Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2015, Article ID 673596, 31 pages
http://dx.doi.org/10.1155/2015/673596

Research Article

Optimization of Lookup Schemes for Flow-Based Packet Classification on FPGAs

Carlos A. Zerbini[1,2] **and Jorge M. Finochietto**[1]

[1] *Laboratorio de Comunicaciones Digitales, Universidad Nacional de Córdoba and Consejo Nacional de Investigaciones Científicas y Técnicas (CONICET), 5000 Córdoba, Argentina*
[2] *Departamento de Ingeniería Electrónica, Universidad Tecnológica Nacional, 5000 Córdoba, Argentina*

Correspondence should be addressed to Carlos A. Zerbini; czerbini@electronica.frc.utn.edu.ar

Received 11 September 2014; Accepted 6 January 2015

Academic Editor: Michael Hübner

Packet classification has become a key processing function to enable future flow-based networking schemes. As network capacity increases and new services are deployed, both high throughput and reconfigurability are required for packet classification architectures. FPGA technology can provide the best trade-off among them. However, to date, lookup stages have been mostly developed as independent schemes from the classification stage, which makes their efficient integration on FPGAs difficult. In this context, we propose a new interpretation of the lookup problem in the general context of packet classification, which enables comparing existing lookup schemes on a common basis. From this analysis, we recognize new opportunities for optimization of lookup schemes and their associated classification schemes on FPGA. In particular, we focus on the most appropriate candidate for future networking needs and propose optimizations for it. To validate our analysis, we provide estimation and implementation results for typical lookup architectures on FPGA and observe their convenience for different lookup and classification cases, demonstrating the benefits of our proposed optimization.

1. Introduction

Nowadays, the increasing network traffic and services claim for much more efficient packet processing. Since most traffic is packet-based, processing time for each packet decreases as data rates do increase, thus, requiring better processing performance. Besides, services which used to require just best effort processing have evolved in terms of number, complexity, and traffic volume. Initiatives such as Software Defined Networking (SDN) allow to modify processing tasks through OpenFlow [1], thus, enabling a much more flexible networking infrastructure.

For these services to be successful, proper performance must be guaranteed. Technology offers several options for implementing packet processing, such as network processors (NPs), custom application-specific integrated circuits (ASICs), and field programmable gate arrays (FPGAs). Among them, FPGAs have been widely adopted for mid-sized and flexible solutions [2]. As resources on these platforms become richer and more efficient as technology evolves, the performance gap with respect to ASICs tends to narrow in time in practical applications due to the higher costs associated to ASIC development. In addition, today's available development tools make these platforms accessible to hardware designers, software developers, and network specialists [3].

In order to enable service-oriented network architectures, new processing functions have been added to traditional longest-prefix routing. Among them, *classification* is one of the most challenging. Packet classification enables a lot of internet services such as policy-based routing, traffic accounting, load balancing, and access control, by assigning packets to different flows and accordingly defining their processing paths. A classifier analyzes a packet header or *key* against a set of predefined filters or *rules* having associated *actions*. According to matched rules, the corresponding actions are applied to the packet.

A packet header can be divided into several *fields* of interest, which are orthogonal to each other. That is, they are defined independently and then combined into rules. On each of these fields, a *lookup* process is performed; from the combination of such lookups on multiple fields packet *classification* is achieved. In the general case, rule fields are specified as *generic ranges* (GR) of field values. In 1-dimensional space (1D) (i.e., one field) these ranges define line segments, while they define hyper-rectangles in the k-dimensional (kD) space. Four particular 1D range cases can be identified, that is, *prefixes* (PX), *exact values* (EX), *arbitrary ranges* (AR), and *wildcards* (WC). Prefixes are ranges bounded by powers of two, so they can be expressed by using "don't care" bits at least significant positions. Exact values are ranges with equal lower and upper bounds; arbitrary ranges have arbitrary bounds, while wildcards cover the entire range of field values. Prefixes are commonly used at network layer (L3 in TCP/IP stack) for Classless-InterDomain Routing (CIDR). Exact values are used for specifying hosts at L3, protocol-based filtering at L3, and port-based filtering at transport layer (L4 in TCP/IP stack). Arbitrary ranges are used for fields such as L4 ports specifying sessions, while wildcards are used for indicating fields whose value is irrelevant.

Keys used for classification are traditionally based on 5 tuples formed by 32-bit L3 source/destination IP addresses (SrcIP/DstIP), 16-bit source/destination L4 ports (SrcPT/DstPT), and 8-bit L4 protocol (Prot), summing altogether 104 bits. Next-generation networks, meanwhile, consider using up to 12 tuples in order to support new specifications such as OpenFlow. This implies taking into consideration fields such as VLAN ID, source/destination Ethernet addresses, Type of Service and Ethernet Type, summing up to 12 fields with 237-bit length. Each of these fields involves different range cases such as PX, AR, or EX.

Given its importance for CIDR in IP networks, the problem of prefix-based lookup has been extensively studied and many effective solutions exist for it on diverse technologies [4–7]. The problem of arbitrary range match, meanwhile, has seen increasing importance for protocols requiring more than best-effort IPv4 routing. Even if some effective solutions have been achieved to date, they stick to specific implementation and lack an objective comparison with other approaches. Moreover, it is unclear how effectively they can fit typical architectures for multifield classification.

Main drivers of current research on packet classification are, namely, (i) line-speed processing, (ii) best-match/multimatch support, and (iii) dynamic updating capability. Line speed means that packets must be dispatched from the classification module at the same rate that new ones arrive to it, even if some latency could be introduced in the process. Since packet size is variable, line speed is commonly associated to the worst case which is the smallest packet. Widely deployed Ethernet networks set this worst case to 40 bytes, that is, 3.2 ns/packet for 100 Gbps Ethernet (i.e., $313 \cdot 10^6$ packets per second). In applications such as IP routing, the result of interest is the best-match (BM) among the considered rules, which can be the most specific prefix as implemented in IP routing or based on some predefined priority scheme assigned to rules during updating. New applications, however, require different actions to be taken according to the particular combination of multiple matches (MM), for example, for accounting or intrusion detection through SNORT [8]. In general, MM poses quite different requirements than BM to packet lookup and classification schemes. This is of utmost importance considering future flow-based networking schemes, which require mixed BM/MM lookups. Dynamic updating, meanwhile, was not an issue in the past as routing tables were relatively static. In present applications, however, classification rules tend to be highly dynamic [9], so updating complexity should be considered in new approaches.

In this work, we take into account the aforementioned factors and our previous experience [10] and apply them to the generic lookup case. On this basis, we make the following main contributions.

(i) We provide upper bounds for lookup and updating complexity which help recognize critical aspects of lookup schemes. On this basis, we provide a through comparative analysis of current approaches for the three main lookup cases, that is, exact/prefix/arbitrary range and BM/MM. We then focus on arbitrary range lookup and provide a general taxonomy of the existing approaches which eases their qualitative comparison. From this taxonomy, we finally extract typical examples and compare them quantitatively.

(ii) We take into consideration not only the required unifield lookup case but also the adopted multifield classification architecture. Contrary to previous work which focuses on one of both aspects, our methodology is in line with current flow-based networking schemes, where MM/BM, PX/AR/EX schemes are required on the one hand and multiple fields are considered on the other hand.

(iii) We extend the geometrical interpretation of rules, commonly applied in multifield classification, to the case of stitched lookup architectures. In particular, we recognize new geometric spaces for AR lookup which are not present in multifield classification. This methodology greatly eases the analysis of lookup and classification architectures on a common base and helps recognize new bounds for implementation.

(iv) We implement representative lookup cases on FPGA technology and compare estimated results with those from real implementation. In particular, we enhance explicit range match and memory indexing architectures for supporting arbitrary range match on FPGAs.

(v) From our analysis of existing lookup schemes, we identify the most appropriate one for MM lookup on FPGAs. Implementations of this scheme currently suffer from very inefficient incremental updating, so we propose an optimization in this aspect. Particularly, we approach incremental updating of RFC, which was not considered to date.

(vi) On the basis of our theoretical and empirical results, we provide selection criteria for lookup architectures on FPGA and how to combine them effectively. To the best of our knowledge, this is the first work which aims to integrate and compare lookup schemes on a common basis, which can help to recognize the most appropriate ones for future networking applications on FPGA.

The paper is organized as follows. In Section 2, we review lookup cases and their recent solutions aiming for FPGA implementation. In Section 3 we discuss the complexities of such cases analytically. From this discussion, we reach a general taxonomy of the lookup process and use it for comparing solutions. Section 4 focuses on trade-offs for implementation of such solutions on FPGA devices. We then compare results of these implementations against estimations in Section 5. Finally, we draw our conclusions in Section 6.

2. Related Work

We must first clearly distinguish packet lookup from packet classification. Lookup processes selected bits of the packet header according to a common matching criterion. These selected groups, commonly consecutive and named packet *fields*, can adopt a number of binary values; selected values of that packet field are then grouped and said to match different *rules*, *entries*, or *filters* in a ruleset. Grouping can be by defining single exact values, hierarchical grouping in prefixes, or arbitrary grouping in ranges of values. Fields are essentially independent in that we can obtain independent match results for each field; so we say that fields are orthogonal to each other. Lookup process on a certain field returns BM or MM results which could be directly used to take some decision on the packet. When implementing classification on multiple fields, however, decisions on different fields must be considered altogether. Two main approaches exist for this, that is, multifield *decision trees* which consider all fields at once and incrementally reduces multidimensional classification space and *decomposition-based* approaches which process field lookups independently and then consider their interrelations to make a decision. In the former, the concepts of lookup and classification are indistinguishable, while they are clearly different in the latter. Multifield decision trees can be more or less efficient depending on the ruleset pattern, while decomposition-based classification adapts better to different rulesets and is naturally suited to highly concurrent hardware architectures. Extensive work exists on packet classification; we can mention [11] as example of tree-based classification and [12–16] as examples of decomposition-based classification. Other schemes were also proposed [17, 18] using hashing techniques and Bloom filters. However, they require multiple memory accesses and high routing complexity on FPGAs, while Bloom filters can return false matches; all of these factors lead to limited performance. In this context, we specifically focus our work on packet field *lookup schemes* applicable to decomposition-based classification. In particular, we exclude hashing-based schemes in favor of simple and deterministic lookup schemes. Some general concepts are naturally common to both classification and lookup; however, essential differences exist which we will note throughout the paper. In the following paragraphs, we review previous work specifically regarding packet lookup.

The naive approach to MM packet lookup is Linear Searching (i.e., the packet field is sequentially matched against rules until the last rule is reached). This scheme requires no more storage than that required by the ruleset but can take excessive time for rulesets containing many entries. Content-addressable memory (CAM) technology and its ternary version (TCAM), collectively known as associative arrays, solve this problem by concurrently matching every single bit of every rule against the packet header. They are the natural choice when the requirements of line-speed and MM lookup must be met at the same time [19, 20]. However, TCAMs are expensive application-specific devices with high energy demands. In addition, they do not natively support arbitrary range match. Given M-bit key and N rules, TCAMs can suffer from rule expansion when encoding a W-bit range which can be upper bounded to M rules in the worst case [21]. Multiple efforts have tried to mitigate both drawbacks by adding power-controlling hardware in the chip or preencoding of ranges for efficient TCAM mapping; we can mention CoolCAMS [22], ETCAMS [23] and Split [24] as representative examples.

TCAMs compare all M bits of a field at the same time against the N rules; this extensive parallelism which makes them so attractive for line-speed lookup is also the main source of their drawbacks. As a result, algorithmic techniques have emerged which incrementally reduce the lookup space in multiple steps while keeping acceptable performance. These techniques have been mapped to hardware by applying pipelining and taking advantage of SRAM and DRAM technology, much cheaper than TCAM cells due to their production scale. One of these algorithmic techniques is based on tries. A trie is a special tree where the path from root to leafs is defined by the state of successive bits of the key. Tries can process key bits one at a time, resulting in simple unibit tries or in fixed/variable strides resulting in multibit tries with different throughput versus resource consumption trade-offs. The processing in strides serves to compress tries from different observations such as multiple single-branch nodes (path compression), multiple complete-branch levels (level compression), or multiple leaves having the same decision in a multibit trie (Lulea scheme) [25, 26]; however, these techniques make incremental updating very difficult and slow. A combination of them, maintaining fast updates, is proposed in tree bitmap scheme [25]. More recent proposals aim at keeping high lookup rates while supporting IPv6 extended-size keys. To this end, they adopt hash-based schemes [5, 27]. Even if trie-based techniques can be effective for IP addresses, they tend to exploit the prefix structure, which is a particular case of arbitrary ranges. Finally, it is worth mentioning that they are mostly subject to patents which hinders its adoption in research work [26].

Binary search tree schemes are another option for IP lookup which makes branching decisions on criteria other than following key bits in descending order. The cost for this abstraction is a M-bit magnitude comparator implemented in hardware and memory storing M-bit range bounds for each tree stage. One possible variant, which still takes advantage of hierarchical nature of IP addresses, is grouping rules according to IP prefix length and taking branching decisions according to such lengths. In this way, lookup on each stage is implemented through hashing on fixed-length prefixes [28]. This architecture is attractive for tables containing many wide-key entries sharing few prefix lengths; on the other hand, the use of hashing makes it unpredictable for worst cases. The other variant, binary search on ranges, is based on defining new, nonoverlapping intervals from original, overlapping IP prefixes. On this basis, lookup process compares against endpoints of such ranges at each tree stage and finds the enclosing interval which is associated with a best-match prefix. Even if they are originally slower than multibit tries, range trees have motivated abundant research [29, 30]. From the tree-based ones, this approach shows as the more amenable to arbitrary ranges. No specific analysis exists, however, about its performance against competing techniques for this match case; that is why we include it in our study.

Tree-based schemes are commonly pipelined on hardware in order to achieve high throughput, where memory is distributed in reduced size modules along tree stages. Since stages require increasing memory as we move towards tree leaves, strong memory imbalance can result. The difficulty in this case resides in achieving the best memory efficiency and throughput. Recent work, such as [30], strongly concentrates on this aspect for the specific case of FPGA technology. Complex combinations of double-port SRAM and DRAM memory technologies, with their specific requirements, are considered in order to achieve balanced memory consumption and scalability for large and wide rulesets. In particular, external DRAM with complex interfaces is exploited at lowest stages of the tree to support rulesets of 256 K IPv6 prefixes at speed as high as 400 mega-lookups per second (Mlps).

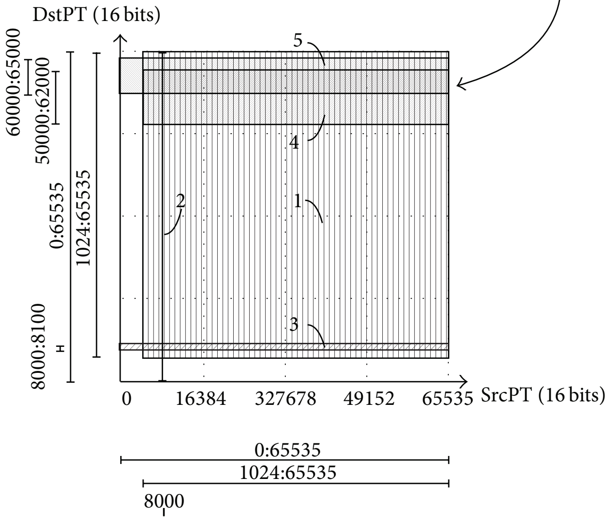
Two techniques commonly applied to trie- and tree-based lookup are *leaf pushing* and *node grouping*. In a general tree, a node staying at an intermediate level can hold *both* pointers to next nodes and decisions on the packet. Leaf pushing pushes down decision information to leaf nodes and make intermediate nodes hold just branching information; in this way, they reduce memory consumption by making a node hold *either* pointers or next hop information but not both. Node grouping aggregates multiple branching decisions in a single node; performance is thus increased since less levels are required, even if memory consumption at each node is higher. Even if good results were obtained for tree-based prefix lookup on large rulesets, they are still quite dependent on ruleset features.

In recent years, increasing interest has emerged on emulation of CAM and its derivatives on FPGA technology. FPGAs offer heterogeneous hardware resources such as combinational logic implemented on Look-Up Tables (LUTs), registers, and SRAM blocks, which enable exploring highly concurrent schemes for CAM optimization with reduced cost and high performance. Since FPGAs do not presently include native TCAM cells, its behavior is emulated with available resources. Moreover, FPGAs allow implementing modified versions of CAMs, such as extracting all individual matches instead of the best match (i.e., getting rid of the priority encoder intrinsic to CAMs). The first work to explore a combination of native TCAM (external to FPGA) and TCAM emulation in FPGA for range support was [31]; however, the adopted emulation scheme was an inefficient replacement for TCAM and no further options were considered at that time. PCIU [32], meanwhile, presented a more complete implementation and evaluation of CAM emulation on both software and FPGAs. StrideBV [33] is a very similar approach which considers a more efficient implementation by taking advantage of different memory form factors present in FPGAs. Our previous work [10], meanwhile, fully explored the potential of RAM-based TCAM emulation on FPGAs and, even more important, identified interesting opportunities for arbitrary range support through controlled RAM addressing expansion. A comparison of StrideBV [33] (distRAM and BRAM versions) against SRL16E-based TCAM is also presented in [34]. Such implementations are not fully portable, while their performance is limited and highly dependant on specific distRAM and SRL16E features. Later work [35] builds on these concepts and optimizes resource consumption for the case of narrow ranges by taking advantage of distributed RAM (distRAM) present in some FPGA devices. Despite the fact that not all FPGAs equally offer or even support distRAM, this kind of memory heavily consumes LUTs for storage of ruleset which could otherwise be used for logic-intensive operations. In addition, [35] adopts explicit range match (ERM) for wide ranges by emulating ETCAM ideas [23] on FPGA. ERM stores bounds explicitly on registers, accesses them concurrently, and compares them against the incoming key for each rule. Strided and clustered bit vector (SCBV) [36], finally, integrates these contributions in a highly pipelined systolic architecture and focuses on efficient updating schemes for it. It achieves high throughput on 1 K 12-tuple OpenFlow rulesets by intensive use of distributed RAM memory present in FPGA LUTs. Authors claim using distRAM for both RAM-based exact/prefix support and ERM-based arbitrary range support, reducing the extensive register consumption of [35]; however, it is unclear how this scheme implements ERM through distRAM. Even if distRAM-based ETCAM emulation has already been proposed and effectively implemented in [12], that architecture is essentially different than that in [36] since match results for each range bound are precomputed; that is, the M-bit comparator is eliminated and replaced by offline computation.

Even if ERM can be a very simple and effective resource for definitively solving the AR match problem of TCAMs, care must be taken since it presents some key trade-offs. ERM requires accessing $2 \cdot M \cdot N$ registers concurrently, so these registers must be routed individually to their associated comparators; in addition, all of them are accessed for each lookup requiring much power. In other words, ERM repeats trade-offs intrinsic to TCAM architecture, even if highest throughput with minimal storage is achieved for moderate rulesets; both are power hungry schemes with limited scalability due to their storage granularity and wiring requirements. Even if these problems can be mitigated by fine-grained pipelining, diminishing returns of such techniques still limit scalability. Even worse, ERM relies on FPGA technology to be implementable, while ETCAMs can already perform the same function on top-performance ASICs. As we will see in our next discussion, in many cases subtle facts about current packet

Rule	SrcIP (32 bits)	DstIP (32 bits)	SrcPT (16 bits)	DstPT (16 bits)
1	204.152.184.34/32	204.152.186.10/21	1024:65535	1024:65535
2	95.105.143.10/32	194.252.142.131/32	8000:8000	0:65535
3	175.77.88.0/23	194.252.142.131/32	0:65535	8000:8100
4	95.105.142.0/23	95.102.0.0/16	1024:65535	50000:62000
5	175.77.89.0/23	36.165.103.1/15	0:65535	60000:64000

(a)



(b)

Figure 1: Real ruleset: (a) table of rule specifications and (b) geometric view of application-level fields (SrcPT/DstPT).

lookup applications can be smartly exploited to implement AR match on FPGAs without recurring to ERM.

One important aspect, common to all mentioned TCAM and ETCAM emulation architectures, is that they assume lookup operations on individual rules (even if ETCAMs add both effective range matching and reduction of power consumption to original TCAMs, we will focus on their features regarding range matching, namely ERM. For sake of generality, we refer to CAMs, TCAMs and ETCAMs with the generic term "CAMs" if not otherwise noted). As mentioned in [34], this fact makes them agnostic of ruleset features; however, it also forces them to scale linearly with N. Even if it is possible to compress N-wide vectors into reduced-width ones [13, 14], CAM emulation schemes cannot internally produce such vectors but need an additional mapping stage. In addition, as we will see, the only lookup scheme that can be easily adapted for reduced-size output is memory indexing. ERM, on the contrary, cannot produce those outputs without additional stages. Considering these facts, we propose a general analysis on FPGA, considering not only memory and logic-based lookups but also rule- and so-called region-based results.

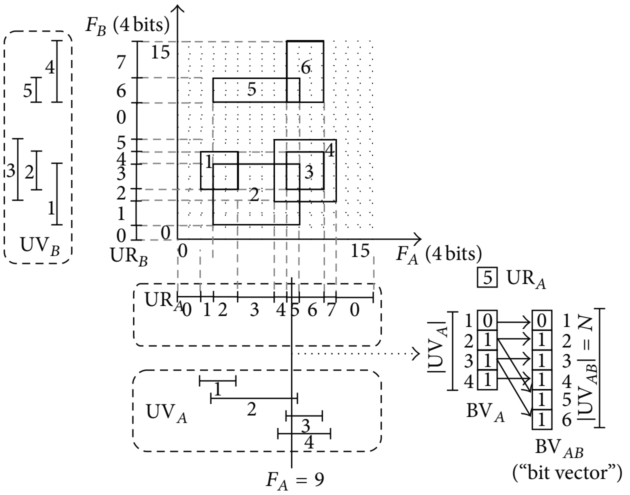
3. Analysis

3.1. General Considerations. We introduce our analysis by considering a simple example of a real ruleset. Rulesets present in real scenarios are difficult to obtain due to security and confidentiality issues. Reference [37] obtained 12 real filter sets from internet service providers (ISPs), a network equipment vendor, and other researchers and defined metrics for characterization of them. On this basis, they developed a set of tools for reproduction and control of such features; with these tools the research community is able to generate new rulesets which keep original structure without need of distributing the original ones. The considered rulesets have three main formats, that is, Access Control Lists (ACLs), Firewalls (FW), and IP Chains (IPCs), all of them considering 5 tuples (i.e., 104 bits), while ruleset size is configurable from tens to some thousands of rules. Two main components define structure of traditional rulesets: SrcIP/DstIP tuple, which defines communicating nets and their sizes and SrcPT/DstPT/Prot tuple which represents applications communicating through such nets.

In Figure 1(a) a small ruleset is considered involving 5 tuples as mentioned in Section 1. Even if this example is intentionally small for clarity, it represents the structure of real sets as those generated with Classbench. In particular, Prot field is excluded since it adopts either EX or WC values. As shown, networking fields SrcIP/DstIP are defined on prefixes, while application fields SrcPT/DstPT can also implement arbitrary ranges. We will now consider the SrcPT/DstPT fields since they are able to represent both PX and AR cases. It is worth noting that even though two fields are considered for illustration purposes these tuples can be eventually aggregated with SrcIP/DstIP and Prot lookup results to obtain multifield classification results. Figure 1(b) shows the so-called *geometrical representation* of application-layer SrcPT/DstPT tuples.

Rule (UV_{AB})	F_A	F_B	UV_A	UV_B	UR_A	UR_B
1	9:11	4:6	1	2	1, 2	3, 4
2	3:9	1:5	2	1	2, 3, 4, 5	1, 2, 3
3	9:11	4:6	3	2	5, 6	3, 4
4	8:12	3:7	4	3	4, 5, 6, 7	2, 3, 4, 5
5	3:9	11:12	2	5	2, 3, 4, 5	6
6	9:11	11:15	3	4	5, 6	6, 7

(a)



(b)

FIGURE 2: Ruleset example: (a) list of entries and (b) geometric view.

In this simple example, we can observe the presence of overlappings between 2D tuples. For example, tuple 2 involves ranges 8000 : 8000 (i.e., EX case) and 0 : 65535 (i.e., WC) on fields SrcPT and DstPT, respectively. Tuple 4, meanwhile, results in aggregating ranges 1024 : 65535 and 50000 : 62000 respectively.

In order to discuss general classification problems and make our conclusions more intuitive, we now consider a generic 2D ruleset. From our previous real example, however, we note that our discussion effectively includes real cases. Even more important, it also considers general cases which makes it valid for future applications involving other rule patterns. For the purposes of our discussion, in Figure 2(a) we show a simple ruleset involving two fields A and B with four bits each, while Figure 2(b) shows the geometrical representation of this ruleset where field values F_A and F_B are represented on orthogonal axes. This representation is indeed a generalization of the real case of Figure 1(b) considering any match case (i.e., generic ranges).

In a general multidimensional classification scheme, rules result from the combination of so-called *unique values* (UVs) at each field. In Figure 2(b), UV_A and UV_B for each field are represented by line segments, and their aggregation results in UV_{AB} as rectangles. UVs represent ranges of key values and can involve $1, 2, \ldots, k$ fields of the key. They are unique for the considered space; for example, 2-field rules 2 and 5 of Figure 2(b) are unique in the 2D space even if they share a single $UV_A = 2$ in 1D field A. Conversely, a UV in a field could be involved in more than one rule (in this work, since we mostly focus on unifield lookup, we may use the terms UV and rule interchangeably unless explicitly noted while making reference to multifield aggregation); for example, $UV_A = 2$ is involved in both $UV_{AB} = 2$ and $UV_{AB} = 5$ while $UV_A = 3$ is involved in both $UV_{AB} = 3$ and $UV_{AB} = 6$. In other words, UVs cannot repeat in their associated spaces, but they can repeat during aggregation into a higher-order one; that is why the number of UVs $|UV|$ generally increases as we consider higher-order spaces. That is also the reason why the complexity of single-field lookup stages in decomposition-based classification is generally much lower than the complexity of multifield trees which consider kD space from the beginning. In Figure 2, $|UV_A| = 4$ UVs exist for field A, $|UV_B| = 5$ for field B, and $|UV_{AB}| = 6 = N$ for aggregation results of both. In this case just 2 fields are considered, so aggregation results $|UV_{AB}|$ are essentially the considered N rules. Rule $UV_{AB} = 1$ results from the combination of $UV_A = 1$ and $UV_B = 2$ and rule $UV_{AB} = 2$ involves $UV_A = 2$, $UV_B = 1$, and so on. As seen in Figure 2, UVs can overlap with each other which causes the MM problem. For discrimination of all overlapping cases, unique *combinations* of UVs define so-called *Unique Regions* (URs) on the considered field or aggregation stage. In Figure 2, for example, $|UR_A| = 8$ URs exist for field A, $|UR_B| = 8$ for field B, and $|UR_{AB}| = 11$ for aggregation results of both (this can be demonstrated by taking all combinations of UR_A and UR_B which are valid for the ruleset). In general, the number of both UVs and URs involved at each stage of a multiple-field decomposition-based classifier increases as we aggregate lookup results; however, they actually increase much slower than their Cartesian product. For every field value F_A and F_B, we can build $|UV_A|$ and $|UV_B|$-sized bitmaps BV_A and BV_B, respectively, representing the match results for the respective UVs. For aggregation purposes, however, we expand them to $|UV_{AB}|$-sized bitmaps where bits represent match state of each UV_{AB} for that field value; this expanded bitmap, which has the same size N for every field, is simply called bit vector(BV) hereafter, as illustrated in Figure 2(b). Similarly, for each field value we can store a URID representing the UR matching that particular value. Both BV and URID are illustrated in Figure 2(b) for a sample field value $F_A = 9$ on field A. URs can be essentially seen as optimally compressed versions of their respective BVs; as such, they are biunivocally related. URs are important since they ultimately discriminate actions to be taken on the packet without involving redundant information

Table 1: |UV| for real rulesets.

Set	Rules	Network (IP)			Application (ports)		
		srcIP	dstIP	SrcIP/DstIP	srcPT	dstPT	SrcPT/DstPT
ACL1	752	96	205	425	1	140	140
FW1	269	56	66	128	13	43	55
IPC1	1550	152	128	941	34	54	85

Table 2: |UR| for real rulesets.

Set	SrcIP[15:0]	SrcIP[31:16]	DstIP[15:0]	DstIP[31:16]	Prot[7:0]	SrcPT[15:0]	DstPT[15:0]
ACL1	96	5	177	78	5	1	141
FW1	55	14	67	9	6	14	44
IPC1	282	40	549	97	8	32	51

as in the case of BV. As we will see, either URs, UVs, or BVs can be used for aggregation of lookup results at each field.

To get an impression of the values of |UV| and |UR| for individual fields and aggregation results in real cases, Tables 1 and 2 show results for ACL1, FW1, and IPC1 rulesets [37]. Similar trends are observed in other synthetic rulesets generated from them. Table 1 also shows 2D UVs resulting from aggregating SrcIP/DstIP and SrcPT/DstPT, respectively, while Table 2 considers splitting SrcIP and DstIP in 16-bit chunks as it will be discussed later.

We now wonder how costly the lookup process can be for a particular field in terms of both UVs and URs; this analysis will help to define our following contributions. To this end, we consider upper bounds for the number of UVs and URs for prefix, exact, and arbitrary range lookup cases. To illustrate our analysis, in Figure 3 we consider a field of width M and 2^M possible field values. The number of UVs at a field is mostly less than N as discussed before, so we will consider a general case of |UV| UVs for the lookup case which will eventually scale to N in a multifield decomposition-based classification scheme. Each of these UVs implies a range of key values defined by either start/end bounds $[s, e]|_{s\leq e}$ or respective offset/scope $(O, S)|_{O\geq 0, S\geq 0}$. Additionally, in the following discussion we consider UR0 as the region where just the default rule (i.e., wildcard WC) is involved. Given |UV| UVs defined over a particular field, their theoretical possible MM cases are given by $2^{|\mathrm{UV}|}$. However, implementing such theoretical case would require UVs with multiple $[s, e]$ points as shown in Figure 3(a), which is not possible in practice. For continuous UVs involving a $[s, e]$ pair, the worst case would really involve |UV| prefix-based UVs as shown in the two cases of Figure 3(b), resulting in an upper bound |UR| = |UV|. For AR-based rulesets, partial overlappings can exist, so the worst case is |UR| = 2|UV| − 1 as shown in Figure 3(c). Exact-match, meanwhile, is essentially a special case of prefix-match with maximal prefix length; therefore, its worst case is |UR| = |UV| as shown in Figure 3(d). Figures 3(e) and 3(f) illustrate mid-points between worst cases 2|UV| − 1 and |UV| when GR-based rules with different overlappings are considered. In Figure 3(e) |UR| = |UV| URs are defined by GR-based rules, while Figure 3(f) shows a general case |UV| = 5, |UR| = 7(|UV| ≤ |UR| ≤ 2|UV| − 1).

Of particular interest is the *minimum number of* UVs ($|\mathrm{UV}|_{\min}$) needed to achieve the aforementioned worst cases. $|\mathrm{UV}|_{\min}$ can be defined in terms of the *offsets O between* UVs and UV *scopes* (S) present in the ruleset. To get the worst-case overlapping, $O_{\min}$ must be 1; that is, a new UR arises for each key value. In addition, there exists a scope $S_{\max} = 2^M/2$ for which this worst case arises. In Figures 3(c) and 3(d) we obtain $|\mathrm{UV}|_{\min}$ for cases where S and O are the same for all of the considered UVs. For them, it can be demonstrated that $|\mathrm{UV}|_{\min} = (2^M - S)$. Figure 3(c) considers the case $(O_{\min}, S_{\max})$ ($|\mathrm{UV}|_{\min} = 8 - 4 = 4$). Note that, if we use $S > S_{\max}$, the worst case |UR| = 2|UV| − 1 cannot be achieved without recurring to UVs where $s > e$, that is, "circular" scopes which are not present in real cases. Figure 3(d) considers the opposite case $(O_{\min}, S_{\min})$ (i.e., exact matches) ($|\mathrm{UV}|_{\min} = 8 - 1 = 7$). Figures 3(e) and 3(f), meanwhile, consider cases where the considered UVs span different Ss. In such cases, no general rule was found to apply for $|\mathrm{UV}_{\min}|$.

From the preceding analysis, we can reach some relevant conclusions. The theoretical worst case $|\mathrm{UR}| = 2^{|\mathrm{UV}|}$ achievable with a BV is not possible in practical rulesets; moreover, real cases are typically far beyond the practical worst cases presented here. Decomposition-based multi-field schemes require lookup stages to deliver MM results, since a BM in a certain field could not be the same BM at others. In this context, URs represent optimally compressed MM results as they gather just effective |UR| UV combinations from the $2^{|\mathrm{UV}|}$ space present in BV (Figure 3(a)); in this way, they hold just the essential information to take an action on the packet. For FPGA implementation, URs require less routing complexity and memory bandwidth than UVs and associated BVs and are more scalable for large number of rules. URs are the result of performing compression of redundant information present in BVs during rule updating; therefore, they require precomputation. BVs, meanwhile, require logic and/or memory to perform such compression at runtime and determine the action to be taken on the packet. Precomputation can be effectively implemented on general-purpose processors (GPPs) while keeping the hardware-software interface as fast as possible to transfer resulting URs from the GPP to FPGA-based lookup engines.

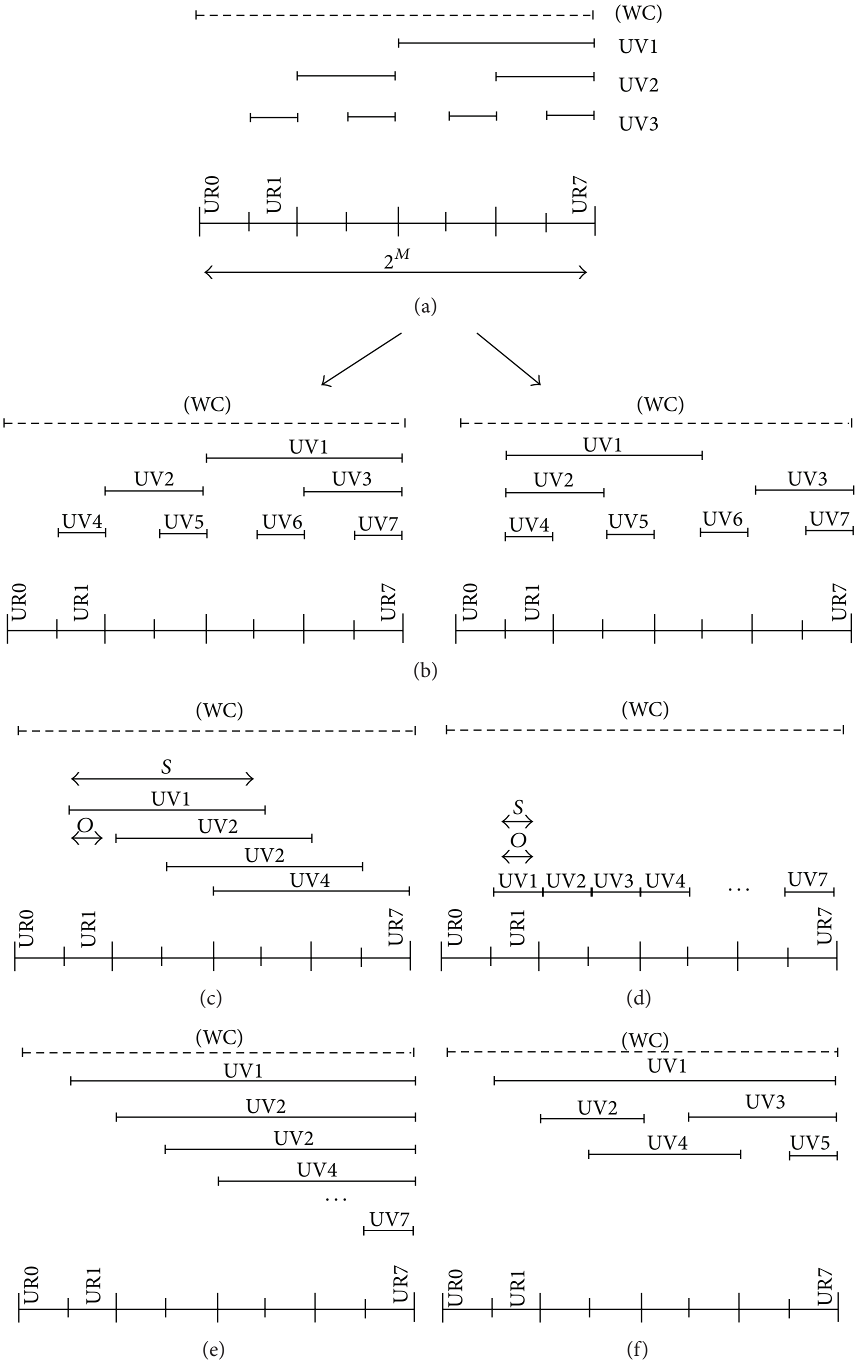


Figure 3: Worst overlappings: (a) theoretical worst-case number of overlappings ($|UR| = 2^{|UV|}$), (b) practical worst-case number of overlappings for PX-based rulesets ($|UR| = |UV|$), (c) practical worst-case for AR-based rulesets ($|UR| = 2|UV| - 1$), (d) practical worst-case for EX-based (i.e., maximum-length PX) rulesets ($|UR| = |UV|$), and (e) and (f) practical worst-cases for GR-based rulesets ($|UV| \leq |UR| \leq 2|UV| - 1$).

3.2. Comparison of Lookup Schemes. In order to effectively compare lookup schemes, we first consider the actual requirements in the context of multifield decomposition-based classification, namely,

(1) fast and simple incremental update;

(2) support of matching on exact values, prefixes, or arbitrary ranges depending on packet processing needs;

(3) appropriate interface between lookup stage and adopted multifield decomposition scheme;

(4) good scalability with respect to the ruleset size, both in length (# of rules) and in width (number of fields and bits/field);

(5) moderate precomputation complexity;

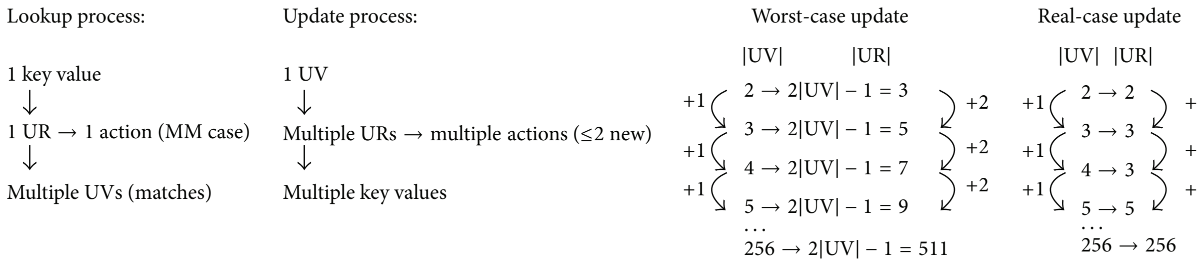


Figure 4: Multimatch (MM) processing.

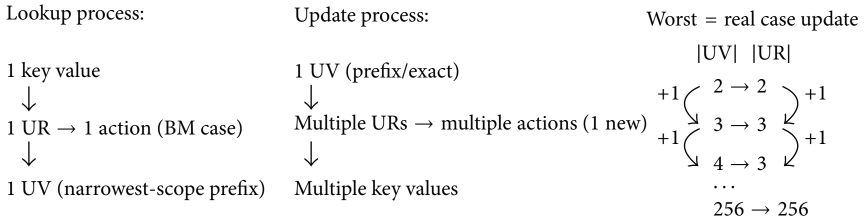


Figure 5: Best-match (BM) processing.

(6) moderate resource consumption;

(7) high throughput (i.e., line speed processing);

(8) moderate power and energy consumption.

Even if this work is about unifield lookup schemes, considering actual trends to flow-based networking schemes, we argue the need of discussing them in the real context of generic, multifield classification applications. In this context, requirements (1), (2), and (3) are naturally related. As context for our following discussion about them, in Figures 4 and 5, we summarize relevant aspects of the update and lookup processes for the MM and BM cases, respectively.

When an incremental update is performed to the lookup engine, we commonly want to add a rule which relates to specific ranges of values at each considered key field $F_k(0\cdots 2^{M_k})$. If we consider a single field, a multifield rule updating operation could involve a UV updating operation (when the new rule uses a UV not already present at that field) or no UV updating (when the new rule reuses a UV already stored for that field). Let us now consider the case when a new UV is added at a lookup stage. The new UV will in general impact multiple URs, at most 2 for MM lookup and 1 for BM lookup (we further discuss this topic later in this section). As a consequence, requirement (1) essentially means that mapping from one UV to multiple URs should be implemented efficiently.

During lookup, in contrast, we typically input one specific key value at the considered field, and the lookup scheme delivers one particular UR representing one particular action. Depending on the type of matching (i.e., BM versus MM), the complexity of determining the involved UR is substantially different. This determines requirement (2). For the case of BM, each UR maps to exactly one different UV, resulting in $|UR| = |UV|$. On the other hand, MM requires $|UR| \leq 2|UV|$ URs, which complicates the lookup process. PX matching adapts naturally for BM since the prefix with the narrowest scope is chosen and either complete or no overlap can occur between prefixes. AR matching presents total, partial, and no overlappings between UVs and BM/MM is possible. For exact match, finally, $|UR| = |UV|$ and no overlapping at all is present, so in this case the lookup complexities for BM and MM are the same and lowest ones.

In Figures 6(a), 6(b), and 6(c) we consider possible overlapping cases and how the addition of a new UV affects them. Figures 6(a) and 6(b) focus on arbitrary ranges which can overlap completely, partially, or not at all, while Figure 6(c) considers prefix ranges which can show either complete or no overlaps. Without loss of generality, we consider a simple 2-UV case and incremental updating with one new UV (i.e., UV3) for offsets causing all possible overlapping cases. Remaining overlapping cases are simply mirrored versions of the considered ones and add no new URs. Updates involving more initial UVs cause the same general overlapping cases, simply involving more UVs as suggested in worst and real updating cases of Figures 4 and 5. As anticipated in the previous paragraph, the addition of a new UV to AR rulesets can add at most 2 URs, while the same addition in prefix-based rulesets adds 1 new UR. As discussed later in this paper, we can take effective advantage of this fact to reduce updating overhead, which is one of the main impairments to the adoption of UR-based schemes.

Requirement (3) is naturally associated with the needs in decomposition-based classification which separates lookup and aggregation as two independent stages (as discussed, the other prevalent classification scheme, i.e., decision-tree based, does not differentiate both stages and is not as well suited for hardware implementation as aggregation-based classification is). It has not been considered in previous approaches to the best of our knowledge, even if each proposed lookup scheme implicitly has an optimal output interface. Lookup schemes, on the one side, have been commonly treated as self-contained, mostly longest-prefix

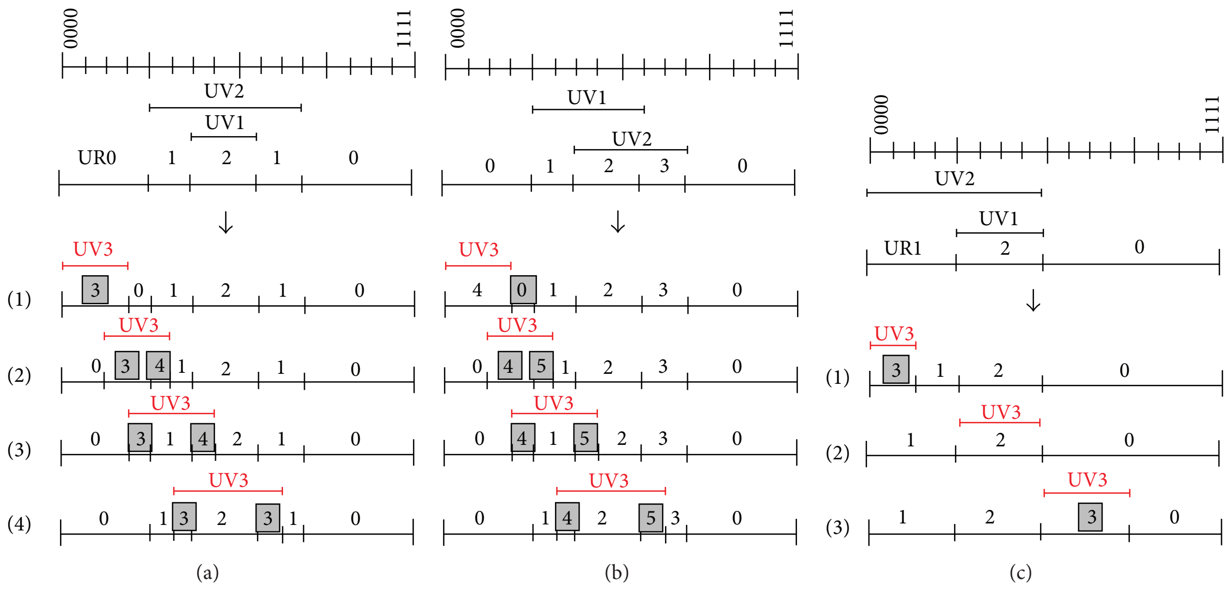


FIGURE 6: Effect of adding a new UV: (a) and (b) arbitrary range cases and (c) prefix case.

matching (i.e., BM) engines, while classification schemes, on the other side, are commonly proposed with a predefined input format in mind (i.e., MM format), and the lookup stage is somehow adapted to it or is not considered at all. As we will discuss, there exist both UR-based and UV-based aggregation schemes which ideally fit the counterpart format at the lookup output. Lookup schemes, meanwhile, can be more or less suited for each output format.

Requirement (4) is quite dependent on the adopted data formats for both lookup and aggregation stages. Requirement (5), meanwhile, relates to (1), (4), (6), and (7). In general terms, the higher the precomputation complexity, the higher the updating complexity, the lower the required memory bandwidth of the lookup engine and the higher the lookup throughput. These facts can be clearly appreciated in UR-based schemes, which are logic consequences of the shift of computation from run-time to update time. By storing preprocessed results in memory, logic resources are saved, memory bandwidth is reduced, and scalability with respect to the ruleset size is enhanced; however, updates require precomputation which must comply with required updating rates.

Power and energy consumption (8), meanwhile, have become important metrics in the last years and should be considered while designing lookup and classification schemes. In general terms, for the considered pipelined schemes, energy consumption involves both total latency and power consumed per pipeline stage [38]; both are also strongly related with (6) and (7); to this respect it can be reduced by selective clock gating.

As starting point for our study on lookup schemes and their suitability for aggregation-based classification, we survey references to lookup stage made in related work on classification; even if no implementation details of such lookup schemes are given in most of them. Works containing such references are Lucent BV [16], Distributed Crossproducting of Field Labels (DCFL) [17], Extended DCFL (DCFLE) [12], StrideBV [33], SCBV [36], and Recursive Flow Classification (RFC) [13]. In addition, [39] provides some interesting insights even if the analysis focuses exclusively on lookup and does not relate it to classification.

For exact match, the most convenient solution seems to be hashing according to [17, 39]. For prefix matching, Binary search on prefix lengths and tree bitmap are proposed in [17], while TCAMs are mentioned in [12]. For arbitrary range match, [17] proposes TCAMs, Balanced Interval Tree, and Fat Inverted Segment Tree; [16] proposes Binary Search; [12] implements ETCAM; and [33, 36] propose ETCAM emulation on FPGAs. Reference [13], meanwhile, adopts memory indexing for all match cases. Even if multiple options are mentioned, there is no clear comparison of their resource consumption versus performance trade-offs. Moreover, the lookup-aggregation interface is not considered. Finally, FPGA implementations are either not considered or completely local to each approach and, therefore, not directly comparable. To fill this lack, in this section we introduce a high-level analysis of main lookup options. We begin by considering CAMs and their multiple emulation and optimization options on FPGAs. We then compare with binary decision trees which relax concurrency in favor of reducing memory bandwidth. As it will be shown, all of the existing options can be studied on a common base from which the most convenient one can be identified depending on the intended application. From this analysis, we propose a new, general taxonomy of lookup schemes for FPGAs.

As mentioned in Section 1, CAMs and their ternary variant TCAMs are the concurrent counterpart of linear search. Both approaches implement *Exhaustive search* with optimal $O(M \cdot N)$ memory efficiency. However, they are not scalable to large N, due to long search time in case of linear search and to high power/area consumption in case of TCAMs. In addition, TCAMs suffer from the *range expansion* problem: arbitrary

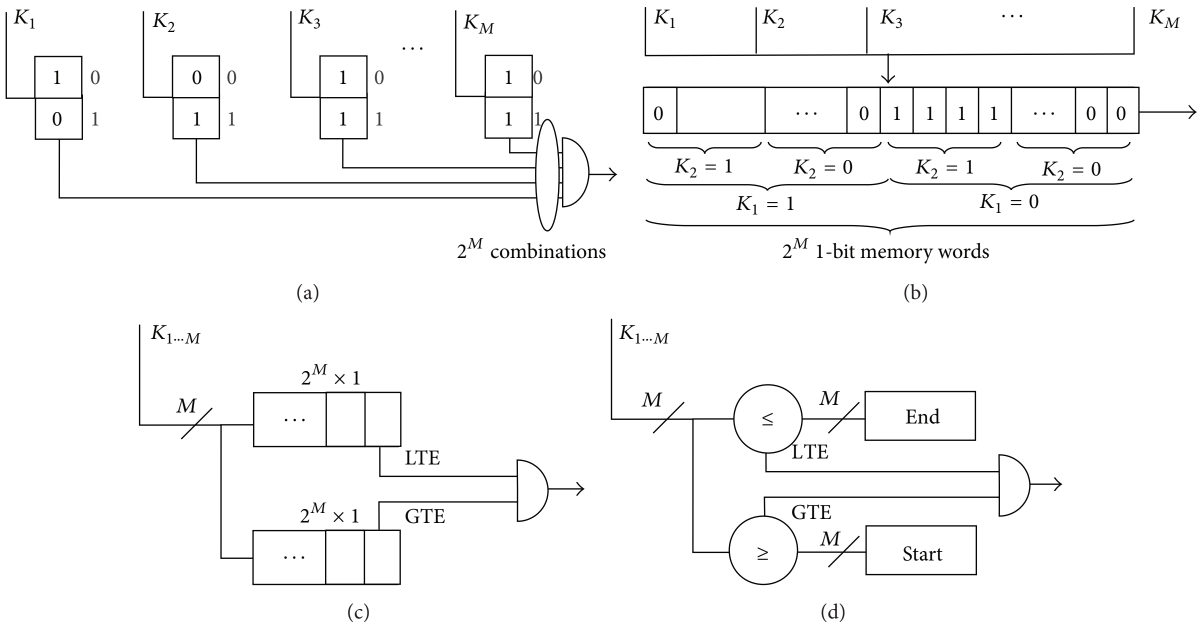


Figure 7: 1-rule lookup: (a) register-based TCAM emulation (no address expansion), (b) memory-based interval (full address expansion), (c) memory-based bounds, and (d) logic-based bounds.

ranges (see Section 1) must be mapped into multiple prefixes which can expand to M entries for keys of width M [21]. Despite these drawbacks, TCAMs are widely used in modern networking equipment where performance guarantees are a must.

The problem of arbitrary range implementation in TCAMs, together with the increasing presence of this kind of matching in rulesets, have led to intensive research on the topic. In FPGA-based lookup, the arbitrary range matching problem for one UV can be approached in two ways, namely, by storing precomputed range match results or by storing range bounds and matching against them by combinational comparators. We call the former memory-based as it stores match results in advance in FPGA memory and the latter logic-based as it computes matches by combinational logic in FPGA. Memory-based schemes are faster by requiring modest $O(1)$ memory bandwidth for one rule but may require much more memory than that required by the original lookup table; that is, the memory utilization factor is $u < 1$. Logic-based matching, on the other hand, resembles native CAM features, were minimal storage is required at cost of high $O(2M)$ memory bandwidth and $O(2M)$ routing complexity for one rule. Focusing on the memory-based case, two variants arise; that is, precomputed match results can be stored considering both start and end bounds together (i.e., interval) or in two independent memories for upper and lower bounds, respectively. In the former, one 2^M-deep memory is enough, while the latter requires two of such memories.

From the technical perspective, three main resources are available in FPGAs for CAM emulation, namely, synchronous registers (flip-flops), Block RAM (BRAM) (even if distributed *RAM* can also be used in combination with BRAMs, this feature is not equally available on all FPGA providers, so it is not included in our analysis), and combinational logic implemented in flash memory-based LUTs. Figure 7 presents emulation options for one rule. In Figure 7(a) a TCAM emulation scheme using registers is shown which exactly mimics a native TCAM. As such, it consumes minimal memory but suffers range expansion (i.e., multiple rows would be needed still achieving just one rule match). One way to alleviate range expansion, at cost of addressing expansion, consists of TCAM emulation through RAM memory as shown in Figure 7(b), which is the memory-based interval case (i.e., it considers the whole $[s, e]$ interval of a rule in a single memory). This approach was early suggested by FPGA providers [40, 41] and extensively studied in our previous work [10] as well as more recent proposals [35].

ETCAM [23] is another effective scheme to mitigate range expansion of TCAMs in ASICs. It stores two M-bit words in ASIC technology for each rule. Both words hold the start and end bounds of each range, respectively, and two M-bit magnitude comparators are needed for each rule. This technique, called explicit range match (ERM) throughout this paper, was ported to FPGAs in [12, 36]. Reference [12] precomputes comparison results for each bound separately and stores them in LUTs used as distributed RAM memory (DistRAM). This approach is shown in Figure 7(c) and is called memory-based bounds case (i.e., it considers both bounds $[s, 2^M - 1]$ and $[0, e]$ in independent memories). Reference [36], meanwhile, stores bounds separately in registers and maps magnitude comparators in general-purpose LUTs (i.e., implements the logic-based case) requiring fully-parallel access to all stored bounds as shown in Figure 7(d). We note that such a high memory bandwidth can only be achieved by using general-purpose registers with high routing complexity.

As already mentioned, lookup schemes can be UV-based or UR-based. Both approaches are indistinguishable for one rule since just one UV defining one UR exists in this case;

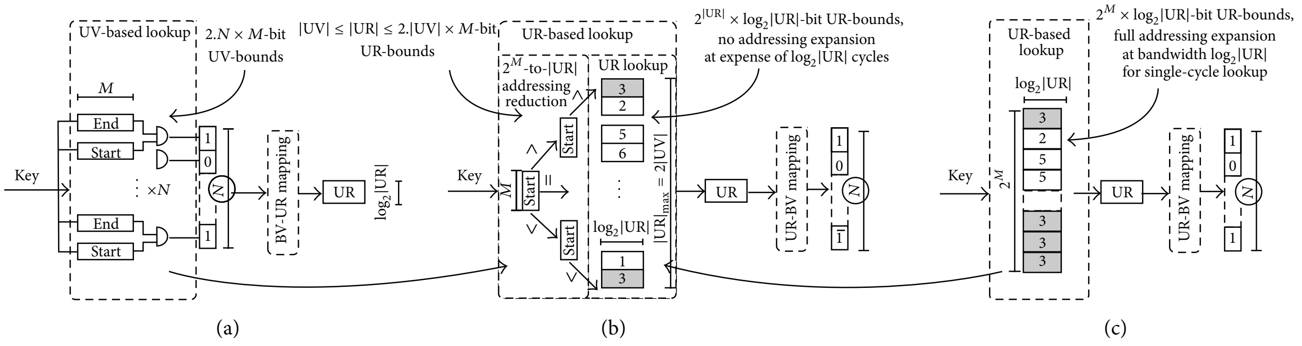


Figure 8: N-rule lookup: (a) UV-based by concurrent matching of rule bounds, (b) UR-based by reducing memory expansion through a binary tree, and (c) UR-based by full memory expansion.

however, for multiple rules they differ depending on rule overlapping. Multiple UVs can be typically matched by a packet, so UV-based schemes must be highly concurrent architectures. In Figure 8(a) we show a typical UV-based scheme which is an extension of Figure 7(d). On the other hand, just one UR can be matched by a packet, so UR-based schemes can achieve high performance with modestly pipelined architectures. Figures 8(b) and 8(c) show two UR-based lookup schemes. In Figure 8(c) the scheme of Figure 7(b) is extended for $|UV|$ UVs which combine into $|UR|$ URs. Figure 8(b), meanwhile, shows a generic binary search scheme where the tree can be seen as multiple-stage address space compressor between the key and the memory of Figure 8(c); in this way memory address space is considerably reduced. In fact, the address reduction stage of Figure 8(b) arranges $2|UV|[s,e]$ UV bounds into $|UV| \leq |UR| < 2|UV|[s,e]$ UR bounds and then uses them as branching functions in $\log_2|UR|$ stages. After that, it implements the scheme of Figure 8(c) with reduced $|UR|$ address space thanks to the compression achieved by the decision tree. From the previous analysis, we can consider the three schemes as solutions to the same problem with different trade-offs regarding storage, time, and preprocessing costs. In addition, each lookup scheme offers a natural associated interface to the aggregation stage for multifield classification. The natural lookup-aggregation interface is BV for Figure 8(a), while it is a URID for Figures 8(b) and 8(c). If these lookup engines are interfaced to the opposite aggregation case, additional UV-UR and UR-UV mapping stages are required as also shown in Figure 8.

Examples of UV-based lookup are [12, 36], while the UR-based lookup is used in [16] (which in fact combines UR-based lookup and UV-based aggregation) and [13]. Concurrent architectures such as native TCAMs and ETCAMs are naturally suited to UV-based lookup of multiple rules, while indexing (emulated TCAM) and decision tree-based ones are by definition UR-based lookup schemes. UV-based schemes scale linearly with the number of rules and do not take advantage of rule overlapping patterns. On the contrary, UR-based schemes are more scalable than UV-based ones for large N and moderate overlapping. UV-based schemes, meanwhile, require no precomputing of UR labels which can be convenient for moderate N or complex overlapping patterns.

4. Architecture

In order to validate and evaluate our analysis, we implemented FPGA designs for the following cases:

(i) memory indexing (i.e., IND);

(ii) binary search tree (i.e., BS);

(iii) explicit range match (i.e., ERM).

For each case, we consider their performance from the match (i.e., exact, prefix, and arbitrary range) and aggregation (i.e., UR-based and UV-based) points of view.

4.1. Memory Indexing. Memory indexing is a convenient choice when flexibility is required regarding the aggregation stage to be used; that is, even if it is essentially a UR-based lookup scheme as we will see, it can also output UV-based results without significant architectural changes (i.e., achieving the UR-to-UV mapping stage of Figure 8(c)). Despite its simplicity, it enables a number of mappings between keys and match results. At each memory address we can store either a N-bit bitmap representing UVs (i.e., a bit vector or BV) or a precomputed $\log_2|UR|$-bit label representing a UR (i.e., a URID). In this way, it can be interfaced to either UR-based or UV-based aggregation stages. Figures 9(a) and 9(b) show both alternatives, respectively, for the example range pattern of Figure 10(c). In addition, multiple *layering* schemes have been explored to date [14, 15, 42] which essentially relax minimum-width UR labels without reaching the width of BVs. They reduce memory bandwidth required by BV by adding precomputation of match results. At the same time, they reduce precomputation requirements of URs by dividing the matching space in layers. These layering schemes, however, require advanced algorithms for finding independent sets of UVs which are out of scope of this work.

Finally, a combination of the UR-mapping scheme of Figure 9(a) and the UV-mapping schemes of Figure 9(b) can be used in a two-stage architecture with two memory modules, *reg_mem* mapping key values to URs followed by *bv_mem*

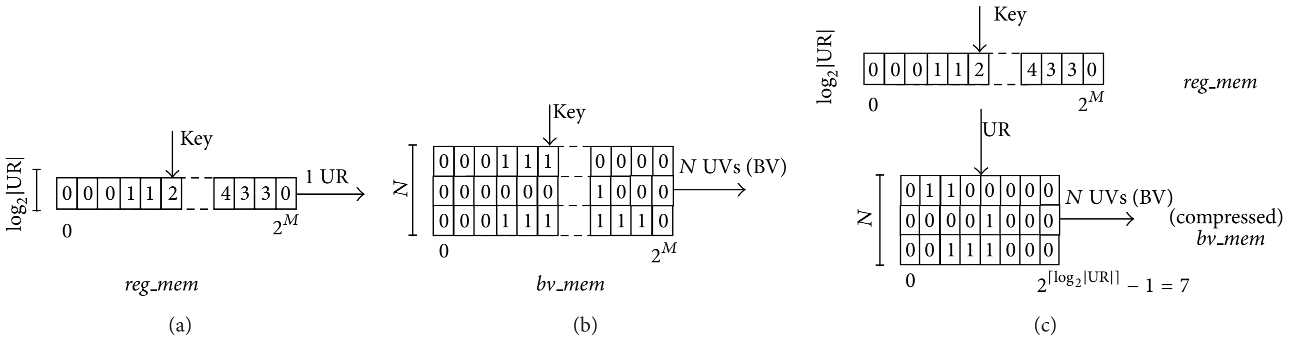


FIGURE 9: Memory indexing: (a) UR output, (b) BV output, and (c) addressing space reduction.

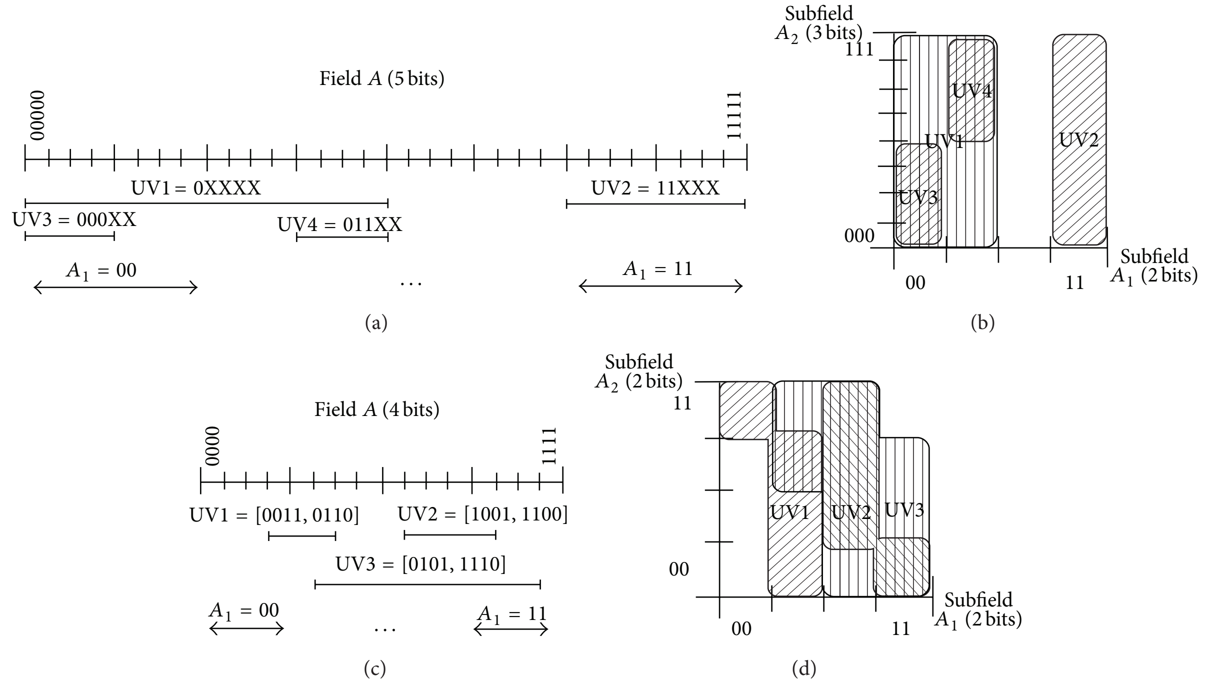


FIGURE 10: Effect of stitching: (a) prefix-based lookup example, (b) prefix stitching, (c) AR-based lookup, and (d) AR stitching.

mapping URs to UVs. This option, also sketched in Figure 9(c), allows interfacing to UV-based aggregation stages while reducing storage from $2^M \cdot N$ to $2^M \cdot \log_2|\mathrm{UR}| + |\mathrm{UR}| \cdot N$. For FPGA implementation, however, this is not always true since memory blocks limit smallest attainable form factors; as a consequence memory and address widths cannot always be optimized to $O(\log_2|\mathrm{UR}|)$.

Memory indexing can be viewed as the end case of a single-stage binary search tree. As such, it is able to perform lookup in a single cycle; however, as mentioned in [10], this approach faces the problem of addressing expansion. In general terms, for a key of width M, 2^M memory words are required to hold match results for each of the 2^M possible key values. This scheme can match any general $S = 2^W$ ($W \leq M$) range with one bit/rule of memory width. We can alleviate addressing expansion by exploiting horizontal or vertical slicing techniques proposed in [43]. In the first case, memory address port is horizontally sliced in $\lceil M/m \rceil$ stitched "chunks" and associated with independent memory blocks, each having 2^m addressing space. This approach can significantly reduce memory consumption and take advantage of optimal memory form factors in FPGAs. However, splitting a key field A into two chunks introduces two different, orthogonal subdimensions A_1 and A_2. This problem is clearly illustrated in Figures 10(a) and 10(b) for a 5-bit prefix ruleset split into 2-bit and 3-bit subfields, and Figures 10(c) and 10(d) for a 4-bit AR ruleset split into two 2-bit sub-fields. As shown, each slice performs lookup independently and has no sensibility of lookup results at other slices. For example, given a key = 00110 for ruleset of Figure 10(a), chunk A_1 matches 00 but is unable to sense if either 0XXXX (UV1) or 000XX (UV3) effectively matches the key; this will depend on result from chunk A_2. As a consequence, match results from these subdimensions must be properly aggregated prior to leaving the lookup stage. This aggregation is similar to that performed in decomposition-based classification (i.e., interfield) but in

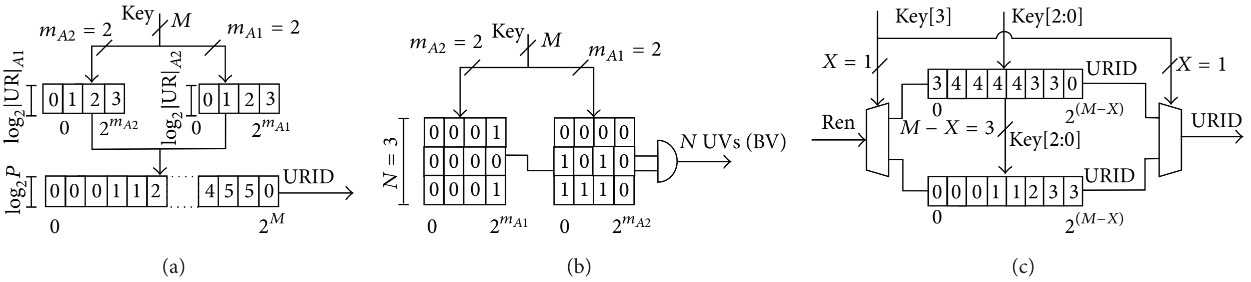


Figure 11: Stitching options: (a) UR-based horizontal stitching, (b) UV-based horizontal stitching, and (c) vertical stitching.

this case is internal to the lookup stage (i.e., intrafield). For the latter case, we note an essential difference between patterns for PX (Figure 10(b)) and AR (Figure 10(d)) cases. In Figure 10(b), the range of subfield A_2 corresponding to each value of subfield A_1 is the same; for example, $F_{A_2} = [000, 111]$ for $F_{A_1} = [000, 011]$ in UV1; this rectangle-shaped pattern holds for all considered PX rules. In Figure 10(d), meanwhile, we see that ranges on A_2 can be different along the range for A_1; for example, in UV3 $F_{A_2} = [10, 11]$ for $F_{A_1} = 01$, $F_{A_2} = [00, 11]$ for $F_{A_1} = 10$, and $F_{A_2} = [00, 10]$ for $F_{A_1} = 11$. In multifield classification, patterns can only be rectangles such as those of Figure 2(b); that is, the AR case poses *new geometrical patterns* for intrafield aggregation. As we will discuss shortly, this case represents an essential problem for stitching.

In Figure 11 we show aggregation architectures for stitching and illustrate results for the ruleset case of Figures 10(c) and 10(d). Two intrafield aggregation techniques can be applied:

(1) obtaining URs at each chunk and applying an UR-based aggregation approach similar to that of RFC [13] to the lookup case (Figure 11(a)): a new UR is obtained as output of the stitched lookup scheme;

(2) similar to aggregation scheme adopted in [16] for classification, a BV can be returned at each chunk in the lookup stage in order to aggregate them through simple ANDing (Figure 11(b)). In this case, an approach similar to that of Figure 9(c) can also be applied at each subfield.

Both schemes can match general ranges (i.e., either exact, prefix, or arbitrary ranges), even considering ranges spanning more than one chunk (i.e., $W \geq m$). For the second case, however, more than one bit per rule must be propagated between chunks.

A second technique for reducing the address space, that is, vertical slicing (Figure 11(c)), replaces some of the internal multiplexing of BRAM with external logic blocks. For this, control signals (i.e., rd/wr enable) are deMUXed and fed to multiple memory blocks, while output data ports are MUXed from such blocks; X bits from address port are used for MUX/deMUX while remaining $M - X$ bits are fed as a common address bus to all involved memory blocks. In this way, shallower and wider memory form factors can be used, which mostly save memory by relaxing address expansion. Even more important, since the lookup result will now reside exclusively in one of the memory slices, significant power can be saved by selectively enabling the involved slice. For the same reason, aggregation of results is not required but just MUXing of slice results. This architecture is mentioned for completeness, even though it is not further explored in our work.

Which of the considered memory indexing schemes is best suited for FPGAs is ultimately dictated by resources available in such devices, that is, number of BRAM modules and possible form factors. Altera FPGA devices include M4K/M9K/M20K general-purpose BRAMs depending on family, which can be configured in different $2^m \times x$ *modes*. M20K BRAMs included in Altera Stratix V FPGAs, for example, can be configured in modes ranging from 20 K×1 to 512×40. In addition, selected Look-Up Tables (LUTs) can be used as 640-bit simple dual-port MLAB memory in memory modes 640 × 10 to 320 × 20, useful for implementing small, shallow, and wide blocks. They provide enough flexibility to implement our proposed variants. Let us consider horizontally sliced memory indexing of Figures 11(a) and 11(b). The respective architectures for FPGA are illustrated in Figure 12 for a case of key width $M = 18$, number of rules $N = 80$, $|UR| = N = 80$ (for AR case, $|UR| < 2N = 160$ as already discussed) and BRAM mode 512 × 40. For this case, the key is divided into two 9-bit chunks and fed to respective 9-bit address ports. In Figure 12(a), the implementation of case in Figure 11(a) is shown along with the used Processing Element (PE). In Figures 12(b) and 12(c), meanwhile, the lookup stage outputs a bit vector (BV) representing multiple matched UVs, corresponding to the scheme of Figure 11(b). In this case, BVs can be directly stored at *bv_mem* (Figure 12(c)) or an additional UR mapping memory *reg_mem* can be included at each PE (Figure 12(b)). For proper scalability with both key width and ruleset size, these architectures are pipelined both horizontally and vertically, resulting in systolic schemes. For updating purposes in Figures 12(a) and 12(b), we need to consider the effects of the incoming (single) UV on (multiple) URs as discussed in Section 3, so precomputation is needed. Such precomputation is commonly implemented externally on a General Purpose Processor- (GPP-) based platform. The selected BRAM form factor depends on the intended application; in general terms, deep-and-narrow factors can be convenient for schemes providing URs such as *reg_mem*, while shallow-and-wide ones are convenient for BV outputs which are intrinsically wide with reduced addressing space such as *bv_mem*.

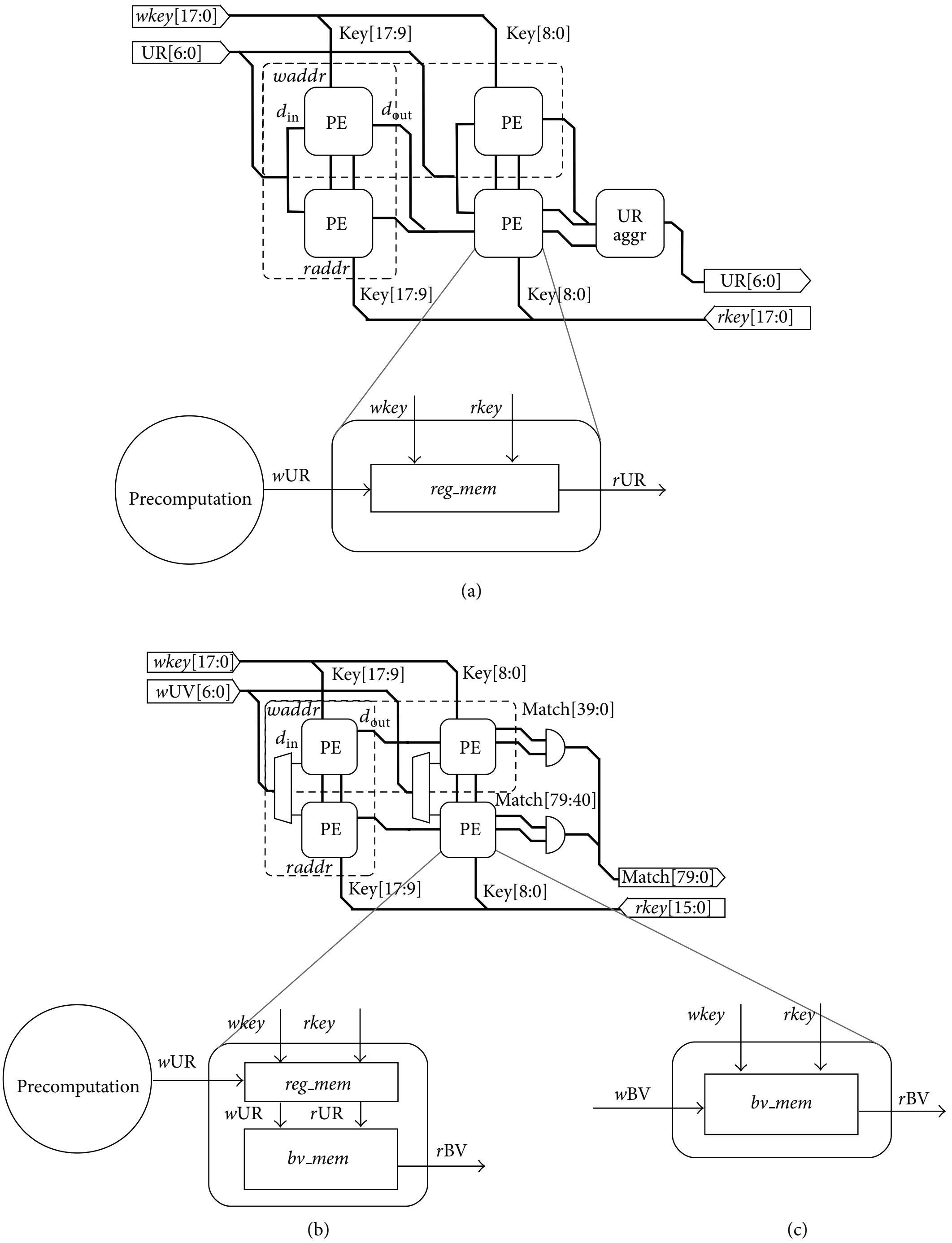
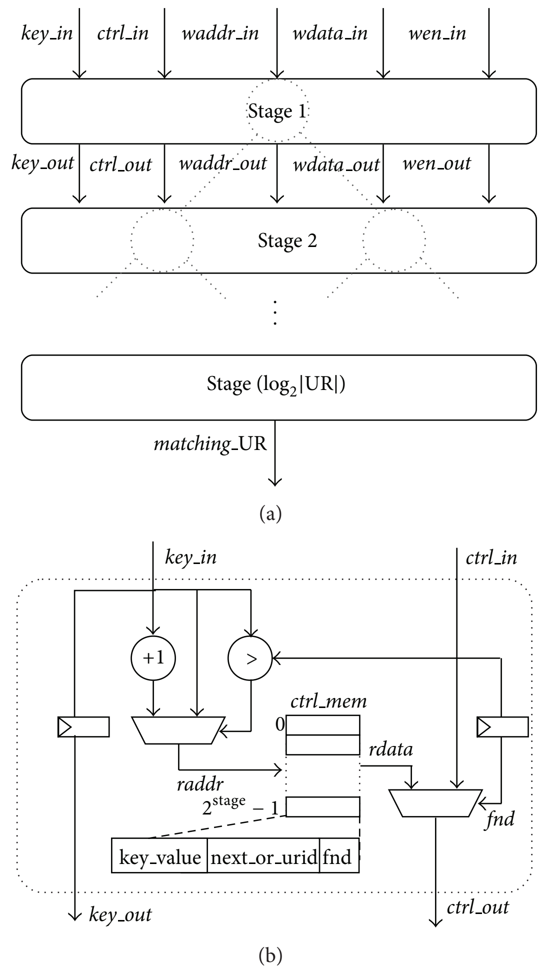


Figure 12: Stitching architecture: (a) lookup for UR-based aggregation and (b) and (c) lookup for UV-based aggregation.

BRAMs in FPGAs can be configured as either single-, simple dual-, or true dual-ported. In true dual port mode, particularly, any combination of independent read- (r-) and write- (w-) operations can be performed at the same time on both ports. This feature is extensively used in related work for reading both BRAM ports with keys from two successive packets, effectively doubling lookup rate with respect to clock rate. However, two aspects must be considered in this case. On the one hand, true dual-port mode limits form factors so that memory utilization is worse than that of simple dual port or single port modes [44]; this translates into more BRAM used for the same ruleset. For example, BRAMs in the architecture of Figure 12 would be limited to form factors 1 K × 20–20 K × 1 instead of 512 × 4–20 K × 1, so $\lceil M/m \rceil \times \lceil N/n \rceil = \lceil 18/10 \rceil \times \lceil 80/20 \rceil = 2 \times 4 = 8$ instead of $\lceil 18/9 \rceil \times \lceil 80/40 \rceil = 2 \times 2 = 4$ blocks would be needed. Of course this is a particular case, but it serves as a clear example of the problem. On the other hand, using both ports in read mode to double lookup rate precludes in-line updating of the architecture since both ports are already in use. In this case, the lookup pipeline would get stuck during updating. In addition, DistRAM is commonly used in related work to improve memory utilization for small blocks; however, DistRAM does not commonly support true dual-port mode, so doubled lookup rate cannot be effectively obtained in designs using mixed block/distributed memory such as [30] or [35]. Considering these facts, and for sake of generality, we use simple dual-port mode BRAM in this work, which enables one read-only port and one write-only port at BRAMs.

For IND schemes such as those of Figure 12, matching is carried out in a single cycle by loading RAM words addressed by key chunks *rkey* on each column of PEs and aggregating the so-defined chunk lookup results. Updating of rules (i.e., adding a new one or erasing an existing one) could require multiple memory accesses. For example, storing the prefix 0x000X in Figure 12 would require to update $2^4 = 16$ words of memory. In the updating circuit of Figure 12, 18-bit data (*wkey*) are split in two and fed to BRAM *waddress* ports for every key value in the scope of a new UV, while this UV is decoded in rows and fed into BRAM *din* ports. Columns of BRAMs are updated concurrently, so updating of the whole ruleset takes at most 2^m clock cycles.

4.2. Binary Search Tree. For the evaluation of decision tree case on FPGA, we adapted the architecture presented in [45] for the single-field case. The original architecture considers multiple key fields at the same time which complicates the design; we are instead interested in just one field for subsequent aggregation. We implemented a generic version of this scheme for comparison with IND and ERM as well as validation of our performance estimates; for more specific implementation details as well as extensive optimizations for better scalability refer to [30, 45]. In Figure 13(a) we show a general decision tree implemented as a hardware pipeline, where branching control at each tree stage is mapped by a memory block *ctrl_mem* as illustrated in Figure 13(b). For stage i, this memory block consists of 2^i words. Each of two words map both branches of a node at that tree stage. Port key_{in} is the value of the incoming key and is propagated through the tree pipeline. Port *ctrl_in*, meanwhile, holds (1) values *key_value* to be compared against the key at each node, (2) branching information *next_or_urid*, and (3) the flag *fnd*, which tells if a match has been found for the incoming key. In general, any node should be able to store both *next_node* and *URID* values for the MM case, since multiple matches can occur while traversing the tree. In our case, since we store UR-bounds instead of UV-bounds, a node needs to store either *next_node* or *URID* values. In this way, just one field *next_or_urid* is stored at each node in the tree. It holds either the offset at the next stage memory block corresponding to the next decision node or the URID in the case that a leaf node has been reached. For dynamic updating of the ruleset, the second port of *ctrl_mem* is configured as write-only, while signals *wdata_in*/*wdata_out*, *waddr_in*/*waddr_out*, and *wen_in*/*wen_out* propagate through the pipeline to write the proper value at the proper node. Data consistency is guaranteed since the key arriving just after updating is initiated is processed according to the updated ruleset as it propagates through pipeline (i.e., tree) stages. It is worth noting that signals *key_in, waddr_in*, *wdata_in*, *and wen_in* must be registered at each stage to guarantee consistency with signals *ctrl_in*/*ctrl_out* which have one cycle of delay introduced by memory read.



(a)

(b)

Figure 13: Binary decision tree: (a) tree architecture and (b) node architecture (1 node/stage).

One of the main problems of tree-based architectures is that memory consumption can vary drastically for different stages, specially for large rulesets. This fact leads to inefficient use of BRAM at early stages and BRAM exhaustion at last stages for very large rulesets. Several optimizations have been proposed [30] to take advantage of various memory types (i.e., DistRAM, Block RAM, and external DRAM) according to requirements of the particular stage in the tree. Such optimizations, even if effective, are highly dependant on the considered ruleset, which is beyond the scope of this work. For our present purposes, we let the synthesis tool choose the most convenient memory for a balanced area/speed tradeoff, which results in BRAM in most of the cases. During our experiments, we found the magnitude comparator to be the main limitation of speed for wide keys; this problem can be mitigated by implementing a combination of vertical and horizontal pipelining to keep comparators narrow [36].

4.3. Explicit Range Matching (ERM). ERM naturally adopts UV-based lookup by matching the key against each UV specification individually. As such, it intrinsically scales with N and requires an additional mapping stage if interfacing with a UR-based aggregation stage is required. This technique, unlike previous ones, implements completely parallel, bit-level matching of key fields with optimal storage requirements. As a drawback, it requires additional logic for implementing magnitude comparison and concurrent access to $2MN$ registers.

Our first implemented architecture was based on that of [36]. In Figure 14(a) we show the general 2D pipeline architecture, where match results are propagated horizontally and key values vertically. The pipelined priority encoders used in [36] are represented here as general BV-to-UR mapping modules, even if the concrete implementation for the MM case can be very difficult. In Figure 14(b) we show the PE architecture used in [36], which was adopted at the beginning of our tests; this architecture propagates just one bit along each pipeline row. Reference [36] claims supporting the AR case with this implementation; as we will demonstrate, however, just the PX case can be supported with it.

In Figures 15(a) and 15(b) we consider two PX and AR rulesets, respectively, which are sliced into 2 chunks. In general, subfields $A_1, A_2, \ldots, A_{M/m}$ group bits in decreasing weight order (i.e., MSbits to LSbits). In Figures 15(a) and 15(b) $M = 4$ and $m = 2$, so A_1 groups the two most significant bits (MSbits), while $A_{M/m} = A_2$ groups the two least significant bits (LSbits). PX presents no special issues and can be implemented by propagating 1 bit/rule between chunks both in IND and in ERM. ARs, however, require more than one bit to be propagated. In Figure 15(b) we consider two ARs UV1 ($s_1 < e_1$, $s_2 < e_2$) and UV2 ($s_1 < e_1$, $s_2 > e_2$). In Figure 15(c), for example, we consider UV1 of Figure 15(b). In Figure 15(d) we show that, if we consider subfield ranges $F_{A_1} = [1, 3]$, $F_{A_2} = [2, 2]$ and take their intersection (AND) in the two-dimensional space, we get an area involving the discontinuous field range $F_A = [6] \cup F_A = [10] \cup F_A = [14]$ which is clearly not the original range $F_A = [6; 14]$ defined for UV1 on field A. To solve this problem, we observe that since A_1 groups MSBits, the ranges in A involving $s_1 < A_1 < e_1$ span whole multiples of the remaining chunks $0 \le A_i \le 2^m$ ($i = 2 \cdots \lceil M/m \rceil$) (i.e., the shaded rectangle in Figure 15(e)). On this basis, we can implement AR by propagating match results for $A_i = s_i$, $A_i = e_i$, and $s_i < A_i < e_i$ separately, that is, 3 bits per rule. In the case of ERM, the results $s_i < A_i$ and $A_i < e_i$ must be also differentiated for cascading magnitude comparators, so 4 bits are propagated per rule in this case. In Figure 15(f) we show UV2 where $s_2 > e_2$; the (wrong) results of implementing $(s_1 \le A_1 \le e_1)\, AND\, (s_1 \le A_1 \le e_1)$ are shown in Figure 15(g) and the (right) results of the corrected architecture for this case are shown in Figure 15(h). The architecture of the new PE supporting ARs is shown in Figure 14(c).

In order to further validate our observations, in Figures 15(i) and 15(j) we add two different ranges $A_3 = [00, 10]$ and $A_3 = [10, 11]$ of a third chunk to the pattern of Figure 15(h). As shown, the shaded block now spans integer multiples of *both* $A_2 = [00, 11]$ and $A_3 = [00, 11]$ for $s_1 < A_1 < e_1 = [01, 10]$, while results for $A_1 = s_1 = 00$ and $A_1 = e_1 = 11$ must be propagated separately. The modified ERM architecture supporting one AR rule is shown in Figure 16(a), where each column is essentially the PE of Figure 14(c). In Figure 16(b), meanwhile, we show the equivalent IND scheme

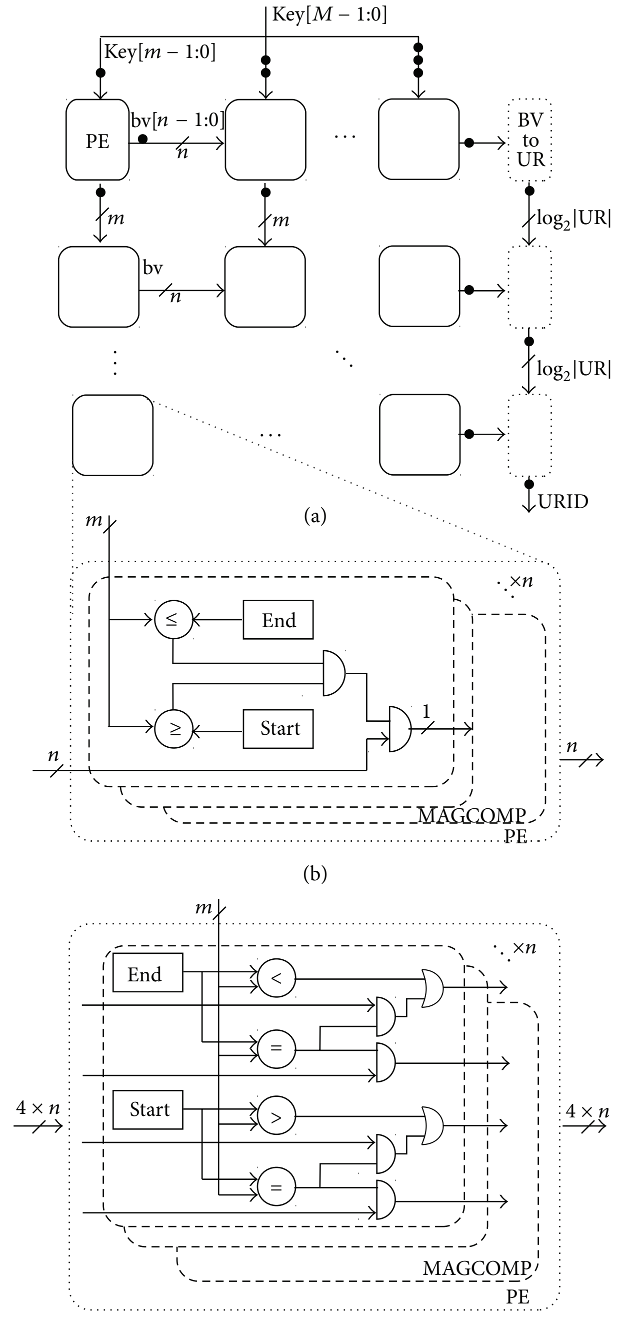


Figure 14: Explicit range match: (a) 2D pipeline of PEs, (b) previous node architecture (PX support), and (c) proposed node architecture (AR support).

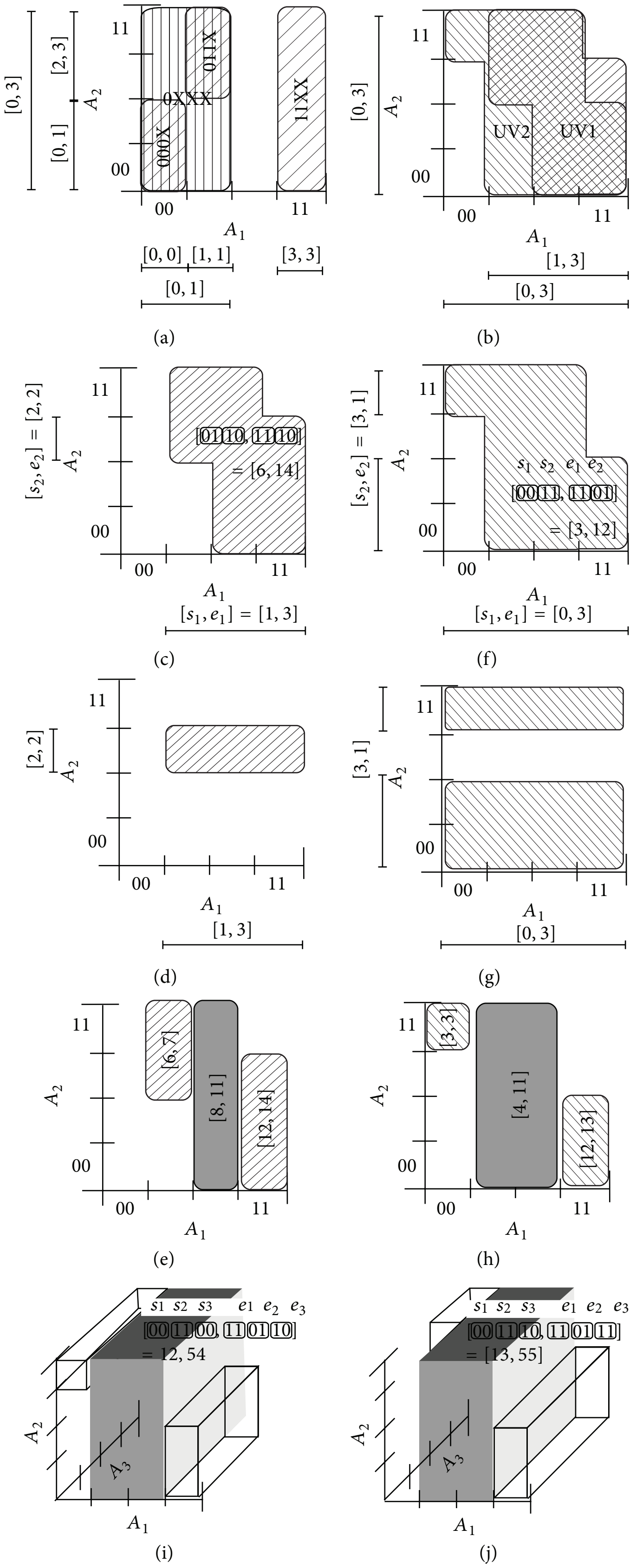


Figure 15: Stitched ERM: (a) PX ruleset, (b) AR ruleset, (c), (d), and (e) AR where $s_2 < e_2$, (f), (g), and (h) AR where $s_2 > e_2$, and (i) and (j) extension of (h) to 3 subfields.

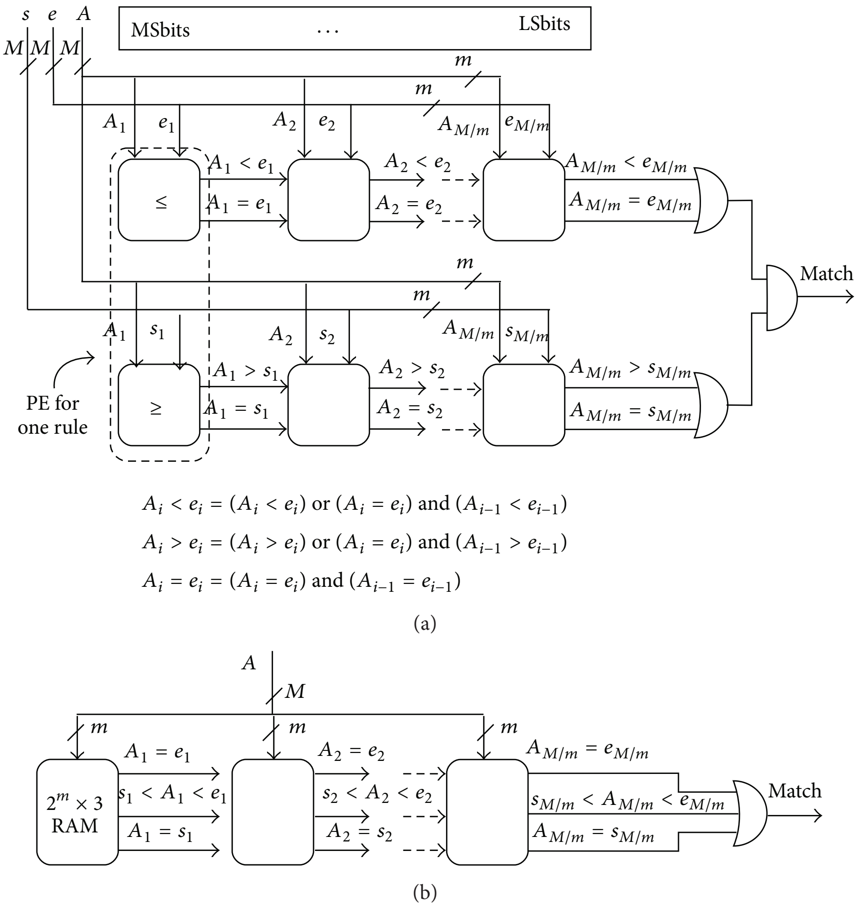


Figure 16: Single-rule lookup: (a) stitched magnitude comparators for ERM and (b) equivalent implementation through stitched IND.

(UV output) supporting ARs where 3 bits are propagated at each chunk. It is worth noting that, by using this scheme, each memory block of IND does not depend on addressing expansion to support ARs. At the end of each row, results of the last chunk are combined through logic to get match result for the considered rule. Both schemes represent a complete lookup row for $n = 1$ when applied to a complete $M \times N$ AR lookup scheme such as that of Figure 14. The net effect for ERM is that $4\lceil M/m \rceil$ instead of $\lceil M/m \rceil$ registers are needed for each rule (plus additional logic in the new PE); as a consequence a PE that supports n PX rules now requires more resources to support the same n AR rules. The same applies to IND lookup with BV output, where wider memory is required to support the same n.

4.4. Ruleset Population and Incremental Update. The ruleset in lookup and classification engines is affected at two occasions, that is, during ruleset population with initial rules and then during service when rules are incrementally added or erased. We refer to the former of them as ruleset population while the latter is called incremental (or dynamic) updating. Ruleset population has less strict requirements as long as the time required is not excessive. Incremental updating, meanwhile, has become an important factor in current applications since it can be required frequently on a running engine. The complexity of both of them essentially depends on the adopted lookup process, namely, UR-based or UV-based. As discussed in Section 3, updating normally consists on the addition or removal of a particular UV; as a consequence, UV-based schemes tend to allow simpler updating than UR-based ones. This is essentially because UV-based schemes do not have to compute new URs introduced by the new UV. In this section, precomputing time and storage requirements are considered, both for updating scheme which is commonly implemented in software (GPP-based platform) and lookup scheme which resides in FPGA logic. In order to compare updating processes, we consider the cases of ERM (UV-based lookup) and memory indexing (UR-based lookup). In addition, two main updating operations can be considered, namely, *set* and *clear*, for insertion and removal of a UV, respectively. A third operation, namely, modification, is essentially a special case of insertion where the associated range of an already used UV is changed. In this work, we concentrate on complexity of *set* operations; remaining ones can be related to it.

In ERM, $2N$ M-bit words are reserved at synthesis time; that is, one row with two M-bit bounds for each rule. In addition, a valid bit is associated with each pair of [start, end] words, indicating whether they store a valid rule or just garbage. Valid bits are ANDed with comparison results before leaving the lookup engine. For both population and updating cases, a *Rule ID* (RID) must be first decoded to determine the corresponding row to be updated in the architecture. A *set* operation implies storing the involved [start, end] words and setting the related valid bit. A *clear* operation, meanwhile, implies clearing the valid bit of the involved rule. Modification, finally, is a special case of *set* for the case where RID is already used in the ruleset. In [36] the updating process is analyzed in detail for a systolic hardware pipeline.

RID is a fundamental concept in UV-based lookup schemes such as ERM. Every RID is related with a group of field values and a specific priority. In the BM case, the RID and its associated priority define actions on packets for itself, since just one of the multiple matching RIDs can be the lookup result. In other words, just N different results are possible for BM lookup, and these results are represented by RIDs. In UV-based lookup such as CAMs or ERM, RIDs are updated independently and a priority encoder selects the highest-priority RID during lookup. For the MM case, however, the specific combination of matching RIDs must be output, so the use of URs arises as a direct way for determining decision on the packet. In other words, $|\mathrm{UR}|$ different results are now possible for the same N RIDs. For this purpose, the priority encoder commonly applied in BM is useless; instead of it, RIDs must be mapped to URs. One possible implementation is updating individual UVs as before and replacing the priority encoder with N-to-$|\mathrm{UR}|$ compression hardware, which would have high cost. The second implementation consists on compressing UVs during update and storing the resulting URIDs in the lookup engine; this option effectively shifts computation from lookup (FPGA) to update (GPP) and is evaluated in this section.

As suggested in previous paragraph, both ruleset population and incremental update in UR-based lookup schemes are more complex than their UV-based counterparts; so their precomputation requirements must be carefully considered. UR-based schemes are specially suited to MM and flow-based classification since they can fit both BM (i.e., RIDs) and MM (i.e., URIDs) cases with very high speed; that is why we consider them worth optimizing. Precomputation for ruleset population in the original RFC scheme [13] could take hours to execute for complex rule overlappings such as those of ruleset *ipc1* [46]. Computation complexity was further optimized in [14, 47], achieving improvements over the original scheme. During ruleset population, URs are computed by considering a fixed ruleset size. Incremental updating, however, has not been considered separately and can take as much computation as populating the entire ruleset does. In this work, we consider incremental updating as a separate case and propose an optimization which significantly reduces the time for adding/erasing individual UVs.

In Algorithm 1 we show the RFC algorithm for populating an IND scheme with horizontal stitching. If needed, the key is split in chunks (line (1)), defining different ranges at each chunk, and fed to respective lookup tables at each PE. In lines (3)–(8), the 2^m addresses at each chunk are scanned and corresponding BVs are constructed for each of them. After building each BV, it is compared against those already built in lines (9)–(12) in order to determine if the particular combination of UVs is already present in the BV table. Let us remember that each particular BV is related to a UR; thus, if the just built BV is already present in the BV array, its UR is simply repeated; if not, a new UR is counted as shown in lines (13)–(15). In lines (16)-(17), finally, the computed UV and UR are stored in their respective arrays. It is worth mentioning that these arrays are held in the memory of the updating system, which is normally GPP-based and has high-volume storage. For lookup purposes, just *ur_mem* will be later transferred to FPGA memory. Another subtle aspect, mentioned in Section 1, is illustrated by the simple ruleset of Figure 2(a). This ruleset consists of $|\mathrm{UV}_{AB}| = 6$ rules; however, just 4 UVs and 5 UVs are present in fields A and B respectively, i.e., $|\mathrm{UV}_A| = 4$ and $|\mathrm{UV}_B| = 5$. This fact, very common in real rulesets, is because field ranges are reused for different rules. As a consequence, UVs really considered while defining URs and the related precomputation at each field can scale much better than N. For ERM; in contrast, N rules must be matched at each field, no matter how they share UVs at such fields.

We now consider incremental update. In this case, $|\mathrm{UV}|$ can increase in 1 unit depending on whether the new UV exists or not in the ruleset. Since the new BV patterns generated by this UV are not known in advance, we should run the algorithm of Algorithm 2 for each new UV and determine whether it defines a new UR or not. If not, it means that the new UV already exists in the ruleset. In this algorithm, unlike the population one, just one UV is considered. As a consequence, we span just the scope of the new UV instead of the whole address space; thus the number of cycles required for updating is $S_{\mathrm{UV}} = (\mathrm{UV.end} - \mathrm{UV.start}) \leq 2^m$. This is illustrated in Figure 17(a) for a ruleset initially formed by UVs 1, 2, 3, and 4 and an incoming UV5. For a medium-scope UV where $S_{\mathrm{UV}} = (2^m)/2$, the time complexity of the updating algorithm is half that of ruleset population one, while they can be the same for the worst case. This is very inefficient for frequent incremental updates.

In Algorithm 3 and Figure 17(b), we show a modification based on our analysis of UR patterns in Section 3; as we will demonstrate, this modification can effectively speedup the updating process of IND-based lookup. As shown in Figures 4, 5, and 6, no more than 2 new UR labels (URIDs) are added to *ur_mem* for each added UV. The new URs can either begin at the start of the new UV or end at the end of the new UV. For example, in cases (1) and (4) of Figure 6(a) and case (1) of Figure 6(b) one new UR is introduced, while 2 URs are introduced in cases (2) and (3) of Figure 6(a) and cases (2)–(4) of Figure 6(b). In addition, the two opposite ends of the new URs are determined by the opposite ends of the already existing URs. We can exploit this fact to radically reduce updating complexity of IND-based schemes. Initially, we make two independent lookups with key values UV.start (UV.s) and UV.end (UV.e), respectively, which according to Figure 4 results in at most two different URIDs; let us denote them as UR_A and UR_B. Both URIDs are to be replaced by new URIDs just along $[\mathrm{UV}.s, \mathrm{UR}_A.e]$ and $[\mathrm{UR}_B.s, \mathrm{UV}.e]$ intervals. In addition, there can be totally overlapped URs such that $\mathrm{UR}.s > \mathrm{UV}.s$ or $\mathrm{UR}.e < \mathrm{UV}.e$, but their respective URIDs are unaffected by the new UV. In

```
Require: Range bounds UV.s, UV.e for all UVs at each chunk, number of rules N
for chunk = 0 to ⌈M/m⌉ do
    p ← 0
    // simulate lookups for each rule and build BV
    for i = 0 to 2^m do
        for j = 0 to N do
            if i ≥ UV[j].s and i ≤ UV[j].e then
                BV ← BV or 2^j
            end if
        end for
        // check if BV already exists for other key value
        j ← 0
        while j ≤ p and BV ≠ bv_mem[j] do
            j ← j + 1
        end while
        // if BV not found, increment UR counter
        if j ≥ p then
            p ← p + 1
        end if
        // store UR and BV for current key value. Just ur_mem will be transferred to lookup engine (FPGA)
        bv_mem[p] ← BV
        ur_mem[i] ← p
    end for
    chunk.|UR| ← p
end for
```

Algorithm 1: Ruleset population algorithm for IND.

```
Require: UV.s, UV.e, chunk.|UR|, chunk.|UV|, and next available URID nxt_urid for all chunks
for chunk = 0 to ⌈M/m⌉ do
    p ← chunk.|UR|
    v ← chunk.|UV|
    // add new UV range to BV
    for i = UV.start to UV.end do
        BV ← bv_mem[i] or 2^(v+1)
        // check if new BV already exists for other key value
        j ← uv.start
        while j ≤ i and BV ≠ bv_mem[j] do
            j ← j + 1
        end while
        // if BV not found, increment UR counter
        if j ≥ i then
            p ← p + 1
            ur_mem[p] ← nxt_urid
        end if
        bv_mem[p] ← BV
    end for
    chunk.|UR| ← p
end for
```

Algorithm 2: Incremental update algorithm for IND.

general, both intervals [UV.s, UR$_A$.e] and [UR$_B$.s, UV.e] are much smaller than [UV.s, UV.e], so a gain is achieved.

We note that a subtle issue must be solved for our enhancement to work. In cases (1), (2), and (3) of Figures 6(a) and 6(b), UR$_A$ and UR$_B$ and their respective [UR.s, UR.e] ranges are well defined. However, in case (4) of Figure 6(a) a problem arises; namely, URIDs resulting from lookup at UV.s and UV.e are equal (i.e., URID$_A$ = URID$_B$ = 3). This is because a UR can imply multiple [start, end] bounds, as long as it is biunivocally defined by *UV overlapping*. In *ur_mem* of Figure 17(b), for example, we consider the same case in a different context of 4 UVs. We perform lookup of

```
Require: Range bounds UV.s, UV.e, chunk.|UR|, chunk.|UV|, next available URID p and ERID q for all chunks
  (1) for chunk = 0 to ⌈M/m⌉ do
  (2)   ER_A ← er_mem[UV.s]
  (3)   ER_B ← er_mem[UV.e]
        // update region and bound memories
  (4)   for i = UV.s to ER_A.e do
  (5)     ur_mem[i] ← p
  (6)     er_mem[i] ← q
  (7)     bnd_mem[i] ← UV.s, ER.e
  (8)   end for
  (9)   for i = ER_B.s to UV.e do
  (10)    ur_mem[i] ← p
  (11)    er_mem[i] ← q
  (12)    bnd_mem[i] ← UV.s, ER.e
  (13)  end for
  (14) end for
```

Algorithm 3: Proposed optimization for incremental update in IND.

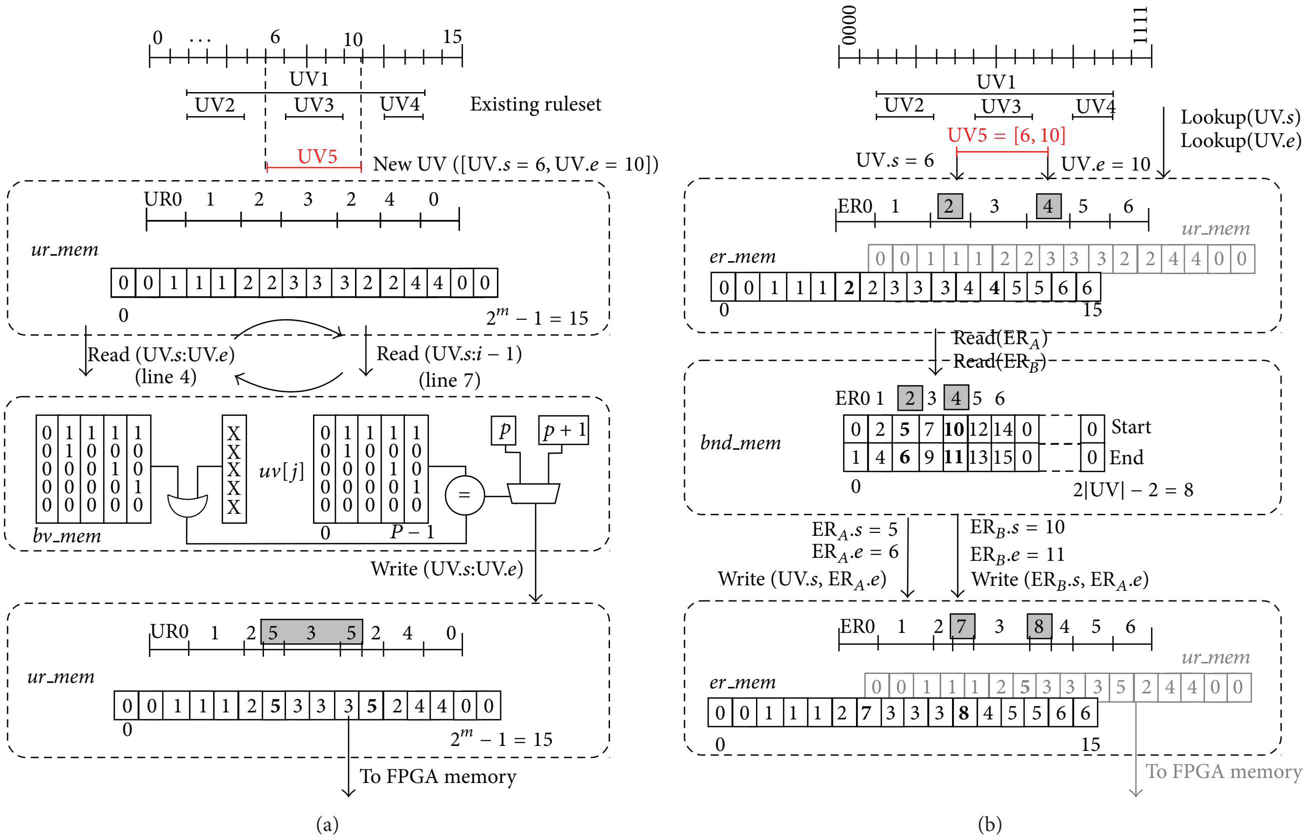


Figure 17: Incremental updating: (a) original architecture for updating of DIR lookup schemes and (b) proposed updating architecture.

keys UV.s = 6 and UV.e = 10 to get URIDs. With such URIDs, we index a second memory *bnd_mem* which maps URIDs to respective [start, end] bounds. Such bounds serve to narrow the number of read/write iterations in line (4) of Algorithm 2. If we performed lookup on *ur_mem*, we would get the same [UR.s, UR.e] tuple for both UR_A and UR_B. To solve this problem, we could either store multiple [UR.s, UR.e] tuples at each *ur_mem* word, which would increase logic complexity, or implement a second array *(er_mem)* storing what we call *expanded regions* (ERs). Expanded regions consider the worst case of |UR|, namely, 2|UV|, so each ER is biunivocally related to one [ER.s, ER.e] tuple. In the example of Figure 17(b), *er_mem* yields the ERIDs ER1 = 2 and ER2 = 4 which biunivocally represent intervals [5, 6] and [10, 11], respectively. Since UV5 = [6, 10], we can determine that intervals [UV.s, ER1.e] = [6, 6] and

Table 3: Comparison of considered lookup schemes.

Lookup scheme	Memory	Logic	Mem BW
Logic-based ERM (BV) [36]	$O(2 \cdot N \cdot M)$	$O(3 \cdot N \cdot M)$	$O(2 \cdot N \cdot M)$
(Optional) postencoding BV → UR	—	—	—
IND, no stitching (UR)	$O(2^M \cdot \log_2 \vert\text{UR}\vert)$	$O(0)$	$O(\log_2 \vert\text{UR}\vert)$
(Optional) postencoding UR → BV	$O(\vert\text{UR}\vert \cdot N)$	$O(0)$	$O(N)$
IND, no stitching (BV)	$O(2^M \cdot N)$	$O(0)$	$O(N)$
IND, BV-based horizontal stitching (BV) [10]	$O(\lceil M/m \rceil \cdot (2^m \cdot \log_2 \vert\text{UR}\vert + \vert\text{UR}\vert \cdot N))$	$O(\lceil M/m \rceil \cdot N)$	$O(N)$
IND, UR-based horizontal stitching (UR) [13]	$O(2 \cdot 2^m \cdot \log_2(\vert\text{UR}\vert/2) + 2^{2 \cdot \log_2(\vert\text{UR}\vert/2)} \cdot \log_2 \vert\text{UR}\vert)$	$O(0)$	$O(\log_2 \vert\text{UR}\vert)$
IND, vertical stitching (UR) [43]	$2^X \cdot 2^{M-X} \cdot \log_2 \vert\text{UR}\vert$	$O(2^X \cdot (1 + \log_2 \vert\text{UR}\vert))$	$O(\log_2 \vert\text{UR}\vert)$
Binary search (UR) [16]	$O(M \cdot \vert\text{UR}\vert + \vert\text{UR}\vert \cdot \log_2 \vert\text{UR}\vert)$	$O(M \cdot \log_2 \vert\text{UR}\vert)$	$O(\log_2 \vert\text{UR}\vert)$
(Optional) postencoding UR → BV	$O(\vert\text{UR}\vert \cdot N)$	$O(0)$	$O(N)$
Binary search (BV)	$O(M \cdot \vert\text{UR}\vert + \vert\text{UR}\vert \cdot N)$	$O(M \cdot \log_2 \vert\text{UR}\vert)$	$O(N)$

$[\text{ER2}.s, \text{UV}.e] = [10, 10]$ must be updated without going through the entire $[0, 2^m - 1] = [0, 15]$ range as RFC requires.

Both *er_mem* and *bnd_mem* are used for updating purposes, but just *ur_mem* is downloaded to the lookup engine. Thus, our modification does not affect the high lookup performance of the original RFC approach. Moreover, memory consumption at the updating platform can be even reduced since *bv_mem* is now replaced by the combination of *er_mem* and *bnd_mem*. *er_mem* and *bnd_mem* scale as $O(2^m \cdot \log_2(2|\text{UV}|))$ and $O(m)$, respectively, while *bv_mem* scales as $O(N)$; thus gain can be obtained in memory consumption. Our proposal can be combined with those of [47] to get an optimized IND-based lookup. Contributions of [47] are, namely, the following: (1) it reduces the number of AND operations for checking BV $\neq$ *bv_mem*$[j]$ (line (7) of Algorithm 2) by adopting aggregated bit vectors (ABVs), (2) it reduces the number of iterations in lines (4)–(15) of Algorithm 2 by accessing multiple words of *bv_mem* through a hash, and (3) it compresses *ur_mem* through grouping of words containing the same URID. Our proposal removes the need of both (1) and (2) since *bv_mem* is eliminated from the beginning; moreover, our solution is convenient since the involved processing is simple and deterministic. (3), meanwhile, is an effective technique which we adopt to reduce the size of the resulting *ur_mem*. We discuss our results in the next section, while extensive evaluation of (3) can be found in [47].

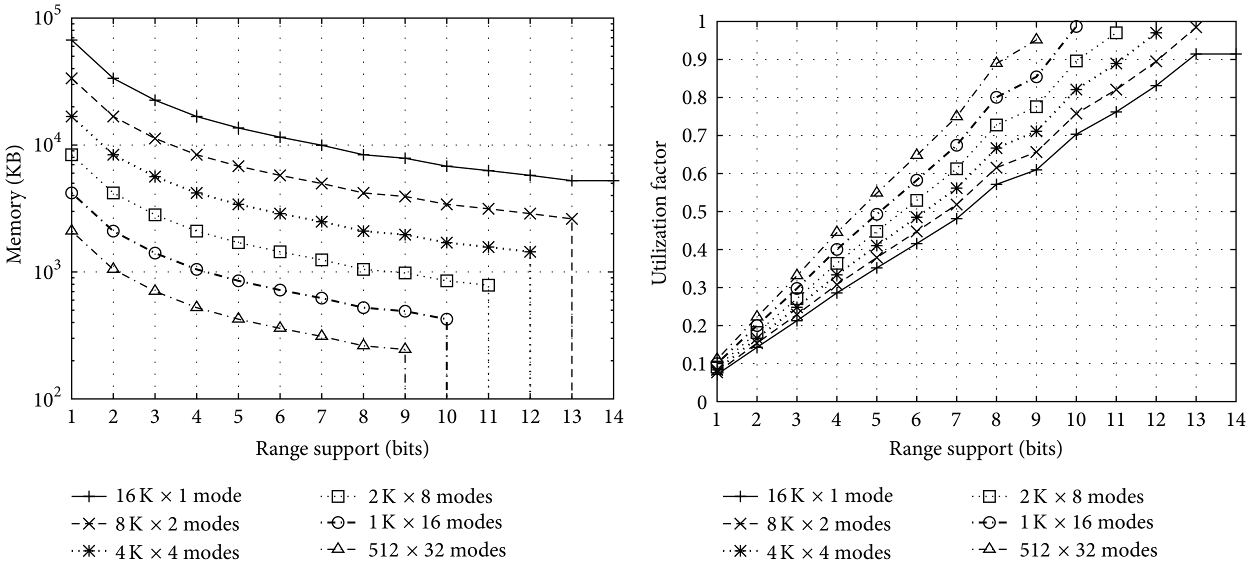
From the technological perspective, meanwhile, we can improve updating performance by observing two facts; on the one hand, a *reg_mem* block would normally use deep-and-narrow modes, since its width scales with $\log_2|\text{UR}|$ instead of N. On the other hand, we can exploit selected mixed-width configurations available for BRAMs so we can write multiple read-side words at a time. In this way, for example, the architecture of Figure 12 with read form factor 4 K×4 can take advantage of write form factor 512×40; that is, 10 words can be written at a time, while $2^4 = 16$ URs are supported by a single BRAM row. By considering both facts and checking UV scopes present in real rulesets [47], we can set trade-offs for selection of read and write BRAM form factors.

5. Results

In this section, we consider relevant performance metrics for the main considered architectures (i.e., IND, BS, and ERM) and we compare them for typical use cases in packet lookup. We finally compare our performance estimates with results of FPGA synthesis for each of them. To summarize our previous analysis of lookup schemes, in Table 3 we provide complexity estimations for each of them; considered metrics are memory footprint, logic consumption, and memory bandwidth (Mem BW) on FPGA. Interfaces to classification stage are noted in parenthesis, that is, (BV) or (UR). Postencoding BV → UR is the counterpart of priority encoding for the MM case; estimations are not provided for it since its implementation would not be practical.

For the particular case of IND, a critical issue is to use BRAM blocks as efficiently as possible due to the address versus range expansion trade-off. In FPGAs, BRAM can be configured in selected modes which ultimately bound possible use cases.

Without loss of generality, let us consider a 32-rule case which is a typical upper bound for x in modern FPGAs (i.e., upper bound for width of M20K BRAMs) and key width $M = 128$. In Figure 18(a) we illustrate memory footprint for different modes of M20K BRAMs available on Altera Stratix V FPGAs. For this analysis, we consider ranges supported by a single chunk when propagating 1 bit/rule (i.e., $W \leq m$). M20K BRAMs can be configured in modes from 20 K × 1 to 512 × 40. For our evaluation, we consider power-of-two range cases, that is, from 16 K × 1 (i.e., form factor 14/1) to 512×32 (i.e., form factor 9/32). For a form factor m/x, address space 2^m defines possible ranges while memory width x defines number of stored URs or UVs (i.e., $\log_2|\text{UR}|$ or n resp.). BRAM stitching allows implementing wider keys and/or wider words. For narrow ranges, shallow-and-wide modes are suitable; as seen in Figure 18(a), 512 × 32 mode is the best option for ranges of $W \leq 9$ or prefix/exact matches which do not require special range support. When support of ranges $9 < W < 15$ is strictly required, deeper modes should be selected at cost of reduced x and poor memory efficiency.



(a) (b)

Figure 18: Range support through BRAM modes: (a) memory footprint and (b) utilization factor.

With the deepest mode available, 14-bit ranges can be covered.

Given a required range support, the selection of BRAM mode mostly aims at minimizing wasted memory. To better appreciate how much of the implemented memory blocks is really used for a particular range case, we consider the utilization factor u in Figure 18(b). As already suggested by Figure 18(a), the required range is most efficiently implemented by the BRAM mode with nearest range support. For example, range = 5 is more efficiently supported by 512 × 32 mode than by other modes with worse address expansion. We note that since $\lceil M/m \rceil$ columns are required where $M = 128$, $9 \leq W \leq 14u < 1$ in these cases.

In Figure 19(a), meanwhile, we explore the opposite point of view, that is, how key widths affect memory footprint while supporting fixed W-bit range for a single rule. We include two hypothetical cases $W = 1, 5$ as well as the two real end cases $W = 9, 14$ for thorough comparison. As expected, memory footprint increases in steps when key width requires new BRAM columns. Due to address expansion, BRAM supporting wide ranges is intrinsically bigger for the same key width. For the end case of 14-bit range and 128-bit key, about 4 Mbits of BRAM are needed to implement a single rule, while a realistic case of 9-bit AR and 32-bit key (also supporting 32-bit prefixes such as IPv4) requires about 40 Kbits. Since each BRAM result is registered and aggregated with results from remaining BRAMs in a row, the introduced latency is also of interest. In Figure 19(b) we observe this parameter for varying range support and key width. As opposed to memory footprint, latency is reduced when supporting wider ranges; this is because each BRAM spans more key bits at a time. For example, we see that lookup supporting 1-bit range can take as high as 128 cycles for 128-bit key, while the case of 9-bit range support implemented in 512×32 mode reduces latency to moderate $\lceil 128/9 \rceil = 15$-cycle latency for 128-bit key. It is worth noting that despite this latency single-cycle throughput is maintained in all cases by proper pipelining.

As shown, 512×32 seems to be the best option regarding memory utilization with acceptable latency. In order to support AR in IND (BV output) with this memory mode, we can also propagate 3 bits/rule as in Figure 16(b); in this way we can use the most efficient mode 512 × 32 at cost of more memory width per rule (i.e., $\lfloor 32/3 \rfloor = 10$ rules supported per BRAM). For IND (UR output), UR-based intrafield aggregation of Figure 11(a) can be used which also enables supporting AR without recurring to address expansion.

We now aim at comparing relevant factors affecting performance of the considered lookup schemes. To this end, we first consider the complexities in Table 3. It is worth noting that all of the considered schemes support line speed lookup, even if different latencies can be introduced. From these schemes, we further concentrate on four generic approaches shown in bold font in Table 3, namely,

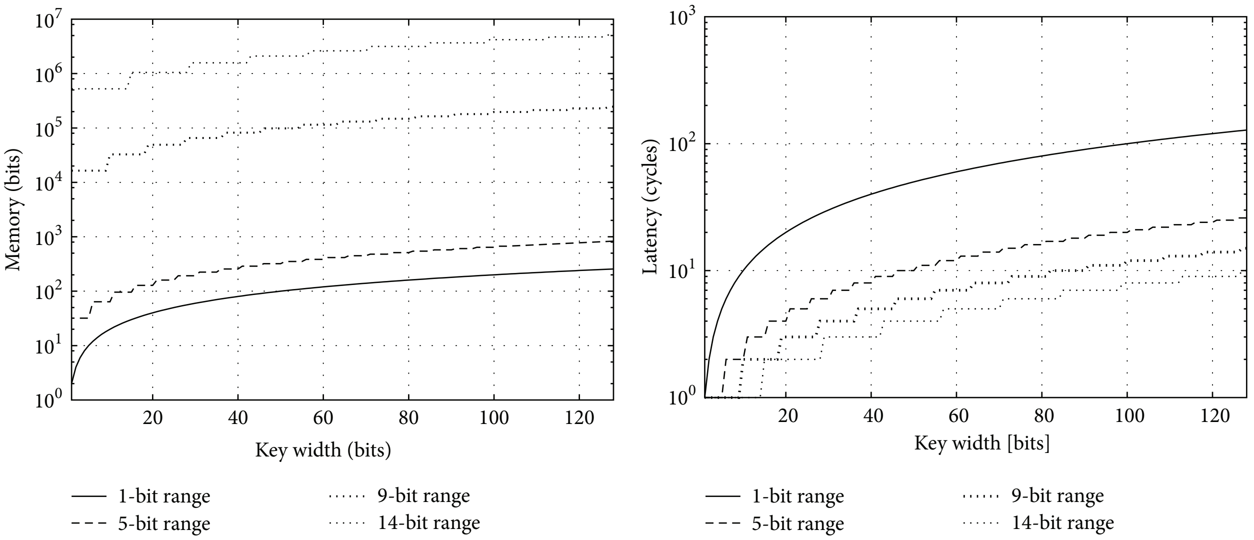
(1) logic-based ERM (BV output);

(2) stitched IND with BV-based horizontal stitching (BV output);

(3) stitched IND with UR-based horizontal stitching (UR output);

(4) binary search tree (UR output).

In addition, we include and compare postfitting synthesis results for cases (1), (2), and (4). In general terms, (1) and (2) are more predictable than (3) and (4) since the first ones depend on N; the last ones are sensitive to rule patterns so our general evaluation considers a midpoint case $|\mathrm{UR}| = |\mathrm{UV}|$, $|\mathrm{UV}| = N$. The advantage of (3) and (4) resides in their better scalability with N and the fact that they are specially suited for MM, flow-based classification. The FPGA architecture for (3) has no major complexities since it is entirely based on



(a)

(b)

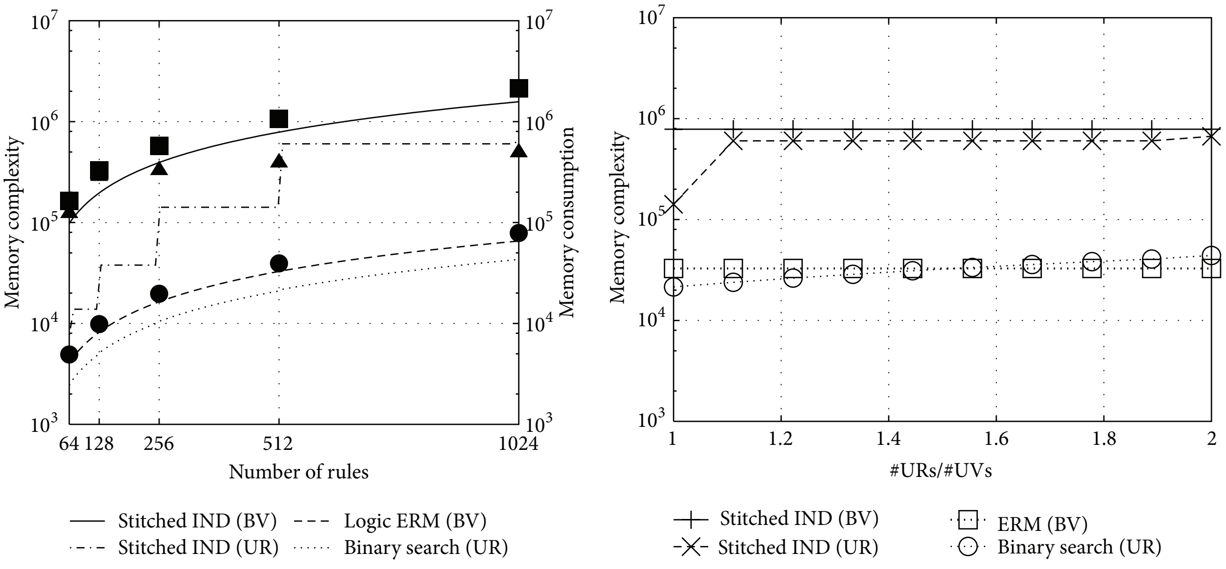
Figure 19: Key width impact: (a) memory footprint and (b) latency.

successive memory indexing, so we discuss its scalability just from estimation results; its implementation is ultimately limited by the maximal addressing expansion supported by memory modes on FPGAs. (4) can be the solution to this limit since it reduces memory expansion to $2^m = |\mathrm{UR}|_{\max} = 2|\mathrm{UV}|$; however, it introduces trade-offs regarding required clock cycles, dependency on rule overlappings, and memory imbalance between tree stages. Since (4) introduces more design trade-offs than (3), we do include a generic implementation for this case.

In Figure 20(a), memory consumption of considered schemes is compared for $M = 32$, $m = 9$, $|\mathrm{UR}| = |\mathrm{UV}| = N$, and $64 \leq N \leq 1024$. For approach (3), which is essentially RFC applied to field chunks, we consider pairwise aggregation of chunk results by multiple iterations of the scheme of Figure 11(a). This is intrafield aggregation (see Figure 10) and must not be confused with posterior interfield aggregation of decomposition-based classification (not considered in this work). For $M = 32$ and $m = 9$, that is, $\lceil 32/9 = 4 \rceil$ memory blocks are required at first aggregation stage, 2 blocks at second one and one block at the last one (which i.e., delivers lookup result). Based on observations on real rulesets [13], we consider the number of URs to double at each (intrafield) aggregation stage. For example, if 256 regions are defined for 32-bit field and this field is sliced in $\lceil 32/9 \rceil = 4$ chunks, then $256/4 = 64$ URs are defined on each chunk of the first stage, $256/2$ URs are defined at the intermediate aggregation stage, and 256 URs are defined at the final aggregation stage. As shown in Figure 20(a), BS requires least storage followed by ERM, this is because we consider a midcase $|\mathrm{UR}| = N$. As we will see in Figure 20(b), the BS versus ERM ratio can vary for $N \leq |\mathrm{UR}| < 2N$. We note that ERM storage exclusively consists of registers. Memory indexing with BV output, on the opposite end, requires most storage due to the contributions of addressing expansion and memory word width N. If the required output are URIDs, meanwhile, storage is significantly reduced at cost of precomputation. It is worth noting that because M20K can actually achieve 9/40 form factor BRAM requirements of IND (BV) (filled squares) are slightly higher than estimations which suppose 9/32 form factors for the sake of generality. Register consumption of ERM (filled circles), meanwhile, matches quite well our estimations. As discussed before, BRAM consumption of BS is quite inefficient in our generic architecture (filled triangles), showing the memory imbalance problem of trees; through optimizations proposed, for example, in [30], one can get closer to estimation results. Figure 20(b), meanwhile, shows memory consumption as function of the ratio $|\mathrm{UR}|/|\mathrm{UV}|$ where $|\mathrm{UV}| = N = 512$ and $1 \leq |\mathrm{UR}|/|\mathrm{UV}| \leq 2$. We observe results which confirm our discussion of Section 3. Both stitched IND (BV) and stitched IND (UR) have high memory requirements due to addressing expansion. Decision trees, meanwhile, can achieve equal or even lower requirements than ERM at cost of multiple stages.

In Figure 21(a), we compare estimated and real combinational logic complexity for varying $N = |\mathrm{UV}| = |\mathrm{UR}|$. For implementation purposes, LUT count is the relevant metric. ERM shows extensive logic consumption, while memory indexing shifts most computation to the update phase requiring much less logic. BS shows least logic consumption, mainly used for magnitude comparison. Figure 21(b), meanwhile, shows logic complexity with $|\mathrm{UR}|/|\mathrm{UV}|$. IND schemes show very small logic consumption (stitched IND (UR) consumes in fact $O(0)$ logic), while ERM has maximum logic consumption which keeps constant for varying $|\mathrm{UR}|/|\mathrm{UV}|$. Decision trees reach logic consumption between those of stitched IND (UR) and stitched IND (BV) by consuming just enough logic to compare against $|\mathrm{UR}|$ M-bit bounds.

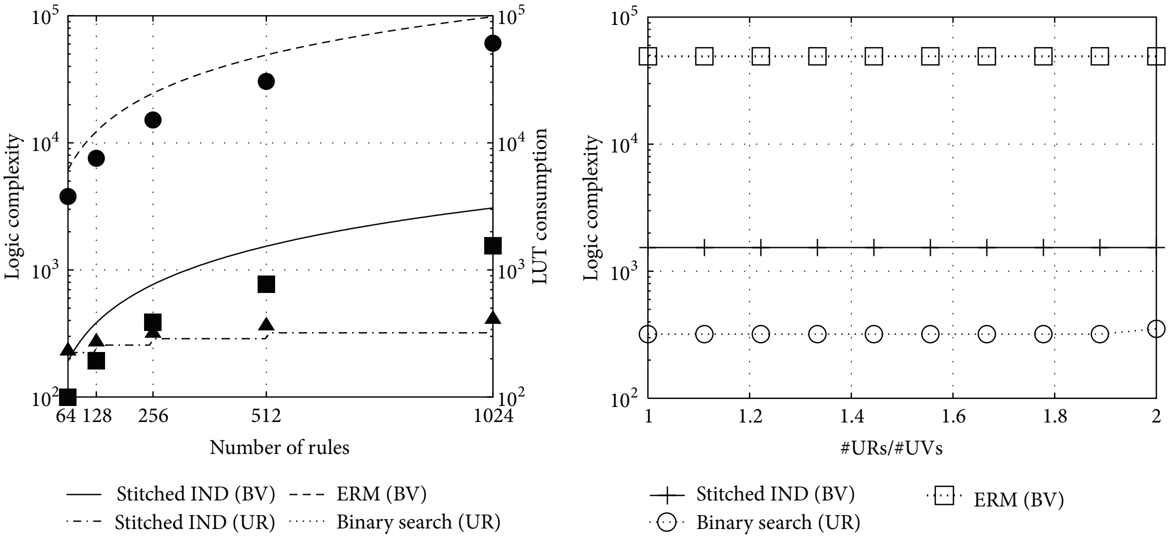
In Figures 22(a) and 22(b), finally, we show that ERM requires most memory bandwidth for both varying $|\mathrm{UV}| = N$ and $|\mathrm{UR}|/|\mathrm{UV}|$, since it essentially depends on concurrent access to $2 \cdot N \cdot M$ independent registers. Schemes delivering



(a)

(b)

FIGURE 20: Memory complexity: (a) $|UR| = |UV| = N$, $64 \le N \le 1024$ and (b) $|UV| = N = 512$, $|UV| \le |UR| \le 2 \cdot |UV|$.



(a)

(b)

FIGURE 21: Logic complexity: (a) $|UR| = |UV| = N$, $64 \le N \le 1024$ and (b) $|UV| = N = 512$, $|UV| \le |UR| \le 2 \cdot |UV|$.

URIDs require minimal memory bandwidth, while stitched IND (BV) stays in the middle since it scales with N but does not require M-bit comparison for each of the rules as ERM does.

From preceding analysis, we note that ERM does not suffer from address expansion; however, its required memory bandwidth and routing complexity are high. Both factors can also increase energy consumption dramatically. ERM memory consumption is optimal as predicted no matter the ratio $|UR|/|UV|$, while its logic consumption is significant compared with memory indexing. Performance of ERM degrades for large N; this is because it is based on TCAM scheme and inherits its drawbacks. Even if not considered here, additional resources are needed for pipelined priority encoding (BV-to-RID mapping) or BV-to-UR mapping at the output of this scheme; this stage is also inherited from TCAMs and critical for large Ns. Memory indexing, meanwhile, has the flexibility to exploit rule patterns through precomputation and storage of URIDs with better adaptability to large rulesets or favorable rule patterns; this is a very useful feature for current applications. However, address expansion limits key width M for arbitrary range support, requiring to mitigate this problem through some of the described stitching schemes or address space compression through binary search trees.

We now aim at evaluating performance metrics for our proposed optimization of incremental updating for UR-based

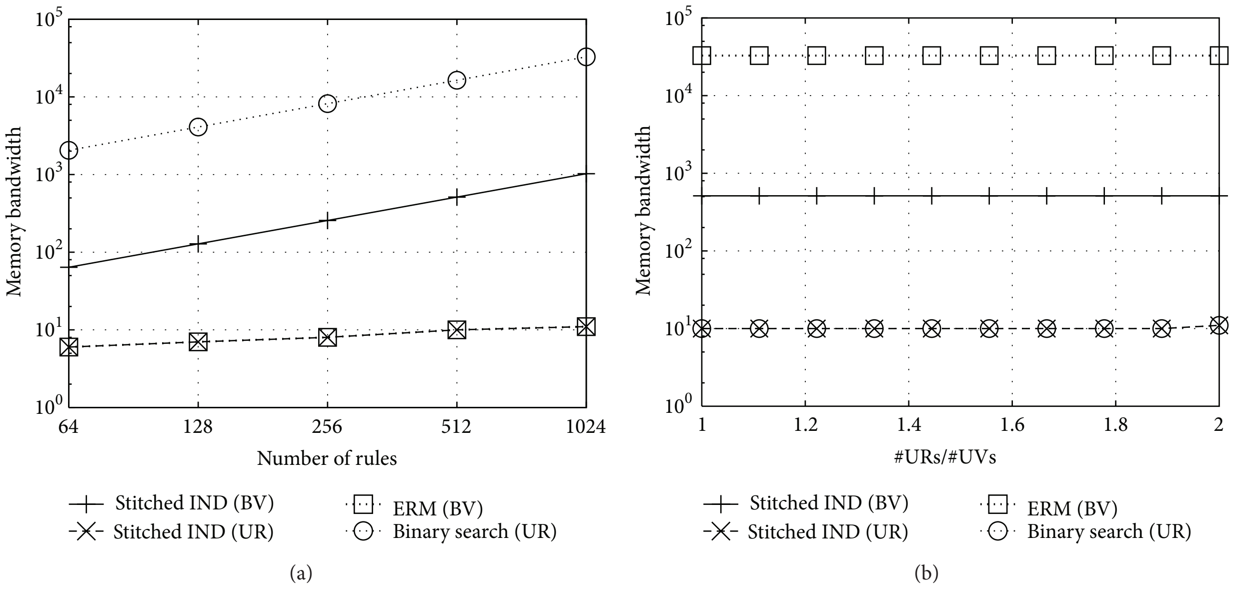
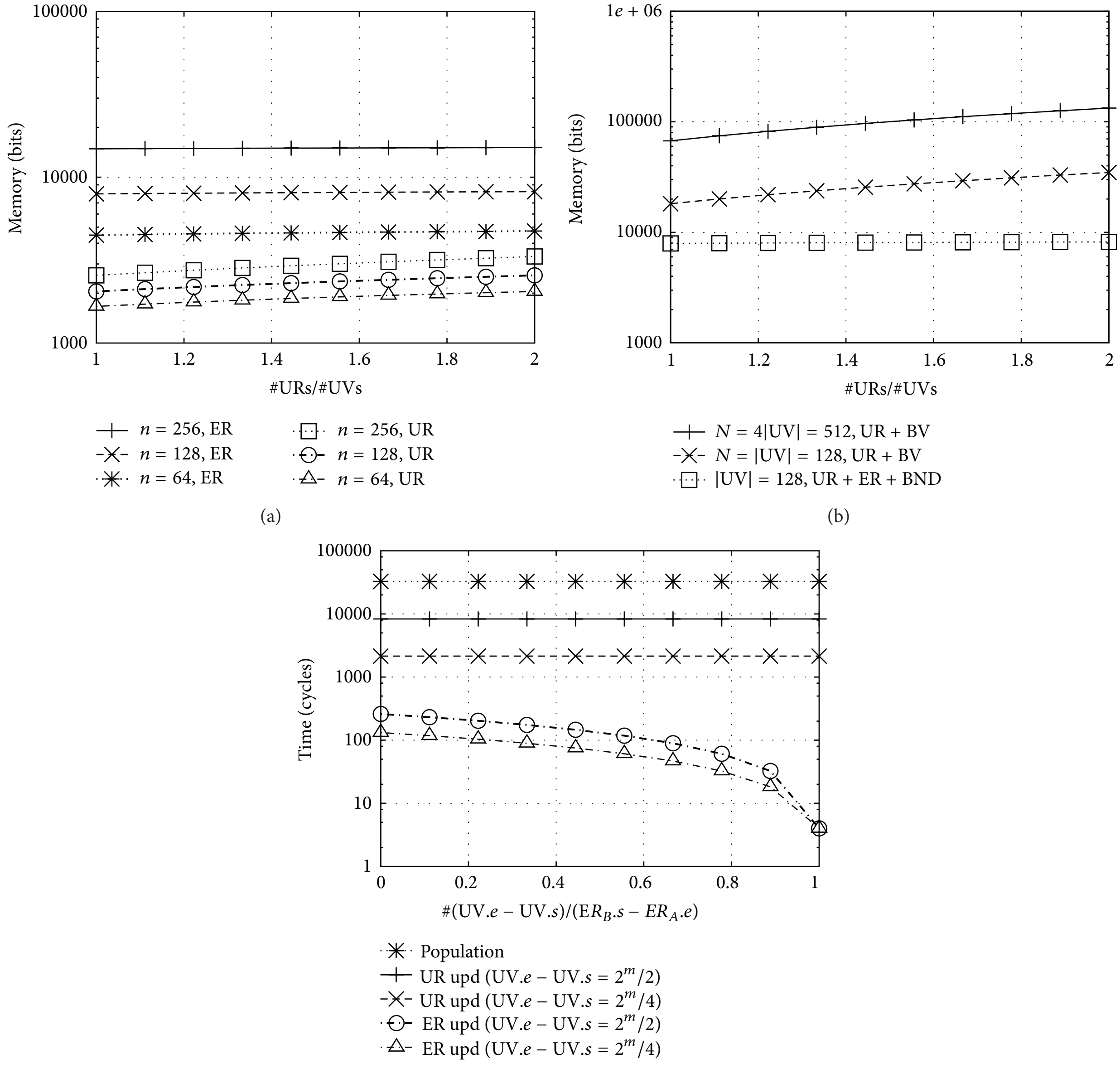


FIGURE 22: Memory bandwith: (a) |UR| = |UV| = N, $64 \leq N \leq 1024$ and (b) |UV| = $N = 512$, $|\text{UV}| \leq |\text{UR}| \leq 2 \cdot |\text{UV}|$.

(UR output) schemes, discussed in Section 4. Based on the algorithms of Algorithms 1, 2, and 3, the scheme of Figure 17, and previous results in [13, 47], we evaluate space and time requirements of previous and proposed schemes. In Figure 23(a) we consider how costly *er_mem* is with respect to *ur_mem* for varying $|\text{UV}| \leq |\text{UR}| < 2|\text{UV}|$, $|\text{UV}| = N$, and $N = 64, 128, 256$. As already discussed, we consider $|\text{UV}| = N$ since the original population/update algorithm simply adds a new bit to BV for each incremental update, no matter if that UV's range was already present in *bv_mem* (lines (4)–(8) of Algorithm 1). As shown, since *er_mem* considers the worst case $|\text{UR}| = 2N$; it has a tolerable higher space cost than that of *ur_mem*. *er_mem*, however, enables storing |ER| m-bit bounds in *bnd_mem instead* of keeping |UR| N-bit words in *bv_mem*; thus, all involved memories *ur_mem*, *er_mem*, and *bnd_mem* now grow as $2^m \log_2 |\text{UV}|$, $2^m \log_2(2N)$ and $2 \cdot N \cdot m$, respectively. In Figure 23(b) we show that storing UR + ER + BND memories can be cheaper than storing UR + BV ones. Moreover, since the original scheme does not take advantage of the fact that $|\text{UV}| \ll N$ in real lookup cases, we show that space cost is even larger for a case where $N = 4|\text{UV}|$ while our proposal keeps its cost. In Figure 23(c), finally, we show that updating reduced scopes $[\text{UV}.s, \text{UR}_A.e]$ and $[\text{UR}_B.s, \text{UV}.e]$ of Algorithm 3 instead of the new UV's scope $[\text{UV}.s, \text{UV}.e]$ of Algorithm 2 can lead to significant gains. As shown, the population algorithm has the highest time cost even if it is just performed when lookup is put in service. UR and ER update times are shown for two general UV scopes $(\text{UV}.e - \text{UV}.s) = 2^m/2$ and $(\text{UV}.e - \text{UV}.s) = 2^m/4$ (i.e., half and quarter of the key values resp.). In addition, we consider variable ratio between the new $[\text{UV}.s, \text{UV}.e]$ and existing $[\text{ER}_A.e, \text{ER}_B.s]$ scopes to show that ER can take effective advantage of it to reduce UR update times. Offsets are not relevant provided that $\text{UV}.s \leq \text{ER}_A.s$ and $\text{UV}.e \geq \text{ER}_B.e$, so they are not considered for our evaluation.

In Tables 4, 5, and 6 we report our space/speed balanced postfitting implementation results for stitched IND (BV output), ERM (BV output), and BS (UR output) schemes, respectively. The considered architectures were described in Verilog HDL and simulated on Mentor ModelSim; then they were synthesized and evaluated on Altera Quartus II and related analysis tools targeting an Altera Stratix V FPGA. We consider $16 \leq M \leq 128$ and $256 \leq N \leq 1024$. IND stores rules in BRAM, with minimal LUT and Reg consumption for AND-based aggregation and pipelining purposes, respectively. ERM, meanwhile, has intensive use of registers for storage of rules and pipelining, while it uses LUTs for implementing magnitude comparators and AND-based aggregation. BS, finally, uses BRAM for implementation of *ctrl_mem* in Figure 13(b), LUTs for magnitude comparison, and registers for pipelining purposes. To guarantee best performance of BS, we tested multiple comparator designs and finally used the *lpm_compare* megafunction provided by Altera, which showed best scalability. BRAM consumption is quite higher that estimation results since (a) BRAM has coarse granularity leading to memory waste at stages near to root and (b) BRAM width is not just M but that of *key_value* + *next_or_urid* + *fnd* (see Figure 13(b)). Based on observations in [36], we used $m = 4, n = 8$ for optimal implementation of ERM (BV); while we used $m = 9$, $n = 40$ for implementation of IND (BV) based on observations in Figure 18. As confirmation of our analysis, we observe that throughput (Mlps) of IND scales better than that of ERM for increasing N and M. Based on these results and on previous work [47], we predict that performance of IND (UR) should be even better than that of IND (BV) since the memory bandwidth is greatly reduced from $O(N)$ to $O(\log_2|\text{UR}|)$. BS, meanwhile, shows moderate BRAM consumption despite its storage inefficiency; for small trees, registers can also be used for storage at early stages for optimal storage efficiency.



(a)

(b)

(c)

Figure 23: Updating of IND schemes: (a) memory consumption of *ur_mem* versus *er_mem*, (b) total memory consumption, and (c) required clock cycles.

Throughput of BS is the lowest of the three schemes due to limited scalability of the magnitude comparator. A possible remedy to this problem, which we do not explore further in this work, could be using pipelined comparators.

We observe that recent proposals [36, 48] use ERM in order to tackle the AR match problem; however, from the beginning they assume BV-based output which suffers scalability problems and lacks flexibility to exploit the ratio $|\mathrm{UR}|/|\mathrm{UV}|$. Moreover, they are unable to exploit the $|\mathrm{UV}|/N$ ratio, while $|\mathrm{UV}| \ll N$ for most fields of real rulesets [12]. For example, for *ipc1* ruleset [46], $N = 1550$, $|\mathrm{UV}|_{\mathrm{SrcIP}} = 152$, and $|\mathrm{UV}|_{\mathrm{SrcPort}} = 34$. We argue that this feature will remain valid for actual and future networking applications. Memory indexing, meanwhile, can be used for both UV- and UR-based outputs; in particular, by using it in combination with UR-based aggregation we can fully exploit its benefits at cost of offline precomputation. Since URs scale as $O(\log_2|\mathrm{UV}|)$, the resulting architecture can be more scalable than UV-based ones. We also note that, in order to take actions on packets, BV results cannot be used directly but require an additional stage which maps them to BM or MM labels (i.e., actions to be taken on the packet). For the case of BM, this stage consists of a pipelined priority encoder which delivers $\log_2 N$-width labels; this is basically the same approach adopted in TCAMs. For the more general case of MM lookup, however, a memory-based stage is required which maps BVs

Table 4: Performance of IND, horizontal BV-stitching, and 2D-pipelined architecture on FPGA ($m = 9$, $n = 40$).

	Rules (N)											
M	256				512				1024			
	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps
16	14	260	108	589	26	533	216	592	52	1078	450	555
32	28	388	216	588	52	775	432	588	104	1543	900	514
128	105	1149	810	509	195	2280	1620	450	390	4627	3375	377

Table 5: Performance of ERM and 2D-pipelined architecture on FPGA ($m = 4$, $n = 8$).

	Rules (N)											
M	256				512				1024			
	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps
16	0	7721	9984	437	0	15433	19968	467	0	31235	39899	392
32	0	15145	19712	453	0	30464	39351	384	0	60902	78775	352
128	0	34845	77791	321	0	69815	155871	336	0	139676	312031	320

Table 6: Performance of BS and pipelined architecture on FPGA ($N \leq |\mathrm{UR}| \leq 2N$).

	Rules (N)											
M	256				512				1024			
	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps	M20K	LUTs	Regs	Mlps
16	8	179	1254	328	10	206	1463	320	13	235	1684	293
32	16	316	2038	246	19	361	2343	229	24	407	2660	245
128	32	1054	6742	191	39	1193	7623	172	53	1332	8516	174

to URs. Such a stage can add much complexity to the ERM scheme. UR-based engines involve such URs as part of the lookup process, so they are naturally suited to MM lookups without extra stages.

6. Conclusion

In this work, we review lookup techniques for packet classification on FPGAs. In particular, we approach the lookup problem both from the lookup case (i.e., BM/MM and EX/PX/AR) and from the lookup-aggregation interface (i.e., BV/URID) points of view. From our analysis, we arrive to a general taxonomy which helps in recognizing most appropriate schemes for current packet lookup needs. Moreover, we provide estimations and implementation results for FPGA-based platforms.

UR-based schemes such as RFC are very fast due to the use of simple memory indexing. They are applicable to both intrafield aggregation (i.e., stitching) and interfield aggregation. From our current study, we find them specially suited for current packet lookup and classification needs. At this point, the main limitations of such schemes are (1) precomputation required for incremental updating and (2) their scalability with respect to key width M due to address expansion. Previous work explored efficient techniques for mitigating them. In this work, we recognize new upper bounds for UR update complexity and smartly exploit them to further optimize those techniques.

Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

Acknowledgments

This work was partially funded by the FONCyT-National Technological University IP-PRH 2007 Postgraduate Grant Program, CONICET Postgraduate Grant Program, and FONCyT PICT 2011-2527 Research Grant. Authors would also like to thank Altera Corporation for resources and support provided.

References

[1] Open Networking Foundation, "Software-Defined Networking: The NewNorm for Networks," ONF White Paper, April 2012.

[2] D. Unnikrishnan, R. Vadlamani, Y. Liao, J. Crenne, L. Gao, and R. Tessier, "Reconfigurable data planes for scalable network virtualization," *IEEE Transactions on Computers*, vol. 62, no. 12, pp. 2476–2488, 2013.

[3] D. Bacon, R. Rabbah, and S. Shukla, "FPGA programming for the masses," *Queue—Mobile Web Development*, vol. 11, no. 2, pp. 40–53, 2013.

[4] S. K. Maurya and L. T. Clark, "A dynamic longest prefix matching content addressable memory for IP routing," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 6, pp. 963–972, 2011.

[5] M. Bando, Y.-L. Lin, and H. J. Chao, "Flash trie: beyond 100-Gb/s IP route lookup using hash-based prefix-compressed trie," *IEEE/ACM Transactions on Networking*, vol. 20, no. 4, pp. 1262–1275, 2012.

[6] Y.-H. E. Yang, Y. Qu, S. Haria, and V. K. Prasanna, "Architecture and performance models for scalable IP lookup engines on FPGA," in *Proceedings of the IEEE 14th International Conference on High Performance Switching and Routing (HPSR '13)*, pp. 156–163, Taipei, Taiwan, July 2013.

[7] A. Rasmussen, A. Kragelund, M. Berger, H. Wessing, and S. Ruepp, "TCAM-based high speed Longest prefix matching with fast incremental table updates," in *Proceedings of the IEEE 14th International Conference on High Performance Switching and Routing (HPSR '13)*, pp. 43–48, July 2013.

[8] F. Yu, R. H. Katz, and T. V. Lakshman, "Efficient multimatch packet classification and lookup with TCAM," *IEEE Micro*, vol. 25, no. 1, pp. 50–59, 2005.

[9] M. P. Fernandez, "Comparing OpenFlow controller paradigms scalability: reactive and proactive," in *Proceedings of the 27th IEEE International Conference on Advanced Information Networking and Applications (AINA '13)*, pp. 1009–1016, March 2013.

[10] C. A. Zerbini and J. M. Finochietto, "Performance evaluation of packet classification on FPGA-based TCAM emulation architectures," in *Proceedings of the IEEE Global Communications Conference (GLOBECOM '12)*, pp. 2766–2771, December 2012.

[11] W. Jiang and V. K. Prasanna, "Scalable packet classification on FPGA," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 20, no. 9, pp. 1668–1680, 2012.

[12] G. S. Jedhe, A. Ramamoorthy, and K. Varghese, "A scalable high throughput firewall in FPGA," in *Proceedings of the16th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '08)*, pp. 43–52, April 2008.

[13] P. Gupta and N. McKeown, "Packet classification on multiple fields," in *Proceedings of the Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication (SIGCOMM '99)*, pp. 147–160, 1999.

[14] J. van Lunteren and T. Engbersen, "Fast and scalable packet classification," *IEEE Journal on Selected Areas in Communications*, vol. 21, no. 4, pp. 560–571, 2003.

[15] X. Sun, S. K. Sahni, and Y. Q. Zhao, "Packet classification consuming small amount of memory," *IEEE/ACM Transactions on Networking*, vol. 13, no. 5, pp. 1135–1145, 2005.

[16] T. V. Lakshman and D. Stiliadis, "High-speed policy-based packet forwardingusing efficient multi-dimensional range matching," in *Proceedings of the ACM Conference on Applications, Technologies, Architectures, and Protocolsfor Computer Communication (SIGCOMM '98)*, pp. 203–214, 1998.

[17] D. E. Taylor and J. S. Turner, "Scalable packet classification using distributed crossproducting of field labels," in *Proceedings of the IEEE 24th Annual Joint Conference of the IEEE Computer and Communications Societies (INFOCOM '05)*, vol. 1, pp. 269–280, 2005.

[18] A. Nikitakis and I. Papaefstathiou, "A multi Gigabit FPGA-based 5-tuple classification system," in *Proceedings of the IEEE International Conference on Communications (ICC '08)*, pp. 2081–2085, May 2008.

[19] M. Faezipour and M. Nourani, "Wire-speed TCAM-based architectures for multimatch packet classification," *IEEE Transactions on Computers*, vol. 58, no. 1, pp. 5–17, 2009.

[20] A. Bremlerr-Barr and D. Hendler, "Space-efficient TCAM-based classification using gray coding," *IEEE Transactions on Computers*, vol. 61, no. 1, pp. 18–30, 2012.

[21] O. Rottenstreich, R. Cohen, D. Raz, and I. Keslassy, "Exact worst case TCAM rule expansion," *IEEE Transactions on Computers*, vol. 62, no. 6, pp. 1127–1140, 2013.

[22] F. Zane, G. Narlikar, and A. Basu, "CoolCAMs: power-efficient TCAMs for forwarding engines," in *Proceedings of the 22nd Annual Joint Conference on the IEEE Computer and Communications (INFOCOM '03)*, vol. 1, pp. 42–52, April 2003.

[23] E. Spitznagel, D. Taylor, and J. Turner, "Packet classification using extended TCAMs," in *Proceedings of the 11th IEEE International Conference on Network Protocols*, pp. 120–131, November 2003.

[24] C. R. Meiners, A. X. Liu, E. Torng, and J. Patel, "SPliT: optimizing space, power, and throughput for TCAM-based classification," in *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '11)*, pp. 200–210, October 2011.

[25] H. J. Chao and B. Liu, *High Performance Switches and Routers*, Wiley-IEEE Press, 2007.

[26] G. Varghese, *Network Algorithmics*, Morgan Kaufmann, San Francisco, Calif, USA, 2005.

[27] F. Pong and N.-F. Tzeng, "Concise lookup tables for IPv4 and IPv6 longest prefix matching in scalable routers," *IEEE/ACM Transactions on Networking*, vol. 20, no. 3, pp. 729–741, 2012.

[28] M. Waldvogel, G. Varghese, J. Turner, and B. Plattner, "Scalable high-speed prefix matching," *ACM Transactions on Computer Systems*, vol. 19, no. 4, pp. 440–482, 2001.

[29] P. Warkhede, S. Suri, and G. Varghese, "Multiway range trees: scalable IP lookup with fast updates," *Computer Networks*, vol. 44, no. 3, pp. 289–303, 2004.

[30] Y. Qu and V. K. Prasanna, "High-Performance pipelined architecture for tree-based IP lookup engine on FPGA," in *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium*, pp. 114–123, May 2013.

[31] W. Jiang and V. K. Prasanna, "Field-split parallel architecture for high performance multi-match packet classification using FPGAs," in *Proceedings of the 21st Annual Symposium on Parallelism in Algorithms and Architectures (SPAA '09)*, pp. 188–196, August 2009.

[32] O. Ahmed, S. Areibi, K. Chattha, and B. Kelly, "PCIU: hardware implementations of an efficient packet classification algorithm with an incremental update capability," *International Journal of Reconfigurable Computing*, vol. 2011, Article ID 648483, 21 pages, 2011.

[33] T. Ganegedara and V. K. Prasanna, "StrideBV: single chip 400G+ packet classification," in *Proceedings of the IEEE 13th International Conference on High Performance Switching and Routing (HPSR '12)*, pp. 1–6, June 2012.

[34] A. Sanny, T. Ganegedara, and V. K. Prasanna, "A comparison of ruleset feature independent packet classification engines on FPGA," in *Proceedings of the IEEE 27th International Parallel and Distributed Processing Symposium Workshops and PhD Forum (IPDPSW '13)*, pp. 124–133, IEEE, Cambridge, Mass, USA, May 2013.

[35] W. Jiang, "Scalable ternary content addressable memory implementation using FPGAs," in *Proceedings of the ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13)*, pp. 71–82, October 2013.

[36] Y. R. Qu, S. Zhou, and V. K. Prasanna, "High-performance architecture for dynamically updatable packet classification on FPGA," in *Proceedings of the 9th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '13)*, pp. 125–136, October 2013.

[37] D. E. Taylor and J. S. Turner, "ClassBench: a packet classification benchmark," *IEEE/ACM Transactions on Networking*, vol. 15, no. 3, pp. 499–511, 2007.

[38] S. Choi, R. Scrofano, V. K. Prasanna, and J. Jang, "Energy-efficient signal processing using FPGAs," in *Proceedings of the ACM/SIGDA 11th International Symposium on Field Programmable Gate Arrays (FPGA '03)*, pp. 225–234, 2003.

[39] T. Ganegedara, V. Prasanna, and G. Brebner, "Optimizing packet lookup in time and space on FPGA," in *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL '12)*, pp. 270–276, August 2012.

[40] "Advanced Synthesis Cookbook," July 2011, http://www.altera.com/literature/manual/stx_cookbook.pdf.

[41] J. Brelet, *An Overview of Multiple TCAM Designs Invirtex Family Devices*, Xilinx Corporation, 1999, http://www.xilinx.com/support/documentation/applicationnotes/xapp201.pdf.

[42] H. Rong and H. Chen, "An independent set packet classification algorithm using priority sorting," *Journal of Networks*, vol. 6, no. 11, pp. 1565–1571, 2011.

[43] R. Tessier, V. Betz, D. Neto, and T. Gopalsamy, "Power-awareRAM mapping for FPGA embedded memory blocks," in *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '06)*, pp. 189–198, February 2006.

[44] Altera Corporation, *Embedded Memory Blocks in Stratix V Devices*, Altera Corporation, 2013, http://www.altera.com/literature/hb/stratix-v/stx5_51003.pdf.

[45] Y. Qi, J. Fong, W. Jiang, B. Xu, J. Li, and V. Prasanna, "Multi-dimensional packet classification on FPGA: 100 Gbps and beyond," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '10)*, pp. 241–248, December 2010.

[46] H. Song and J. S. Turner, "Toward advocacy-free evaluation of packet classification algorithms," *IEEE Transactions on Computers*, vol. 60, no. 5, pp. 723–733, 2011.

[47] X.-Y. Gong, W.-D. Wang, and S.-D. Cheng, "ERFC: an enhanced recursive flow classification algorithm," *Journal of Computer Science and Technology*, vol. 25, no. 5, pp. 958–969, 2010.

[48] T. Ganegedara, W. Jiang, and V. K. Prasanna, "A scalable and modular architecture for high-performance packet classification," *IEEE Transactions on Parallel and Distributed Systems*, vol. 25, no. 5, pp. 1135–1144, 2014.

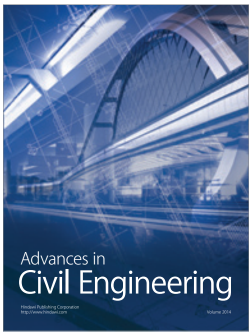
Journal of
Engineering

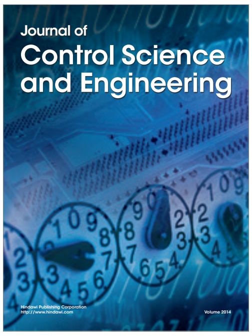
The Scientific World Journal

International Journal of
Rotating Machinery

Journal of
Sensors

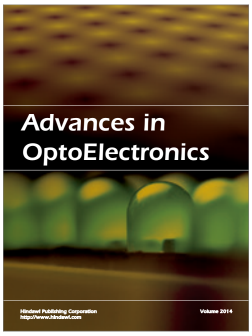
International Journal of
Distributed Sensor Networks


Advances in
Civil Engineering


Journal of
Control Science and Engineering

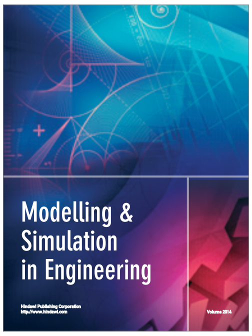
Journal of
Robotics

Journal of
Electrical and Computer Engineering

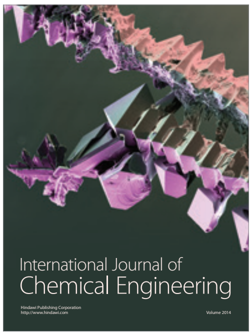

Advances in
OptoElectronics

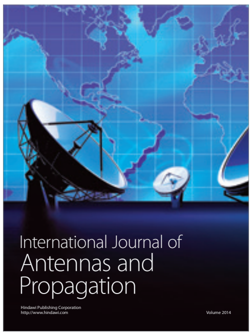
VLSI Design

International Journal of
Navigation and Observation

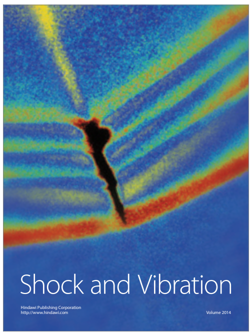

Modelling & Simulation in Engineering

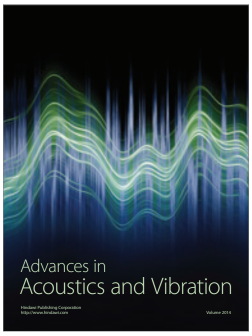
International Journal of
Aerospace Engineering


International Journal of
Chemical Engineering


International Journal of
Antennas and Propagation

Active and Passive Electronic Components


Shock and Vibration


Advances in
Acoustics and Vibration