Network Processing on an SPE Core in Cell Broadband Engine^{TM}

Yuji Kawamura,
Takeshi Yamazaki,
Tatsuya Ishiwata and Kazuyoshi Horie
Microprocessor Development Dept
Sony Computer Entertainment Inc.
Tokyo, Japan
yuji01_kawamura@hq.scei.sony.co.jp,
tyamaki@rd.scei.sony.co.jp,
{tatsuya_ishiwata,kazuyoshi_horie}@hq.scei.sony.co.jp

Hiroshi Kyusojin
Technology Development Group
Sony Corporation
Tokyo, Japan
kyu@sm.sony.co.jp

Abstract

Cell Broadband Engine^{TM} is a multi-core system on a chip and is composed of a general-purpose Power Processing Element (PPE) and eight Synergistic Processing Elements (SPEs). Its high computational performance is achieved mainly through the SPE's processing power.

New high-speed NICs such as 10-Gbps Ethernet require significant amounts of processing power. Even the full processing power of PPE is insufficient to attain the maximum bandwidth on 10-Gbps Ethernet, when running Linux on Cell Broadband Engine^{TM}.

In order to avoid the bottlenecks of PPE processing, we implemented a NIC driver and a protocol stack on an SPE. We selected a small protocol stack that is designed for embedded systems and made size reductions to put both a protocol stack and a NIC driver onto a single SPE. Due to the size limitation of the SPE's local storage (256-KB).

As a result, the protocol processing on an SPE is almost at wire speed for UDP and about 8.5 Gbps for TCP with lightly tuned code, and it requires no assistance from the PPE while in the data transfer phase.

Our work shows that the use of the SPE instead of the PPE for network processing can help resolve network performance problems that can arise from handling a high-speed NIC, including the costs of protocol processing and memory copies.

The results indicate that our approach can lead to a sufficient level of transfer rate performance.

1. Introduction

The Cell Broadband Engine^{TM} is a multi-core system on a chip developed by Sony, IBM and Toshiba [1] [12]. The Cell Broadband Engine^{TM} is composed of a Power Processing Element which is a processor core with a PowerPC instruction set architecture and eight Synergistic Processing Elements (SPEs). The SPEs are SIMD instruction set processors with 256-KB of local storage (LS). An SPE differs from conventional processors with cache hierarchies in that it relies on a DMA engine to move code and data between the LS and the main memory, which in turn enables explicit data transfers between memory hierarchies. With a clock speed of 3.2 GHz, the Cell Broadband Engine^{TM} has a theoretical peak computational performance of 230.4 GFlop/s (single-precision floating point). The SPEs provide a significant portion of the computing power in a Cell Broadband Engine^{TM} [10]. Therefore, the key to fully utilizing the performance of the Cell Broadband Engine^{TM} is to fully exploit the SPEs.

Another important characteristic of the SPE is its predictability, which stems from its simple pipeline structure and a flexible DMA engine. The SPE's predictability enables the development of finely tuned application codes that precisely and allows control hardware mechanisms, application programmers to achieve highly sustainable performances with real applications [9] [5]. In addition, the simplified features enable efficient implementations with high performance-to-area ratios [11] [12].

It has become increasingly difficult to gain improvements in the performance of semiconductor chips through scaling theory. Therefore, it is increasingly important to exploit a computation engine with a good performance-to-area ratio like the SPE.

The goal of this work is to use SPEs to support system infrastructure with the elements essential to computer systems, such as high-speed networks and storages.

The following characteristics make SPEs potentially superior to existing solutions for high-speed device support.

- The allocation of dedicated resources for individual devices is effective in achieving short and stable response times [7]. Because SPEs occupy a smaller chip area than general-purpose processor cores of equivalent performance, when one whole processor core is used as a dedicated processor, the allocation of one SPE has less impact than that of general-purpose core.
- The use of DMA enables the scheduling of a large number of memory access requests that are asynchronous to the processor core. SPEs can access the system memory in a more flexible and tightly scheduled way than general cache-based systems during computation.
- The communication between the SPE and the device is possible with a low overhead because the devices can access the high-speed LS directly.
- An SPE differs from a special-purpose device such as a network processor in that it is a general-purpose resource that can be used for a variety of computational tasks. Thus SPEs can be used for different purposes depending on the status of system usage.

However, it is necessary to split execution codes and data for the 256-KB LS, and codes and data must be swapped as required via explicit DMA requests. This necessitates a different programming technique from that used in conventional processors, and increases the level of technical difficulty in particular when porting existing large-scale applications.

This paper provides an overview of the implementation of network protocol stack processing in an SPE and an evaluation of its performance.

The standard protocol stacks incorporated in operating systems such as Linux have large memory footprints, and require extensive modification when they are ported to an SPE. We prioritized design time and selected for a base a protocol stack with a small footprint designed for an embedded device.

The protocol stack NetX [2], provided by Express Logic, and the microkernel ThreadX which is required for its operation were ported to SPE. Porting was enabled by placing a small number of data structures (connection tables, packet buffers, etc.) in the main memory and using DMA access.

Generally, protocol stacks use scalar code and it is difficult to use SIMD instructions to tune performance. In our experiment, speed was increased through the adjustment of data alignment and optimization of branching. A transfer performance of 8.5 Gbps using TCP-IP was achieved when a single SPE operating at 3.2 GHz was allocated exclusively to support a 10 Gbps optical Ethernet NIC.

As indicated by Foong,et al. [8], standard TCP-IP network protocol stacks require a processing power of 1 Hz/bps on a Pentium 4. At a time when 1 Gbps is standard for Ethernet NIC and the use of 10-Gbps Ethernet is becoming more widespread, network protocol stack processing consumes the largest CPU cycle time among the services that support a single device.

In recent years, HPC cluster systems that use multi-core processors have become common. On these systems the load on the CPU assigned to protocol stack processing may increase, and problems may arise from load unbalancing. To prevent this, a single core at each node is frequently assigned to dedicated network processing [6]. In the Cell Broadband Engine^{TM}, the use of one of the eight SPEs for this purpose makes it possible to employ 10-Gbps Ethernet. This represents a considerable advantage over a system using conventional high-performance processor cores.

Section 2 below considers related research in order to clarify the positioning of the project discussed here. Section 3 discusses the characteristics of implementation of protocol stack processing using SPEs and the methods of optimization, while Section 4 presents the results of performance evaluations. Section 5 presents the conclusions of the project, and Section 6 considers problems and future directions.

2. Related Works

An SPE has an unique instruction set architecture in which all the computational instructions are SIMD instructions. This gives the SPEs a high level of floating-point computation performance per chip area, and they are employed in various areas of numerical computation applications [5] [14]. However, system services like the kernel services of operating systems and IO device processing are generally scalar tasks with complicated data structures, and there is no publication handling this theme of poring these types of processing to an SPE.

We ported a network protocol stack on an SPE. One SPE was separated from processor scheduling on the Cell Linux kernel and assigned for processing a network protocol stack. As a result, this SPE looks like a network processor [7].

As network interfaces began to handle high packet rates and interrupts reduced system performance. And they become a problem [13]. To avoid performance reduction from frequent interrupts, Aron,et al. [4] proposed the software-timer-based network processing, which can eliminate higher frequent interrupts from NIC with timer-based polling. This approach's main objectives are to limit net-

Table 1. RC-101 10-Gigabit Ethernet Adapter

Features	Items	
Hardware	Vendor	Sony Corporation. Engineering samples available
	MAC	original
Technical Features	IEEE standard	IEEE802.3ae 10GBASE-SR
	Wiring	Multi-mode fiber (300m)
	Host bus	PCIe (x1, 4, 8)
	Transceiver	original
	Power consumption	7.5W
Software Features	Checksum offload	IP, UDP, TCP
	Jumbo frame	9k
	Interrupt moderation	original

work processing cycles and to eliminate context switching. Our approach also eliminates interrupts from NIC, because all the interactions from NIC are handled by polling on dedicated SPE.

Assigning dedicated resources for network processing has multiple benefits [15] [19]. Wu,et al. [18] shows that process scheduling by a Linux kernel may largely affect the performance of a protocol stack. Our approach eliminates any interference between protocol stack processing and other kernel tasks.

There was another challenge of controlling system-wide performance by restricting CPU cycles for network processing [16]. But our approach can completely offload network processing from the PPE on which the operating system is running.

From a security perspective, separating the protocol stack from a monolithic kernel is preferable [17]. This suggests that our schemes which utilizes separate resources for protocol stacks can potentially improve system security.

3. Implementing the Network Protocol Stack in an SPE

This section discusses the system constructed in this project: hardware and software configurations, the functions of the implemented system. It also addresses the processes and performance improvements required to implement a protocol stack on an SPE.

3.1. System Configuration

Figure 1 shows the configuration of the hardware system constructed for this work. We used the Sony RC-101 10-Gigabit Ethernet Adapter. This hardware is not yet commercially available but it is available for evaluation. Table 1 shows the main specifications of the RC-101 10-Gigabit Ethernet Adapter. The Cell Broadband Engine^{TM} is connected to RC-101 using a PCIe x8 interface through a prototype high-performance FlexIO-PCIe bridge.

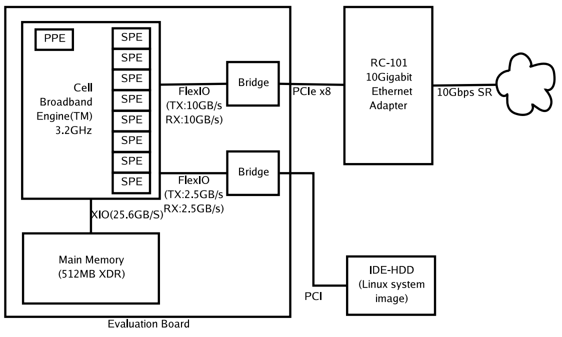
PPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE
SPE
Cell Broadband Engine(TM) 3.2GHz
FlexIO (TX:10GB/s RX:10GB/s)
Bridge
PCIe x8
RC-101 10Gigabit Ethernet Adapter
10Gbps SR
FlexIO (TX:2.5GB/s RX:2.5GB/s)
Bridge
XIO(25.6GB/S)
Main Memory (512MB XDR)
PCI
IDE-HDD (Linux system image)
Evaluation Board

Figure 1. Hardware System

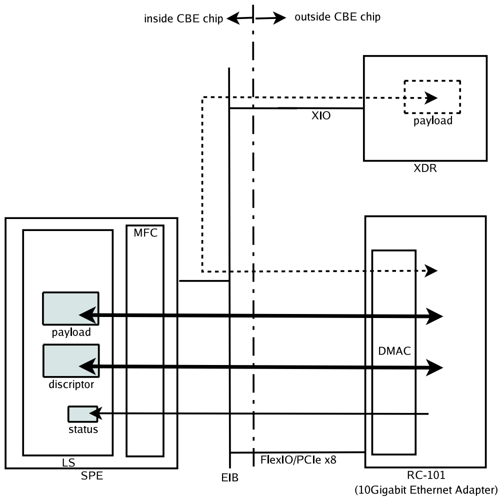

Figure 2. Connection between RC-101 and SPE

3.2. Basic Strategies for Supporting TCP/IP on SPE

3.2.1 Protocol stack

The standard protocol stacks incorporated in operating systems such as Linux have large memory footprints that exceed the LS size of 256-KB, and extensive modifications are therefore necessary when they are ported to an SPE. We prioritized design time, and used a protocol stack for embedded applications, which has a small footprint, as a base.

The protocol stack NetX, manufactured by Express Logic, and the microkernel ThreadX that is required for its operation were ported to an SPE.

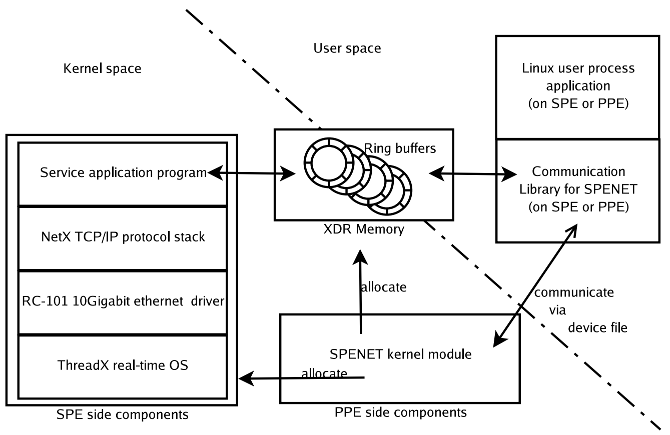
Kernel space
User space
Linux user process application (on SPE or PPE)
Service application program
Ring buffers
Communication Library for SPENET (on SPE or PPE)
NetX TCP/IP protocol stack
XDR Memory
RC-101 10Gigabit ethernet driver
allocate
communicate via device file
SPENET kernel module
ThreadX real-time OS
allocate
SPE side components
PPE side components

Figure 3. Software System Overview

3.2.2 Pinning

The linux distribution for Cell Broadband Engine^{TM} treats the SPEs as virtualized resources, which means that user applications can generate more SPE threads than available SPEs, and context switching of the SPE thread may occur while user applications are running. However, as Figure 2 shows, we stored descriptors in the LS and attempted to operate the NIC directly from the SPE, and it was therefore necessary to pin an SPE. One SPE was pinned and used exclusively for network processing.

We considered using multiple SPEs to fix the code footprint problem, but did not apply this method, in order to free as many number of SPEs.

3.3. Software Module

Figure 3 shows the configuration of the software system. The Linux 2.6.20 kernel with a patch for our evaluation board is used. The kernel is running on the PPE.

One SPE was pinned for the SPE-side components which are shown on the left. In the SPE, the ThreadX real-time OS, RC-101 NIC driver, NetX TCP/IP protocol stack and a service application are running. The service application communicates with a user space application and executes requests from the user space application by calling NetX API. This SPE runs in the Linux kernel space.

The PPE-side component shown in the center (bottom) is in charge of initializing, managing and terminating the SPE-side components. Because there is no standard kernel API call method from SPE to PPE in Linux, the SPE-side components cannot directly call kernel API existing in the PPE core. The PPE-side component is implemented as a kernel module and calls various kernel API as requested by the SPE side components, including the allocation/deallocation of memory, obtaining an IO space address and probing the network adapter. The memory region shown in the center box in Figure 3 is a kernel space memory region that is allocated and mapped to the user space by the PPE-side component. Communication with the user space is conducted via ring buffers that are placed in this memory region.

Table 2. implemented functions

Function	Support	Enabled
ARP	yes	yes
RARP	yes	no
ICMP	yes	no
IGMP	yes	no
IPv4	yes	yes
IPv4 checksum	yes	no
IPv6	no	
TCP	yes	yes
TCP checksum	yes	no
TCP selective ack	yes (receive only)	yes
TCP slow start algorithm	no	
TCP fast retransmit	no	
TCP window scaling	no	
UDP	yes	yes
UDP checksum	yes	no
API	NetX specific socket like API	

3.4. Functions of Protocol Stack

The NetX is used for supporting TCP and UDP. The TCP protocol stack implemented in Linux has a variety of advanced functions. However, NetX is designed for embedded systems where code size is important. Thus, advanced functions such as window scaling are omitted from NetX. Table 2 shows the functions of NetX and whether they were activated in this project. Of the functions supported by NetX, only the essential functions were selected, in order to prevent the code size from becoming too large.

3.5. Solving the Code Footprint Problem

We implemented the real-time OS and the protocol stack onto an SPE, as well as a simple test program in the same SPE. This exhausted local storage, and no margin remained for additional functions or packet memory.

Because the registers and operands in an SPE are 128-bit, more instructions are required to access a structure without alignment restrictions than when a standard 32-bit RISC processor is employed. Therefore, code sizes can easily become large in programs like protocol stacks that access structures frequently. In order to control the code size, we added alignment restrictions to the main structures in the NetX. Because we employed gcc as the compiler, we used the directive __attribute__((aligned(N))), which was developed for adding alignment restrictions. As a result, while the data size increased, the code size was reduced. The size of the .text segment was reduced from 145-KB to 113-KB after the alignment, representing a reduction of approximately 21%. However, the .bss segment increased approx-

Table 3. local store usage

Segment name	.text	packet_pool	.bss	stacks	.rodata	.data	tcp	thread	udp	.debug_ranges	other	total
Before	144592	49920	39328	17408	4656	1264	480	352	336	128	3680	262144
After	113168	57600	45728	17408	4720	1264	1856	1312	1072	128	17888	262144
Ratio	0.78	1.15	1.16	1.00	1.01	1.00	3.87	3.73	3.19	1.00	4.86	1.00

imately 18% in size following the alignment, from 39-KB to 46-KB. Table 3 shows a comparison of the segment sizes in the LS before and after alignment. "packet_pool" represents the packet pool,with the section that maintains the packet payload at an MTU of 1500 bytes and the structure that controls the packets. The term, "stacks" refers to the stack area, which also holds a thread context, while "tcp" is the TCP control block, and "udp" is the UDP control block. In the table, "total" represents the size of the LS. Overall, the addition of alignment restrictions to the structures resulted in a size reduction of approximately 5%.

Additionally, the alignment restrictions resulted in a 27% improvement of loopback performance in the TCP and 22% in the UDP.

3.6. Insertion of Branch Prediction to Increase Speed

Because SPEs do not have a branch prediction hardware, static hint branch instructions are used to indicate the fetch direction. However, even when branch instructions were counted by the number of "if" statements in the C source code, there were more than 1200. It was impractical to input the gcc builtin_expect code manually. Instead, we used the gcc profile-guided optimization (PGO) to enable branch prediction. Communication was via loopbacks from the test program, and statistics on the direction of branching were gathered and fed back during compiling. Using PGO, we were able to improve TCP loopback performance by approximately 22%.

3.7. Making the System Non-preemptive to Improve Performance

The protocol stack that we employed in this project, NetX, is a commercial stack, and is designed for use upon the embedded real-time OS ThreadX, which is also manufactured by Express Logic. We ran NetX by porting ThreadX to the SPE. And preventing ThreadX from being triggered by any type of interrupt processing other than the timer processing of the thread by SPE decrementer interrupts. Interrupts of the SPE core can potentially be operated by NICs from the MFC Memory Mapped I/O (MMIO) register, but the SPE processing speed is fast enough, and at low load, tasks can be rescheduled in round-robin cycles of approximately 400 nano seconds. Given this high-speed response, polling was employed. In addition, the exclusive use of an SPE made it unnecessary to run in coordination with unknown non-network processes (unlike CPU cores running a normal OS) and therefore the maximum delay could be controlled.

ThreadX supports preemption, but we did not enable preemptive processing. As indicated above, the system is in control of the entire task processing in the SPE. Therefore non-preemptive task processing can be conducted without delays, and exclusion processing such as mutexes can be eliminated, resulting in higher speeds. Eliminating mutex processing enabled us to increase the bandwidth of loopback processing by approximately 13%.

3.8. Use of the SPE to Drive NIC

Because SPEs have a DMAC in every core, they are also capable of efficient NIC register accesses. On a conventinal processor core (like the PPE), the pipeline stalls for an extremely long period in units of instruction cycles during MMIO register access. In contrast, the DMAC in an SPE can be explicitly controlled by software in an SPE, enabling instructions to be issued without waiting for time-consuming MMIO accesses to complete, and programs can therefore continue to be executed during MMIO accesses. For example, when sending data, the NIC is notified of the update in the transmission descriptor, and this notification is normally implemented by writing the NIC MMIO register. With SPEs, a DMA transfer instruction can be issued without waiting for this write to complete, and prevents from stalling.

RC-101 uses 32-byte descriptors placed in the ring buffer for sending and receiving. We placed the descriptors in the LS to control RC-101. When the send descriptor is updated, the RC-101 register is tapped and the descriptor read from RC-101 is kicked. Packet payloads are placed either in the LS or on main memory. When placed in the LS, the data flow is as shown by the solid line in Figure 2. The data flow when the packet payloads were placed on main memory is shown by the dotted line in the same figure. In the latter case, the payload has to be retrieved from the main memory, doubling the load on the memory system when compared to LS. However, because it is difficult to place multiple 9KB payloads in the LS in order to increase the MTU, it is necessary to place payloads on main memory and reduce the data resident in the LS for high bandwidth.

3.9. Increasing the Number of Supported Sessions

When all functions were handled in the LS without using the main memory, the maximum number of connected sessions that could be supported simultaneously was 16. We increased the number of sessions in order to demonstrate that the SPE can be used for more realistic applications. It is difficult to increase the number of sessions because of the packet pool capacity and the sizes of the TCP control block (TCB) and the UDP control block (UCB). The TCB and UCB occupy approximately 1-KB and 512-B of the LS, respectively. Therefore if the TCP supports 128 sessions, for example, half the LS capacity is used. We were able to support 128 simultaneous TCP sessions by using the techniques described below.

- Software cache for TCB and UCB

  We placed the TCB and UCB on main memory. The protocol stack retrieves these into a small area in the LS when required. If the TCB and UCB were modified they are restored to the main memory. These operations are handled by a software in the SPE.

  The TCB and UCB include the port number field which are frequently searched. We extracted such data into the LS and made resident for speed-up.

- Placing queues in the main memory

  Because the packet pool capacity is essential to guaranteeing the TCP retransmit and keeping the data receiving. The data placed in the TCP queues was saved to the main memory and the packet pool in the LS was used as a cache area.

3.10. API for User Space Application

We implemented the following application interfaces to enable the use of the protocol stack from Linux user space application processes.

- Proprietary socket interface API (SOCK API)

  The SOCK API provides the NetX API without requiring changes in the user space application programs. The service programs that provide these API are located in the service application program section of the SPE-side components shown in Figure 3. These service programs use resources guaranteed by the PPE kernel module, and employ a memory area shared between the user space and the kernel space to communicate with user processes. In addition, by calling NetX API, the service programs use the NetX functions to realize TCP or UDP communication.

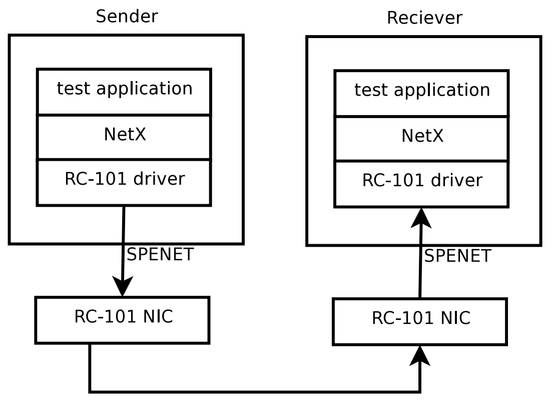

Figure 4. Test Environment for End to End Communications

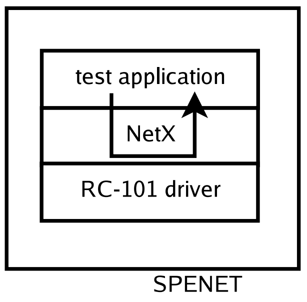

Figure 5. Test Environment for Loopback Communications

- Remote Direct Memory Access protocol API (RDMA API)

  The RDMA API follows the instructions issued by user applications, and copies data using TCP to communicate between remote nodes. Unique protocols were used in TCP.

- SDP on RDMA API (SDP)

  To enable the use of Infiniband SDP functions [3], we used the library provided in the SDP through emulation of the HCA driver in kernel space. This is to support applications using standard sockets.

The SOCK API and the RDMA API cannot coexist at the same time.

4. Performance Evaluation

This section discusses the measurement environment used in performance measurements, and the measurement results.

4.1. Measurement Environment

The measurement environment of end-to-end communications is shown in Figure 4. During measurements, RC-101 units were connected directly with an optical cable with no switches in between. During end-to-end communications, the sender side only transmitted data and the receiver side received data.

The measurement environment of loopback communication is shown in Figure 5. In loopback, the packets were reproduced in the protocol stack and returned by means of memory copies in NetX. During loopback, send and receive transmissions were executed on a single SPE using two threads.

Performance data is measured by running the test program shown in Figures 4 and 5 on the SPE where the SPE-side components are running. The NetX socket API is called directly from the test program. 10 Gigabits of data is transferred, and the throughput which includes the IP header and MAC header length is measured. We measured UDP throughput by varying UDP payload size. Also, we measured TCP throughput by varying TCP Max Segment Size (MSS) and TCP window size.

4.2. Loopback Communications Performance

Loopback performance measures only the performance of the protocol stack. However, because the packets are copied and returned when NetX is used, results are largely affected by the additional memory copies.

4.2.1 UDP loopback

Figure 6 shows the results of the UDP loopback test. The bandwidth reaches 7.5 Gbps when MTU is 1472 bytes.

4.2.2 TCP loopback

Figure 7 shows the results of TCP loopback communications. The bandwidth reaches 4.6 Gbps when MSS is 1460 and window size is approximately 26 KB (MSS $\times$ 18).

4.3. End-to-End Communications Performance

Two types of data structures are used for communication between the device driver and the protocol stack. The packet descriptor carries control information and is placed on LS to enable low overhead polling by SPE. The other data structure is a packet buffer. The large data structures

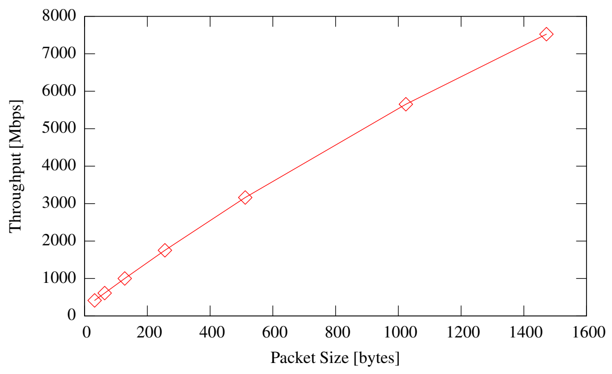

Figure 6. UDP Loopback Performance

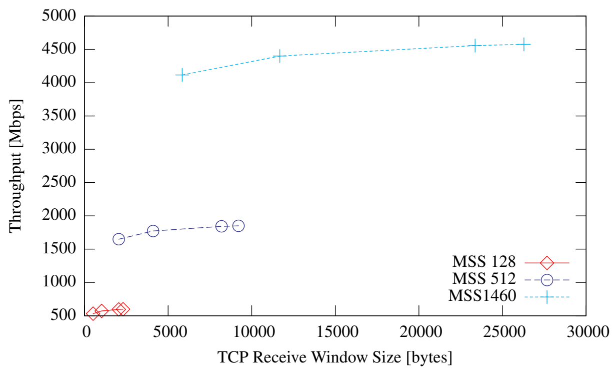

Figure 7. TCP Loopback Performance

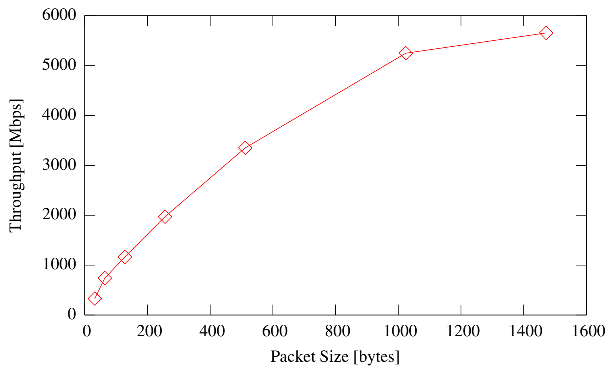

Figure 8. UDP End-to-End Performance (Using LS Only)

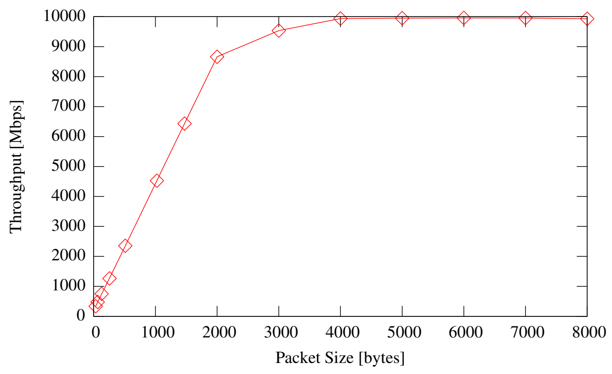

Figure 9. UDP End-to-End Performance (Using the Software Cache)

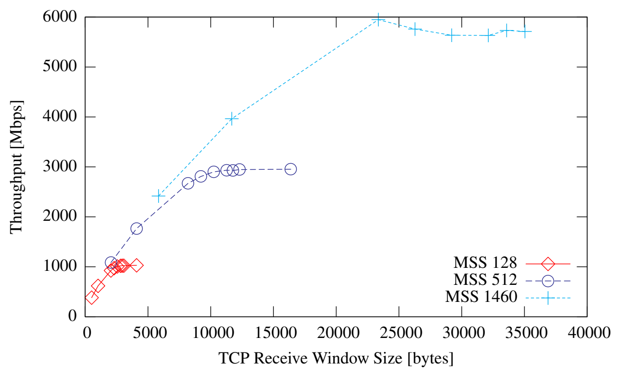

Figure 10. TCP End-To-End Performance (Using LS Only)

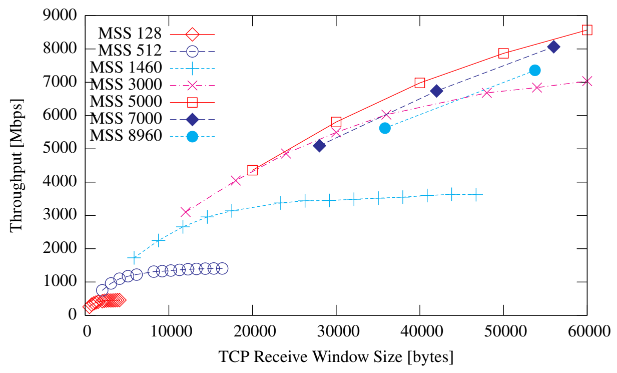

Figure 11. TCP End-To-End Performance (Using The Software Cache)

that are accessed by the protocol stack are UCB and TCB. We compared two ways of arranging these data structures.

- LS configuration

  This configuration puts the entire packet buffer, the UCB and the TCB on LS. This is good for data access latency and main memory loads. And thus attains the better packet handling rate. However, because of the LS size limitation, we limited the MTU size to 1500 bytes. With this configuration, eight descriptors are assigned for sending and four descriptors are assigned for receiving.

- Main memory configuration

  This configuration uses LS as a cache memory. The data structures that can be cached is UCB, TCB, and packet queues. The total size of these data structures can be larger than the LS size. Thus the MTU size and packet pool size can be configured to be large enough. 64 descriptors for RC-101 NIC are assigned for sending and 32 descriptors are for receiving.

4.3.1 UDP

This section discusses the results for UDP end-to-end communications. Figures 8 and 9 show the results for UDP end-to-end communications. The figures represent the LS configuration and the main memory configuration, respectively. As shown in Figure 9, wire speeds were almost attained for UDP. The result of Figure 8 is not up to wire speed, because of the limitation on MTU size and the number of descriptors. In the main memory configuration, there is only one session, so there are no cache misses for UCB. But incoming packets with higher rates than the processing power are put into the UDP receive queue. This operation causes a main memory access and lowers the throughput.

4.3.2 TCP

This section discusses the results for TCP end-to-end communications. Figures 10 and 11 show the results for TCP end-to-end communications. The figures represents the LS configuration and the main memory configuration, respectively. The throughput of Figure 10 shows much smoother results than Figure 11, because Figure 10 involves no main memory accesses and has static packet processing times. In Figure 10, the maximum throughput of 5.9 Gbps is reached when is MSS 1460 bytes and the TCP window size is 23-KB (MSS $\times$ 16).

In Figure 11, a higher maximum throughput of 8.5 Gbps is reached at a 5000 bytes MSS and a 60-KB TCP window size (MSS $\times$ 12). This is because the main memory configuration can utilize a larger MTU size and larger descriptors than the LS configuration. But in the main memory configuration, both the sender and receiver must access the main memory at each packet processing to maintain transmit queue and receive queue. The overhead of these main memory accesses resulted in limiting throughput of the main memory configuration.

4.3.3 Maximum Packet Rate

During transfers from LS, packet processing performance is approximately 870K [packet/sec]. During transfers from the main memory, packet processing performance is approximately 530K [packet/sec]. These are measured in UDP end-to-end communications.

5. Conclusion

The Cell Broadband Engine^{TM} features eight processors with a unique design, called SPEs, on a single chip. We implemented a network protocol stack on one of these SPEs. On this single SPE, we operated the protocol stack NetX, developed for embedded applications, and the microkernel ThreadX required for its operation. Both were manufactured by Express Logic. In addition there are communication interfaces enabling the protocol-stack-driven SPE to be accessed from device drivers and other processors.

The most significant characteristic of SPE architecture is its omission of a cache hierarchy; the SPE has an LS that is controlled via DMA. This necessitates a different form of programming than that used for a conventional processor. In implementing the protocol stacks, we employed software caches for processing a number of data structures.

We optimized the implementation of the NIC device drivers and the protocol stack in the fallowing various ways.

- Communication between the network hardware and the processor was realized with a low overhead by means of direct access to the LS from the NIC.

- Data alignment was adjusted to prevent performance from declining and the code size from increasing. (The SPE uses 128-bit load/store instructions, rather than scalar data load/store instructions.)

- Profile-based static branching prediction was used to support code scheduling and branch hint instruction scheduling by the compiler.

As a result, we achieved a TCP performance of 8.5 Gbps for 3-KB packet sizes using a single SPE operating at 3.2 GHz. This result indicates a competitive network protocol processing performance, considering that we employed a processor designed for a variety of computational applications rather than a dedicated network processor, and demonstrates the potential of the application of SPEs in this field.

6. Future Directions

With the results achieved in this work as a starting point, the following directions can be indicated for the future development of network processing using SPEs.

- Further optimization for SPE use

  We used 32-bit scalar code, and therefore did not make effective use of the SPE's 128-bit SIMD data path. We believe that there is potential for optimization in numerous areas through the significant modification of code structures. For example conditional branches can be replaced by select bit instructions or shuffle byte operations can be used for packet header arrangement. Further optimization can be expected to result in improved performance.

- Seamless integration in Linux kernels

  Because NetX is designed for embedded applications, it is not compatible with standard Linux protocol stacks, and therefore cannot replace Linux protocol stacks. Adding the required functions and integrating the protocol stack in a Linux kernel would make it accessible from a Linux application in the same way as a standard protocol stack. We believe that a Linux kernel patch shared with a full TCP-IP offload using a network processor [7] could be used to enable this integration.

- Implementation of various types of network packet processing

  In this work, we focused on basic protocol stack processing, but further network packet processing can be considered for the future. Encryption-related processing such as IP-SEC and DTCP over IP, packet filtering, and other applications can be named as the next targets. SPEs have shown excellent performance in encryption and decryption processing, as typified by AES [5]. SIMD processing can also be expected to increase the speed of pattern matching processing, and may be applicable for accelerating packet filtering.

As the next step in our work, we are developing a test application that embeds the protocol stack discussed here. To avoid the time required for rewriting a large-scale software for the SPE, and to generate usable results in a short period, we are attempting to apply the concept of web services that employ combinations of highly independent services. Specifically, this combines a group of major element technologies and the Cell Broadband Engine^{TM} to realize an independent service package that can be accessed via a network.

In addition to the network interface driven by the protocol stack discussed here, we intend to port and install a storage device, a file system and a database engine onto the SPE. By combining a single Cell Broadband Engine^{TM} with commodity parts, we will implement a media data service package that includes media streaming on a bandwidth of several hundred MB/s, and meta-data processing.

7. Acknowledgements

The authors would like to thank Bill Lamie and David Lamie of Express Logic for their cooperation and for disclosing the NetX and ThreadX source code.

The authors also thank to Hisashi Tomita, Youichi Mizutani, Masahiro Kajimoto, Yuichi Machida, Kazuyoshi Yamada, Tsuyoshi Kano, Mitsuki Hinosugi of Sony Corporation for giving us the resources for evaluation with the RC-101 10-Gigabit Ethernet Adapter and for helping in performance tuning.

And the authors thank to Mariko Ozawa, Ken Kurihara, Takuhito Tsujimoto and Daniel Toshio Harrel for reviewing this paper.

Finally, the author would like to thank children of Awafune Football Club for their exuberant energy.

References

[1] Cell Broadband Engine^{TM} Home Page at Sony Computer Entertainment Inc. `http://cell.scei.co.jp/`.

[2] Express Logic Home Page . `http://www.rtos.com/`.

[3] Openfabrics alliance home page. `http://www.openfabrics.org/`.

[4] M. Aron and P. Druschel. Soft timers: efficient microsecond software timer support for network processing. *ACM Transactions on Computer Systems (TOCS)*, 18(3):197–228, 2000.

[5] T. Chen, R. Raghavan, J. Dale, and E. Iwata. Cell Broadband Engine Architecture and its first implementation- A performance view. *IBM Journal of Research and Development*, 51(5):559–572, 2007.

[6] P. Crowley, M. Fluczynski, J. Baer, and B. Bershad. Characterizing processor architectures for programmable network interfaces. *Proceedings of the 14th international conference on Supercomputing*, pages 54–65, 2000.

[7] W. Feng, P. Balaji, C. Baron, L. Bhuyan, and D. Panda. Performance Characterization of a 10-Gigabit Ethernet TOE. *Proceedings of the IEEE International Symposium on High-Performance Interconnects (HotI)*, 2005.

[8] A. Foong, T. Huff, H. Hum, J. Patwardhan, and G. Regnier. TCP performance re-visited. *Performance Analysis of Systems and Software, 2003. ISPASS. 2003 IEEE International Symposium on*, pages 70–79, 2003.

[9] M. Gschwind. Chip multiprocessing and the cell broadband engine. *Proceedings of the 3rd conference on Computing frontiers*, pages 1–8, 2006.

[10] M. Gschwind, P. Hofstee, B. Flachs, M. Hopkins, Y. Watanabe, and T. Yamazaki. A novel simd architecture for the cell heterogeneous chip-multiprocessor. *Hot Chips 17*, August 2005.

[11] H. Hofstee. Power efficient processor architecture and the cell processor. *High-Performance Computer Architecture, 2005. HPCA-11. 11th International Symposium on*, pages 258–262, 2005.

[12] J. Kahle et al. Introduction to the Cell multiprocessor. *IBM Journal of Research and Development*, 49(4):589–604, 2005.

[13] I. Kim, J. Moon, and H. Yeom. Timer-Based Interrupt Mitigation for High Performance Packet Processing. *Proceedings of 5th International Conference on High-Performance Computing in the Asia-Pacific Region*, 2001.

[14] J. Kurzak and J. Dongarra. Implementation of the Mixed-Precision High Performance LINPACK Benchmark on the CELL Processor Technical Report UT-CS-06-580.

[15] G. Regnier, D. Minturn, G. McAlpine, V. Saletore, and A. Foong. ETA: Experience with an Intel Xeon Processor as a Packet Processing Engine. 2004.

[16] K. Ryu, J. Hollingsworth, and P. Keleher. Efficient network and I/O throttling for fine-grain cycle stealing. *Proceedings of Supercomputing01*, 2001.

[17] A. Sinha, S. Sarat, and J. Shapiro. Network subsystems reloaded: a high-performance, defensible network subsystem. *Proceedings of the USENIX Annual Technical Conference 2004 on USENIX Annual Technical Conference table of contents*, pages 19–19, 2004.

[18] W. Wu and M. Crawford. Potential performance bottleneck in linux tcp. *Int. J. Commun. Syst.*, 20(11):1263–1283, 2007.

[19] B. Wun and P. Crowley. Network I/O Acceleration in Heterogeneous Multicore Processors. *Proceedings of the 14th IEEE Symposium on High-Performance Interconnects*, pages 9–14, 2006.