September 2008

Hardware Acceleration for Verifiable, Adaptive Real-Time Communication

Sebastian Fischmeister
University of Waterloo

Insup Lee
University of Pennsylvania, lee@cis.upenn.edu

Robert Trausmuth
University of Applied Sciences

To be published in *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2008.

Hardware Acceleration for Verifiable, Adaptive Real-Time Communication

Abstract
Distributed real-time applications implement distributed applications with timeliness requirements. Such systems require a deterministic communication medium with bounded communication delays. Ethernet is a widely used commodity network with a large number of appliances and network components and represents a natural fit for real-time application; unfortunately, standard Ethernet provides no bounded communication delays.

Network Code Processor is a soft processor implementation for real-time communication on Ethernet. The system provides a smart network-card functionality and can be seen as a co-processor for time-triggered communication. Its most distinguishing feature, the programmability of the processor via the Network Code language, allows developers to write adaptive but verifiable communication schedules tailored to the application needs. In this work we present results around the development of the soft processor, discuss the specific challenges of how to build a reliable and fast communication system, the tradeoffs involved when moving from a generic software prototype to a programmable hardware implementation.

Comments
To be published in *Proceedings of the 13th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, September 2008.

In Proc. of the 13th IEEE Intl. Conf. on Emerging Technologies and Factory Automation (ETFA),
Hamburg, Germany, September 2008.
Revised version.

Hardware Acceleration for Verifiable, Adaptive Real-Time Communication

Sebastian Fischmeister
University of Waterloo
Waterloo, Canada
sfischme@uwaterloo.ca

Insup Lee
University of Pennsylvania
Philadelphia, USA
lee@cis.upenn.edu

Robert Trausmuth
University of Applied Sciences
Wiener Neustadt, Austria
trausmuth@fhwn.ac.at

Abstract

Distributed real-time applications implement distributed applications with timeliness requirements. Such systems require a deterministic communication medium with bounded communication delays. Ethernet is a widely used commodity network with a large number of appliances and network components and represents a natural fit for real-time application; unfortunately, standard Ethernet provides no bounded communication delays.

Network Code Processor is a soft processor implementation for real-time communication on Ethernet. The system provides a smart network-card functionality and can be seen as a co-processor for time-triggered communication. Its most distinguishing feature, the programmability of the processor via the Network Code language, allows developers to write adaptive but verifiable communication schedules tailored to the application needs. In this work we present results around the development of the soft processor, discuss the specific challenges of how to build a reliable and fast communication system, the tradeoffs involved when moving from a generic software prototype to a programmable hardware implementation.

1. Introduction

Modern real-time systems are used to implement distributed applications with timeliness requirements. An intrinsic property of such a system is that the correctness of the system depends on the correctness of values and the correctness of timing. This implies that a correct value at an incorrect time can lead to a failure. Consider a car with a brake-by-wire system, where the pedal communicates to the brakes when force is applied to the wheels. In this system, a correct value means that the brakes apply force to the tires only when the driver hits the brake pedal, and correct timing means that the time between the two events of one "hitting the pedal" and two "applying force" should be bounded. It is obvious that the system is only useful, if both—correct timing and correct values—are guaranteed.

A distributed real-time system adds the complexity of decentralized control to a shared communication medium. Connected nodes can access the medium and cause collisions in the network communication, which scrambles data and typically results in retransmissions. Since collisions are difficult to predict and retransmissions make it hard to place a bound on the communication delay, one primary research goal is to investigate effective coordination models for controlling access to this shared medium.

Ethernet is a widely used network technology in the embedded systems industry besides field bus systems. The market provides a large number of appliances and network components, therefore it is natural to try using Ethernet for real-time communication. Unfortunately, Ethernet's intrinsic non-determinism caused by the collision detection and binary back-off mechanism for resolving contention make it hard to provide upper bounds for communication delays on this platform. A number of systems propose different schemes, usually called real-time Ethernet, with different arbitration schemes to provide bounded delays and enable real-time communication.

Initial work on this topic proposed customized hardware [5, 20, 23] that provided guarantees for the system analysis and for high-level real-time software. At the time this initial research was done, custom hardware was an illusive assumption, because manufacturing it was too expensive. This motivated research to move towards commercial off-the-shelf (COTS) Ethernet components. Approaches using COTS advocate either statistical methods [13, 4, 14, 17] for traffic shaping and traffic prediction or higher-level communication frameworks [26, 21, 22, 25, 9, 6, 12] on top of the standard Ethernet card with a separate arbitration mechanism. However, running the framework and arbitration control on the workstation can cause a huge computation overhead in the processor [18] and is subject to high jitter (see Section 1.1).

Now the assumption of custom chips and custom logic is no longer illusive. Field programmable gate array (FPGA) technology now allows systems researchers to inexpensively build custom hardware running their real-time communication frameworks [24]. This development brings a number of benefits: (1) it provides a low jitter and high throughput environment which is unaffected by the

This research has been sponsored by AFOSR FA9550-07-1-0216, NSF CNS-0509327, NSF CNS-0721541, NSF CNS-0720703, NSERC DG 357121-2008, and by CIMIT under U.S. Army Medical Research Acquisition Activity Cooperative Agreement W81XWH-07-2-0011. The information contained herein does not necessarily reflect the position or policy of the Government, and no official endorsement should be inferred.

interrupt load inside the workstation and experiments produce more trustworthy data, (2) it removes as many layers as possible from the network stack and allows replacing them with customized layers removing side effects from the operating system, and (3) FGPA technology promotes reuse and custom real-time communication frameworks can be encapsulated into IP cores and reused by other research groups or industry.

1.1. Motivation

The general goal of our work aims for building an adaptive and verifiable communication system. The goal of this work is to investigate whether such system is technically feasible and what constraints it imposes on the environment in terms of speed, generality, and integration.

For example, the communication system of the original Network Code prototype was implemented in software [7] and resided in the network driver of a real-time Linux system. Although the code sits as close to the hardware as possible considering a full-blown operating system, still the system experiences high jitter which limits its applicability to experiments in industrial settings.

Figure 1 shows two box plots for execution jitter of instructions. The figure provides evidence that standard components introduce high jitter in a system. Let's consider the instruction send() which enqueues a message in the output queue. The statistical mode of this instruction is 372ns. If we consider the 99th percentile, then the execution time lies between with 371-733ns. If we increase the percentile and thus increase the timing reliability of our system (a correcter estimate of the execution time leads to less frequent fault caused by missed deadlines), then we will observe a drastic increase in execution time. For example the 99.9999th percentile leads to an upper bound of 19.090μs (26 times the original value). Although parts of the software might be optimized by correlating delays and dependencies using for example statistical models [16], the high variance still remains.

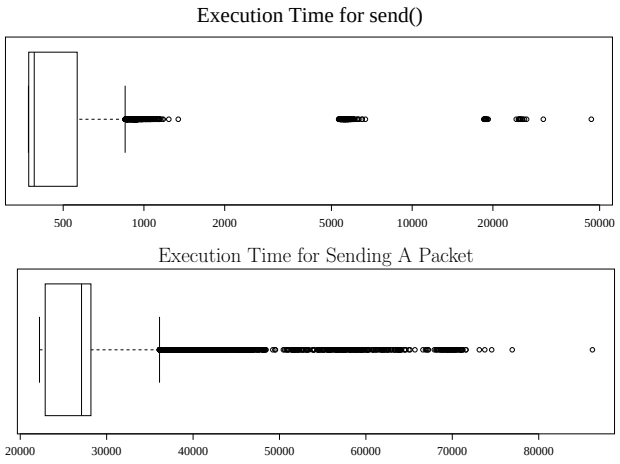


Figure 1. Execution jitter in [ns].

The aims of the work presented in this paper are multifold: 1) Is it possible to build a reliable and fast communication system for Network Code using programmable hardware? Fast means for us that the throughput is comparable with raw Ethernet; reliable means for us a low mean time to failure when meeting timing constraints. 2) What are the tradeoffs when moving from a software prototype to programmable hardware? Pure software still provides more flexibility in terms of programming constructs and available resources than programmable hardware, so we may need to trade system features and functionality for practicability in the system development. 3) How can we integrate the system with the computation system and its environment? For example, we will explore whether we can maintain the standard OS network driver interface, so legacy drivers work without changes.

2. Overview of Network Code

Network Code represents a domain-specific language for programming communication schedules and arbitration mechanisms for real-time communication. Network Code programs of a certain structure remain verifiable [7], analyzeable [2], and composable [3]. Furthermore, Network Code and its runtime can be seen as a programmable communication layer [8].

Network Code provides two distinct types of QoS: best effort and guaranteed. Messages sent using the *best effort* quality class have no bounded communication delay, as the transmission can fail infinitely often for various reasons including getting blocked by guaranteed traffic or collisions. Messages sent using the *guaranteed* quality class have bounded communication delays. We can apply static verification [7] and analysis [2] to compute bounds on communication delays as long as the traffic follows a well-defined temporal pattern.

Network Code also provides data control functionality for buffers. This functionality allows the developer to create messages from these buffers and transmit them on the network. The developer can use this to replicate buffers across multiple nodes following a specific temporal pattern. For example, given that a specific buffer holds the sensor readings: The developer can write a Network Code program that transmits the sensor readings to all nodes every ten milliseconds. Replicated buffers can act as input to control-flow decisions in the program. The conditional branching instruction if() allows the developer to code alternatives. For example, if the last sensor reading lies below a threshold, then the sensor will suspend sending updates for some time.

Figure 2 shows an overview of the programmable arbitration layer used for Network Code, and how it interacts with the queues and the computation tasks. For further details, see the prototype software implementation [7].

The Network Code language consists of nine instructions which control timing, data flow, control flow, and error handling. In the following, we provide two brief examples to demonstrate how Network Code works. Most of the instructions and parameters are intuitive, and parameters, which are unimportant for this work, are masked with the symbol '_'. For detailed descriptions, we direct the interested reader to [7].

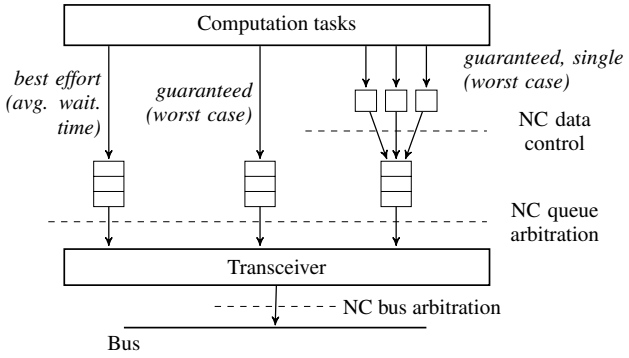


Figure 2. Overview of queues and controls.

As an example for virtual circuit-switched communication consider the following programs. Note that for sake of simplicity, we assume that both nodes start at the same time and there is no clock skew; also wait() is a composite instruction used for instructive purposes and not atomic.

```
Sender:
0 L0: create(msg_a, A)
      send(1, msg_a, _)
2     future(10, L0)
      halt()
```

```
Receiver:
0     wait(9)
  L1: receive(1, A)
2     future(10, L1)
      halt()
```

The sender first creates a packet from variable A using the alias msg_a. Then, it sends the message on channel 1, and sets up an alarm in ten time units to continue at label L0. It then halts execution (the halt() instruction) and waits for the alarm to resume operation. The receiver first waits nine time units for the first delivery of a message and then receives a this from channel 1 into the local variable A every ten time units.

As an example for packet-oriented communication, consider the following programs using the same assumptions as before:

```
Node 1:
0 L0: mode(soft)
      wait(4)
2     mode(hard)
      future(11, L0)
4     halt()
```

```
Node 2:
0 L1: wait(5)
      mode(soft)
2     wait(4)
      mode(hard)
4     future(1, L1)
      halt()
```

The instruction mode() controls the mode of operation of the run-time system. In the soft mode, the system offers best-effort communication, in the hard mode it provides guaranteed communication, and the init mode is used for setting up the system. The system guards access to the network through temporal isolation. Node 1 gets exclusive access to the medium during the first four time units, and Node 2 for time five to nine. While they have exclusive access, both nodes communicate soft values. Messages are automatically received through the transceiver and best-effort–traffic messages are logically separated from guaranteed-traffic messages (see Figure 2).

Note that Network Code also supports raw communication. In the previous example, only one node was in the soft mode at a time. If several nodes are in the soft mode, all of them might concurrently access the network.

3. Network Code Processor

The goals mentioned in Section 1.1 require an efficient architecture which maximizes data throughput and minimizes latency. Our architecture of choice for achieving this is a super-scalar application-specific instruction set processor (ASIP) [10, 11] with independent execution units for the individual instructions of Network Code.

The ASIP was designed, optimized and implemented by hand. Although there are several tools available for doing this, namely MESCAL [19] or commercially available packages like the Tensilica cores [15], we chose this approach for really having the hardware on our fingertips. Future research will show whether we can get similar results by using such tools.

In this section, we describe the analysis and development which lead to the ASIP called Network Code Processor (NCP). First, we analyze the control, data, and hardware dependencies among individual instructions. Second, we describe the concurrency controller in the superscalar architecture, and finally, we show an example that demonstrates the speed up compared to standard sequential execution.

3.1. Instruction Dependencies

Based on the operational semantics of Network Code, we can identify three types of dependencies: data dependencies, control-flow dependencies, and mode dependencies.

Control Dependence. Given two successive instructions, the second one will be control dependent on the first one, if its execution depends on the evaluation of a conditional guard expressed in the first instruction. Obviously, the instruction if() creates control dependencies in program. The instruction at the target address is control dependent on the if() instruction.

However, Network Code also has non-obvious control dependencies resulting from the instructions halt() and sync(). The instruction halt() terminates the current execution until an alarm trigger wakes up the runtime to resume operation. Clearly, the NCP cannot concurrently execute instruction sequences such as "halt(); create(...);", because it must halt after the first statement and continue only after a trigger event. The instruction sync() synchronizes distributed nodes by means of a synchronization packet. Nodes that wait for such a synchronization packet must not resume operation before (a) such a packet is received or (b) a timeout occurs. Therefore, the NCP cannot concurrently execute instruction sequences such as "sync(c,3000); create(...);". The same goes for the sender and specific instructions that cause packet transmissions, because the NCP must preserve causal ordering of packet transmissions.

Data dependence. Two successive instructions are data dependent, if they access or modify the same resource [1]. In our system, all data dependencies originate from the read/write access to the shared buffers in between the individual microcode blocks which implement instructions. For example, the two instructions "create(msg_a,_);

send(_,msg_a,_);" cannot be executed in parallel, because one instruction writes to a shared buffer containing the created message while the other instruction reads it.

Mode dependence. Two successive instructions are mode dependent, if the second instruction executes a mode change to a target mode and the first instruction is unavailable in this target mode. Typically, each instruction assumes a specific system state when it executes. A mode change might violate this assumption. The NCP can be in one of three operational modes: hard, soft, and sync. From this, we can derive the mode dependencies among instructions. For example, the instruction send() is used solely in the hard mode, and its operational semantics assume that this holds. However, this assumption creates a mode dependency between the instructions send() and mode(). For example, the following instruction sequence is valid "send(...); mode(soft);" and can be executed concurrently, while the following cannot "mode(soft); send(...);".

Summary. Table 1 shows a summary of the dependencies among instructions. The symbols $\xrightarrow{c}$, $\xrightarrow{d}$ and $\xrightarrow{m}$ denote a control, data, and mode dependence, respectively. The symbol $a \xleftrightarrow{\star} b$ denotes a dependence $a \xrightarrow{\star} b$ and $b \xrightarrow{\star} a$. The set G_1 consists of all guards except *AlwaysFalse*, the set G_2 contains all guards except *AlwaysTrue*, and set $G_3 := \{\text{TestVar}, \text{GreaterVarVar}, \text{CompareVarVar}, \text{LessVarVar}\}$.

Type	Dependence
Control	$if(G_1, jmp) \xrightarrow{c} (instr(jmp) \setminus \{nop\})$
	$if(G_2, \_) \xrightarrow{c} (instr(next(a)) \setminus \{nop\})$
	$halt \xrightarrow{c} instr(next(a))$
	$sync \xrightarrow{c} \{send, receive, halt, mode, if\}$
Data	$halt \xleftrightarrow{d} if$
	$sync(c, \_) \xleftrightarrow{d} if(StatusTest, \_)$
	$receive \xleftrightarrow{d} if(G_3, \_)$
	$receive \xleftrightarrow{d} create$
	$create \xleftrightarrow{d} send$
	$create \xleftrightarrow{d} if(\textit{SendBufferEmpty}, \_)$
	$destroy \xleftrightarrow{d} send$
	$destroy \xleftrightarrow{d} if(\textit{SendBufferEmpty}, \_)$
Mode	$mode \xleftrightarrow{m} \{sync, receive, create, destroy, send\}$
	$sync \xrightarrow{m} mode$
	$sync(c, \_) \xrightarrow{m} halt$

Table 1. Dependence summary

3.2. Concurrency Control

To minimize the number of stalls of concurrently executing microcode blocks, we optimized a number of cases that frequently occur in Network Code programs. For example, one of the most frequent instruction sequences is "create(); send();", which first creates a message in the send buffer and then transmits this message. According to the data dependencies shown in Table 1, these two instructions must be executed sequentially. However, as they occur that frequently, we optimize the NCP to allow concurrent execution of these two instructions by means of a data pipeline. We achieve this by (1) a FIFO queue between the two microcode blocks and (2) the send() instruction's delayed reading from this FIFO queue. The FIFO queue enables concurrent access, because while the microcode block implementing the create() instruction is still filling the queue, the microcode block implementing the send() instruction can already start reading from this queue. However, we have to make sure that the FIFO queue always contains data. To guarantee this, the send() microcode block first creates the Ethernet telegram's header (requiring about 30 cycles) before it starts reading the FIFO. Meanwhile, the concurrently executing create() block can already start filling the FIFO queue. Also, the send() block reads data four times slower than the create(), because the internal memory bus is 32 bits wide whereas the MAC interface only supports 8 bits.

Table 2 shows the summary of all dependencies for the NCP after optimizations. The meaning of the characters in the table are '*w*' for wait until finished, '*c*' for continue with next instruction and '*b*' wait until the memory bus is available. The table is read the following way: given two sequential instructions "x(); y();", the instruction x() specifies the column and y() specifies the row. For example, the snippet "if(); send();" results in a sequential execution as specified by *w*, while "send(); if();" can be executed in parallel as the Table 2 provides a *c*.

	nop	create	send	receive	sync	halt	future	mode	if
nop	w	c	c	c	c	w	c	w	c
create	w	w	c	b	c	w	c	w	w
send	w	c	w	c	w	w	c	w	w
receive	w	b	c	w	w	w	c	w	w
sync	w	c	w	c	w	w	c	w	w
halt	w	c	c	c	w	w	c	w	w
future	w	c	c	c	c	w	w	w	w
mode	w	c	w	w	w	w	c	w	w
if	w	b	c	b	w	w	c	w	w

Table 2. Summary of final instruction dependencies.

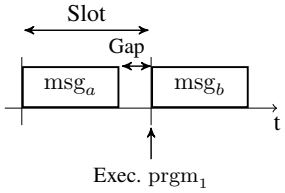
To simplify the implementation, the instructions mode() and nop() are synchronous instructions which always have to finish before the next instruction can start. The halt() instruction stops program execution, and the processor starts working only after receiving an interrupt set up by an earlier future() instruction.

The controller uses the running states of all the instruction blocks to calculate the locking conditions during the decoding phase. If Table 2 permits concurrent execution, the controller will trigger both microcode blocks. Otherwise, it will only trigger one and enter a waiting loop until the lock is resolved. After starting to execute one instruction, the controller immediately decodes the next instruction.

Before switching modes, the locking condition ensures that the network is available. In practice this means mode switching in a saturated network only occurs whenever a currently running transmission on the network reaches its inter-frame gap.

3.3. Example

Let's consider an illustrative example to show the benefit of the our selected architecture. Listing 3(b) shows one of the most common program snippets found in Network Code. Figure 3(a) shows how this program fits into the slot structure. The node executing this program first creates a telegram containing variable B which is then transmitted as telegram msg_b using channel 1. It also receives a telegram msg_a from the previous slot and stores its content in variable A.



(a) Visual structure.

```
1 L0:  create(msg_b, B)
       send(1, msg_b, _)
3      receive(msg_a, A)
       future(1, L0)
5      halt()
```

(b) Program prgm_1.

Figure 3. Most common structure in the programs and its encoding in the sending and receiving program.

Listing 3(b) must be executed within 10μs, because the instruction future() specifies a delay of 1 time unit which, in our implementation, equals 10μs. The future() instruction takes three cycles, and the halt() instruction requires two cycles to complete. Assuming that the size of the variables A and B are 128 words, the instructions create(), send(), and receive() then require 135, 547 and 543 cycles, respectively. The sequential execution of the whole program block requires 1230 cycles. However, since 10μs accommodates exactly 1000 cycles, this program cannot be executed sequentially.

Executing the same program on our superscalar architecture with the instruction dependencies as specified in Table 2, this program executes fast enough. First, the two instructions "create();send();" are executed in parallel, because the instruction send() can start right after create() has begun to fill the send FIFO. The instructions "send();receive();" can be executed in parallel, but the receive() instruction has to wait for the data bus occupied by the create() instruction. The program will thus be ready after 145 cycles and the processor will be halted; except for the receive() instruction which will still be active for another 533 cycles. Since this is less than 1000 cycles, this program can be executed by our processor.

Figure 4 shows the execution trace as a Gantt chart of the NCP for executing Listing 3(b). For each instruction, it first shows the loading time and then the actual execution in the microcode block. The upper part shows the sequential execution, which requires more than 1000 cycles. The lower part shows the execution trace of the NCP, which executes instructions in parallel and thus can execute the program in less than 1000 cycles therefore satisfying the requirements for the future(1,_) statement.

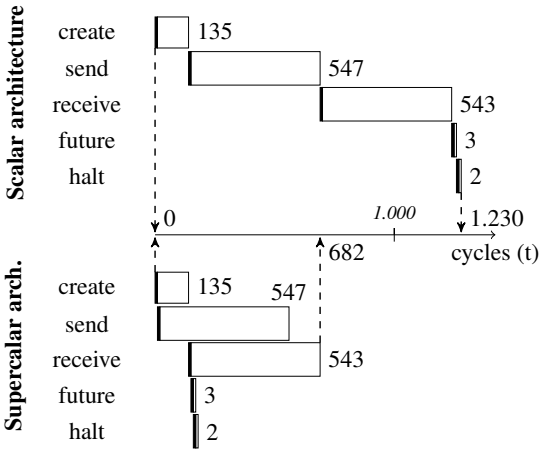


Figure 4. Scheduling of the example program shown in Listing 3(b).

4. Measurements and Results

For measurements and experimentation, we use two nodes that are directly connected with no active network components in between. The two nodes communicate with each other via a ping-pong program; specifically, Node A periodically transmits variable A, and node B receives it.

4.1. Throughput of FPGA Solution

The execution speed of the create(), send() and receive() instructions grows linearly with the size of the variable, which makes the system predictable. Because of this, the system throughput is a direct function of the execution speed and the variable size. Note that we calculate the actual throughput with a high precision, because the hardware is free from jittery influences such as interrupts, cache misses, and page faults. Figure 5 shows the maximal throughput of the FPGA implementation depending on the data size. The x-axis shows the variable size in Bytes, and the y-axis shows the throughput in kB/sec. Note that the data throughput differs from the actual network utilization: (1) Ethernet telegrams include a header which introduces overhead, and (2) telegrams have a specific minimum size, so padding must be added until 64 bytes and incurs overhead.

To calculate the performance of the FPGA implementation, we can use Equations 1 and 2. t_p specifies the computation time of the NCP, and t_s is the time required by the MAC layer to transmit a telegram. The components of t_p are instruction cycles executed at a speed of 100 MHz with 8 cycles setup time for the create() microcode block, 5 cycles for the send() microcode block, and $B/4$ cycles for copying the variable content of B bytes. The components of t_s are the size of the message (frame with 26 and the body with a minimum of 28 bytes and 10 bytes of inter-frame gap) times the transmission duration of 80ns per byte in the MAC.

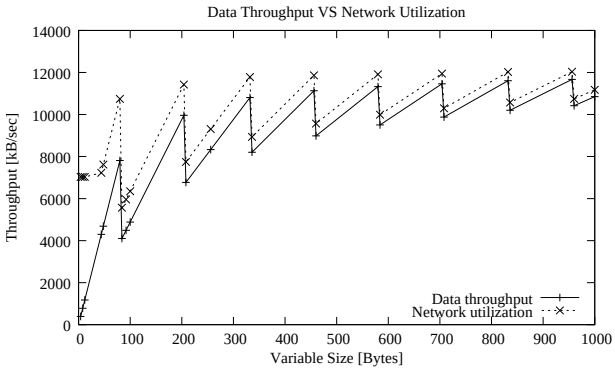


Figure 5. Throughput of the FPGA implementation.

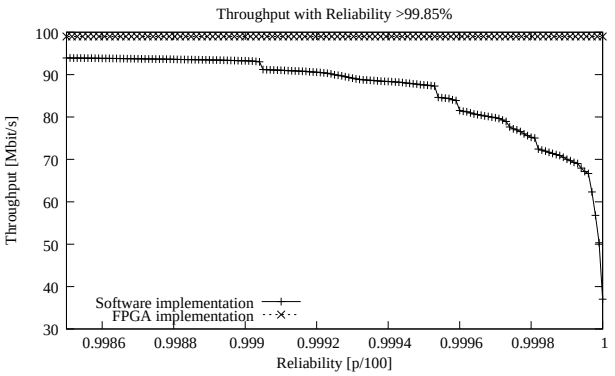


Figure 6. Throughput of the two prototypes.

$$t_p(B) = (8 + \tfrac{B}{4} + 5) * 0.010\mu s \quad (1)$$

$$t_s(B) = (26 + \max(B, 28) + 10) * 0.08\mu s \quad (2)$$

The data throughput TP can now be calculated by the following equation where step marks the minimal time granule in μs in the system:

$$TP(B, \mathrm{step}) = (\lceil \frac{t_p + t_s}{\mathrm{step}} \rceil * \mathrm{step})^{-1} * \frac{B}{1,024} \quad (3)$$

Figure 5 shows the throughput resulting from (3).

4.2. Software vs FPGA

To compare the performance between the software prototype [7] and the FPGA implementation, we use the same ping-pong program as mentioned before. The software prototype runs on the hardware setup outlined in [7] and the FPGA prototype uses the hardware mentioned before. The core of the quantitative evaluation is now to identify that maximum throughput while still obeying the following premises:

1. The slot structure must be preserved. The sending node must only communicate during its slot, so the i-th communication must take place in the time slot $[i \cdot \mathrm{step}, (i + 1) \cdot \mathrm{step})$.
2. The input queue must not overflow. The receiver must be fast enough to process the input queue as new telegrams arrive.

In the performance test, we run these programs on the software implementation and on the FPGA with different throughput values. We fixed the variable size to 4 bytes. We then evaluated the reliability of the system in terms of how many successful transmissions took place versus how many unsuccessful ones happened. A successful transmission is one which keeps the premises stated above. An unsuccessful one violates at least one of them. So, for example, programming an arbitrary throughput and running the programs, if the premises are kept on average every other transmission, then the reliability of this throughput equals 50%.

Figure 6 shows the throughput of the two prototypes. The data bases on about one million measurements per data point, the data for the FPGA implementation bases on the results from the cycle-accurate FPGA simulator and sample measurements. The x-axis displays the reliability of the traffic according to the definition above. The y-axis show the throughput in Mbits/s. The figures show that the FPGA implementation clearly outperforms the software implementation. The difference becomes even more significant as the reliability approaches 1. The software version also requires and additional safety margin for industrial cases. Looking at the other end of the spectrum, the software asymptotically approaches the upper limit as the reliability moves towards 0.

4.3. On Chip Resource Usage

The current implementation uses a XILINX Virtex 4 FX 12 chip, which provides one PPC 405 core and two Ethernet MACs on chip. The FPGA has 36 memory blocks, and the NCP currently uses 20. The CLB usage is moderate (30% of the FX12 chip) which leaves lots of space for the host processor system integration. The host processor uses another four memory blocks for the boot loader, and it starts the operating system from a flash card. The full system including FLASH card, NCP, VGA and keyboard/mouse driver on chip covers 75% of the CLBs on chip. The host operating system (in our case linux) is booted from the FLASH card.

4.4. Timing and Data Throughput

Since the Network Code program is time triggered (the future() instruction uses a time value for the parameter *dl*), correct timing is important and needs to be analyzed throughout the whole system.

The FPGA runs at 100 MHz. Every critical function is implemented as an IP core and has a well-known timing behavior. Although the execution time of some instructions depends on the length of the concerned variables, all this information is known at design time and timing properties can be statically checked beforehand.

Programs can operate at a (message) resolution of 100 kHz, therefore the current time quantum (minimal value) for the future() instruction is 10μs. Since we use a 100 MBit Ethernet connection, the quantum is more than the minimum transmission time of an Ethernet telegram, which is 6.8 μs for 64 bytes plus preamble and IFG that gives a throughput of 6MB/s. Note different payload sizes result in more or less throughput (see Figure 5). However, active networking components can introduce an additional delay that has to be considered. In our experiments we used a Cisco Catalyst 3500 which added approximately 25, 135, and 271μs for a transmission of a variable with the size of 4, 500, and 1 000 bytes.

4.5. Discussion

Going from Software to Programmable Hardware. Software systems rarely face resource limitations of the storage resource. If the developer faces such a limitation, the typical solution is to either move to a larger chip (e.g., in microcontroller systems) or to add more memory and disk storage to the computer. However, the developer cannot apply this solution to programmable hardware, especially FPGAs, because current production and available boards limit the available options. We therefore revisited each instruction and made a case again why this feature should be part of the system and should be present in the hardware solution. Among the features, which we cut out are message buffers for outgoing messages and multiple concurrent future() instructions. Both features were rarely used in the software prototype. As a consequence of the former, the create() and send() instructions can only use one send buffer. Therefore, one packet must be prepared after the other has been sent. The latter results in more complicated code, but does not reduce the functionality of the system.

In the software implementation, the developer can code arbitrary branch guards via C functions. In the hardware implementation, we now provide a predefined set of branching functions, but still leave the developer an option of extending the set with own functions synthesized onto the FPGA. These predefined branching conditions fall into three categories: value comparators, state comparators, and counter comparators. Value comparators compare two values in the dual RAM and branch, for instance, if the value A is greater than value B. State comparators allow the developer to branch depending on the internal status bits. These conditions include for example checks whether messages have been received in particular channels or whether the output buffer is filled. Finally, counter comparators provide convenience to the developer, because now the developer can set/reset and compare the counters inside the Network Code program without requiring a high-level application. For example, the developer can now easily encode that the program follows a particular branch every other round.

The FPGA implementation provides a decoupled processor for real-time communication. In the software prototype, the application and the communication were still tightly coupled, because they executed on the same processor. In the FPGA implementation, these two elements are disjoint and we require additional means for communicating between them. We therefore provide a signal() instruction in the hardware implementation to generate interrupts in the host processor. The application software in the host processor can listen to this interrupt and respond appropriately.

Lessons from Using Ethernet COTS vs FPGA. Our measurements show that software-based real-time communication frameworks in which the arbitration control is located inside the kernel or at a higher level can only be used for applications which require low throughput or relaxed timing constraints. For case studies, this implies that one should only consider applications with short run times, because a long run time will inevitably eventually cause violations in the slot structure and thus create errors. However, short run times inevitably cast doubt on whether the tested system actually works with industry-grade use cases, especially since programmable hardware is readily available. Network components such as switches further aggravate this and support our argument that real-time communication experiments conducted only with high-level software prototypes should be handled with care.

On the other hand, using programmable hardware for validating real-time communication frameworks bore more advantages than drastic throughput improvements. For example, the timing variance for each code instruction and action differs among workstations, because of differences among interrupt controllers, motherboards, and processors. The FPGA allows cycle-accurate simulation and offers similar delays on each board instance. Thus, our current and future experiments lead to precise, reproducible results. This increase in precision allows researchers to place more confidence in the results.

Programmable hardware also enabled us to implement our model more faithfully than software-based implementations. Again, this is partially due to the increase in determinism, but also due to the natural way of implementing concurrently executing structures. Concurrent tasks inside the communication framework can be implemented as parallel processes on the FPGA board, and they will truly concurrently execute. For example, if we want to extend the hardware implementation of the NCP to allow multiple concurrent threads via multiple future() instructions. We can achieve this easily by synthesizing multiple NCPs onto the FPGA that run in parallel.

Finally, hardware synthesis also requires careful thinking about the system model, functionality, and timing. Debugging is difficult and programming by trial and error is virtually impossible. This leads to a clean and well- documented implementation.

Verification Step Simplifies Software Requirements. Using verification on Network Code programs [7] significantly reduces the required functionality in the NCP. This is important, because Network Code provides a programmable framework and the developer can program own communication schedules. Since the developer cannot be trusted, the NCP would need to provide functionality for error detection and error recovery. However, we

can check programs for structural and behavioral errors and thus, we can substantially reduce the functionality for error detection/recovery and free these resources. For example, the NCP does not require checks on internal state corruption such as invalid program counters, invalid memory cell accesses, and incompatible data formats and type checking when receiving messages and storing the values in the variable space. This significantly contributes to the NCP's low footprint.

5. Conclusion

We have presented the Network Code Processor (NCP) which is a processor IP core for Network Code programs, and a co-processor for time-triggered protocols in general. The processor implements a superscalar architecture in which multiple instructions execute concurrently. We discussed the development of the NCP, specifically its concurrency controller and presented an example which clearly shows the benefits of the superscalar architecture. Measurements showed that the the NCP clearly surpasses the software implementation and moreover meets the design goal to provide a real-time–capable communication system comparable with standard Ethernet. Finally, we also captured our experiences during the development and our design rationals in the discussion section of this work.

References

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers—Principles, Techniques, and Tools*. World Student Series of Computer Science. Addison Wesley, 1986.

[2] 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)*, pages 122–131, Seoul, South Korea, Oct. 2006.

[3] M. Anand, S. Fischmeister, and I. Lee. Composition Techniques for Tree Communication Schedules. In *Proc. of the 19th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 235–246, Pisa, Italy, July 2007.

[4] R. Caponetto, L. lo Bello, and O. Mirabella. Fuzzy Traffic Smoothing: another step towards Statistical Real-Time Communication over Ethernet Networks. In *Proc. of the 1st International Workshop on Real-Time LANS in the Internet Age (RTLIA)*, 2002.

[5] R. Court. Real-time Ethernet. *Comput. Commun.*, 15(3):198–201, 1992.

[6] Ethernet Powerlink Standadisation Group (EPSG). *Ethernet Powerlink V2.0 – Communication Profile Specification*, version 0.1.0 edition, 2003.

[7] S. Fischmeister, O. Sokolsky, and I. Lee. A Verifiable Language for Programming Communication Schedules. *IEEE Transactions on Computers*, 56(11):1505–1519, Nov. 2007.

[8] S. Fischmeister and R. Trausmuth. A Programmable Arbitration Layer For Adaptive Real-Time Systems. In *Proc. of the Intl. Workshop on Adaptive and Reconfigurable Embedded Systems (APRES)*, pages 27–31, 2008.

[9] E. Group. Real-time Ethernet control automation technology (EtherCAT). IEC/PAS 62407, 2008.

[10] P. Ienne and R. Leupers. *Customizable Embedded Processors: Design Technologies and Applications*. Morgan Kaufmann, 1st edition, July 2006.

[11] M. Jacome, M. Jacome, and G. De Veciana. Design challenges for new application specific processors. *IEEE Design & Test of Computers*, 17(2):40–50, 2000.

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

[13] S. Kweon, K. Shin, and G. Workman. Achieving Real-Time Communication over Ethernet with Adaptive Traffic Smoothing. In *Proc. of the Sixth IEEE Real Time Technology and Applications Symposium (RTAS 2000)*, page 90, Washington, DC, USA, 2000. IEEE Computer Society.

[14] S.-K. Kweon and K. Shin. Statistical Real-Time Communication over Ethernet. *IEEE Trans. Parallel Distrib. Syst.*, 14(3):322–335, 2003.

[15] S. Leibson. *Designing SOCs with Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*. Elsevier Morgan Kaufmann Publishers Inc., 2006.

[16] M. Li, T. V. Achteren, E. Brockmeyer, and F. Catthoor. Statistical Performance Analysis and Estimation of Coarse Grain Parallel Multimedia Processing System. In *Proc. of the 12th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 277–288, Washington, DC, USA, 2006. IEEE Computer Society.

[17] J. Loeser and H. Haertig. Low-Latency Hard Real-Time Communication over Switched Ethernet. In *Proc. of the 16th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 13–22, Washington, DC, USA, 2004. IEEE Computer Society.

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

[19] K. K. M. Gries. *Building ASIPs; the MESCAL methodology*. Springer–Verlag, 2005.

[20] N. Malcolm and W. Zhao. The timed-token protocol for real-time communications. *Computer*, 27(1):35–41, 1994.

[21] P. Pedreiras, L. Almeida, and P. Gai. The FTT-Ethernet protocol: merging flexibility, timeliness and efficiency. In *Proc. of the 14th Euromicro Conference on Real-Time Systems*, pages 134 –142. IEEE Press, June 2002.

[22] 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 Transactions on Industrial Informatics*, 1(3):162–172, Aug. 2005.

[23] K. Shin and C.-J. Hou. Analytic evaluation of contention protocols used in distributed real-time systems. *Real-Time Syst.*, 9(1):69–107, 1995.

[24] K. Steinhammer. *Design of an FPGA-Based Time-Triggered Ethernet System*. PhD thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria, 2006.

[25] 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)*, pages 794–799, 3001 Leuven, Belgium, Belgium, 2006. European Design and Automation Association.

[26] C. Venkatramani and T. Chiueh. Design, Implementation, and Evaluation of a Software-based Real-Time Ethernet Protocol. In *SIGCOMM: Proc. of the conference on Applications, technologies, architectures, and protocols for computer communication*, pages 27–37, New York, NY, USA, 1995. ACM Press.