UC Davis
UC Davis Previously Published Works

Title
A Dynamically Reconfigurable System for Closed-Loop Measurements of Network Traffic

Permalink
https://escholarship.org/uc/item/7hp1d1tf

Journal
IEEE Transactions on Computers, 63(2)

ISSN
0018-9340

Authors
Khan, Faisal
Ghiasi, Soheil
Chuah, Chen-Nee

Publication Date
2014

DOI
10.1109/tc.2012.228

Copyright Information
This work is made available under the terms of a Creative Commons Attribution-ShareAlike License, available at https://creativecommons.org/licenses/by-sa/4.0/

Peer reviewed

eScholarship.org Powered by the California Digital Library
University of California

A Dynamically Reconfigurable System for Closed-Loop Measurements of Network Traffic

Faisal Khan, Soheil Ghiasi, and Chen-Nee Chuah

Abstract—Streaming network traffic measurement and analysis is critical for detecting and preventing any real-time anomalies in the network. The high speeds and complexity of today's networks, coupled with ever evolving threats, necessitate closing of the loop between measurements and their analysis in real time. The ensuing system demands high levels of programmability and processing where streaming measurements adapt to the changing network behavior in a goal-oriented manner. In this work, we exploit the features and requirements of the problem and develop an application-specific FPGA-based closed-loop measurement (CLM) system. We make novel use of fine-grained partial dynamic reconfiguration (PDR) as underlying reprogramming paradigm, performing low-latency just-in-time compiled logic changes in FPGA fabric corresponding to the dynamic measurement requirements. Our innovative dynamically reconfigurable socket offers 3× logic savings over conventional static solutions, while offering much reduced reconfiguration latencies over conventional PDR mechanisms. We integrate multiple sockets in a highly parallel CLM framework and demonstrate its effectiveness in identifying heavy flows in streaming network traffic. The results using an FPGA prototype offer 100 percent detection accuracy while sustaining increasing link speeds.

Index Terms—Reconfigurable hardware, network monitoring, parallel circuits

1 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, rerouting traffic, or detection of anomalies, require extraction of real-time statistics from network traffic. A high-quality network measurement tool is crucial for extracting such patterns of interest and making informed decisions to ensure proper network operation [1], [2], [3].

Today's high-speed networks see huge amounts of streaming traffic, posing enormous computational and storage requirements for accurate traffic measurements. Traditionally, the measurements are performed by maintaining limited information of the streaming data. This is done by programming conservative sampling factors over to the routers that maintain some very limited local storage. The collected sample is next periodically expired to high-end servers where it is postprocessed in answering some higher level user-queries comprising of spatiotemporal patterns of interest, such as amount of traffic passing through a subnet or detecting a network anomaly. A high-level depiction of the traditional paradigm is shown in Fig. 1a.

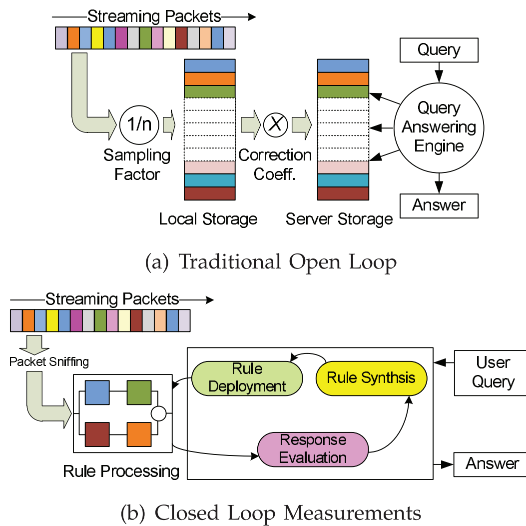
- F. Khan is with the Department of Electrical and Computer Engineering, University of California, Davis, 95616. E-mail: fnkhan@ucdavis.edu.
- S. Ghiasi and C.-N. Chuah are with the Department of Electrical and Computer Engineering, University of California, Davis, Kemper Hall, One Shields Avenue, Davis, CA 95616.
E-mail: ghiasi@ucdavis.edu, chuah@ece.ucdavis.edu.

Manuscript received 6 Apr. 2012; revised 11 Aug. 2012; accepted 30 Aug. 2012; published online 11 Sept. 2012.
Recommended for acceptance by M. Guo.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2012-04-0254.
Digital Object Identifier no. 10.1109/TC.2012.228.

Though conceptually simple to realize, the traditional approach not only incurs significant measurement inaccuracies due to sampling, but the measurements are also *orthogonal*, or blind, to the requirements [10]. Due to the separation between measurements and user requirements, the traditional paradigm is referred to as *open-loop* measurements. Furthermore, the offloading of the measurements for processing over to higher software layers at the servers (the slow-path) is also latency intensive and infeasible for detecting and preventing anomalies in real time.

We previously addressed the problem with open-loop paradigm using smart, goal-oriented *closed-loop measurement* solution, as shown in Fig. 1b [4]. Central to our proposed scheme is a tight integration between the measurement requirements and the actual measurements. This is achieved by bringing in the requirements on a fast-path that directly observes the streaming traffic, such as routers. To cope with the limited computation capabilities of the fast-path, the CLM works by breaking a higher level user-query into multiple *rules* (a *rule-set*) of finer granularities. The rule-set is processed in the fast-path and iteratively refined over time until the user-query gets answered. The contention is that the interesting traffic patterns could be detected or learned on-the-fly, via iterative rule-based traffic measurements, online analysis of collected information, and closed-loop evolution of subsequent rules for further and finer traffic inspection. Each round in the iterative process guides the subsequent measurements toward the goal, thereby reducing redundant measurements and leading to answering the user-query over time.

The increasing speeds and scale, the sizes, and complexities of today's networks, however, bring about a number of challenges in closing the loop between measurements and the user requirements in a streaming setup. At the heart of the streaming CLM framework is the measurement platform that is faced with two core challenges 1) link-speed



(a) Traditional Open Loop

(b) Closed Loop Measurements

Fig. 1. Network measurement paradigms.

processing of incoming packets with the rule-set, and 2), a high degree of programmability to quickly and dynamically adapt to changing rule-sets. The latter implies an additional critical dimension to our problem of CLM over plain rule-matching problems, such as traffic classification, where the rules generally remain static or have very infrequent updates over the lifetime of operation.

We previously addressed the CLM challenges by developing a customized rule-processing solution mapped on to a field programmable gate array (FPGA) [4]. In doing so, the solution targeted the computation opportunities that conventional routers present with their incorporation of FPGAs. However, to speed up the programmability of the solution with changing rule-sets, the scheme made heavy use of FPGA registers that can quickly be rewritten. The registers are usually a prime commodity on the FPGAs and provide for far less storage capability than other device resources, such as lookup tables (LUTs). As such, the high register usage quickly depleted the critical resource, and leading to an overall underutilization of the device. In practical terms, the solution implies trading off the ability to process higher number of rules for an ease in rule programmability.

To increase rule-processing parallelism, one could map the rules on to the FPGA LUTs that offer much higher density than the device registers. However, the reprogrammability of LUTs is typically done through *hardware compilation (or compilation)* of the updated design using proprietary tools; a process involving large amounts of memory and latencies, and clearly being impractical for streaming CLM setup. One way to reduce the compilation cost is to selectively compile and dynamically replace the device logic that is exclusively associated with the modified design, while the rest of the device remains operational. The approach, referred to as partial dynamic reconfiguration (PDR), has been a major focus in the research community [5], and is traditionally done utilizing *module-based* PDR flow [6]. However, the compilation latencies with the approach though much reduced, still run in seconds to even minutes and as such are prohibitively expensive for CLM.

A relatively less known PDR flexibility that Xilinx FPGAs provision is in dynamically performing minute logic changes for a limited set of device resources [7]. Such a fine-grained PDR scheme constrains device routing to remain static while certain programmable points, such as LUTs, could be dynamically reconfigured. We earlier proposed a novel use of such fine-grained PDR in developing a statically routed and dynamically reprogrammable, basic rule-processing unit, dubbed as *Dynamically Reconfigurable Socket (or Socket)* [8]. Our initial results were promising, offering significant increase in rule-processing capacity.

In this paper, we utilize the Sockets to offer a highly parallel, programmable, and scalable realization of the CLM system. The proposed hardware software codesigned system performs goal-based, resource-aware, and Just-in-Time (JiT) compiled minute logic changes on a Virtex-II FPGA using PC-based control unit connected over Ethernet. The salient contributions of this paper are summarized as follows:

- We provide tools and methodologies and demonstrate their effectiveness in integrating multiple Sockets in a system, utilizing fine-grained PDR as underlying reconfiguration technology. To the best of our knowledge, our work presents the first utilization of fine-grained PDR in a practical system.
- We evaluate our PDR-based CLM measurement system with traditional statically mapped logic-based solutions. Our evaluations demonstrate 3.3× logic savings by using fine-grained PDR over conventional solutions, while having much reduced reconfiguration latencies over traditional module-based PDR schemes.
- We demonstrate a practical application of the CLM system in isolating heavy volume flows in passing traffic, sustaining link speeds with 100 percent detection accuracy.
- We provide a rigorous analysis of the latencies associated with fine-grained PDR mechanism. Our study, therefore, compliments the earlier studies that discussed reconfiguration latencies for module-based PDR flows [9].

We stress that many of our presented techniques and algorithms are quite generic and transcend the needs of the current application. The CLM problem falls in a class of problems that are faced with competing requirements of high performance and dynamic programmability. As such, the presented hardware and software solutions can be adapted to support many similar applications, where programmed patterns are not known a priori, such as cryptography (too many keys, plaintexts, or ciphertexts), neural networks (too many topologies and/or coefficients), pattern matching (when patterns are known at runtime only), and generic code accelerators, among others.

The rest of the paper is organized as follows: We discuss related concepts and works on traffic measurements and PDR in Section 2. This is followed by techniques to accelerate rule processing in Section 3. We next discuss our rule-processing unit based on fine-grained PDR, the Socket, in Section 4. We then discuss the system level issues and our solutions while integrating multiple Sockets in

TABLE 1
Rule Composition and Boolean Mapping

Three Rules
$R_1 =< 192./4, *, *, *, * >$
$R_2 =< 200./6, *, *, *, * >$
$R_3 =< *, 212./6, *, *, * >$
Boolean Rule Mapping
$R_1 = s_1.s_2.s'_3.s'_4$
$R_2 = s_1.s_2.s'_3.s'_4.s_5.s'_6$
$R_3 = d_1.d_2.d'_3.d_4.d'_5.d_6$
Query composition using Rules
$Q_1 = (R_1)$
$Q_2 = (R_2 \cap R_3)$

a CLM system in Section 5 followed by a discussion of our CLM system in Section 6. An evaluation of the proposed CLM framework and fine-grained PDR is presented in Section 7. We conclude the paper in Section 8.

2 BACKGROUND

2.1 Network Traffic Measurements

Network traffic measurement fundamentally involves quantification of traffic that satisfies some criteria. The traffic is generally quantified in terms of *flows*, where a flow refers to a set of packets that have the same n-tuple value in their header fields. Typical definitions of the flow include 6-tuple: $\{sip, dip, prt, tos, spt, dpt\}$, where sip and dip are the source and destination IP addresses, prt is the protocol field, tos is type of service, and spt and dpt are the source and destination ports, respectively. We define a *rule* to be an aggregation of flows. For instance, the classless interdomain routing (CIDR) prefix is a particular type of a rule that aggregates over all the flows that have matching significant bits corresponding to the size of the prefix. Table 1 defines three rules in the 6-tuple definition.

Traditionally, the measurements have been open loop based by maintaining unique *per-flow*-based statistics. However, such approach is not scalable due to huge number of flows. As an alternative, packet sampling is typically deployed to cope with increasing line rate, but it is known to introduce bias and affects the effectiveness of various anomaly detection schemes [10]. Moreover, measurements performed under open-loop solutions are oblivious to application requirements, leading to potential redundancy or inaccuracy.

Recently, there has been an interest in developing streaming/online closed-loop schemes to address the challenges in network measurement and analysis [4], [11]. The key observations of the schemes are top-down, goal-oriented measurements as desired by the user-query rather than the blind, bottom-up offline measurements as is done conventionally.

The smart measurements are based on subdividing the user-query in multiple rules, or a rule-set. A rule can, thus, be also viewed as an intermediate question in pursuit of the user-query that if answered can help lead the search in a more intelligent manner. For instance, searching for an anomalous heavy flow in an n-tuple space could be broken down into multiple rules of smaller dimensions (tuples). Table 1 shows a simple query composition using rules.

2.2 Rule-Processing

We hereby define *rule-processing* as a two-step process involving checking, or matching, 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, thus, represents the number of packets matching a given rule.

The count of matched-rules represents *state* of the network in the CLM problem. This state leads to future refinement of the rules, before an eventual network level decision can be made. As such, the CLM entails a temporal dimension to other pattern-matching problems such as packet classification and intrusion detection [12] that instantly base their decisions on the evaluation of their rule-sets, without keeping a state or history of the network. Furthermore, the need to quickly update the rules with refined rule-sets brings about a critical challenge of fast rule reprogrammability to the CLM that is generally missing in other rule-matching problems.

Rule-matching has been the focus of numerous studies [13], [14]. The rule-matching algorithms can basically be classified into two categories: decomposition based and decision-tree based. The decomposition-based algorithms (e.g., parallel bit vector [15]) work independently on individual tuples before partial results could be merged. Due to their parallel nature, the decomposition algorithms are suitable for hardware implementations. The downside of decomposition-based approach is their difficulty in scalability with respect to number of rules. In contrast, decision-tree based algorithms (e.g., HyperCuts [16]) map the rule-set into a tree structure. The algorithms work by iteratively reducing the n-dimensional search space into smaller subspaces, until the reduced subspace matches a unique rule or the reduced subspace can easily be searched. The decision tree algorithm is superior in terms of accommodating higher rule-set sizes. However, the tree structure is difficult to maintain in hardware requiring specialized circuits [17]. The difficulty is compounded in the face of rule updates, where a rule change may trigger multiple changes in the decision tree, potentially leading to replacement of the entire tree in the worst case.

2.3 Partial Dynamic Reconfiguration

FPGAs provide an interesting blend of programmability and performance. Their low-cost, off-the-shelf availability and particularly their ability to meet the link speeds have led to their deployment in a number of networking applications [12], [18], [19], [20]. At their core, the FPGAs utilize a *configuration memory* to store the configuration of its various programming points, such as lookup tables, multiplexers, routing logic. The (re)programming, commonly referred to as *(re)configuration*, of the FPGAs involves updating the configuration memory with an updated *bitstream*. The mapping of the bitstream to programming points inside the FPGA is, however, proprietary information. The recommended way to generate the bitstreams is to perform compilation of the updated design using proprietary CAD tools that deal with synthesis and mapping of the design onto the FPGA resources; a process involving large amounts of memory and latencies. In the case of CLM, where rules are generated on-the-fly during operation, the approach implies prohibitive latencies for closing the loop in a streaming manner.

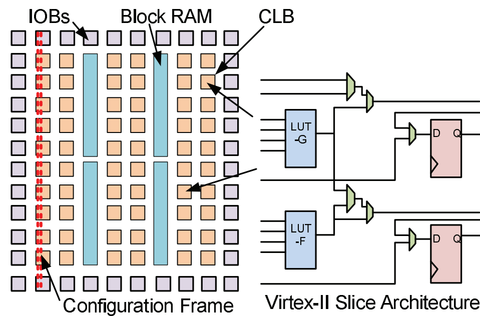


Fig. 2. High-level Virtex-II FPGA architecture.

A high-level depiction of Xilinx's [21] Vitex-II FPGA architecture is shown in Fig. 2. The device consists of an array of basic computational units, the configurable logic blocks (CLBs), that communicate with each other using a programmable interconnect architecture (not shown). The CLBs are broken down in Slices that embed certain number of LUTs and storage elements for mapping user logic. The compositions and complexity of CLBs and slices vary across device families. A high-level layout of Xilinx Virtex-II Slice comprising of two 4-input LUTs and two storage elements, the flip-flops (FFs) is also shown in the figure.

The basic unit of reconfiguration in Xilinx devices is a *configuration frame*, spanning over multiple FPGA resources in a vertical column, as shown in Fig. 2. The unit implies that to (re)configure any programming point, one needs to rewrite the entire frame containing that programming point in the configuration memory. Furthermore, Xilinx requires an extra pad-frame at the end of configuration frames for flushing purposes.

PDR of FPGAs tries to reduce the compilation costs by updating portions of the bitstream that correspond to the modified (dynamic) design, while the remaining logic (static) remains active. Xilinx has a matured PDR methodology, whereas the other FPGA giant, Altera, has only recently introduced PDR in their Stratix-V devices. The recommended design flow for Xilinx PDR, that is almost universally followed in the research community, is to exclusively perform compilation for the dynamic portions (referred to as *modules*) of the design using Xilinx proprietary tools [6]. However, the compilation latencies with such a conventional module-based PDR still run in minutes. To ensure speedier reconfigurations, the module-based flow is typically done by having the dynamic components compiled at design-time (statically) and readily available for quick reconfiguration during operation.

The module-based conventional PDR schemes have seen their fair share in networking community. Some interesting applications include a programmable network switch [22], accelerators for pattern matching [23], as well as a reprogrammable IP forwarding engine [24]. Such applications make use of relative latency insensitiveness of the application or a priori availability of programmability requirements to offset the high latency costs of the compilation. However, in the streaming CLM setup, such flexibility is not available because the rules defining the reprogrammability needs are generated on-the-fly based on the network conditions.

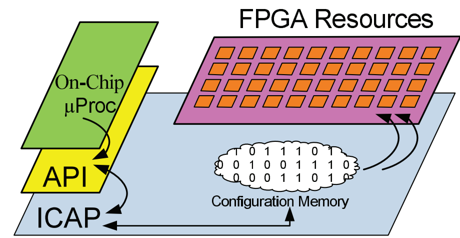


Fig. 3. Fine-grained PDR.

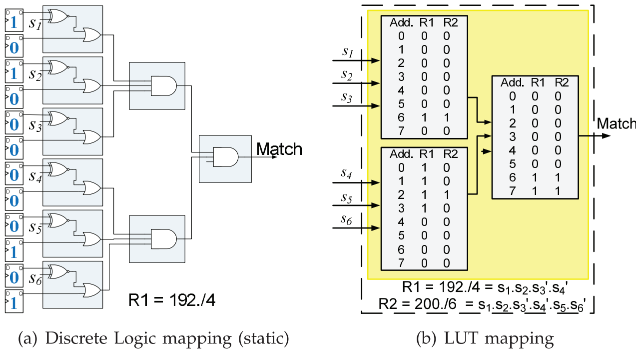
To avoid compilation, much attention has been given in deciphering and directly manipulating the proprietary bitstream. JBits [25], a long discontinued tool, aimed to provide such a flexibility to be able to design and update circuits in a true software development environment. More recently, there have been a number of efforts to reverse engineer the bitstream structure [26]. In contrast to above, Xilinx also provisions for PDR at a very fine granularity targeting limited set of programming points, such as LUTs [7]. The fine-grained PDR is achieved by self-reconfiguring (SR) the device through the embedded internal configurable access port (ICAP) using proprietary APIs running at FPGA-based generic processors. This can be best visualized as a stack of horizontal layers on the FPGA fabric as shown in Fig. 3. To reconfigure a specific resource such as a LUT, the API performs three operations: 1) reading-in of the frame involving the target LUT, 2) modifying the required LUT bits in the frame, and 3) writing-back of the frame to the device followed by the pad-frame.

The idea behind the fine-grained PDR has been to offer a faster on-chip alternate to JBits running on slow off-chip Java virtual machine. In addition, the scheme stripped off the JBits cumbersome manipulation of the device layout, leaving it to be modifiable only through conventional CAD tools assisted module-based PDR flows. However, the necessity to hide proprietary composition of the bitstream imposed several design challenges that limited the fine-grained PDR's feasibility in practical applications. In this paper, we discuss and address these challenges and present a novel hardware-software codesigned CLM solution that employs fine-grained PDR as underlying reconfiguration paradigm.

3 Accelerating Rule Processing

The CLM entails breaking down a user-query into a rule-set comprising multiple rules that are answered over time. For accurate streaming measurements, every incoming packet needs to be processed against the rule-set until a match is found. Unlike other pattern matching problems, such as packet classification that only involve rule-matching, the CLM additionally requires maintaining running statistics of successful matches along with frequent updates to the rule-set. The frequent rule update nature of the CLM implies that a decision-tree-based structure is quite infeasible for maintaining the rule-set.

Proposition 1. *Let τ denotes the total worst-case rule-processing time on a sequential processor involving matching incoming packets against a rule-set of size R rules. Each matching operation involves time to fetch a rule (r) and processing it*



(a) Discrete Logic mapping (static)

(b) LUT mapping

Fig. 4. Rule mapping on hardware.

against the incoming packet (p). Finally, a memory location is updated on a successful match, further involving latencies of memory fetch (r), count update (u), and memory write (w) operations. If λ denote the packet arrival rate, then for streaming closed loop measurements, $R(r + p) + (r + u + w) \leq \frac{1}{\lambda}$.

The above translate to a latency budget of 32 ns to process the rule-set for the case of minimum sized IP packets of 40-bytes streaming over 10-Gbps wire-speed. On a 2-GHz CPU, this implies 64 clock cycles to match incoming packet with the entire rule-set in the worst case. The small latency budget along with high processing overheads makes rule-processing very difficult to be performed in real time using conventional sequential processors.

The multicore architectures conceptually offer reduced processing times by parallelizing rule-processing. However in practice, the multicores are faced with bottlenecks due to their shared memory architecture. As the CLMs require frequent memory updates, the bottleneck implies serialization of memory requests, or loss of enhanced processing abilities of the multicores.

3.1 Rule Evaluation Using Discrete Logic

One solution to speed up rule-processing is by employing discrete logic gates. Such a solution for Rule-R_1 is shown in Fig. 4a. The solution uses a combination of discrete-logic gates in evaluating a CIDR prefix-type rule, where the information corresponding to the rule and the significant bits are stored in registers. The logic yields a *Match* answer on a successful match which can be subsequently logged by incrementing a rule-counter, completing the two stages in rule-processing. The solution can also be mapped on an FPGA that employs lookup tables and registers as main computational and storage elements. One such FPGA mapping using nine three-input LUTs and 12 single-bit registers is also highlighted in the figure. Such a mapping remains intact during operation, while the rules are updated by overwriting the register values. Due to the static mapping during the course of operation, we call such a solution as *static*.

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. FPGAs usually deploy almost similar number of LUTs and registers on the device. The high usage of registers creates a disparity in FPGA resource usage, quickly depleting the device registers, and thereby reducing the overall device utilization. Indeed, such was the case with the discrete logic FPGA mapped solution in [4], where the high usage of registers became the bottleneck for a highly parallelized solution.

3.2 Rule Mapping on LUTs

Lookup tables 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. Fig. 4b illustrates the idea using an example of three 3-input LUTs that collectively implement the Rule-R_1 of Table 1 using column-$P1$. In this scheme, the LUTs are programmed with entries that yield a *Match* answer, if the incoming packet header bits match with the programmed rule. As with the static, this “Match” answer can be subsequently logged by incrementing a rule-counter, completing the two stages in rule-processing.

The fusion of rules within LUTs clearly offer sizable area savings over static-logic-based implementations. These savings are due to an application specific fusion where rules, their wild-card patterns, as well as the rule-matching circuitry is all fused together in the LUTs, thereby saving dedicated logic and resources for rule-matching and storage. However, a practical realization of rule-processing using LUTs needs not only be generic enough to admit practical rules, but also be quick enough for updating the rules in a streaming setup. Unfortunately, rule updates become quite challenging while employing LUTs, requiring reconfiguration of FPGA fabric. We address these challenges in the design of the rule-processing unit in the next section.

4 Dynamically Reconfigurable Rule-Socket

The LUT-mapping of the rules employ a network of rule-matching LUTs. In designing the rule-matching network of LUTs, one has to strike a balance between area efficiency (i.e., the number of required LUTs) and the generality of admissible rules. Overly restricted networks cannot admit all possible rules, while disregarding practical rule features (i.e., treatment of a rule as an arbitrary Boolean function on header bits) would result in enormous logic waste. For example, the two-level LUT network of Fig. 4b is designed to admit $s_1.s_2 + s_1'.s_2'$, but cannot implement $s_1.s_6 + s_1'.s_6'$ Boolean checks. We note that the CLM rules also exhibit specific structure as they are composed from individual CIDR prefixes. This can be seen in Table 1 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. Furthermore, we contend that a conjunction of several prefixes in a rule can always be decomposed in multiple single prefixes. Such decomposition can be processed by simpler single-prefix-based rule-processing units.

The potential downside of the LUT design, however, is that it makes dynamic rule updates considerably more complicated than static. By fusing the rules into matching logic as done in LUT-network implies that the FPGA fabric

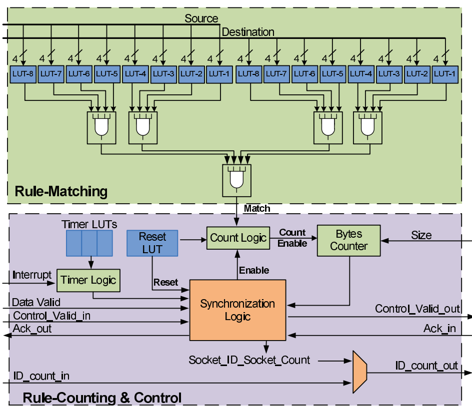


Fig. 5. Dynamically reconfigurable socket.

implementing the logic function has to be reconfigured at runtime to admit a new rule. To address the problem, we make novel use of fine-grained PDR for dynamically reconfiguring the LUTs. 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 Fig. 4b that is initially programmed with rule-R_1 is to be reprogrammed with rule-R_2, it would only require updating the two first level LUTs with contents given in the third column, while keeping the placement and routing consistent.

We now discuss details of our rule-processing unit, dubbed as dynamically reconfigurable rule socket (Dynamic or just Socket). A high-level design of the Socket is shown in Fig. 5 targeting 64 bits of rule-matching. The Socket is composed of two high-level components corresponding to rule-processing requirements: a generic *rule-matching* module combined with *rule-counting and control* logic. Both components have static layouts throughout the lifetime of the system.

The rule-checking module combines a first-level layer of reprogrammable LUTs in a customized reduction-tree to check for patterns in adjacent header bits. While the tree is effective for practical rules based on CIDR prefixes, 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 16 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, and expand the reduction tree.

New rules are dynamically updated or plugged into the Socket during runtime. Plugging of a new rule is achieved using fine-grained PDR of the rule-matching LUTs. The result of rule-matching is forwarded to the rule-counting and control logic that maintains a streaming count representing the aggregate size of the flows that have matched the programmed rule. The aggregation continues until a programmed duration has expired. This duration is programmed into three reprogrammable LUTs, and is tracked by counting external interrupts arriving at the Socket. Upon the expiration of the programmed number of interrupt epochs, the Socket sends out the collected statistics, and is ready for being reprogrammed for a new measurement phase.

The Socket can be optionally reprogrammed with a new rule and collection duration, before being reseted. We exploit flexibility of fine-grained PDR in not only rule reconfiguration on the LUTS, but also for reprogramming duration as well as resetting of the Socket, implying 20 reprogrammable LUTs per Socket (shown in blue in Fig. 5). These innovations help in not only simplifying the logic-footprint within the Socket, but also reduce the logic-overhead of its system-wide integration. We next detail the specifics of these innovations at a system level, but stress that their effectiveness transcends the needs of our application.

5 System Design Challenges

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 highly parallel and asynchronous design with a centralized control raises possibilities of significant communication overheads. These overheads in terms of FPGA-fabric translate into increased logic area utilization and clogging of limited routing resources on the device, yielding loss in available parallelism. For instance, provisioning independent resets or collecting asynchronously generated results from the Sockets may naively be done using dedicated channels between the centralized control and individual Sockets, incurring significant logic and routing overheads. In this section, we discuss our strategies in dealing with these issues.

The fine-grained PDR also presents several interesting challenges. As stated earlier, dynamic reconfiguration requires maintaining a consistent interface between reconfigurable (dynamic) and nonreconfigurable (static) regions. Moreover, for fine-grained PDR, an exact location of the reprogrammed resource on the FPGA is necessary. In our case, where the reconfigurable resources are the LUTs, we further need to be aware of the mapping of the incoming inputs to LUT input-pins as they can get internally altered by the CAD tools during compilation. One would assume that such 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 (SPT) and LUT pin-mapping algorithm that will also be the subject of this section.

5.1 Asynchronous Reset

An incoming reset intimate the Sockets to start-off with a new measurement phase. As Sockets operate asynchronously and independently to one another, a common reset line for every Socket in a multiple-Socket system is clearly infeasible. To provision a unique reset for individual Sockets, one can either 1) have a dedicated reset line between each Socket and the centralized control, or 2) afford additional target Socket

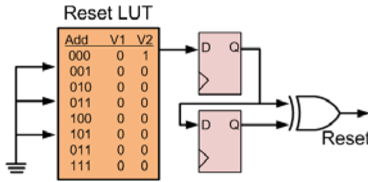


Fig. 6. Reset mechanism.

information using extra identification bits along with a common reset-signal. Whereas the former solution requires multiple reset lines, the latter involves additional identification bits that are to be decoded at the target Socket. Thus, both solutions incur overheads in terms of FPGA interconnect and logic.

To address the problem, we propose an innovative remote-synchronization mechanism that completely does away with explicit signals. The proposed scheme, used in the context of reset here, works over a dedicated LUT as shown in Fig. 6. The idea is to produce a logic-inversion at the LUT output that is decoded to produce a signal. This is achieved by reconfiguring the LUT through fine-grained PDR using codes that cause logic inversion at the LUT output. Two such LUT configurations, $V1$ and $V2$, are shown in Fig. 6. Since the remote-synchronization is done asynchronous to the clock, the LUT output is double-flopped to remove any metastability before being used as reset signal.

The innovative PDR-based synchronization and reprogramming protocols employed in our work not only simplify Socket external interface by reducing the number of IO-pins, but also reduce its logic footprint. We will further discuss the savings when we compare the Socket with the conventional logic-based static solution in Section 7.

5.2 LUT Pin Mapping Inference Algorithm

The functionality of the LUT is defined by a bit-vector, the LUT code, whose individual bits describe the LUT's output. The bit-vector is indexed by the Boolean combination at the LUT's input pins. To assist in better routing, the Xilinx CAD tools that we employ in our work can internally alter how the external-inputs map to the LUT's input pins, thereby altering the bit-arrangement in the LUT code. One such altered pin-mapping, whose alteration information is also not readily available to the end user, is shown in Fig. 7. The knowledge of LUT input pin-mappings is, therefore, critical for fine-grained PDR as it defines how the LUT codes need to be assembled.

A quick way to avoid the tools from altering the pin-mappings could be to lock them down at known positions. However, such constrained mapping reduces the routing flexibility and results in poor routing yield and static-timings [8]. Our solution to the problem is by employing a novel algorithm using which the LUT pin-mapping can be inferred after the Sockets have been configured 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 key observations behind our algorithm are:

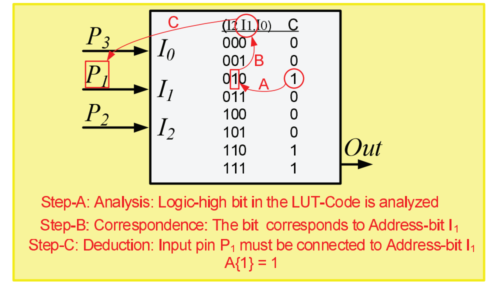


Fig. 7. LUT Pin-mapping inference.

- That the number of minterms, or logic-high bits in the LUT code, remain identical; though they may vary in their position as the pin-mappings get altered.
- That 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.

We use the above observations in developing a LUT pin-mapping inference algorithm tabulated in Algorithm 1. For the algorithm to operate, the LUTs whose pin-mappings are desired are initially compiled with a specific beacon-code at runtime. The code has a unique pattern in that its minterms follow Gray-code such as $S1.S2'.S3' + S1.S2.S3' + S1.S2.S3$. During compilation, the code may get internally rearranged by the CAD tools, similar to code-C in Fig. 7.

ALGORITHM 1: Pin Inference Algorithm

Input : $P\{\}$:Set of LUT-Input Pins
Input : $C\{\}$:The Read LUT Bit-Code
Output: $A\{\}$:Set of Pin Assignments

```
1 for i ≤ |P| do
2     for j ≤ |C| do
3         logic_highs ← Parse(C{j})
4         if logic_highs = i then
5             A{i} ← j
6             for k ≤ i do
7                 A{i} ← A{i} − A{k}
8             break
```

The main steps involved in the algorithm are depicted in Fig. 7. The algorithm works by reading-in the internally altered beacon-code using similar fine-grained PDR APIs that we used for LUT writing. Since any alteration in the pin-mapping is preserved in a similarly altered LUT code, the algorithm proceeds to check the code for the programmed Gray-code by first searching for a minterm that involves only a single logic-high input-pin. This minterm corresponds to input-pin combination $I_2' I_1 I_0'$. As input-pins have a one-to-one mapping with external inputs, the algorithm deduces that this minterm must be $S1.S2'.S3'$ involving a single high input. In other words, we can deduce that input-pin I_1 is connected to input $S1$. The algorithm next proceeds 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 the mapping of pin $I1$ to $S1$ has already been evaluated, the algorithm deduces that the other high input-pin in the

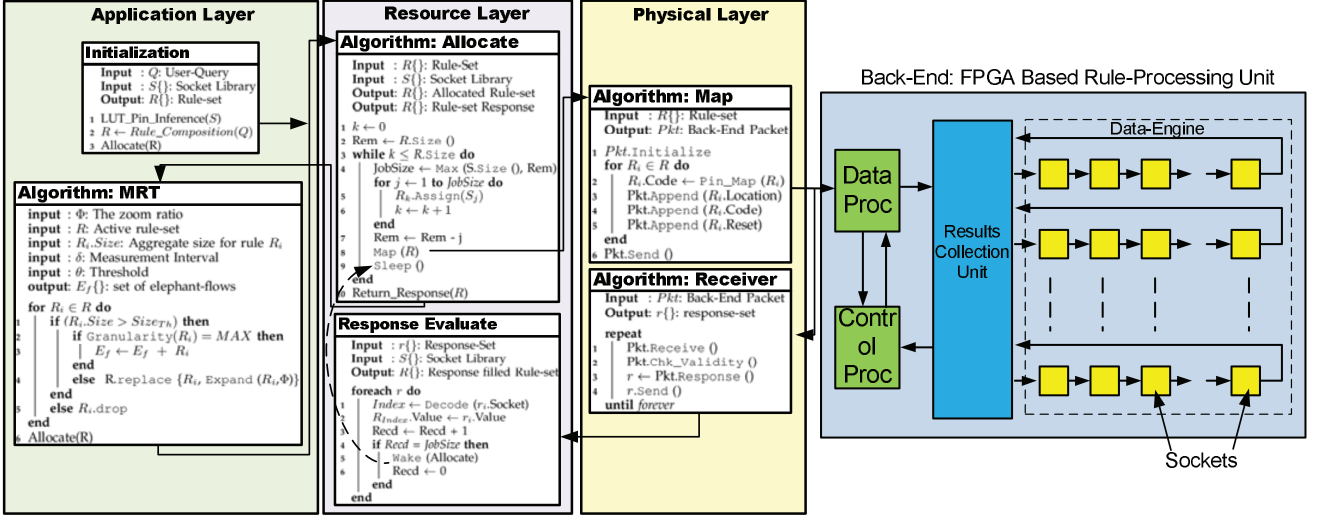


Fig. 8. Closed-loop measurement system architecture.

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.

5.3 Socket Results Synchronization

There could be a number of ways to realize a multiple-Socket system. Depending on the design, the propagation and collection of Sockets' results in the system may involve different challenges. As an instance, a naive system realization could be to directly collect the results using dedicated channels with a collection unit. Such a scheme may not only be expensive in routing resources, but also it may further require logic-expensive serialization mechanisms to be able to collect multiple asynchronously generated results in a multi-Socket system.

A solution to the above issues could be to automate the results serialization by chaining the Sockets such that their results hop through multiple Sockets sharing the same bus/channel before their results are received at the target. However, such a scheme needs to address two challenges for its feasibility: 1) the results need to be identified with the issuing Socket for them to be properly associated with a rule at higher control layers, and 2) bus-conflicts, due to multiple Sockets in a chain simultaneously producing results, must be resolved.

A naive solution to the first problem could be to tie every system Socket with a unique tag in the form of identification-bits. In a system having a large number of Sockets, the solution raises possibility of a high number of the identification bits, resulting in excessive logic and routing overheads. We address the issue by breaking the multiple-Socket system into a number of shorter chains, as shown in the Data-Engine section of Fig. 8. We then uniquely address each Socket in the Data-Engine using the combination of the chain number and identification tag within that chain, thereby saving excessive logic and routing resources.

The issue of bus-conflicts is addressed using our novel *Hole-Propagation-based Synchronization* scheme. The basic principle of the technique is to withhold passing on the results to the next Socket in a chain unless it has the capacity to store, or a *hole*, available there. The presence of the hole lets the preceding Socket in a chain to forward its results, thereby propagating the hole one Socket backwards. These holes are created when the result from the last Socket in a chain is collected by the results-collection unit. Thus, as results move forward within a chain, the holes proceed backward, ensuring correct reception of all the results.

5.4 Socket Placement Tool

We address the issue of LUT identification for reconfiguration by locking the LUTs that are dynamically reconfigured to resources with known locations. A SPT was developed that automates generation of a multi-Socket system and the locking of reprogrammable components within the 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 reprogrammable LUTs within a Socket are constrained. The rest of the Socket is left for unconstrained layout by the CAD tools. Furthermore, the SPT also makes use of the frame-based reconfiguration paradigm to come up with placement constraints that minimize the total number of reconfiguration frames for the system. This is done in the hope of reducing the total number of API calls for system reconfiguration. As frames correspond to resources in vertical columns on the device, the SPT tries to pack maximum number of reprogrammable LUTs in vertical resources. However, in doing so, the SPT makes sure that enough FPGA resources are left nonutilized near reprogrammable LUTs. This is done so that a Socket's components do not get significantly displaced on the FPGA die or else the system will yield poor static-timings.

6 Real-Time CLM System

We now present our application specific CLM system. The proposed solution partitions the closed-loop rule-processing and rule programmability in a hardware-software codesigned scheme as shown in Fig. 8. The division is broken on the domain lines that are more suitable for respective types of processing: with rule-programmability and analysis implemented using software on general purpose processors (front-end) while a customized FPGA-based hardware unit (back-end) being responsible for link speed rule-processing.

6.1 Front-End Software Architecture

The front-end comprises of a three-layer software architecture, interacting through a number of algorithms as shown in Fig. 8. At the top is the application layer where the user programs a high-level formulation of the measurement requirements. The layer incorporates a dynamic rule synthesizer that translates the user-query into intermediate rules. The rules are passed on to resource layer that manages the hardware resources and allocates (or binds) the rules with the back-end rule-processing resources, the Sockets. Finally, the binded rules are mapped into Boolean bit vectors at the physical layer for programming specific LUTs corresponding to the Sockets before passing them on to the back-end.

The physical layer also receives the responses from the back-end where it validates their reception before reassociating them back with the active rules at the resource layer. The rules' responses are next evaluated at the application layer where user policies are enforced and search space exploration algorithms, such as MRT (discussed later), are deployed.

6.2 Back-End FPGA Design

The rule-processing FPGA unit is subdivided into a customized *Data-Engine* and two embedded 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. Each Socket can be independently configured for concurrent rule-processing. The task level parallelism of the Sockets is combined with architectural pipelining that streams back the results using synchronization mechanisms discussed in the previous section. It is assisted by the *Results Collection Unit* that acts as a glue logic between hardware and embedded processors.

The embedded processors on the FPGA fabric assist in communication with the front-end (Data Proc) and programming of the LUT codes (Control Proc). The Control Processor is where the actual fine-grained PDR APIs are invoked for dynamically reprogramming the system Sockets with the updated rules, in accordance with the self-reconfiguring nature of the fine-grained PDR as discussed in Section 2.3.

7 Empirical Evaluation

7.1 Experimental Setup

A prototype of the presented design is developed using Xilinx Virtex-II Pro XC2VP30[1] FPGA on Xilinx XUP Board and a PC-based workstation running Microsoft Windows-7 on Intel Core i7 Q740 Quad-core processor running at 1.73 GHz and having 4-GB memory. We used Xilinx 10.1.03 ISE and EDK tools in deploying the proposed rule processor. Xilinx Microblaze soft-core processors were instantiated on the FPGA fabric connected with FIFO-based Fast Simplex Links (FSLs). The data path, or the FPGA back-end, of the prototypes using the either the dynamic or static rule-processing unit maintained an operational clock frequency of 100 MHz.

1. The choice was made because API bugs were discovered in newer Virtex-V devices while performing fine-grained PDR.

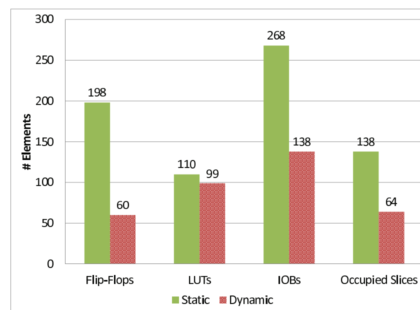


Fig. 9. Dynamic versus static comparison.

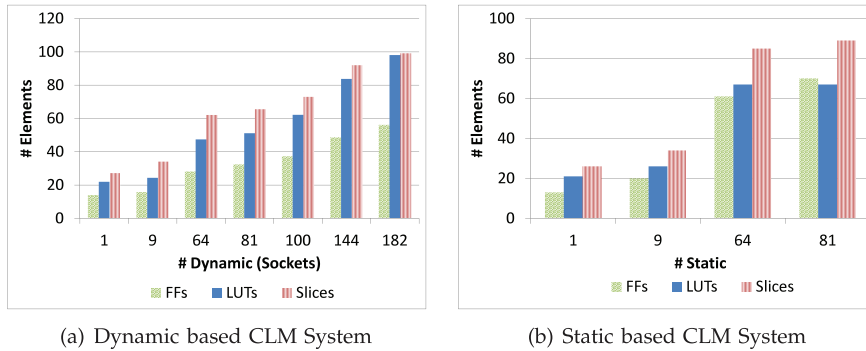
The availability of the Ethernet channel on the XUP board and the relative ease of deploying an Ethernet connection using available IP-cores in Xilinx EDK led us to choose Ethernet as our means of communication interface between the host-controller and the back-end. The Xilinx Ethernet IP-cores interface with the XUP board's Ethernet port using a soft-IP that is accessible via the data processor. As the processors are slow, the ease of deployment of Ethernet is traded off with lower Ethernet throughput. We, however, stress that this setup is only for prototype purposes and the data/control interface can easily be substituted with any suitable interface depending on the deployment requirements, budget, and available resources. The complete system setup is shown in Fig. 8.

7.2 Area Results

We first compare the area resources taken of the proposed Socket (also referred to here as dynamic) with an equivalent static solution mapped on the same FPGA. The discrete logic-based static follows the discussion in Section 3. The comparison is presented in Fig. 9.

It can be noticed that the static employs higher number of FFs and LUTs than the dynamic. These two elements form the core of device-slices and as such reflect in a higher slice consumption budget of static compared to the dynamic, a $2.15\times$ increase. However, as highlighted in Section 3.2, the main bottleneck for the static remains the significant FFs usage that it employs for rule and wild-card pattern storage. The issue is addressed in the dynamic through the rule fusion in LUTs. The results show the efficiency of our LUT fusion of rules in reducing the FF usage by 3.3 times. Interestingly, even though dynamic relies on LUTs for rule-matching, it still slightly outperforms the static in LUT usage, thereby leading to significant savings in FPGA slices.

Another novelty of our dynamic design has been the innovative mechanisms of remote reconfiguration and synchronization, instead of explicit input signals as is the case with static. The dividends of the scheme are partially reflected in the results, where dynamic employs nearly half the IO pins to its static counterpart.



(a) Dynamic based CLM System

(b) Static based CLM System

Fig. 10. CLM system area.

We next evaluate the back-end with different Data-Engine sizes, by varying the number of dynamic Sockets. The logic consumption distribution for the designs is shown in Fig. 10a. The base-design, that consists of all the logic except the Data-Engine, is seen to be taking approximately 20 percent of the logic budget. This can be noticed from the statistics of the design using a single socket. Despite the base-design cost, we were able to fit 182 sockets using the dynamic.

We also synthesize static-based CLM's Data-Engine for evaluating actual increase in rule-processing opportunities obtained using dynamic. To obtain a quick estimate, we make use of some simplifications on the part of the static that reduces its system level logic footprint. In particular, we disregard additional logic that may be required in uniquely identifying a static in a system employing multiple static rule-processing units. The results so obtained are presented in Fig. 10b.

The largest Data-Engine that we were able to synthesize using static consisted of 81 static units. The higher static Data-Engines, though within the device logic budget, failed to get mapped to the device. This is due to the increase in interface complexity of the static, that puts immense burden on the limited routing resources on the device, leading to reduced parallelism. It is also interesting to note that the difference between system sizes involving static and dynamic is less than what we have expected considering the logic differences between the two. This is due to the logic costs of the base-design that had significant LUT usage. The higher LUT usage leaves less logic resources for LUT-dominant dynamic than for FF intensive static. Nevertheless, the results still demonstrate a significant $2.2\times$ more rule-processing opportunity using dynamic over the static DEs.

7.3 Reconfiguration Latencies

We next discuss the reconfiguration latencies involved in reprogramming the socket using fine-grained PDR and compare it with those involved in static. The prototype, mapped to the XC2VP30 FPGA device, employs an 8-bit ICAP port having a frame size of 824 bytes and operating at 100 MHz. The basic unit of our reconfiguration, the LUT, is a 4-input LUT on the device that translates into 16 bits of reconfiguration data.

We not only discuss details of the latencies involved in our target prototype, but have a more generic discussion on possible future enhancements that we feel are likely in future fine-grained PDR FPGAs. Thus, besides empirical measurements using available API-based reconfiguration schemes (Measured), we also provide ideal theoretical bounds (Theoretical) that are based on the device parameters above and do not carry APIs implementation overheads.

The different reconfiguration scenarios (Sc) are detailed in Table 2. In the table, Sc-A represents an ideal reconfiguration scheme where addressing resolution zooms down to individual LUTs. Sc-B to G deal with frame-level addressing using ideal and available API-based reconfiguration schemes. These cases differ whether the API provisions for a single or multiple frame changes per API call, if the reconfiguration involves reading-in of the frame from the device (1R), and finally if writing of the frame in configuration memory is followed by a pad-frame (2W) or not (1W). The parameter *Remote* represents front-end (host) initiated reconfiguration over the Ethernet. Combining the above, Sc-G represents our front-end initiated Xilinx API-based LUT reconfiguration. For the sake of completeness, we also discuss reconfiguration delays using static schemes in the last two scenarios. The last column in Table 2 represents if the respective configuration is empirically measured (M) or theoretically calculated (T), or a combination of both.

The latencies associated with the above-mentioned scenarios are shown in Fig. 11. Every scenario is broken down in three cases: 1) a base value for a single LUT reconfiguration, 2) the 20-LUT dynamic reconfiguration, and 3) reconfiguration of a Data-Engine using 144 dynamic sockets (2,880 LUTs).

Ideally, reconfiguring a single LUT should take only $0.02\ \mu s$. This can be observed as Virtex-II LUT reconfiguration involves 16 bits to be rewritten using 8-bit ICAP port operating at 100 MHz. The reconfiguration of a socket or the data path in the ideal case involves simple scaling of the base value. However, things get more interesting in the frame-addressing modes as latencies directly depend on number of frames accessed for reconfiguration. The number

TABLE 2
Reconfiguration Scenarios

Case	Description	T/M
A	Ideal (Direct 1W)	T
B	Multi-Change, 1W	T
C	Multi-Change, 1R+1W	T
D	Multi-Change, 1R+2W	T
E	Multi-Change, 1R+2W + Remote	T
F	Single-Change, 1R+2W	M
G	Single-Change, 1R+2W + Remote	M
H	Static	M
I	Static + Remote	T

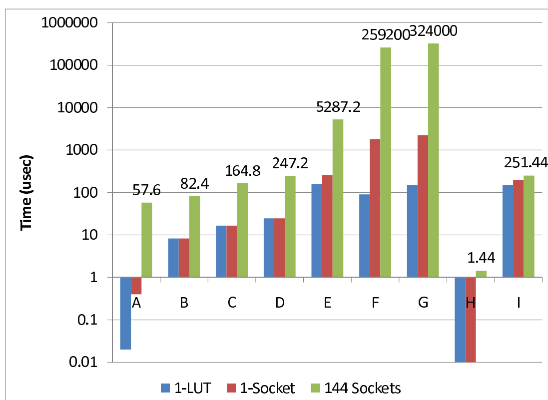


Fig. 11. Reconfiguration latencies.

of frame accesses in turn is tied with the placement of reconfigurable resources. In an optimally placed design, maximum number of reconfigurable resources should get targeted in minimum number of API calls. In our experiments, we consider such an optimally placed reconfigurable LUT network, spanning over entire frames. Indeed, the SPT places the CLM system's Data-Engine such that the reconfigurable LUTs span over entire frame lengths, and hence minimizing number of reconfigurable frames.

The Scenarios-B to Sc-E, provision for multiple frame changes. As such they can reconfigure the 20 LUTs placed in the same frame in a single call. The scenarios differ in number of frame reads and writes and are seen to have little effects on their latencies. The latencies, however, show significant increase when the API is constrained to provision for single frame change, as seen in Sc-F from Sc-D. The two cases should have identical latencies for reconfiguring single LUT. Furthermore, Sc-F also incorporates implementation overheads of the APIs running over slow on-chip processors. Similarly, Sc-E and Sc-G are comparable cases. The added latency in Sc-G is again due to on-chip processors API provisioning single-change per call.

In contrast to dynamic, static can be reconfigured in a single clock cycle. Sc-H presents latencies associated with such an ideal static Data-Engine. Sc-I accounts for remote static reconfiguration over Ethernet and is comparable with Sc-E and Sc-G.

7.4 Discussion

The logic-area results demonstrate the area efficiency of our proposed rule-matching fusion. By avoiding dedicated comparators, we were able to do the rule-processing using almost half the device resources (FFs and LUTs combined) than the static. The key enabler for such a fusion has been our proposed application specific fine-grained PDR using which we reconfigure the rules by only updating specific LUTs on the device. Furthermore, in a highly parallelized solution such as the proposed CLM solution, the dynamic has further area benefits that are external to the unit. This is due to its simplified external interface that no longer requires additional logic and routing resources to route reconfiguration and control information, as is the case with the static.

The latency results show that theoretically fine-grained PDR can be quite fast, as is evident in Sc-A. However, there exists a gap between theoretical and practical latencies using state-of-the-art reconfiguration mechanisms. The major contributor to the 0.25-second full Data-Engine reconfiguration in Sc-F is due to restrictions in available reconfiguration mechanisms. Our proposed scheme requires updating specific LUT bits within the FPGA configuration data. Detailed composition of the configuration data, however, is a Xilinx proprietary information. Xilinx API, which enables constrained manipulation of the LUT bits, was our only mechanism for updating the configuration bits of a specific LUT within the configuration data. Nevertheless, with conventional PDR latencies running in minutes, the available fine-grained PDR mechanisms still offer much lower reconfiguration overheads.

We believe that the current frame-based Xilinx reconfiguration paradigm is geared toward conventional PDR schemes and is quite easy to be improved for fine-grained PDR. One possible improvement could be to have higher addressing resolution of individual programmable points, such as LUTs, rather than the current frame. Yet, another desirable change that could significantly lower the latency bounds is by provisioning changes to the frame in a single API call. Furthermore, it would be quite reasonable to have fine-grained PDR being triggered remotely through the host processor using existing JTAG programming standards instead of happening over slow on-chip processors.

Fine-grained PDR enabled our CLM solution to pack higher number of Sockets and offer higher rule-processing parallelism than the static. The higher parallelism in answering of the rules imply higher accuracy in reported results. However, the available reconfiguration mechanisms come with increased reprogramming latencies as compared to static. The increase in latency implies lower accuracy in reported results as more data can now go unobserved while the rules are being programmed. As Sockets can process incoming data at much higher throughput (100 million packets per second) than the current network speeds, the problem can be quickly fixed by buffering the data that could potentially be missed being observed. We next discuss the interplay of higher rule-processing parallelism and reconfiguration latencies toward accuracy in the context of a real-time application.

7.5 Case Study: Heavy-Hitter Identification

As discussed earlier, traditional network measurements rely on conservative sampling and/or open-loop measurements and offline processing to cope with high link speeds. As such, the schemes not only incur inaccuracies in reported results but they are also quite slow for real-time network monitoring. One such network measurement problem is that of heavy-hitter identification that has generated quite an interest in the research community [3], [11]. In this section, we demonstrate the effectiveness of our fine-grained PDR-based CLM framework by mapping the problem of heavy-hitter identification to our CLM framework and addressing it in a goal based and streaming manner. We, however, stress that the proposed CLM framework is independent of any chosen problem and can be easily adapted for other applications by making due changes at its algorithmic-layer.

We hereby define the *heavy-hitter* or an *elephant* flow as a flow in two-dimensional {*source*, *destination*} tuple space that consumes θ-fraction of the entire traffic. At the algorithmic-layer of our CLM solution, we utilize a recursive top-down rule generation heuristic in isolating the heavy-hitter flows within a crowd of normal traffic. The

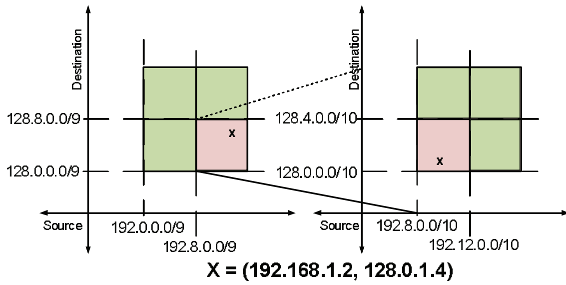


Fig. 12. MRT with zoom ratio of four.

scheme, called multiresolution tiling (MRT) algorithm [11], relies on a simple but powerful observation that if a subnet does not contain θ fraction of the entire traffic, then no flow in that subnet can be a heavy-hitter. Since subnets are usually described using standard CIDR prefix notation, the algorithm states that if a prefix is not a heavy-hitter, all the constituent prefixes of higher sizes can be discarded from further consideration.

An MRT iteration is illustrated in Fig. 12. Initially, the two-dimensional sample space is equally partitioned into four subregions. Corresponding rules to every subregion collect statistics for a measurement interval. The subregions that exceed the threshold, marked with a cross in the figure, are next selected for further zooming-in in the next iteration. MRT thereafter continues iterating between partitioning, statistics-collection and zooming-in phases until the heavy-hitter is isolated.

We evaluate our CLM solution by using CAIDA Backscatter data traces [27]. The CAIDA data trace is a low-volume trace, and as such, a good candidate for inserting varying amounts of traffic in it. We inserted 10 random flows contributing from 0.5 to 1.4 percent of the total traffic in the trace. We used heavy-hitter threshold value θ to be 1 percent. As such, the inserted flows split evenly between true and false heavy-hitters around the threshold value, thereby producing edge test cases for the system. We use the experimental setup discussed above using a Data-Engine employing 169 sockets and MRT measurement interval of 1 second.

We define a parameter *True-Score* (TS) to quantify the progress of MRT in identifying the heavy-hitters. Mathematically,

$$True\text{-}Score = \frac{\sum max|P_i|}{\sum |H_i|}, \quad (1)$$

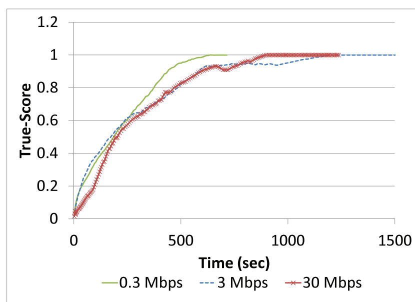
where $|P_i|$ represents the size of a prefix, P_i, in an MRT iteration that matches the heavy-hitter, H_i. As there could be a number of prefixes of various sizes that match a heavy-hitter in any given MRT iteration, we only take into account the maximally matching prefix or the prefix with the highest number of matching bits with the heavy-hitter. The parameter, thus, represents the degree by which the algorithm has correctly identified all the inserted heavy-hitters, with the maximum value 1 meaning all the heavy-hitters being completely identified.

The progression of TS for increasing link speeds using the proposed FPGA-based CLM solution is shown in Fig. 13. It can be noticed that the solution is able to correctly identify all the true heavy-hitters with increasing link speeds, as shown in Figs. 13a and 13b. As the proposed CLM solution is operational at 100 MHz and being able to process a packet per clock, the prototype can process 100-million packets/second; a figure high enough to easily meet the current link speeds.

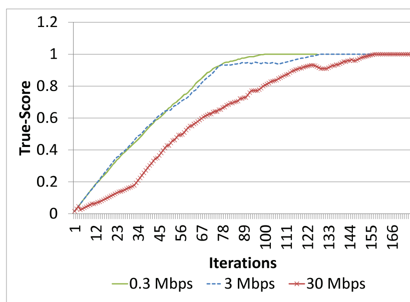
We also investigate the various latencies that contribute in the latency budget of our FPGA-based CLM solution, as shown in Fig. 13c. It is observed that communication over Ethernet and rule-deployment latencies contribute to almost 63 percent of the total delay. We believe that these values present opportunities for drastic latency improvements, not just because Ethernet can be readily replaced with faster solutions, but also because better and faster reconfiguration paradigms are likely. Finally, but quite significantly, the rule-synthesis latencies can be observed to be taking minimal times. The negligible amounts of rule-synthesis latencies showcase the ability of the solution to generate new rules on-the-fly, a critical requirement in closing the loop in a real-time setup.

8 Conclusion

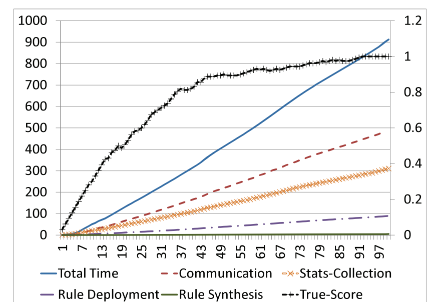
The work presented a novel realization of a closed-loop measurement system employing innovative usage of fine-grained PDR as underlying reprogramming paradigm and demonstrated its effectiveness in isolating heavy-hitter flows. We discussed the challenges associated with fine-grained PDR and offered solutions for its system wide integration, with many of the presented techniques and algorithms being quite generic and transcending the needs of CLM system. We integrated our techniques in an innovative dynamic rule-processing unit that offers 3.3× logic improvements as well as 48 percent reduction in IO interface over conventional solutions, leading to 2.2× increase in system-level rule



(a) TS Progression with Time



(b) TS Progression with MRT Iterations



(c) Latency Breakdown

Fig. 13. Proposed FPGA-based CLM solution.

processing resources. The savings are achieved with much reduced latency overheads as compared to conventional PDR schemes. Our analysis of the latencies suggests that they can be further improved with simple updates to ICAP API, leading to application of fine-grained PDR in a wide variety of applications.

REFERENCES

[1] "Cisco NetFlow," http://www.cisco.com/warp/public/732/Tech/netflow, 2013.

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

[3] C. Estan and G. Varghese, "New Directions in Traffic Measurement and Accounting: Focusing on the Elephants, Ignoring the Mice," *ACM Trans. Computer Systems*, vol. 21, no. 3, pp. 270-313, 2003.

[4] F. Khan, L. Yuan, C.-N. Chuah, and S. Ghiasi, "A Programmable Architecture for Scalable and Real-Time Network Traffic Measurements," *Proc. ACM/IEEE Fourth Symp. Architectures for Networking and Comm. Systems (ANCS '08)*, 2008.

[5] P. Garcia, K. Compton, M. Schulte, E. Blem, and W. Fu, "An Overview of Reconfigurable Hardware in Embedded Systems," *EURASIP J. Embedded Systems*, vol. 2006, p. 13, 2006.

[6] D. Lim and M. Peattie, "Two Flows for Partial Reconfiguration: Module Based or Difference Based," *Xilinx Application Notes*, vol. 1.2, 2004.

[7] O. Blodget, P. James-roxby, E. Keller, S. Mcmillan, and P. Sundararajan, "A Self-Reconfiguring Platform," *Proc. Field Programmable Logic and Applications*, pp. 565-574, 2003.

[8] F. Khan, N. Hosein, S. Vernon, and S. Ghiasi, "BURAQ: A Dynamically Reconfigurable System for Stateful Measurement of Network Traffic," *Proc. IEEE Ann. Symp. Field-Programmable Custom Computing Machines*, pp. 185-192, 2010.

[9] K. Papadimitriou, A. Dollas, and S. Hauck, "Performance of Partial Reconfiguration in FPGA Systems: A Survey and a Cost Model," *ACM Trans. Reconfigurable Technology Systems*, vol. 4, no. 4, pp. 36:1-36:24, Dec. 2011.

[10] J. Mai, A. Sridharan, C. Chuah, T. Ye, and H. Zang, "Impact of Packet Sampling on Portscan Anomaly Detection," *IEEE J. Selected Areas in Comm. - Special Issue on Sampling the Internet*, vol. 24, no. 12, pp. 2285-2298, Dec. 2006.

[11] L. Yuan, C.-N. Chuah, and P. Mohapatra, "ProgME: Towards Programmable Network Measurement," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '07)*, pp. 97-108, 2007.

[12] H. Song and J.W. Lockwood, "Efficient Packet Classification for Network Intrusion Detection Using FPGA," *Proc. Int'l Symp. Field-Programmable Gate Arrays (FPGA '05)*, 2005.

[13] D.E. Taylor, "Survey and Taxonomy of Packet Classification Techniques," *ACM Computing Surveys*, vol. 37, no. 3, pp. 238-275, Sept. 2005.

[14] P. Gupta and N. McKeown, "Algorithms for Packet Classification," *IEEE Network*, vol. 15, no. 2, pp. 24-32, Mar./Apr. 2001.

[15] T.V. Lakshman and D. Stiliadis, "High-Speed Policy-Based Packet Forwarding Using Efficient Multi-Dimensional Range Matching," *ACM SIGCOMM Computer Comm. Rev.*, vol. 28, pp. 203-214, 1998.

[16] S. Singh, F. Baboescu, G. Varghese, and J. Wang, "Packet Classification Using Multidimensional Cutting," *Proc. ACM Conf. Applications, Technologies, Architectures, and Protocols for Computer Comm. (SIGCOMM '03)*, pp. 213-224, 2003.

[17] W. Jiang and V.K. Prasanna, "Scalable Packet Classification on FPGA," *IEEE Trans. Very Large Scale Integration Systems*, vol. 20, no. 9, pp. 1668-1680, Sept. 2012.

[18] J.W. Lockwood, N. McKeown, G. Watson, G. Gibb, P. Hartke, J. Naous, R. Raghuraman, and J. Luo, "NetFPGA—An Open Platform for Gigabit-Rate Network Switching and Routing," *Proc. IEEE Int'l Conf. Microelectronic Systems Education*, pp. 160-161, 2007.

[19] N. Weaver, V. Paxson, and J.M. Gonzalez, "The Shunt: An FPGA-Based Accelerator for Network Intrusion Prevention," *Proc. ACM/SIGDA 15th Int'l Symp. Field Programmable Gate Arrays (FPGA '07)*, pp. 199-206, 2007.

[20] N. McKeown, T. Anderson, H. Balakrishnan, G. Parulkar, L. Peterson, J. Rexford, S. Shenker, and J. Turner, "Openflow: Enabling Innovation in Campus Networks," *ACM SIGCOMM Computer Comm. Rev.*, vol. 38, pp. 69-74, Mar. 2008.

[21] "Xilinx," http://www.xilinx.com, 2013.

[22] J.W. Lockwood, N. Naufel, J.S. Turner, and D.E. Taylor, "Reprogrammable Network Packet Processing on the Field Programmable Port Extender (FPX)," *Proc. ACM/SIGDA Ninth Int'l Symp. Field Programmable Gate Arrays (FPGA '01)*, 2001.

[23] G. Memik, S.O. Memik, and W.H. Mangione-Smith, "Design and Analysis of a Layer Seven Network Processor Accelerator Using Reconfigurable Logic," *Proc. IEEE 10th Ann. Symp. Field-Programmable Custom Computing Machines*, 2002.

[24] E.L. Horta, J.W. Lockwood, D.E. Taylor, and D. Parlour, "Dynamic Hardware Plugins in an FPGA with Partial Run-Time Reconfiguration," *Proc. 39th Conf. Design Automation (DAC '02)*, pp. 343-348, 2002.

[25] S. Guccione, D. Levi, and P. Sundararajan, "JBits: A Java-Based Interface for Reconfigurable Computing," *Proc. Second Military and Aerospace Applications of Programmable Devices and Technologies Conf. (MAPLD)*, 1999.

[26] E. Bergeron, L. Perron, M. Feeley, and J.P. David, "Logarithmic-Time FPGA Bitstream Analysis: A Step Towards JIT Hardware Compilation," *ACM Trans. Reconfigurable Technology and Systems*, vol. 4, article 12, May 2011.

[27] "CAIDA: Cooperative Association for Internet Data Analysis," http://www.caida.org, 2013.

Faisal Khan received the BE degree from NED University, Karachi, Pakistan, and the MS degree from KFUPM, Dhahran, Saudi Arabia in 2001 and 2005, respectively, and the PhD degree in electrical and computer engineering from the University of California, Davis, in 2012. He is currently working on developing high-speed communication protocols at Altera, San Jose. He has also been a computational intern at Lawrence Livermore National Laboratory. His research interests include providing embedded system solutions for mission critical and real-time problems.

Soheil Ghiasi received the BS degree from Sharif University of Technology, Tehran, Iran in 1998, and the MS and PhD degrees in computer science from the University of California, Los Angeles in 2002 and 2004, respectively. He is an associate professor of electrical and computer engineering at the University of California, Davis. His research interests include architecture, design methodologies, and design automation techniques for embedded systems. He has served on the organizing and technical program committees of numerous conferences, and currently serves as an associate editor of the *Journal of Reconfigurable Computing*.

Chen-Nee Chuah received the BS degree from Rutgers University, and the MS and PhD degrees in electrical engineering and computer sciences from the University of California, Berkeley. She is a professor in electrical and computer engineering at the University of California, Davis. Her research interests include Internet measurements, network management, anomaly detection, online social networks, and vehicular ad hoc networks. She received the US National Science Foundation (NSF) CAREER Award in 2003, and the Outstanding Junior Faculty Award from the UC Davis College of Engineering in 2004. In 2008, she was named a Chancellors Fellow of UC Davis. She has served on the executive/technical program committee of several ACM and IEEE conferences and is currently an associate editor for *IEEE/ACM Transactions on Networking*.