An RTOS in Hardware for Energy Efficient Software-based TCP/IP Processing

Naotaka Maruyama
Kernelon Silicon Inc.
Kugenuma-Ishigami,
Fujisawa, 251-0025, Japan

Tohru Ishihara
Kyushu University
3-8-33 Momochihama, Sawara-ku
Fukuoka, 814-0001, Japan

Hiroto Yasuura
Kyushu University
6-10-1 Hakozaki, Higashi-ku
Fukuoka, 812-8581, Japan

Abstract- **Many functions of embedded systems are implemented by software for flexibly dealing with frequent upgrade and for quickly fixing unpredictable bugs in applications. This system architecture is however generally less energy efficient than that implemented by application specific hardware. As a remedy of this issue, this paper proposes a processor-based platform having an RTOS in hardware for energy efficient and flexible TCP/IP processing. Unlike application specific hardware, implementing RTOS in hardware does not lose the flexibility of the applications while the energy efficiency is comparable to the application specific hardware. Experiments with an actual TCP/IP application demonstrate that our approach achieves a 7 times improvement in energy efficiency over an existing commercial firmware RTOS.**

I. Introduction

The strong demand of system integration, high-performance computation, and low-power consumption has made many chip vendors adopt expensive advanced semiconductor manufacturing technologies. This trend results in a significant increase of design complexity and causes an explosion of NRE (non-recurring engineering) costs. A remedy for the NRE explosion is to reduce the number of designs and sells tens of millions of chips under a fixed hardware design. In such a situation, embedded software running on embedded processors plays much more important role than today. However, a typical embedded processor used in embedded systems does not have enough performance to handle today's network applications. For example, the TCP/IP throughput by ARM9 at 50MHz operational clock is only about 11Mbps. Even in embedded systems, there is a strong demand of having a throughput of more than 100Mbps like PCs, since ever expanding ubiquitous society makes not only PCs but also various kinds of consumer products connected to the internet. Employing a high-end processor used in PCs for embedded systems is however impractical since the cost and the power consumption are too large for those embedded applications. On the other hand, protocol processing such as TCP/IP processing needs flexibility since they are too complex to be implemented by hardwire logics and they also need bug fixing, up grading, future expanding etc. after implementation. Therefore complete hardware implementation of the protocols is also impractical.

This paper proposes a processor-based platform for firmware protocol processing such as TCP/IP which is executed faster on it with lower operational clock rate than conventional software-based approaches. The platform is called ARTESSO, a Advanced Real Time Embedded Silicon System Operator. In order to accelerate the TCP/IP firmware processing with lower clock frequency, ARTESSO employs a fully hardwired RTOS. The hardwired RTOS provides rich functions which are the same as those provided by the commercial firmware RTOS, such as a priority-based First-come and First-serve scheduler, enough number of tasks, semaphores, event flags and mail boxes, and variety types of system calls. Typical TCP/IP firmware uses RTOS functions with extraordinary frequency, consequently CPU time occupancy for RTOS during TCP/IP processing is about three times much than it of essential TCP/IP processing. Therefore implementing RTOS in hardware drastically improves the performance of TCP/IP. Our RTOS achieves several tens of times higher performance compared to the conventional firmware RTOS with preserving the richness of the functionality. The key of our RTOS in hardware is a Virtual Queue. The Virtual Queue is a novel queue structure which virtually realizes a large number of queues with a small number of logic gates.

This approach does not lose the flexibility of the system since almost of all TCP/IP functions are implemented by software. In our approach only RTOS and a small number of TCP/IP functions are implemented in dedicated hardware. Those functions are matured and therefore version ups of functionality are done infrequently. In the rest of this paper, motivations of this work and related work are presented in Section II. An overview of ARTESSO and a detailed architecture of the ARTESSO RTOS are described in Section III. Section IV summarizes the performance evaluation results. Section V concludes this paper.

II. Related Work and Our Approach

A. Bottleneck of TCP/IP firmware

The network protocol firmware such as TCP/IP consists of several processes. They need running concurrently without violating their deadlines, communicating with each other, and handling interrupts frequently. Since RTOS can satisfy all of such demands, it is employed in almost of all embedded network systems.

Table 2-1 shows our analysis results of CPU time occupancy for each process during TCP/IP operation. The simulation setup used in this analysis is as follows;

- The target firmware consists of TCP, IP and Ethernet driver.
- The proprietary 32-bit RISC is used for a core CPU.
- TCP sends two 1460 Bytes frames and receives one ACK frame. This is repeated many times.

As can be seen from the result, the essential protocol processing occupies only 10.5% of the total CPU time spent for the entire TCP/IP operations. If all the other processes

Table 2-1 TCP/IP Firmware Analysis

Process Category	Time Occupancy (%)
Protocol Processing	10.5
RTOS	32.1
Header Rearrangement	15.3
TCP Checksum	32.2
Memory Copy	9.9
Total	100.0

can be more effectively processed, the CPU power dissipated for the TCP/IP processing is drastically reduced. One of the most straightforward approaches for reducing the loads of the processes is to implement the processes in hardware. However main part of the TCP/IP is still implemented by firmware to keep its flexible.

In our platform, processes for Header Rearrangement, TCP Checksum and Memory Copy are handled by simple dedicated hardware in a straightforward way which releases the CPU from these processes. Implementing above mentioned three processes in hardware does not sacrifice the flexibility of TCP/IP processing since they are commonly used basic processes.

Unlike above three processes, implementing a core process of RTOS in hardware has a lot of issues to be resolved. The following subsection summarizes the issues for implementing the RTOS in hardware.

B. *Issues in Hardware RTOS*

For improving RTOS performance, several techniques have been proposed. Some of them appropriately partitions RTOS functions into hardware and software [7-12]. Some others implement main functions of RTOS in hardware [1-6].

One of the most critical issues which prevent us to practically implement the RTOS in hardware is a large number of queues required. This increases a chip cost if they are all implemented in hardware.

In a typical RTOS, at least four states, STOP, RUN, READY and WAIT, are needed. Tasks belonging to the WAIT state wait for a trigger, such as, time out, satisfaction of event flag conditions, release of a semaphore and receiving a message of a mailbox. If the trigger occurs when there are several tasks waiting for the same trigger, the first coming task is selected based on the priority-based First Come and First Serve, FCFS, policy. If 32 semaphore IDs and 16 priorities are defined for a target system, each semaphore ID needs 16 queues, and therefore 512 queues are needed in total for the semaphore wait. Similarly, if an event flag is defined by 32 IDs and a mailbox is defined by 192 IDs, 4,096 queues are needed in total for the wait queue, which involves a huge hardware cost. If an RTOS is implemented in hardware, a hardware FIFO is typically employed for implementing a queue. Supposing an RTOS handles 32 tasks, the FIFO needs a 5-bit by 32-word memory cells. If it is implemented by flip-flops, 5 x 32 FFs are needed for each FIFO. This roughly corresponds to 960 logic gates. Since a FIFO controller needs about 190 gates, the FIFO consists of 1,150 logic gates. Totally, the gate count of entire queues in the RTOS is 4,096 x 1,150 = 4,710,400 gates.

Our new concept, virtual queue which is presented in Section III-C, can be implemented by only 38,300 gates in total. This is more than 100 times less than that of the conventional approach.

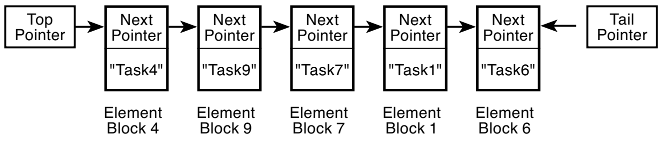


Fig. 2-1 Queue implementation in software

There are several existing implementations of RTOS in hardware; RTU [1-3] and Silicon TRON [6]. They employ only a small number of WAIT queues in hardware. For example, the Silicon TRON [6] has three events and three semaphores.

C. *Bottleneck of RTOS*

Another critical issue in a conventional RTOS is its poor performance for searching and retrieving a target element from a queue, which is an essential function for today's commercial RTOSs. According to the FCFS policy, newly coming data is appended into the end of a queue and the oldest one is read from the top of the queue. However, an element which is not located at the top of the queue sometimes has to be retrieved. This is one of essential functions in RTOS. Suppose a task is waiting for a semaphore release. If a timeout of the task occurs before the semaphore release, the task has to be removed from the wait queue. In this case, the RTOS has to search the task from the wait queue and remove it from the queue even if it is not located at the top of the queue. This function is provided by system calls in a typical commercial firmware RTOS. The hardware RTOSs presented in [1-3] and [6] do not employ the function. Therefore these are insufficient with respect to the functionality, and as a result, they are not applicable to commercial applications such as TCP/IP processing.

A conventional firmware RTOS typically employs a chaining list structure as shown in Fig.2-1 to implement a queue mechanism. If the searching and retrieving function is implemented by software, the queue operation takes long time, which degrades entire RTOS performance.

D. *Our Approach*

Our approach is to implement processes in hardware except for protocol processing mentioned in the Table 2-1. As shown in Fig.2-2 (b), memory copy, TCP checksum, header rearrangement and RTOS are realized by hardware. Those functions spend about 90% CPU time at the conventional approach. On the other hand protocol processing and application processing is still implemented by software, therefore the ARTESSO realizes both high TCP/IP performance and flexibility.

Furthermore, in order to realize RTOS in hardware, the ARTESSO introduces a new concept, virtual queue, which takes only one clock cycle for each queue operation, such as writing data into a queue, reading data from a queue, and retrieving target data from a queue. This drastically reduces the CPU energy dissipated for RTOS operations. Our approach also reduces a CPU load, which makes it possible to provide enough performance of applications with a middle-end CPU while the conventional approach needs a higher-end CPU to achieve the comparable performance.

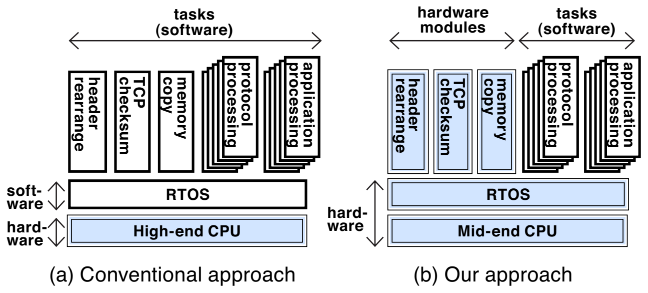


Fig. 2-2 Conventional vs. our approach

III. ARTESSO

The following subsection describes the ARTESSO architecture and an embedded hardwired RTOS which is the key of the ARTESSO.

A. *ARTESSO Architecture*

Fig.3-1 shows a configuration of ARTESSO. The ARTESSO resolves concerns which cannot be resolved by the conventional approaches. More specifically, our ARTESSO resolves the problems presented in section II in the following ways;

(1) Employ a fully hardwired RTOS which is tightly connected with a CPU core.
(2) Employ a dedicated hardware module for the "Header Rearrangement"
(3) Employ a dedicated hardware module for the "TCP checksum" between a local memory and Ethernet MAC.
(4) Use DMA transfer for the "memory copy". The BUS Arbiter and BUS Exchange shown in Figure 3-1 provide dedicated bus accesses from CPU and DMAs to Memories in parallel.

Hence the ARTESSO provides enough time for CPU to dedicate only to the protocol processing. This is because the ARTESSO releases the CPU from (1) RTOS job, (2) header rearrangement job, (3) memory copy job and (4) TCP checksum job.

Above described (3) and (4) are not new. To the best of our knowledge, (2) is new. However it can be done in a straightforward manner. The biggest contribution of the ARTESSO architecture is (1). Therefore, the rest of this section focuses on the hardwired RTOS which is integrated in the ARTESSO.

B. *ARTESSO Hardware RTOS*

The ARTESSO hardware RTOS provides more than 30 system calls with which almost of all embedded systems are satisfied. They include system calls for obtaining and releasing semaphores, setting and waiting flags, sending and receiving mail boxes, changing priorities, releasing tasks from queues, and etc. The ARTESSO RTOS can handle more than 256 tasks, events, semaphores, and mail boxes respectively. Each task can be set 16 priorities. In contrast, Silicon TRON [6] has only three tasks, semaphores and events, respectively.

Since the ARTESSO RTOS is implemented in hardware, system parameters such as the maximum number of tasks, semaphores, events and mail boxes cannot be modified after the chip fabrication, which sometimes causes a loss of flexibility. However, this is not very significant. Since, application processes including protocol processes sometimes need modification after the product design, production or shipment, however in general the RTOSs which are used in embedded systems don't need any modification after design completion.

Fig.3-2 shows a structure of the ARTESSO RTOS. The "RTOS Main Controller" controls all of RTOS processes. It executes system calls sent from the CPU, selects a task which should be dispatched, and etc. All queues needed for the RTOS operations are implemented in the "Virtual Queue" module.

Our "Virtual Queue" resolves issues in conventional hardwired RTOS and firmware base RTOS as mentioned in section II. The concept of the "Virtual Queue" is quite new. It provides functions like "enqueue", "dequeue" and "retrieving a specific element from the queue" within a single cycle. Note that the enqueue represents a function to append an element to a queue. Contrarily, the dequeuing is a function of obtaining an element from the queue. In the rest of the paper, we refer to "Virtual Queue" as VQ. The hardware cost of VQ is much less than that of the conventional hardwired queues as mentioned in section II-B. The VQ is the key in performance acceleration, energy reduction and cost reduction of the hardwired RTOS operations. The following subsection describes the VQ in detail.

C. *Architecture of Virtual Queue*

Fig.3-3 and Fig.3-4 show queuing structures of a conventional FIFO-based approach and our VQ, respectively. Eight semaphores, four priorities, and eight tasks are handled in this example. Hence 32 queues are defined and, as a result, 32 FIFOs are needed in the conventional FIFO-based approach. Each FIFO consists of a "Read Pointer", a "Write Pointer" and a memory which keeps Task-IDs. The total number of tasks is

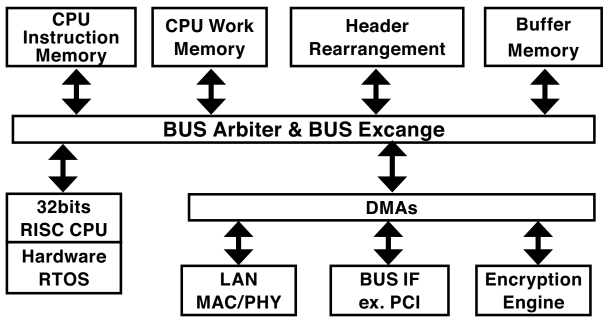


Fig. 3-1 Configuration of ARTESSO

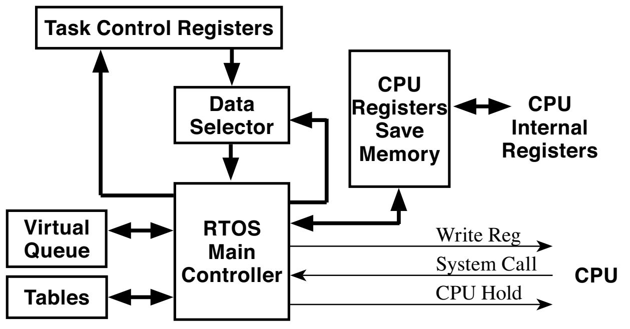


Fig. 3-2 Archtecture of Hardware RTOS

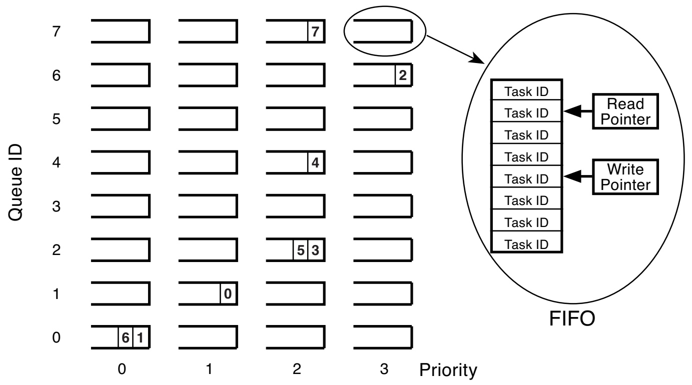


Fig. 3-3 Structure of Conventional FIFO Queue

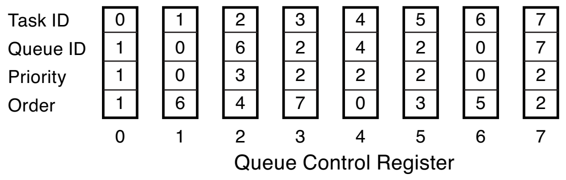
8 in this example. Therefore depth of each FIFO has to be eight for the worst-case. However needed memory space throughout all FIFOs at a time is only eight. Therefore almost of all queues are empty as shown in Fig. 3-3. This is a waste of a hardware cost in many cases. Furthermore, conventional hardware FIFO circuits do not provide the searching and retrieving function as indicated by its architecture shown in Fig. 3-3. Contrarily, our VQ only saves Task-ID, Queue-ID, priority and queuing order for each task as shown in Fig.3-4. This is much more compact than conventional one. Therefore, the area drastically decreases compared with the conventional method. If the bit width of the Queue-ID registers or the priority registers increases by one bit, the number of queues which can be managed by VQ is doubled. However the hardware volume increases by only one bit. Therefore the VQ can handle a large number of queues with a small hardware cost.

Since the number of queues implemented in actual embedded systems is larger than that shown in the example, the VQ has much bigger advantage over the conventional queue. If a system handles 32 tasks and 4,096 queues, the conventional method needs 4,096 FIFOs. However the VQ needs only 32 registers which memorize Task-ID, Queue-ID, priority and queuing order.

Fig.3-5 shows a configuration of the VQ. The VQ consists of "Queue Control Register part", "Queue Control part" and "Task Selecting part". The "Queue Control Register part" keeps queue information described in Fig.3-4. We refer to the “Queue Control Register” as simply QCR in the rest of the paper.

The "Task Selecting Part" selects a task having a requested queue ID and realizes FCFS. If a queue ID is requested by the Q_ID signal, the "Task Selecting part" returns a Task ID of a QCR which is the top element of a queue whose priority is the highest in QCRs having the requested queue ID. The "Task Selecting part" consists of “CMP & SEL” modules which are connected with each other as shown in the Fig.3-5. They realize a tournament circuit. Each “CMP & SEL” module has two predecessors. One having higher priority than another is selected. When priorities of the predecessors are the same from each other, a task enqueued earlier is selected. It can be identified from the queuing order value in the QCR. Finally, the ID of a task enqueued earliest and having the highest priority is obtained from the final stage of the “CMP & SEL” module.

According to our implementation with a 90nm commercial process technology, the result of the tournament operation can



Task ID	0	1	2	3	4	5	6	7
Queue ID	1	0	6	2	4	2	0	7
Priority	1	0	3	2	2	2	0	2
Order	1	6	4	7	0	3	5	2
	0	1	2	3	4	5	6	7

Queue Control Register

Fig. 3-4 Structure of Virtual Queue

be obtained within a few clock cycles at 150MHz when even if 256 tasks are managed in a ARTESSO RTOS.

As described in section II-C, queues in conventional firmware are configured as a chaining list structure on a memory. Therefore, operations such as modification of pointer and searching an element from the list are needed. The queue operations are divided into several sub-operations which should be manipulated sequentially. This needs several clock cycles. On the other hand, queue information of the VQ is maintained by each task, and as a result, there is no physical memory required for queue structure in the VQ. Therefore, manipulations for QCRs can be done individually and simultaneously by the Queue Control Part. This makes it possible to complete each queue operation within a single clock cycle.

The VQ performs queue operations such as enqueue, dequeue and retrieve by changing the Queuing Order Registers and Queue ID Registers. This takes only single clock cycle to complete each operation. Table 3-1 and Fig.3-6 shows an example of searching and retrieving operation. The example shows the case of serching and retteiving “Task ID 2” from the queue shown in Fig.3-6. Table 3-1 shows how the values in the QCR are modified for the operation.

IV. Evaluation

A. *Experimental Setup*

This section shows comparisons between our ARTESSO

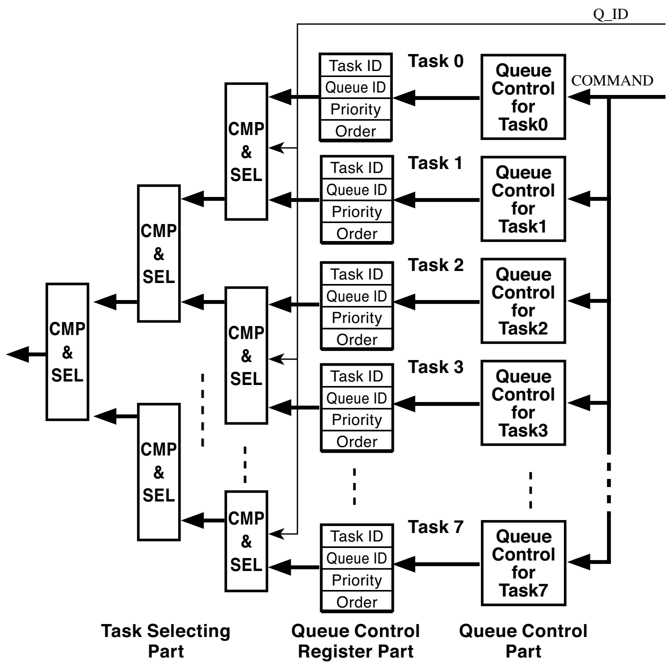


Fig. 3-5 Archtecture of Virtual Queue

Table 3-1 Task Queue Control Registers at retrieving

Task ID Register	Queuing Order Register	Queue ID Register	Priority Register
0	5 ➔ 4	1	0
1	2	0	0
2	4 ➔ Non	0 ➔ Non	1
3	7 ➔ 6	1	0
4	0	1	0
5	1	0	1
6	6 ➔ 5	0	1
7	3	0	0

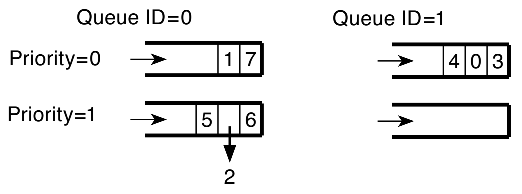


Fig. 3-6 An example of retrieving

RTOS and a conventional RTOS. NORTi is chosen as the conventional firmware RTOS. NORTi is a commercial firmware package which is compatible with ITRON [13]. It includes ITRON-compatible RTOS kernel and ITRON TCP/IP API-compatible protocol stack. NORTi is one of the best commercial firmware packages in terms of TCP/IP processing performance. In this experiment, NORTi is run on a commercial ARM926 evaluation board for measuring the number of clock cycles needed for each system call of NORTi RTOS. The performance results of ARTESSO RTOS shown in Table 4-1 are obtained using Verilog^{TM} RTL simulator.

CPU time occupation rates for TCP/IP processing are also compared. The experimental setup is the same as what described in section II-A. For estimating the CPU time occupation rates and throughputs of ARTESSO, our originally developed simulator is used. It simulates every modules presented in Fig.3-1 in a cycle-accurate manner.

Throughput of TCP/IP processed on NORTi is also measured by a commercial ARM926 evaluation board. The clock frequency is set to 50MHz. The ARTESSO is fabricated with a 150nm process gate array technology by ourselves. Throughput of the ARTESSO is measured by the gate array chip whose operational clock is set to 50MHz. The ARTESSO is also fabricated as an ASIC using a 90nm process technology.

Table 4-1 RTOS Performance

System Call	Dispatch	NORTi (ARM926)	Silicon TRON	ARTESSO RTOS
Sleep Task	Yes	628	64	10
Wakeup Task	Yes	496	82	10
Change Priority	Yes	541	103	11
Receive from Mailbox	No	224	-	7
Receive from Mailbox	Yes	591	-	11
Send to Mailbox	No	360	-	8
Send to Mailbox	Yes	541	-	11
Obtain Semaphore	No	216	-	6
Obtain Semaphore	Yes	558	83	9
Release Semaphore	No	344	-	7
Release Semaphore	Yes	536	83	11

Note : "-" means not implemented or no data

B. *Experimental Results*

Table 4-1 shows performance comparison results between the ARTESSO RTOS and existing RTOS. The value represents the number of clock cycles needed for each system call. The results demonstrate that the performance of the ARTESSO RTOS is about 30 to 60 times higher than that of NORTi. The table also shows performance of the Silicon TRON [6]. In the Silicon TRON, most sub functions of system calls are implemented in hardware. However some parts of system call processing are implemented in firmware. The results show that the ARTESSO RTOS achieves about 6 or 9 times higher performance than the Silicon TRON, in spite that the ARTESSO RTOS handles larger number of queues.

Table4-2 shows CPU time occupation results of NORTi and ARTESSO for TCP/IP processing. In ARTESSO, "Header Rearrangement" and "TCP checksum" occupy 0% CPU time since they are implemented in dedicated hardware. CPU occupation time for RTOS also decreases to 2.8%. Consequently more than 92% of CPU time is used for the application tasks in ARTESSO. This leads to an improvement of TCP/IP throughput.

The result of experiment of TCP/IP throughput of the NORTi on the 50MHz ARM926 is about 11Mbps while the ARTESSO achieves 125Mbps at the same clock rate. In other words, ARTESSO can achieve the 125Mbps throughput with a middle-end CPU while the NOTRi needs to employ a high-end CPU which typically dissipates several watts to achieve the same throughput.

Fig. 4-1 shows the power consumption results of ARTESSO and NORTi TCP/IP running on ARM946. The power consumption of ARTESSO is calculated using gate-level VerilogTM simulator and Power Compiler at 90nm process. Note that the power consumption results of ARTESSO include the power consumptions of dedicated circuits implemented for memory copy, TCP checksum and header rearrangement. The power consumption of NORTi running on ARM946 is calculated using a power and performance specifications at 90nm process in the ARM datasheet. The results include not only processor power consumption but also memory power consumption. Supposing the power consumption of the ARM946 is linear to the clock frequency, the power consumption of NORTi running on ARM946 is around 454mW at 100Mbps TCP/IP throughput. The power consumption of ARTESSO is only 65mW at the same throughput as shown in Fig. 4-1. This is about 7 times lower than that of NORTi running on ARM926.

C. *ASIC Implementation results*

The ARTESSO processor shown in Fig.3-1 have been fabricated as an ASIC using a commercial 90nm process

Table 4-2 Occupation of CPU time

Process Category	Time Occupancy (%)	
	NORTi	ARTESSO
Protocol Processing	10.5	92.7
RTOS	32.1	2.8
Header Rearrangement	15.3	0.0
Checksum	32.2	0.0
Memory Copy	9.9	4.5
Total	100.0	100.0

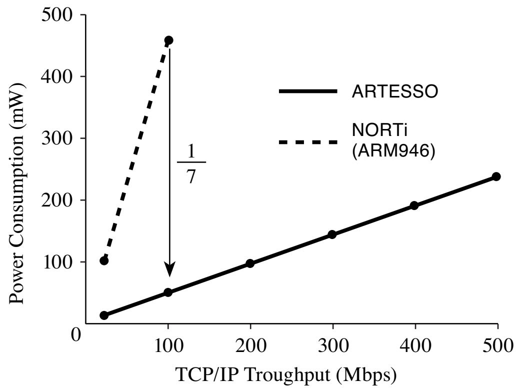


Fig. 4-1 Throughput vs. Power Consumption

technology as shown in Fig. 4-2. It includes the ARTESSO RTOS shown in Fig.3-2. The RTOS handles 32 tasks for each of which 16 priorities are defined. It provides 32 semaphores, event flags and 192 mailboxs for the tasks. Hence the total number of wait queues virtually equipped is 4,096.

Table4-3 shows area of each function of the LSI. The area for the RTOS is only 540,000µm^2. The number of gates of the logic part for the RTOS is about 170,000 gates. The total number of gates required for the VQ part in the RTOS is only 38,300. Regarding the performance of the ASIC, it achieves a 515Mbps TCP/IP throughput at the 150MHz operational clock.

V. Summary and Conclusion

Traditional approaches targeting several hundreds of Mbps for TCP/IP processing typically employ a high-end CPU having a GHz clock frequency. This consumes several watts in the processor, which degrades the processor reliability. It is also impractical for embedded systems to adopt this approach since the cost is huge. A new processor, ARTESSO, uses a much lower clock rate without losing the performance and the flexibility of the applications.

Our ARTESSO solution provides a 500Mbps throughput with 250mW for TCP/IP processing, which results in a 7 times improvement in energy efficiency compared to the existing commercial firmware RTOS. Our future work will be devoted to extend the ARTESSO to use it effectively in the other applications like robotics and the automotive applications.

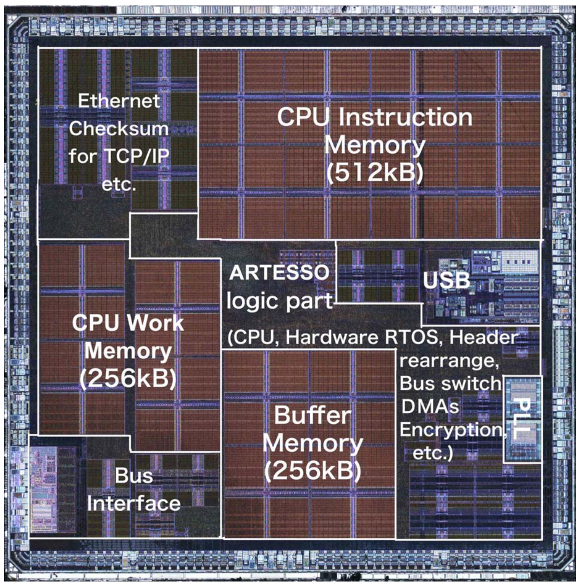

Ethernet Checksum for TCP/IP etc.
CPU Instruction Memory (512kB)
ARTESSO logic part
USB
(CPU, Hardware RTOS, Header rearrange, Bus switch DMAs Encryption etc.)
CPU Work Memory (256kB)
Buffer Memory (256kB)
PLL
Bus Interface

Fig. 4-2 ARTESSO 90nm ASIC

Table 4-3 Area of constituents

					Area (μm^2)
Total area					27,601,284
	RAM for Instruction, Data, Buffer				14,851,332
	ARTESSO				2,919,025
		RTOS			540,170
			Logic		372,964
				Virtual Queue	30,213
				Others	342,751
			RAM		167,206
		Others (CPU, Header Rearrange, DMA, Encryption, etc.)			2,378,855
	Others (Ethernet, USB, Bus Interface)				9,830,927

References

[1] Lindh, L., “Fastchart-a fast time deterministic CPU and hardware basedreal-time-kernel,” in Proc. of Euromicro Workshop on Real Time Systems, pp.36-40, Jun, 1991

[2] Lindh, L. Starner, J. Furunas, J. , “From single to multiprocessor real-time kernels in hardware,” in Proc. of the Real-Time Technology and Applications Symposium, pp. 42-43, May 1995.

[3] Adomat, J.; Furunas, J.; Lindh, L.; Starner, J. “Real-time kernel in hardware RTU: a step towards deterministic andhigh-performance real-time systems,” in proc. of the 8th Euromicro Workshop, pp.164-168, Jun 1996

[4] Nordstrom, S. Lindh, L. Johansson, L. Skoglund, T., “Application specific real-time microkernel in hardware,” in Proc. of Real Time Conference, 2005.

[5] Samuelsson, T. Åkerholm, M. Nygren, P. Johan Stärner, J. Lindh, L. “A Comparison of Multiprocessor Real-Time Operating Systems Implemented in Hardware and Software,” in proc. of Int’l Workshop on Advanced Real-Time Operating System Services (ARTOSS’03), 2003.

[6] Nakano T., Utama A., Itabashi M., Shiomi A., and Imai M., “Hardware implementation of a real-time operating system,” in Proc. of 12th TRON Project International Symposium (TRON’95), pp. 34–42, 1995

[7] Kohout, P. Ganesh, B. Jacob, B. “Hardware support for real-time operating systems,” in Proc. of the 1st International Conference on Hardware/Software Codesign and System Synthesis, pp.45-51, Oct. 2003

[8] Chandra, S. Regazzoni, F. Lajolo, M., “Hardware/software partitioning of operating systems: a behavioral synthesis approach,” in Proc. of the 16th ACM Great Lakes symposium on VLSI, pp. 324-329, 2006

[9] Parisoto, A. Souza, A., Jr. Carro, L. Pontremoli, M. Pereira, C. Suzim, A., “F-Timer: dedicated FPGA to real-time systems design support,” in Proc. of 9th Euromicro Workshop on Real-Time Systems, pp. 35-40, 1997.

[10] V.Mooney III, J. Lee, A. Daleby, K. Ingstrom, T. Klevin, and L. Lindth,, “A comparison of the RTU hardware RTOS with a hardware/software RTOS,” in Proc. of Design Automation Conference (DAC’03), 2003, pp. 683–688.

[11] Mooney, V.J., III Blough, D.M. “A hardware-software real-time operating system framework for SoCs,” IEEE Design & Test of Computers, pp. 44 – 51, 2002

[12] V. Mooney III., “Hardware/software partitioning of operating systems,” in Proc. of Design, Automation and Test in Europe Conference (DATE’03), 2003, pp. 338–339.

[13] TRON ASSOCIATION, “µITRON4.0 Specification,” 1999.