Shared Address Space I/O: A Novel I/O Approach for System-on-a-Chip Networking

Di-Shi Sun and Douglas M. Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology
Atlanta, GA 30332-0250, USA
{sundishi, dblough}@ece.gatech.edu

Abstract

For real-time system-on-a-chip (SoC) network applications, high-speed and low-latency network I/O is the key to achieve predictable execution and high performance. Existing network I/O approaches are either not directly suited to SoC applications, or too complicated and expensive. This paper introduces a novel approach, referred to as shared address space I/O, for real-time SoC network applications. This approach facilitates building of heterogeneous multiprocessor systems with application intensive processors (main processor) and I/O intensive processors (I/O processor), where network I/O processing can be offloaded to a specialized I/O processor. With the shared address space I/O approach, in such a system, communication and synchronization between main and I/O processors can be implemented with a shared address space. This approach is realized through Atalanta, a heterogeneous real-time SoC operating system we have developed. In this paper, we demonstrate that shared address space I/O can provide high-speed and low-latency network I/O for SoC network applications.

1 Introduction

1.1 Motivation for problem

The shared address space I/O approach was inspired by three factors.

First, real-time applications demand high-speed and low-latency I/O. I/O-caused interrupts are a major impediment to real-time processing. Existing approaches for I/O with DMA capability involve interruption of the main processor when an I/O operation occurs. These approaches require main processor intervention for every I/O operation and can increase I/O latency due to excessive copying of data. When considering real-time threads executing on the same processor that is interrupted for I/O, these approaches also make it difficult to achieve predictable execution or to guarantee the worst-case execution time.

Second, traditionally, the processing of network protocols is accomplished by software running on the central processor of system. Recently, the growth of network, especially Ethernet with TCP/IP protocol stack, has surpassed the growth of

microprocessor performance. This causes network I/O to become a major bottleneck for many network applications. Therefore, it is becoming common to offload network I/O, including TCP/IP protocol processing, from host processors to other processors or specialized ASICs.

Another factor is that, based on our prior research, for heterogeneous multiprocessor applications, processors can directly share the address space rather than using DMA or other means (such as shared memory based message-passing approach) to transfer data. This is a great convenience of our shared address space I/O approach. Since processors in a SoC system can use a shared address space, it allows I/O processors to do true zero-copy I/O by depositing incoming data directly into the final application buffer or copying directly from the source application buffer with absolutely no involvement from the main processor. The shared address space approach also provides a simple and standard programming model to be used for I/O processors.

1.2 Basic idea

The basic idea of shared address space I/O is to offload network I/O, including TCP/IP protocol processes, from main processors (application intensive processor) to I/O processors (interrupt intensive processor) and use shared address space to implement communication and synchronization between main processors and I/O processors [1]. Figure 1 depicts the differences between shared address space I/O approach and a uni-processor I/O approach.

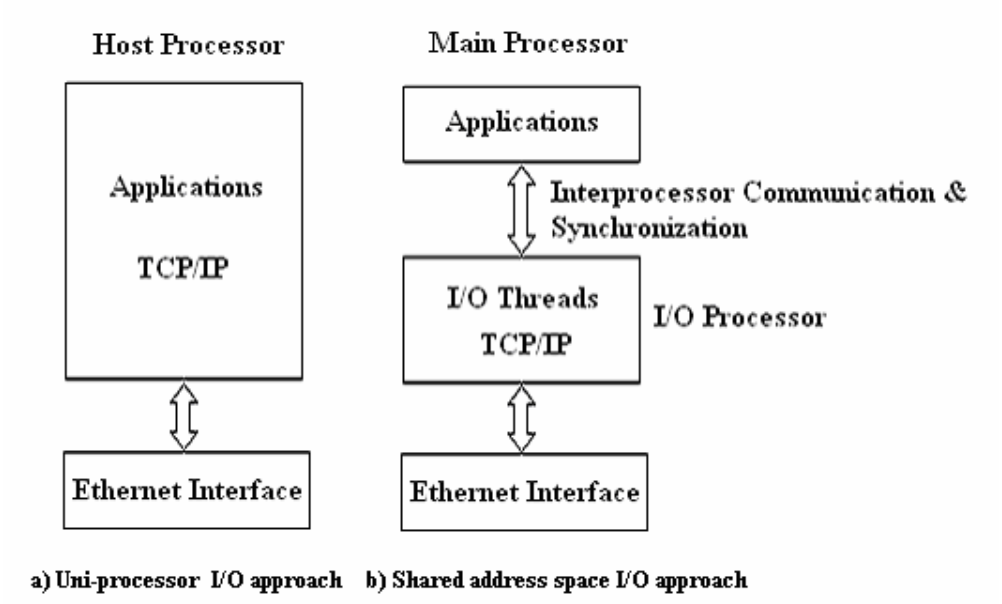

Figure 1: Uni-processor I/O approach vs. shared address space I/O approach.

Here, the network I/O not only includes servicing interrupts, reading/writing hardware ports, etc., but also includes all of the processing of TCP/IP protocols, such as connection establishment, data transform/reception, connection tear-down, error handling, etc.

The I/O processor could be a general purpose microprocessor, but a specialized interrupt intensive network processor is more reasonable. These kinds of processors provide fast I/O throughput for optimal performance. On the other hand, the main processor could be a general purpose microprocessor.

For SoC applications, it is relative easy to build a heterogeneous multiprocessor hardware platform with shared memory. The byte ordering, alignment, cache coherence and other problems that used to be important are no longer significant because of advances in microprocessor design and compiler technology. For example, more and more processors support both big-endian and little-endian byte orderings, and most commercial compilers support different alignments.

1.3 Brief mention of results

The shared address space I/O approach has been implemented in the Atalanta RTOS developed at Georgia Tech [3] and tested using the Mentor Graphics Seamless® Co-Verification Environment (Seamless CVE) [7]. We evaluated our approach with a set of applications running on a heterogeneous multiprocessor SoC hardware platform to compare against a uni-processor I/O approach. We evaluate these applications under different workload conditions (applications, network I/O, and system overhead). Our evaluation results show that, in balanced and heavy workload conditions (applications that occupy 70-80% of main processor resources, where network I/O occupies 20-30% of main processor resources), the shared address space I/O approach show better performance than a uni-processor approach.

2 Related Work

There are some researches concerning I/O bottleneck, especially network I/O bottleneck problem. The most interesting works include the I_2O^{TM} Architecture [5] from I_2O Special Interest Group (I_2O SIG^{TM}) led by Intel. Their main idea is that an I/O processor is dedicated to handling I/O requests, and to test, support unique drivers for every combination of I/O device and OS on the market. They pay more attention to increasing network server performance and reducing total-cost-of-ownership (TCO).

TCP/IP Offload Engine (TOE) [6] is another interesting topic. The idea is to offload the processing of TCP/IP protocols from the host processor to the hardware on the adapter or in the system. Their main purpose is to address new markets, such as iSCSI storage area networks (SANs) and high-performance network-attached storage (NAS) applications.

These approaches both face the communication bottleneck between main processors and I/O processors. In order to eliminate the communication bottleneck, some implementations suggest implementing inter-processor communication and synchronization protocols using specialized hardware. It obviously increases the cost and complexity of systems, and does not utilize the features of SoC architectures.

Our shared address space I/O approach adopts the main idea of the I_2O^{TM} Architecture approach and TOE approach to offload network I/O processing from main processor to I/O processor. At same time, our approach utilizes the feature of supporting shared address space of SoC architectures to implement inter-processor communication and synchronization. It avoids the side effect of offloading network I/O.

To our knowledge, there has been no research about optimizing network I/O for real-time SoC network applications similar to the shared address space I/O approach.

3. Atalanta RTOS

Atalanta RTOS is an embedded RTOS designed at Georgia Tech [3]. It provides key RTOS features including multitasking capabilities; event-driven, priority-based preemptive scheduling; precise timers; and inter-task communication and synchronization. Atalanta also supports special features such as priority inheritance and user configurability. Atalanta's small, compact, deterministic, modular and library-based architecture is important for multiprocessor SoC applications. Atalanta has been designed with support for heterogeneous processors as a primary goal since the target architecture assumes mixed systems involving RISC processors, DSP processors, and perhaps other specialized processors. Atalanta provides both "pure" message-passing approach and shared address space based message-passing (shared message) approach for inter-processor communication and synchronization. The shared message approach is a key to support shared address space I/O as described in this paper. The shared message approach allows much better use of shared memory, thereby providing greatly increased performance as compared to a "pure" message-passing approach.

Traditionally, general purpose heterogeneous RTOS rely primarily on message-passing approach for communication and synchronization between tasks on different processors. This was necessary because it is very difficult to implement closely coupled general communication architectures for broad processor families. Furthermore, the message-passing approach provides a clean separation between tasks because communication between tasks is loosely coupled and asynchronous. In traditional implementations of the message-passing approach, most messages are copied twice as they go from a sender task to a recipient task. The first copy is from the sender's preparation area to an RTOS-owned memory (usually size-fixed). The second copy is from the RTOS memory to the recipient task's reception area. When the messages are large, the copying overhead can be substantial.

For SoC applications, it is relative easy to implement a shared address space. As a result, a more efficient message-passing approach, called the shared message approach, is implemented in Atalanta RTOS. This approach simply transfers a pointer of the message from the sender task to the recipient task. The payload never actually moves, remaining in its original location in shared memory. In Atalanta, a task allocates messages from the kernel's shared memory pool. The allocated message is owned by the allocating task that uses memory as desired. When the task is ready to send the message, it submits the message address and specifies a queue. The address of the message is placed in the queue. The ownership of the message is given to the

destination task upon receipt of the message. When a task is ready to receive a message, it specifies from which queue to receive. When a message is available, the address of the message is returned and the ownership of the message is transferred to the receiving task, which can then use or modify the message contents as needed. The message can then be sent to another task or returned to the shared memory pool.

The shared message approach simplifies inter-processor communication and synchronization, and eases programming while enhancing performance for applications. It is especially fit for stream-based applications that do not need to cross-access data at the same time.

4 Description of Approach

We assume that the target applications are real-time SoC network applications and propose the concept of shared address space I/O as a mechanism to achieve high performance and low-latency network I/O for such applications.

For real-time applications, I/O-caused interrupts are a major impediment to achieve predictable execution since I/O interrupts are unpredictable and handling interrupt requests always takes priority over application tasks. DMA-capable devices free processors from involvement with the data transfer, but do not eliminate interrupts. For a uni-processor I/O approach, real-time threads are running on the same processor that processes I/O, each I/O operation will generate an interrupt that will cause real-time threads to be suspended for an unpredictable interval. It is difficult to obtain predictable execution or assure the worst-case execution time under these conditions.

Ethernet has become the most popular network protocol for local area networks. Since the rapid growth of Ethernet transmission speed, network I/O is becoming a major bottleneck in delivering high performance for almost all of network applications. The main reason for the bottleneck is that the processing of TCP/IP protocols consumes more and more processor resources. Reassembling out-of-order packets and resource-intensive memory copies put a high workload on the host processor. Although in a well-designed real-time system, processing of TCP/IP protocols should not influence predictability of execution, it obviously affects performance. In some applications using a uni-processor I/O approach that we have evaluated, the host processor has to allocate more than 20% of its resources to handle the network processing.

We propose the shared address space I/O approach to offload as much as possible network I/O from main processor to I/O processor, and to eliminate I/O interrupts and TCP/IP overhead as mentioned above. Since network I/O is completely removed from main processor, I/O-caused interrupts are eliminated. This enables real-time threads to execute in a much more predictable and guaranteed manner. In addition to achieving predictable execution, a great amount of resources that used to be consumed by the processing of TCP/IP protocols is saved and system performance is improved.

Although offloading network I/O from host processors can eliminate interference from I/O interrupts and reduce the resources occupied by network I/O, it sometimes may cause more resources to be consumed in other ways. Communication and synchronization between the processors that process network I/O and the processors

that run applications might become a new bottleneck of system performance. Our simulation results present in Section 5 show that this overhead is minimal on a number of representative I/O micro-bench programs.

Recently, the design of processors has become more and more specialized. Some processors are designed as application intensive processors, and others are designed as interrupt intensive processors. This makes system designs more complicated. In order to get best application and I/O performance, we would use application intensive processors as main processors to deal with applications, and interrupt intensive processors as I/O processors to deal with I/O operation. Processors communicate with each other by some kind of inter-processor communication and synchronization mechanisms. The problem is, in these kinds of heterogeneous multiprocessor architectures, the design of efficient inter-processor communication between processors is an extremely difficult job. Normally, message-passing approach is used with additional resource occupation. Some vendors design specialized hardware to implement communication protocols to improve communication performance. It obviously increases the complexity and cost of systems. It also does not fit well with embedded SoC designs.

In the shared address space I/O approach, we propose to structure systems with one or more application processors and one or more I/O processors as shared memory heterogeneous multiprocessor systems. One I/O processor is responsible for one or more DMA-capable I/O devices in the system. All processors in a system, including main processors and I/O processors, access the shared memory using a shared address space. We assume that hardware and firmware provide cache-coherence for shared memory. In [4], we describe a simple but effective approach to achieving cache coherence in heterogeneous multiprocessor systems. Since each processor in a system supports cache coherence, and the number of processors is limited, the shared memory architecture does not detract from system performance. With this kind of hardware model (see Figure 2), we can expect best application and I/O performance.

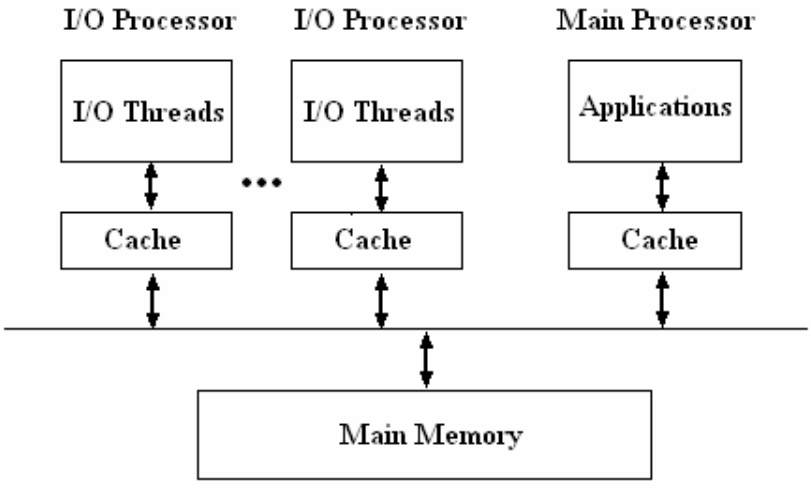

Figure 2: Hardware model for the shared address space I/O approach.

In order to solve the communication problem, we view I/O processors and main processors as a shared-memory multiprocessor. With this view of a system, I/O processors can directly execute code that shares data with application threads executing on main processors. This mechanism is implemented using a shared address space approach. From user's point of view, the programs running on main processors and I/O processors can access the same data structures, and we do not need additional data copying between a local buffer and a shared system buffer as required in shared memory message-passing approaches. This allows I/O processors to communicate directly with application memory without main processor intervention. Hence, data coming from an I/O device can be deposited directly by the I/O processors into the appropriate location in application memory without any main processor control. This approach eases communication bottlenecks that drive up latencies of I/O operations.

There is currently no commercial embedded RTOS that supports shared address space approach for heterogeneous SoC applications. It is partly because of the variety of embedded processors and embedded applications. We have implemented an embedded RTOS, known as Atalanta [3], which supports shared address space approach for heterogeneous multiprocessor architectures. Our development of Atalanta together with the simulation results in the next section prove that a shared address space approach is feasible, and provides better performance than a shared memory message-passing approach for SoC applications.

The shared address space I/O approach offers the potential for applications to interact directly with I/O interfaces without any involvement from main processor and to provide predictable and guaranteed execution of real-time threads. In addition to improving I/O performance, the shared address space I/O approach also provide tremendous flexibility in I/O processing. For example, it is an easy way for applications to push processing to I/O processors by specifying their own custom threads that are tailored to their particular I/O requirements.

5 Simulation Environment

The shared address space I/O approach has been implemented in the Atalanta RTOS and tested using the Mentor Graphics Seamless® Co-Verification Environment.

5.1 Seamless CVE

Seamless® Co-Verification Environment (Seamless CVE) [7] is an interactive hardware and software co-simulation tool provided by Mentor Graphics. It can create a virtual prototype of SoC system. By executing software on simulated hardware, it allows us to fully verify the hardware/software design without a physical prototype.

5.2 Hardware model

The hardware model tested under Seamless CVE uses a standard 10 Mbps Ethernet interface, a 100 MHz PowerPC MPC755 main processor, and a 50 MHz ARM

ARM920T for network I/O, and is shown in Figure 3. This architecture was tested using the Atalanta RTOS with a shared address space approach and no special hardware for communication between processors. This architecture and programming model are compared against a standard uni-processor architecture with DMA capability in which the main processor is responsible for network I/O processing.

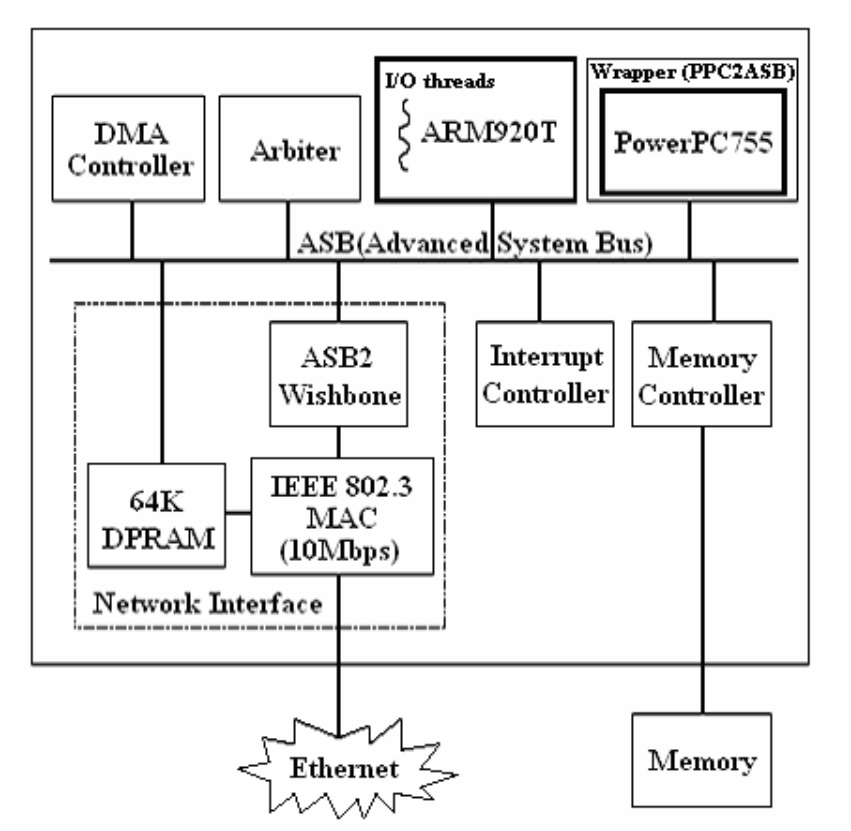

Figure 3: Evaluated hardware model.

5.3 Software Model

The software model used for evaluation includes an application task running on the main processor, and an I/O thread executing a TCP/IP stack on the I/O processor. The communication and synchronization between processors is implemented by shared address space. The application task runs a set of stream-based decoding applications, and the I/O thread is in charge of network I/O and directly deposits incoming data into application buffer, then informs the application task using system services provided by the Atalanta RTOS.

6 Simulation Results

We evaluate the shared address space I/O approach using a set of stream-based decoding applications running on the software/hardware models described in Section

4.3 and 4.4 to compare against a uni-processor I/O approach. These applications contain various decoding operations on streams of data, including CRC, MD5, DES and Huffman decoding. We ran every application for fixed amount of time (1000ms) and measured the total number of packets that main processor processed. For the uni-processor approach, we also measured the percentage of processor cycles used by I/O processing. In order to study the effect of workload, we tested these applications under different workload conditions (application, network I/O, and system overhead).

For light application workload (CRC and MD5 in Figure 4), the main processor has enough resource to process both I/O and application. The bottleneck is the I/O capability of the processors. We used the MPC755 in the uni-processor approach. It is faster than ARM920T that we used as the I/O processor for I/O thread in the shared address space I/O approach. In this situation, the uni-processor approach has better performance than the shared address space I/O approach can provide. For high application workloads (DES and HUFFMAN in Figure 4, 5 and 6), I/O only occupies less than 10% of the main processor resources. Since shared address space I/O approach has to do inter-processor communication and synchronization, we might expect shared address space I/O approach to perform worse than the uni-processor approach. In fact, the two approaches get almost identical results in this situation.

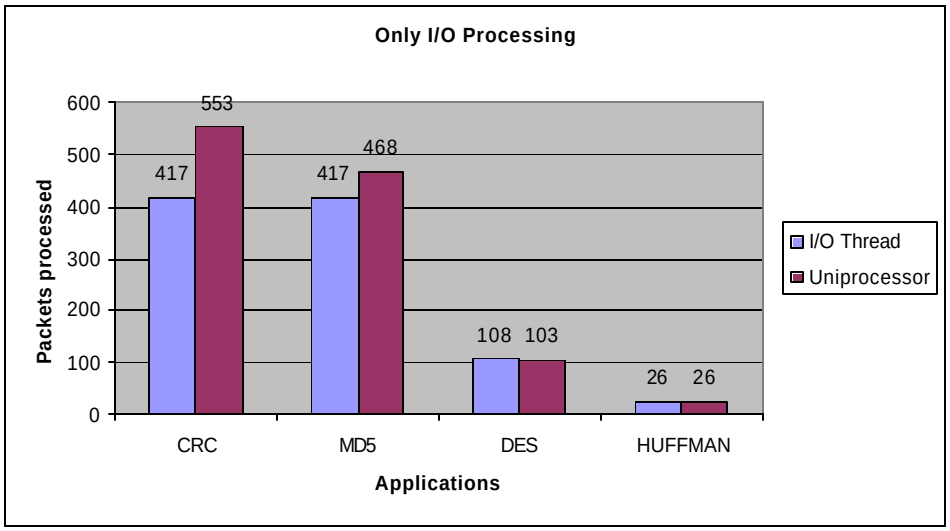

Figure 4: Simple applications for shared address space I/O approach vs. uni-processor approach.

It is difficult to find a real application consuming a certain percentage of processor resources. For example, it is difficult, if not impossible to find an application consuming exact 70% of processor resources compared to I/O operation occupying 30% of processor resources. So, we studied two ways to achieve applications consuming greater percentages of processor resources. One method is doubling the application processing. Every packet is processed twice instead of only once. Another method is adding an additional application with high priority running on the main processor. This additional application does nothing but increases a counter. Since the counter variable is cached, these additions only consume main processor resources and do not cause any side effect. In every million second, this application does 1000 times addition operations then suspends itself until the next time interrupt wakes it up.

It consumes about 25% of main processor resources. In these two ways, we can achieve tight control of workload.

For heavy and balanced workloads, where applications occupy 70-80% of main processor resources and I/O occupies 20-30% of the resources (CRC and MD5 in Figure 5 and Figure 6), the shared address space I/O approach show better performance than the uni-processor approach.

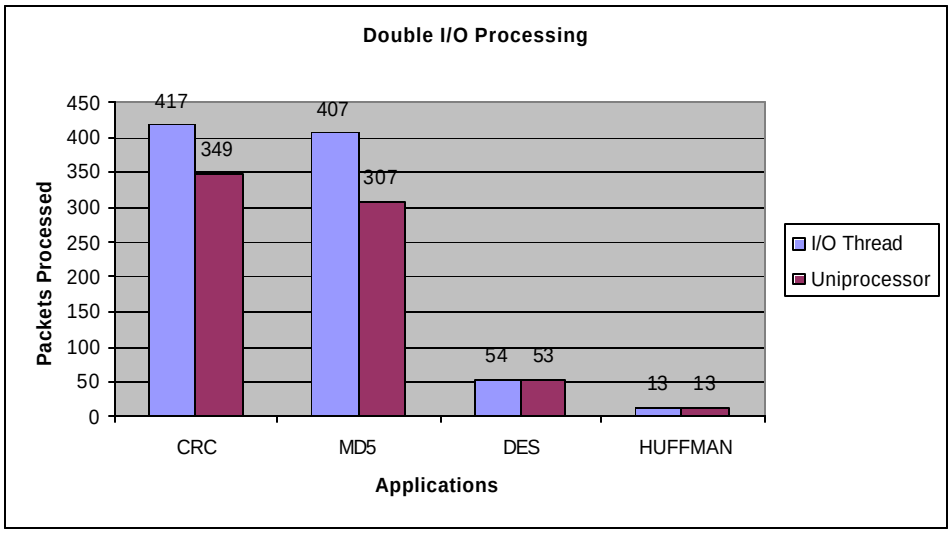

Figure 5: Double processing applications for shared address space I/O approach vs. uni-processor approach.

For CRC decoding, in the double packet processing test, the shared address space I/O approach was 19.5% better than the uni-processor approach. In the test with an additional application, the shared address space I/O approach performed 8.8% better than the uni-processor approach. For MD5 decoding, in the double packet processing test, the shared address space I/O approach were 32.6% better than the uni-processor approach. In the test with an additional application, the shared address space I/O approach performed 31.2% better than the uni-processor approach. The improved performance of the shared address space I/O approach in this situation is due to the fact that the I/O processor is no longer a bottleneck. For the uni-processor approach, and CRC and MD5 applications, I/O consumes about 27% of the main processor resources. The cost of increasing inter-processor communication and synchronization is less than the savings of offloading the I/O for the shared address space I/O approach.

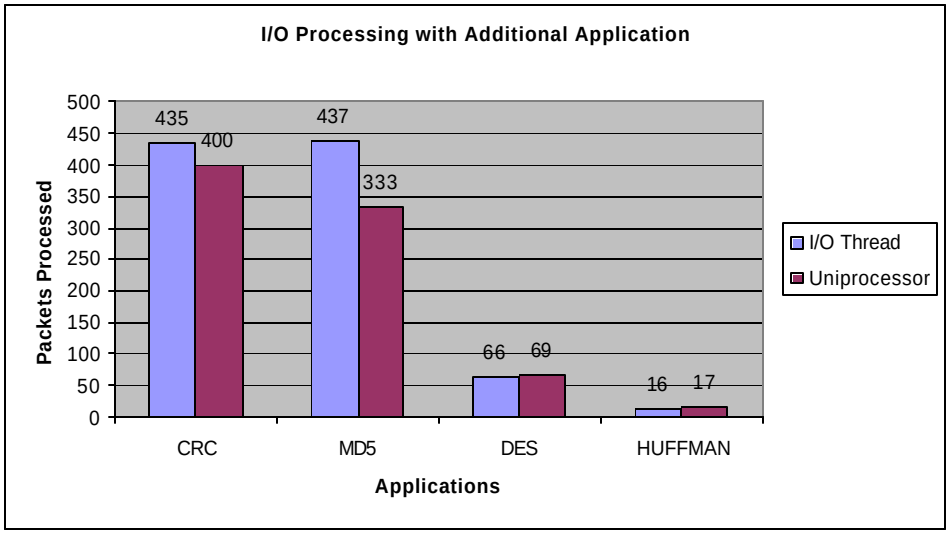

Figure 6: With additional application for shared address space I/O vs. uni-processor approach.

For I/O processing with an additional application, one more task runs on the main processor. The system overhead is increased more significantly for the shared address space I/O approach than for the uni-processor I/O approach. So, the difference between shared address space I/O approach and uni-processor approach for double I/O processing is more obvious than for I/O processing with an additional application.

We also should mention that the performance of shared address space I/O is worse than it could be because cache coherence has not been implemented.

From our evaluation results, we can say that, for balanced and heavy load conditions (applications occupying 70-80% of main processor resources, and network I/O occupying 20-30% of main processor resources.), the shared address space I/O approach's performance is better than the uni-processor I/O approach.

7 Conclusions

The shared address space I/O approach described in this paper, adopts the main idea of the I_2O^{TM} Architecture approach and TOE approach to offload network I/O processing from main processor to I/O processor. It utilizes the support of shared address space of SoC architectures, without any additional hardware to implement inter-processor communication and synchronization, and provides high-speed and low-latency network I/O for SoC network applications. Our simulation results show that, for balanced and heavy load conditions, shared address space I/O perform better than the traditional uni-processor I/O approaches.

References

[1] V. J. Mooney and D. M. Blough, "A Hardware-Software Real-Time Operating System Framework for SOCs", IEEE Design and Test of Computers, pp. 44-51, November-December 2002

[2] V. J. Mooney, "Hardware/Software Partitioning of Operating Systems", Proceedings of the Design Automation and Test in Europe Conference (DATE'03), pp. 338-339, March 2003.

[3] D. Sun, D. M. Blough, and V. J. Mooney, "Atalanta: A New Multiprocessor RTOS Kernel for System-on-a-Chip Applications", Technical Report GIT-CC-02-19, Georgia Institute of Technology, 2002,

[4] T. Suh, D. M. Blough, and H. S. Lee, "Supporting Cache Coherence in Heterogeneous Multiprocessor Systems", Technical Report GIT-CERCS-03-15, Georgia Institute of Technology, September 2003,

[5] I_2O SIG, "I_2O Tutorial: I_2O Architecture Specification, rev. 1.5", PC Developer Conference Session #241T.

[6] Gigabit Ethernet Alliance, "Introduction to TCP/IP Offload Engine (TOE) Version 1.0", 10GEA White Papers, April 2002

[7] Mentor Graphics, Seamless Hardware/Software Co-Verification, http://www.mentor.com/seamless/.