Automated Dynamic Reconfiguration for High-Performance Regular Expression Searching

Ken Eguro

Microsoft Research
Redmond, WA USA
eguro@microsoft.com

Abstract—**Dynamic reconfiguration can be necessary to produce fast and flexible FPGA-based applications. However, in practice very few developers actually use this capability. One reason for this is that it is very difficult to write and execute applications that are spread across multiple configurations. This paper uses the problem of regular expression searching for e-mail spam filtering to illustrate the potential advantages of dynamic reconfiguration and the inherent development problems associated with the conventional design methodology. To solve these problems, we present a regular expression system compiler. This automated tool includes 1) a mechanism to split a large set of searches into multiple hardware configurations and 2) a control system to manage reconfiguration and I/O marshalling during execution. Even with very rudimentary reconfiguration support from the platform used in our testing, we are able to perform 3 to 4 orders of magnitude faster than software.**

I. Introduction

Perhaps the most powerful feature of most modern commercial FPGAs is that they are configured merely by changing bits held in memory. RAM-based configuration allows FPGAs to be quickly reprogrammed an essentially infinite number of times. This reconfigurability opens the door for FPGAs to be extremely flexible and high-performance devices. That said, very few FPGA application developers truly make full use of this capability.

Most FPGA-based applications today only configure the device once – at power-on. This usage pattern makes the FPGA into an essentially static computing platform and the reconfigurability of the system could be viewed as more a liability rather than an asset. Even among applications that do make use of reconfiguration, FPGAs are only generally reprogrammed on a task-by-task basis. That is, an FPGA might run task *A* for 30 seconds before being reconfigured to perform task *B* for the next 30 seconds. In some sense, this is even the model of execution that is used by the canonical example of dynamic reconfiguration: software-defined radios. Reconfiguration is almost never used during the processing of a single task.

Intra-task runtime reconfiguration may be necessary to build practical FPGA-based solutions for many applications. In this paper, we will discuss why dynamic reconfiguration is needed to perform regular expression searching for e-mail spam filtering. We will also investigate the issues that make implementing runtime reconfiguration difficult. We address these concerns by introducing a complete regular expression system-level compiler. This tool automatically divides and executes regular expressions across multiple virtual configurations without user intervention.

II. Regular Expression Searching & FPGAs

Regular expressions are widely used in many different fields, ranging from network intrusion detection to DNA sequencing. Executing regular expression searches on spatial computing devices, such as FPGAs, rather than conventional microprocessors is particularly attractive. This is because FPGAs can take advantage of the huge amount of inherent parallelism present in this search problem. There have been a number of previous research efforts that have performed highly parallel regular expression searching [1][3][4][5] using FPGAs. One issue that these approaches have is that they only consider the case in which the regular expressions are implemented on a single static configuration.

III. Implications of Static Configuration

Only using static configuration can lead to problems for application developers. In this section, we will focus on how the lack of dynamic reconfiguration can cause two troubles for e-mail spam filtering: issues with problem scaling and low resource utilization.

The most serious side effect of static-only FPGA configuration is that it creates a hard capacity limit. One key advantage of microprocessors is that their sequential execution model naturally virtualizes the computational resources. This virtualization is important because it allows the performance of the system to gracefully decline as a problem becomes more complex.

In contrast, if an FPGA application developer only considers static execution, their computation must fit within a single configuration. Although there may be hundreds of thousands of LUTs and flip-flops in the FPGA, if we ignore any time-multiplex sharing built into the circuit, all of the resources are statically allocated. Thus, if the application is run through the traditional set of CAD tools and it turns out that the resource requirements exceed the capacity of the device, the system will simply fail catastrophically.

This is a serious issue for applications such as spam filtering in which the problem size is constantly growing. Although most of the regular expressions used by the filtering process will eventually be retired, far more may be added to take their place in the meantime. With purely static configuration, the only upgrade path to accommodate additional regular expressions is to add more FPGAs to the

system. When the user has enough regular expressions to fill the first FPGA, they must get another. However, as will be discussed below, the first FPGA may not truly be "full".

Since FPGAs can take advantage of so much parallelism, they can actually be too fast for an application. In the case of spam filtering, any real e-mail system will be connected to a network with a fixed incoming capacity. For that matter, while the system may be flooded by mail in short bursts, the latency of message delivery is not terribly important, at least within reasonable bounds. Thus, the processing required during periods of high traffic can be amortized over periods of low traffic.

Statically configured FPGAs cannot take advantage of this fixed data rate and may be underutilized. For example, a user might want to look for 1,000 regular expressions on an e-mail system with a nominal load of 10 Mbps. Let us assume that the user's regular expressions completely fill a single FPGA configuration and that it is capable of processing at a rate of 1 Gbps. This performance is well above the nominal workload and if the user is only able to statically configure the FPGA, the device will be idle 99% of the time.

If the user were able to take advantage of dynamic reconfiguration, they would gain the ability to trade off performance for capacity. They could map multiple sets of searches to different configurations and quickly swap between them. This enables two key features. First, as the number of searches grows, the throughput of the system will gradual decline rather than suddenly fail. Second, when each individual computation is faster than the required data rate, the system can perform multiple computations on the same virtualized fabric to maximize utilization.

IV. Roadblocks to Dynamic Reconfiguration

Despite the advantages of dynamic reconfiguration, it is seldom used. However, this is generally not caused by some intrinsic restriction of the FPGA platform itself. Rather, the problem is that single-configuration application development is the only easy path through existing commercial FPGA CAD tools. There are two fundamental problems that users can face. First, how can we effectively divide a large set of problems into smaller groups that can fit on a given device? Second, how do we actually execute these sub-problems when they are spread across multiple configurations?

The task of dividing a set of regular expressions among multiple configurations can be extremely laborious and time consuming. Coincidentally, this is also a problem if we do not want dynamic reconfiguration, but simply want to spread a problem across multiple FPGAs. Dividing a workload is troublesome because current FPGA CAD tools provide too little feedback too late in the compilation process to be useful.

In order to divide a set of regular expressions into a small number of different configurations, a developer would need to make countless manual iterations through the CAD tools. If the first regular expression could fit on a single configuration, we could try the first five. If these fit, we could try the first ten. If not, perhaps the first three. This trial-and-error search process is troublesome because each run through the mapping tools could take hours. As will be discussed in more detail in Section V.B, this problem can be solved by providing a fast estimate of the resource requirements of each regular expression. After we have this information, we can build a system to automatically partition the problem into smaller sub-tasks.

The simple execution of an application that is spread across multiple configurations is also an issue. This is because such an arrangement requires a custom-made control system to reprogram the device with the correct configuration at the appropriate time and marshal the correct input and output data to and from the various configurations. Developing the software and hardware for such a control system requires manual intervention each time that the regular expressions are modified. This time-intensive and potentially error-prone process can make dynamically configured systems impractical. As will be discussed in Section V.C, this process can be automated so that a user does not need any expert knowledge to map searches to the system.

V. Regular Expression System Generation

To be truly deployable, applications that rely on dynamic reconfiguration cannot be time-consuming to create or require meticulous custom development. In this section we describe a method to automatically generate a complete regular expression execution engine. This system provides a very simple interface that makes the actual implementation and execution of the regular expressions invisible to the user. We first outline the basic architecture of a single configuration. Our discussion continues with a description of how a large set of regular expressions can be divided into a minimal number of difference configurations. Finally, we show how these configurations can be run without user intervention.

A. Regular Expression Compilation and System Design

The process of simply converting a list of regular expressions into gate-level state machines is relatively well understood. Our approach is fairly basic in that we take an incoming list of regular expressions and convert them into Non-deterministic Finite Automata (NFA) using Thompson's Algorithm [6]. These NFA are then turned into one-hot-encoded state machines by using techniques similar to those in [5]. The operations that we support are shown in Fig. 1.

As shown in Fig. 2, each regular expression in our system is turned into a unique state machine. The individual matching units (Fig. 1a) within each state machine are fed by either a byte decoder or a character class ROM. The byte decoder indicates if the current input character matches a single character. On the other hand, a character class ROM is a 256x1-bit memory capable of matching the current character against multiple values (i.e – is the input character a vowel?).

The output of each regular expression is fed to a saturating N-bit counter to determine how many times the regular expression is matched during a given message. These results are captured by an I/O controller that manages the transfer of the input and output data with a system controller running on the host computer.

B. Resource Estimation and Problem Partitioning

To handle problems that require more resources than one configuration can offer, we need an intelligent way to split the searches into smaller, more appropriately-sized groups. This must be done without iterative trial and error through the CAD tools. Towards this end, we present a method to quickly and accurately estimate the resource requirements of a given set of regular expressions.

Our approach begins by estimating the resource requirements of all of the desired regular expressions individually. This is accomplished using the method shown in Fig. 3. Each basic matching unit (Fig. 1a) requires 1 LUT to implement its AND gate. As discussed earlier, if a regular expression uses a character class, it requires a 256x1-bit ROM. On the Virtex-5 device used in our testing, this requires 4 LUTs. All of the other basic operations rely on OR gates. The resource requirements of a given OR gate depends upon its fan-in, as shown in the *orEst* equation.

After the resource requirements of the regular expressions are calculated, we can partition them into separate configurations. As shown in Fig. 4, the partitioning process is given a LUT threshold. This threshold represents the maximum number LUTs a single configuration of regular expressions should require. In our testing, we determined that a reasonable threshold is 88% of the LUTs in the target FPGA. This resulted in good utilization while offering a reliable buffer for consistent placement and routing. This threshold is certainly platform specific, but is likely easy to determine through minimal empirical testing.

During the partitioning process, the system first evaluates the LUTs needed by the I/O controller and byte decoder. After this, it considers every regular expression in turn to determine if the search will fit within the current configuration. If it can, it can be added it to the existing circuit. If not, we create a new configuration and continue. When a configuration is filled, we record the indices of the regular expressions that we put into the configuration. When all of the regular expressions have been split up, we generate the corresponding logic and state machine HDL files for each configuration. These HDL files are sent through the normal CAD toolflow to produce the actual FPGA bitstreams.

Two things should be noted. First, the packing algorithm we use is very simple. Much better utilization may be obtained by performing knapsack solving. However, a knapsack algorithm is only feasible because we can reliably predict the resource utilization of the various regular expressions. Second, our resource estimation is only that – an approximation of the resources required by a regular expression after it is mapped to the hardware. While we would like these estimates to be as accurate as possible, it is critical that these estimations remain a pessimistic upper bound. Any underestimation may result in the CAD tools failing during compilation due to capacity problems.

C. Customized Runtime Support

The last part of our regular expression system generator is responsible for automatically running the searches spread across multiple configurations. Although various aspects of the logic within each individual hardware configuration change depending upon how the regular expressions are split up, the system controller shown in Fig. 2 is the portion of the

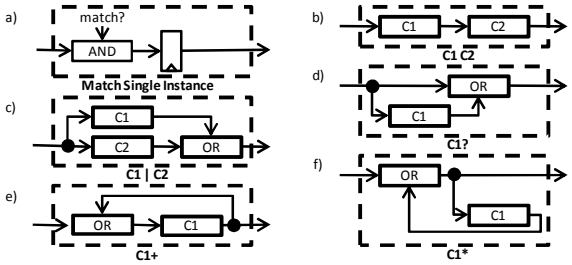

Fig. 1. Gate-level implementations for fundamental NFA operations.

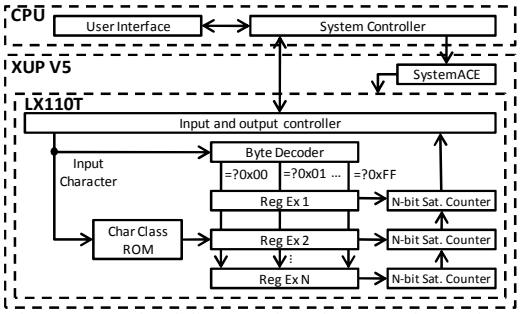

Fig. 2. System-level diagram of regular expression engine.

```
resourceEst(NFA for Reg Ex or sub-expression){
  current LUT count L = 0;
  for all sub-expressions S in X{
    if (S is sub-expression) L += resourceEst(S);
    else if (S is match single char) L += 1;
    else if (S is match char class) L += 1 + charClassLUTs;
    else if (S is OR) L += orEst(S.fanin);
  }
  return L;
}
```

$$orEst(fanin) = \left(\sum_{n=0}^{n=\lfloor dept\,h \rfloor - 1} lutInputs^{n} \right) + \left\lceil lutInputs^{\lfloor dept\,h \rfloor} * \{depth\} \right\rceil$$

where $depth = \log_{lutInputs}(fanin)$

Note: For the Virtex-5, *charClassLUTs* = 4 and *lutInputs* = 6.

Fig. 3. Resource estimation pseudo-code.

```
partition(set of Reg Exes R, LUT threshold T){
  current LUT count L = I/O controller + byte decoder;
  current configuration C.start = 0;
  for all Reg Exes r in R{
    tempLUT = resourceEst(r) + saturating counter;
    if (tempLUT > T - I/O controller + byte decoder)
      exit(-1);
    else if ((tempLUT + L) < T) L += tempLUT;
    else{
      C.end =r.index - 1;
      make new configuration C;
      C.start = r.index;
      L = I/O controller + byte decoder + tempLUT;
    }
    add r to configuration C;
    next r;
  }
  C.end = last r.index;
  return all C information;
}
```

Fig. 4. Partitioning pseudo-code.

```
systemController(configuration bitstreams B,
    configuration information C,  input messages M,
    results buffer R, configuration interval I){
  configure FPGA with B[0];
  currMessageSet = M[0] to M[I];
  currEndMessage = I;
  while (currMessageSet.first < M.last){
    for all bitstreams b in B{
      for all messages m in currMessageSet{
        send message m to FPGA and receive results;
        place results into R[m][b][C[b].start to C[b].end];
      }
      configure FPGA with next b;
    }
    currEndMessage += I;
    currMessageSet = M[I+1] to M[currEndMessage]
      or M[last];
  }
  return R;
}
```

Fig. 5. System controller pseudo-code

engine most seriously affected. The system controller is a C program running on the host PC that provides the user interface. It receives input messages to be processed from the user and returns the completed results. It is also responsible for determining which configuration is mapped to the FPGA, when it is reconfigured, what data to send to the FPGA and what to do with the results that come back from the hardware.

As seen in Fig. 5, the system controller takes in the configuration bitstreams and configuration index information generated from the partitioning process. It also receives the input messages and a results buffer from the user. The last parameter given to the system controller is a configuration interval. The configuration interval determines how many messages we process sequentially before we reconfigure the device with another bitstream. Before execution is started, the input messages are divided into sets of *I* messages. The configuration interval can affect the performance of the system because, as we will discuss in Section VI, reconfiguration can be relatively time consuming. Increasing the configuration interval allows us to reconfigure fewer times and amortize the reconfiguration delay that we do incur over more messages.

Execution begins by mapping the first bitstream to the FPGA. Then, each of the messages in the current set of input data is sent to the FPGA for processing. The results that return from the FPGA are placed in the results buffer. The results are reordered based upon which message they correspond to, which configuration the message was processed with, and what regular expression indices were mapped to that particular configuration. When the last message in the current set has been processed with the first configuration, the system controller reconfigures the FPGA with the next bitstream. When all of the messages in current input data set have been processed through all of the configurations, the system controller moves to the next set of messages.

VI. Testing and Results

We tested our automated regular expression engine using a set of ~49.6K regular expressions. These searches represent the complete filtering list used for all e-mail received by the Microsoft domain. All but the 14 largest regular expressions in this list were implemented in our evaluation. The remaining 14 search terms use extensive nested quantification, resulting in circuits that require 50% or more of the resources provided by our target platform, the Virtex-5 LX110T on the Digilent XUP-V5 board. Synthesis, placement and routing were performed using the tools in ISE 10.1.

Our first experiment involved testing our resource estimation and partitioning tool. The tool divided the 49.6K regular expressions into 45 configurations. All 45 configurations successfully placed, routed, and met timing constraints for operation at 125 MHz. Across all 45 configurations, our tool overestimated the resource requirements by an average of 7.7%. This suggests that, although sufficient for our proof-of-concept system, we may benefit from using a more sophisticated estimation algorithm that can account for some of the optimizations performed by the Xilinx tool during synthesis.

On the other hand, our simple packing algorithm works acceptably. As mentioned earlier, our target was filling the device to 88% capacity. We averaged 78.2% utilization across all of the configurations. Taking into account our average 7.7% overestimate during partitioning, we are likely coming very close to our desired resource utilization.

We also tested the performance potential of our system. For comparison, we used a single-threaded software implementation running on a E6850 Core 2 Duo machine with 4GB of RAM. Five different sets of regular expressions were tested with 1.1K, 2.2K, 4.5K, 8.9K and 49.6K searches. The input messages used for execution were taken from the Enron mail corpus in [2]. The best results from 3 independent runs are shown in Fig. 6.

The four sets of searches with 1.1K, 2.2K, 4.5K and 8.9K regular expressions were also mapped to the FPGA. These lists required 1, 2, 4 and 8 configurations, respectively. Unfortunately, due to technical considerations we were not able to test the hardware using the full set of regular expressions. As seen in Fig. 2, our current implementation relies on a SystemACE controller [7] to reconfigure the FPGA. We used this setup because ISE 10.1 does not implement support for partial reconfiguration on the Virtex-5. Thus, our options for implementing dynamic reconfiguration were relatively limited. The SystemACE offers the capability of reconfiguring the FPGA with up to 8 bitstreams held on a CompactFlash card. Since we were limited to 8 configurations, the breadth of our performance testing was limited. Testing on the hardware was repeated multiple times using configuration intervals between 1 (reconfigure once for every bitstream needed during the processing of each message) and 32K (reconfigure once for every bitstream needed during the processing of every group of 32K messages).

All of our testing results assume that the regular expressions have been pre-compiled (either into NFAs for the software version or into bitstreams for the FPGA) and that all necessary data begins and ends in the CPU's main memory. The software results only include the actual search time, while the FPGA results also include the CPU⇔FPGA transfer time and the SystemACE reconfiguration time.

Looking at Fig. 6, we can make several interesting observations. First, as the number of regular expressions is increased, the performance of the software-based searches degrades faster than that of hardware-based searches. This is likely because while a small number of regular expressions can be implemented in software within the cache, as the number of regular expressions is increased the system very quickly requires the capacity of main memory.

A second observation is that the performance of our hardware implementations scales extremely predictably. For a given configuration interval, the hardware's performance almost exactly halves when we double the number of configurations used from 2 to 4 to 8. This means that, with a fair degree of confidence, we can extrapolate the performance of the hardware implementation if the SystemACE were able

TABLE I . Average Matching Rate (Normalized to CPU Results)

Average Matching Rate (norm)		1.1 K RE (1 confi)	2.2 K RE (2 config)	4.5K RE (4 config)	8.9K RE (8 config)	49.6K RE (45 config)
CPU		**1.00**	**1.00**	**1.00**	**1.00**	**1.00**
FPGA	**I = 1**	601.19	0.15	0.25	0.93	*1.12*
	I = 2	693.68	0.31	0.50	1.87	*2.25*
	I = 4	819.80	0.62	1.00	3.73	*4.49*
	I = 8	819.80	1.24	2.00	7.46	*8.98*
	I = 16	901.79	2.47	3.99	14.90	*17.94*
	I = 32	901.79	4.94	7.97	29.70	*35.76*
	I = 64	901.79	9.89	15.94	59.03	*71.05*
	I = 128	901.79	22.85	37.05	138.34	*166.52*
	I = 256	901.79	44.41	71.97	262.36	*315.81*
	I = 512	901.79	84.67	133.16	499.89	*601.73*
	I = 1K	901.79	147.32	252.03	917.17	*1104.01*
	I = 2K	901.79	266.47	386.34	1423.40	*1713.36*
	I = 4K	901.79	421.40	674.99	2490.95	*2998.38*
	I = 8K	901.79	584.52	932.12	3479.42	*4188.22*
	I =16K	901.79	724.81	1174.48	4340.66	*5224.90*
	I = 32K	901.79	823.65	1365.67	4925.92	*5929.39*

"I=" refers to configuration interval. Italics indicate extrapolated results.

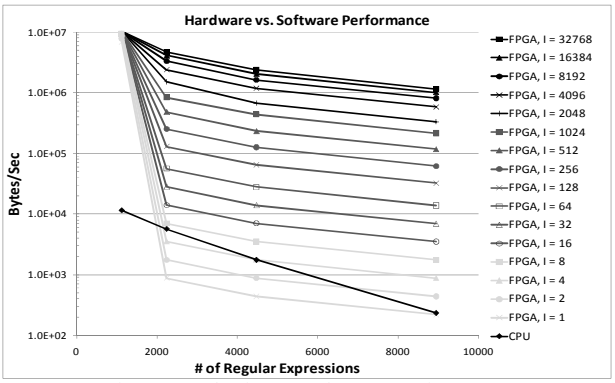

Fig. 6. Graph of CPU and FPGA performance.

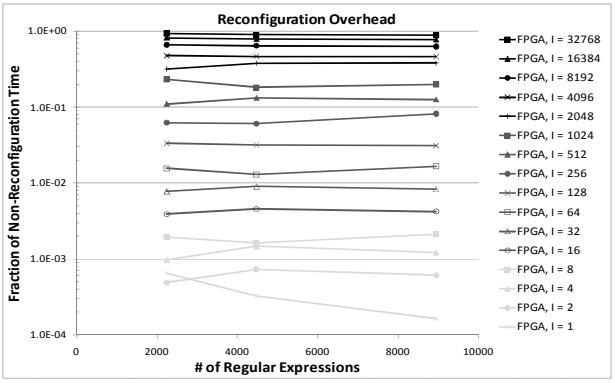

Fig. 7. Graph of time spent **not** waiting for device reconfiguration.

to accommodate 45 configurations. The estimated performance of searching for the full set of regular expressions is shown with italicized data in Table I.

A third observation is that the amount of reconfiguration we perform can drastically affect performance. Looking at Table I, increasing the configuration interval by a factor of 2 almost exactly doubles the achievable performance. Largely, this is because the reconfiguration time dominates the runtime of most of the hardware tests – the SystemACE on the XUP board requires ~1.5 seconds to complete each reconfiguration. As shown in Fig. 7, the tests that used $I \leq 32$ spent 99% or more of their time waiting for reconfiguration. This likely indicates that finding a faster reconfiguration mechanism is a high priority.

Looking at Table I and Fig. 7 together, we can see that the massive parallelism that the FPGA implementations offer can still overcome the handicap of the reconfiguration overhead. For example, searching for 8.9K regular expressions using I = 128, the hardware spends 97% of its time reconfiguring. It only spends 3% of its runtime transferring data and actually executing. However, it still manages to perform 138x faster than the software implementation. The achievable speedup also increases with larger configuration intervals. Looking I = 512, the speedup over software is 500x. At I = 32K, the speedup is nearly 5000x.

VII. Conclusions

In this paper we have shown that dynamic reconfiguration is necessary to perform fast and flexible regular expression searching on an FPGA. However, we highlighted two problems that can discourage application developers from using dynamic reconfiguration. First, when a user has a large set of problems that cannot be implemented on a single configuration, the existing toolflow makes it very difficult to intelligently split them across multiple configurations. Second, executing an application that is spread across multiple configurations requires manual customization.

We solved these problems by developing an automated regular expression system compiler. This tool uses fast resource estimation so that it can divide a set of regular expressions among a minimal number of separate configurations. Once the application has been split, it can be run using an automatically generated controller that manages device reconfiguration and I/O marshalling. During testing, we showed that this system can achieve very high performance. Although we believe that it could benefit from a faster reconfiguration mechanism, we were able to perform up to 5000x faster than a software implementation with only very basic reconfiguration support. When we are able to incorporate partial reconfiguration in the future, this will be a sophisticated and deployable regular expression system.

Overall, dynamic reconfiguration gives FPGAs a capability essential to any practical computing platform: resource virtualization. This is an underutilized and relatively poorly understood area of FPGA research. Further work is necessary to make this feature truly accessible to application developers.

References

[1] J. Bispo, I. Sourdis, J. Cardoso, and S. Vassiliadis, "Regular Expression Matching for Reconfigurable Packet Inspection," *IEEE Conference on Field Programmable Technology*, 2006, 119 – 126.

[2] B. Klimt and Y.Yang, "Introducing the Enron Corpus," *Conference on Email and Anti-Spam*, 2004.

[3] S. W. Lee, S. H. Hwang, and N. Park, "A High Performance NIDS using FPGA-based Regular Expression Matching," *ACM Symposium on Applied Computing*, 2007, 1187 – 1191.

[4] C. H. Lin, C. T. Huang, C. P Jiang, and S. C. Chang, "Optimization of Regular Expression Pattern Matching Circuits on FPGAs," *Conference on Design, Automation and Test in Europe*, 2006, 12 – 17.

[5] R. Sidhu, and V. K. Prasanna, "Fast Regular Expression Matching using FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001, 227 – 238.

[6] K. Thompson, "Regular expression search algorithm," *Communications of the ACM* 11(6) , June 1968, 419 – 422.

[7] Xilinx Inc., "System ACE CompactFlash Solution," DS080 v2.0, 2008.