2010 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines

BURAQ: A Dynamically Reconfigurable System for Stateful Measurement of Network Traffic

Faisal Khan, Nicholas Hosein, Scott Vernon, Soheil Ghiasi
University of California, Davis
{fnkhan, nhosein, stvernon, ghiasi}@ucdavis.edu

***Abstract*—Programmable analysis of network traffic is critical for wide range of higher level traffic engineering and anomaly detection applications. Such applications demand stateful and programmable network traffic measurements (NTM) at high throughputs. We exploit the features and requirements of NTM, and develop an application-specific FPGA based Partial Dynamic Reconfiguration (PDR) scheme that is tailored to NTM problem. PDR has traditionally been done through swapping of statically compiled FPGA configuration data. Besides being latency intensive, static compilation cannot take into account exact requirements as they appear during real-time in many applications like NTM. In this paper, we make novel use of fine-grained PDR, performing minute logic changes in real-time and demonstrate its effectiveness in a prototype solution for programmable and real-time NTM. We specifically make use of flexibility available through the application and present a number of novel tools and algorithms that enabled developing the BURAQ system. Our results show 4x area and 1.3x latency improvements of BURAQ from a comparative statically compiled recent solution.**

I. Introduction

Accurate traffic measurement and monitoring is keystone in a wide range of network applications such as detection of anomalies and security attacks, and traffic engineering. A number of critical network management decisions such as blocking traffic to a victim destination, re-routing traffic, or detection of anomalies, require extraction of real-time statistics from network traffic. A high-quality network measurement tool is crucial in judiciously making such decisions [1], [2].

A network packet type can be identified using any number of specific fields in an IP header, commonly referred to as *tuples*. As an instance, packets originating from a particular subnet can be classified as a type of packet. Figure-1 graphically shows an example of two types of packets in two tuple space. We next define a *rule* as a combination on packet types. For example, the two packet types of Figure-1 are combined together in the rule R_1 as shown in Table-I.

Packet classification problem has seen a number of FPGA implementations [3], [4]. The problem fundamentally involves matching the incoming packets with a prioritized rule-set and taking an action per incoming packet corresponding to the highest matched rule. Thus packet classification requires no knowledge of past packets, or in other words it is memoryless or stateless. On the other hand, our problem of Network Traffic Measurement (NTM) involves maintaining *state* of network over a period of time which is later analyzed to take an action. One form of network state could be quantification of different types of traffic observed. Such an information is useful for network managers in a wide variety of applications such as traffic engineering, accounting, routing and sometimes observing network for any type of malicious activity like DOS attacks.

Maintaining exhaustive state information as a count for even a single tuple, e.g. 32-bit source IP addresses, requires excessively large amount of storage. The overheads and latencies associated with accessing high-density storage media limits maintaining individual count information in real-time at backbone line speeds. The issue has been traditionally resolved using conservative packet sampling, which discards most of traffic from consideration [5], [6].

Another solution to limit storage requirement is to maintain state of *interesting* traffic only. This involves maintaining count information for only the packets that the user defines in *rules* [7]. A rule can be viewed as a set operation (union, intersection, etc) on a set of prefixes. The prefixes can be any of the tuples found in the packet-header. The term prefix here is utilized in a general framework as a notation following CIDR prefix type format. As an instance, Table I illustrates an example of two rules R_1 and R_2 using four prefixes.

We hereby define *rule-processing* as a two step process involving checking of incoming packet with the rule, referred to as *rule-checking* or *rule-matching*, and finally incrementing a *rule-counter* upon a successful match. The rule-counter represents number of packets that matched the rule. In this work, we target a real-time Rule-driven Network Traffic Measurement (RNTM). A RNTM has superior accuracy than sampling based mechanisms, however, it also introduces two new challenges for its real-time implementation: (1) the rule-processing is to be fast enough such that an incoming packet can be compared with potentially complete rule-set in the worst-case, and (2), the solution needs to be programmable enough for frequent dynamic updates in the rule-set.

The rather contradicting requirements of performance and programmability impose challenges for implementation of any RNTM system. The increasing higher network data rates and more complex rule sets dismiss software as a viable implementation platform, requiring custom processing solutions like FPGAs tailored for the application. FPGAs provide an interesting blend of programmability and performance. We exploited this flexibility of FPGA devices previously in developing a prototype for a custom RNTM solution [8]. Our highly parallel solution provided the processing and flexibility

978-0-7695-4056-6/10 $26.00 © 2010 IEEE
DOI 10.1109/FCCM.2010.36

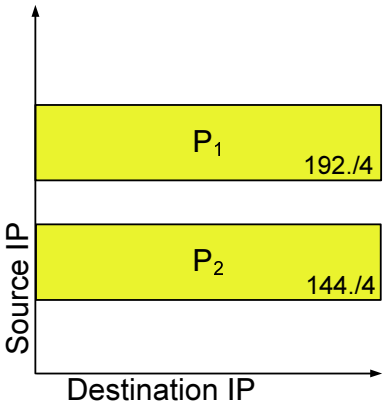

Fig. 1. Two prefixes in 2D space

Two Prefixes
$P_1 =< 144./4, *, *, *, * >$
$P_2 =< 192./4, *, *, *, * >$
$P_3 =< 201./6, *, *, *, * >$
$P_4 =< 212./6, *, *, *, * >$
Rule composition using Prefixes
$R_1 = (P_1 \cup P_2)$
$R_2 = (P_3 \cup P_4)$
Boolean Rule Mapping
$P_1 = s_1.s_2'.s_3'.s_4$
$P_2 = s_1.s_2.s_3'.s_4'$
$P_3 = s_1.s_2.s_3'.s_4'.s_5.s_6'$
$P_4 = s_1.s_2.s_3'.s_4.s_5'.s_6$
$R_1 = s_1.s_3'.(s_2'.s_4 + s_2.s_4')$
$R_2 = s_1.s_2..s_3'.(s_4'.s_5.s_6' + s_4.s_5'.s_6)$

TABLE I
RULE COMPOSITION & BOOLEAN MAPPING

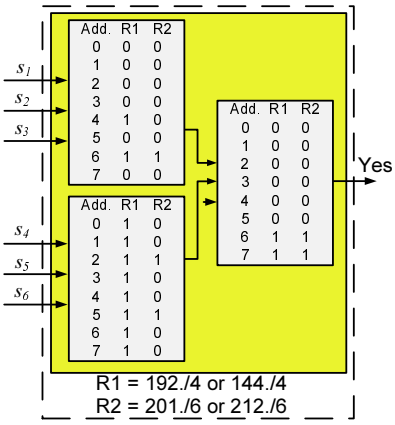

Fig. 2. LUT mapping of the Rule

for real-time needs. However, its large logic footprint for its basic rule-processing component limited its ability to handle a large-enough number of user rules simultaneously.

Partial Dynamic Reconfiguration (PDR) of FPGAs has seen a lot of interest in the research community [9], [10]. Several interesting solutions exploit the feature to further increase the reprogrammability and/or to save logic-area of an FPGA device. However, central to most of the techniques, is either a static compilation or just-in-time compilation of the reconfigurable FPGA-region. The static compilation is infeasible for real-time problems like RNTM where future rules cannot be predicted statically. In this work, we make novel use of fine-grained partial dynamic reconfiguration feature involving very minute logic changes in developing an innovative RNTM-specific dynamic reconfiguration scheme of FPGA fabric. We utilize this underlying idea to develop a novel rule-processing building block, referred to as *Dynamically Reconfigurable Socket (DRS)*. Furthermore, we present algorithms and tools that are developed to enable efficient synthesis of a programmable, scalable architecture, BURAQ, containing many DRSs. We stress that many of our presented techniques and algorithms are quite generic and transcend the needs of the current RNTM application.

We demonstrate 4x area improvement of the BURAQ employing multiple DRS over the previous static-logic based solution, enabling answering of at least 100 64-bit prefixes simultaneously. The 64-bits can easily map 32-bits source and destination tuples or a combination of other tuples in the packet header. In practice, the actual parallelism may even be better depending on the rule-set, thanks to the many aggregation possibilities in prefixes, as was done in Rule-R_1. We also demonstrate a modest 1.3x rule-deployment latency improvement, requiring 2.25ms for a DRS reconfiguration. However, the rule-deployment latency is a function of both Xilinx PDR interface (ICAP) [11] and FPGA PDR latency. In this paper, we also analyze the two and provide insights into latency improvement opportunities using simple updates to Xilinx ICAP software APIs.

II. Conventional Approaches

The rule matching mechanism of non-software RNTM systems can be broadly divided two categories: logic-based bit manipulation or memory look-up. Logic-based implementations usually utilize dedicated static logic, for e.g. a network of comparators, to match the incoming packets against the given rules [4], [12]. Static-logic based implementations usually employ device registers for storing the rules. Our previous work [8] reports such a static-logic implementation in which, the rules are stored in the form of end-points of associated contiguous regions like the ones shown in Figure-1.

The use of registers for rule storage is beneficial as they can easily be overwritten to update the rule-set. However, the scheme demands rather large amount of logic resources to implement bit manipulation functions over the wide word defined by packet header bits. Note that the more complex rules involving disjoint regions in the space defined by header bits have to be decomposed to enable architectural mapping, which increases the required logic resources.

Memory lookup-based solutions typically utilize Ternary Content Addressable Memories (TCAMs). TCAMs have seen applications in some related networking applications like packet classification [13] and IP prefix-matching [14]. TCAMs are typically suited for packet forwarding-type applications, where the comparisons are priority-driven and are performed with regular IP-prefixes. However, the user rule-set in RNTM can be more complex than an IP-prefix in that the rules are usually independent and have equal priority. Thus, mapping of RNTM rule sets to TCAMs would demand disentangling of the rule-implied regions to eliminate the impact of priority-based matching, that may translate to exponential expansion of TCAM entires in the worst case.

III. RNTM-Specific Dynamic Reconfiguration

Look-up tables (LUTs) are the primary logic block of SRAM-based commodity FPGAs. To map a given combinational logic function onto an FPGA, it has to be decomposed into a network of input-constrained single-output auxiliary functions. Such an auxiliary function can be directly mapped

to a LUT [15]. In logic-based RNTM systems, e.g. [8], the network of bit manipulation gates forms the logic function that is mapped to LUTs at design time. The mapping is kept intact at runtime.

We observe that RNTM rules also exhibit specific structure as they are composed from individual prefixes. This can be seen in Table-I where the rules are composed of a significant portion of source address bits, s_i, followed by don't care bits. A rule can therefore be viewed as a *special* Boolean function in that it characterizes only a subset of all possible functions on the packet header bits. Rules typically check for patterns (including wild cards) in adjacent bits, and complex global patterns are unlikely. The rules depicted in Table-I are realistic examples. On the other hand, the rule function $s_1.s_6 + s_1'.s_6'$ is quite unlikely to appear in practice. Having made this observation, we propose to exploit FPGA architectural features to better serve RNTM applications.

Specifically, we propose to directly fuse user rules into LUTs, and thus, utilize the FPGA circuitry for rule-matching. This is in contrast with static-logic based implementations that utilize FPGA registers to store, and other resources like LUTs for building rule-matching circuitry such as comparators. Figure-2 illustrates the idea using an example of three 3-input LUTs that collectively implement the rule R_1 given in Table-I. In this scheme, the LUTs are programmed with entries that yield a *Yes* answer, if the incoming packet header bits match with the programmed rule. This 'Yes' answer is subsequently logged by incrementing a rule-counter, completing the two stages in rule-processing.

A clear advantage of the proposed scheme is its sizable area savings over static-logic based implementations, which translates to processing more rules in parallel on a given FPGA. The flip side, however, is that runtime rule updates become more challenging than logic-based schemes, since it demands Partial Dynamic Reconfiguration (PDR) of FPGA resources [16]. PDR has traditionally been region-based, which refers to reconfiguring FPGA logic and interconnection resources within a defined region of the chip. A condition imposed to perform these changes is to maintain a consistent interface between reconfigurable (dynamic) and non-reconfigurable (static) regions.

In RNTM, the rules usually are dynamically derived. As an instance, a network user may find an abnormal activity in a subnet and wishes to zoom-in further. This operation will involve increasing the size of the original subnet CIDR prefix, recompiling the design and reprogramming the rule-processing unit, similar to what is done in [17] for IP-forwarding. Although such a conventional PDR scheme reduces reconfiguration latency by restricting the region size to be recompiled, the just in-time compilation of the dynamic design is still quite slow. The associated CAD tools that deal with synthesis and mapping of the design onto FPGA resources, especially placement and routing, have very long latencies. As such, it seems impractical to generate designs on-the-fly in real-time.

To address the problem, we exploit the inherent characteristics of RNTM rules, and develop an application-specific reconfigurable unit that can quickly admit new rules at runtime. The basic idea of our design is a generic-enough network of LUTs that are carefully placed and interconnected, such that mapping a new rule only requires updating the content of the LUTs while keeping their placement and routing intact. As an instance, if the rule-matching unit of Figure-2 that is initially programmed with rule-R_1 is to be reprogrammed with rule-R_2, it would only require updating the three LUTs with contents given in the third column, while keeping the placement and routing consistent.

The proposed RNTM-specific fine-grained dynamic reconfiguration only performs minute logic changes at the LUT-level. These changes are expected, and are demonstrated, to be fast, since the latency of just in-time compilation is substantially reduced by eliminating the need for placement and routing, and the reconfigured region is quite small. The characteristics of RNTM problem renders the architecture effective in handling real-life rules.

IV. Dynamically Reconfigurable Rule Socket

We now discuss our rule-processing block, the Dynamically Reconfigurable Rule Socket (DRS or just Socket). A high level design of DRS is shown in Figure-3 targeting 64-bits of rule-matching. The DRS is composed of two high-level components corresponding to rule-processing requirements: a generic *rule-matching* module combined with *state update and control* logic. Both components have static layouts throughout the lifetime of the system.

The rule-checking module utilizes a reduction-tree to check for patterns in adjacent header bits. While the tree is effective for practical rules, it cannot admit a hypothetical rule that refers to a complicated global pattern. The design is intentionally constrained to improve the logic footprint of the module for practical application scenarios by trading off rule-generality. The illustrated rule-checking module is an example that involves 23 4-input LUTs, and admits a rule on 64-bit source and destination addresses. To support more tuples in the rule one would have to add more LUTs, corresponding to the size of the new tuples, to expand the reduction tree.

New rules are dynamically updated or plugged into a socket during runtime. Plugging of a new rule into the DRS follows PDR of rule-matching LUTs. The result of rule matching is forwarded to the *state update and control* logic that maintains a streaming count representing number of packets that have matched the programmed rule. The count is continuously checked against a user-programmed threshold value, that raises a *threshold-met* (or simply *met*) alert whenever the threshold is met. The met signal is basically a socket's unique identification (ID) code in a system.

The DRS goes in idle mode once the threshold has been met. Its count value needs to be reseted before it can proceed with the next round of statistics collection. A simple resetting scheme could be through the use of an explicit *reset* signal going into the DRS. In a system involving multiple DRS, this could be achieved by either having a unique reset signal for

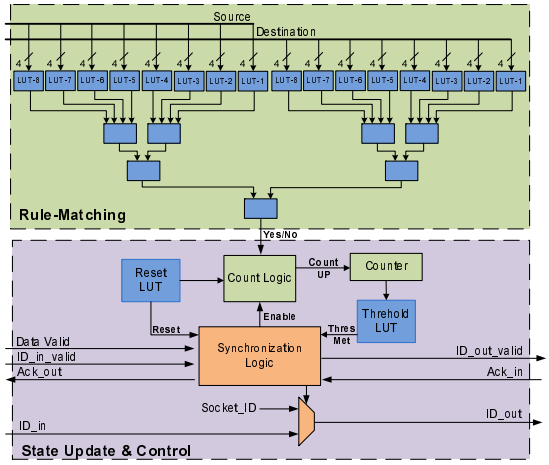

Fig. 3. Socket Architecture

every individual DRS or by pipelining a common reset line across multiple DRS. Whereas the former solution requires multiple reset lines, the latter option involves additional identification bits along with the reset-signal for the reset to be properly recognized at the target DRS. Thus, both solutions incur overheads in terms of FPGA interconnect and logic area.

We propose an innovative remote-synchronization mechanism that completely does away with explicit input-signals. The proposed scheme, used in the context of reset here, works over a dedicated LUT as shown in Figure-4.

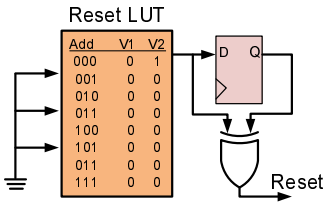

Fig. 4. Reset Mechanism

The Reset-LUT is remotely overwritten using fine grained PDR using a control-processor with a configuration that has an effect of inverting its current output. Two such LUT configurations that can invert the LUT's output are shown in Figure-4 in columns V1 and V2. The associated logic surrounding the LUT then decodes this logic-inversion and generates a signal that can be used for any kind of remote synchronization between the activating processor and custom logic: DRS-reset in the current case. The innovative PDR based synchronization and reprogramming protocols employed in our work not only simplify DRS external interface by reducing the number of DRS IO-pins, but also reduce its logic footprint. We will further discuss the savings when we compare our DRS with a previous static-logic solution in Section-VII.

V. System Design Challenges

We have so far presented our base socket. A practical use of the socket requires its system integration. The integration poses several challenges. In this section, we discuss the challenges and our strategies in dealing with them.

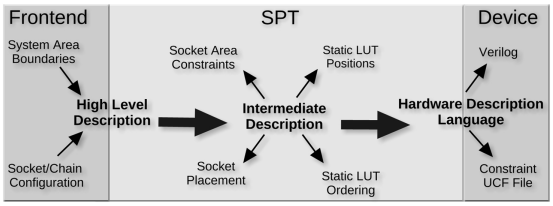

Fig. 5. SPT Software Architecture

A foremost question, that we also briefly discussed in Section-III, is careful placement and interconnection of the socket to enable RNTM-specific PDR. This is because one exactly needs to be aware of two things for reprogramming the LUTs in the socket: (1) their exact FPGA placement, and (2) the LUT pin-mapping to incoming inputs. One would assume that such an information might be readily available during the course of synthesis and placement using the CAD tools. However, the tools do not detail such fine place and route information for an automated retrieval, requiring visual lookups in the complicated routed design. Such a latency intensive step is clearly beyond a network-manager's job description and must need to be addressed. We address the two issues using our socket placement tool and LUT pin-mapping algorithm that will be presented shortly.

The sockets are independent rule-processing units. In practice, one would like to have as many sockets in the system as possible to concurrently process maximum number of rules permitted by the device resources. A multiple-socket design raises possibilities of interesting applications as well as design challenges. One of them being area scaling of the system with increasing sockets. Another subtle, but interesting design question is collection of met-signals by the individual sockets. The challenge here lies in designing a mechanism that guarantees proper collection of the asynchronously generated met-signals within a predictable latency.

A. Socket Placement Tool

We address the issue of LUT identification for rule-update in a socket by locking the LUTs that contribute to the rule-matching functionality to resources with known locations. A Socket Placement Tool (SPT) was developed that automates generation of a multi-socket system and the locking of reprogrammable components within a socket.

The SPT performs three main functions prior to hands-off to synthesis and placement tools:

1) A quick feasibility check of the desired multi-socket system in terms of FPGA area.
2) Evaluation of placement possibilities for sockets and locking down their reprogrammable LUTs to known FPGA locations, i.e. generation of placement constraints.
3) Generation of the RTL Verilog files that correspond to the user required multi-socket system.

Note that only reprogrammalbe LUTs within a socket are constrained. The rest of the socket is left for unconstrained

layout by the CAD tools. However, in doing so, the SPT makes sure that enough FPGA resources are left non-utilized near reprogrammable LUTs. This is done to make sure that a socket components do not get significantly displaced on the FPGA die or else the system will yield poor static-timings.

The three SPT steps are illustrated in Figure-5. The SPT tool relies on empirically developed models for socket placement. For the sake of brevity, we skip the fine details of the models but will present system level results in Section-VII

B. LUT Pin Mapping Detection Algorithm

The knowledge of LUT input pin mapping is critical for PDR as it defines how the LUT-codes need to be assembled. A LUT-code is a bit-vector whose individual bits define the LUT output corresponding to an input that it uses as an address. As an instance, the two pin-mappings A and B as shown in Figure-6, correspond to same LUT functionality but different LUT codes, $B1$. The Xilinx CAD tools, that we employ in our work, can internally come up with any such pin-mappings which are also not readily available to an end-user. One way to avoid this is by having the SPT lock down the pin-mappings as well at the expense of reduced routing flexibility.

We present a novel algorithm using which the LUT pin mapping can be evaluated even after the sockets have been programmed into the FPGA, thus making the pin-locking constraints redundant and letting the CAD tools have full routing flexibility. Our technique involves programming the LUTs with *beacon-codes* that can help decipher the pin-mappings. The main steps involved in the algorithm are depicted in Figure-6. The key observations behind our algorithm are that:

- The number of minterms, or set-bits in the LUT code, remain identical; though they may vary in their position as the pin-mappings get altered.
- The minterms correspond to the specific input combinations for which the LUT is programmed for. An alteration in position of these inputs on the LUT pins results in a similar repositioning of minterms in the LUT codes.

As an instance, consider the code $B1$ in Figure-6 that programs the LUT for a simple Boolean function, $S1.S2'.S3'$. The two pin-mappings have identical number of minterms in the LUT-code. Furthermore, repositioning of input-pins in pin-mapping B results in an alteration of LUT-code but the functionality of the LUT remains consistent with the programmed Boolean function.

We use the above observations in developing a LUT pin mapping detection algorithm. The algorithm initially programs the LUT whose pin mapping is desired with a beacon code such as $B2$. The beacon code has a unique pattern in that its minterms follow Gray-code: in this case $S1.S2'.S3' + S1.S2.S3' + S1.S2.S3$.

Combining the above two observations, we know that any alteration in pin-mapping will be preserved in a similarly altered LUT-code. We read the internally altered code, such as $B2$ in pin-mapping B, using similar fine-grained PDR APIs that we used for LUT writing. The read code is next checked for a minterm that involves only a single logic-high input-pin. This minterm corresponds to input-pin combination $I_2' I_1 I_0'$. Since input-pins have a one-to-one mapping with inputs, we conclude that this minterm must be $S1.S2'.S3'$ that also involves only a single high input. In other words input-pin I_1 must be connected to input $S1$. We next proceed to the minterm involving two logic-high input-pin combination: $I_2 I_1 I_0'$. Following the same argument as above, this minterm should correspond to $S1.S2.S3'$ in the programmed Boolean function. As we have already found out the mapping of pin $I1$ to $S1$, the other high input-pin in the minterm, i.e. I_2, must be mapped to input $S2$. The pin-mapping of the final input S_3 then becomes implicit to the remaining input-pin I_0.

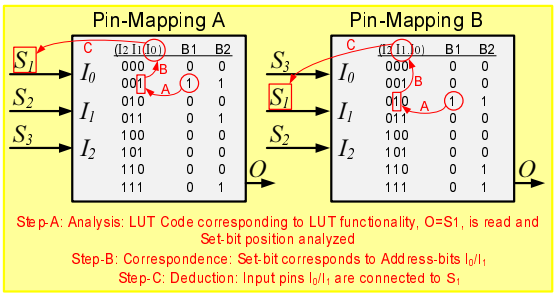

Fig. 6. LUT Pin Mapping Detection Algorithm

C. Met-Signal Synchronization

There could be a number of ways to realize a multi-socket system. Depending on the design, the met-signals, propagation and collection in the system may involve different challenges. As an instance, a naive system realization could be a direct socket connection with a collection unit. Since the collection unit can only process a single met-signal at a time, the asynchronously generated met-signals from multiple sockets would require serialization. The serialization will consequently require logic-expensive arbitration schemes in case multiple sockets need to send their met-signals simultaneously.

A solution to the area expensive arbitration could be to automate the met-signal serialization by chaining the sockets such that the mets hop through the sockets before being received at the destination. The presented DRS is designed for ease in such kind of chaining, or pipelining. The chaining removes arbitration costs. However, as the mets actually are sockets' unique ID codes in the system, long chains tend to increase their sizes resulting in logic overheads.

We address the arbiter size versus socket-ID size issue by having a combination of the above two solutions using multiple short chains as shown in the DataEngine section of Figure-7. However, as the sockets are independent entities with met-signals being asynchronously generated and propagating on a common bus in a chain, they require some form of synchronization to avoid bus conflicts. For example, a socket may overwrite a met-signal generated by the next socket in the chain in the absence of a synchronization scheme. We resolve the issue using our novel *Hole Propagation based Met-signal Synchronization* scheme. We will discuss details of the

scheme when we discuss the threshold-collection unit in the next section.

VI. BURAQ: DRS Based Stateful Programmable Real-time RNTM System

The BURAQ system is comprised of a PC based control and analysis front-end and an FPGA based back-end Data Processing unit. A high level depiction of the system is given in Figure-7. The division is broken on the lines of domains that are more suitable for respective types of processing: general purpose processors for processing control and configurations on a customized hardware data-engine that matches incoming flows in real-time with the configured Boolean mappings. We next discuss the specifics of our system.

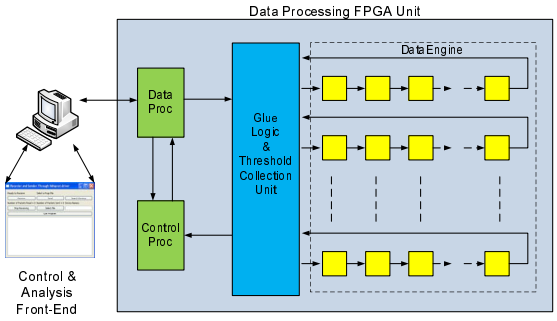

Fig. 7. BURAQ System Architecture

A. Control and Analysis Module

The PC based front-end Control and Analysis module is responsible for interacting with the user and mapping his requirements on the FPGA processing unit. It collects the user-queries, converts them into the bit-vector based DRS mapping and sends them over an Ethernet link to the FPGA processing unit. It also collects the measurement results back from the processing unit that are eventually conveyed back to the user.

B. Data Processing FPGA Unit

The Data Processing FPGA unit is subdivided into a customized data-engine and two soft-core processors. The data-engine itself is composed of number of sockets employing a high degree of parallelism that can be scaled according to the needs and resources of the deployment.

The sockets are arranged in parallel and pipelined fashion. Each socket can be independently configured for concurrent rule-processing. The task level parallelism of a socket is combined with architectural pipelining to maintain scalability of the system. The pipelining only ensures that incoming packet data and met-signals stream through the sockets.

*1) **Data Processor**:* The Data Processor (DP) works at the network layer and is primarily responsible for outside communication over the Ethernet interface. The processor receives two types of packets, data and control, and returns back status packets. It extracts the data and control information out of the packets and forwards them to the data-engine and to the control processor respectively. It also receives status or threshold responses from the data-engine through control processor that it routes back to the front-end.

*2) **Control Processor**:* The Control Processor (CP) acts as an arbiter between the data-engine and front-end Control and Analysis unit. It receives the LUT read and write requests through DP and honors them using PDR of the sockets. The reconfiguration is done by invoking lower level Xilinx APIs and modules that interact directly with FPGA-fabric for reconfiguration. This can be best visualized as a stack of horizontal layers on an FPGA fabric as shown in Figure-8. It is to be noted that the APIs are solely available in the soft-core processors instantiated on the FPGA fabric and work over a dedicated hardware unit, the Internal Configuration Access Port (ICAP).

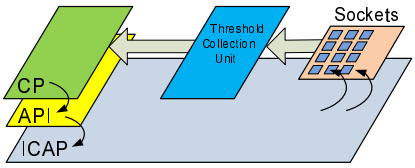

Fig. 8. Socket Reconfiguration and Threshold Collection

*3) **Glue Logic and Threshold Collection Unit**:* A classic co-designed problem is in interfacing fast, cycle accurate and parallel hardware with non-cycle accurate and slow software. The *Glue-Logic* addresses this problem by acting as a hand-shaking middleman to co-ordinate data transfer between the DP and the data-engine.

The issue of threshold-met signal synchronization is also addressed here using the novel Hole-Propagation scheme. The basic principle of the technique is to withhold met signal going over to the next socket in a chain unless there is space, or a hole, available there. These holes are initially created when the last socket in a chain is polled by the threshold-collection unit, that continues polling end of the chains in a round-robin fashion. The presence of a hole lets the preceding socket to forward its met-signal, thereby transferring the hole one socket backwards. Thus, as met-signals move forward within a chain, the holes proceed backward, ensuring all the met-signals get properly received at the CP.

VII. Empirical Evaluation

A prototype of the presented design is developed using Xilinx Virtex-II Pro XC2VP30 FPGA and Xilinx Embedded Design Kit (EDK) 9.1. Xilinx Microblaze soft-core processors were instantiated on the FPGA fabric connected with FIFO-based Fast Simplex Links (FSLs). The prototype was operated at a clock frequency of 100-MHz.

A. Single Socket Evaluation

We compare our DRS with an outstanding example of a static-logic implementation [8], referred here as Static in short. We compare FPGA logic utilization and rule deployment latency. The logic utilization comparison is presented

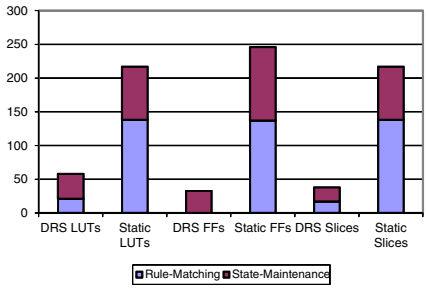

Fig. 9. Area Comparison

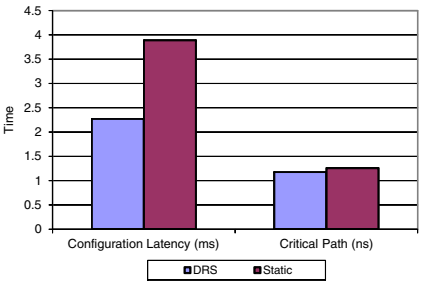

Fig. 10. Reconfiguration Latency Comparison

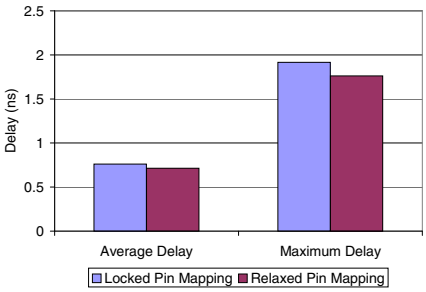

Fig. 11. Socket Delay

in Figure-9, where it is broken down into rule-matching and state-maintenance components.

It can be noticed that Static employs quite a significant number of flip-flops (FFs) (246) and LUTs (223) than DRS (33 and 58 respectively). These two elements form core of device-slices and as such reflect in a higher slice consumption budget of Static (137) compared to a DRS (37), a 3.7x improvement. Static further requires a high number of FFs. This is due to the use of FFs for rule storage as was discussed in Section-III. The issue is resolved in the DRS that maps the rules on LUTs.

The novel synchronization mechanisms utilizing PDR further help DRS in having a much simplified external interface. This is because dedicated inputs are no longer required for DRS reprogramming. As such, the number of IO pins in a DRS have been reduced to 75 from 160 in the Static.

A comparison of reconfiguration latency between the DRS and the Static is next presented in Figure-10. DRS involves reprogramming of 25 LUTs, 23 in rule-matching and 2 in the state-update (Reset and Threshold LUTs). We program all the 25 LUTs in the DRS, though in practice lesser number of LUTs may actually need to be programmed. Configuration in the case of Static is however more involved. The configuration latency of the Static reported in Figure-10 is derived from the smallest 3x3 Static architecture as reported in [8]. The critical path latency is about the same in both the solutions.

B. Xilinx API Evaluation for PDR

The units of partial dynamic reconfiguration in Xilinx devices is a *frame*, involving a number of FPGA resources. Our proposed scheme requires updating specific bits in a frame that correspond to the contents of the target LUT. Detailed composition of frame bits, however, is a Xilinx proprietary information. Xilinx provides ICAP application programming interface (API), which enables constrained manipulation of frame bits. The API was our only mechanism for updating configuration bits of a specific LUT within the frame bits.

Reconfiguration latency of the DRS is due to calls to three API operations (1) reading-in of the frame involving the target LUT, (2) modifying the required LUT bits in the frame, and (3) writing-back the frame to the device. To the best of our knowledge, there exists no mechanism in the API to allow multiple LUT changes within a frame, thereby requiring multiple API calls that adversely affect reconfiguration latency.

The XC2VP30 device has a frame size of 824 bytes. Xilinx requires at least 2 frames (one pad frame) to be fed for a frame to be configured on an 8-bit ICAP port. Assuming ICAP operating at 100-MHz, a frame can be rewritten back in 24.72μs, compared to 90μs that the API rewritting involves.

We suspect that it is quite easy for Xilinx to improve ICAP API to provision for multiple LUT changes within the same frame in a single call. Such simple updates to frame manipulation API, which constitute the chunk of reconfiguration latency for DRS, can substantially improve the latency of DRS reconfiguration, which translates to avoiding packet loss and/or less buffering requirement in the RNTM system.

C. LUT Pin Mapping Algorithm Evaluation

We next evaluate the benefits of LUT pin mapping detection algorithm. We do this by generating a number of designs involving chains of LUTs of different logic depths. The designs are constrained for locking LUT-inputs at particular LUT-pins, and are implemented for Xilinx Virtex-II pro device. The results so obtained for average and maximum delay for all the nets/wires corresponding to constrained and unconstrained designs are presented in Figure-12 and 13. Each point in the plots is an average of five experiments for pin-to-pin delays involving a chain of LUTs of size given on x-axis.

It can be seen that locking input-pins degrades the routing quality, yielding inferior timing results. The average delay is consistently better while having a relaxed pin assignment. The maximum pin-to-pin delay is also better without constraints in the majority of cases, as the tools have more flexibility in optimal routing in an unconstrained environment. The same observations are also noticed for our socket, presented in Figure-11. The maximum delay shows that the LUT-pin mapping detection algorithm hands off a considerable 8% static timing improvement to our socket timing.

D. BURAQ System Evaluation

We finally evaluate BURAQ's total system area for various data-engine sizes. We use SPT tool for quick generation of systems of varying total sockets and chain lengths, i.e sockets per chain. The results are presented in Figure-14. Each point in the Figure represents a system having total number of sockets as given in the legend, broken into chains represented on x-axis. Thus the mid-point of the top-most curve represents

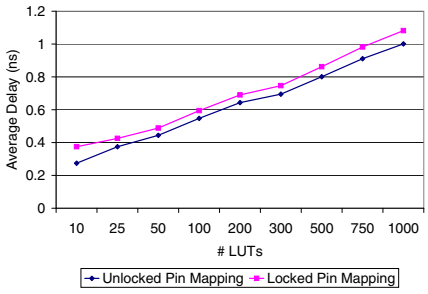

Fig. 12. Average Delay

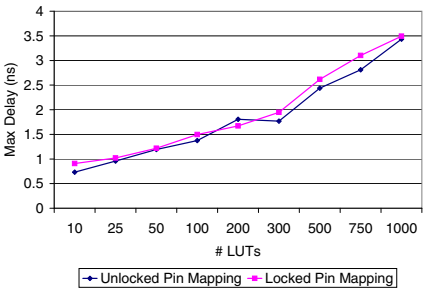

Fig. 13. Maximum Delay

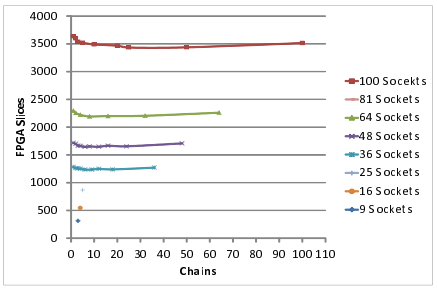

Fig. 14. BURAQ System Area

100 sockets broken into 50 chains or a chain length of 2 sockets/chain.

We note that the biggest BURAQ system can have up to 100 sockets, an improvement of 4x over Static system [8], despite the sockets being slightly less than 4x size improvement over the Static. This is because a socket's simpler design simplifies associated logic, making room for a bigger system. We further note that there is a slight dip in the curves. The dip is due to a completely parallel or a serial data-engine corresponding to the end points, both of which incur logic overheads as was discussed in Section-V-C. The SPT utilizes these results to quickly come up with a best possible data-engine configurations for a user required system.

VIII. Conclusions and Future Directions

We presented a self-reconfigurable platform tailored to NTM applications. The key building block in our architecture, called dynamically reconfigurable socket (DRS), contains a reduction tree composed of carefully placed LUTs, which match packet headers against a rule using FPGA look up circuitry. Plugging a new rule only requires updating the contents of the LUTs, while keeping their placement and routing intact. Xilinx ICAP interface is utilized to dynamically map incoming rules to the architecture, via reconfiguring of LUT programming bits.

We also presented several issues involved in development of a NTM system that contains many DRSs. Specifically, we discussed an algorithm for detection of LUT pin mapping, and a tool for efficient floorplanning of the DRS array. Also, the integration of the DRS array into a complete NTM solution was discussed.

Our evaluations demonstrate 4x improvements over a competitor that utilizes static logic, which lead to a higher level of parallelism under area constraint. To an end user, these savings translate into a higher number of rules that can be answered in parallel using the same resources. Our scheme is also competitive with static logic-based implementations, in terms of reconfiguration latency. Additionally, our study provides helpful insights to reduce the latency via simple updates in the ICAP API.

Our future work include investigation of just in-time and incremental compilation of processing rules to further reduce dynamic reconfiguration latency. On the application front, we plan to interface our NTM system with a data mining and rule generation engine to automatically search for patterns of interest in traffic. Our architecture complemented with such an engine could extract signature-less spatiotemporal traffic patterns, such as new traffic anomalies.

References

[1] "Cisco NetFlow," http://www.cisco.com/warp/public/732/Tech/netflow.

[2] N. Brownlee, C. Mills, and G. Ruth, "Traffic Flow Measurement: Architecture," RFC 2722, 1999, http://www.ietf.org/rfc/rfc2722.txt.

[3] H. Song and J. W. Lockwood, "Efficient packet classification for network intrusion detection using FPGA," in *FPGA '05: International symposium on Field-programmable gate arrays*, 2005.

[4] N. Weaver, V. Paxson, and J. M. Gonzalez, "The shunt: an FPGA-based accelerator for network intrusion prevention," in *FPGA '07: International symposium on Field-programmable gate arrays*, 2007, pp. 199–206.

[5] N. G. Duffield, "Sampling for passive internet measurement: A review," *Statistical Science*, vol. 19, no. 3, 2004.

[6] A. Ramachandran, S. Seetharaman, and N. Feamster, "Fast monitoring for traffic subpopulations," in *IMC '08: Proceedings of the 8th ACM SIGCOMM conference on Internet measurement*, 2008, pp. 257–270.

[7] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: towards programmable network measurement," in *SIGCOMM '07: Proceedings of the 2007 conference on Applications, technologies, architectures, and protocols for computer communications*, 2007, pp. 97–108.

[8] F. Khan, L. Yuan, C.-N. Chuah, and S. Ghiasi, "A programmable architecture for scalable and real-time network traffic measurements," in *ANCS '08: Proceedings of the 4th ACM/IEEE Symposium on Architectures for Networking and Communications Systems*, 2008, pp. 109–118.

[9] M. Majer, "An FPGA-based dynamically reconfigurable platform: From concept to realization," in *FPL '06: Field Programmable Logic and Applications*, Aug. 2006, pp. 1 – 2.

[10] J. Becker, M. Hubner, G. Hettich, R. Constapel, J. Eisenmann, and J. Luka, "Dynamic and partial FPGA exploitation," in *Proceedings of the IEEE*, Feb. 2007, pp. 438–452.

[11] B. Blodget, S. McMillan, and P. Lysaght, "A lightweight approach for embedded reconfiguration of FPGAs," in *Date'03: Design, Automation and Test in Europe Conference*, 2003, p. 10399.

[12] V. Paxson, R. Sommer, and N. Weaver, "An architecture for exploiting multi-core processors to parallelize network intrusion prevention," in *proceedings of IEEE Sarnoff Symposium*, 2007.

[13] D. E. Taylor, "Survey and taxonomy of packet classification techniques," in *ACM Computing Surveys (CSUR)*, vol. 37:3, 2005, pp. 238 – 275.

[14] G. Varghese, *Network Algorithmics*. Morgan Kaufmann, 2005.

[15] S. Hauck and A. Dehon, *Reconfigurable Computing*. Morgan Kaufmann, 2008.

[16] D. Lim and M. Peattie., "Two flows for partial reconfiguration: Module based or difference based," in *Xilinx Application Notes*, vol. 1.2, 2004.

[17] E. L. Horta, J. W. Lockwood, D. E. Taylor, and D. Parlour, "Dynamic hardware plugins in an FPGA with partial run-time reconfiguration," in *DAC '02: Proceedings of the 39th conference on Design automation*, 2002, pp. 343–348.