Embedded TCP/IP Controller for a RISC-V SoC

Chun-Jen Tsai† and Yi-De Lee
†cjtsai@cs.nctu.edu.tw
Department of Computer Science, National Yang Ming Chiao Tung University
Hsinchu, Taiwan

***Abstract*—In this paper, we present the design of an open-source RISC-V application processor with an embedded TCP/IP network module. Traditionally, the TCP/IP stack is a software layer of the OS kernel due to its complex control behavior. However, previous studies show that a hardwired logic can perform the TCP/IP control algorithms much more efficiently than a software implementation. However, to allow a processor to invoke a hardware TCP/IP logic efficiently is not a trivial task. This paper proposes an efficient interface logic between the processor core and the hardware TCP/IP stack through user-defined RISC-V instructions. The proposed architecture is implemented and verified on a Xilinx FPGA development board. Experimental results show that the average end-to-end packet delay can be reduced by up to 99% using the proposed network module when compared against the software network stack under the FreeRTOS real-time operating system. Therefore, the proposed architecture can be very useful for deeply-embedded IOT devices where a low-power processor can be used to handle low-latency high throughput IP packet transmissions.**

Keywords—application-specific processors; TCP/IP hardware; RISC-V SoC design; application-specific ISA; processor architecture; FPGA

I. INTRODUCTION

Traditional computing systems which implement the TCP/IP network stack in software suffer higher transaction latency. Previous studies showed that an IP packet gains extra latency of 15-20 μsecs just to go through the Linux network stack [1][2]. Such extra overhead comes from the inefficiency of the CPU in network protocol processing. The TCP/IP network stack requires multithreading synchronization control, bit-level manipulation of datagrams, and constant data copying operations. These computing behaviors are less efficient for instruction-based processors. Therefore, TCP/IP offload engines have been proposed to relief the CPU from part of the computing loads, such as MAC address filtering or checksum computations, using custom hardware accelerators [3][4]. However, for Internet-of-Things (IOT) devices with a weak CPU, the firmware architecture required to communicate with the offload engine for TCP/IP packet communications may still be costly. On the other hand, a fully-hardwired network stack can efficiently perform high degree of task- and data-level parallel operations, flexible bit-level data manipulations, and internal burst data copying without any overhead induced by instruction executions or system local bus transactions.

There have been several projects on complete hardware implementations of the TCP/IP network stack either as a stand-alone IC [5][6] or a reusable IP [7][8][9]. A stand-alone IC often connects to the processor using a general-purposes off-chip bus such as the SPI bus [5] that limits the packet throughput. In addition, the incoming/outgoing packets have to go through both the main memory and the data cache and suffers extra memory overhead. Lightweight implementations of the TCP/IP stack as soft-core IPs are presented in [7] and [8] using HDL. However, the design focuses of both researches are mainly on the IP alone without touching the issues of processor integration. There are some designs targeted for specific networking applications [10][11][12][13]. Nevertheless, application-specific integration of the TCP/IP hardware IPs into an SoC are usually more straightforward. The systems for packet filtering in [10] and for high-frequency trading in [11] only require one-time incoming packet processing on-the-fly, therefore, both systems integrate the TCP/IP hardware stack with a custom logic for information extractions from the IP packets. The integration of the processors and the network modules are not tightly-coupled. A mission-critical nuclear power plant control system is presented in [12] where tens of thousands of embedded sensors must collect data and send the information back to control servers. In order to guarantee hard real-time data transmissions, each embedded sensor module contains a microcontroller with hardwired TCP/IP module interfaced by memory-mapped registers. In [13], a small web-server interface is presented for an embedded microcontroller. Dual-port memory blocks are used as memory-mapped interface buffers between the AVR processor core and the hardware TCP/IP module. The system only uses on-chip memory and is not intended for general-purpose applications. In summary, all the aforementioned systems adopt simplified application-specific interfaces between the processor and the TCP/IP stack.

In this paper, we present the architecture design of a RISC-V application processor with an embedded network module for general-purpose applications. The system should contain DRAM as the main memory and relies on a data cache to increase the memory performance. For the TCP/IP stack, the hardware module presented in [9] is selected. The work in [9] is unique in that the high-level synthesis (HLS) design flow using C++ has been adopted for the development of the hardware model. HLS flow reduces the design effort significantly. It is also easy to modify the model to incorporate new network protocols. To integrate an HLS logic with other HDL models, EDA tools can be used to convert the HLS model to an RTL model in HDL such as Verilog or VHDL. However, the TCP/IP controller in [9] uses an AXI-stream interface to exchange control signals and data with the user logic. It is easy to connect the controller to a custom logic for application-specific networking. Nevertheless, to integrate the logic into a general-

purpose application processor, some interface logic must be carefully designed to maintain the efficiency for instruction-based invocations and data exchange.

A communication IP typically has two types of interfaces: the low-bandwidth control plane and the high-speed data plane. Therefore, for the processor to communicate with the TCP/IP controller, the interface logic should provide two different instruction-driven communication schemes. For the control plane, the traditional memory-mapped I/O interface can be used. However, for the data plane, the processor and the TCP/IP logic should exchange data directly through the data cache or the performance will be hindered significantly. The idea of direct cache access (DCA) has been proposed in [14]. According to the analysis based on workload simulations in the paper, the DCA scheme can reduce the cache read misses by up to 11% for networking applications. However, the study in [14] is not conducted using a real system.

The DCA scheme is adopted in this paper to allow highly efficient data exchange between the processor pipeline and the TCP/IP controller using the L1 data cache of the processor core. The data exchange operations are triggered by application-specific user-defined instructions of the RISC-V ISA. Therefore, the proposed approach makes it programmable and can be applied across different hardware accelerators, not just the TCP/IP controller. Our experiments show that using DCA can reduce the end-to-end packet delay by five times slower than using the traditional DMA when the packet size is larger than 512 bytes.

The contribution of the paper is as follows. First, we have designed a direct cache access controller and proposed some application-specific instructions to significantly reduce the overhead of packet processing while maintains software programmability. Secondly, we have implemented a socket API wrapper for the FreeRTOS kernel to interact with the hardware TCP/IP stack. Both the RTL model of the proposed SoC and the FreeRTOS socket API wrapper will be made open-source for other researchers to verify and improves the proposed architecture [1]. The organization of this paper is as follows. In section II we give an overview of the proposed application processor. Section III presents the design details of the key modules of the SoC. Section IV shows the experimental results. Finally, conclusions and future work are given in section V.

II. Overview of The Proposed RISC-V SoC

An overview of the application processor architecture is shown in Fig. 1. The SoC contains an open-source RISC-V RV32-IMA processor core, Aquila [15]. The Aquila core is a reusable IP core that integrates a classical five-stage in-order single-issue pipeline, an on-chip dual-port tightly-coupled memory (TCM) block, 4-way set associative L1 instruction and data caches, an atomic-instruction monitor (within the core), and a Core-Local Interrupt (CLINT) module with a system timer. It supports RISC-V CSR instructions and is capable of executing multi-threading real-time OS such as FreeRTOS. The RTL model of Aquila is written in Verilog. When synthesized using

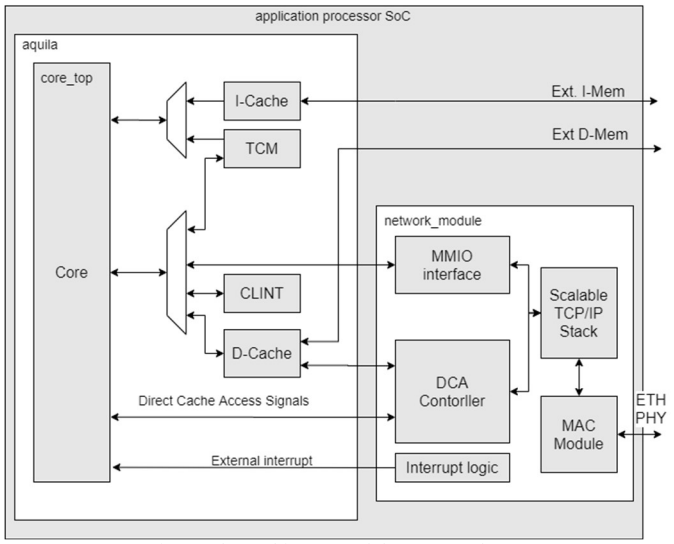

Fig. 1. The architecture of the proposed SoC.

a Xilinx FPGA, the performance of the Aquila core is 0.97 DMIPS/Mhz.

In this paper, the Aquila core is extended with a network module that contains a DCA controller, a TCP/IP hardware stack, and an MAC IP. The DCA module is used for direct access of the CPU data cache from the accelerator. In other words, the DCA controller allows the processor and the hardware TCP/IP stack to share the data cache. The role of the DCA is crucial to the performance of processing IP packets. Without the DCA controller, the incoming packets received by the hardware TCP/IP stack will be stored in the main memory. The processor will not be able to process the packets wihtout going through a series of cache misses. Similarly, for the processed outgoing IP packets in the data cache. The processor has to flush out the packets before the TCP/IP stack can grab the packets and send them to the Ethernet MAC for transmissions.

Although Aquila supports the ARM AXI bus protocol, where the Accelerator Coherence Port (ACP) is often used for an hardware accelerator IP to access the data cache of the processor port directly. However, we do not choose to support the ACP bus to share the data cache with the hardware TCP/IP stack. Using ACP will invole a more sophisticated implementation of the AXI bus infrastructure and the goal of the application processor is to support the lightweight deeply-embedded applications of IOT devices that require high throughput network connections. With the proposed design, the Aquila core and the embedded network module can flexibly used to construct an SoC with or without the AXI bus infrastructure.

The MMIO interface in Fig. 1 provides a slave interface for the processor to control the hardware TCP/IP stack through memory-mapped I/O instructions. In addition, user-defined RISC-V instructions are implemented in Aquila to allow highly efficient data exchange between the RISC-V core and the

[1] The source code of this work is available at https://github.com/eisl-nctu/aquila-net

TCP/IP controller though the DCA controller. The details of the instruction extension will be presented in the next section.

Finally, the scalable TCP/IP stack in Fig. 1 is from the open source project developed by the system group of ETH Zürich [9]. The logic is designed using the HLS approach with C++. Xilinx Vitis EDA tool is used to synthesize the C++ model into an RTL model for integration into the proposed SoC. The TCP/IP logic is then connected to an MAC IP from the Verilog Ethernet project [16] for ethernet transmissions. In the next section, we will focus on the main contribution of this paper, namely, the designs of the MMIO interface, the DCA controller, the user-defined RISC-V isntructions for IP packet processing, and the socket API wrapper library.

III. Architecture of the Proposed Network Module

A. The Software Interface Module

In order for the processor to talk to the hardware TCP/IP stack for packet transmission, a glue logic is required since the processor only performs word-based load/store operations while the sclable TCP/IP stack is designed to use the AXI stream bus for communication. The AXI stream bus is a lightweight AXI bus that is optimized to support only in-order burst transmissions of data words. As Fig. 3 shows, the The MMIO interface module is designed to be a middleman logic to statisfy the communcaion requirements on both ends. However, the MMIO interface only establish the contorl plane between the processor and the TCP/IP controller, not the data plane. The data plane will be handled by the DCA controller interface to be discussed in section III.B.

The hardware TCP/IP stack is an AXI stream device with both mater and slave interfaces. There are five signals in an AXI stream bus: ready, last, keep[7:0], data[63:0], and valid. The MMIO interface maps the first four signals to the processor address space, one signal per one or two words, so that the processor can use the load/store instructions to control these signals. For example, for the processor to transfer a control word to the TCP/IP stack, the following hand-shaking protocol will be used:

1. Read the ready signal to see if the device is ready.
2. Control the last and the keep signals for marking the data beats and the end of a burst.
3. Write each 64-bit data beat to data[63:0] with two store instructions. The valid signal will be raised automatically by the MMIO interface to inform the TCP/IP stack the arrival of a data beat.

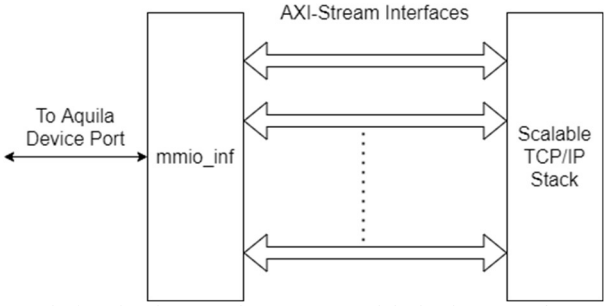

Fig. 3. The interface between the processor and the hardware TCP/IP stack.

The addresses of these signals are mapped to the memory address space that are reserved for the local I/O devices. The

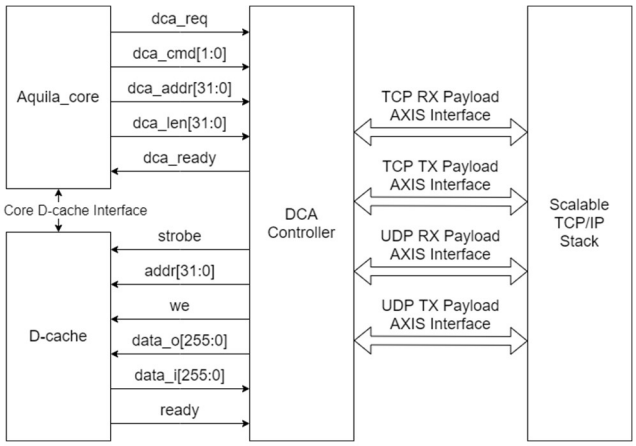

Fig. 2. The interface of the Direct Cache Access Controller.

Aquila core supoorts two types of I/O devices, the local on-chip I/O devices managed by a local bus decoder and the external devices managed by the AXI bus decoder. Therefore, accesses to the local devices do not have to go throuth the AXI bus decoder and has lower latency.

B. The Direct Cache Access Controller

While the MMIO interface handles the control plane of the communication channel between the processor and the TCP/IP stack, the DCA controller handles the data plane between the two modules. However, since the payload of an IP packets can contain over a thousand bytes, the processor cannot possibly use the register file to receive the data packets directly. If the main memory is used as the buffer for packet reception, the performance will be hindered significantly for the succeeding data accesses to the payload data due to large amount of cache misses.

A more efficient way is to allow the TCP/IP stack to send the packet data directly to the data cache of the processor (or retrieve the packet data from the cache for outgoing transmissions). For the proposed SoC, a direct cache access interface is added to the data cache of the Aquila core for this purpose. For our current implementation, the interface does not work concurrently with the main cache interface to the processor core because the processor pipeline will be stalled when an DCA instruction is executed. The interfaces among the processor core, the data cache, the DCA controller, and the TCP/IP hardware module is shown in Fig. 2.

In order to increase the efficiency of the accelerator accesses to the data cache while at the same time simplify the DCA controller logic, the data transfer size of the DCA port allows only one cache block per transfer (256 bits in our implementation). For an IP packet that is not of the size of multiples of 256-bit, zero-padding will be applied to the tailing bytes. The normal load/store instructions of the RISC-V ISA use the registers as the source or target of the data transfer. Therefore, each instruction can only move 32- or 64-bit of data. With the DCA instructions, each transfer can move 256-bit of data, which is 8 times faster than the normal load/store instructions.

Therefore, new instructions must be defined in the ISA to enable such data transfer operations. The design of the DCA instructions will be discussed in section III.C.

When the processor core executes a DCA instruction, the control signals decoded from the instruction will be send to the DCA controller. The signal dca_cmd[1:0] contains the type of the operation. When the DCA controller is done with the operation, the dca_ready signal will be raised to inform the processor that the operation is done and the requested data has been stored in the data cache or transferred to the TCP/IP hardware. Note that in the current implementation, the DCA controller is not responsible for buffering the incoming packets. The hardware TCP/IP stack has a small buffer for the incoming packets. To avoid overflow of the buffer, the software API library has to execute the DCA instructions to pull the packets to user space buffers in time. Since Aquila is a single-issue in-order pipelined processor, the processor pipeline will be stalled during the execution of a DCA instruction. Therefore, the processor and the DCA controller do not access the data cache concurrently. In other words, when the DCA controller takes control (due to the execution of a DCA instruction), it has exclusive access to the data cache. The processor core will not resume execution until the DCA controller finish its manipulation of the data cache. This design simplifies the hazard handling circuit since the succeeding instructions have high probability of depending on the content in the data cache and such hazard cannot be easily handled by forwarding.

Since the DCA controller moves data on a cache-block basis, the data destination/source in the cache must be aligned to the cache block boundaries to avoid the extra overhead of handling packets that lay across the cache blocks. Therefore, the application software must be implemented carefully to make sure all packet buffers are aligned to the cache block boundaries (e.g. multiples of 32-byte in our implementation). This is a minor constraint in software programming since most compilers can apply pragma to set such constraints for all pre-declared data structures, and it is easy to adjust the dynamic data allocated through malloc() to satisfy such a requirement as well.

C. The User-defined Instructions for the DCA operations.

Modern ISA designs typically allow for user-defined instruction extensions. A user-defined instruction does require system bus traffics for the processor core to communicate with the custom IP. In addition, the parameters for the operations and the returned value can be passed in-and-out directly using the processor registers.

In the proposed architecture, we define some DCA instructions to allow the programmer to control the transmissions of data packets between the TCP/IP hardware and the data cache. The RISC-V custom-0 (with opcode 0001011) user-defined instruction is selected for the proposed DCA instructions. The instruction format is of R-type as shown in Fig. 4. The 12th bit of the instruction codeword specifies whether a direct cache load (1) or store (0) operation is requested. The 13-bit specifies whether a UDP (1) or a TCP (0) packet transmission is requested. The source register field rs1 stores the base address of the packet buffer and the source register field rs2 stores the number of bytes to be transferred between the data cache and the TCP/IP hardware. When the DCA instructions are executed, the control signal dca_strobe in Fig. 2 will be raised to the DCA controller, the dca_cmd signal specify one of the possible operations: TCP store (00), TCP load (01), UDP store (10), and UDP load (11). The parameters, the based address of the packet buffer and the length of the buffer in bytes, for each operation will be extracted from the registers rs1 and rs2 and converted to the signals dca_addr[31:0] and dca_len[31:0]. When the operation is completed by the TCP/IP hardware, the dca_ready signal will be acknowledged.

Fig. 4. User-defined DCA instructions.

D. The Socket API Library.

Although the DCA instructions can be used to send/receive the IP packets, it is less convenient to use than a standard socket API library. Therefore, we have also implemented a subset of the standard Berkeley Socket API that uses the DCA instructions to establish connections and send/receive packets. The implemented API functions include: socket_init(), socket(), bind(), connect(), accept(), listen(), recv(), send(), recvfrom(), sendto(), and close(). Since socket programming typically requires multi-threading, it is important to make the library functions thread-safe. The operating system we have used to provide multi-threading capability and mutex protection of the shared resources is the FreeRTOS real-time OS. There are three main threads in the socket library once a socket is created for packet reception: tcp_accept_poll(), tcp_recv_poll(), and udp_recv_poll(). A link list data structure is used to store the packets from each socket. Each thread is triggered using the interrupt-driven mechanism. The incoming packets to the TCP/IP hardware device will cause I/O interrupts that wake up the corresponding threads to handle the packets and sort them into the corresponding packet buffers.

On the other hand, sending IP packets is relatively simple since there is no need to maintain a complex data structure for outgoing packets. A DCA instruction will be executed to cast out the packet for the API functions such as send() or sendto(). However, these functions should still be protected by mutexes to ensure the functions to be thread-safe.

IV. EXPERIMENTAL RESULTS

The synthesizable RTL model of the proposed application processor is written in Verilog and HLS C++ (only for the TCP/IP stack). It has been verified on a Xilinx KC-705 FPGA development board using Vivado 2021.1. The target frequency for the Aquila SoC is 100MHz. The cache size is set to 32 KB for both the instruction and data caches. The overall resource utilization of the application processor SoC on a Xilinx Kintex-7 XC7K325T FPGA device is shown in TABLE I. The break-down numbers of resource usages of the key components are shown in TABLE II.

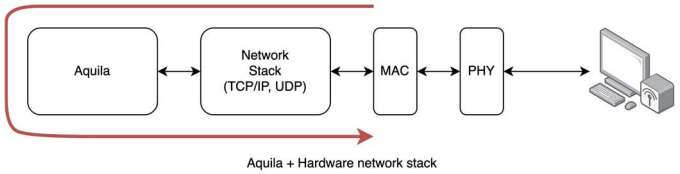

Fig. 5. Illustration of the round-trip latency of an echo packet. The block "Aquila" stands for the application-level software executed by the μP.

For performance evaluation, a simple echo server is used to measure the round-trip latency of packet transmissions. A hardware counter is implemented around the I/O ports of the MAC module to calculate the round-trip time of an echo packet within the network subsystem of the application processor, as shown in Fig. 5. To be more specific, we measure the time it takes to receive a packet and echo it back, starting from the input of the MAC module, through the application layer, then back to the output of the MAC module. The proposed hardware-based TCP/IP stack is compared against a common software Lightweight TCP/IP (LwIP) library on the same application processor under FreeRTOS.

A. Comparisons between Hardware and Software TCP/IP

TABLE III shows the performance differences between the software and hardware approach using the low-level API. For LwIP, low-level API means that the API functions interact with the TCP/IP stack directly for sending/receiving packets without maintaining their own link list buffers or handling thread-safe synchronizations among multiple threads. For the proposed approach, this means the echo server receives and sends back packets using the DCA instructions directly instead of using the socket API library described in section III.D. This table shows the maximal potential of the proposed hardware-based TCP/IP approach in packet transmissions. As one can see, the average hardware end-to-end packet delay can be 107.3 times lower than that of the software approach. Note that the entries in TABLE III shows the end-to-end packet delay cycles, which include the reception of the packet data from the MAC module, moving the packet data into the data cache, and sending the data out to the MAC module.

TABLE IV shows the comparison when the more sophisticated socket API library is used. Again, the hardware packet delays are much lower than the software ones. The average hardware packet delay is 3.5 times lower than that of the software. Since a major portion of the end-to-end packet delay comes from the execution of the socket API library, the proposed hardware has no advantage than the LwIP software in this regard unless the socket API library can also be accelerated by the hardware. However, 3 times average speed up is still a significant achievement for the proposed architecture. One thing we do have noticed in the socket API library is that mutex protection overhead is quite high, probably due to the unoptimized mutex system call of the FreeRTOS for the Aquila SoC.

To better understand the effectiveness of the direct cache access scheme, we have made a comparison on the packet delay when the DCA controller is turned on or off. TABLE V shows the results. As one can see, as the packet size becomes large, the performance degradation due to cache misses can become large. When the packet size is of 1024 bytes, the end-to-end packet delay without using the DCA controller can be more than 6 times higher than using DCA.

TABLE I. Resource usage on Xilinx Kintex-7 XC7K325T.

	LUT	Flip-Flop	BRAMs	DSPs
Entire SoC	83,239	101,251	172	4
FPGA %	40.84%	24.84%	38.65%	0.48%

TABLE II. Resource usages of the key components.

	LUT	Flip-Flop	BRAMs	DSPs
Aquila Core	22,984	28,954	50	4
MMIO interface	577	1,025	0	0
dca_controller	288	100	0	0
tcp_ip_stack	44,287	54,735	118	0
mac_module	1,111	2,084	3	0

TABLE III. The end-to-end packet delay in clock cycles using the low-level API for both the LwIP and the proposed HW.

#bytes	LwIP software		Proposed SoC		Latency reduction	
	TCP	UDP	TCP	UDP	TCP	UDP
32	10,642	6,240	242	167	44.0×	37.4×
128	15,619	10,561	293	194	53.3×	54.4×
256	22,207	16,323	358	234	62.0×	69.8×
512	35,597	27,840	497	314	71.6×	88.7×
1024	62,232	50,879	776	474	80.2×	107.3×

TABLE IV. The end-to-end packet delay in clock cycles using the socket API for both the LwIP and the proposed HW.

#bytes	LwIP software		Proposed SoC		Latency reduction	
	TCP	UDP	TCP	UDP	TCP	UDP
32	22,217	12,831	5,343	5,147	4.2×	2.5×
128	27,632	18,422	7,593	7,379	3.6×	2.5×
256	35,410	26,439	10,580	10,338	3.3×	2.6×
512	50,442	41,797	16,578	16,275	3.0×	2.6×
1024	80,554	72,584	28,643	28,224	2.8×	2.6×

TABLE V. The packet delays when DCA is off or on using the low-level API with the hardware TCP/IP stack.

#bytes	DCA off		DCA on		Latency reduction	
	TCP	UDP	TCP	UDP	TCP	UDP
32	343	291	242	167	1.41×	1.74×
128	644	563	293	194	2.19×	2.90×
256	1,036	951	358	234	2.89×	4.06×
512	1,821	1,619	497	314	3.66×	5.15×
1024	3,389	3,027	776	474	4.36×	6.38×

B. Comparisons between the ARM Cortex A9 software and the proposed Aquila SoC

In this section, we use a commercial ARM Cortex A9 core to compare against the proposed soft-core Aquila SoC. Note that both the ARM Cortex A9 and the Aquila cores are 32-bit processors. However, the ARM is running at 667MHz and the Aquila is running at 100MHz. The ARM processor we used is the Zynq 7020 SoC (on the ZedBoard platform by Digilent) and the Aquila SoC is again synthesized using KC-705. Both systems are running FreeRTOS.

TABLE VI shows the comparison of the ARM Cortex A9 running an optimized version of the LwIP for ZedBoard and the Aquila SoC with the proposed hardware TCP/IP module. Note that the low-level APIs are used for both the LwIP and the hardware TCP/IP stack. As one can see, the Aquila SoC still

outperforms the Cortex A9 processor. This is significant since the Aquila core is merely a standard 5-stage in-order single-issue processor at 100MHz while Cortex A9 is a superscalar at 667MHz.

When the socket API is used, the situation changes in favors of the ARM processor (see

TABLE **VII**). However, the main differences come from the fact that the LwIP used for the ARM platform is optimized by Xilinx for the Zynq 7020 SoC while the socket API library of the proposed SoC is not optimized yet. A deeper investigation into the hotspots of the proposed system reveals that the current implementation contains too much overhead for IP datagram copying in the proposed system. The optimization of the socket API library will be handled in the future work.

TABLE VI. Comparison to ARM Cortex A9 running LwIP using the low-level API (delays in μsec).

	ARM@667MHz with LwIP software		Proposed Aquila SoC@100MHz		Latency reduction	
#bytes	TCP	UDP	TCP	UDP	TCP	UDP
32	6.5 µs	6.7 µs	2.4 µs	1.7 µs	2.7×	4.0×
128	7.1 µs	6.9 µs	2.9 µs	1.9 µs	2.4×	3.6×
256	7.6 µs	7.1 µs	3.6 µs	2.3 µs	2.1×	3.0×
512	8.7 µs	7.8 µs	5.0 µs	3.1 µs	1.8×	2.5×
1024	10.8 µs	8.4 µs	7.8 µs	4.7 µs	1.4×	1.8×

TABLE VII. Comparison to ARM Cortex A9 running LwIP using the socket API (delays in μsec).

	ARM@667MHz with LwIP software		Proposed Aquila SoC@100MHz		Latency reduction	
#bytes	TCP	UDP	TCP	UDP	TCP	UDP
32	26.4 µs	22.6 µs	53.4 µs	51.5 µs	0.5×	0.4×
128	28.0 µs	23.6 µs	75.9 µs	73.8 µs	0.4×	0.3×
256	29.4 µs	24.4 µs	105.8µs	103.4µs	0.3×	0.2×
512	32.4 µs	26.0 µs	165.8µs	162.8µs	0.2×	0.2×
1024	35.3 µs	29.1 µs	286.4µs	282.2µs	0.1×	0.1×

V. Conclusions and Future Work

In this paper, we present the design of an application processor SoC with an embedded hardware TCP/IP stack for the deeply-embedded IOT devices. The proposed design has been implemented based on an open-source RISC-V core, Aquila, and is made open source (available at https://github.com/eisl-nctu/aquila-net) for other researchers to verify and improve our work. The proposed system can reduce the end-to-end IP packet delay by 67 times on average when compared to the popular LwIP software for embedded systems using the low-level API. When a standard socket API software is wrapped around the proposed network module, the performance gain is still 3 times better on average. Initial analysis shows that the current socket API wrapper software for the proposed hardware is not implemented efficiently. For future work, we will look into the optimization or hardware acceleration of the socket API wrapper to further improve the performance.

Acknowledgment

This work was partly funded by the Ministry of Science and Technology (MOST), Taiwan, ROC, under Grant # MOST 110-2221-E-A49-158 -, and by the Taiwan Semiconductor Research Institute (TSRI), NARL, Taiwan, ROC.

References

[1] G.W. Morris, D.B. Thomas and W. Luk, "FPGA Accelerated Low-Latency Market Data Feed Processing", *Proc. of 17th IEEE Symposium on High Performance Interconnects (HOTI 2009)*, pp. 83-89, 25-27 Aug. 2009.

[2] S. Narayan, P. Shang and N. Fan, "Network performance evaluation of Internet Protocols IPv4 and IPv6 on operating systems," *Proc. of IFIP International Conference on Wireless and Optical Communications Networks*, Apr. 28-30, 2009.

[3] S.-M. Chung, C.-Y. Li, H.-H. Lee, J.-H. Li, Y.-C. Tsai, and C.-C. Chen, "Design and Implementation of the High Speed TCP/IP Offload Engine," *Proc. of 2007 Int. Symp. on Communications and Information Technologies*, Sydney, NSW, Dec. 2007.

[4] G. Sutter, M. Ruiz, S. L'opez-Buedo, and G. Alonso, "FPGA-based TCP/IP Checksum Offloading Engine for 100 Gbps Networks," *Proc. of 2018 Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*, Cancun, Mexico, Dec. 2018.

[5] WIZnet, *W5500 Datasheet v1.0.9*, Dec. 2013.

[6] A. Choudhary, D. Porwal, A. Parmar, "FPGA Based Solution for Ethernet Controller as Alternative For TCP/UDP Software Stack," *Proc. of 2018 6th Int. Conf. on Wireless Networks & Embedded Systems (WECON)*, Rajpura, India, Nov. 2018.

[7] L. Charaabi, I. Jaziri, "Designing a tiny and customizable TCP/IP core for low cost FPGA," *Proc. of 2017 Int. Conf. on Engineering & MIS (ICEMIS)*, Monastir, Tunisia, Feb. 2017.

[8] T. Uchida, "Hardware-Based TCP Processor for Gigabit Ethernet," *IEEE Trans. on Nuclear Science*, Vol. 55, No. 3, June 2008, pp. 1631-1637.

[9] D. Sidler, G. Alonso, M. Blott, K. Karras, K. Vissers, and R. Carley, "Scalable 10 Gbps TCP/IP Stack Architecture for Reconfigurable Hardware," *Proc. of IEEE 23rd Annual Int. Symp. on Field-Programmable Custom Computing Machines*, May 2-6, 2015.

[10] K. Nakamura, A. Hayashi, and H. Matsutani, "An FPGA-Based Low-Latency Network Processing for Spark Streaming," *Proc. of 2016 IEEE Int. Conf. on Big Data*, Dec. 5-8, 2016.

[11] C. Leber, B. Geib, H. Litz., "High Frequency Trading Acceleration using FPGAs," *Proc. of 21st Int. Conf. on Field Programmable Logic and Applications*, Sep. 5-7, 2011.

[12] A. Gour, T Mathews, R P Behera, M Sakthivel, T Jayanthi, B K Panigrahi , "Development of FPGA based TCP/IP Communication Module for Embedded Systems of Nuclear Reactors," *Proc. of 2020 Int. Conf. on Emerging Trends in Information Technology and Engineering (ic-ETITE)*, Vellore, India, Apr. 2020.

[13] R. A. Melo, D. M. Caruso, S. E. Tropea, "Memory-mapped I/O over Dual Port BRAM on FPGA," *Proc. of 2012 VIII Southern Conf. on Programmable Logic*, Bento Gonçalves, Brazil, Mar. 2012.

[14] R. Huggahalli, R. Iyer, S. Tetrick, "Direct Cache Access for High Bandwidth Network I/O," *Proc. of 32nd Int. Symp. on Computer Architecture (ISCA'05)*, Jun. 4-8, 2005.

[15] The Aquila SoC project, available on-line at https://github.com/eisl-nctu/aquila.

[16] A. Forencich, the Verilog Ethernet project. Available online at Github: https://github.com/alexforencich/verilog-ethernet.