In Proc. of the 18th IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM),
Charlotte, United States, May 2010.
Revised version.

A TDMA Ethernet Switch for Dynamic Real-Time Communication

Gonzalo Carvajal
Dept. of Electrical Engineering
Universidad de Concepcion, Chile
gcarvaja@udec.cl

Sebastian Fischmeister
Dept. of Electrical and Computer Engineering
University of Waterloo, Canada
sfischme@uwaterloo.ca

Abstract

A real-time communication medium must provide a special coordination mechanism to guarantee bounded communication delays. Implementing this mechanism in software offers flexibility but reduces reliability and performance. On the other hand, customized hardware solutions deliver high throughput and predictability, but they increase the implementation cost and are unable to adapt to the specific needs of individual applications.

In this work, we introduce a switch that implements a programmable dedicated time-triggered packet switching mechanism on top of Ethernet. The switch, called the Network Code Switch bases on the NetFPGA system and executes flexible but verifiable state-based schedules encoded in the Network Code programming language. This permits the user to tailor the communication behavior to the needs of the distributed application with verifiable performance.

We discuss our experience starting at the designing to the implementation of the prototype, and describe how we exploited modularity and code reutilization to reduce the implementation costs and increase the flexibility of the architecture. We also validate our design by evaluating the overhead and throughput of the implemented prototype.

1. Introduction

Distributed real-time applications require an interconnect network that provides delay guarantees for communication among co-operative tasks running on multiple nodes. Nowadays, Ethernet is the most popular wired Local Area Network (LAN) technology, thus it is natural to consider using the existing Ethernet infrastructure for real-time communication. However, the non-deterministic behavior of the Ethernet protocol (e.g., retransmissions caused for the collision detection mechanism and/or dropped packets and Nagle's algorithm [1]) prevent it from providing hard bounds to communication delays. Therefore, Ethernet nodes require a different arbitration mechanisms to coordinate access to the medium and provide a higher degree of predictability.

These mechanisms are usually implemented in software as a higher-level communication layer running on top of a standard Ethernet card [2]–[4]. However, previous work showed that executing this software layer requires expensive computational resources [5] and generates high jitter in the communication delays [6], limiting its applicability to soft real-time systems with reduced throughput requirements and relaxed timing constraints.

In a recent work [6], we addressed this limitation by introducing the Network Code Processor (NCP), which is an Application-Specific Instruction Set Processor (ASIP) that coordinates access to the shared medium by executing Time-Division Multiple Access (TDMA) based schedules on each node. Using Field Programmable Gate Array (FPGA) devices, the implemented system exploits hardware parallelism and the deterministic behavior of customized devices to provide a throughput close to the line rate of a 100Mbps Ethernet network with high predictability of communication delays [7]. This paves the way for using the system on industrial applications with hard-real time requirements. However the previous solution required a special hardware device for each node connected to the network, we will now try to alleviate by creating a special network switch that contains all the hardware for the network in a central place.

In this work, we present the concepts and work for a scalable architecture that interconnects multiple instances of the NCP running concurrently on a single FPGA fabric. By using programmable TDMA schedules to coordinate passing data between the core instances, the system provides a dedicated hard real-time switching service on top of Ethernet. Using this architecture, we implemented a functional prototype on the state-of-the-art NetFPGA platform [8] and accessible for download[1]. The resulting system delivers a switching throughput that exceeds 200Mbps, and it offers remarkable advantages like full integration with any standard workstation, allowing us to exploit all the performance and predictability benefits related to the use of customized hardware, at the same time of providing familiar interfaces that simplify the task of debugging and reconfiguring the system to target application.

Besides the interesting technical aspects, we also documented some of the experiences gathered through the process of designing the multi-core architecture from the existing single-core prototype. We describe how we exploited code reutilization and modularity to reduce considerably the implementation costs, and simplify the task of customizing and scaling the hardware design to fit any specific requirement.

The remainder of this paper is organized as follows: Section 2 reviews the basic concepts of the Network Code framework. Section 3 presents the definitions for the multi-core architecture. We describe the practical implementation over the NetFPGA platform in Section 4, and report related measurements in Section 5. Section 6 summarizes some

1. http://netfpga.org/foswiki/bin/view/NetFPGA/OneGig/RealTimeSwitch

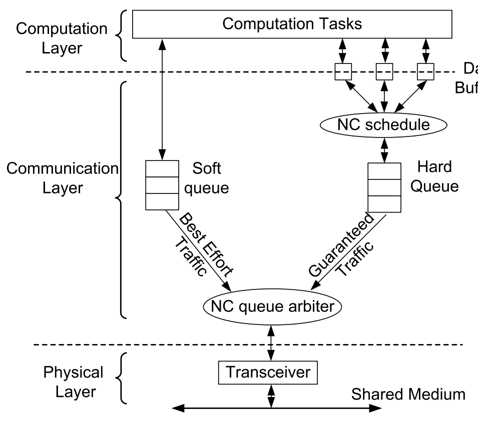


Fig. 1. Overview of the Network Code Framework

lessons learned during the implementation process, and finally the paper ends with the conclusions in Section 7.

2. The Network Code Framework

The Network Code framework enables implementing real-time communication systems for distributed applications. Its main components include: (1) a domain-specific language to represent state-based TDMA schedules, (2) a compiler with a verification engine that translates the programs into checked executable schedules, and (3) the interpreter entity that executes the schedule. We have extensively described these elements in previous works [6], [7] and now summarize the key elements related to the Network Code system definition, the programming language, and the hardware implementation of the interpreter, which is the main focus of this work.

2.1. System Overview

The Network Code framework consists of a layered architecture that separates the computation task from the communication system. Figure 1 provides an overview of the system [6]. The computation tasks reside on top and they access the medium through the services provided by the programmable communication layer.

Tasks produce and consume predictable or *hard values*, and non-predictable or *soft values*. The system guarantees delivery of hard values *within well-defined time bounds*, and they are useful for communicating time-critical computation results. On the other hand, the system transmits soft values on a *best-effort* basis without delay guarantees. Soft values have limited use for time-critical data, but they are still valuable for transmitting QoS-based services such as background communication or status information. In this paper, we focus on describing the system implementation for hard values and guaranteed traffic. For further details about the soft values and its system implementation, we redirect the interest reader to [6].

The computation and communication layers are two independent entities that interact in a time-triggered fashion. The layers exchange data by reading and writing into predefined data buffers at specific times. The data flow for guaranteed traffic works as follows: the task writes a computed value into a specific buffer. The Network Code schedule specifies the precise times for moving this value from the buffer to the hard queue and transmitting it on the medium. The communication layer is ignorant of the underlying physical medium and communicates with the Physical Transceiver (PHY) using queues. The receiver node knows that the medium will carry a data packet at a specific time, and its schedule is ready to receive the packet and store its contents into a specific data buffer. The task on the receiver node then reads this buffer and processes the received data.

To use guaranteed traffic, all the nodes must synchronize their local clocks to a global clock. We achieve this by synchronizing all nodes with a start of cycle packet; however, more sophisticated methods are also viable [9]. Furthermore, all nodes must share their buffer data structures. In previous work, we showed that as long as the traffic follows a well-defined temporal pattern, we can apply static verification [7] and analysis [10] to compute hard bounds for the communication delays, evaluate system correctness, and detect potential problems before than they occur in the final implementation.

A unique feature of Network Code is that it implements state-based schedules, meaning that a schedule can branch within a communication round based on history information, signals, and counters. For example, depending on the previously computed data and required precision, a node might sometimes require communicating two slots within one round instead of just one slot. A state-based schedule can express this with a decision point after the first slot and will continue either with the second slot for the node or with some other slots. This requires either reliable communication medium achievable through dedicated hardware [11] or special mechanisms in the schedule for preventing and handling faulty decisions.

2.2. Network Code Language

In the following paragraphs, we briefly describe the basic instruction set necessary to implement state-based schedules. For simplicity, we intentionally omit some parameters and the semantic formalities described in [12].

The create() instruction creates a message from data stored in a named data buffer. The send() instruction encapsulates a message into a network packet and signals the physical layer to start transmitting. The receive() instruction stores the data of an incoming packet into a data buffer. The branch() instruction implements conditional jumps based on buffer values, counters, or other status information. The sync() instruction signals a new communication round and synchronizes all nodes to the round. The instructions future() and halt() implement temporal control using timers that resume execution at particular program labels at specified points in time.

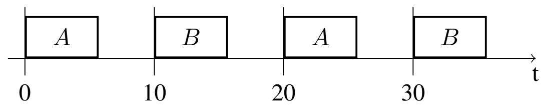
Figure 2 shows program examples that provide bounded communication delays to four nodes connected through a shared medium. Figures 2(a) and 2(b) are nodes sending hard values. Figures 2(d) and 2(e) are nodes receiving hard values. For simplicity of the example, we assume that all the nodes start simultaneously at time 0 and there is no clock skew. We also assume that 5 time units are sufficient time to create and propagate a message through the medium.

```
L0: create(msga, A)
    send(1, msga)
    future(20, L0)
    halt()
```

(a) Sender 1 (snd_1)

```
    future(10, L0)
    halt()
L0: create(msgb, B)
    send(2, msgb)
    future(20, L0)
    halt()
```

(b) Sender 2 (snd_2)



(c) Visual schedule.

```
    future(5, L0)
    halt()
L0: receive(1, A)
    future(20, L0)
    halt()
```

(d) Receiver 1 (rcv_1)

```
    future(15, L0)
    halt()
L0: receive(2, B)
    future(20, L0)
    halt()
```

(e) Receiver 2 (rcv_2)

Fig. 2. Network Code schedules for guaranteed traffic.

At time 0, $snd1$ creates a message from variable A using the alias $msga$, sends it using the logical channel 1 (all channels are mapped to the same communication medium), and then set an alarm in 20 time units to continue at label L0. All the other nodes stay on a waiting state. Node rcv_1 is waiting for $msga$, and then at time 5 leaves its waiting state, receives the message from channel 1, and stores the data into the local variable A. The schedules on snd_2 and rcv_2 follows a similar behavior, but with a delay of 10 time units in relation to the other nodes. At time 20 the operation starts again, and this behavior continues endlessly as the system runs.

2.3. Single-Core Network Code Processor

Figure 3 shows a block scheme of the Single-Core Network Code Processor (SC-NCP), the hardware ASIP that executes the programmed schedules. The system has three main components: the memory space, the Network Code Core (NCC), and the Ethernet core.

The memory space consists of three blocks. The *Program ROM* stores the programmed schedule. The *Msg-config* block defines the data buffers through an initial address and length that are mapped to locations on the *Msg-data*, which holds the actual data and serves as the interface between the computation task and the communication layer.

The NCC block executes the programmed schedule using a super-scalar architecture. Hardware units independently execute Network Code instructions. The controller reads the

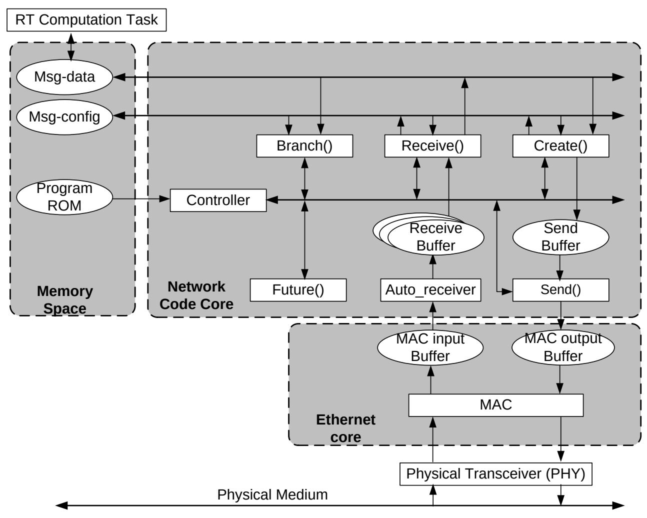


Fig. 3. Block diagram of the Single-core NCP.

instructions from the Program ROM, and processes them using a MIPS-like *fetch-decode-execute* approach [13]. The controller exploits hardware parallelism by calculating dependencies between consecutive instructions, triggering concurrent execution whenever it is possible [6].

Finally, the Ethernet core provides the interface to the physical layer. It uses an Ethernet MAC to process network packets, and interfaces the NCC using FIFO buffers.

The data flow for transmission of a given variable A works as follows: the create() block copy the data from buffer A to the internal send buffer. The send() block encapsulates the data into a Ethernet frame, stores it in the MAC output FIFO, and signals the MAC to start transmission. The created frames use a special identifier for the *EthType* field, and in addition to the actual data, the payload also includes information such as the length of the transmitted variable and the channel identifier. Finally, the MAC adds the headers and checksum, and put the packet into the communication medium.

On the receiver side, the MAC automatically detects incoming packets and triggers the asynchronous autoreceive() block. This block performs a deep packet inspection to verify the frame and store the useful data in the corresponding receive buffer (there is one buffer per channel). The receive() block can then read the data from the specified buffer and store it on the Msg-data location corresponding to variable A. If the controller does not execute a receive() instruction, the automatically received data stays in the buffer until being replaced by another packet arriving from the same channel.

3. Going Multi-core

We now introduce the Network Code Switch (NCS) model. The proposed system expands the superscalar architecture of the Network Code Processor to a multi-core device that connects multiple instances of the SC-NCP running concurrently in a single chip, offering a Network Code-based real-time packet switching service to external Ethernet nodes.

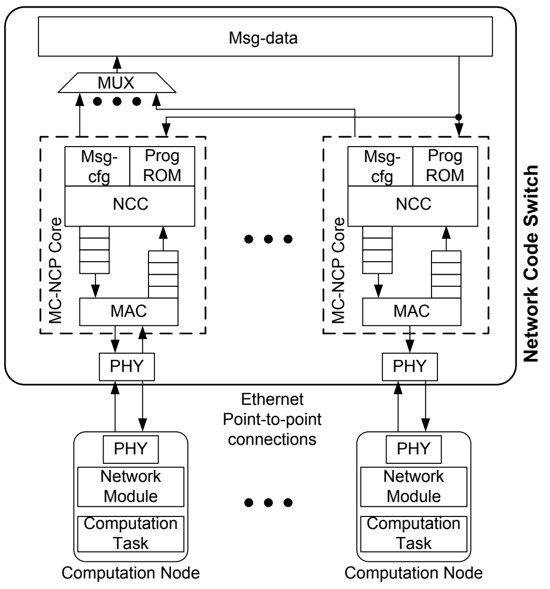


Fig. 4. Overview of the multi-core architecture

3.1. The Network Code Switch Model

Figure 4 shows a block diagram of the proposed multi-core system. The system contains multiple instances of the *Multi-Core Network Code Processor* (MC-NCP) core, the main functional block that encapsulates all the components defined in Figure 3 for the single-core processor, with the exception of the Msg-data memory block. The data memory is now a single block that all the cores share through a multiplexer. Each core interfaces to an independent PHY that connects it to external nodes through a point-to-point Ethernet connection.

The shared memory scheme brings two practical advantages in comparison to assigning independent blocks to each core:

- It optimizes the usage of memory resources by avoiding the need of holding redundant variables. Also, with each core holding its own Msg-config block, we can define asymmetric maps to distribute the memory according to the specific needs of each core.
- It offers a direct way to exchange data between the cores. We can define schedules to receive data from one port, store it in memory, and transmit it later through a different port, granting the system with the real-time switching capabilities through time-triggered communication.

The only valid interaction between the cores is the data exchange using the shared memory. Accesses to this memory are coordinated by the single multiplexer that assures that only one core will access the memory at the same time. Similar to the stated on the SC-NCP design, where the verification characteristic of Network Code schedules allows the system to omit specific hardware mechanisms for error detection and recovery from abnormal conditions (e.g. invalid memory accesses or incompatible data formats) [6], we now assume that the final schedules are checked beforehand to avoid simultaneous access requests to the shared memory from multiple instances at execution time. This assumption contributes to retain a low footprint.

To retain the design as simple as possible, we specifically targeted the system as a dedicated time-triggered communication device for network nodes performing safety-critical distributed applications. Therefore, unlike the system proposed in [4], our device offers no support for handling standard (or event-triggered) Ethernet traffic, and it will only process frames tagged with an specific identifier in the *EthType* field.

To illustrate an usage scenario for the new device, let us consider the closed-loop control system for distributed medical devices described in [14]. The selected clinical environment required certified performance to assure the patient integrity. On the reported setup, each medical device connected to a standard Ethernet switch through a NCP-based communication dongle executing a verified Network Code schedule. To assure the integrity of the communication between nodes, the schedules must consider the latency by the switch. However, this is an statistical parameter that varies among manufacturers and models. The developer must then assume always the worst case delay to assure the validity of the schedules, reducing the effective throughput of the system.

Unlike standard switches, the latency of the NCS will be known in advance with a cycle-accurate resolution. Besides, the user can apply model checking to compute and verify the switching times before the execution. Then, considering the previous example scenario, replacing the generic switch with the NCS will bring different technical advantages:

- We can use the NCS to complement the individual communication dongles on each node. In this case, we could apply model checking to the complete communication path, and we will rely on deterministic delay parameters to coordinate the communication between the nodes with a cycle-accurate resolution, maximizing the effective throughput.
- If the nodes use a full-duplex point-to-point link to connect to the NCS (see Figure 4), it would be possible to generate real-time channels for generic Ethernet nodes, not necessarily based on Network Code schedules. However, the developer must take special care in the system design to guarantee the communication performance, because in this usage scenario, we can only apply model checking and verification to the internal switching parts and not to the whole system.

3.2. Architectural definitions

Instead of a very specific off-the-shelf multi-core system with a predefined number of cores, we want to offer a modular architecture enabling the user to easily create a n-core system according to its specific needs and available resources.

The architecture must scale and be easily upgradeable. Programmable hardware requires good planning to use limited resources, then, apart from minimizing the hands-on process of adding or removing a core instance, we must also reduce the overhead and provide a mechanism to predict the resource usage beforehand when scaling the system.

Figure 4 illustrate the decomposition of the overall system functionality into a set a modules. The main functional module is the MC-NCP core. This block encapsulates the main components of the SC-NCP architecture, providing a high-level of abstraction. Also, minimizing the shared resources among the modules increases the robustness, because a failure in one core can occur without affecting the rest of the system.

The shared memory and the multiplexer are always present independent of the number of cores. We defined a basic main wrapper to hold these elements and provide the environment to interconnect the MC-NCP instances. The address and data-input buses of the memory are connected to the output of the multiplexer. Each MC-NCP instance includes status flags to signal an access request (for reading or writing) to the memory. We use these flags as the control inputs to the multiplexer, which places the corresponding values into the memory buses.

Considering the previous definitions, the user can scale the system by simply adding a new MC-NCP instance, adapting accordingly the size of the multiplexer, and adding the logic to interface the corresponding PHY. This approach complies with the required characteristics: (1) it minimizes the overhead when changing the number of cores, and (2) the impact in resource usage of the main wrapper will be negligible in comparison to the size of the core instances. Consequently, the amount of used resources will be linearly related to the number of cores and we can easily estimate it in advance. However, despite the simplicity of the approach, the access to the multiplexer for the shared memory can become a bottleneck and future work would need to investigate a multi-core design with point-to-point connected shared memory.

4. System Implementation

In the previous section we presented the requirements for the building blocks of the modular architecture. In this section, we describe the customized system that uses these building blocks to implement a quad-core prototype based on the NetFPGA platform .

4.1. The NetFPGA Platform

The NetFPGA [8] is a low-cost and open platform targeted to the implementation of reusable hardware code for high-performance networking applications. The complete framework has three main elements: (1) a FPGA-based hardware board, (2) software tools used to communicate with the card from a host workstation, and (3) fully functional examples and application projects offered in an open-source basis.

The plug-in hardware board attaches to the Peripheral Communication Interconnect (PCI) bus of any standard PC. Its main components are one Xilinx Virtex-II Pro 50 FPGA that holds customized logic, some buffer memory (SRAM and DRAM), and a quad-port 1 Gbps PHY. For a detailed hardware spec of the current 2.1 version see [15].

The software tools include a Linux device driver and a set of basic utilities that allow programs running on the host PC to communicate with the hardware board. Using these utilities, the user can easily read and write the NetFPGA's internal memory, and even reconfigure the FPGA functionality by downloading bitfiles directly from the workstation without needing any additional cable or physical access to the board.

Finally, the most remarkable aspect of the NetFPGA comes from the third element: all the codes for example projects and applications, including the HDL code for the FPGA logic and all the software utilities, are offered in a open-source way through the official website [8]. This reduces considerably the cost of implementing new components by either directly using already tested modules or adapting them for any specific needs.

4.2. Implemented System

We implemented and tested the individual modules described in Section 3.2 using HDL languages. Apart from the basic functionality, we also included in the main wrapper additional logic to interface the customized logic with the PCI port of the host-workstation. A user can then utilize these modules to synthesize an up to four-core system on the NetFPGA platform. By doing this, we have verified the scalability properties of the architecture. We will discuss the details about the overhead and timing requirements of the scalable system in Section 5.

The resulting system is a board that we can attach to any workstation with a standard PCI port. Using a provided Linux driver and software utilities, the user has access to all memory mapped internal registers of the FPGA directly from the workstation. This allows the developer to define the device's functionality by writing the programmed schedule and the content of the Msg-config block that maps the shared memory using a friendly and familiar environment. Under the same principle, the user can also check the system status by reading the FPGA registers at any time, which provides a powerful debugging environment. This also expands the applicability of the system by offering a real-time communication service to computation tasks running locally on the host-PC.

The integration with the host-PC provides a high level of abstraction that allows the user to overcome one common practical limitation of customized hardware systems: the lack of friendly interfaces that complicate debugging and reconfiguring the system. As the scheme in Figure 5 shows, the user can create customized high-level tools and GUIs to perform sophisticated control and analysis without requiring physical access to the hardware or any knowledge of the underlying technology. In fact, it is possible to change, test, and debug the system functionality remotely from anywhere on the Internet.

5. Measurements and Technical Details

In this section we summarize the results related to the implementation of the multi-core system in the Virtex2 50

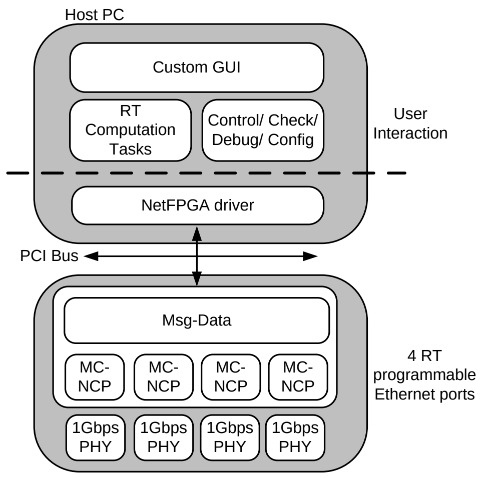


Fig. 5. Overview of the NetFPGA based system

chip included in the NetFPGA board. We synthesized the HDL design using the Xilinx ISE Foundation 10.1.3 tool.

5.1. Resource Usage and Timing on the FPGA

We mapped the logic of the MC-NCP block to generic logical slices, with the exception of the *Prog-ROM* and *Cfg-data* blocks that use one dedicated 16 Kb Block RAM (BRAM) each. The main wrapper uses logical slices for the multiplexer and the interfaces to the PHYs and PCI port, and it can use any amount of the remaining BRAMs or slices for the shared memory. The wrapper also uses Digital Clock Managers (DCM). We require one DCM to generate a global transmission clock for the PHYs, and one extra DCM per each instance to drive the clock for the PHY reception logic [16].

5.1.1. Multi-core Overhead. Figure 6 summarizes the reported device utilization for different numbers of cores. Starting from a fully functional single-core system, we successively added new instances modifying accordingly the size of the multiplexer and adding the interface to the PHY.

We can note that the usage of slices grows linearly at an approximated rate of 12.2% per instance. The synthesis tool reported that this increasing is mostly related to the logic of the MC-NCP cores, with the main wrapper having a negligible impact estimated in less than 0.3%. Considering a quad-core system (the NetFPGA board only provides four PHYs), the MC-NCP instances will use only 8 BRAMs (around 4% of the total available), leaving 96% of BRAMs, 51% of slices and 37.5% of the eight available DCMs for implementing the data memory or any additional functionality.

5.1.2. Multi-core Clocking. Table 1 summarizes the maximum speed for the clock driving the NCC logic that executes the programmed schedule on each instance. Unlike resource usage, it is complicated to extract a direct relation between the number of cores and the clock speed.

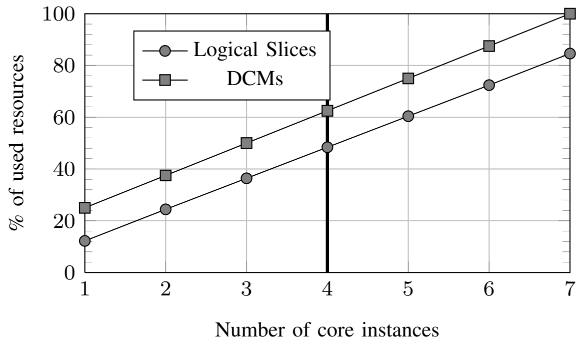


Fig. 6. Resource Usage on the NetFPGA

For a 1Gbps Ethernet connection, the PHY logic runs at 125 MHz. Because this speed is faster than the requirements for the NCC, each instance will work on two different clock domains: one driving the super-scalar NCC block that executes the schedule, and another one driving the Ethernet MAC logic that interfaces the PHY. The NetFPGA board provides the 125 MHz clock for the PHY, and we use a shift register to generate a 41 MHz common for all the instances.

TABLE 1. Clock requirements for the NCC logic

	1 core	2 cores	3 cores	4 cores
Period [ns]	14.1	13.9	16.3	16.3
Frequency [MHz]	70	71	61	61

5.2. Data Timing and Throughput

We now investigate the timing and derive the throughput expressions using the same method as outlined in [6].

Because the MAC logic runs faster than the processing speed of the NCC, the MAC must wait until all the data is stored into the MAC output buffer before start transmission (see Figure 3). Therefore, the time needed for a transmission operation through a create()-send() sequence is $t_{tx} = t_{tx,n} + t_{tx,m}$, where $t_{tx,n}$ and $t_{tx,m}$ are the times required for moving the data from the Msg-data block to the MAC output buffer, and from the MAC buffer to the PHY, respectively.

Using a logic analyzer to measure the *exact* number of cycles taken for both transmission stages, we derive the following expressions that give an accurate value for the time of a transmission operation for different variable lengths:

$$t_{tx,n}(B) = (30 + \max(B, 40)) * 0.0244[\mu s] \quad (1)$$

$$t_{tx,m}(B) = (44 + \max(B, 40)) * 0.008[\mu s] \quad (2)$$

where B is the size of the data in bytes.

A *tick* is the quantum for the timer used by the future() instruction, and t_{tick} is the duration of a future(1,_) instruction. On this implementation, we set the value of t_{tick} to $4[\mu s]$, which according to (1) and (2), is enough time to generate and transmit an Ethernet packet of up to 90 bytes.

Because the system operates at a message resolution of t_{tick}, the effective transmission time $\hat{t_{tx}}$ is an entire multiple of t_{tick}:

$$\hat{t_{tx}} = ticks * t_{tick} = \left\lceil \frac{t_{tx}}{t_{tick}} \right\rceil * t_{tick} \quad (3)$$

where $\lceil\ \rceil$ is the ceiling function. We then calculate how often Eq. (3) fits into one second, and multiply the result with the transmitted bytes per variable to obtain the effective transmission throughput:

$$TP_{tx}(B) = \frac{1}{\hat{t_{tx}}} * \frac{B}{128} \text{ [Kb/s]} \quad (4)$$

We used the same methodology to obtain an expression for the switching throughput, where a packet is received in one port and transmitted later by a different port. Because only one core can access the shared memory at the same time, transmission will only start in the tick following the one at which the recipient finish to write the data into the memory; thus, the minimum length for this operation is two ticks. The recipient port can move the incoming data from the medium to the receive buffer concurrently with the transmission through another port. Thus, we reduce the contribution of the recipient node to the total switching time, to the time for writing from the receive buffer to the shared memory:

$$t_{rx,n}(B) = \left(6 + \frac{B}{4}\right) * 0.0244[\mu s] \quad (5)$$

Then, the resulting expression for the switching throughput is:

$$TP_{sw}(B) = \left(\hat{t_{tx}} + \left\lceil \frac{t_{rx,n}}{t_{tick}} \right\rceil * t_{tick}\right)^{-1} * \frac{B}{128} \text{ [Kb/s]} \quad (6)$$

Figure 7 shows the graphs resulting from Eqs. (4) and (6) for transmission and switching throughput, respectively. As we can see, the maximum throughput for both operations lies around 200Mbps. Even when the maximum throughput duplicates the obtained from the previous prototype based on a 100Mbps connection [6], it only corresponds to 1/5 of the line rate of the 1Gbps connection. The reason for this limitation comes from the clocking characteristics described in Section 5.1.2. Because the NCC block runs slower than the PHY, the MAC must wait until the complete packet is stored in the output buffer before start the transmission, generating a bottleneck that translates to unused cycles in the PHY. Although this characteristic can be seen as a clear performance limitation, we need to remark that in hard real-time applications speed predictability is much more relevant than speed [17]. We also point out that the clocking characteristics of the prototype are related to the specific hardware used for the implementation, and we could increase the throughput by using more powerful FPGAs to synthesize the design.

6. Discussion

Being based on an already existing and tested single-core system, the multi-core architecture allows us to exploit the concept of code reutilization to significantly reduce the implementation time and for fast prototyping of the new device.

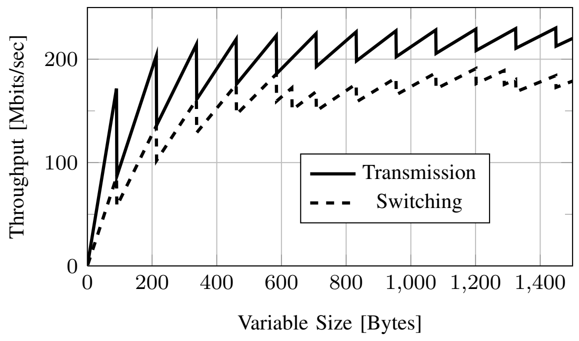


Fig. 7. Transmission and switching throughputs.

In this section we summarize some technical details and lessons learned during the process of porting the HDL implementation of the original Virtex4 based SC-NCP prototype [6] to the new NCS device targeted for the Virtex2 chip.

6.1. Exploiting Modularity

Modularity is a current trend in system design [18], [19] that consists on implementing high-level functionality by interconnecting more specific lower-level modules. This approach allows developers to reduce the design and implementation costs of new customized devices, because we can modify or expand the high-level functionality by simply adding, removing or updating existing modules.

We already discussed how modularity of the final architecture simplifies the task of scaling the multi-core system. However, we also validated the advantages of this approach when porting the existing SC-NCP design for implementing the new MC-NCP block. At the implementation level, the superscalar architecture of the original SC-NCP intrinsically led to the interconnection of well-defined HDL modules (see Figure 3). In the new system, we reused most of the original modules, removing the data memory block was a direct step, and we just needed to focus on adapting some specific modules for the new hardware and timing characteristics. We performed these changes at an internal level on each individual module without affecting the functionality of the others.

6.2. Hardware Dependencies and Timing

The original NCP prototype was targeted for the Virtex4 architecture, and it uses specific hardware components like an embedded Ethernet MAC core and dedicated FIFO support for BRAMs to implement the internal buffers. These elements are unavailable in the Virtex2 chip used in the NetFPGA, and thus we had to replace them with equivalent versions using logical slices. Although this modification incremented the usage of standard cells for customized logic, removing the specific hardware dependencies also increased the portability of the MC-NCP block, which can be now directly implemented in a wide range of FPGA chips without any modification.

Changing the hardware and the network speed also modified the timing requirements. In the original prototype, the NCC block runs at 100 MHz and the PHY uses a 12.5 MHz clock for the 100 Mbps connection. This allows the MAC to start transmission as soon as the first byte arrives to the output buffer, enabling to operate with a throughput close to line rate. Due to the new clocking characteristics (see Section 5.1.2), we required to modify the communication interface between the NCC and the MAC to start transmission only when the complete packet is available in the buffer. Fortunately, and thanks again to the well-defined modular approach of the NCC, we only needed to perform minor changes on the receive command logic in a transparent way for the rest of the blocks.

6.3. Advantages of Open-source Projects

A remarkable characteristic of the NetFPGA platform is the availability of fully functional application projects in an open-source way. Among other advantages, open-source projects allow the developers to easily build on top or integrate the work of others, and increase the scrutiny of the code. Although our design does significantly differ from the reference pipeline proposed in [20], we were still able to reuse some functional blocks to provide the interfaces to the PHY and the PCI port with minor modifications. Thanks to the open-source characteristic, we relied on the detailed documentation and valuable feedback from the NetFPGA's users community to easily adapt these blocks to our specific needs.

7. Conclusion

We presented our work on moving from a single-core to a multi-core design in the specific case of a bespoke processor for real-time communication. The architecture relies on the previously reported Network Code Processor which led us to design the the architecture with the concepts of code-reuse and modularity in mind.

We used the architecture to implement a quad-core system on the NetFPGA platform that offers a switching service with bounded delays to four Ethernet nodes. The platform can be plugged in a host-PC and controlled, configured, and debugged through the command line in the host operating system which provides a familiar environment and ease of use.

Although the system incurs overhead and thus is unable to compete with a native 1Gb/s Ethernet connection, we need to point out that the aim of the system is real-time communications and it enables developers to encapsulate flexible but verifiable communication behavior that is important especially in safety-critical system.

The source code to the switch is available on the NetFPGA's website. We are currently looking for practical case studies to evaluate the impact of the system specifically in time-critical applications. We are also considering to add the capability of handling standard Ethernet traffic by inserting the system in the reference pipeline proposed for the NetFPGA applications.

Acknowledgment

The authors would like to thank Robert Trausmuth and Kevin Perry for their assistant in the Network Code Processor, Ronald Valenzuela for his work on the PCI and Linux interfaces, and Israel Figueroa for the valuable feedback on networking formalities. This research has been partly sponsored by NSERC DG 357121-2008, ORF RE03-045, and the Graduate Students' Exchange Program (GSEP) funded by Foreign Affairs and International Trade Canada (DFAIT).

References

[1] J. Nagle, "Congestion control in IP/TCP internetworks," http://www.rfc-editor.org/rfc/rfc896.txt, 1984.

[2] C. Venkatramani and T. Chiueh, "Design, implementation, and evaluation of a software-based real-time Ethernet protocol," in *Proc. of the Conf. on Applications, technologies, architectures, and protocols for computer communication*. NY, USA: ACM Press, 1995, pp. 27–37.

[3] P. Pedreiras, P. Gai, L. Almeida, and G. Buttazzo, "FTT-Ethernet: a flexible real-time communication protocol that supports dynamic QoS management on Ethernet-based systems," *IEEE Trans. on Industrial Informatics*, vol. 1, no. 3, pp. 162–172, Aug. 2005.

[4] K. Steinhammer, P. Grillinger, A. Ademaj, and H. Kopetz, "A Time-Triggered Ethernet (tte) switch," in *Proc. of the Conference on Design, Automation and Test in Europe (DATE)*. Belgium: European Design and Automation Association, 2006, pp. 794–799.

[5] J. Loeser and H. Härtig, "Real time on Ethernet using off-the-shelf hardware," in *Proc. of the 1st Intl Workshop on Real-Time LANs in the Internet Age (RTLIA 2002)*, 2002.

[6] S. Fischmeister, R. Trausmuth, and I. Lee, "Hardware acceleration for conditional state-based communication scheduling on real-time Ethernet," *IEEE Trans. on Industrial Informatics*, vol. 5, p. 3, 2009.

[7] S. Fischmeister, O. Sokolsky, and I. Lee, "A verifiable language for programming communication schedules," *IEEE Trans. on Computers*, vol. 56, no. 11, pp. 1505–1519, Nov. 2007.

[8] "Official NetFPGA project webpage," http://www.netfpga.org.

[9] H. Kopetz and W. Ochsenreiter, "Clock synchronization in distributed real-time systems," *IEEE Trans. in Computers*, vol. 36, no. 8, pp. 933–940, 1987.

[10] M. Anand, S. Fischmeister, and I. Lee, "An analysis framework for Network Code programs," in *Proc. of the 6th Annual ACM Conference on Embedded Software (EmSoft)*, South Korea, Oct. 2006, pp. 122–131.

[11] H. Kopetz, *Real-time systems: Design Principles for Distributed Embedded Applications*. Kluwer Academic Publishers, 1997.

[12] S. F. et al., "Network code language specification," University of Pennsylvania, Tech. Rep., 2007, manual & specification.

[13] J. Henessy and D. Patterson, *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 2006.

[14] D. Arney, R. Trausmuth, S. Fischmeister, J. M. Goldman, and I. Lee, "Plug-and-Play for medical devices: Experiences from a case study," *Biomedical Instrumentation & Technology*, vol. 43, p. 313 to 317, 2009.

[15] G. Gibb, J. Lockwood, J. Naous, P. Hartke, and N. McKeown, "NetFPGA : An open platform for teaching how to build gigabit-rate network switches and routers," *IEEE Trans. on Education*, vol. 51, pp. 364–369, 2008.

[16] X. Inc., "Using the RGMII to interface with the gigabit Ethernet MAC," 2006, application Note XAPP692.

[17] J. A. Stankovic, "Misconceptions about real-time computing: A serious problem for next-generation systems," *Computer*, vol. 21, no. 10, pp. 10–19, 1988.

[18] C. Baldwin and K. Clark, *Design Rules: The Power of Modularity*. The MIT Press, 2000.

[19] Y. Ro, J. Liker, and S. Fixson, "Modularity as a strategy for supply chain coordination: The case of U.S. Auto," *IEEE Trans. on Engineering Management*, vol. 54, pp. 172–189, 2007.

[20] J. Naous, G. Gibb, S. Bolouki, and N. McKeown, "Netfpga: reusable router architecture for experimental research," in *Proc. of the ACM Workshop on Programmable routers for extensible services of tomorrow*. New York, NY, USA: ACM, 2008, pp. 1–7.