Scalable Memory-Less Architecture for String Matching With FPGAs

Ideh Sarbishei, Shervin Vakili, J.M. Pierre Langlois, and Yvon Savaria
Department of Computer and Software Engineering
Polytechnique Montréal, Canada
{ideh.sarbishei, shervin.vakili, pierre.langlois, yvon.savaria}@polymtl.ca

***Abstract*—String matching hardware engines generally utilize Ternary Content Addressable Memories (TCAMs). Although TCAM-based solutions are fast, they are expensive and power hungry. This paper proposes a high-performance memory-less architecture for string matching called Split-Bucket. It offers a performance comparable to TCAM-based solutions. Moreover, it is reconfigurable and scalable to the size of the target string set and the width of the string. The architecture is characterized using the Longest Prefix Match problem for IP address lookup and is implemented on a Virtex-7 FPGA. For a real-world routing table with 524 k IPv4 prefixes, the Split-Bucket architecture achieves a throughput of 103.4 M packets per second and consumes 23% and 22% of the Look Up Tables and Flip-Flops of a Xilinx XC7V2000T chip, respectively.**

Keywords—FPGA; Latency; Resource Utilization; String Matching; Ternary Content Addressable Memory (TCAM)

I. INTRODUCTION

A string matching engine searches for all occurrences of a given string inside a predetermined target string set. When a match is found, its location and associated information are returned. Each string is an ordered vector of symbols of a given alphabet (e.g. Latin, Binary and DNA). String matching has a broad range of applications in computer science and bioinformatics, such as genes/proteins/DNA matching, Network Intrusion Detection Systems (NIDS), and IP lookup systems in network routers. Many of these applications require real-time processing, such as solving the Longest Prefix Match (LPM) problem, which is a crucial part of the IP lookup process [1-2].

Existing approaches for string matching can be categorized into Ternary Content Addressable Memory (TCAM)-based [1], trie-based techniques [2-4], and TCAM-emulation, [5-6]. TCAM-based techniques exploit the TCAM capability to perform high-speed parallel searches in the target string set, but have limited access speed, high power consumption, and high cost. To mitigate these limitations, TCAM-based techniques minimize the size of required TCAMs. TCAM-emulation techniques aim at reducing TCAM cost and power consumption using components that replicate TCAM functionality. Trie-based techniques create search trees from the given target string set and store the tree information in memory using a trie data structure [4]. For applications such as LPM using large prefix sets, trie information may not fit in on-chip memories, thus increasing search latency. Several approaches have been proposed to minimize external memory accesses [4].

This paper proposes the Split-Bucket, a novel, scalable, flexible, high-performance and memory-less architecture for real-time string matching using Field Programmable Gate Arrays (FPGAs). The main goal of Split-Bucket is to provide a fast parallel search that addresses the shortcomings of TCAM-based and trie-based techniques, such as high TCAM cost and power consumption, and nondeterministic latency and external memory access of trie-based techniques. Moreover, as the target string set is hardcoded in the logical resources of the FPGA, the Split-Bucket does not require any internal or external memories.

The paper is organized as follows. Section II presents related works. Section III describes the proposed Split-Bucket architecture. Section IV presents experimental results for a LPM case study. Section V compares the Split-Bucket with existing TCAM-emulation and trie-based techniques. Section VI concludes the paper and highlights some directions for future work.

II. RELATED WORK

This section reviews some related work on string matching using TCAM-based, TCAM-emulation or trie-based techniques. Certain TCAM-based approaches have employed specific characteristics of existing commercial TCAMs to meet specific application constraints. Zheng *et al.* [1] applied multiple parallel partitioned TCAM chips to achieve high throughput and low power consumption. They proposed a new method to evenly distribute the prefix loads and the associated lookup traffic among the TCAM partitions. TCAM-emulation technique approaches emulate TCAM functionality to support high performance string matching, while avoiding TCAM costs. Clark and Schimmel [5] suggested an FPGA implementation of a scalable string match that allows adjusting the trade-off between capacity and throughput according to application requirements. They applied multi-character decoders to their design in order to improve performance and eliminate redundant comparisons. At the industrial level, Xilinx has developed an IP core that can be configured to utilize either SRLs or BRAM resources to emulate the functionality of different TCAM core sizes [6]. Trie-based approaches utilize the trie data structure to perform string matching. Baker and Prasanna [3] presented a tool that provides automatic synthesis of highly efficient NIDS on an FPGA. This tool applies tree-based prefix sharing to reduce redundant comparisons. Moreover, it applies high-level graph-based string partitioning to reduce decoder size and share the shift registers. Yang, Erdem and Prasanna proposed a trie-based architecture implemented on FPGA for IP lookup [2]. They suggested a method to perform a LPM in $log(L\text{-}c)$ phases, where L is the size of the IP address and c is a design constant. Each phase has an individual local table mapped to an FPGA using on-chip and off-chip memories. Using external memories, it supports very large routing tables with high throughput. Matoušek *et al.* [4] proposed a trie-based approach introducing memory efficient dedicated hardware for IP lookup. They suggested a representation of IP prefix set in memory applying novel node types and an algorithm to map the nodes to a tree leading to low memory usage.

978-1-4673-6853-7/17/$31.00 ©2017 IEEE

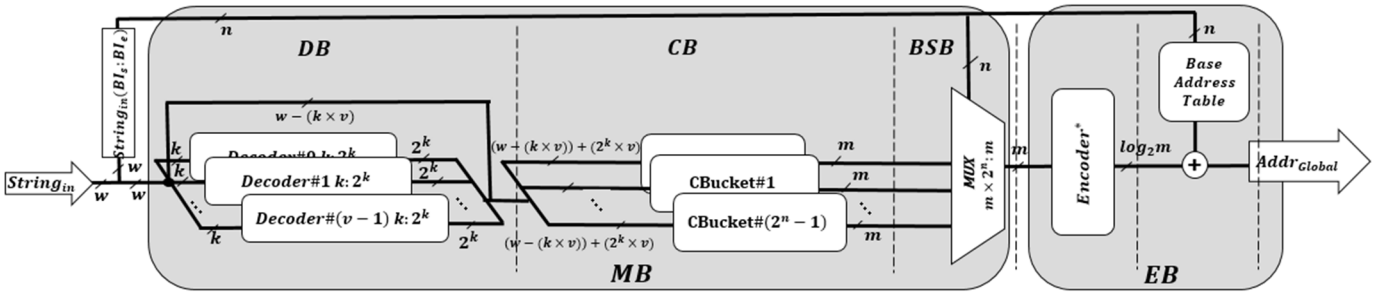

Fig. 1: The proposed Split-Bucket architecture. The encoder can be replaced by a priority encoder for applications where multiple matches are possible.

III. Split-Bucket Architecture

Fig. 1 shows the proposed Split-Bucket architecture. Similar to TCAM-based and TCAM-emulation approaches [5], the string matching process consists of two major steps. First, in a Match Block (MB), the input string is compared in parallel with all strings in the target string set to find a match. Next, the location of the match is reported by an Encoder Block (EB).

A. Match Block

A parallel comparison with all the strings in the target string set is performed in the MB using the following three pipelined sub-blocks: the Decoder Block (DB), the Comparator Block (CB) and the Bucket Selection Block (BSB).

A large target string set may contain several strings with identical segments. Duplicating comparisons with these segments would waste resources. The input string is thus split into v segments of k bits, each one decoded by a k-to-2^k binary decoder. For non-binary alphabets, symbols should first be encoded in binary to allow the utilization of binary decoders. Clark and Schimmel [5] employed a similar decoding technique to improve the efficiency of the comparisons for large sets.

The Split-Bucket architecture employs a string partitioning scheme to improve string matching efficiency [1], [3]. It partitions the target string set into a predefined number of buckets using a *bucket identifier*. The bucket identifier is an n-bit portion of the string ($String(BI_s:BI_e)$). The BI_s, BI_e indicate the starting and ending indexes of the bucket identifier, respectively. Strings with the same bucket identifier share a bucket. The bucket identifier of the input string ($String_{in}(BI_s:BI_e)$) defines the bucket that contains useful comparison results. The string distribution among the buckets depends on the target string set. As shown in Fig. 1, the CB consists of parallel components, called *CBucket*. Each *CBucket* compares the decoded input string with the target strings of the corresponding bucket using several AND operations, each corresponding to a target string. When a string is added to the target string set, the corresponding AND operation must be appended to the appropriate *CBucket*.

The AND operation inputs are the outputs of the decoder for the corresponding segment. The number of input ports of the a^{th} AND operation ($InPorts_a$) depends on the bit-width of the a^{th} string in the target string set (BW_a). If k divides BW_a, the $InPorts_a$ is equal to $\frac{BW_a}{k}$. Otherwise, the number of input ports of the a^{th} AND operation for this case is calculated by (1):

$$InPorts_a = \left\lfloor \frac{BW_a}{k} \right\rfloor + BW_a \bmod k, \qquad a = 0,1,\dots,N-1 \quad (1)$$

The first $\left\lfloor \frac{BW_a}{k} \right\rfloor$ bits of the input string are handled by the decoders while the $BW_a \bmod k$ ending bits will not completely fill the k input ports of a decoder. Therefore, the remaining bits are directly connected to the corresponding AND logic.

The BSB consists of a multiplexer that selects the valid comparison results and passes the outputs of the appropriate *CBucket* according to the input string bucket identifier. The multiplexer has $m \times 2^n$ inputs, where

$$m = \mathrm{MAX}\big(size(CBucket\#b)\big),\ b = 0,1,\dots,2^n-1 \quad (2)$$

and *size(CBucket#b)* is the total number of strings that have been allocated to the b^{th} bucket.

B. Encoder Block

The EB is composed of two pipelined stages. The first stage encodes the MB output using a binary encoder while the second stage calculates the global match address. The encoder receives the comparison results of the appropriate *CBucket*. The architecture shown in Fig. 1 is for the case where at most one match can be found among the results. For applications with multiple possible matches, a priority encoder would replace the encoder The encoder passes the local address of the match ($Addr_{Local}$) in the appropriate *CBucket*. The global match address ($Addr_{Global}$) is calculated using (3):

$$Addr_{Global} = Addr_{Local} + BaseAddr_b, b = 0,\dots,2^n-1, (3)$$

where b is the input string bucket identifier and $BaseAddr_b$ is the base address of the corresponding bucket. Since a fixed-number of target strings are assigned to each bucket, the base address of each bucket is a constant value stored in the Base Address Table. The proposed architecture is memory-less, which indicates that it does not require any RAM blocks. However, a small register bank with flip-flops is needed to store the Base Address Table entries, which are found with (4), with $BaseAddr_0 = 0$:

$$BaseAddr_b = \sum_{i=0}^{b-1} size(CBucket\#i), b = 1,\dots,2^n-1 \quad (4)$$

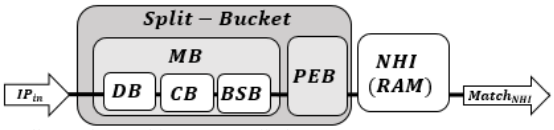

Fig. 2. Split-Bucket architecture applied to LPM

TABLE I. Synthesis Results for Different Table Sizes on Virtex-7

N*	m*	Frequency (MHz)	LUTs	FFs	Latency (ns)
16 k	115	189.7	20.7 k	17.7 k	26.3
20 k	143	181.3	24.5 k	21.5 k	27.5
32.7 k	220	161	36.8 k	34.8 k	31
65.5 k	427	139.9	62.8 k	69.4 k	35.7
90 k	591	137	81.1 k	94.9 k	36.4
131 k	846	119	115.9 k	142 k	42
524 k	3316	103	282.3 k	549.7 k	48.5

*N is the routing table size and m is the size of the largest *CBucket*

IV. IP Address Lookup With the Split-Bucket

We evaluated the Split-Bucket architecture with the LPM IP address lookup application, where multiple matches can occur simultaneously. As shown in Fig. 2, the EB is replaced with a Priority Encoder Block (PEB) to support multiple matches. Moreover, an additional block is required to supply the Next Hop Information (NHI) when a match is found. Port numbers are stored in a RAM, but its cost is not included in the experimental results. An IP address lookup engine based on the Split-Bucket was implemented on a Virtex-7 FPGA using the Xilinx ISE 13.4 tool. The IPV4 routing table used for all experiments was extracted from routing information services (RIS) raw data [7].

In a first set of experiments, the hardware cost and the performance were evaluated for different routing table sizes shown in Table I. In order to achieve a uniform distribution of IP addresses among the buckets, the bucket identifier was chosen as proposed by Zheng *et al.* [1]. We chose $n = 8$, which partitions the routing table into 256 buckets. Using Zheng's approach for 256 buckets, bits 16 to 23 were found to be the most appropriate bucket identifier. Since $k = 8$ and $v = 4$, the second chosen segment is equivalent to the bucket identifier. Therefore, the second decoder in DB is redundant and can be removed from the design, since its function is implicitly executed by the BSB. The hardware area consumption of the Split-Bucket depends on the target table size, and the maximum table size is determined by the total available resources in the target FPGA. In this experiment, the important parameters of the proposed architecture have the following values: $w = 32$, $k = 8$, $v = 4$, $BI_s = 9$, and $BI_e = 16$. Table I shows the area cost in terms of the number of Look Up Tables (LUTs) and Flip-Flops (FFs) for various routing table sizes. The number of LUTs increases almost linearly with the problem size, whereas the number of FFs shows a sub-linear increase. The proposed architecture was evaluated using a table size of up to 524 k entries. The results show that implementing the largest tested table on a XC7V2200T FPGA consumes 23% and 22% of the available LUTs and FFs, respectively. Since the 5-level pipelined Split-Bucket architecture provides one result every clock cycle, the frequency column, shown in Table I, also corresponds to the throughput of the architecture.

In the second set of experiments, resource utilization was characterized for a fixed value of BI_s and different values of n and N. Changing the value of n does not significantly affect the hardware complexity of the DB and CB blocks. Indeed, the BSB and PEB are the main blocks that vary by changing n. Hence, the hardware complexity is estimated by summing up the LUT usage of the BSB and PEB blocks for each value of n. The PEB contains an adder that combines two values of length $log_2 m$ bits. Since increasing n leads to a negligible decrease of $\log 2m$, ignoring variations in resource utilization of the adder in the final estimation does not significantly change the result. However, the resource utilization of the priority encoder in the PEB is dependent on the value of n. For a given number of table entries N, increasing the number of buckets 2^n means that the buckets will be smaller, hence smaller m. Since the PEB only depends on the value of m, a smaller m leads to a smaller priority encoder. When n increases, the resource utilization of the PEB is reduced exponentially. The BSB has a near-linear increase in the LUT usage as n increases. Fig. 4 illustrates the final hardware complexity estimation for different values of n and N. These results show that changing the number of buckets from 16 ($n = 4$) to 128 ($n = 7$) has a limited impact on the overall LUT consumption. The optimal choice for the number of buckets is $n = 5$, which minimizes resource utilization for almost every N.

In the third set of experiments, resource utilization was characterized for a fixed routing table size ($N = 16$ k) and different values of BI_s and n. The CB, BSB and PEB blocks are the main blocks that vary when changing BI_s and n. Hence, resource utilization is estimated by summing up the LUT consumption of CB, BSB and PEB. The CB consists of $N+E$ AND operations, where E is the total number of expansions in the routing table. Every IP address with a prefix size (*prefix*) smaller than the ending index of a bucket identifier (BI_e) is expanded into $2^{BIe-prefix}$ IP addresses with prefix size BI_e. For example, if IP_1="234.84.212.153", $prefix_1 = 13$, $BI_s = 10$ and $BI_e = 15$, IP_1 requires $2^{15-13} = 4$ IP address expansions with a prefix size of 15. A function is proposed to calculate the value of E for a given routing table based on BI_s and n. The complexity of the CB is estimated by summing up the value of N and E. The BSB and PEB consist of an $m \times 2^n$ multiplexer and an m–input priority encoder, respectively. The value of m is determined after the expansion and the bucket division processes to calculate the LUT consumption for BSB and PEB. The estimation function sweeps the value of BI_s and n in the ranges of (1:23) and (2:11), respectively, to estimate the total LUT consumption of the CB, BSB and PEB. The design parameters that minimize the total LUT consumption are selected as the optimal values for the parameters. Fig. 4 shows the design space exploration on BI_s and n for the moderate routing table size of 16383. The optimal value (*minCost*) occurs for a $BI_s = 9$ and $n = 7$. In the reported synthesis results of the IP-Split-Bucket in the first set of experiment, we used 9 and 8 as the values for BI_s and n, respectively. As shown in Fig. 4, there is a negligible difference in the total estimated LUT consumption for a design with $BI_s = 9$ and $n = 8$ compared to the optimal point of $BI_s = 9$ and $n = 7$.

V. Comparison of Split-Bucket and Existing Work

This section compares some existing TCAM-emulation and trie-based approaches on string matching with the Split-Bucket using the FPGA synthesis results shown in Table II. We compare the Split-Bucket with two existing memory-less TCAM-emulation approaches [5-6]. Xilinx developed an FPGA-based TCAM core that can be configured to use either SRLs or RAMs [6]. Since Split-Bucket avoids using memory resources, we compared it with the 32-bit wide SRL-based TCAM core as shown in the sixth column of Table II. The Split-Bucket was evaluated by implementing test cases of comparable sizes on XC5VLX220 FPGAs. The results show that, for a routing table with 1024 IP prefixes, the proposed architecture consumes 83.4% fewer LUTs and offers 3.5× higher throughput compared to the Xilinx TCAM core [6]. Clark and Schimmel [5] evaluated their design using largest test cases among TCAM-emulation works. Therefore, we implemented a comparable table size of 18 k on the Split-Bucket. Table II shows that the Split-Bucket consumes 39% and 64% fewer LUTs and FFs, respectively.

Three trie-based approaches are included in Table II [2-4]. Matoušek's [4] paper presented the work that is most recent and most comparable to ours. We implemented an equivalent table size of 442.7 k on a XC6VSX475T FPGA, using different tested routing tables. The results show that while the Split-Bucket does not use memory resources, Matoušek's approach consumes 7.7 Mb of internal memory. On the other hand, Matoušek's approach requires 85% and 90.5% fewer LUTs and FFs. Moreover, the Split-Bucket achieves 90.8% lower latency

TABLE II: DETAILED COMPARISON OF STRING MATCHING TECHNIQUES

Approaches / Metrics	[3] **2006**	[2] **2011**	[4] **2013**	[5] **2004**	[6] **TCAM core**	**Split-Bucket**				
Device	Virtex-2	Virtex-6	Virtex-6	Virtex-2	Virtex-5	Virtex-6	Virtex-5	Virtex-7		
# of Patterns × # of Char	602 × 20.4	9.5 M × 32	442,748 × 32	17,537× 32	1024 × 32	442,747 × 32	1024 × 32	1024 × 32	18,001 × 32	524,287 × 32
LUTs	6 k	7 k	88K	55K	13K	593K	2.2K	2.3K	22K	282K
FFs	6K	22 k	44K	55K	63	464K	1.9K	1.9K	19K	550K
Memory (Kb)	~ 0	28 k (BRAM) + 4 External SRAMs	7780	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0	~ 0
Frequency (MHz)	216	156	127	219	80	116	362	415	184	103
Throughput (MLPS)	216	312	254	233	80	116	362	415	184	103
Latency (ns)	37	13	472	146	12	43	13.5	12	27	48
Technique	Trie-based			TCAM-emulation						

compared to Matoušek *et al.* [4] while having 54% lower throughput. Yang *et al.* [2] presented a Trie-based approach implemented with memories that supports very large routing tables (up to 9.5 M entries). Although Yang *et al.*'s approach [2] achieves better performance, it consumes a large amount of internal and external memories. Yang *et al.* have not reported the size of the utilized external memories.

VI. Conclusion

This paper proposed a generic scalable memory-less high performance architecture for string matching called Split-Bucket. This architecture was evaluated for the IP address lookup application. Implemented on a Virtex-7 FPGA, the complexity of the proposed architecture is 60% less than previous memory-less string matching approaches, when configured for similar moderate size tables (18 k entries). Moreover, the closest reported solution that can handle comparable size string set tables requires a large (7.7 Mb) internal memory, while Split-Bucket requires no memory.

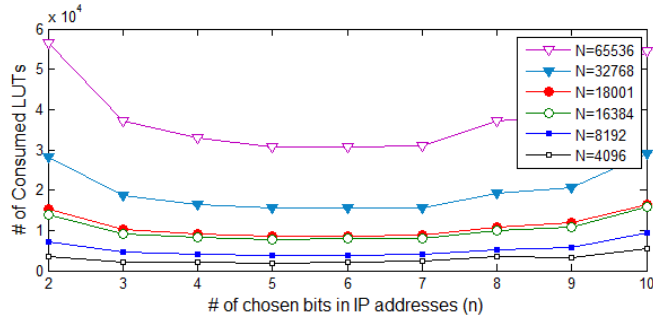

Fig. 3. Resource utilization as a function of the number of bits in the bucket identifier

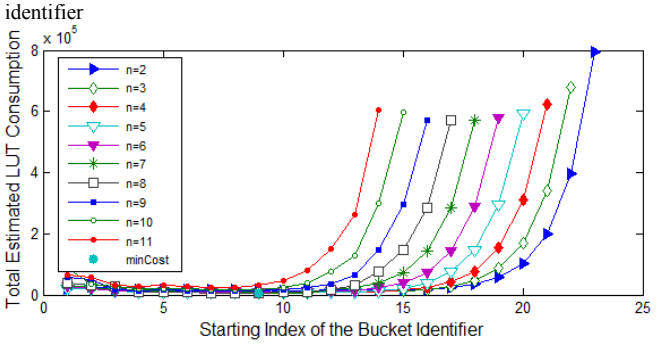

Fig. 4. A design space exploration on BI_s and n

References

[1] K. Zheng, C. Hu, H. Lu and B. Liu, "An Ultra High Throughput and Power Efficient TCAM-Based IP Lookup Engine," *Twenty-third AnnualJoint Conference of the IEEE Computer and Communications Societies INFOCOM,* vol. 3, pp. 1984-1994, 2004.

[2] Y.-H. E. Yang, O. Erdem and V. K. Prasanna, "High performance IP lookup on FPGA with combined length-infix pipelined search," *19th Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM),* pp. 77-80, 2011.

[3] Z. K. Baker and V. K. Prasanna, "Automatic synthesis of efficient intrusion detection systems on FPGA," *IEEE Transactions on Dependable and Secure Computing,* pp. 289-300, 2006.

[4] J. Matoušek, M. Skačan and J. Kořenek, "Memory efficient IP lookup in 100 GBPS networks," *23rd International Conference on Field programmable Logic and Applications,* pp. 1-8, 2013.

[5] C. R. Clark and E. D. Schimmel, "Scalable Pattern Matching for High Speed Networks," in *12th Annual IEEE Symposium on Field-Programmable Custom Computing Machines, 2004. FCCM 2004.*, 2004.

[6] K. Locke, "Parameterizable Content-Addressable," Xilinx Application note, 2011.

[7] "RIPE Network Coordiante Centre," 1 10 2015. [Online]. Available: https://www.ripe.net/analyse/internet-measurements/routing-information-service-ris/ris-raw-data.

[8] H. le, W. Jiang and V. K. Prasanna, "Scalable High Throughput SRAM-based Architecture for IP-Lookup Using FPGA," *International Conference on Field Programmable Logic and Applications (FLP),* pp. 134-142, 2008.

[9] D. Pao, "TCAM Organization for IPV6 Address Lookup," *The 7th International Conference on Advanced Communication Technology,* vol. 1, pp. 26-31, 2005.

[10] I. Sourdis and D. Pnevmatikatos, "Pre-decoded CAM for efficient and high-speed NIDS pattern matching," *Field-Programmable Custom Computing Machines, 2004. FCCM 2004. 12th Annual IEEE Symposium on,* pp. 258-267, 2004.

[11] Y. Sun and M. S. Kim, "A Hybrid Approach to CAM-Based Longest Prefix Matching for IP Route Lookup," *Global Telecommunications Conference,* pp. 1-5, 2010.

[12] A. Rassmussen, A. Kragelund, M. Berger, H. Wessing and S. Ruepp, "TCAM-based High Speed Longest Prefix Matching with Fast Incremental Table Updates," *International Conference on High Performance Switching and Routing,* pp. 43-48, 2013.

[13] Z. Ullah, M. Kumar Jaiswal, Y. Chan and R. C. Cheung, "FPGA Implementation of SRAM-based Ternary Content Addressable Memory," *International Parallel and Distributed Processing Symposium Workshops & PhD Forum,* pp. 383-389, 2012.