9).

¹SNORT. <http://snort.org>, BRO. <http://bro-ids.org>

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 [21]. Nevertheless, since our focus is on Boolean, regular expression-based complex event detection, we have derived a simplified and less verbose version of the language.

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
(<i>r</i>)	grouping: bypass default binding
<i>r</i> *	closure: zero or more repetitions of <i>r</i>
<i>r</i> ₁ <i>r</i> ₂	sequence: <i>r</i> ₁ followed by <i>r</i> ₂
<i>r</i> ₁ <i>r</i> ₂	choice: either <i>r</i> ₁ or <i>r</i> 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:

```
1 PATTERN ( A NOTB* C | A NOTC* D | A NOTD* E )
2 DEFINE
3   A   AS (checkpoint = 'Staten Island')
4   NOTB AS (checkpoint != 'Brooklyn')
5   C   AS (checkpoint = 'Queens')
6   NOTC AS (checkpoint != 'Queens')
7   D   AS (checkpoint = 'Bronx')
8   NOTD AS (checkpoint != 'Bronx')
9   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 (`NOTx` definitions above).

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.

2.3 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:

```
1 PARTITION runner
2 PATTERN ( A NOTB* C | A NOTC* D | A NOTD* E )
3 DEFINE
4   A   AS (checkpoint = 'Staten Island')
5   . . .
```

The partitioning attribute (`runner`) 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 we have developed and its key components. The system is connected directly to the Ethernet MAC component of the physical network interface to achieve full wire speed performance. Figure 1 depicts the placement of the FPGA in the data path between network interface and CPU.

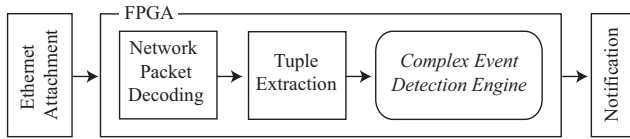


Figure 1: FPGA placed in Data Path

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.

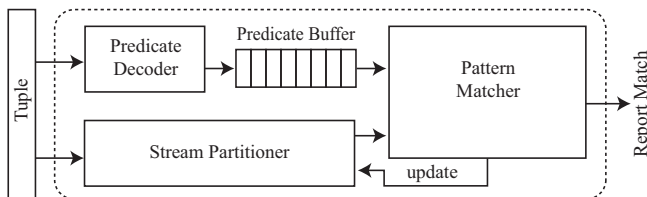


Figure 2: Complex Event Detection Engine

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 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 in software is a well studied problem [7]. However, the implementation in hardware is very different given the completely different design constraints. 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 does not hold. Thinking out-

side 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.

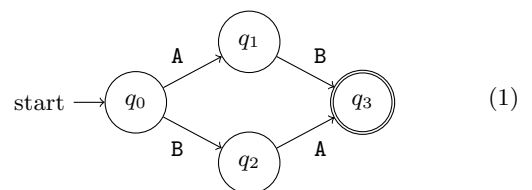
To compile a regular expression into a DFA, the expression is usually mapped to an NFA first, *e.g.*, using Thompson’s algorithm [17]. Then the NFA is converted into an equivalent DFA with the *powerset construction* [7]. This eliminates non-determinism by inserting new DFA states that incorporate all 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

$$(0|1)^* 1 (0|1)^i \quad (*)$$

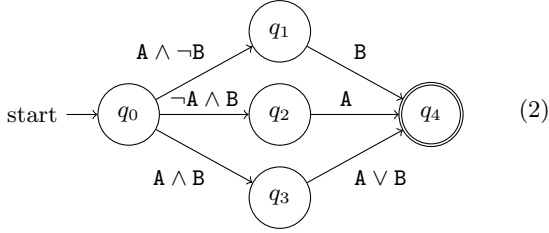
where $(0|1)^i$ denotes an i -fold repetition of subexpression $(0|1)$. Whereas a non-deterministic automaton for this expression can be built with $1 + (i + 1)$ states, a corresponding DFA requires at least $1 + (2^{i+1} - 1)$ states. In general, a DFA may require up to 2^n states compared to an equivalent NFA with only n states [7]. 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 \rightarrow 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. As an example, 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:



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:



With the possibility of overlapping predicates, the 2^n factor in the number of DFA states becomes a 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 turn into k -way conjunctive predicates — with the corresponding high cost for evaluation.

4.3 NFAs in Hardware

The main asset provided by FPGAs is an amount of *chip space* that contains resources of different types. Most importantly, *lookup tables* (LUTs) are a configurable means to implement Boolean logic and *flip-flop registers* can be used to hold individual bits of state (more details in Appendix A).

The availability of both resource types is limited and may quickly become a problem when DFA sizes become too large. On the other hand, FPGA hardware operates in an inherently parallel manner, which we can exploit to construct NFAs that run equally fast as their DFA counterparts.

Figure 3 illustrates how an NFA can be realized using FPGA resources. It implements the Automaton 1 shown above (*i.e.*, the regular expression $AB|BA$). Each of the four states q_0 to q_3 is mapped to one flip-flop that signifies whether that state is active or not (D and Q contacts of each flip-flop indicate its data input and output port, respectively). State transitions become wires in the interconnect network, and they are conditioned using Boolean logic (AND and OR gates built from FPGA lookup tables).

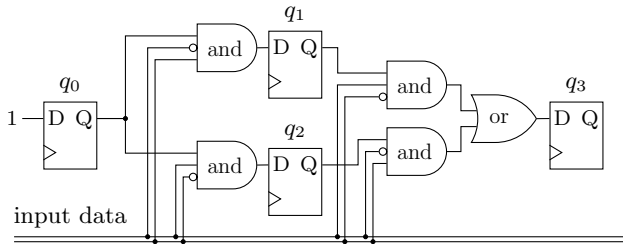


Figure 3: Hardware implementation of non-deterministic automaton 1 (regular expression $AB|BA$)

In Figure 3, we assume that the input character set is a 2-bit alphabet where $A \equiv 01$ and $B \equiv 10$. The top-left AND gate is used in such a way that state q_1 becomes active when state q_0 is active and the next input symbol is A (\neg indicates negated input). Likewise, state q_3 becomes active when state q_1 is active and the next input symbol is B or when state q_2 is active and the next input symbol is A (thus the OR gate left of state q_3).

We note that this NFA design fits well the typical architecture of FPGAs (see Appendix A). In particular, lookup tables and flip-flops are paired within so-called *slices* in the same way as the predicate-state combination in Figure 3.

4.4 Evaluation: State Explosion

The space savings on the chip due to the use of an NFA can be significant. To demonstrate the effect, we determined the different automata types for the regular expression $(\star)^i$ that we discussed above. As mentioned before, the corresponding NFA will consist of $1 + (i + 1)$ states, while the equivalent DFA will need as many as $1 + (2^{i+1} - 1)$ states. This is reflected in Figure 4, where the use of an NFA shows significant scalability advantages over DFA-based approaches for increasing values of i .

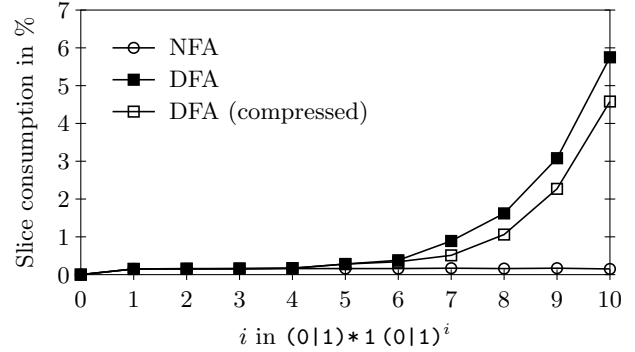


Figure 4: Effect of State Explosion

To obtain Figure 4, we conducted experiments with NFAs and two different variants of DFAs. For each configuration, we compiled the respective automaton into a VHDL circuit description and used vendor-specific FPGA tools (Xilinx ISE 11) to obtain space occupation numbers. Figure 4 shows the consumption of slices for each approach. Flip-flop and lookup table consumption are shown in Appendix D.

NFA (\circ). Figure 4 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 .

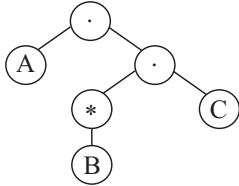
DFA (\blacksquare). This variant of deterministic finite automaton was implemented in VHDL with a large *switch-case* statement. It turned out that the Xilinx synthesizer would use *one-hot* encoding for DFA state representation. Therefore, flip-flop consumption is proportional to the number of states of the DFA, which is reflected in the exponential consumption of slices shown in Figure 4. 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 (\square). 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 (see Appendix D), 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 [17] 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 [20]. 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, used for further processing.



For instance, the complex event pattern AB^*C translates 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 \cdot 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 5. 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.

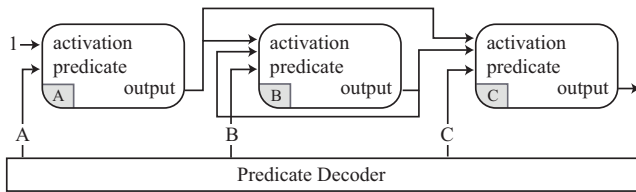


Figure 5: Hardwired NFA for AB^*C

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 5 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 6 illustrates this idea and each step is explained below.

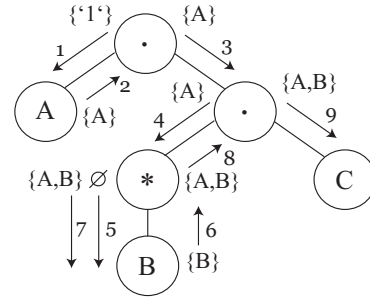


Figure 6: Wiring algorithm

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. Decoding predicates outside the NFA in a separate component may reduce area consumption significantly (see Appendix C). Our approach is based on a similar idea suggested in [2], where an 8-to-256 *character pre-decoder* was used to share the character comparators among states of their NFAs.

6. STREAM PARTITIONER

A key difficulty in complex event processing on FPGAs, as compared to, *e.g.*, pattern matching in intrusion detection systems, comes from the *partitioning* functionality outlined in Section 2.3. Thereby, a *partition identifier* (an attribute in the input stream) divides an input tuple stream into multiple logical sub-streams (partitions).

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 have 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.

6.1 A Pipeline-Based Stream Partitioner

One can think of many different approaches to implement 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 [16], a pipeline exhibits very good scaling properties on FPGAs, where large fan-outs and long signal paths need to be avoided (see Appendix A). 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.

6.2 The Pipeline Design

Our pipeline is a chain of *pipeline elements* (pe_1, \dots, pe_k). 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* ($pe_i.pid$) and the NFA *state vector* ($pe_i.statevec$) 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 is then retrieved from that element and passed to the NFA (jointly with the *predicate vector* ($predvec(t)$) of the respective tuple t), which in return stores the updated state vector in the last pipeline element ($pe_k.statevec$) again.

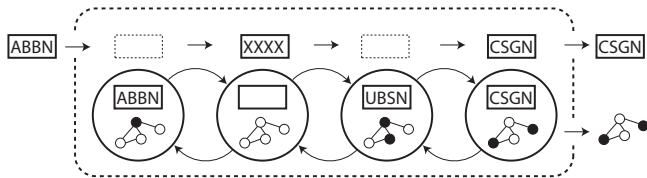


Figure 7: Stream Partitioner Pipeline

To exemplify, a pipeline is illustrated in Figure 7. 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* ($nullvec$) for the current state vector and stores the updated version in the last pipeline element as well. The swapping algorithm is displayed in Figure 8.

```

1  foreach tuple t ∈ InputStream do
2      i ← 1;
3      while i < k do
4          if pei.pid = t.pid then
5              swap (pei, pei+1);
6          else if pei = free and pei+1.pid ≠ t.pid then
7              swap (pei, pei+1);
8          i ← i + 1;
9      /* process last element in pipeline */
10     if pek.pid = t.pid then
11         pek.statevec = NFA (pek.statevec, predvec(t));
12     else if pek = free then
13         pek.pid = t.pid;
14         pek.statevec = NFA (nullvec, predvec(t));
15     else
16         discard (t);

```

Figure 8: 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.

6.3 Temporary Pipeline Element Allocation

The number of partitions that can be handled is limited by the size of the pipeline, *i.e.*, by FPGA real estate, as we show in the next section. We can relax this limitation somewhat by dynamically allocating (see previous section) and releasing pipeline elements, as they are needed. The difficult question is at what point in time to release a pipeline element. We could release a pipeline element when the NFA detects a match in the corresponding partition (ignoring overlapping matches), but for those partitions where a match never occurs the pipeline elements would be kept allocated indefinitely. Thus, we propose to add a lightweight timer (four bits) to every pipeline element allowing each element to be automatically freed when its timer runs down. The timeout-range can be tailored to meet application-specific requirements by adjusting the frequency of a global *update-timer* signal. For example, if we want to allow a time window of one second, the timer needs to be updated roughly every 8×10^6 clock cycles of a 125MHz clock since a 4-bit timer can be decremented 16 times.

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. Addressing this matter, we have conducted experiments on a Virtex-5 FPGA from Xilinx. It should be noted at this point that next generation Virtex-6 FPGAs have significantly more resources to offer (see Appendix B). In Figure 9 we show the percentage of FPGA resources consumed by a pipeline of varying depths. In this experiment, the partition identifier was 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.

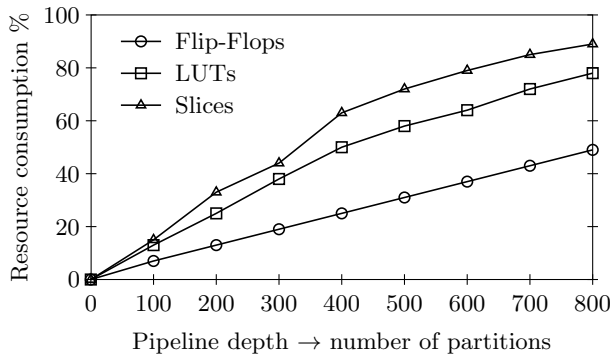


Figure 9: FF, LUT and Slice Consumption

From the graph in Figure 9 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.

Pipelining is a common technique in electrical engineering to increase throughput. However, the longer a pipeline the higher is the latency. In our pipeline, a partition identifier progresses from one pipeline element to the next every 16 nanoseconds (half the outside clock frequency of 125 MHz). Therefore, with an 800 element-deep pipeline the latency is 800×16 nanoseconds = 12.8 microseconds. Thus, the latency is still very low and we consider it irrelevant when compared with the arrival rate of standard streams.

7. SYSTEM EVALUATION

In this section, we present evaluation results of our complex event detection system as a whole. The implementation is on a Virtex-5 FPGA from Xilinx (see Appendix B). Besides verifying the 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.

Experimental 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 through-

put directly on the FPGA with additional circuitry developed especially for this purpose.

7.1 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 10 we measured tuple and (Ethernet) frame throughput of our complex event detection system with varying UDP packet sizes.

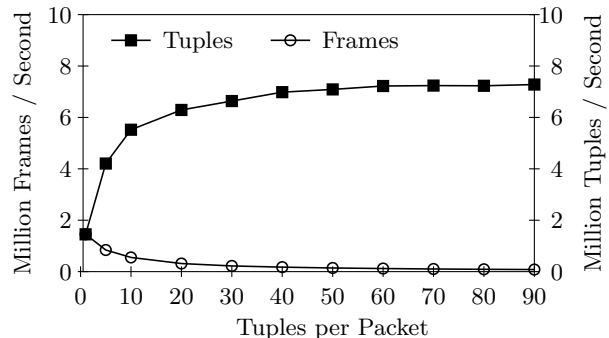


Figure 10: Experimental Results: Throughput

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 ($528+90 \times 128$ 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:

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

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 [12], 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:

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

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

While traditional database engines are increasingly hitting the limitations of commodity computing architectures, several commercial solutions already demonstrate the advantages of FPGA acceleration for database processing. Most notably in this respect are Netezza's TwinFin system [13] or the appliances offered by Kickfire [8] and XtremeData [19]. An architectural difference of our system is that we connect the FPGA directly to the network interface, rather than using it as a co-processor next to the CPU.

On the research side, the use of FPGAs has been proposed, *e.g.*, for XML filtering [11], stream processing [12, 18], or financial trading [14]. Our pipelined implementation of stream partitioning was inspired by our earlier work in [16], where we used a similar design to parallelize the computation of frequent items.

The importance of *complex event detection* [1, 6] is manifested in an ongoing effort to standardize an SQL extension with pattern matching support [21]. While this SQL extension has already been implemented in software systems, *e.g.*, ESPER [4] or DejaVu [3], we are the first to look at hardware-based complex event detection.

Since Floyd and Ullman [5], in 1982, first studied implementing regular expressions in hardware as NFAs, there have been a number of publications on regular expression matching with FPGAs, for example [2, 10, 15, 20]. Whereas earlier work used FPGAs for pattern matching on strings, in our work, we detect complex event patterns in sequences of events, which are based on arbitrary attributes. This adds significant complexity to the problem, *e.g.*, amplified state explosion due to overlapping predicates and the necessity for stream partitioning functionality.

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.

Acknowledgments. 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/>).

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] C. Clark and D. Schimmel. Scalable Pattern Matching for High Speed Networks. In *FCCM '04*, Washington, DC, USA, 2004.
- [3] N. Dindar, B. Güç, P. Lau, A. Özaland M. Soner, and N. Tatbul. DejaVu: Declarative Pattern Matching over Live and Archived Streams of Events. In *SIGMOD'09*, Providence, RI, USA, 2009.
- [4] EsperTech Inc. <http://esper.codehaus.org/>.
- [5] R. W. Floyd and J. Ullman. The Compilation of Regular Expressions into Integrated Circuits. *J. ACM*, 29(3), 1982.
- [6] D. Gyllstrom, E. Wu, H. Chae, Y. Diao, P. Stahlberg, and G. Anderson. SASE: Complex Event Processing over Streams. In *CIDR'07*, Asilomar, CA, USA, 2007.
- [7] J. Hopcroft, R. Motwani, and J. Ullman. *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*. Addison Wesley, 2000.
- [8] Kickfire. <http://www.kickfire.com/>.
- [9] R. McNaughton and H. Yamada. Regular Expressions and State Graphs for Automata. *IEEE Transactions on Electronic Computers*, 9, 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] A. Mitra, M. R. Vieira, P. Bakalov, V. J. Tsotras, and W. A. Najjar. Boosting XML Filtering Through a Scalable FPGA-Based Architecture. In *CIDR'09*, Asilomar, CA, USA, 2009.
- [12] R. Müller, J. Teubner, and G. Alonso. Streams on Wires - A Query Compiler for FPGAs. In *VLDB'09*, Lyon, France, 2009.
- [13] Netezza Corp. <http://www.netezza.com/>.
- [14] M. Sadoghi, M. Labrecque, H. Singh, W. Shum, and H. Jacobsen. Efficient Event Processing through Reconfigurable Hardware for Algorithmic Trading. In *VLDB'10*, Singapore, 2010.
- [15] R. Sidhu and V. Prasanna. Fast Regular Expression Matching Using FPGAs. In *FCCM'01*, Los Alamitos, CA, USA, 2001.
- [16] J. Teubner, R. Müller, and G. Alonso. FPGA Acceleration for the Frequent Item Problem. In *ICDE'10*, Long Beach, CA, USA, 2010.
- [17] K. Thompson. Programming Techniques: Regular Expression Search Algorithm. *Commun. ACM*, 11(6), 1968.
- [18] P. Vaidya, J. Lee, F. Bowen, Y. Du, C. Nadungodage, and Y. Xia. Symbiote: A Reconfigurable Logic Assisted data Stream Management System (RLADSMS). In *SIGMOD'10*, 2010.
- [19] XtremeData, Inc. <http://www.xtremedata.com/>.
- [20] Y. Yang, W. Jiang, and V. Prasanna. Compact Architecture for High-Throughput Regular Expression Matching on FPGA. In *ANCS'08*, San Jose, CA, USA, 2008.
- [21] F. Zemke, A. Witkowski, M. Cherniack, and L. Colby. Pattern Matching in Sequences of Rows. In *Technical Report ANSI Standard Proposal*, 2007.

APPENDIX

A. FPGA ARCHITECTURE

In a nutshell, *field-programmable gate arrays* (FPGAs) are chip devices that host a pool of resources, which can be configured to implement user-specified logic circuits directly in hardware. Essentially, an FPGA consists of many so-called *configurable logic blocks* (CLBs) arranged in a 2-dimensional array. The CLBs are connected to each other by an *interconnect network* as illustrated by the parallel wires between the CLBs in Figure 11. At every intersection of two wire channels, a configurable *switchbox* determines how the CLBs are interconnected.

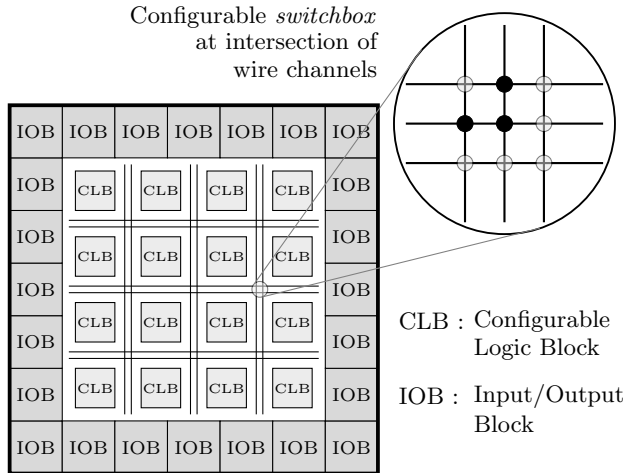


Figure 11: Basic FPGA Architecture

Internally, each CLB is divided into two *slices* which, in turn, are composed of *lookup tables* (LUTs) and *flip-flop registers*. In Virtex-5 chips, each slice embodies four LUT-flip-flop pairs, *i.e.*, there are eight lookup tables and eight flip-flops per CLB.

Lookup tables and flip-flops are combined into pairs as shown in Figure 12. Lookup tables provide a configurable type of Boolean logic and can be used, *e.g.*, to implement AND or OR gates. More precisely, a 6-to-1 lookup table in a Virtex-5 chip can implement any function $\{0,1\}^6 \rightarrow \{0,1\}$ that has six binary inputs and one binary output.

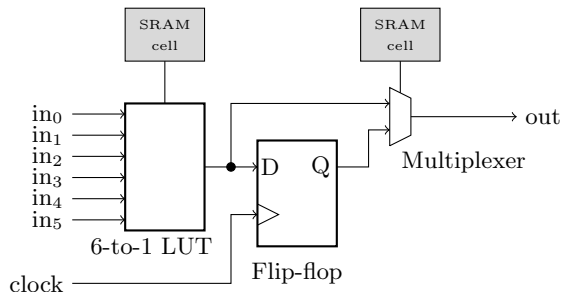


Figure 12: Virtex-5 LUT-flip-flop pair

As shown in Figure 12, each lookup table drives the input of a flip-flop register, which provides storage for a single bit of information. As such, the internals of an FPGA slice are

an excellent fit for the NFA implementation strategy that we outlined in Section 4.3.

Typically, FPGAs are not programmed at the lookup table and flip-flop level. Rather, the specification of a user logic is described using a higher level *hardware description language* (HDL), *e.g.*, VHDL or Verilog. The HDL code is then transformed (*synthesized*) into logic circuits which are mapped to the specific FPGA architecture.

Nevertheless, understanding this underlying architecture is crucial, in particular when designing large circuits. The larger a circuit, the more difficult it gets to map the circuit to the somewhat rigid FPGA structure. A common pitfall is that a design induces long signal paths across vast parts of the FPGA chip, which usually leads to a decreased clock rate, *i.e.*, a loss in performance.

A good design principle therefore is to build a large circuit from smaller self-contained sub-circuits, which each fit into a compact building block on the FPGA. In Section 4.3, we did so for the implementation of NFAs, where we mapped each state and its respective incoming transition to a LUT-flip-flop pair. Furthermore, the FPGA architecture presented above suggests that interacting sub-circuits should be placed close to each other on the FPGA chip so that communication can remain local and long signal paths are avoided. This is why the pipeline design that we propose in Section 6 for the *Stream Partitioner* scales well with an increasing number of partitions.

B. FPGA CHARACTERISTICS

Commonly available resource types hosted by FPGAs include *lookup tables* (LUTs) to realize combinatorial logic, on-chip storage in terms of *Block RAM* (BRAM) and *flip-flops*, 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 (XC5VLX110T) from Xilinx.

It is worth noting that next generation Virtex-6 FPGAs offer significantly more resources. In Table 3 we list some characteristics of the large Virtex-6 LX760 chip.

LUTs (6-to-1 lookup tables)	474,240
flip-flops (1-bit registers)	948,480
block RAM (total kbit)	25,920
block RAM (number of 36 kbit blocks)	720

Table 3: Resources available in a Virtex-6 FPGA (XC6VLX760) from Xilinx.

C. PREDICATE DECODER

Decoding predicates in a separate component makes sense from an architectural point of view and 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 13 illustrates the NFA that matches this regular expression without the support of a separate *Predicate Decoder*.

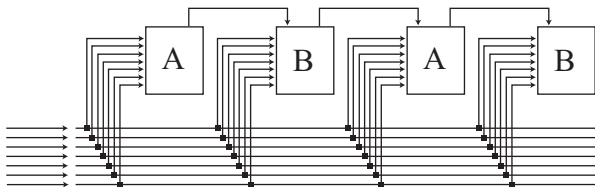


Figure 13: NFA for $ABAB$

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 14 the same NFA is depicted, however, with predicate decoding offloaded to a separate component.

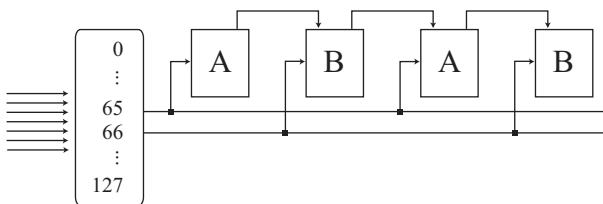


Figure 14: NFA for $ABAB$ with *Predicate Decoder*

Now it suffices to inspect a single signal at each NFA state to check whether a given predicate was satisfied. 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 leading to a reduced consumption of lookup table.

The *Predicate Decoder*, in this case, is a simple *demultiplexer* converting 7-bit ASCII encoding into 128-bit *one-hot* encoding. 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.

Our measurements show that it is wrong to assume that the synthesizer detects and optimizes all redundant structures on its own. In Figure 15 we have compared a design with a separate *Predicate Decoder* against a design without. These 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.

D. 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 16 depicts the number of flip-flops (in percent) required by the respective finite automata types corresponding to the regular expression $(0|1)^*1(0|1)^i$.

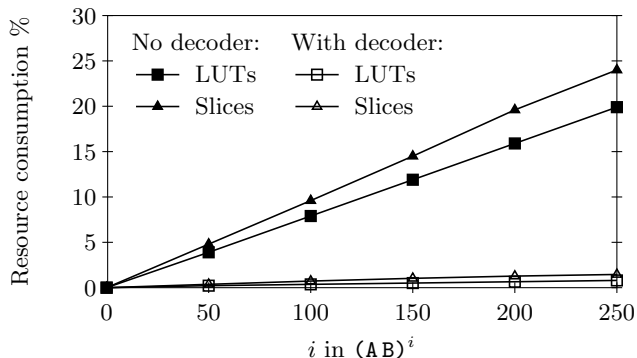


Figure 15: Effects of using a *Predicate Decoder*

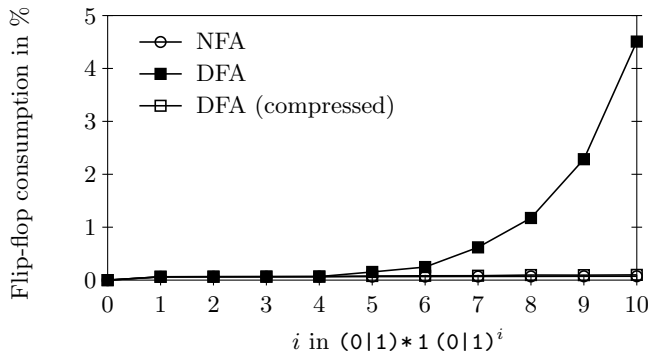


Figure 16: Flip-flop Consumption \rightarrow NFA vs. DFA

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 flip-flops, whereas DFAs with *one-hot* encoded states exhibit exponential flip-flop consumption.

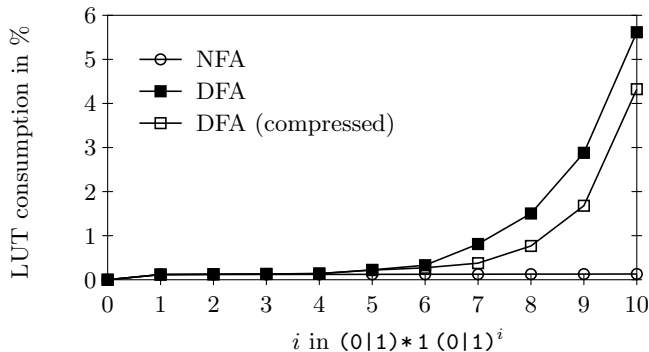


Figure 17: LUT Consumption \rightarrow NFA vs. DFA

In Figure 17 lookup table (LUT) 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 .