RC23204 (W0405-040) May 7, 2004
Other

IBM Research Report

A Multiprocessor System-on-a-Chip Design Methodology for Networking Applications

Valentina Salapura, Christos J. Georgiou, Indira Nair
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10598

Research Division
Almaden - Austin - Beijing - Haifa - India - T. J. Watson - Tokyo - Zurich

A Multiprocessor System-on-a-Chip Design Methodology for Networking Applications

Valentina Salapura, Christos J. Georgiou, Indira Nair
IBM T.J. Watson Research Center
P.O. Box 218
Yorktown Heights, NY 10583
{salapura, georgiou, indira}@us.ibm.com

Abstract

This paper presents a System-on-a-Chip design methodology that uses a microprocessor subsystem as a building block for the development of chips for networking applications. The microprocessor subsystem is a self-contained macro that functions as an accelerator for computation-intensive pieces of the application code, and complements the standard components of the SoC. It consists of processor cores, memory banks, and well-defined interfaces that are interconnected via a high-performance switch. The number of processors and memory banks are parameters that can vary depending on the application to be implemented on the chip. Applications such as protocol conversion, TCP/IP off-load engine, or firewalls can be implemented with processor counts ranging from 8 to 128.

1. Introduction

The market shift toward storage area networks (SAN) and network attached storage (NAS) systems, as well as the massive expansion of the Internet, have placed new demands on server and storage designs. General purpose CPUs either cannot meet the computational requirements of the network protocols, or are too expensive in terms of unit cost, space and power. This has led to the offloading of many of the networking and protocol processing functions from host processors into host-bus-adapters (HDAs) or network interface controllers (NICs). Initially, most HDAs and NICs were implemented in ASICs using hardwired logic. But as the need to implement complex network protocols arose, such as TCP/IP or iSCSI, programmable solutions have become attractive because of a number of advantages they offer: they can accommodate different and evolving protocols; they are easily upgradeable via program changes; they offer a faster time to market. An example of such a programmable solution that is implemented in the form of a scalable, multiprocessor architecture can be found in [1].

Another trend that has emerged in the design of high-speed networking is the system-on-chip (SoC) approach [2]. SoC designs have provided integrated solutions for communication, multimedia, and consumer electronics applications. To meet performance and functionality requirements, current SoC designs typically include a general-purpose processor, one or more DSPs (Digital Signal Processors), multiple fixed function silicon (FFS) accelerators, and embedded memory. For example, a CDMA cell phone might have 800,000 to 3 million gates dedicated to FFS accelerators, and up to four DSPs performing the remainder of the processing.

The building of current state-of-the-art SoCs typically requires the designer to assemble a complex system from basic components, e.g., processors, memory arrays and controllers, and to interconnect them reliably under usually strong time-to-market pressure. More specifically, the SoC designer must: a) select the basic system components; b) model bus-contention between the different devices and select appropriate bus structures; c) integrate all hardware components; and d) finally, integrate all system components using custom software to provide the required services, such as Fiber Channel, Ethernet, TCP/IP, iSCSI, or other standardized protocols.

However, there are inherent problems with current SoC design methodology: it is labor-intensive and error-prone; it requires highly-skilled designers familiar with a particular application domain; and it demands high cost for bus modeling and/or contention on a common system bus. Other approaches to SoC design where multiple sub-systems are integrated on a card or board exhibit drawbacks due to component count that drives system cost, increased failure susceptibility, and the cost of high-interconnect multi-layer boards.

These limitations of current SoC methodology have led us to pursue a more flexible and efficient approach. The approach utilizes a multi-processor system comprising memory and local interconnect, which is

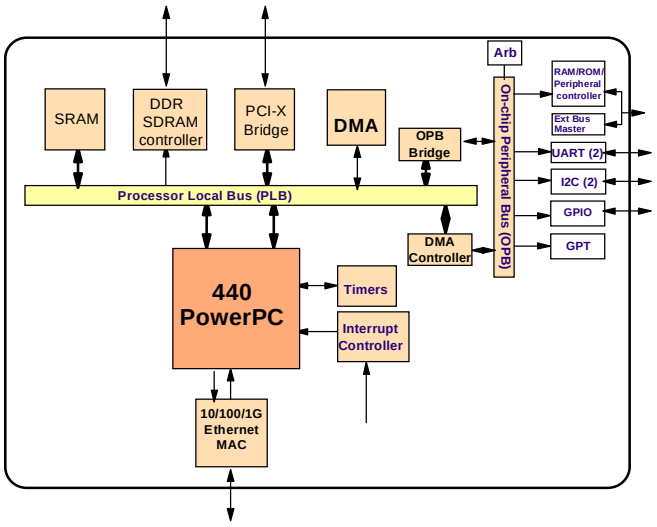

Figure 1: Single processor-based SOC

embedded as a macro attached to a common bus (e.g., CoreConnect PLB [3]). The macro is customized for a specific functionality by including application software running on it. The SoC designer, then, needs to determine performance parameters required for the specific application to be implemented (e.g., number of multiprocessor processor cores, amount of memory, line speed), and possibly add some application-specific hardware interface macros. The basic SoC structure and modeling will already be in place, thus significantly simplifying the SoC development process.

Some possible applications that can be implemented using this approach are IPSec VPN tunneling engine, TCP/IP Offload Engine, network processing for iSCSI, or encryption engine (i.e., for compression/decompression).

The paper is organized as follows: In section 2, we describe the overall SoC architecture as well as the architecture of the multiprocessor macro. In section 3, we present a case study on the performance analysis of a TCP/IP off-load engine, and in section 4, we show the results of the case study.

2. System-on-a-Chip Architecture

A typical System-on-Chip design is illustrated in Figure 1. It comprises a processing element, a local processor bus (PLB), on-chip peripheral bus (OPB), and a number of components, such as SRAM, DDR controller, PCI-X bridge, DMA controller, OPB bridge, etc. This particular implementation uses the IBM embedded PowerPC[1] 440 processor core [4] and the CoreConnect local bus [3], but similar configurations can be found that use other embedded processor cores, such as ARM [5], MIPS [6], etc.

The approach based on a single embedded processor can provide a cost-effective, integrated solution to some applications but may lack the computational power required by more demanding applications. The computational capabilities of the SoC were enhanced, in a number of networking applications, through the addition of special-purpose processor cores attached to the common bus, as shown in Figure 2, operating in parallel with the processor core (i.e., PowerPC 440). These additional special-purpose processor cores are usually small in silicon area, as many of the features found in typical general-purpose processors (e.g., memory management unit to support virtual addressing, etc.) were excluded. Examples of this approach are the IBM PowerNP [7], and the NEC TCP/IP offload engine [8]. Although these systems are programmable and, consequently, more flexible compared to hardwired accelerators, they suffer from several drawbacks: a) they induce additional traffic on the SoC bus, as the bus must now support both instruction and data streams to the processor accelerators possibly causing bandwidth contention and limiting system performance; b) the SoC bus is often not optimized for multiprocessor performance but for compatibility with standardized components and connection protocols in a SoC system; c) the processor accelerators often implement only a very limited instruction set and use assembler language, thus making the development and maintenance of applications running on the processor accelerators very difficult and costly.

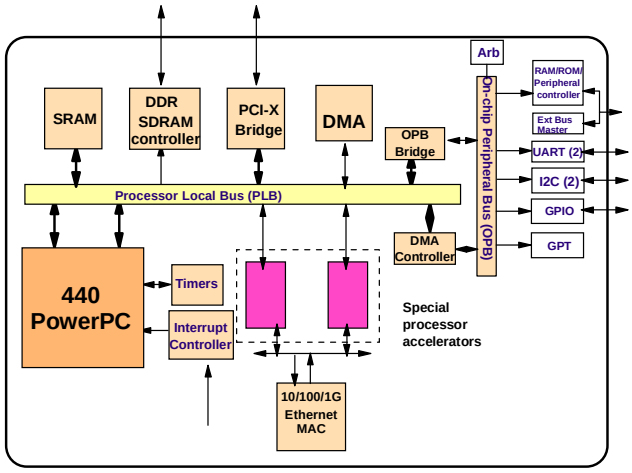

Figure 2: SoC employing multiple processor accelerators

[1] IBM and PowerPC are registered trademarks of International Business Machines Corporation

2.1 Architecture description

The above limitations of current architectures led us to the development of a programmable scalable macro that is easily reconfigurable in terms of processing power and memory capacity. In our methodology, we have replaced the collection of special processors shown in Figure 2 with a single macro comprising a self-contained processor-based subsystem (as illustrated in Figure 3). This subsystem is integrated as a component in the SoC and is connected to the main processor bus via a bridge. The processor based subsystem comprises one or multiple processor clusters, one or more local memory banks for storing data and/or instructions, and a local interconnect means implemented via a crossbar switch. The self-contained processor-based subsystem macro is illustrated in Figure 4.

Our subsystem comprises many simple processor cores with a reduced general purpose instruction set derived from the PowerPC architecture. The processor cores have a single-issue architecture with a four-stage deep pipeline. Each core has its own register file, ALU, and instruction sequencer [1]. Four cores share a 16KB local SRAM, while a cluster of eight cores share a 32 KB instruction cache. The I-cache bandwidth is sufficient to prevent instruction starvation, as one cache line (16 instructions) can be delivered to the processors every cycle. The size of the instruction cache is sufficient for network applications, as most processor working sets fit in the I-cache, as shown in [9]. Our implementation of the Fibre Channel protocol shows that less than 4 Kbytes of I-cache per processor are sufficient to get instruction hit rates of more than 98%, because of the small footprint of the code [1]. Additional banks of globally accessible SRAM can be provided accessed via the crossbar switch.

The exact number of processor clusters in the subsystem needed to support sufficient computation power, e.g. one, two, or even 16 processor clusters (comprising 128 cores), depends on the application requirements, as it will be shown in Section 4. For example, implementing endpoint functionality for Fibre Channel network protocol requires less computational power than the more complex TCP/IP termination.

Another feature of our processor-based subsystem is the use of embedded memory for storing the application program, current control information, and data used by the application. Sufficient amounts of memory to provide smooth operation under normal operating conditions can be placed in the subsystem without excessively increasing its size. A further advantage of embedded memory, as compared to conventional off-chip memory, is that it offers short and predictable access times, which can be accurately accounted for in the time budget estimates for the processing of packets.

All elements in the subsystem are interconnected via a switch that is, in turn, connected to the SoC processor

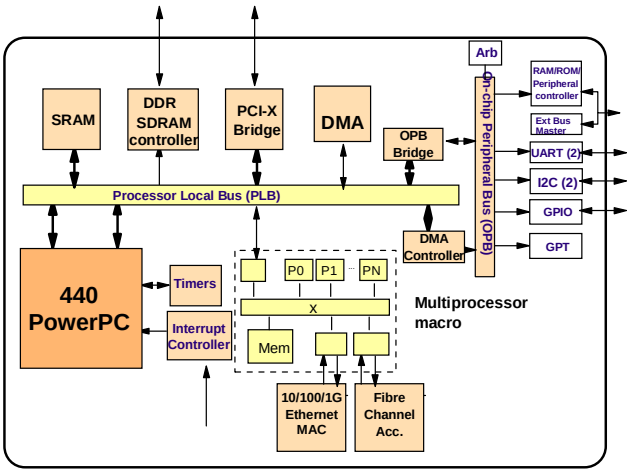

Figure 3: SoC employing self-contained multiprocessor macro

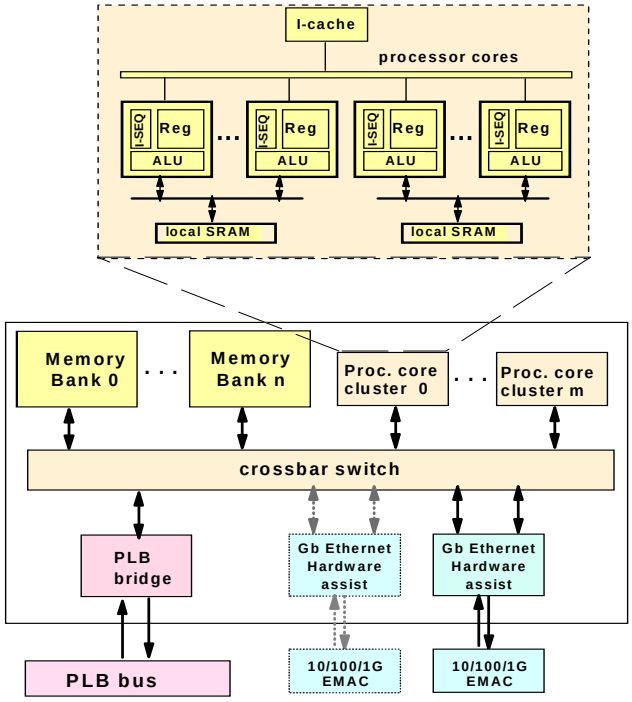

Figure 4: Multiprocessor subsystem macro

bus by means of a bridge macro. The bridge can be adapted to accommodate different speeds, bus widths, signals, and signaling protocols. The implementation of a standard interface between the subsystem macro and the embedded processor local bus (i.e., PLB) offers the advantage of allowing the integration of the subsystem macro into the SoC component library. Additional advantages resulting from the separation of the subsystem and the processor buses are as following:

- The only traffic between the subsystem and the SoC system is the data flow traffic (data receive and send), thus minimizing bandwidth contention
- The subsystem interconnect fabric (i.e., switch) can be designed to provide an optimized high-performance solution to the multiprocessor, without the need to accommodate the standard component interfaces and connection protocols of the overall SoC

2.2 Boot-up and initial program load

The multiprocessor subsystem can be viewed as an accelerator to the SoC processor (e.g., a PowerPC 440) that is used for offloading computation-intensive tasks. The initial program load into the multiprocessor subsystem is done by the SoC processor. The multiprocessor subsystem has only volatile storage and thus it contains no boot-up code. After power on, and after the SoC processor is booted-up, the SoC processor initializes the loading of a small bootloader into the SRAM memory of the multiprocessor subsystem. Upon its completion, a processor core in the multiprocessor subsystem begins loading the particular application into the subsystem, and assigning system tasks to each processor.

In our implementation, we use a pipelined approach for multiprocessor packet processing, where the packet processing operations are partitioned into stages that are assigned to separate processors. We chose this approach because of better utilization of the hardware resources, such as, for example, I-caches. Examples of network operations that can be assigned to separate pipeline stages are header handling, packet validation, generation of an acknowledgment response, packet reordering and message assembly, and end-to-end control.

The scheduling of protocol tasks to processors is done statically during initialization, i.e., each processor executes the same set of operations on various packets. Likewise, to avoid overhead associated with dynamic memory management, such as garbage collection, static memory management is used. All memory structures used are initialized during system bring-up. These include memory areas for storing data packets, control and status information of existing network connections, program code, and work queues. Further details on scheduling tasks and on the various memory structures used in our architecture can be found in [1].

2.3 Operation

As an illustration of the data flow in the multiprocessor subsystem, we describe a possible implementation of TCP/IP off-load engine in which a TCP/IP segment is transmitted over the Ethernet link. This involves the following steps:

- The SoC processor sets a request for data processing and sends the request and the pointer to the data in the external DDR memory to the multiprocessor subsystem via the bridge. In our implementation, an interrupt signal rises, but this can be implemented by writing data to some dedicated register or pre-specified memory location instead
- The multiprocessor subsystem recognizes the request and activates the DMA engine to transfer data from the external memory to its local memory
- Data are transferred to the memory in the processor-based subsystem, and end of data is signaled
- The multiprocessor subsystem implements the specific protocol tasks required, such as partitioning of the data into a series of IP packets, generation of IP packet headers, generation of Ethernet packets, etc., and moves the packets to the Ethernet MAC macro. If there is a need to retransmit packets, as defined by the protocol, this takes place without interference from the SoC processor
- When all data are transmitted, the SoC processor is notified about the task completion. This can be implemented by sending an interrupt to the PowerPC440, or writing to some predefined location which is regularly polled by the PowerPC440

2.4 Area estimates

The architecture of our multi-processor subsystem is cellular allowing the design to be custom scaled. In our design, the number of processor cores and embedded memory blocks can be easily adapted to the application requirements without making significant design changes. We have found that in the following networking applications, the required computational capacity of the multiprocessor subsystem operating at line speeds of 10 Gb/s can vary as follows:

- Protocol conversion: 14 processors [1] (i.e., two 8-core processor clusters). A chip that includes 64 Kbytes of I-cache, 64 Kbytes of data SRAM, a PowerPC440 and the other macros shown in Figure 3, would require approximately 35 mm^2 in 0.13μm ASIC technology.
- TCP/IP offload engine: 32 processors, i.e., four processor clusters (see Section 4). Assuming 128 Kbytes of I-cache and 128 Kbytes of SRAM, this would occupy 50 mm^2 in the technology above.
- Integrated firewall: 128 processors (estimate), i.e., 16 processor core clusters. Assuming 512 Kbytes of I-cache and 512 Kbytes of SRAM, the resulting chip would be about 150 mm^2.

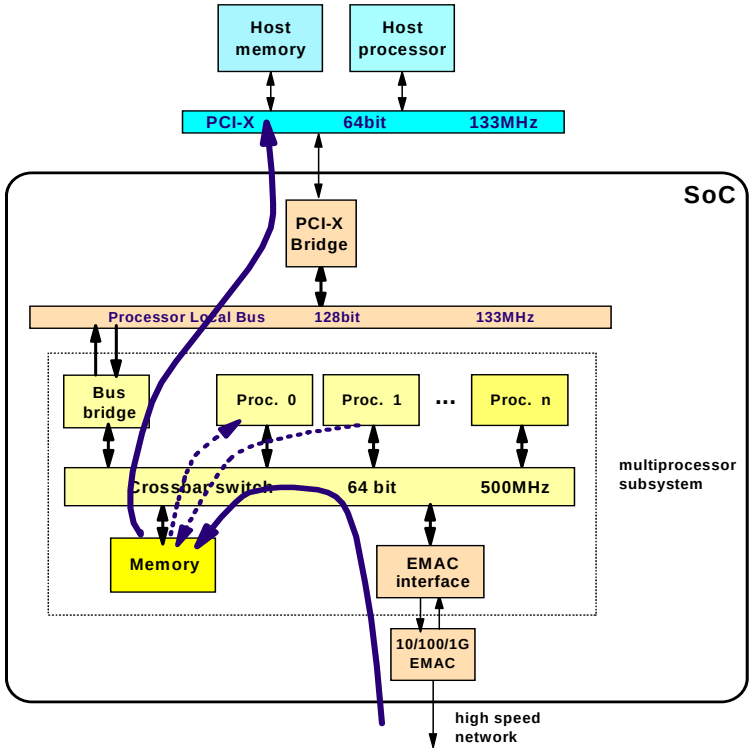

Figure 5: Multiprocessor macro used for host-bus adapter application

3. Case study

To prove our methodology, we evaluated a SoC design using our multiprocessor subsystem in an HBA (Host-bus adapter) application, as shown in Figure 5. The HBA functions as an interface between a host computer and a storage network which communicate with each other using the iSCSI protocol [10]. The host applications and the storage devices are presented with a traditional SCSI (ANSI standard X3.131-1986) interface, while the TCP/IP protocol is used for the actual communication between the two end-points.

The HBA performs the protocol handling operations and is used to transfer data in the form of TCP/IP packets from the Ethernet to the host (receive) and from the host to devices on the network (transmit). The main components of the system involved in the receive and transmit operations are shown in Figure 5.

As previously discussed, the multiprocessor sub-system consists of several simple cores (which are packed into clusters of eight cores) and an internal memory that are connected via a crossbar switch. The module also includes a bus interface and an interface to an Ethernet media access controller (EMAC). Both these interfaces contain buffers and implement DMA controllers for data coming into and going out of the multiprocessor subsystem. In the system shown in Figure 5, a PCI bridge provides an interface between the host's PCI bus and the SoC bus. Note that we show single processor cores rather than processor core clusters in the Figure 5, and does not specify the size of the memory. This is because our analysis will determine the number of processor cores and size of the memory needed for particular network speed and packet size.

Increased system throughput is achieved through coarse-grained parallelism, where a dispatch process running on a computing element in the multiprocessor subsystem allocates the processing of a single incoming packet as a single task to another available computing element.

In the receive mode, TCP/IP packets that arrive over the Ethernet are received via the SoC's Ethernet port. The packets are then moved (via DMA) from the EMAC's internal FIFO into the multiprocessor subsystem's internal memory over the crossbar switch. After the packets have been processed at the subsystem, they are transferred via the SoC bus and the PCI-Bus bridge to the host that is connected to the PCI bus. The basic flow associated with the reception of a single packet is shown in Figure 5. The solid connector lines show the basic packet movement in the system, while the broken connector lines depict the packet header being read in and written out by the processing elements. In the transmit case, packets stored on the PCI device are read into the system, processed by

the microprocessor macro and then transmitted over the Ethernet link.

The actual protocol conversion code is performed on the processing elements contained in the microprocessor macro. The macro has several processes P0, P1 … Pn running in parallel - one set for each direction (i.e., receive and transmit). Each of these processes is mapped to one of the macro's processing elements. Three different kinds of processes run on the macro's processors:

1. Dispatch: a process that allocates tasks to processors
2. Protocol processing: protocol-processing tasks
3. Collect: sets the DMA controller to transfer the packet out of the core's internal memory as well as to perform some memory management functions, after the packet has been transferred.

Communication between these processes is accomplished via work queues which are basically dedicated areas in memory. An idle process determines whether it has any pending work by periodically polling its work queue.

4. Results

The amount of actual computation that is required of the microprocessor macro depends on the number of packets that are flowing through the system. This, in turn, is determined by the workload, i.e., the Ethernet data rate, as well as the size of the packets that are being processed.

We decided to adopt a simulation-based approach that we assumed would help us correctly understand and analyze the behavior of the system while processing a workload. We captured the behavior of all system blocks shown in Figure 5 in an abstract model that was written in SystemC 2.0 [11]. SystemC is a C++ class library that is emerging as a standard for modeling hardware and software systems. It provides the necessary constructs for modeling system behavior in terms of timing, concurrency and reactivity. In building our system model, we abstracted out the functionality of the cores and considered their performance impact alone.

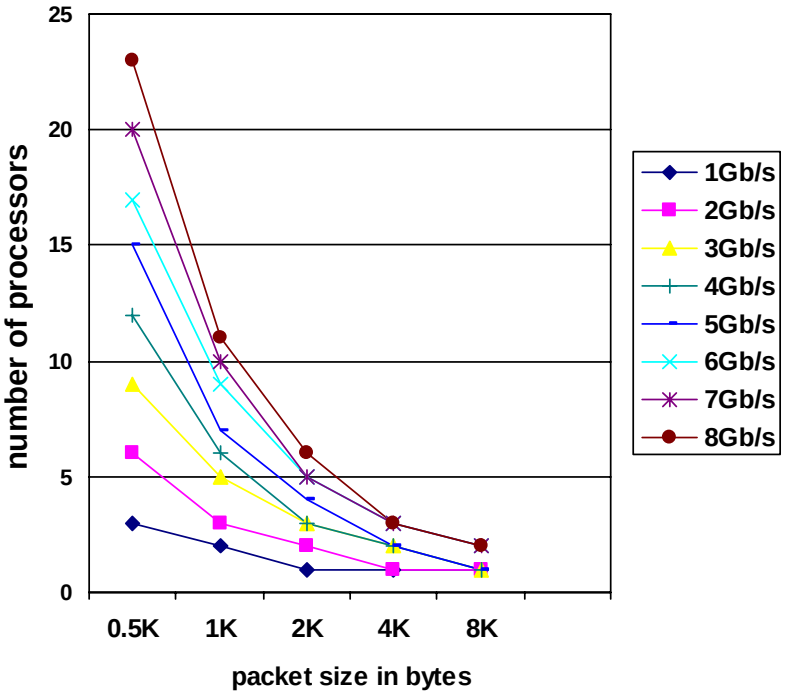

Figure 6: System requirements as a function of packet size

In our analysis, we used an estimate for the processing demands that would be made by the application software running on the processors. This estimate in terms of processor cycles needed, etc., was made by analyzing profiling data from sample application runs. For our experiments, we assumed that the TCP/IP connection setup and teardown functions are being performed on the host processor.

We conducted a set of experiments to determine the number of processors that would be needed to process a workload consisting of a continuous stream of equal-sized packets being received from the Ethernet with data rates being varied from 1 Gb/s to 10 Gb/s.

Figure 6 shows the results obtained. The checksum computation is performed by the microprocessor subsystem's EMAC interface during data transfer. Thus, the amount of time spent by the processors on packet header processing is basically independent of packet size. A workload consisting of higher data rates and shorter packets will result in higher packet frequencies, thereby requiring a greater number of processors to handle the increased processing demands. Our system was unable to process Ethernet data rates of 10 Gb/s not because of any limitations in the microprocessor subsystem, but because the PCI-X bus became the system bottleneck.

As shown in Figure 6, the number of processors needed to handle traffic on a single high speed network decreases with an increasing packet size, as the per packet computation is the same (in our design, the CRC calculation and data copying are supported by hardware). Thus, to handle a continuous stream of 512-byte packets with the minimal inter-frame gap –which is a worst case scenario, not a typical mode of operation- the multiprocessor subsystem has to instantiate anywhere from 3 processor cores, for 1 Gb/s network, to 23 processor cores, for 8 Gb/s network. But even at this high speed, only three processor cores were sufficient to handle jumbo Ethernet packets (i.e., 9,000 bytes).

Speed	Proc.	Clusters	Min.Buff	SRAM
1Gb/s	3	1	3KB	32KB
2 Gb/s	6	1	6KB	32KB
3 Gb/s	9	2	9KB	64KB
4Gb/s	12	2	12KB	64KB
5Gb/s	15	2	15KB	64KB
6Gb/s	17	3	18KB	96KB
7Gb/s	20	3	21KB	96KB
8Gb/s	23	3	24KB	96KB

Table 1: System requirements as function of network speed

Further details on memory requirements for various network speeds are given in Table I. The minimal number of processor cores needed to handle packets increases as the network speed increases. But since processor cores are organized in clusters, increases are made in terms of the number of clusters. This can result in the underutilization of some processor cores in a cluster, but it is more than justified by the high cost to redesign and re-verify a cluster of variable size.

We also list minimal memory requirements to provide buffering of packets during the processing of a single packet on both the receiving and transmitting sides, and the SRAM size typically associated with a cluster. The memory is sized to provide smooth operation during packet reception and transmission. More memory can be added to provide higher level of buffering, e.g., for handling congestion for a specific network interface or for buffer reordering. Note that we do not have to keep the entire routing table in the subsystem memory but only the active open connections, which are usually of fairly small size.

Because of the simplicity of the processor cores, area requirements for the microprocessor subsystem are minimal. For example, to incorporate a module into the SoC design capable of handling 2 Gb/s network speeds, we need an 8-processor core cluster with 32 Kbytes of memory. This is sufficient to handle the worst case workload and provides enough buffering for smooth operation under normal conditions. An 8-processor cluster with its associated 32Kbytes of I-cache and 32 Kbytes of local SRAM occupies approximately 8 mm^2 in 0.13 μm process ASIC technology.

5. Summary and Conclusions

In this paper we have shown a methodology for designing SoCs for networking applications. The methodology is based on the notion that computation intensive parts of the application code can be off-loaded to a multiprocessor accelerator on the chip. The multiprocessor subsystem is a self-contained macro whose computational power (i.e., number of processors and memory banks) are treated as a parameter at the time of the SoC specification, depending on the type of application that needs to run on the SoC. We have shown through simulations how the parameters can be chosen for a TCP/IP offload engine application, based on line-speed performance.

References

[1] C.J.Georgiou, V. Salapura, and M. Denneau, "Programmable Scalable Platform for Next Generation Networking," Proceedings of 2nd Network Processor Workshop, NP2, in conjunction with HPCA-9, Feb. 2003, Anaheim, CA, pp. 1-9.

[2] A. Brinkmann, J.C. Niemann, I. Hehemann, D. Langen, M. Porrmann, and U. Ruckert, "On-Chip Interconnects for Next Generation System-on-Chips," Proceedings of ASIC2003, Sept. 26-27, 2003, Rochester, New York.

[3] IBM Corporation, "CoreConnect bus architecture," http://www-3.ibm.com/chips/products coreconnect/

[4] IBM, "IBM introduces PowerPC 440 embedded processor," http://www-3.ibm.com/chips/news/2003/ 0922_440ep.html

[5] ARM, "Processor Cores Overview," http://www.arm.com/armtech/cpus?OpenDocument
[6] MIPS, "MIPS32 4KP – Embedded MIPS Processor Core", http://www.ce.chalmers.se/~thomasl/inlE/mips32_4Kp_brief.pdf
[7] M. Heddes, "IBM Power Network processor architecture," Proceedings of Hot Chips 12, Palo Alto, CA, USA, August 2000, IEEE Computer Society
[8] "NEC's New TCP/IP Offload Engine Powered by 10 Tensilica Xtensa Processor Cores," http://www.tensilica.com/html/pr_2003_05_12.html
[9] T. Wolf and M.A. Franklin, "Design Tradeoffs for Embedded Network Processors", in Proceedings of International Conference on Architecture of Computing Systems (ARCS) (Lecture Notes in Computer Science), vol. 2299, pp. 149-164, Karlsruhe, Germany, April 2002. Springer Verlag
[10] Internet Engineering Task Force, "Small Com-puter Systems Interface protocol over the Internet (iSCSI) Requirements and Design Considerations," Request for Comments: 3347, ftp://ftp.rfc-editor.org/in-notes/rfc3347.txt
[11] Functional specification of SystemC 2.0, http://www.systemc.org/