A Flexible Accelerator for Layer 7 Networking Applications

Gokhan Memik and William H. Mangione-Smith
Department of Electrical Engineering
University of California, Los Angeles
{memik, billms}@ee.ucla.edu

ABSTRACT

In this paper, we present a flexible accelerator designed for networking applications. The accelerator can be utilized efficiently by a variety of Network Processor designs. Most Network Processors employ hardware accelerators for implementing key tasks. New applications require new tasks, such as pattern matching, to be performed on the packets in real-time. Using our proposed accelerator, we have implemented several such tasks and measured their performance. Specifically, the accelerator achieves 25-fold improvement on the performance of pattern matching, and 10-fold improvement for tree lookup, over optimized software solutions. Since the accelerator is used for different tasks, the hardware requirements are small compared to an accelerator group that implements the same set of tasks. We also present accurate analytic models to estimate the execution time of these networking tasks.

Categories and Subject Descriptors

B.2.4 [**Arithmetic and logic structures**]: High-Speed Arithmetic, Cost/Performance.

General Terms

Algorithms, Measurement, Performance, Design.

Keywords

Application-Specific Processor, Networking Applications, Network Processor, Accelerator, Table Lookup, Pattern Matching.

1. INTRODUCTION

Traditionally, the processing elements in networks are either ASICs or variations of general-purpose processors. Both schemes have their advantages and shortcomings. ASICs generally provide better performance, but they have higher manufacturing costs and lack the flexibility of programmable processors. If there is a change in the protocol or application, it is hard to implement the change in the design. General-purpose processors, on the other hand, are not optimized for networking applications and hence do not provide satisfactory performance for many networking applications.

Network Processors (NPUs) are a new type of processor optimized for networking applications that combine the advantages of ASIC and general-purpose processors. By utilizing specially designed hardware, these designs achieve performance comparable to ASIC. Since they are software programmable, they have the flexibility comparable to general-purpose processors.

Soon after their introduction [2, 10], the NPU market became one of the fastest growing segments of the microprocessor industry. Currently there are more than 30 companies with a variety of NPU designs [8].

New networking applications coupled with the higher link speeds demand new and more complex tasks to be performed efficiently in the processing elements. A significant portion of these new applications search or modify Layer 7 application payload information in the packet that was not accessed by the traditional network processing elements. In this paper, we present design details of an accelerator used to implement key networking operations such as tree lookup and pattern matching. These operations are expected to be in demand in the near future by applications that must access and modify Layer 7 payloads. For example, pattern matching is performed by several classification engines to categorize packets for security, QoS, or similar purposes.

Accelerators have been widely used by traditional routers and NPUs. However, due to their proprietary nature, design details of these accelerators have not been publicly discussed. We provide design details of such an accelerator and provide a thorough study of the performance of the accelerator. Specifically, we

- present a novel accelerator design and discuss the effect of the applications on our design decisions,
- discuss specialized algorithms used to implement different tasks on the accelerator,
- present analytical models to estimate the execution times of these algorithms,
- compare the performance of the accelerator against software solutions.

Currently implemented accelerators in the NPUs are designed for a single task. For example, each protocol processor in IBM PowerNP [6] has eight separate coprocessors for tasks such as accessing tree data structures and calculating CRCs. Our proposed accelerator can efficiently implement several different tasks. Hence, it reduces the chip area required to implement a set of accelerators.

The next section summarizes the related work. In Section 3, we explain the networking applications and present how frequently the accelerator can be used by these applications. In Section 4, we explain the accelerator design and the algorithms implemented in detail. Section 5 presents experimental results. We conclude the paper with Section 6, which summarizes the paper.

2. RELATED WORK

There is a wide variety of Network Processor design methodologies, which can be grouped into three major categories: VLIW-based processors, Simultaneous Multithreaded (SMT) processors, and single-chip multiprocessor systems. Crowley et al. [5] evaluate different design methodologies (a VLIW, an SMT, a single-chip multiprocessor, a fine-grain multiprocessor) for NPUs. Nie et al., on the other hand, discuss a RISC-based Network Processor that has hardware support for thread switching and bit-wise data manipulation [11].

Vendors such as Clearwater [4] implement smart memory management units for bulk packet data transfers. Wuytack et al. discuss a memory-oriented synthesis methodology and uses embedded network applications as an example [15].

Several NPU vendors implement accelerators to enhance the performance of the processor. Although the design details of these accelerators have not been disclosed, it emphasizes the importance of accelerators in Network Processors. Design details of ClearSpeed Table Lookup Engine are also not discussed, but the performance of the engine is presented for different key sizes used for tree lookup [3]. In IBM PowerNP [6] two processors share eight different accelerators such as checksum calculator that calculates and verifies frame; control access bus controller that controls access to data structures; string copier that accelerates data movement between coprocessors and the shared memory pool; tree search engine that performs analysis through tree searches; and semaphore manager that assists in controlling access to shared resources. C-port's C-5 employs six accelerators for up to 15 processing cores [2]. The accelerators perform tasks such as buffer and queue management and tree lookup. The table lookup engines in both PowerNP and C-5 are based on hashing techniques.

3. NETWORKING APPLICATIONS

Network Processors are special-purpose architectures designed for networking applications. Hence, we need to understand the networking applications in detail to be able to judge the implications of design decisions for these processors. For this study, we have examined several applications from the NetBench [9] suite. NetBench is designed for NPUs and contains ten applications listed in Table 1. CRC performs the 32-bit cyclic redundancy checking. DH, MD5, and SSL are security applications. DH implements the Diffie-Hellmann public key encryption/decryption algorithm. MD5 is the message digest algorithm version 5, which generates secure signatures for packets. SSL implements the secure sockets layer, which first performs RSA and DSA authentication and then uses the blowfish and 3DES encryption/decryption algorithms. DRR is the deficit-round-robin scheduling algorithm used to achieve fair share of the available bandwidth between the connections going through the switch or router. Ipchains is a common firewall application used to filter malicious packets from a network. Nat implements the network address translation application used by several service providers to increase the utilization of the available IP addresses. Route implements the IPv4 routing. Tl is the table lookup routine implemented as a radix-tree search and URL implements the URL-based switching, which performs intelligent switching according to the contents of the packet.

Table 1 also presents the fraction of the execution cycles spent in functions that can be accelerated by the proposed accelerator (*accel*), the remaining cycles executed excluding the memory stalls (*rest*), and the number of instructions executed per packet processed (*inst/packet*). To find the execution fraction that can be accelerated, we have performed a thorough analysis of the applications with the provided input sets. We have examined each function in the applications and decided whether the function (or a sub-task in it) can be implemented using the accelerator proposed in this work. Then we have measured the fraction of the execution that is spent in these functions. The fraction of the execution time that can utilize the proposed accelerator varied from 0% (for CRC) to 81.2% (for URL) with an arithmetic mean of 37.2%.

Table 1. Application information: Input parameters used for applications (*input*), the portions of the executed instructions that can be accelerated (*accel*), remaining execution in the applications excluding memory stalls (*rest*), and number of instructions executed per packet (*inst/packet*)[1].

App.	Input	Accel [%]	Rest [%]	Inst/packet [K]
CRC	50000	0	98.6	23.9
DH	20	27.3	71.4	121700[1]
DRR	128 50000	32.4	55.7	6.1
Ipchains	9 64 1000	37.1	53.9	7.4
MD5	50000	44.5	49.5	20.4
Nat	128 50000	33.4	57.7	2.1
Route	128 50000	35.1	49.5	1.8
SSL	1	34.7	62.6	408000[1]
Tl	128 50000	46.1	37.4	1.2
URL	128 50000	81.2	12.6	17.1

4. ACCELERATOR DESIGN

In this section, we explain the design details of the accelerator and present novel algorithms that utilize it.

Figure 1 shows the overall accelerator design. The design of the unit resembles a barrel shifter. In a traditional barrel shifter, each module selects either the shifted input bit or the unshifted input bit to perform the conditional shifting operation. In our design, however, the input to each module is a byte and all control signals affect complete bytes. In addition, modules are capable of performing some logical operations on their input values. Specifically, the module is selected to perform one of the following functions on their input signals:

- output = left input (shift right)
- output = right input (pass without shift)
- output = (left input AND right input)
- output = (left input OR right input)
- output = (left input XNOR right input)
- output = (left input BYTE-WISE-OR right input)
- output = (left input BYTE-WISE-AND right input)

The utility of each these operations will be examined during the discussion of the applications in the following subsections. AND, OR, and XNOR are bit-wise operations, performing the logical operation on the corresponding bits of the two input signals.

[1] For DH this column corresponds to instructions per key generation and for SSL it corresponds to instructions for performing all supported key generations and performing encryptions on the packet.

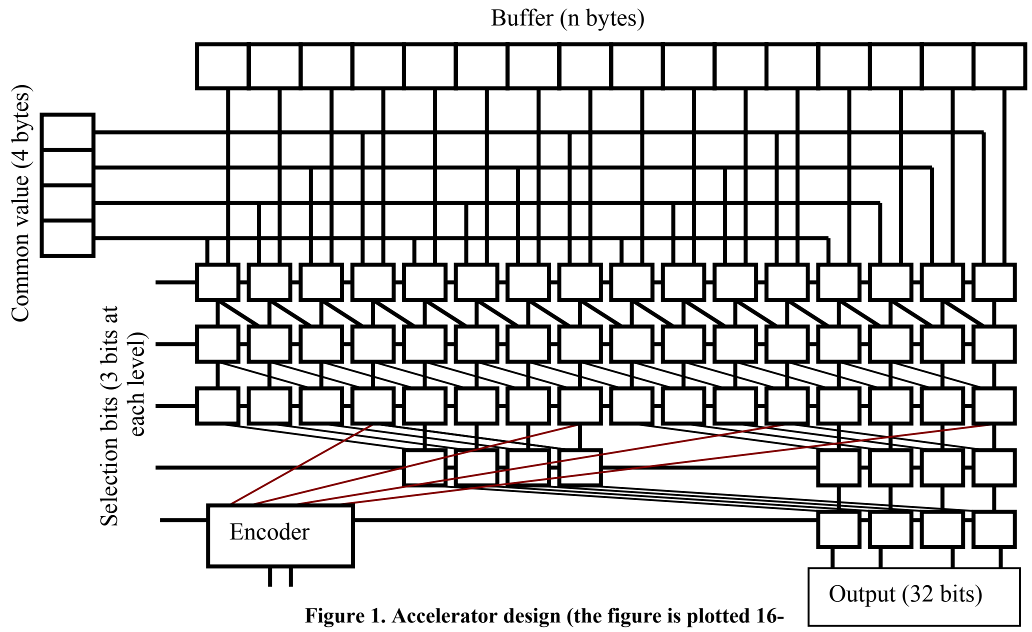

Figure 1. Accelerator design (the figure is plotted 16-bytes wide for illustrative purposes)

Byte-wise-or results 0xFF if one of its inputs is equal to 0xFF, 0x0 otherwise. Similarly, byte-wise-and results 0xFF if both of its inputs are equal to 0xFF, 0x0 otherwise. Figure 2 illustrates the design of the modules in the accelerator. All modules are equivalent. The leftmost AND gates in the figure are 8-bit AND gates taking all their input either from left input or from right input byte and performing the AND operation on these bits. Hence, they output 1 if the corresponding input equals to 0xFF. The outputs of these gates are connected to 2-bit and and or gates. The results of these 2-bit logic gates are rippled to form 8-bit inputs to the multiplexer. Hence, the signals on their output are 0xFF or 0x00.

The accelerator uses a common value register and a wide buffer (n bytes) to store the packet information. The common value is 4 bytes wide. We have selected this size, because it is an efficient width for many tasks. For example, this size is equal to the size of an IP address in IPv4. Hence, many address related tasks (lookup, match, etc.) can be efficiently performed. We have performed several simulations with different widths (n) of the accelerator. The results for the simulations with the accelerator widths ranging from 64 to 512 bytes are given in Section 5. Using the selection bits, the operations on each level is selected. Using these selection bits and appropriate setting of the common value and the buffer, a wide variety of tasks can be accomplished by the accelerator.

The accelerator also uses an encoder, which is embedded in the third level of the modules. It has (n / 4) inputs chosen one bit from every four modules starting with the fourth leftmost module as shown in Figure 1. This encoder gives the index of the leftmost input that is high.

For illustrative purposes, the accelerator in Figure 1 is drawn 16-byte wide. In our design, however, we have selected the unit to be 128-byte wide, which gives a good performance / cost ratio. The results are explained in Section 5. This architecture directly supports the extraction of any 4-bytes from the input buffer without requiring that they be contiguous. This function is very useful for traditional packet analysis operations.

We have implemented an RTL level description of the accelerator in VHDL and synthesized it using Synopsis design compiler [11]. We have used a library representing a .25μ technology. The accelerator with 128-byte buffer required 115K transistors and has a total delay of 6.65 ns.

In the following subsections, we describe algorithms to implement key emerging tasks for networking applications and present analytical models to estimate the execution time of these tasks.

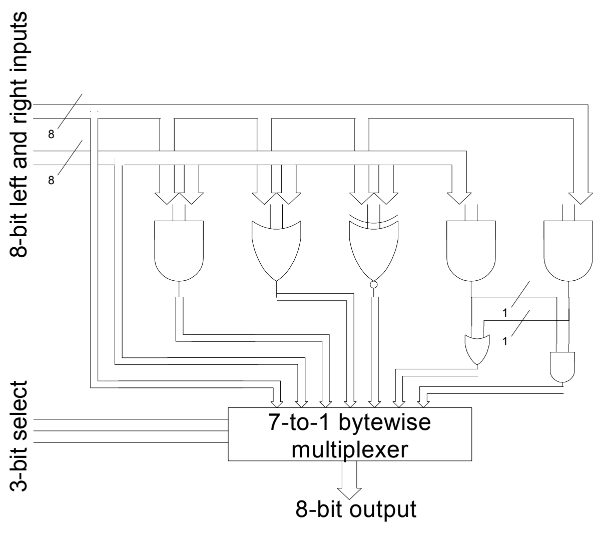

Figure 2. Block diagram of the modules

4.1 Pattern Matching

One of the common tasks in the networking applications is pattern matching. For example, in URL-based switching, a known pattern is searched in the payload to decide the type of the HTML request encoded in the packet. Similarly, several security applications search for known patterns in the payload to detect malicious packets [12].

Consider an example, in which we are searching for the pattern '*.jpg*' in the payload. The pseudo-code in Figure 3 illustrates the pattern-matching algorithm. We first load the common value with the '*.jpg*' value and load the payload into the buffer. Then, we configure the accelerator to perform 'xnor' in the highest level, 'and' in the next two levels and 'byte-wise-or' in the following levels. If there is no match in the payload, the 'xnor' results will not be equal to 0xFF and this will in turn make the output equal to zero. If the payload contains '*.jpg*' and the first character ('.') is aligned with the first character of the common value ('.'), then there will be four consecutive 0xFF values at the end of the first level ('xnor') and at least one 0xFF after the third level ('and'). Hence, the result will not be equal to zero. Note that, to guarantee whether there is a match or not, we have to perform the operation with different alignments. This is achieved by rotating the common value. After four iterations, if there is not a match, we can conclude that the payload section examined does not contain the pattern. Note that the pattern matching code contains two unique instructions: load-buffer and op. 'load-buffer r, add' results in loading r bytes from address add into the accelerator input buffer. 'Op r1, r2, r3' activates the accelerator with the selection bits stored in r2 and the common value stored in r3. The result of the operation is written to r1.

```
load                 $2,ACCELERATOR_WIDTH
load                 $3,'.jpg'
load-buffer          $2,payload
load                 $4,0x2db6da93 #conf.
op                   $2,$3,$4
branch-not-equal     $2,$0,$match
rotote-right         $3,$3,8
op                   $2,$3,$4
branch-not-equal     $2,$0,$match
rotote-right         $3,$3,8
op                   $2,$3,$4
branch-not-equal     $2,$0,$match
rotote-right         $3,$3,8
op                   $2,$3,$4
branch-not-equal     $2,$0,$match
#no match
```

Figure 3. Code for pattern matching

If the pattern of interest is larger than four bytes, we first divide it into four byte sub-patterns. Then each sub-pattern is searched in the buffer. If all sub-patterns are found, we return the buffer to the processor to be examined for the whole pattern. If one of the sub-patterns is not matched, we indicate a mismatch to the processor. Note that, if the pattern of interest is smaller than four bytes, we can use the accelerator directly by modifying the selection bits.

We have developed an analytical model to estimate the execution time of the pattern matching algorithm. The number of cycles required can be estimated using:

$$\#cycles = \lceil haystack / n \rceil * (Mem(max(n, haystack)) + exec)$$

Equation 1. Cycles required for pattern matching

In Equation 1 haystack corresponds to the size of the buffer that is searched, Mem (x) corresponds to the number of cycles spent in reading x amount of data from memory and exec is number of cycles required by the accelerator when the buffer data is available. For example, assuming a delay of 4 cycles for the accelerator, exec equals to 26 (1 cycle for setting common value, 1 cycle for setting the select bits and 24 cycles to produce the results, because we have to perform 4 iterations each lasting for 6 cycles). In Section 5.2, we compare the estimations for the model against simulation numbers and show that the model estimates the number of execution cycles within 22% in worst case.

We have performed several simulations to see the performance effect of the accelerator. The accelerator is able to improve the performance by as much as 25 times over a software solution. The experiments and the results are explained in Section 5.3.

4.2 Tree Lookup

Most networking applications contain significant amount of table lookup operations. For example, in traditional IP routing, routing tables must be traversed to decide what rules to apply on the packet. An efficient method for implementing these tables is in the form of trees. Particularly, radix-tree lookup is commonly used in several UNIX systems and routing architectures [14].

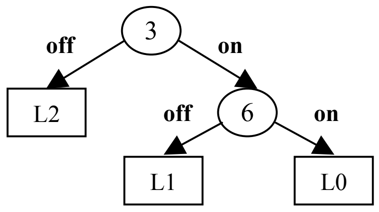

Figure 4. A radix-tree example

In radix-tree lookup, the tree is traversed from the root to the leaves. Each node has a specified offset or position. If the position of the address (the bit at the position) is set, then the right sub-tree is traversed, otherwise, the lookup continues on the left sub-tree (the selection of right or left for the active bit is implementation dependant). An example tree is given in Figure 4. For the example tree, if the third bit of the searched address is zero, the lookup will return leaf node 2. Otherwise, the sixth bit will be examined.

The radix-tree algorithm for the accelerator uses the fact that the "on" locations used to reach a leaf uniquely determines the resultant leaf node of the lookup. For example, leaf 0 in Figure 4 will be reached if and only if both the third and sixth bits of the searched address are equal to one. Our algorithm first examines the tree to find the correct buffer setting for the accelerator. This process is given in Figure 5. The leaf nodes are sorted from right to left and a mask value for each leaf node is found. These masks have a 0 at the on positions to reach the leaf and 1 at all other positions. For our example tree, this will result in the following buffer values: 0xFFFFFFB7, 0xFFFFFFF7, 0xFFFFFFFF. This processing must be performed whenever there is a change in the network route (hence the tree). Since in several networking applications the tree is mostly stable (for example, in the traditional routing it changes if there is a topology change in the network, which occurs infrequently) the overhead is small.

```
void
setup_bufs (struct tree *tree, unsigned int masks)
{
  if (tree->type == leaf)
    {
      bufs[order] = masks;
      order++;
      return;
    }
   setup_bufs (tree->right, masks ^ (1 << tree-
>position));
   setup_bufs (tree->left, masks);
}
```

Figure 5. C code for preparation of the buffer value for the for radix-tree lookup

After the buffer values are set, the common value in the accelerator is set as the address from the packet. The highest level is set to perform 'or' and the following two levels perform 'byte-wise-and'. If there is a 0xFF value at the result of the third level, the index of the module will determine the result of the lookup operation. This index will be equal to the output of the encoder. This value is later used for retrieving the leaf information from the array of leaf nodes.

If the required buffer size exceeds the width of the accelerator, an indirect addressing is used. The first accelerator operation identifies the buffer to be retrieved, for which the accelerator gives the index of the resultant leaf node.

We have developed an analytical model to estimate the number of execution cycles for a tree lookup:

$$\#cycles = \lceil \log_{(n/4)} leaves \rceil * (Mem(avg.) + exec)$$

Equation 2. Cycles required for table lookup

In Equation 2 leaves correspond to the number of leaf nodes, Mem (avg.) corresponds to the number of cycles spent in reading data from memory and exec is number of cycles required by the accelerator when the buffer data is available. For example, assuming the delay of the accelerator is 4 cycles, exec equals to 7 (1 cycle for setting common value, 1 cycle for setting the select bits and 4 cycles to produce the results and 1 cycle for reading the encoder value). In Section 5.2, we compare the estimations for the model against simulation numbers and show that the model estimates the number of execution cycles within 3% in worst case.

We have performed several simulations to measure the effectiveness of the accelerator for tree-lookup. The accelerator is able to improve the performance of software solution by as much as 12-times. The detailed results are explained in Section 5.3.

4.3 Security

Many encryption/decryption algorithms use permutation to find unique signatures for a packet or to change the original packet payload. For example, the MD5 algorithm use the permutations listed in Figure 6. Such permutations can be directly mapped into the accelerator. For example, to implement 'F' from Figure 6, the buffer is loaded with x, y, ~x, and z series. Then, in the first two levels the buffer values are forwarded. In the third level, 'and' and in the fourth level 'or' operation is performed. The results at this stage are propagated one by one to the output register to get the desired section from the series.

```
#define F(x, y, z) (((x) & (y)) | ((~x) & (z)))
#define G(x, y, z) (((x) & (z)) | ((y) & (~z)))
#define H(x, y, z) ((x) ^ (y) ^ (z))
#define I(x, y, z) ((y) ^ ((x) | (~z)))
```

Figure 6. MD5 code segment for permutations

5. EXPERIMENTS

5.1 Experimental Setup

In the simulations performed, the programmable core is an out-of-order execution processor much like Alpha 21264 [7]. We have modified the SimpleScalar [1] simulator by implementing the accelerator as a function unit in the processor. Three different accelerator widths are studied: 64 bytes, 128 bytes and 512 bytes. The delay for the 64 and 128-byte accelerators is set to 4 cycles and the delay for the 512-byte accelerator is set to 5 cycles. The simulated core has 32 KB of separate L1 data and instruction caches and a 512 KB unified L2 cache. The latencies for L1 and L2 caches are set to 1 and 10 cycles, respectively.

5.2 Validating Analytical Models

In Sections 4.1 and 4.2, we have presented analytical models to estimate the execution latency for pattern matching and tree lookup algorithms. In this section, we compare the estimations made using these models and simulation values from the SimpleScalar simulator.

Figure 7 summarizes the estimations for the pattern matching task. The worst estimation is made for 512-byte wide accelerator for the haystack size of 640 bytes, for which the error is 21.85%.

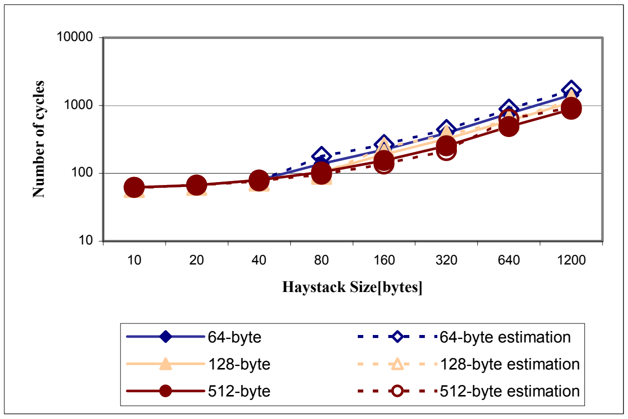

Figure 7. Estimations for pattern matching

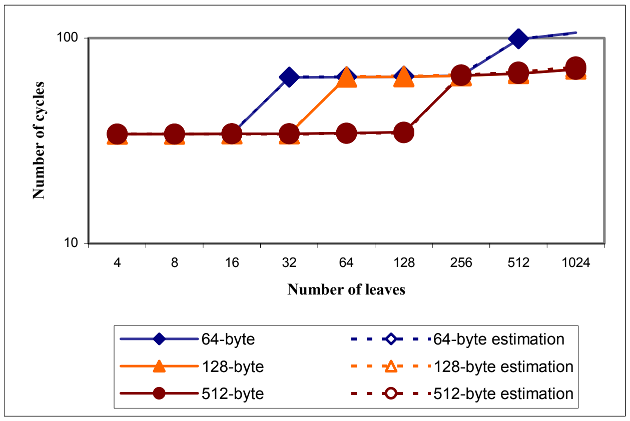

Figure 8. Estimations for tree lookup

The results for tree lookup are summarized in Figure 8. In all the configurations, the analytical model for tree lookup almost exactly matches the simulation numbers. The largest error is 2.6% for lookup in a tree with 1024 rules with 512-byte wide accelerator.

5.3 Performance Simulations

The simulated applications in this section are kernels that perform the radix-tree lookup and pattern matching tasks. Our goal in these simulations is to measure the performance of the accelerator for different input parameters.

Figure 9 summarizes the results for pattern matching. The improvement achieved by the accelerator increases with the increased string size. For large strings, the accelerator achieves up to 25-fold improvements (e.g., the accelerator with 512-byte width achieves 24.2-fold improvement for 1200-byte haystack).

The results for tree lookup are summarized in Figure 10. As the table size is increased the improvement of the accelerator increases. However, the speed-up is non-monotonic. It can be seen that these jumps exactly occur when the buffer is not large enough to hold all the leaves. At these sizes, the accelerator has to perform multiple accesses, creating the saw tooth pattern observed in the results. In addition, the overall trend in the improvement for 64-byte wide accelerator is downward, whereas it is upwards for the 512-byte wide accelerator and is almost constant for the 128-byte design.

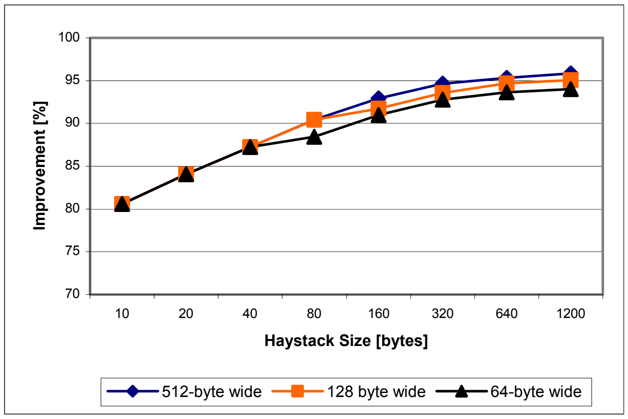

Figure 9. Improvement by accelerator for pattern matching

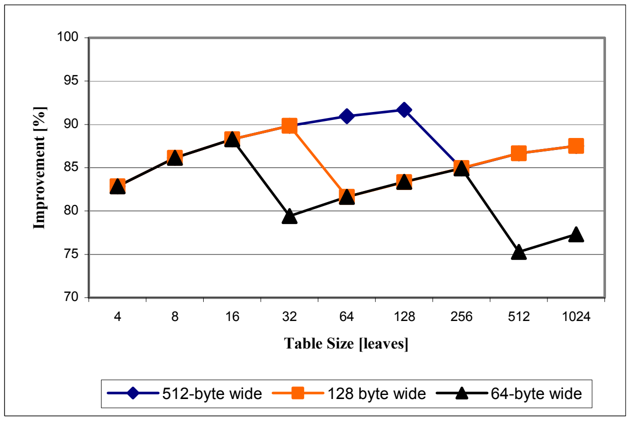

Figure 10. Improvement by accelerator for tree lookup

We have also performed simulations to measure the effectiveness of the accelerator for security applications. We have modified the MD5Transform routine (including the Decode sub-routine) to utilize the function unit. The accelerator was able to increase the performance of MD5Transform by 58.54%.

6. CONCLUSIONS

New networking applications coupled with the higher link speeds demand new and more complex tasks to be performed efficiently in the processing elements. A significant portion of these applications searches or modifies Layer 7 information in the packet that was not accessed by the traditional network processing elements. Hence, implementing accelerators for such tasks is a necessity to achieve high performance. In this paper, we have presented design details of such an accelerator. We have also presented novel algorithms to implement such key tasks. We have shown that our proposed accelerator can perform tasks such as tree lookup and pattern matching efficiently. Specifically, it achieves up to 25-fold improvements for pattern matching and 10-fold improvement for tree lookup over optimized software solutions. Finally, we have performed a study for the utilization of the accelerator in a multi-core environment.

REFERENCES

[1] Burger, D. and Austin, T. The SimpleScalar Tool Set, Version 2.0. Technical Report CS-TR-97-1342, University of Wisconsin, June 1997.

[2] C-port Corporation. C-5 Network Processor Architecture Guide. C-port technical library C5NPD0-AG/D, May 2001.

[3] ClearSpeed Technology Ltd. The ClearSpeed Table Lookup Engine. White paper 02-WP-1021 v3.0. http://www.clearspeed.com/papers/tle_book.pdf

[4] Clearwater Networks. Introducing the CNP810 Family. http://www.clearwaternetworks.com/clearwater-overview.pdf

[5] Crowley, P., Fiuczynski, M. E., Baer, J. L, Bershad, B. N. Characterizing Processor Architectures for Programmable Network Interfaces. In *Proc. of International Symposium on Supercomputing*, Santa Fe / NM, 2000.

[6] International Business Machine Corporation. IBM PowerNP NP4GS3 Network Processor Datasheet. IBM technical library, np3_DLTOC.fm.08, May / 2001.

[7] Kessler, R. The Alpha 21264 Microprocessor. In IEEE Micro, 19(2), Mar/Apr 1999.

[8] Mangione-Smith, and W. H. and Memik, G. Network Processing: Applications, Architectures and Examples. Tutorial at *International Symposium on Microarchitecture*, Austin / TX, Dec. 2001.

[9] Memik, G. and Mangione-Smith, W. H. NetBench: A Benchmarking Suite for Network Processors. In Proc. of *International Conference on Computer-Aided Design*, San Jose / CA, Nov. 2001.

[10] MMC Networks, Inc. Leading the Network Processor Revolution. http://www.mmcnet.com/Solutions

[11] Nie, X., Gazsi, L., Engel, F., Fettweis, G. A New Network Processor Architecture for High-Speed Communications. In Proc. of *IEEE Workshop on Signal Processing Systems*, Taipei / Taiwan, Oct 1999.

[12] Roesch, M. Snort: The Open Source Network Intrusion Detection System Web Site. http://www.snort.org

[13] Synopsys, Inc. Synopsys design compiler – Overview. http://www.synopsis.com/products/logic/design_comp_cs.html

[14] Villamizar, C. OSPF Optimized Multipath (OSPF-OMP). Internet Draft ietf-ospf-mpp-02, Feb, 1999.

[15] S. Wuytack, J.L. da Silva, Fr. Catthoor, G. de Jong and Ch. Ykman. Memory Management for Embedded Network Applications. *IEEE Transactions on Computer-Aided Design,* Volume 18, Number 5, pp. 533-544, May 1999.