

The Role of Memory in NPU System Design

White Paper

This paper examines the role of memory in network processor system design as it relates to the table look-ups that must be performed for network processing protocols and outlines a few application examples.

ABSTRACT

Packet processing and classification are the two main functions of network processing. Normally two chip sets are required, one for processing and a second for classifying. The network processor (NPU) accounts for one to four chips, depending on whether the NPU is half or full duplex and whether it integrates traffic management. Classification consists of multiple chips, the number and type of which vary, according to the specific application for which the system is designed.

Most network processor solutions require external classifiers based on a combination of CAMs and SRAMs. For many applications, these can account for a dozen or more chips, over 50 watts in power dissipation and thousands of dollars in costs. Consequently, classification exercises severe constraints on key line-card design aspects and impacts the overall system's competitiveness. It directly and adversely affects board size, board complexity, cooling and air flow sub-systems, selection of power supplies and chassis size. It furthermore lacks the flexibility to scale classification needs for changing market requirements.

An alternative solution is to use DRAMs for classification tables, instead of the CAMs and SRAMs. DRAM offers 28 times the density of the largest CAM or SRAM (512 Mbit versus 18 Mbit), 250 times lower per-bit power dissipation (0.001W versus 0.25W per 1 Mbit) and 800 times lower per-bit cost (\$0.01 versus \$8 per 1 Mbit). DRAM, therefore, can reduce the overall number of chips required for processing and classifying, the overall power dissipation and cost. Its high density also provides for large classification memory headroom for future growth.

However for an NPU to take advantage of DRAM, it must overcome some key technology barriers. The NPU must provide the search functionality, since DRAMs are simple memory elements without any embedded logic. It must also overcome DRAM's higher latency and lower utilization as compared to CAMs and SRAMs.

The use of DRAM requires technology enablers for integration of the classification functions into the NPU, high-bandwidth embedded memory and unique search algorithms that reduce the number of memory accesses required to complete table look-ups.

DIFFERENT TYPES OF LOOK-UPS

Three major types of look-ups are used in networking applications:

- Direct table look-ups where the key is the index of the entry in the table. These are typically used in smaller tables for input port, VLAN ID-based look-ups and where the key size is 2 bytes or less.
- Hash table look-ups are practical where the key size is longer but fixed in length. Typically used for 5-tuple flow look-ups, MPLS tag look-ups, and MAC address look-ups.
- Tree based look-ups. Binary tree look-ups are for IP longest prefix match, or Access Control List (ACL) with first match policy. Character tree look-ups include URL strings for load balancing in web-switching applications.

All three types of look-ups need to be supported by the DRAM-based look-up engine.

DRAM-BASED LOOK-UPS: THE MOTIVATION

The motivation is obvious. DRAM provides a high-capacity and low-cost medium for storing tables for networking applications. The deployment of IPv6 capable equipment is also increasing by 4-fold the size of tables required, due to the longer IPv6 address size, and intensifies the need to migrate away from NPU designs based on external CAM and SRAM.

The following chart outlines the cost involved with external classification engines based on CAMs and SRAMs in four mainstream applications:

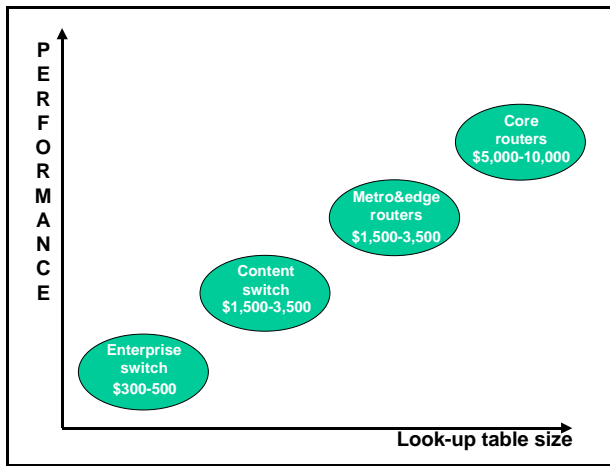


Figure 1. Cost of CAMs, SRAMs and classifiers¹

The cost of the memory subsystem required by the NPU can vastly exceed the cost of the NPU itself.

In addition, the number of CAMs, SRAMs and co-processors is directly related to a specific application. Any increase in table size for a new application will require modification in the line card's hardware. Deploying line cards with significant table space headroom would greatly increase their life-span once deployed in the field, but given the cost of the traditional CAM, SRAM-based memory subsystems, this is impractical.

Another advantage of DRAM is in character tree look-ups for load balancing and content switching applications. Since CAMs are fixed width, they are impractical to use for this type of look-up.

One last advantage of DRAM versus CAM is during database updates. With DRAM, there is no need to push entries down to create space in a given priority range, unlike with CAMs.

DRAM-BASED LOOK-UPS: THE ISSUES

Today's memory technology does not allow the development an NPU design that solely relies on external memory for table look-ups. A 200 MHz 64-bit DDR FCRAM provides a raw bandwidth of 25 Gb/s. Increasing available bandwidth by supporting multiple external memory interfaces is problematic given the vast number of pins needed by the NPU for its other interfaces (Host CPU, framer interface, switch fabric interface).

The budget for a 64-byte minimum size Ethernet packet on a 10 Gb/s interface is very small, around 67 ns, which given a 200 MHz clock cycle would provide

around 13 clock cycles (5 ns per cycle) for look-up processing.

Consider the case of performing a longest prefix match for an IP address. Finding a match in a tree with 128K entries typically involves accessing the memory 17 times ($\log_2 128K=17$), which already exceeds the maximum budget for the memory clock cycles.

Fortunately, embedded memory technology offers a complementary solution. Typical bus widths ranging from 256 to 512 bits at a clock rate of 200 MHz provide 50 to 100 Gb/s of available bandwidth. A combination of embedded memory and external is needed to achieve the type of performance look-ups that is needed for a 10-Gigabit networking application.

Designing an NPU with fast external memory access and as much embedded memory as physically possible is not enough. It is necessary to integrate the look-up engine into the NPU design to handle memory accesses in a pipeline and deal with the memory contention issues as efficiently as possible. Maximizing the memory utilization is crucial.

With a standard RISC-based NPU, it is up to the software running on the NPU to implement the look-up algorithms (tree or hash-based look-ups) and deal with the memory latencies as well as contention to the memories. Essentially, the software performing the look-up has to be "threaded" on itself to perform multiple look-ups in parallel and take advantage of the "delay cycles" accessing to the memories. But dealing with memory contention is not possible and the programmer must estimate the number of idle cycles on average to fill in the "delay cycles" with something useful. This system cannot take into account the contention issues which decrease or increase the number of delay slots for a given memory access.

Another possible solution is to leave the look-up to the software running on the NPU, but provide a thread environment in hardware that suspends the software while doing a memory access and lets another thread take account during the delay. Here the difficulty is to calculate the optimal number of threads to implement in a given CPU engine to maximize efficiency while minimizing the threads' context switching overhead. The complexity of thread-model programming and the inter-locking mechanisms between the threads are also relevant issues.

In both these cases, where the software drives the look-up engine, it is extremely difficult to maximize the utilization of the look-up memory. Even when using high-performance CAMs, the issue of how to maximize utilization of the look-up memory remains and is dependent on the architecture of the NPU's packet processing unit.

¹ CAM used for calculations is 9-Mbit/\$200 (configurable 256Kx36bit,...16Kx576bit), SRAM used is DDR or QDR 18-Mbit/\$50.

DRAM-BASED LOOK-UPS: THE SOLUTION

Given all these possibilities, the next alternative with higher integration is to embed the entire look-up engine into the NPU and essentially provide a “look-up” instruction where the programmer specifies the key and gets the result. The entire look-up process – the look-up algorithm (tree, hash, direct table), the access to the memories and the arbitration to these memories – is dealt with by the hardware look-up engine. By making a tight coupling in between the look-up engine and the look-up memory, much better memory utilization is achieved, which is important when the look-up memory is DRAM-based.

Even when providing such a hardware look-up engine, programming flexibility is still needed for “chained” look-ups where the result of a first look-up is used by itself, or is perhaps recombined with a field from the packet, to form a new key for performing a subsequent look-up. This can be achieved through a dedicated engine that supports a very minimal number of macro-level instructions to perform the look-up, recombine pieces of data into keys, and perform some conditional tests and jump to the next look-up based on the result of previous a look-up.

So, the integrated look-up engine as part of the network processor would look like this (see Figure 2):

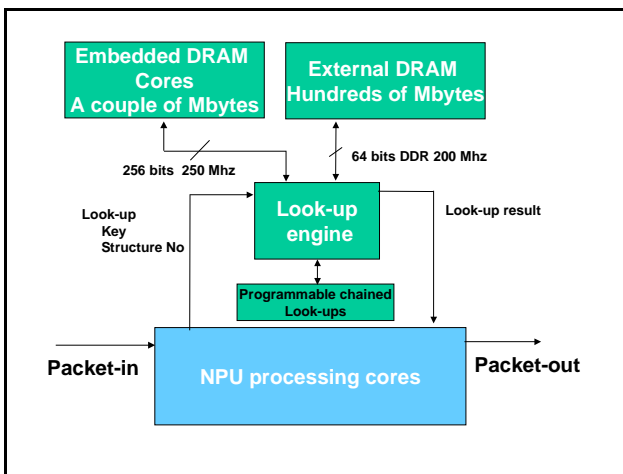


Figure 2. NPU with integrated look-up engine architecture

The packet processing code specifies a key or multiple keys for the look-up and gets the result back from the look-up engine.

The advantage of embedding the look-up engine within the NPU becomes more obvious with long look-ups for layer-7 applications like web switching, URL load balancing, and intrusion detection, where the key length for the look-up can be up to the frame length. For this case, the look-up engine ideally passes a pointer to the location in the packet where the key

begins and accesses the frame directly within the frame memory (see Figure 3).

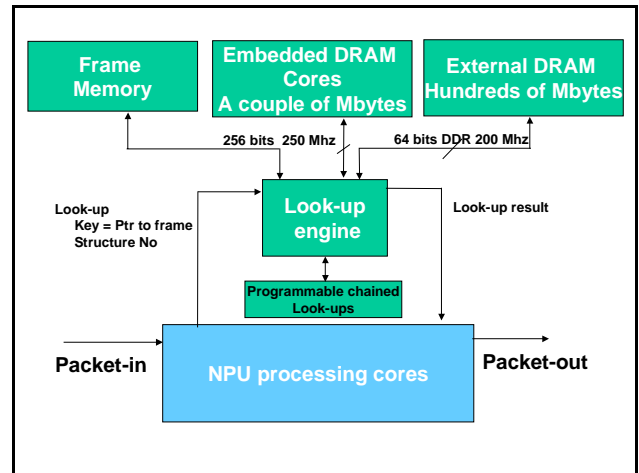


Figure 3. Long look-up support

The NPU integrates several of these look-up engines into the processing pipeline (see Figure 4). Each packet processing core works on a single packet and prepares the keys to pass to the look-up engine. A look-up engine dispatcher assigns one look-up engine to work on a key and perform the look-up. So, multiple packets are being processed through the look-up engine at any given time. The results for each packet is retrieved and passed to a specific processing core for post-lookup processing.

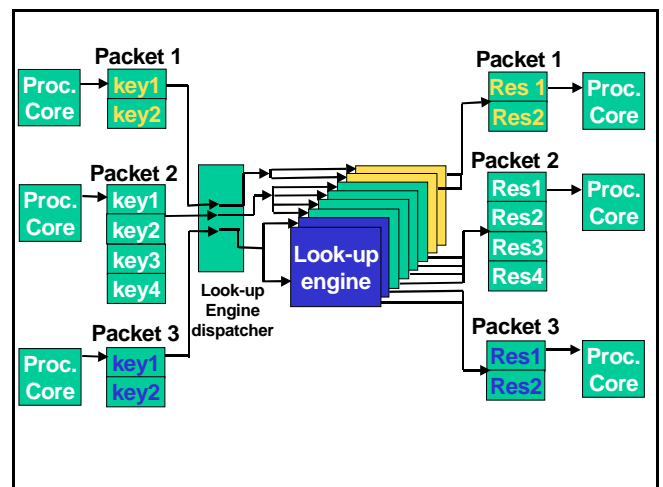


Figure 4. Look-up engine interaction within the processing pipeline

The simplicity of the programming model should be particularly noted. For a simple look-up, the packet processing core simply prepares the keys and the results are passed back to post-lookup processing cores without any programming necessary. For a complex (“chained”) look-up, a very small program within the look-up engine consisting of a few instructions needs to be written for each look-up in the chain. This requires hardly any performance tuning from the software point of view. This offers a

tremendous advantage over NPU architectures where the look-up is done through software programming and requires a lengthy tuning process.

Given this architecture, let's now take a look at direct table, hash and tree look-ups and map them into specific sample applications.

DIRECT TABLE LOOK-UPS

This is the simplest case. The key is the index of the entry within the table. A single memory access is needed to retrieve the data. No look-up algorithm needs to be implemented. Performance is purely a factor of the memory bandwidth.

HASH LOOK-UPS

The main issue with hash look-ups in a standard implementation is their non-determinism, due to the collisions on a given key. Once a collision occurs, then a linear search is performed to get the correct entry, which causes more memory accesses to get the entry that matters and increases bandwidth demand on the table memory by having to look at entries that don't matter.

The hardware look-up engine must provide a mechanism to resolve the search into a hash table in a predictable manner so that performance can be guaranteed. This mechanism must also provide the look-up performance in a predictable manner independent of the table size.

It is possible to provide a mechanism that guarantees to find the result of a hash look-up within two memory accesses, but no less. In the following example, such a mechanism is assumed.

EXAMPLE OF AN INTENSIVE HASH LOOK-UP APPLICATION: VPLS

Layer 2 VPNs implemented with "Draft Martini"² are becoming increasingly popular to provide Ethernet connectivity between remote sites over an MPLS packet switch network. Draft Martini specifies a Point to Point service and VPLS allows the interconnection of multiple sites.

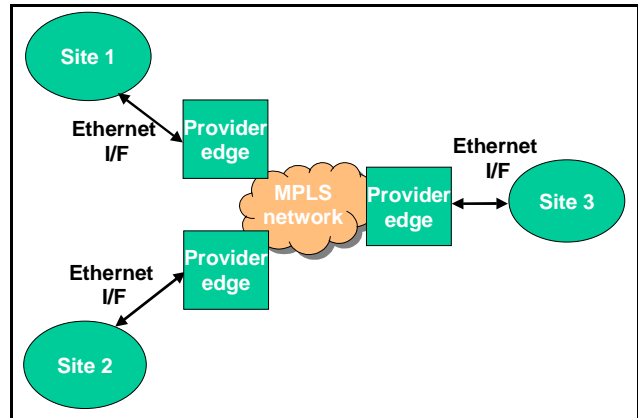


Figure 5. Draft Martini MPLS network

On the Ingress LSR (Encapsulating node), in the common path, a look-up of the Destination MAC address is performed in a MAC hash table, and its result is used to find the correct tunnel to send the packet on. In this direction, a single hash look-up is needed and the rest of the table look-ups are in direct tables.

In the Egress LSR (Decapsulating node), four hash look-ups have to be performed: (1) to determine whether the Tunnel exists, (2) to get the Martini VC characteristics and send it out to the correct Ethernet port, (3) with the customer's MAC table, and (4) the MAC VC hash table for learning purposes.

Table 1. VPLS tables to memory mapping

Table Name	Direction	No of entries	Table size	Mem type
MAC VC Hash table	Ingress LSR	100K	4 Mbytes	External
Tunnel Direct table	Ingress LSR	32K	256Kbytes	Embedded
Tunnel hash table	Egress LSR	100K	4 Mbytes	External
VC hash Table	Egress LSR	4K	96Kbytes	Embedded
MAC Cust Hash table	Egress LSR	10K	200Kbytes	Embedded
VLL direct table	Egress LSR	4K	128Kbytes	Embedded

² Internet Engineering Task Force's (IETF) extension to the LDP protocol for Layer 2 VLAN services.

The MPLS tunnel hash table and the MAC hash tables are very large and stored in the external memory (see Table 1). The customer's MAC table, VLL and VC hash tunnels are small and stored in the embedded memories.

In total, it ends up using around 700 Kbytes of embedded memory and 8 Mbytes of external memory in this application. Given the density of existing DRAM chips, this is only a small percentage of the external memory and leaves a significant amount of headroom for growth.

Given a typical 5-fold bandwidth factor between the embedded and external memories, it is interesting to note that by storing less than 10% of the look-up tables in embedded memory, a more than 400% improvement in look-up performance is achieved when compared to storing all the tables in external memory.

TREE LOOK-UPS

Tree look-ups are used typically for IPv4 and IPv6 longest prefix match and Access Control Lists. The most well-known implementation of a tree look-up is the "Patricia tree".

In a standard Patricia-tree, the number of memory accesses needed to access the entry is $\log N + 1$ where N is the number of entries in the tree. So, in a tree with 256K entries, it takes 19 memory accesses to get the result for the look-up. This clearly exceeds the 10 Gb/s wire access budget of DRAMs.

Clearly, the look-up engine implementation needs to find ways to reduce the number of external memory accesses required.

The tree internal nodes could be stored in such a way that multiple levels of the tree can be walked at once. The number of memory accesses to perform the tree look-up is then reduced to $(\log N)/3 + 1$. So, this takes us from 19 down to 8 memory accesses.

The next step in external memory reduction is to store the tree in such a way that its internal nodes are stored inside the embedded memories and only the leaf nodes and results are stored externally. This would take us down to 2 external memory accesses for the same tree look-up.

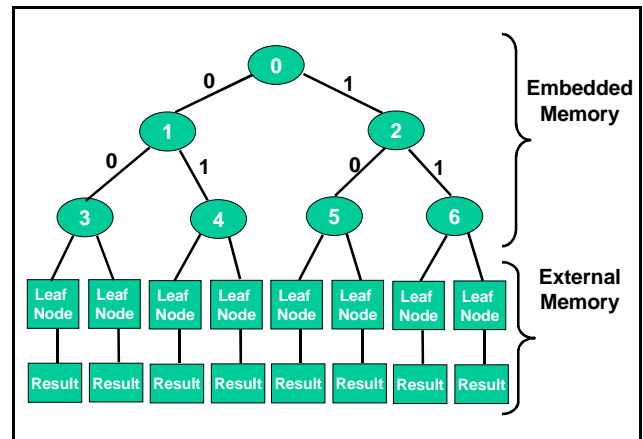


Figure 6. Tree mapping in memory

By mapping the tree in such a way with both memory types, the number of accesses becomes $(\log N)/3 + 1$ which now fits nicely in the 10 Gb/s minimum frame size budget and provides spare time for other table look-ups.

Considering the ratio of the bandwidth differential between the embedded and external memories (roughly 2.5 times in this example), we can see that the access time has been reduced by 90% by storing less than 10% of the tree within the embedded memory.

EXAMPLE OF A TREE INTENSIVE APPLICATION: IPV6 ROUTER WITH ACLS

In this example, the IPv6 router is meant to support 256K routes and a 4K ACL. The Route look-up is implemented using a longest prefix match on a tree, which is stored in a combination of embedded and external memory. The ACL look-up is implemented via a tree with first match policy stored completely in the embedded memory.

NEXT-GENERATION NPU: LOOK-UPS IN THE FUTURE

Given the advances in external memory speeds, it is possible to base new NPU architectures on designs that will require less embedded memory to achieve the same or even improved DRAM-based lookup performance.

For example, RLDRAM 64-bit DDR at 400 MHz provides a raw bandwidth of 51 Gb/s. With a second memory interface, it could provide around 100 Gb/s of table look-up bandwidth.

Freeing up some space within the NPU, by reducing the amount of embedded memory, will also allow a larger number of packet processing cores to be embedded in the NPU.

CONCLUSION

Having an NPU design where the look-up tables can be stored in external commodity DRAMs is highly desirable from the cost, power, real-estate points of view. But the memory speeds are insufficient for a 10 Gb/s application that the NPU can solely rely on it. The external DRAMs need to be complemented by a custom-hardware look-up engine and memory embedded within the NPU as well as unique algorithms that reduce the memory accesses required for each lookup, in order to sustain the demands of 10Gb/s applications.

EZchip Technologies Ltd.

1 Hatamar Street, PO Box 527, Yokneam 20692, Israel, Tel: +972-4-959-6666, Fax: +972-4-959-4166

EZchip Technologies Inc.

900 E. Hamilton Ave., Suite 100, Campbell, CA 95008, USA, Tel: (408) 879-7355, Fax: (408) 879-7357

Email: info@ezchip.com, Web: www.ezchip.com