A Generic Lookup Cache Architecture for Network Processing Applications

Janardhan Singaraju John A. Chandy
University of Connecticut
Storrs, CT, USA

***Abstract*— In this paper, we introduce a novel architecture for constructing caches for lookup operations that are used in a variety of network processing applications. The distinguishing feature of the cache is the ability to match on keys that of arbitrary lengths. We show through an FPGA implementation, that the proposed design can speed up lookup operations significantly compared to a software implementation. In conjunction with a network processor, the use of such a cache can greatly improve the response time of lookup intensive applications such as DNS resolution, directory lookup in network storage, and LDAP queries.**

I. Introduction

Many networking applications often require the ability to cache certain frequently used values. This is particularly true for operations that use tables that map one known value to another value. Examples include IP to Ethernet ARP address mappings and IP routing table lookups. These key sizes are fixed, but other mapping applications such as DNS lookup or LDAP searches require mapping a text string to an ID or object data structure. This process can be computationally expensive, and as a result, hardware implementations are desirable.

In this paper, we present a generic hardware cache architecture that is flexible enough to handle variable sized keys as well as provide a mechanism to do mapping in addition to string matching. The architecture is based on temporally cascaded content addressable memories memories (CAMs) and is an extension of a string matching design used in network intrusion detection [2].

II. Overall Architecture

We envision the use of a generic lookup cache as a key component in a network processor design. The cache could be integrated into the network processor core or perhaps off-chip. An architectural overview of the lookup cache is shown in Figure 1. The core of the lookup cache is a CAM character match array and a PE array. The CAM stores the keywords as 8-bit characters in an 8 by n array of CAM cells where n is equal to the number of characters in the CAM. Each CAM cell consists of a storage cell along with a bit comparator. that pulls the output match line low if there is a match. The column match line is shared as a wired-NOR line between all the cells in the column. Thus, when an input character is applied to a column, the column match signal is active high when all 8 bits match.

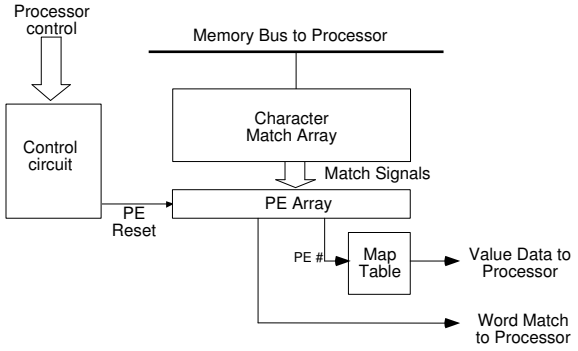

Fig. 1. Lookup Cache Architecture.

The PE array is similar in concept to the cellular automaton processor arrays proposed by Motomura et. al [1]. To perform a search on the lookup cache, the network processor will on successive clock cycles apply each character of the key to the CAM array. If a input character matches a column in the CAM array, the match signal is set and passed onto the corresponding PE for that column. Each PE holds a binary value called a *flag* which indicates that the input keyword matches a keyword in the CAM up to this point. As each character from the input sequence is presented to the CAM array and subsequent matches are generated, the PEs will forward its match flag on to subsequent PEs, and as long as the incoming characters continue to match, the flag will move to the end of the keyword.

As an example, consider a DNS cache with three entries: `foo`, `fubar`, and `bar`. Figure 2a shows the initial contents of the character match array and the PE array. Figure 2b shows the arrays when looking for a match on `foo`. When the first character, `f`, is presented to the CAM on the first clock cycle, there is a match on character 0 and character 3 of the character match array. Since each is the first character of a keyword (`foo` and `fubar` respectively), $PE(0)$ and $PE(3)$ are set to 1. On the next clock cycle, the second character, `o`, causes a match at two locations in the character match array - columns 1 and 2. The PE array evaluates and sets $PE(1)$ to 1 but does not set $PE(2)$ to 1 because the PE in the previous column, i.e. column 1, did not have its flag set. Note that the flag in $PE(3)$ drops out because

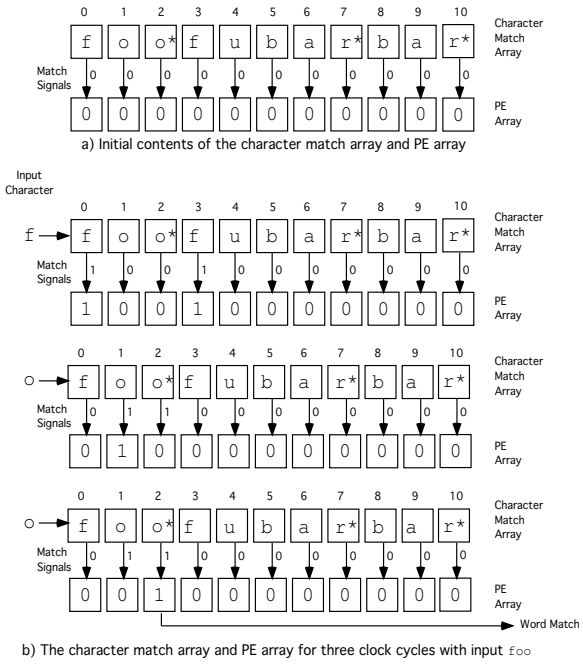

Fig. 2. Lookup Cache Example.

the incoming character did not match the u in position 4. On the next clock cycle, character o is presented to the CAM, moving the PE match bit along as well. At this point, the PE flag has reached the end of a keyword and we know that a word match has been found. The PE number is used as an index into a map table which stores either the 32-bit value itself or an indirect pointer to the value object.

Overall, with the design as shown, the time to search for a m byte keyword is m cycles. Since the communication path between the lookup cache and the processor is most likely larger than a byte, it seems reasonable that the lookup cache should be able to process multiple bytes per cycle rather than a single byte per cycle. In order to do so, we have altered the CAM array to match multiple bytes against the CAM in each cycle by aligning each keyword on a quad-byte boundary. This reduces the search time to $m/4$ cycles.

Replacement in a cache with variable sized keys provides challenges that do not exist in a normal cache. If keys and values are equal sized, one can simply mark an entry as invalid and then reuse that entry without loss of capacity. With variable sized keys, however, when we mark an entry as invalid, the next key to be inserted may or may not fit into that space, thus leading to holes in the CAM array space. In order to avoid this problem, whenever an entry in the cache is invalidated, all entries that are to the right of the invalidated entry are shifted an appropriate number of elements left. Shifting in a CAM array is easily accomplished by wiring connections from each CAM cell to the CAM cell in the previous column.

Number of characters	256	512	1024
Slices	2403	4812	9880
Frequency (MHz)	380.1	476.9	460.2
Throughput (Gb/s)	12.2	15.3	14.7
Searches per second	254M	318M	307M
Searches per second (sw)	887K	766K	632K

TABLE I
FPGA IMPLEMENTATION CHARACTERISTICS

The shift process can be done in a number of cycles equal to the number of shifts. Deletion time is thus proportional to m.

III. FPGA IMPLEMENTATION

We have designed a prototype implementation of this lookup cache architecture using the Xilinx Virtex-II Pro FPGA (XC2VP230-7). The design was written in VHDL and Xilinx's ISE 7.1 design environment was used for all parts of the design flow including synthesis, mapping, and place and route. We have implemented different sized *hosts* files in the lookup cache and the FPGA implementation characteristics are shown in Table I. The results show that the cache can get very high throughput using an FPGA implementation. Assuming an average keyword size of 6 characters, the cache can perform over 300 million searches per second. As a point of comparison, a software implementation using a hash search function performs nearly three orders of magnitude slower when run on a 1 GHz PowerPC computer.

IV. CONCLUSION

We have demonstrated a generic lookup cache for use in networking applications. It is clear that the lookup cache will function well with fixed-size keys such as with ARP caches or IP routing. Based on the current implementation of the lookup cache, we can process incoming streams at rates of over 10 Gb/s. The cache is unique in that it can accommodate variable string based searches making it amenable to a wide variety of applications. Any lookup that is based on non-fixed-size keys would be an ideal candidate to take advantage of this lookup cache. Some examples include directory lookup in network storage applications, DNS lookup, and LDAP processing. The cache can also be used in dictionary search applications such as network intrusion detection.

REFERENCES

[1] M. Motomura, H. Yamada, and T. Enomoto, "A 2k-word dictionary search processor (DISP) with an approximate word search capability," *IEEE Journal for Solid State Circuits*, vol. 27, pp. 883–891, June 1992.

[2] J. Singaraju, L. Bu, and J. A. Chandy, "A signature match processor architecture for network intrusion detection," in *Proceedings of IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 235–242, Apr. 2005.