Architectural Analysis and Instruction-Set Optimization for Design of Network Protocol Processors*

Haiyong Xie†, Li Zhao, and Laxmi Bhuyan
Computer Science & Engineering Department
University of California, Riverside
Riverside, CA 92521
{zhao, bhuyan}@cs.ucr.edu

ABSTRACT

TCP/IP protocol processing latency has been an important issue in high-speed networks. In this paper, we present an architectural study of TCP/IP protocol. We port the TCP/IP protocol stack from the 4.4 FreeBSD to the SimpleScalar simulation environment. The architectural characteristics, such as instruction level parallelism and cache behavior, are studied through simulation. We also compare the characteristics of TCP/IP protocol to that of SPECint benchmark programs. It turns out that the former is quite different from the latter due to the unique processing structure. Furthermore, in order to improve the effectiveness of instruction cache, frequent instruction pairs are analyzed, and corresponding architectural optimizations are made to the instruction set architecture. The performance is evaluated in the simulator. We find that a 23% improvement can be achieved by taking advantage of the optimization. The instruction set optimizations proposed in this paper will be helpful for the design of new programmable protocol processors in future.

Categories and Subject Descriptors

C.3 [**Special-Purpose and Application-Based Systems**]: Real-time and Embedded Systems

General Terms

Measurement, Design, Performance

Keywords

Network processing, Architecture simulation, Instruction optimization, TCP/IP protocol

*The research was supported in part by NSF grants CCR-0220096 and ACI-0233858.

†Haiyong Xie is currently with Computer Science Department at Yale University, New Haven, CT 06520. His current email address is yong@cs.yale.edu.

CODES+ISSS'03, October 1–3, 2003, Newport Beach, California, USA.

1. INTRODUCTION

The gaps between the available network bandwidth, CPU computation power, and the speed of memory system are expanding day by day. In the past decade, the speed of the physical layer has increased exponentially due to break-through advances in technology. The bandwidth of the Internet's trunk lines has increased over the years to 10 Gbps and will continue to grow at a rate of 4 times per year. This growth is much faster than that of the CPU performance and has led to a growing gap between the transmission bandwidth and processor speed [4][18].

The demands on needed computation power of CPUs and speed of memory systems to sustain higher network bandwidth are greater than ever before. For instance, in order to process a typical 64-byte TCP control packet, the CPU has to complete TCP/IP protocol processing and deliver the packet to the network interface card in 512 ns to sustain the 1Gbps bandwidth. However, as revealed by our study, the TCP/IP protocol stack executes more than 1600 instructions to process such a packet on a RISC machine (similar results in [15]). This means that for a CPU running at 1GHz with 1 instruction per cycle, it will take more than 1600 CPU cycles, or equivalently more than 1600 ns, to process such a packet. This time is for critical path processing, and does not include time on the connection establishment and tear-down, nor does it take into consideration factors such as memory access latency, the overhead introduced by the operating system, and the capability of upper layer applications. We conclude, therefore, that with special optimizations current computer systems do not have enough capability to fully sustain networks operating at 1Gbps bandwidth or higher.

TCP/IP protocol processing latency is vital to the overall performance of special network devices such as web switches [5][7], which perform not only transport-layer but also application-layer functions. With increasing demand on highly secure transmissions over network, future routers will have to look into both layer 4 and 5 information. Currently, network processors with hardware accelerators, multi-threading and multiprocessing are being designed and marketed aggressively. However, implementing protocol processing in network processors is a major concern, and poses several challenges.

TCP/IP processing latency can be attributed to two aspects: data-touching operations and non-data-touching operations [11][12]. Data-touching operations such as checksum computations dominate the processing latencies of large packets, and consume approximately 70% of the total processing overhead. However, non-data-touching operations, such as protocol data structure manipulation and protocol specific operations, are dominant in case of small- and medium-payload packets. Extensive research has been conducted to maximize the throughput by improving data-touching operations [1][8].

However, when a realistic distribution of message size is considered, non-data-touching operations introduce 84% of total protocol processing latency for TCP and 60% for UDP [11][12], which indicates that it is more important to improve processing of non-data-touching operations. Hence, we primarily focus on reducing the latencies for non-data-touching operations.

In this paper, we take TCP/IP protocol processing as an independent workload and study its various architectural characteristics by means of simulation. We measure the impact of architectural parameters like ILP and cache behavior on the TCP performance. Comparisons between TCP/IP and SPECint2000 [19] benchmark programs are also made in order to highlight the different characteristics.

We port the reference implementation of TCP/IP protocol stack in 4.4 FreeBSD [14] to the SimpleScalar simulation environment [17]. To the best of our knowledge, we are the first to run the complete standard TCP/IP protocol stack from Berkeley UNIX implementation in an execution-driven simulator. We extend the SimpleScalar simulation tool-set to break down the execution time into three categories: CPU execution time, data access time and instruction access time. This extended simulator is used to study architectural characteristics of, and evaluate the optimization for, TCP/IP protocol processing.

It is observed that the instruction cache has a greater impact on TCP/IP performance. In order to improve effectiveness of an instruction cache, we identify the frequently used instruction pairs by extending the simulator and profiling the protocol stack. We design and implement some new instructions for the simulated CPU so that the frequently used instruction pairs can be replaced with corresponding new instructions. By doing so, we expect to not only reduce the total number of instructions needed to process each packet, but also improve instruction cache performance and reduce the execution time. We make only minor modification to the instruction set architecture and evaluate the overall performance of this new ISA in the simulator. We find that 23% improvement can be achieved by taking advantage of the minor extensions to the instruction set architecture. This observation can be used to help design new programmable protocol processors in the future.

The rest of this paper is organized as follows. Section 2 describes the methodology employed in our work and the parameters of the simulation environment. The synthetic network traffic trace is described as well. Section 3 presents various results of performance evaluation with SimpleScalar. We also make comparisons between TCP/IP workload and SPECint benchmark programs. Section 4 analyzes the patterns of frequent instruction pairs and makes architectural optimizations to the instruction set architecture in order to improve the latency of non-data-touch operations. Section 5 presents background and related work. Section 6 concludes the paper.

2. METHODOLOGY

This section describes the methodologies used to evaluate the performance of TCP/IP protocol processing, including execution-driven simulation and synthetic network traces.

2.1 Execution-driven simulation

We use SimpleScalar Tool Set 3.0 to study the architectural characteristics of protocol processing. The default processor architecture we simulate has 16K 4-way set-associative L1 instruction and data caches. Both have block sizes of 32 bytes and LRU as the replacement strategy. The data cache is write-back, write-allocate, and non-blocking with 2 ports. There is a 256K 4-way set associative unified L2 cache with 64-byte blocks and a 6-cycle hit latency. The processor has an instruction level parallelism of 4, and bimodal as the default branch predictor. The bimodal branch predictor used in our reference processor is not state of the art, however, for TCP/IP protocol workload, it is almost as good as the perfect predictor, which is not shown in this paper due to space limitations.

We extend the detailed timing simulator, *sim-outorder*, to further break down the total execution time into three categories: CPU execution time when there are instructions being executed without stalling for data or instruction access, data access time when CPU stalls due to data cache misses, and instruction access time when CPU stalls due to instruction cache misses. The overlapped access time when CPU stalls waiting for both data and instruction access is taken as data access time.

2.2 Network traffic trace

TCP/IP protocol processing latency depends directly on the packet size. The packet sizes of both local area networks and wide area networks conform to bimodal distribution [20]. We analyze traffic traces available from NLANR for both local area networks (LAN) and wide area networks (WAN). The results in Figure 1 are similar to those in early studies. Approximately 52% of the packets in local area networks are smaller than 200 bytes in size. In the WAN traces, most of the packets (approximately 99%) are smaller than 1500 bytes, and the size distribution of these packets is similar to that of LAN traces.

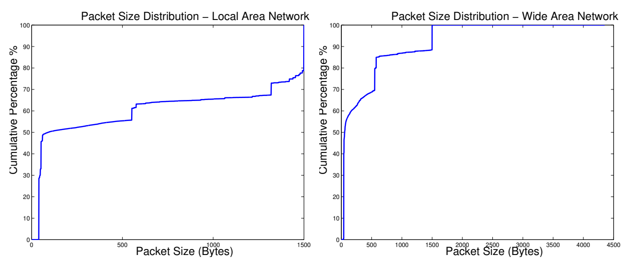

Figure 1: Packet size distribution of LAN and WAN

It is important to study the characteristics of TCP/IP protocol stack with realistic network traffic traces in execution-driven simulators. We instrument a client program with knowledge of the distribution of realistic network traffic. The program establishes TCP connections and transmits data, using predefined packet sizes, the distribution of which conforms to that in Figure 1. By doing so, we generate a synthetic trace of 10K packets. We call this trace as realistic or synthetic network trace throughout this paper. We evaluate the TCP/IP protocol stack in SimpleScalar using this synthetic trace.

3. PERFORMANCE EVALUATION THROUGH SIMULATION

In this section, we present the impact of various architectural parameters on TCP/IP. The results are obtained by running the protocol stack in the simulation environment with the synthetic network trace.

3.1 Instruction mix and ILP

We study the instruction mix and the impact of ILP on execution time by varying the instruction fetch, decode, and issue width and the number of functional units. The results are compared with that of SPECint benchmarks as well, as shown in Figure 2. Note that the execution time has been normalized to that of SPECint programs with ILP being 1. Both the benchmarks and TCP/IP run in the simulation environment with default configuration described in Section 2.

The TCP/IP protocol stack has an unexpected high percentage of unconditional branch instructions. This leads to not-so-good program structures that introduce higher instruction cache miss ratio. The TCP/IP protocol stack has a lower percentage of conditional branch instructions compared to SPECint benchmark. The reason is that the protocol processing is more like a streamlined processing with only a small number of loops. These program properties require different cache configurations compared to SPECint benchmark programs.

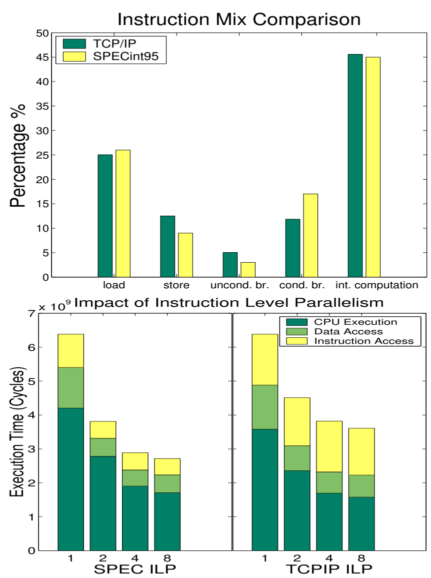

Figure 2: Instruction mix and ILP (top: instruction mix, bottom: ILP)

It is observed that TCP/IP protocol processing is less sensitive to instruction level parallelism, compared to SPECint programs. SPECint programs obtain improvement on both CPU execution time and instruction cache when ILP increases. However, for TCP/IP protocol processing, it does not improve data or instruction access time very much. For instance, the total execution time of SPECint programs improves 40% and 24%, while that of TCP/IP processing improves 29% and 15%, respectively, when the instruction level parallelism increases from 1 to 2 and 2 to 4. However, the execution time improves little (5%) with ILP greater than 4 for both SPECint and TCP/IP programs. The instruction level parallelism in TCP/IP is limited, and other techniques such as multi-threading are needed to go beyond this speed-up. The major problem with TCP/IP lies in the long instruction access time, which is unaffected by increasing ILP. Instruction cache plays a more important role in improving TCP/IP processing latency, as can be seen in later sections.

3.2 Cache behavior

We study both the L1 instruction and data cache behavior by varying such parameters as cache size, line size, and set associativity. Data cache behavior is similar to that of SPECint programs, which is not presented here due to space limitations. The results of instruction cache are presented in Figure 3, and those of SPECint benchmark are also shown in these figures for comparison purpose.

Figure 3 shows the different impact of cache size on L1 instruction cache (the figure on the upper-left). The miss ratio of instruction cache decreases as cache size increases. There is a 60% improvement as the cache size increases from 16KB to 32KB and 74% from 32KB to 64KB. From 64KB and beyond, the improvement is not so much as that from 8KB to 16KB and 64KB, which indicates that capacity misses become dominant. Due to the program structure of the TCP/IP protocol stack, which is a monolithic program with a small number of loops, the requirements on cache size are higher compared to that of SPECint benchmark programs, as can be seen in the figure. Instruction cache is very sensitive to cache size. It is because that TCP/IP has a very large kernel code size and has a large capacity miss. Additionally, data cache does not benefit from larger cache size than 8K and beyond (not shown here). On the contrary, a much smaller cache is enough to bound the miss ratio to a sufficiently low level for SPECint programs.

Similar behavior with varied associativity and line size is shown in Figure 3 as well. Higher associativity and larger line size both have greater impact on instruction cache. From Figure 3 we can see that there exists a great difference between TCP/IP and SPECint programs in terms of instruction cache performance. It is observed that TCP/IP processing almost always performs much worse than SPECint benchmark given the same cache parameters. For instance, a 16KB cache size is nearly enough for most of the programs in the benchmark to reduce instruction cache miss ratio to less than 1%, however, it is not the case with TCP/IP processing, which needs 4 times more cache size to obtain miss ratio less than 1%. It can also be seen that cache associativity and line size have much greater impact on TCP/IP performance than SPECint benchmark programs. For instance, cache miss ratio improves by 51% for TCP/IP processing compared to 25% when line size increases from 16 to 64 bytes. Instruction cache miss ratio continuously improves even when line size is larger than 64 bytes, while SPEC benchmark programs perform worse due to increased miss penalty.

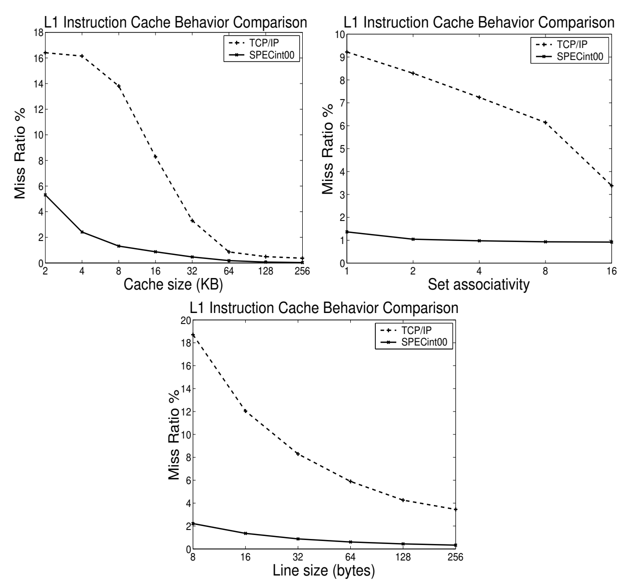

Figure 3: Impact of cache parameters (upper left and right: size, associativity; bottom: line size)

TCP/IP protocol stack benefits more from a instruction cache with larger size, higher associativity, or larger line size, compared to SPECint benchmark programs. The above observation leads to the conclusion that given a total L1 cache size on the chip, more area should be devoted to I-Cache and less to D-Cache. Compiling techniques aimed at minimizing the code size will help as well. In addition, there is a greater need to replace some of the code using hardware units. A detailed hardware/software co-design of the TCP/IP protocol processing is an interesting research direction.

3.3 Execution time analysis of cache behavior

The previous subsection presents the behavior of L1 instruction cache as a function of various architectural parameters. However, we are more concerned with the impact of these parameters on the categorized execution time, the results of which are shown in Figure 4.

It can be seen that CPU execution time accounts for less than a half of the total time in the baseline cases, where the configuration has 16KB 4-way set associative L1 instruction cache with 32-byte line size. In these baseline cases, more than half of the total execution time is spent in waiting for completion of data or instruction accesses. Instruction access time is at least 3 to 4 times longer than data access time when instruction cache performs worse than the baseline cases. It can also be seen that the total execution time is more sensitive to instruction cache performance. For instance, the reduction in execution time is 26% by increasing the cache size from 16KB to 32KB or 32KB to 64KB. Improving instruction cache performance has the greatest impact on overall performance. Data cache has almost negligible impact (around 1% improvement at best) on the overall performance compared to its counterpart.

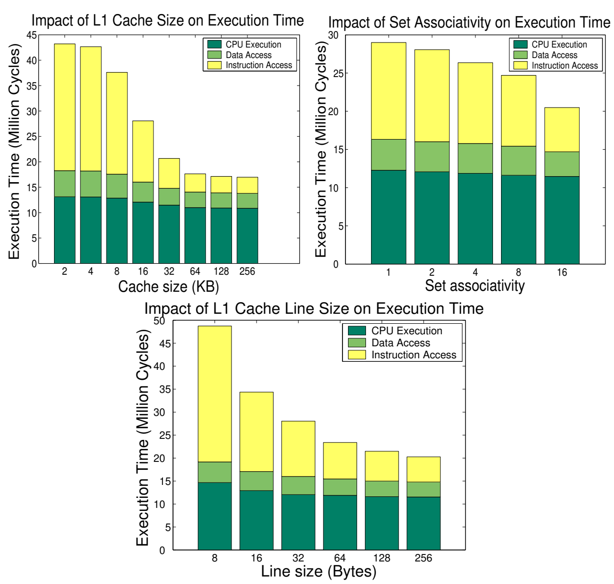

Figure 4: Impact of cache parameters on execution time (upper left and right: size, associativity; bottom: line size)

4. INSTRUCTION-SET OPTIMIZATION FOR NON-DATA-TOUCHING OPERATIONS

Non-data-touching operations dominate the protocol processing latency of the synthetic workload. Based on results in previous sections, instruction cache plays a more important role and therefore should be improved first. In this section, We will extend the ISA with some new instructions so that the frequently used instruction pairs can be replaced with corresponding new instructions. By eliminating frequently used instruction pairs, we expect to not only reduce the total number of instructions executed to process each packet, more importantly, but also improve instruction cache performance and decrease the execution time. The overall performance with this new ISA is evaluated using SimpleScalar.

4.1 Frequent instruction pairs

Frequent instruction pairs are also present in a lot of other network-related programs, e.g., ADD-SUBCC is the most frequent instruction pair occurring in CommBench benchmark programs [21]. The idea of replacing conventional frequent instruction pairs with single instructions has been brought forth by many researchers. The destination register of the first instruction has to be one of source registers of the second instruction. This dependence makes it possible to combine the instruction pair into a single new instruction that performs the same functionality. We extend the simulator to obtain profiling of instruction pairs with such dependence. We get the most frequent instruction pairs as shown in Table 1 with the extended simulator. To concentrate on the behavior of non-data-touching operations, we turn off data-touching operations such as checksum computation.

Table 1: Most frequent pairs with dependence

1st Instruction	2nd Instruction	Occurrence
ADDIU	BNE	4.91%
ANDI	BEQ	4.80%
ADDU	ADDU	3.56%
SLL	OR	3.38%
LUI	ADDU	2.50%
SLTUI	BNE	2.47%
ANDI	SRL	2.47%
SRL	ANDI	2.45%
ADDU	LW	2.29%
ADDIU	BEQ	2.26%

Table 1 shows that the first instruction of most frequent instruction pairs is mostly integer computation instruction. Non-data-touching operations such as header validation and packet classification need to first compute the values and then decide to continue processing or not. That is why such instruction pairs as compute-branch and comparison-branch occur so frequently in protocol processing.

4.2 ISA extension

We elect not to make radical changes to the original ISA. The instruction format and encoding are not changed. Only some new instructions, having the same format as old ones, are introduced.

Note that some of the instruction pairs cannot be combined in a single instruction, e.g., ADDU-ADDU, which may have 5 different register operands and it is impossible to put all of them in a single instruction. We elect to optimize those pairs that have either integer or logic operations followed by branch instructions, which are chosen from Table 1.

The first instruction of the selected pairs has to be either R-type or I-type instruction[1]. By taking advantage of unused bits in an instruction word, all the new instructions have either three register operands and one immediate operand for target address, or two register operands and two immediate operands (one for immediate number, the other for target address). Due to this limitation, only

[1]The format of R-type instructions is opcode rd, rs, rt; the format of I-type instructions is opcode rd, rs, imm. Details can be found in [17].

two types of instruction pairs are considered suitable for replacement, as shown in Table 2. Each type of instruction pairs can be further divided into R-type and I-type.

Table 2: Types of candidate instruction pairs

Type	Conditions	R-type example	I-type example
1	*rd1 == rs1 and rd1 == rs2*	*addu \$2,\$2,\$3 beq \$2,\$5,\$L1*	*addiu \$2,\$2,5 beq \$2,\$5,\$L1*
2	*rt2 == \$0 and rd1 == rs2*	*addu \$2,\$3,\$4 beq \$2,\$0,\$L1*	*addiu \$2,\$3,5 beq \$2,\$0,\$L1*

In type one instructions, the source register *rs* of the first instruction, denoted as *rs1*, is the same as the destination register *rd1*; in type two, register *rt* of the second instruction, *rt2*, is \$0. In both types, the destination register of the first instruction (*rd1*) is the same as one of the source registers of the second instruction (*rs2*). These two types are divided in such a way that the total number of operands can only be 4, either three different register operands and one target address, or two different registers, one immediate number, and one target address (register \$0 in type 2 instructions can be safely omitted since it always holds 0).

Since we combine two operations in one, which means that the integer operation (integer computation or logic operation) and the branch operation must be done in one cycle, we need a simpler ALU to do the comparison and branch. The additional simpler ALU won't introduce significant delays to the data path, therefore the ISA with this minor modification will have nearly the same CPU cycle time as the original one.

4.3 Performance evaluation

We have sixteen alternatives to implement the above ISA extension. We can optimize I-type, R-type, or both for either type 1, type 2, or both instruction pairs. We name these alternatives using two letters, which represent the type of optimization for the first and second instruction, respectively. For instance, `rb` stands for optimizing R-type pairs of type 1 and both R- and I-type of type 2 pairs, while `--` stands for no optimizations, which can be taken as the baseline case for comparisons; `i-` stands for optimizing I-type pairs of type 1 only and no optimization for type 2 pairs.

We apply the ISA extension to the simulator, compile TCP/IP protocol using new instructions to replace corresponding pairs for each alternative optimization, and run the protocol stack with synthetic network traffic trace in the simulation environment. Results of performance improvement due to this optimized ISA and the corresponding execution time break-down are shown in Figure 5.

It is found that the number of instructions reduced is not necessarily proportional to the execution time improvement. Therefore, performance improvement of optimizing both type 1 and 2 pairs will not necessarily be the sum of them. I-type and R-type optimizations have different impacts on instruction cache behaviors. Optimizing I-type pairs of both type 1 and 2 has the highest improvement, which is 23% in total and twice as much as the second highest one. We further break down the improvement into 3 major sources: CPU execution, data access, and instruction access, as we did in previous sections. The result is shown in the bottom figure in Figure 5. The improvement comes mainly from improvement of instruction cache accesses. For the case with highest improvement, the CPU stall time due to instruction cache misses improves 47%. On the contrary, CPU execution time and data access time improve 3% and 14% respectively.

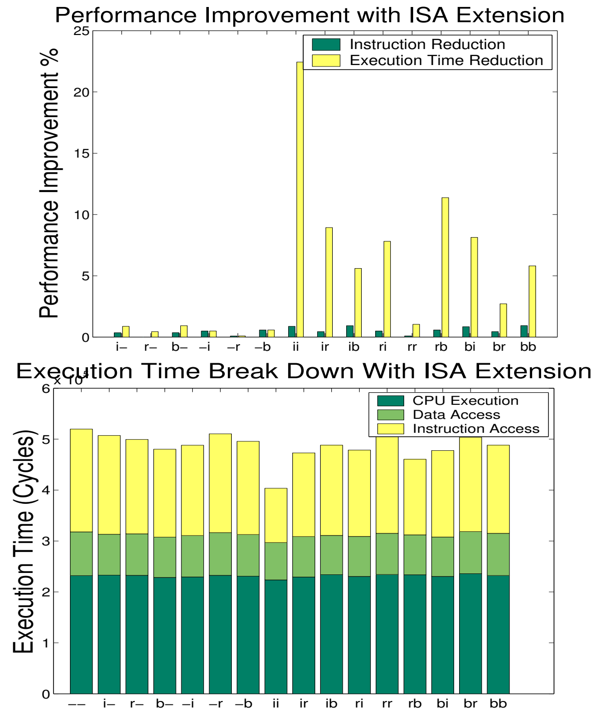

Figure 5: Performance of extended ISA (top: improvement of all configurations, bottom: execution time break-down; −−=no optimization, i=I-type, r=R-type, b=both I- and R-type)

5. RELATED WORK

Many researchers have addressed issues of network protocol performance in the past few years. A lot of techniques have been devised to speed up the protocol processing. The research can be broadly divided into two categories. The first is to design and implement low-latency and high-throughput protocols, or to optimize current TCP/IP protocol stack with either the help of compilers to improve the protocol processing latency[15][16] or hardware units to eliminate the data-touching operations such as checksum computation [1][2]. The second is to design new architectures such as network processors to optimize protocol processing [9][13].

Most of the researchers focus on improving the performance of TCP/IP protocols by designing new mechanisms or optimizations. A relatively small number of researchers have studied various architectural characteristics of network protocols.

Some researchers found that the memory system is a key factor in dominating protocol latency and throughput, and the instruction cache has a greater impact on latency [3][15][16]. Erich Nahum *et al.* analyzed the cache behavior of network protocol in *x-kernel* [10]. They found that larger caches and increased associativity improve the performance. However, *x-kernel* is designed as a platform for quickly developing efficient network protocol software and is not implemented in the protocol stacks of real operating systems. We choose the implementation from FreeBSD because it is widely used in the variants of BSD systems and its derivatives exist in a lot of other commercial operating systems. The software structures have not been radically changed in the derivative protocol implementations in various commercial operating systems, although they have been improved over time. We also identified the peculiarity of TCP/IP protocol processing and compared its performance with SPECint benchmark programs.

In [15], the authors explored such compiler techniques as outlining, cloning, and path-inlining to reduce processing latency by improving the instruction cache effectiveness. Another solution to speed up TCP/IP protocol processing is the design of special hardware units to eliminate bottlenecks such as checksum computation and data movement. This solution has been explored in much detail. One example is offloading TCP/IP protocol processing onto intelligent network interface cards [2][1]. However, this solution is not effective in improving protocol processing latency, especially for small messages dominant in local area networks.

Network processors have emerged as another totally new and different solution to speed up protocol processing. Most of the network processors have been optimized with new instruction sets and various hardware supports (such as hardware multi-threading) to reduce processing latencies. However, network processors are not general-purpose processors for end hosts to improve the performance.

6. CONCLUSION

In order to evaluate the performance of TCP/IP protocol and study the headroom of improving protocol processing latency, we ported the complete TCP/IP protocol stack from FreeBSD operating system, which contains the standard implementation of TCP/IP protocol suite, to SimpleScalar simulation environment. Choosing the TCP/IP protocol stack that is widely adopted in both BSD-variant or BSD-derivative operating systems is more meaningful to capture the characteristics of the real systems.

We evaluated the performance of TCP/IP protocol stack and studied such architectural characteristics as ILP and cache behavior. We also identified the peculiarity of TCP/IP protocol processing and compared it with SPECint benchmark programs. They turn out to be quite different due to the unique processing structure and data sets of TCP/IP.

We also extended SimpleScalar simulation environment by dividing execution time into CPU execution time, data access time, and instruction access time. Thus, we could see clearly how the architectural parameters affect the overall performance of protocol processing.

Finally, in order to improve instruction cache which has greater impact on processing latency, frequent instruction pairs were analyzed. To optimize non-data-touching operations, we incorporated new instructions into the SimpleScalar ISA to replace the frequently used instruction pairs. The extension to the original ISA will not introduce much delay into the data path. The reduction in the number of total instruction improves effectiveness of instruction cache. The overall improvement is 23% with only 1% percent of instructions being eliminated when I-type pairs of both type 1 and type 2 are replaced. These results can be used to help design new programmable protocol processors in the future.

7. REFERENCES

[1] Boon S. Ang. An Evaluation of an Attempt at Offloading TCP/IP Protocol Processing onto an i960RN-based iNIC. *HP Labs Technical Report HPL-2001-8*, January 2001

[2] M. Benz. An Architecture and Prototype Implementation for TCP/IP Hardware Support. *TERENA Networking Conference 2001*, Turkey, May 2001

[3] Trevor Blackwell. Speeding Up Protocols for Small Messages. *ACM SIGCOMM Symposium on Communications Architectures and Protocols*, 1996

[4] Ross Callon. Predictions for the Core of the Network. *IEEE Internet Computing*, 4(1), February 2000

[5] Cisco CSS 11000 series content services switches, http://www.cisco.com/warp/public/cc/pd/si/11000/

[6] Ethereal Network Analyzer, http://www.ethereal.com

[7] Foundry Networks, Inc. BigIron, http://www.foundrynet.com/products/l3backbone/bigiron/

[8] A. Gallatin, J. Chase, K. Yocum. Trapeze/IP: TCP/IP at Near-Gigabit Speeds. *USENIX Technical Conference*, June 1999

[9] T. Halfhill. Intel Network Processor Targets Routers. *Microprocessor Report*, September 1999

[10] Norman Hutchingson, and Larry Peterson. The *x*-kernel: An Architecture for implementing network protocols. *IEEE Transactions on Software Engineering*, 17(1):64-76, January 1991

[11] J. Kay and J. Pasquale. The Importance of Non-Data Touching Processing Overheads in TCP/IP. *Proceedings of the SIGCOMM '93 Symposium on Communications Architectures and Protocols*, September 1993.

[12] J. Kay and J. Pasquale. Profiling and reducing processing overheads in TCP/IP. *IEEE/ACM Transactions on Networking*, Vol. 4, No. 6, December 1996

[13] K. Krewell. Rainier Leads PowerNP Family. *Microprocessor Report*, January 2001

[14] M.K. McKusick, K. Bostic, M.J. Karels, and J.S. Quarterman. *The Design and Implementation of the 4.4BSD Unix Operating System*. Adison-Wesley Publishing Company, MA, 1996

[15] D. Mosberger, L.L. Peterson, P.G. Bridges, and S. O'Malley. Analysis of Techniques to Improve Protocol Processing Latency. *Proceedings of SIGCOMM '91 Symposium on Communication Architectures and Protocols*, 1996

[16] E. Nahum, D. Yates, J. Kurose, and D. Towsley. Cache Behavior of Network Protocols. *SIGMETRICS*, 1997

[17] SimpleScalar LLC, http://www.simplescalar.com

[18] Jonathan M. Smith. Seleted Challenges in Computer Networking. *IEEE Computer*, 32(1), Jan 1999

[19] Standard Performance Evaluation Corporation, SPEC CPU 2000 Version 1.2, December 7, 2001

[20] K. Thompson, G. Miller, and R. Wilder. Wide Area Internet Traffic Patterns and Characteristics. *IEEE Network*, 11(6):10-23, November 1997

[21] T. Wolf and M. Franklin. Commbench: A Telecommunications Benchmark for Network Processors. *IEEE International Symposium on Performance Analysis of Systems and Software*, April 2000