Automated Dynamic Reconfiguration for High-Performance Regular Expression Searching

Ken Eguro

Microsoft Research
Redmond, WA
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 that runtime reconfiguration is not more commonly used 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 platforms available today contain an SRAM-based FPGA alongside a non-volatile memory. In the most common use, the non-volatile memory will feed a single configuration to the FPGA at power-on and this configuration will remain resident until power-off. With this type of use, the reconfigurability of the FPGA could be viewed as more a liability rather than an asset. Although such a system could perform firmware updates and the like, the FPGA is really being used as a static computing platform.

Even among applications that do make use of reconfiguration, FPGAs are only generally reprogrammed on a task-by-task basis. For example, 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: problems in the classical design methodology, limitations of the conventional CAD toolflow, and restrictions of the common execution model. 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. Regular expression searches have a few characteristics that make implementing them on spatial computing devices, such as FPGAs, rather than conventional microprocessors particularly attractive.

First, as shown in Fig. 1, regular expression searching is eminently parallel. In this example we would like to search an input stream for the words "CAT", "POP" and "POD". With very simple parts (AND gates, flip-flops, and logic that can match given letters) we can fully exploit the available parallelism and search for all three strings simultaneously. An equivalent memory-efficient search running on a microprocessor would generally need to service these searches serially. Since the number of regular expressions that can be found in parallel on an FPGA is only a function of the capacity of the device, it is possible for an FPGA-based solution to have essentially constant throughput, regardless of the number of regular expressions implemented. This is in stark contrast to software in which the performance degrades linearly with respect to the number of desired searches.

A second advantage that FPGA-based regular expression searching has is that the circuits we build can have completely deterministic performance. The circuit in Fig. 1 will always be able to accept one new character of input data per clock cycle, regardless of the searches desired or content of the input data. The throughput of software, on the other hand, can

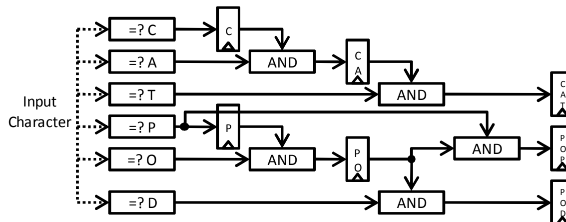

Fig. 1. Circuit to search for "CAT", "POP" and "POD" in parallel.

depend upon the nature of the searches desired (more or less complex regular expressions) and the nature of the input data (an input stream that has a high hit ratio versus one with a low hit ratio). This is on top of that fact that software is generally beholden to unpredictable events such as cache misses.

The natural affinity that regular expression searching has for reconfigurable devices has lead to a number of previous research efforts. Although some of these projects have suggested systems more akin to processor-based solutions [1] [3], most have, similar to our example in Fig. 1, spread the regular expression processing spatially to fully exploit parallelism [2][9][12][15]. 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

Although FPGAs are capable of very high throughput parallel processing, 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.

A. *Hard Capacity Limit*

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 on a reconfigurable computing platform, their computation must fit within a single configuration. Although there may be hundreds of thousands of LUTs and flip-flops on 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 – suddenly and catastrophically.

This may not be an issue for applications in which the resource requirements are constant. However, spam is a constantly growing problem. While we might have 1,000 regular expressions today, tomorrow we will have the same 1,000 plus a few more. While most of the regular expressions 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 in the next section, the first FPGA may not truly be "full".

B. *Inefficient Resource Utilization*

Since FPGAs can take advantage of so much parallelism, they can actually be too fast for an application. This is because spam filtering is a naturally fixed data rate application. 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.

That said, 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. If the user is only able to statically configure the FPGA, the device will be idle 99% of the time.

C. *Dynamic Configuration*

If the user were able to take advantage of dynamic reconfiguration, they could use it in two different ways. One option is that the user could plan for the future. Rather than purchasing an additional FPGA when they need 1,001 regular expressions, they could map new searches to other configurations and quickly swap between them. Ignoring some practical considerations for the moment, this technique will increase the potential capacity of the system to 100K regular expressions while still maintaining adequate performance. Beyond this point, the user still has the option of running additional regular expressions with a gradually increasing penalty in terms of throughput.

Another option is that the user could simply purchase a cheaper platform with a smaller FPGA. In this case, the user could opt to split their regular expressions across 100 different configurations mapped to a device $1/100^{th}$ the size. The various configurations could be quickly swapped onto the FPGA and still keep up with the expected workload. This would increase the utilization of the board.

Either of these options give the user better alternatives to either living with a fixed amount of logic, cursing the system because a computation is too large and errors out in the CAD tools, or buying lots of potentially underutilized devices.

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 monolithic, single-configuration application development is the only easy path through existing commercial FPGA CAD tools. Simply put, spreading a computation across multiple configurations can be a long and complicated process. 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 allow 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 a large problem because each run through the mapping tools could take hours. This is an even more daunting task if we consider re-ordering the regular expressions to maximize utilization. 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 control can be automated so that a user does not need any special knowledge regarding FPGAs or hardware design to use 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 [16]. These NFA are then turned into one-hot-encoded state machines by using techniques similar to those in [15]. The operations that we support are shown in Fig. 2.

Fig. 2a shows the structure of the most basic unit in our system. This building block can match a single character of input data against a single character in a regular expression. As will be discussed later, this structure can be extended to also match against a single class of characters. Fig. 2b shows that these simple units can be concatenated together to allow multi-character strings. This is accomplished by simply feeding the output of one matching unit into the input of another. Fig. 2c shows how two sub-expressions can be OR-ed together. Fig. 2d shows how we can match zero or one instance of a sub-expression. Similarly, we can also match zero or more or one or more instances of a sub-expression. This is shown in Fig. 2e and Fig. 2f, respectively. More sophisticated regular expression operators such as bounded and unbounded quantification ({N}, {N, M}, {N, }) can be implemented by combining these basic operations.

As shown in Fig. 3, each regular expression in our system is turned into a unique state machine. The individual matching units (Fig. 2a) within each state machine are fed by either a byte decoder or a character class ROM. The byte decoder simply indicates if the current input character matches a single value between 0 and 255 (is the input character an 'a'?). On the other hand, a character class ROM is a 256x1-bit memory capable of matching the current character against multiple values (is the input character a digit?).

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 control structure. This I/O controller manages the transfer of the input data and output results to/from a system controller running on the host computer.

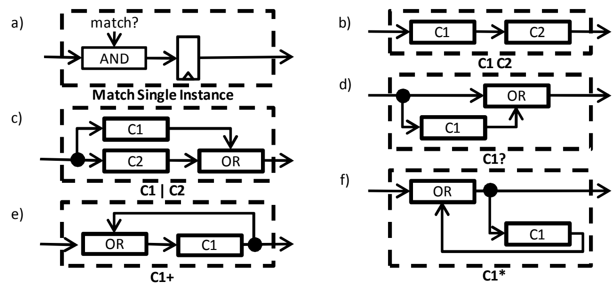

Fig. 2. Gate-level implementations for fundamental NFA operations

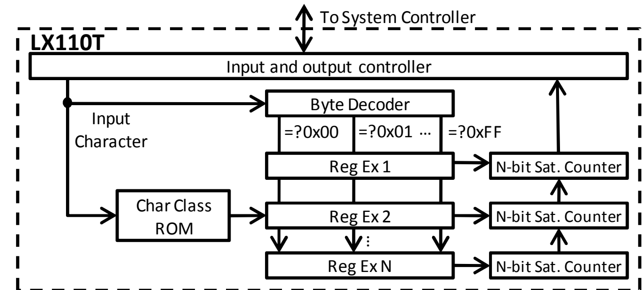

Fig. 3. High-level diagram of the regular expressions mapped to one configuration.

B. Resource Estimation and Problem Partitioning

The conversion of set of regular expressions into a single monolithic configuration is only the beginning of our solution. 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. 4. Each basic matching unit (Fig. 2a) 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. Each of the 6-input LUTs in the Virtex-5 can accommodate up to a 6-input OR. Wider expressions require a tree of cascaded LUTs. The resource requirements of a given OR gate can be computed using the *orEst* equation.

Notice that we only track the LUT requirements of the regular expressions and not the number of flip-flops. This is acceptable for the platform used in our testing because each slice in the Virtex-5 contains 1 flip-flop per LUT. The largest ratio of flip-flops to LUTs needed for any of our basic units is 1:1. Platforms with fewer flip-flops per LUT may need to also track the number of flip-flops used.

After the resource requirements of the regular expressions are calculated, we can partition them into separate configurations. As shown in Fig. 5, we give the partitioning process 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, each configuration first evaluates the LUTs needed by the I/O controller and byte decoder. After this, we consider every regular expression in turn to determine if it could fit within the current configuration. If it can, we add it to the system. If not, we create a new configuration and continue. When a configuration is filled, we record the indices of 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.

It is possible that a single regular expression may be too large to fit on the target device. Although a single regular expression could be spread across multiple configurations, we did not deal with this situation in our proof-of-concept system. Along the same lines, 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.

It should also be noted that our resource estimation is only that – an approximation of the resources required by a regular expression after it is mapped to the hardware. We do not consider any optimizations that the synthesis tool might make. For example, if two regular expressions in the same configuration use the same character class, the Xilinx toolflow will realize that the ROMs are identical and remove one from the system. Although this inaccuracy may result in overestimating the hardware requirements in a configuration, the resource estimation routine could be modified to identify and compensate for these compiler optimizations. What is critical, though, is that these estimations remain a pessimistic upper bound. Any underestimation may result in the CAD tools failing during compilation due to capacity problems.

```
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;
}
```

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

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

Fig. 4. Resource estimation pseudo-code. For the Virtex-5, charClassLUTs = 4 and lutInputs = 6.

```
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. 5. Partitioning pseudo-code.

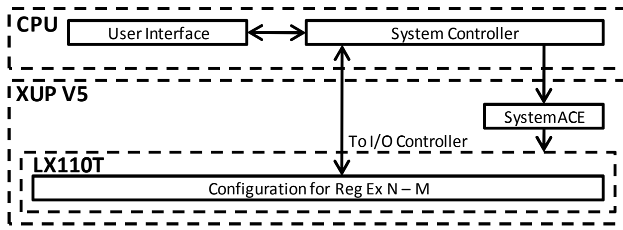

Fig. 6. System-level diagram of regular expression engine.

```
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. 7. System controller pseudo-code.

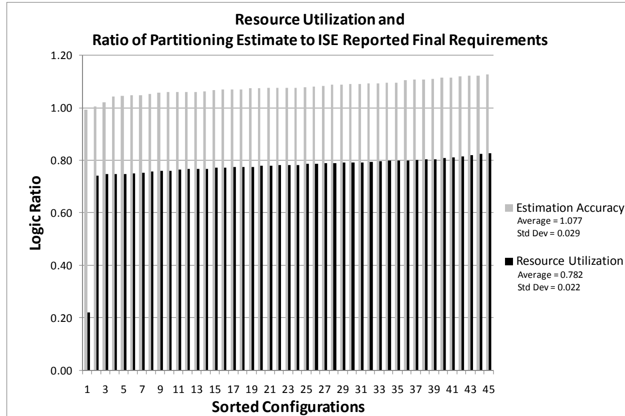

Fig. 8. Sorted graphs of resource utilization and estimation accuracy – 49K regular expressions split into 45 configurations. The two sets of data are sorted independently. The reported average and standard deviation of resource utilization does not include the spurious 0.22% utilization data point that resulted at the end of the regular expression list.

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. 6 is portion of the 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. 7, 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 during execution 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 were obtained from the IT system engineers responsible for spam filtering at *large corporation*. They represent the complete white and black term-lists used on all messages received by the 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.

Fig. 8 shows the results of 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. In gray, this graph shows the sorted ratios of the LUT requirements estimated by our tool during the partitioning phase versus the LUT/FF pair requirements as reported by ISE after compilation. 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.

Looking at this data more carefully, though, there is the potential for one small problem. As discussed earlier, the resource estimation must be an overestimate to avoid encountering possible capacity problems. Our tool very slightly underestimated (< 1%) the resource requirements for one configuration. This small discrepancy may be the result of some unforseen constraint for logic block packing or some logic duplication performed during compilation to help meet timing. Nevertheless, this small underestimation is unlikely to cause problems in practice since we are only targeting 88% of the platform's capacity. Purposefully underutilizing an FPGA by 10-15% is often customary to ensure successful placement and routing. That said, with further investigation this behaviour could be characterized and the estimation algorithm could be modified to compensate.

On the other hand, our simple packing algorithm seems to work acceptably. In black, Fig. 8 shows the sorted resource utilization of the configurations as reported by ISE after compilation. As mentioned earlier, our target was filling the device to 88% capacity. Disregarding the small final configuration that formed from the last handful of regular expressions in the list, we averaged 78.2% utilization. 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 2.54GHz/6MB L2/4GB RAM Core 2 Duo machine. 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 [8]. These messages had an average message size of 3.1KB. The best results from 3 independent runs are shown in Fig. 9.

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. 6, our current implementation relies on a SystemACE controller [18] 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. We elected to use the SystemACE because this offered the capability of reconfiguring the FPGA with up to 8 bitstreams held on a CompactFlash card. Since the SystemACE can only switch between these 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 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. 9, 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 (ignoring when we only have a single configuration). This means that, with a fair degree of confidence, we can extrapolate the performance of the hardware implementation if the SystemACE were able to accommodate 45 configurations. The estimated performance of searching for the full set of regular expressions is shown with italicized data in Tables I and II.

A third observation is that the amount of reconfiguration we perform can drastically affect performance. Looking at Table II, 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. 10, 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.

On the other hand, though, looking at Fig. 10 and Table II, 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. That said, the achievable speedup does increase with larger configuration intervals. Looking $I = 512$, the speedup over software is 500x. At $I = 32$K, the speedup is nearly 5000x.

One concern with large configuration intervals is that they may not be practical in a real e-mail system. Batching a large number of messages together may introduce unacceptable latency or may require too much buffering for input message replay and results reordering. To perform some general calculations concerning the buffering requirements, let us assume that the input messages are an average of 3.1KB a

TABLE I
AVERAGE MATCHING RATE (RAW DATA)

Average Matching Rate (bytes/sec)		1.1 K Reg Ex (1 FPGA config)	2.2 K Reg Ex (2 FPGA configs)	4.5K Reg Ex (4 FPGA configs)	8.9K Reg Ex (8 FPGA configs)	49.6K Reg Ex (45 FPGA configs)
	CPU	1.14E+4	5.68E+3	1.75E+3	2.34E+2	3.55E+1
FPGA	**I = 1**	6.86E+6	8.74E+2	4.37E+2	2.18E+2	*3.64E+1*
	I = 2	7.92E+6	1.75E+3	8.77E+2	4.38E+2	*7.31E+1*
	I = 4	9.36E+6	3.51E+3	1.75E+3	8.77E+2	*1.46E+2*
	I = 8	9.36E+6	7.02E+3	3.51E+3	1.75E+3	*2.92E+2*
	I = 16	1.03E+7	1.40E+4	7.00E+3	3.50E+3	*5.83E+2*
	I = 32	1.03E+7	2.81E+4	1.40E+4	6.97E+3	*1.16E+3*
	I = 64	1.03E+7	5.62E+4	2.79E+4	1.39E+4	*2.31E+3*
	I = 128	1.03E+7	1.30E+5	6.49E+4	3.25E+4	*5.41E+3*
	I = 256	1.03E+7	2.52E+5	1.26E+5	6.16E+4	*1.03E+4*
	I = 512	1.03E+7	4.81E+5	2.33E+5	1.17E+5	*1.96E+4*
	I = 1K	1.03E+7	8.37E+5	4.42E+5	2.15E+5	*3.59E+4*
	I = 2K	1.03E+7	1.51E+6	6.77E+5	3.34E+5	*5.57E+4*
	I = 4K	1.03E+7	2.39E+6	1.18E+6	5.85E+5	*9.75E+4*
	I = 8K	1.03E+7	3.32E+6	1.63E+6	8.17E+5	*1.36E+5*
	I =16K	1.03E+7	4.12E+6	2.06E+6	1.02E+6	*1.70E+5*
	I = 32K	1.03E+7	4.68E+6	2.39E+6	1.16E+6	*1.93E+5*

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

TABLE II
AVERAGE MATCHING RATE (NORMALIZED TO CPU RESULTS)

Average Matching Rate (norm)		1.1 K Reg Ex (1 FPGA config)	2.2 K Reg Ex (2 FPGA configs)	4.5K Reg Ex (4 FPGA configs)	8.9K Reg Ex (8 FPGA configs)	49.6K Reg Ex (45 FPGA configs)
	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.

piece and that we would like 1-bit saturating counters for each regular expression (only determine if a regular expression was present or not present in a message). For 8.9K searches and I = 128, the system controller would only need about 530 KB of buffering (input replay: 128 messages * 3.1KB/message ≈ 390KB, reordering: 128 messages * 8.9K regular expressions * 1 bit/regular expression/message ≈ 140KB). In comparison, for the same 8.9K searches and I = 32K, the system controller would need about 130MB of buffering space.

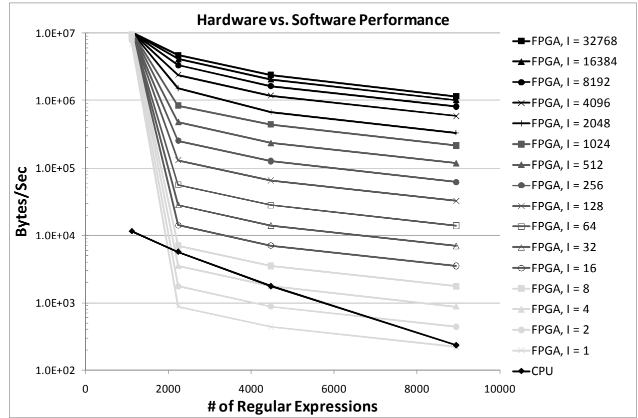

Fig. 9. Graph of CPU and FPGA performance. I refers to the configuration interval.

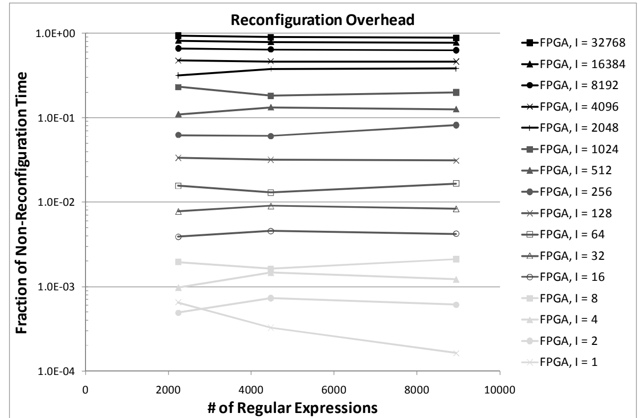

Fig. 10. Graph of time spent during FPGA execution **not** waiting for device reconfiguration. I is the configuration interval.

Even though very large configuration intervals may be not be useful, the data collected during these tests provide motivation for future work. This is because if we had access to a faster reconfiguration mechanism, we could achieve much better performance with small configuration intervals. For example, if we were to shrink the reconfiguration time by a factor of 10-20x (from 1.5 seconds to 75-150 milliseconds), we expect the utilization curves in Fig. 10 to slide up correspondingly. This would allow us to obtain level of performance that currently requires I = 32K with only I = 256 or I = 128.

VII. RELATED WORK & FUTURE DIRECTIONS

Dynamic reconfiguration is defining characteristic of RAM-based FPGAs. Thus, many previous projects have looked into ways of using it or improving it. For example, runtime reconfiguration is a natural part of research efforts that explore dynamically extensible processors such as [6] and [17]. These systems feature a CPU that can issue customized instructions to a tightly coupled FPGA. However, since the RFU operations that these systems generally map to the reconfigurable fabric are fairly small and have limited or fixed

I/O requirements, they do not encounter the same capacity and data marshalling problems as a system for larger, more open-ended applications such as regular expression searching.

There have also been multiple papers that have focused on multi-context FPGAs [7][14] or bitstream compression/ caching [11][10] to improve reconfiguration time. Unfortunately, none of these techniques have been adopted by commercial FPGA manufacturers. As we mentioned earlier, although our results could be improved dramatically with access to faster reconfiguration schemes, we are still able to achieve compelling results with the level of support present in today's commercial devices.

That said, we would like to incorporate partial reconfiguration through the internal ICAP in the future. To accommodate this, the I/O controller in Fig. 3 would reside in static logic. In addition to its current duties, it would also be responsible for reconfiguring the regular expression logic in rest of the FPGA. The bitstreams for the regular expressions could come directly from the system controller or could be stored in an external memory. Such an implementation would provide two benefits. First, it would remove the 8 configuration limit that currently restricts the system. Second, since the partial bitstreams would be smaller and the internal ICAP is much faster than the JTAG from the SystemACE, reconfiguration could be performed considerably faster. However, the concept of partial reconfiguration may introduce some problems that we didn't address in this paper. For example, [5] suggested a framework for simplifying the process of dynamically scheduling the hardware resources in the system and moving or reorienting partial bitstreams.

Finally, as we mentioned in Section VI, we would also like to investigate more sophisticated resource estimation algorithms. There has been previous research [4] [13] looking into estimating the detailed resource requirements of applications that are specified at a relatively high level. We believe that incorporating some of these ideas could improve our estimation accuracy and resource utilization.

VIII. 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. The first issue is that that 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. The second concern is that executing an application that is spread across multiple configurations requires a user to carefully design a custom control structure.

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 divided adequately, 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] Z.K. Baker and V. K Prasanna, "Time and Area Efficient Pattern Matching on FPGAs," *ACM Symposium on Field-Programmable Gate Arrays*, 2004, 223 – 32.

[2] 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.

[3] I. Bonesana, M. Paolieri, and M. D. Santambrogio, "An Adaptable FPGA-based System for Regular Rexpression Matching," *Conference on Design, Automation and Test in Europe*, 2009, 1262 – 1267.

[4] C. Brandolese, W. Fornaciari, and F. Salice, "An Area Estimation Methdology for FPGA Based Designs at SystemC-Level," Design Automation Conference, 2004, 129 – 132.

[5] J. Burns, A. Donlin, J. Hogg, S. Singh, and M. De Wit, "A Dynamic Reconfiguration Run-time System," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1997, 66 – 75.

[6] W. W. S. Chu, R. G. Dimond, S. Perrott, S. P. Seng, and W. Luk, "Customizable EPIC Processor: Architecture and Tools," *Conference on Design, Automation and Test in Europe*, 2004, 236 – 241.

[7] D. Jones and D. Lewis, "A Time-Multiplexed FPGA Architecture for Logic Emulation," *IEEE Custom Integrated Circuits Conference*, 1995, 495 – 498.

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

[9] 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.

[10] Z. Li, K. Compton, and S. Hauck, "Configuration Cache Management Techniques for FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2000, 22 – 36.

[11] Z. Li and S. Hauck, "Configuration Compression for Virtex FPGAs," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 2001.

[12] 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.

[13] P. Milder, M. Ahmad, J. Hoe and M. Puschel, "Fast and Accurate Resource Estimation of Automatically Generated Custom DFT IP Cores," *ACM Symposium on Field-Programmable Gate Arrays*, 2006, 211 – 220.

[14] S. Scalera and J. Vazquez, "The Design and Implementation of a Context Switching FPGA," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1998, 78 – 85.

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

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

[17] M. Wirthlin and B. Hutchings, "A Dynamic Instruction Set Computer," *IEEE Symposium on Field-Programmable Custom Computing Machines*, 1995, 99 – 107.

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