Computers and Electrical Engineering 96 (2021) 107403

Contents lists available at ScienceDirect

Computers and Electrical Engineering

journal homepage: www.elsevier.com/locate/compeleceng

Fast and efficient address search in System-on-a-Programmable-Chip using binary trees[☆,☆☆]

Jesús Lázaro [*], Unai Bidarte, Leire Muguira, Carlos Cuadrado, Jaime Jiménez

Department of Electronics Technology, University of the Basque Country, Bilbao, Spain

ARTICLE INFO

Keywords:
FPGA
AVL tree
Ethernet
MAC

ABSTRACT

One processing task in Ethernet nodes is to manage Media Access Control (MAC) addresses: search, insert new, and delete old ones. For this purpose, Content-Addressable Memorys (CAMs) offer low latency and no collisions; however, they consume too many electronic resources, and working frequency is constrained. On the other hand, hash tables demand few circuits allowing fast operations; unfortunately, collisions often occur, causing delays in the process. Finally, binary trees arise as one efficient technique to search addresses by hardware, although updating them is complex.

The design presented in this paper, based on an Adelson-Velsky and Landis (AVL) binary tree, takes advantage of the mixed hardware/software capabilities of Multiprocessor Programmable System-on-a-Chip (MPSoC) devices. It forwards frames on the fly: a hardware core, searches addresses in an AVL tree, and a program inserts and deletes them. This solution requires few resources and, to the best of our knowledge, is the first to manage MAC addresses in an AVL tree and to exploit a hardware/software System-on-a-Chip (SoC) for this purpose.

1. Introduction

Nowadays, Ethernet has turned into the de-facto standard for local area networks, not only in service companies or marketing and financial departments, but also in industrial environments [1]. One of its key sublayers, the MAC, is expected to efficiently manage a table of addresses: look up the target, insert the new and delete the obsolete ones. Since Industrial Ethernet works in real-time, the architecture must search the addresses with known and fixed latencies [2]. The time required by table processes may be lengthy depending on both the algorithms to be performed and the hardware needed to host them.

Such computing requirement has driven to leverage modern FPGAs, as they have evolved to integrate field configuring capabilities (Programmable Logic (PL), hardware) and high-performance processors (Processing System (PS), software) in the same chip. This article presents a hardware–software implementation of a searching algorithm; the hardware part is in charge of looking up and can host any binary search tree (unbalanced, AVL, RB…). The software part can be adapted to any tree; in this proof-of-concept, AVL has been selected as a simple yet efficient algorithm. To the best of our knowledge, this fast — it overcomes FPGA usage of previous implementations as explained in Section 7 — and efficient — just 0.49 % of flip-flops as explained in Section 5 — architecture for a MAC address tree is novel and can be extended to other binary search algorithms in chips that contain PL and PS structures.

☆ This work has been supported by the Ministerio de Economía y Competitividad of Spain within the project TEC2017-84011-R and FEDER funds as well as by the Department of Education of the Basque Government, Spain within the fund for research groups of the Basque university system IT978-16.
☆☆ This paper is for regular issues of CAEE. Reviews processed by the Editor-in-Chief Dr. Manu Malek.
* Corresponding author.
E-mail address: jesus.lazaro@ehu.eus (J. Lázaro).

https://doi.org/10.1016/j.compeleceng.2021.107403
Received 28 June 2020; Received in revised form 10 March 2021; Accepted 30 August 2021
Available online 13 October 2021

Acronyms

AMBA	Advanced Microcontroller Bus Architecture
AVL	Adelson-Velsky and Landis
AXI	Advanced eXtensible Interface
BRAM	Block RAM
CAM	Content-Addressable Memory
CPU	Central Processing Unit
distRAM	dual port distributed RAM
FIFO	First In, First Out
FPGA	Field Programmable Gate Array
FSM	Finite-State Machine
GPU	Graphics Processing Unit
IO	Input Output
IP	Intellectual Property
LUT	Lookup table
LUTRAM	RAM built LUT
MAC	Media Access Control
MPSoC	Multiprocessor Programmable System-on-a-Chip
PE	Processing Element
PL	Programmable Logic
PS	Processing System
RAM	Random Access Memory
SoC	System-on-a-Chip
SRL	Shift Register LUT

The remainder of the paper is organized as follows. Section 2 shows a brief state of the art about content searched by hardware and SoC devices. AVL trees are explained in Section 3, while Section 4 describes the proposed hardware/software design for such a structure, oriented to SoCs. Measurements and results are given in Section 5, validated in Section 6, and compared with other research works in Section 7. Finally, Section 8 concludes the paper.

2. Content search in System-on-Chip

This section covers several search architectures that have been used for network processing. In addition, the SoC for this work, and other hardware implementations of binary search algorithms are described.

2.1. Content-addressable memory

CAM is a special type of structure exploited in specific very-high-speed searching applications that compares input lookup data (tag) with a table of stored entries and returns the address of matching data. Usually, a parallel search is performed over all stored words, and the match position or address can be found in as few as one clock cycle. This approach is faster than comparing each address location in a standard memory architecture. Several examples of CAMs are exploited for networking, and their use in reconfigurable devices has been analyzed [3].

CAM designs usually take advantage of some configuration alternatives or parameters, such as these, taken from [4]:

- Memory is implemented in shift registers or Random Access Memory (RAM).
- Not ternary mode or ternary mode ('X' or do not care bits are allowed).
- Unencoded or encoded match address.
- Multiple match addresses allowed or not.
- Different initialization modes.
- Simultaneous read and write allowed or not.

CAM is a low latency memory, and there is no risk of collision (every new value has its memory position), but, on the other hand, it is very circuit consuming and stops working properly at high frequency. The main resources needed to implement the Xilinx Parameterizable Content-Addressable Memory described by [4], in the case of 256 tags of 60 bits (for Xilinx Zynq-7000 devices and with Vivado 2018.1), are summarized in Table 1. There are two versions, one using BRAM and the other, Shift Register LUT (SRL) as memory elements.

Table 1
Hardware resources for 256 tags of 60 bits CAM using XAPP1151 in a xc7z020clg484-1.

Resource	BRAM option		SRL option	
BRAM	48	17.0%	0	0%
RAM built LUT (LUTRAM)	120	0.7%	3840	22%
Lookup table (LUT)	723	1.4%	4597	8%

We have implemented both options, BRAMs and SRLs, for 64/128/256 tags, always of 60 bits. For 125 MHz frequency requirement, only BRAM implementation for 64 tags meets the timing requirements; thus, apart from resource problems, it is also more challenging to meet timing as memory capacity grows. These drawbacks necessitate the development of other solutions.

2.2. Hash table

A hash is any function that can map information of arbitrary size onto data of a fixed size. A hash table uses a hash function to compute an index into an array of buckets or slots, from which the desired value can be found [5].

The main advantages of hash tables are reduced circuit consumption and fast lookup, insertion, and deletion. The main problem arises with collisions, which appear when two elements produce the same hash output value; it should be avoided, but since collisions are intrinsic to a hash table, they decrease address table capacity and cause inefficient bandwidth utilization [5]. There are different ways to minimize collisions: using extra memory, double hash…; nevertheless, hash tables are overcome by CAM and binary trees in networks that critically require bandwidth dependability and robustness [5].

2.3. Binary search tree

Binary search trees [6] are a well-known type of data structure that allows faster lookup than the outdated linear or sequential search of $\mathcal{O}(n/2)$ delay. Keys are kept sorted to allow lookup, insertion, and deletion to exploit the basis of binary search. I.e., from the root, the tree is traversed to leaves: firstly, when the key and the value in the root are the same, the search has succeeded, and the initial node is returned. Otherwise, if it is smaller, the left subtree will be searched. On the other hand, if the key is greater, the right subtree will be searched. Consequently, each comparison lets the operations skip, on average, half of the tree. The procedure continues until finding the key or no more nodes remaining in the subtree. Binary search runs for logarithmic time in the worst case, making $\mathcal{O}(\log n)$ comparisons. Usually, the lookup algorithms require precomputation before searching the list [7].

Insertion and deletion are more complex operations since they require maintaining the in-order sequence of the nodes. The order in which insertions and deletions have been performed configures the binary tree's physical structure, which can become degenerate. Past a long random sequence of interlaced insertions and deletions, the height of the tree is expected to approach $\sqrt{n}$, the square root of the number of keys, which increases greatly faster than $\log n$. Hence, some methods have been introduced to balance the tree: to avoid any branch being more than one unit longer than the other ones [8]. Tries, red-black [9], AVL, and B-trees are the most popular self-balanced binary trees.

Consequently, insertions and deletions require certain algorithms to keep trees balanced. Furthermore, these trees can be ordered by more than one algorithm — plenty of them have been proposed. Finally, most of the algorithms may be used in more than one class of tree. [10] founded out that their technique offers the fastest lookup and occupies the least memory in the AVL tree.

The address lookup is a recurrent topic in packet data communication systems. Many times, the best solution is a combination of programmable hash algorithms, binary search ones, and a CAM, as it is explained in [11].

2.4. SoC technology

SoC devices allow building application-specific circuits in the PL and custom software in the PS, which is faster than solutions based on two chips, such as microcontroller and FPGA.

This work has been implemented in Xilinx UltraScale+ MPSoC architecture. This family of products integrates a feature-rich 64-bit quad-core or dual-core ARM Cortex-A53 and dual-core ARM Cortex-R5 based PS and Xilinx PL UltraScale architecture in a single device, where Input Output (IO) peripherals, on-chip memory, external memory interfaces, and high-bandwidth connectivity within PS and between PS and PL are also included.

The architecture presented in this paper takes advantage of both PS and PL modules since low latency tasks are performed in specific circuits (PL) and complex algorithms, such as sequential programs in the processor (PS). This is possible because both modules are tightly connected via Advanced Microcontroller Bus Architecture (AMBA) AXI4 interfaces.

2.5. AVL hardware implementations

One of the key points of this work is the implementation of the AVL algorithm in an SoC. This new technology provides a key advantage: a hardware/software design can be efficiently built.

There are multiple electronic examples of AVL trees in the literature and, in general, binary search trees. These implementations run on Central Processing Unit (CPU) [9,12], Graphics Processing Unit (GPU) [13,14], or FPGA [15–17].

It may be unfair to compare performances on such different architectures: CPU and GPU solutions are oriented to massive trees that cannot be synthesized in an FPGA. Although their latencies are more significant, they run at very high-speeds and may contain multiple cores. This is especially true for GPU approaches, where hundreds of cores can perform many searches in parallel.

However, in network processing, latency is a critical requirement [18]; thus, we will focus on FPGA implementations.

Senhadji-Navarro et al. [16] present a new architecture to implement a binary-tree-based Finite-State Machine (FSM) that overcomes 41 % speed of other similar state machines. Since it is arranged for one-bit input, its latency is equal to the key number of bits. The authors state a clock speed in the synthesis of 325 MHz; nevertheless, their binary trees are static: neither allow to insert new entries nor delete old ones.

Melikoglu et al. [19] present a deeply pipelined and massively parallel binary search tree accelerator for FPGAs. The design relies on the BRAMs architecture of FPGAs; to achieve significant throughput for the search operation, the authors introduce several novel mechanisms, including tree duplication as well as horizontal, duplicated, and hybrid (horizontal–vertical) tree partitioning. A Xilinx Virtex-7 VC709 is used as the electronic platform; since the application runs massive parallel searches and the memory is replicated, the hardware is intensively exploited. In addition, all the extra circuits lead to speeds between 142 MHz to 200 MHz, depending on the duplication. The authors do not state how much time is needed for each search but that it is faster than in a non-optimized tree. One shortcoming is that, even with lots of memory duplications, the worst case is the same as that of the single tree. However, the approach presented in this paper is focused on low and fixed latency applications; therefore, this kind of implementation is not valid.

Behdadfar et al. [15] present a new prefix lookup algorithm that leverages the prefixes as scalar numbers. This algorithm can be applied to different tree structures such as Binary Search Tree and some other balanced ones, such as RB-tree, AVL-tree, and B-tree. Nevertheless, the search, insert or delete procedures must be modified to make them capable of finding the prefixes of an incoming string, e.g., an IP address. They report ten clock cycles of latency with a synthesis clock period of 3.6 ns, inferior to the presented performance.

In [20], Qu et al. propose a scalable lookup engine on FPGA for large decision-trees; it supports high throughput, even if the tree is scaled up, concerning the number of fields and leaf nodes. The proposed engine is a 2-dimensional pipelined architecture that also supports dynamic updates of the decision-tree. Each leaf node is mapped onto a horizontal pipeline; each field of the tree corresponds to a vertical pipeline. The authors take advantage of dual port distributed RAM (distRAM) in each Processing Element (PE); the resulting architecture for a generic decision-tree accepts two search requests per clock cycle. Post place-and-route results show that, for a typical decision-tree consisting of 512 leaf nodes, with each one storing 320 bit data, the lookup engine can perform 500 Million Lookups Per Second. This leads to a latency of 2 ns, at the cost of 70 % of a Xilinx Virtex-7 xc7vx1140t. Hence, the throughput per slice is much poorer than ours.

Owaida et al. [17] present an FPGA tree ensemble classifier, together with a software driver, to efficiently manage the configurable internal memory. The classifier architecture efficiently utilizes the FPGA's resources to fit half a million tree nodes in on-chip memory, delivering up to 20x speedup over a 10-threaded CPU. The setup consists of an Intel's HARP v1, it is a two-socket machine with a 10-core Intel Xeon E5-2680 v2 CPU (clocked at 2.8 GHz) in one socket and an Altera Stratix V 5SGXEA in the other. The classifier fills up to 72 % FPGA at 200 MHz, ours 0.49 % at 400 MHz. This setup is focused on massive amounts of data on multiple trees. No latency is specified, but it is capable of running around 12.8 billion lookups per second. This implementation uses hardware and software, but they reside in two separate chips. Our approach exploits both of them in only one chip, reducing latencies and complexity.

3. AVL tree

Among the binary search trees, we chose the AVL one since it reaches a compromise between speed to search a key, complexity to insert and remove nodes, and the number of electronic resources demanded. It has been exploited in applications such as image analysis, patterns matching, including content access [21]; geoposition information [22], management of processor networks [13,14], key systems in security and authentication, encryption schemes [23], memory management, and test case generation [24].

To compensate for the length of the branches, regardless of the number of new nodes, AVL trees balance themselves: automatically keep their height (maximal number of levels below the root) minimum, despite arbitrary item insertions and deletions. In an AVL tree, the height difference [6,8] between the right subtree and the left one must differ in one, at most. This is called a *balanced tree* and follows (1):

$$BalanceFactor(N) \in \{-1;0;1\} \tag{1}$$

Read-only operations do not differ from an unbalanced binary search tree. On the other hand, modifications to the tree must restore the height balance of the sub-trees (see Fig. 1).

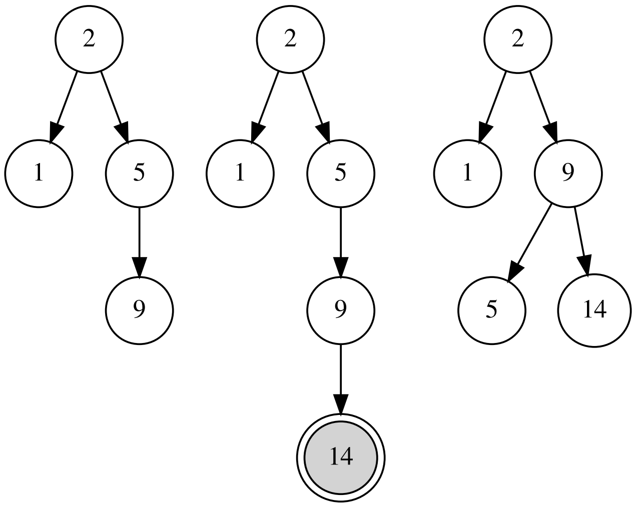

Fig. 1. From left to right. Balanced AVL tree. The insertion of "14" leads to an unbalanced tree. The tree balances itself.

Searching for a specific key in an AVL tree is done as with any binary search. The number of comparisons required for success is limited by the height (h) and for unsuccessful search is very close to h.

When inserting an element into an AVL tree, the process is the same as inserting it into a binary search tree. After this insertion, it is necessary to check the balance factor and the ancestors to decide whether the tree must be rebalanced. Deleting a node follows a similar pattern. After deleting the item, the tree must be retraced to rebalance it.

4. Description of the architecture

The AVL tree is implemented in two different modules: the search is performed in a hardware core, while the writing and update are done in software. The core receives the MAC addresses to be searched; in the case of destination addresses, it will send the search result to the frame processor. For the source addresses, missing ones will be sent to the software write and update module.

This latter block is implemented in the real-time processor of the MPSoC Xilinx FPGA, which is built around an ARM R5 (the MPSoC FPGA provides several processors; among them, the real-time ones are the most suitable for this task). As the implementation is pure software, it could be moved to another hard processor (A53) or to a soft one, such as Microblaze. This second module receives the destination addresses and needs to perform two operations:

- If not in memory, learn it.
- If in memory, update its age.

The operation can be described in the following steps:

- A destination MAC arrives at the core. It is searched and the result is sent to the following modules for packet switching.
- The corresponding source MAC of the packet arrives at the second port of the module and is searched. If found, the write core (*R5 processor*) is informed to update its age. If not found, the write core is informed to learn it.
- Periodically, the R5 processor performs a search in its copy of the AVL tree (*SW RAM*) to age the entries and, if appropriate, forget the corresponding one.
- In the case of aging and learning, if an entry is deleted or inserted, the R5 processor performs the write operations, including rebalancing.
- Once the software copy of the AVL tree is modified (*SW RAM*), it is mirrored into the working AVL tree RAM (*AVL RAM*).

4.1. Hardware architecture

The architecture, depicted in Fig. 2, is divided into several parts:

- *Search Core*. This circuit is in charge of finding the MAC in the AVL tree. It is fully hardware and designed for minimal latency — just one clock cycle, because the new values are registered and pipelined. It has two inputs, one for the search of MACs, while the other is for the learning and aging of MACs. In other words, the former is for the destination address (*MAC DST*), while the latter is for the source address (*MAC SRC*).
- *AVL RAM*. This is the memory that holds the AVL search tree. It is built using FPGA internal BRAM memory.

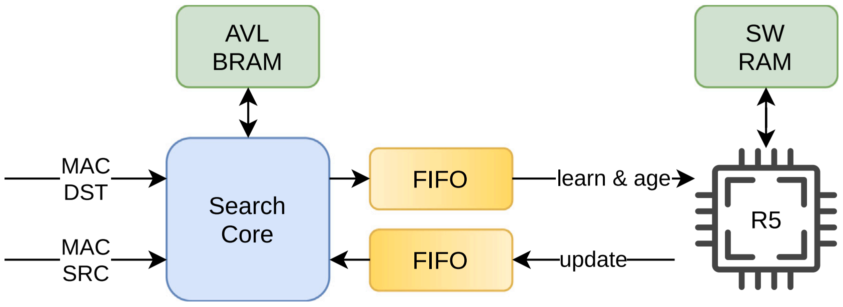

Fig. 2. Block diagram of the whole system.

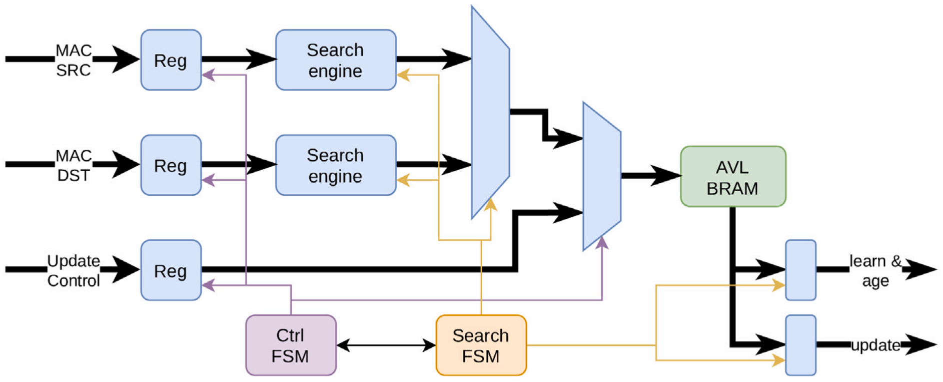

Fig. 3. Block diagram of the search core. Two search engines can work at the same time due to the BRAM latency.

- R5. This real-time, low latency processor receives learning commands from the read core through a First In, First Out (FIFO) and, using a copy of the tree residing in its memory (*SW RAM*), it can perform write operations. Afterward, these changes are mirrored to the actual memory (*AVL RAM*).

4.1.1. Search core

The search core is shown in Fig. 3. Considering that BRAMs inside Xilinx FPGAs have a minimum latency of one clock cycle, two searches can be performed simultaneously. While the result of one search is obtained from the memory, the address of the other is inserted. Thus, the read operations are pipelined in the memory. This way, a new value every clock cycle, i.e., the latency is one. The circuit is basic (a comparator to decide whether to take the left or right branch) but has been done in detail to allow both searches simultaneously. Apart from the search engine, one state machine is in charge of the search operations, while the other is in charge of only allowing changes when appropriate and doing them atomically (uninterruptible series of operations). The state machine is also in charge of sending the output result to the appropriate output port.

This basic structure is complemented with all the required interface connections to make it AXI compatible. This enables an easy interaction with the rest of the circuit.

The main ports are:

- SEARCH_A. 64 bit AXI-Stream for source address search.
- SEARCH_B. 64 bit AXI-Stream for destination address search.
- SEARCH_Ctrl. 128 bit AXI-Stream that carries the memory update commands.
- RESULT_A. 64 bit AXI-Stream with the result of the search in channel A.
- RESULT_B. 64 bit AXI-Stream with the result of the search in channel B.

The search core input ports' data format is the Ethernet input port and MAC address's concatenation. The data width is increased to 64 bit to match standard AXI-Stream formats.

The control input port's data format comprises the memory address to write, memory content, operation type, and whether it is the last operation to be performed atomically.

Both result ports are made up of a match flag (to know whether the input data have been found), a copy of the input data, and its associated value (output port).

It is worth mentioning that, since both search paths are equal, they could be taken to search two destination addresses simultaneously. The reason to process source addresses is to make the processor load lighter when several tables are used in the FPGA.

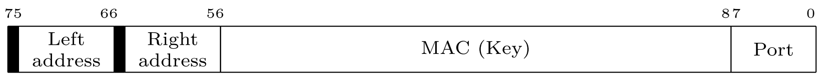

Fig. 4. Data organization in the memory. The length depends on the size of the memory available for the tree and on the number of ports. In this example, the address bus has 9 bits (512 positions) while the port width is 8 bits. Bit 65 tells us whether the right address is real or NULL. Bit 75 does the same for the left address (if the address is negative, it is invalid).

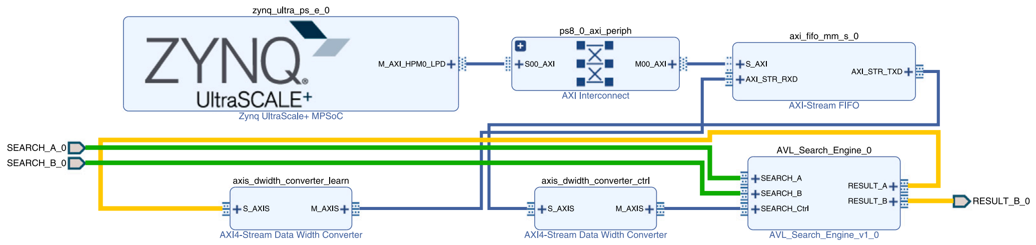

Fig. 5. Block diagram of the update hardware/software engine. This block is in charge of sending the information from the core to the R5 and the learning result from the R5 to the core.

4.1.2. AVL RAM

In this memory, the tree is stored, and all hardware searches are performed inside. BRAM in FPGAs may be set in dual-port read/write configuration. In this way, by doubling the search core, this architecture would perform two simultaneous searches, doubling the performance.

If both ports of the true dual-port were to be used, it would require more resources. The main reason is that, due to the wide data path, more memories are needed. In this case, both ports are being used but to perform the same search, resulting in lower resource utilization. In other words, we are concatenating both ports to generate a bigger single port BRAM.

First and foremost, the left address, the right one, the key, and the associated value must be stored in the memory. Simultaneously, the system needs to know whether the left and right addresses point to correct values or are invalid. In a pure software application, they would point to NULL (a value saved for indicating that the pointer or reference does not bring a valid object); in this case, if the stored value is negative, the address is invalid. This configuration can be seen in Fig. 4.

The left and right addresses require 9 bits plus one extra for the sign; in hardware, this is simplified to checking the most significant bit. The memory of 512 positions also stores the key — in this case, a 48 bit MAC address — and the value associated with it. Since the tree is being exploited for switching applications, the output port could be stored. In this example, we have left 8 bits for the data.

4.1.3. Processor (R5)

As mentioned, both cores interchange information through the AXI-Stream interface. Periodically, the R5 processor will age the table and will purge old records. The AVL search engine provides a MAC, whether it has been found and, if so, the port. Upon receiving this information, the R5 processor performs several AVL tree-related operations. If not found, it will learn the MAC address. If so, it will update its age counter.

In both cases, if the table is modified, it will send the change commands to the AVL search engine. These may be of two types, selectable employing one bit in the command:

- Root address change. This does not change the memory but the initial address to begin the search.
- Memory update. This performs a memory change. The command is composed of the address to be changed and the new value.

In both cases, an extra bit indicates whether the atomic operation has ended. The main reason is not to leave the tree in an unusable state. All atomic operations are performed serially and cannot be stopped.

A state machine inside the AVL search core is in charge of avoiding collisions. If a search is being performed, no update can be done in the memory; if not, it can be updated. Even if search requests arrive, an atomic operation is not stopped.

4.2. Platform and operation description

The overall system is depicted in Fig. 5. The main components are:

- AVL search engine as described in this paper.
- AXI Stream FIFO. This Intellectual Property (IP) core converts AXI-4 Lite read/writes from the processor to AXI-Streams data towards the AVL search engine.
- AXI Stream data width converter. The AVL search engine processes very wide data, while the R5 has a 32 bit data bus. This block performs this conversion.

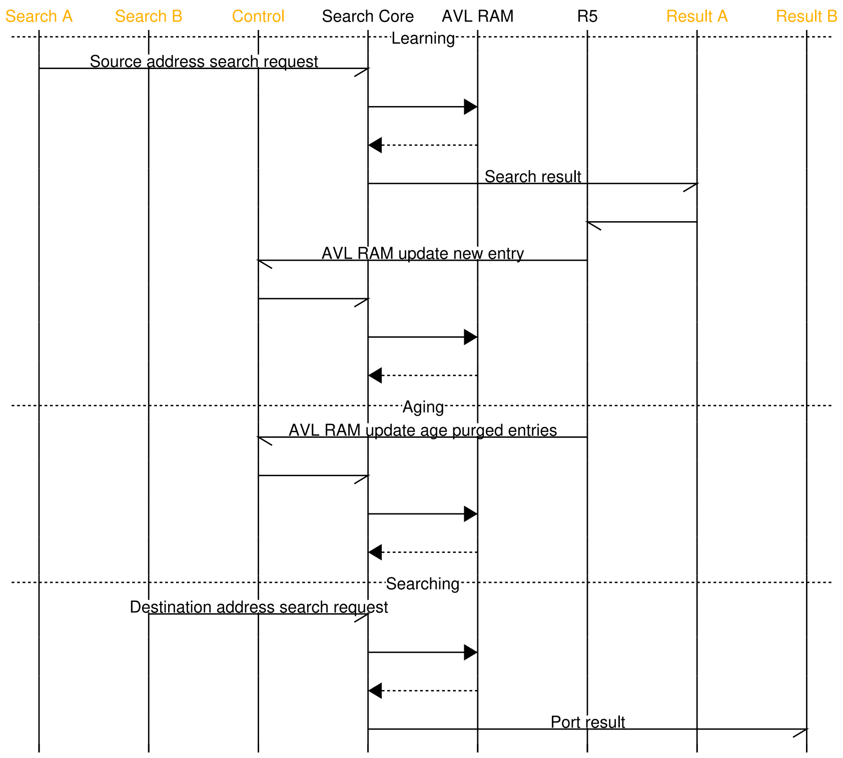

Fig. 6. Learning, aging, and searching in action (orange names denote AXI interfaces). First, a new entry is learned. It is searched, and the result is passed to the processor. Since it is a new entry, the processor will update each internal tree and send the update commands. Then, aging: as source addresses arrive, their age is updated. Periodically, all entries are purged. Lastly, a destination address search: the result is passed to the frame processor.

- MPSoC. This is the processor block in the FPGA. Among others, it hosts two R5 real-time processors. One of them is in charge of the AVL tree.

The main steps of the process diagram, shown in Fig. 6, are:

- **learning:** Source addresses arrive through `Search A`. The MAC address is searched in the *AVL RAM* (*Source address search request*), and the result (*Search result*) is passed to the processor (R5) by means of the `Result A` AXI interface. Once the result of the search arrives at the processor, two different options arise:
  - If the entry was not found in the *AVL RAM*, it is inserted in the internal copy of the tree (*AVL RAM update new entry*) at the processor, and the resulting tree is transferred, decomposed in atomic operations, to the *Search Core* through the `Control` AXI interface.
  - If the entry is in the memory, its age is updated.
- **aging:** Whenever a *learning* process is performed, age is refreshed. On the other hand, a periodic task loops through the processor copy of the tree, aging it. Meanwhile, some entries may be purged. The resulting tree is written back to the *AVL RAM* (*AVL RAM update age purged entries*).
- **searching:** Destination addresses arrive through `Search B` interface (*Destination address search request*). They are looked up in the RAM, and the result is passed through `Result B` to the frame processor (*Port result*).

It must be noted that the source address search and the destination one are done at the same time in the *Search Core*. The timing diagram in the different interfaces can be seen in Fig. 7.

The BRAM memory inside the FPGA has a latency of one clock cycle. This allows two searches at the same time. While the memory outputs the value of the address memory searched for a source, a destination address search is in the memory address bus.

Updates and searches cannot be performed concurrently since the tree could be in an unlawful state. Therefore, updates are done when no searches are in progress and atomically.

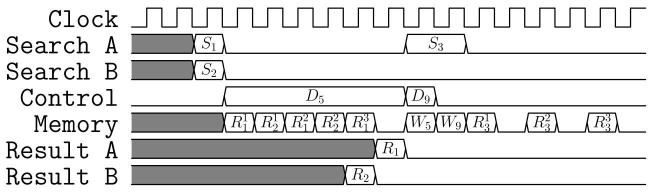

Fig. 7. Timing diagram of the sequential searches. Two searches are initiated at the same time (S_1 and S_2) from `Search A` and `Search B` ports. The memory is accessed sequentially with data from the first and the second searches. While the search is in progress, an update command (D_5) arrives through the `Control` port. It is stalled until all searches are finished. At this moment, the write operations begin (W_5). Even if a new search arrives (S_3), it is not served until all the atomic writes are performed. Then, the third search begins.

Table 2
Hardware resources for the AVL search core (synthesis) in a `xczu6eg-ffvc900-1-i`.

Resource	Utilization	Available	%
LUT	176	214604	0.08
Flip-flops	259	429208	0.06
BRAM	1.5	714	0.21

Table 3
Hardware resources for the whole system (synthesis) in a `xczu6eg-ffvc900-1-i`. The block diagram is depicted in Fig. 5.

Resource	Utilization	Available	%
LUT	1629	214604	0.76
LUTRAM	131	144000	0.09
Flip-flops	1920	429208	0.45
BRAM	3.5	714	0.49

5. Timing and resources

5.1. Timing

Having been optimized, the AVL Search core can work at frequencies over 250 MHz, allowing two operations of 1 Gigabit Ethernet (which uses a 125 MHz clock signal). Synthesis results offer working frequencies of 400 MHz for the core.

The amount of memory is critical since it determines the maximum height of the tree (h) [6]. Both, the depth of the RAM (RAM_{depth}) and the height of the tree (h), are related by (2):

$$RAM_{depth} = 2^{h+1} - 1 \tag{2}$$

As the search is performed sequentially, the delay in finding the result is variable, although the maximum latency is proportional to h. The larger the memory, the greater h gets, and, therefore, the longer the maximum latency of the circuit is. The single clock cycle latency of the memory also increases the latency of the system since searches for the same path are performed every two clock cycles. By and large, the overall maximum latency, in clock cycles, can be expressed using (3):

$$latency_{\max} = 2 \cdot h + 1 = 2 \cdot \log_2(RAM_{depth} + 1) - 1 \tag{3}$$

5.2. Hardware resources

Table 2 holds the results for the AVL search core, post-synthesis, and using a single port RAM (due to the large data width, 1.5 BRAMs are used: 36 kb+18 kb). By means of both ports of the BRAM to make a broader data bus, fewer resources are required. These represent less than 0.25% of the FPGA and can be replicated as needed to accommodate all the ports of the system. Although the FPGA may look too big for this core, it must also host an Ethernet switch that requires many resources, depending on the number of ports and capabilities.

In Table 3, the overall results can be seen; the resources increase significantly, mainly due to the AXI infrastructure. It must be noted that this infrastructure can be reused for other cores.

6. Validation

The proposed architecture has two different aspects:

Table 4
Comparison of single leaf search in various implementations. Latency and clock cycles per search are presented.

Method	Hardware	Clocks/search	Latency
This work	FPGA	1	4 ns
Qu [20]	FPGA	1	2 ns
Melikoglu [19]	FPGA	1	5 ns
Zhou [13]	GPU		15 ns
Behdadfar [15]	FPGA	10	36 ns
Saikkonen [12]	CPU	83	38 ns
Shekar [14]	GPU	29	43 ns
Howard [9]	CPU	323	147 ns
Senhadji [16]	FPGA	48	147 ns

Table 5
Comparison of area resources for different FPGA implementations. Many do not state any data.

Method	FPGA	Max %	Flip-flops
This core	xczu6eg	0.21	259
This system	xczu6eg	0.49	1920
Qu [20]	xc7vx1140t	70	≈ 797440
Melikoglu [19]	xc7vx690t	–	–
Behdadfar [15]	xc6vlx75t	–	–
Senhadji [16]	xc6slx75	–	500
Owaida [17]	5sgxea	72	81472

- Write operations: done in software.
- Read operations: done in hardware.

In the case of writing, the software has been executed in the processor of the proposed SoC. The validation has been performed by running the same data set in both the embedded software and PC program. For the latter, an open-source version has been used [25]. In the embedded version, since the tree resides in the memory of the PL, a static memory allocation version has been developed.

Read operations have been emulated, according to the write operations described before. The *AVL RAM* has been filled with the results obtained from the different write operations, and data have been searched. The result of the searches has been compared with a software performed search. Fig. 8 shows read operations at the core levels. One interesting point is the continuous stream of output data from the memory. As described in Section 4, the IP has been designed to obtain the memory's maximum throughput.

Fig. 9 shows AXI transactions running. The key element in the AXI specification is the `VALID` and `READY` handshake. Data are only transferred when both are '1'.

Due to this two-step approach, and the lack of a real system to test, power efficiency data are preliminary: the estimation for the full hardware system is 2.896 W. If only the PL is considered, the power goes down to 0.057 W. This is due to the very efficient implementation that demands minimal resources, as Table 3 shows.

7. Comparison

To the best of our knowledge, no hardware/software implementation of the AVL engine exists on a single chip. Moreover, the use of AVL for MAC address resolution is also new.

To compare the architecture presented in this paper with others present in the literature, it must be highlighted that data insertion, update, and deletion are made in a pure software environment. Hence, any AVL algorithm from the scientific community could be exploited. In addition, any binary search algorithm may be used since the hardware is tree agnostic. On the other hand, due to the processor and configurable logic being hosted in the same chip, this architecture, as opposed to others, performs the search in hardware. From the information we have collected, this is the first time this kind of chips have been used, although FPGAs for binary tree searches are not new, as mentioned in Section 2.5. In Table 4, this architecture is compared with other implementations made in different kinds of hardware (FPGA, GPU, and CPU).

As can be seen, the architecture presented in this paper spends the fewest clock cycles per search. When normalized for different clock frequencies, this proposal also performs in a very promising way. The implementation presented in [20] provides lower delays, but, as commented, at the cost of occupying the whole FPGA, without a place for the other modules in the node. Table 5 shows a comparison of different implementations.

Since many authors do not state the occupation or number of employed elements, sometimes, only overall occupation is mentioned, and flip-flop data have been extrapolated. In all the cases, it is unclear whether only the search element or the overall system has been considered for occupation data; thus, both results are added in our implementation.

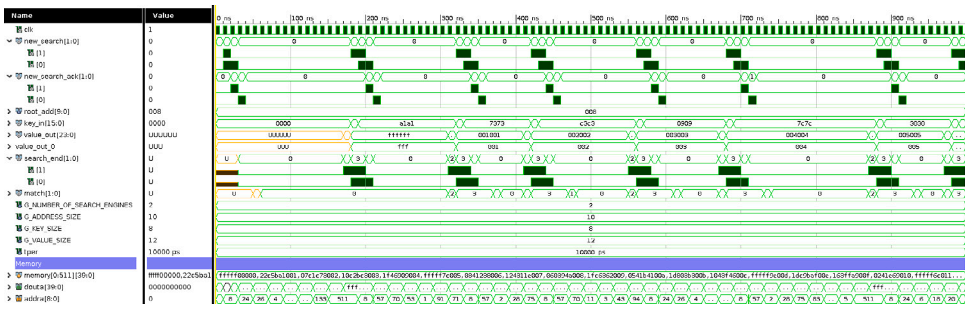

Fig. 8. Internal emulation of the core.

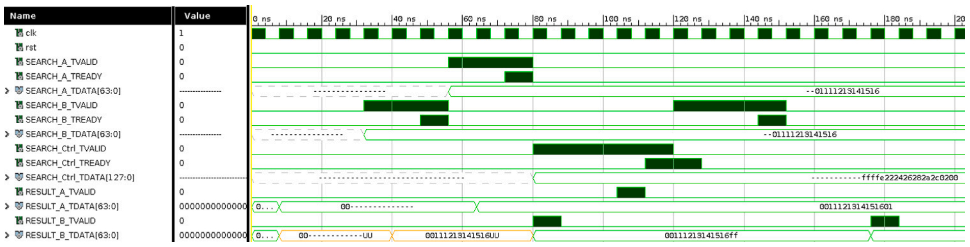

Fig. 9. External emulation of the core.

Our approach is focused on having minimal area impact on the rest of the system. It must also be highlighted that the BRAM has been optimized to reduce the area. If optimized for speed, the proposed algorithm could be comparable to [20], at the cost of doubling the resources. Although the system can be clocked at higher than 250 MHz, this frequency has been chosen because it is multiple of the input data rate, greatly simplifying the rest of the circuit.

GPU implementations are also noticeable since their higher clock frequencies and the massive number of cores make them very useful in these applications. Additionally, their latency is relatively low and comparable to those in FPGAs.

8. Conclusions

This paper presents a hybrid hardware/software search architecture based on an AVL tree for MAC searching, learning, and aging. It perfectly suits the internal architecture of modern SoC FPGAs that contain both a programmable part and processing cores.

The AVL tree performs searches in an easy and fast way by hardware. They are scalable and require few resources. On the other hand, write and delete operations are complex, but a hard microprocessor can perform them. The presented architecture can do fast searches in hardware and write and delete operations in software by mixing both approaches.

One drawback is the AXI resources required. This infrastructure demands some hardware but may be reused for other IP cores presented in the design.

In any case, this core can perform two searches per 1 Gbps channel, and implementing it in an FPGA allows us to update the design efficiently, according to new standards or versions.

CRediT authorship contribution statement

Jesús Lázaro: Conceptualization, Software, Writing - original draft, Writing - review & editing, Supervision. **Unai Bidarte:** Validation. **Leire Muguira:** Investigation. **Carlos Cuadrado:** Software. **Jaime Jiménez:** Methodology, Writing - review & editing.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

References

[1] International Electrotechnical Commission. IEC 624393:2012 industrial communication network - high availability automation networks - Part3: Parallel redundancy protocol (PRP) and high-availability seamless redundancy (HSR). 2012.

[2] Wu X, Xie L. Performance evaluation of industrial Ethernet protocols for networked control application. Control Eng Pract 2019;84:208–17. http://dx.doi.org/10.1016/j.conengprac.2018.11.022.

[3] Ullah Z. LH-CAM: Logic-based higher performance binary CAM architecture on FPGA. IEEE Embedded Syst Lett 2017;9(2):29–32. http://dx.doi.org/10.1109/LES.2017.2664378.
[4] Locke K. XAPP1151: parameterizable content-addressable memory. Technical report, Xilinx; 2019, URL https://www.xilinx.com/support/documentation/application_notes/xapp1151_Param_CAM.pdf.
[5] Huntley C, Antonova G, Guinand P. Effect of hash collisions on the performance of LAN switching devices and networks. In: Proceedings. 2006 31st IEEE conference on local computer networks; 2006. p. 280–4. https://doi.org/10.1109/LCN.2006.322112.
[6] Knuth D. The art of computer programming. Reading, Mass: Addison-Wesley Pub. Co; 1973.
[7] Waldvogel M, Varghese G, Turner J, Plattner B. Scalable high speed IP routing lookups. SIGCOMM Comput Commun Rev 1997;27(4):25–36. http://dx.doi.org/10.1145/263109.263136.
[8] Haeupler B, Sen S, Tarjan RE. Rank-balanced trees. ACM Trans. Algorithms 2015;11(4):30:1–26. http://dx.doi.org/10.1145/2689412.
[9] Howard PW, Walpole J. Relativistic red-black trees. Concurr Comput: Pract Exper 2013;26(16):2684–712. http://dx.doi.org/10.1002/cpe.3157.
[10] Sun Q, Zhao X, Huang X, Jiang W, Ma Y. A scalable exact matching in balance tree scheme for IPv6 lookup. In: ACM SIGCOMM 2007 data communication festival; 2007. URL http://www.cu.ipv6tf.org/pdf/1569043111.pdf.
[11] Spinney BA. Address lookup in packet data communications link, using hashing and content-addressable memory. 1993, US Patent US5414704A.
[12] Saikkonen R, Soisalon-Soininen E. Cache-sensitive memory layout for dynamic binary trees. Comput J 2015;59(5):630–49. http://dx.doi.org/10.1093/comjnl/bxv090.
[13] Zhou S, Singapura SG, Prasanna VK. High-performance packet classification on GPU. In: 2014 IEEE high performance extreme computing conference. IEEE; 2014, http://dx.doi.org/10.1109/hpec.2014.7041005.
[14] Shekhar A, Goyal J. Parallel binary search trees for rapid IP lookup using graphic processors. In: 2nd international conference on information management in the knowledge economy; 2013. p. 176–9. URL https://ieeexplore.ieee.org/document/6915094.
[15] Behdadfar M, Saidi H, Hashemi M-R, Ghiasian A, Alaei H. IP Lookup using the novel idea of scalar prefix search with fast table updates. IEICE Trans Inform Syst 2010;E93-D(11):2932–43. http://dx.doi.org/10.1587/transinf.e93.d.2932.
[16] Senhadji-Navarro R, Garcia-Vargas I. High-performance architecture for binary-tree-based finite state machines. IEEE Trans Comput Design Integrated Circ Syst 2018;37(4):796–805. http://dx.doi.org/10.1109/tcad.2017.2731678.
[17] Owaida M, Zhang H, Zhang C, Alonso G. Scalable inference of decision tree ensembles: Flexible design for CPU-FPGA platforms. In: 2017 27th international conference on field programmable logic and applications. IEEE; 2017, http://dx.doi.org/10.23919/fpl.2017.8056784.
[18] Park T, Shin S, Shin I, Lee K. Formullar: An FPGA-based network testing tool for flexible and precise measurement of ultra-low latency networking systems. Comput Netw 2021;185:107689. http://dx.doi.org/10.1016/j.comnet.2020.107689.
[19] Melikoglu O, Ergin O, Salami B, Pavon J, Unsal O, Cristal A. A novel FPGA-based high throughput accelerator for binary search trees. 2019, URL https://arxiv.org/abs/1912.01556.
[20] Qu Y, Prasanna V. Scalable and dynamically updatable lookup engine for decision-trees on FPGA, cited By 1. In: 2014 IEEE high performance extreme computing conference; 2014. https://doi.org/10.1109/HPEC.2014.7040952.
[21] Wang D, Yeo CK. Superchunk-based efficient search in P2P-VoD system. IEEE Trans Multimed 2011;13(2):376–87. http://dx.doi.org/10.1109/TMM.2011.2106485.
[22] Bai L, Yan L, Ma Z. Interpolation and prediction of spatiotemporal data based on XML integrated with grey dynamic model. ISPRS Int J Geo-Inf 2017;6(4). http://dx.doi.org/10.3390/ijgi6040113.
[23] Reddy KS, Ramachandram S. A novel dynamic order-preserving encryption scheme. In: 2014 first international conference on networks soft computing; 2014. p. 92–6. https://doi.org/10.1109/CNSC.2014.6906720.
[24] Pham LH, Le QL, Phan Q-S, Sun J, Qin S. Testing heap-based programs with java StarFinder. In: Proceedings of the 40th international conference on software engineering: Companion proceeedings. New York, NY, USA: ACM; 2018, p. 268–9. http://dx.doi.org/10.1145/3183440.3194964.
[25] Thompson T. A quick AVL tree implementation in c. 2011, URL https://gist.github.com/tonious/1377768.

Jesús Lázaro is a Full Professor at the Department of Electronics Technology of the University of the Basque Country. He is the author or co-author of 4 patents, 35 articles in international scientific. His main research areas are high-speed circuits based on reconfigurable devices and communications devices.

Unai Bidarte is, since 2009, an Associate Professor at the Department of Electronics Technology of the University of the Basque Country. He is co-author of 3 patents, more than 10 papers in international magazines and more than 60 contributions to other magazines, and conferences.

Leire Muguira is, since 2018, a Researcher and Lecturer at the Department of Electronics Technology of the University of the Basque Country. She has participated in 8 competitive research projects supported by public institutions. She is the author of 5 articles in international scientific magazines. Her main research areas are high-speed circuits based on reconfigurable devices and communications devices.

Carlos Cuadrado is, since 1999, a Researcher and Lecturer at the Department of Electronics Technology of the University of the Basque Country. He is the author or co-author of 9 articles in international scientific magazines. His main research areas are high-speed circuits based on reconfigurable devices, digital control architectures, and communications devices.

Jaime Jiménez is, since 1998, a Researcher and Lecturer at the Department of Electronics Technology of the University of the Basque Country. He is the author or co-author of 23 articles in international scientific magazines. His main research areas are high-speed circuits based on reconfigurable devices and communications devices.