Hardware Acceleration for Conditional State-Based Communication Scheduling on Real-Time Ethernet

Sebastian Fischmeister, *Member, IEEE*, Robert Trausmuth, *Member, IEEE*, and Insup Lee, *Fellow, IEEE*

***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 many appliances and network components and represents a natural fit for real-time application; unfortunately, standard Ethernet provides no bounded communication delays.**

Conditional state-based communication schedules provide expressive means for specifying and executing with choice points, while staying verifiable. Such schedules implement an arbitration scheme and provide the developer with means to fit the arbitration scheme to the application demands instead of requiring the developer to tweak the application to fit a predefined scheme. An evaluation of this approach as software prototypes showed that jitter and execution overhead may diminish the gains.

This work successfully addresses this problem with a synthesized soft processor. We present results around the development of the soft processor, the design choices, and the measurements on throughput and robustness.

***Index Terms*—Networks, programmable hardware, real-time systems, time-division multiaccess.**

I. 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. Obviously, the system is only useful, if both—correct timing and correct values—are guaranteed.

Manuscript received October 30, 2008; revised March 07, 2009, May 22, 2009, and June 24, 2009. Current version published August 07, 2009. This work was supported in part by NSF CNS-0834524 NSF CNS-0721541, NSF CNS-0720703, NSF CNS-0720518, NSF CNS-0509327, AFOSR FA9550-07-1-0216, NSERC DG 357121-2008, and in part by ORF RE03-045. Paper no. TII-08-10-0135.

S. Fischmeister is with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON N2L 3G1, Canada (e-mail: sfischme@uwaterloo.ca).

R. Trausmuth is with University of Applied Sciences, Wiener Neustadt, Austria (e-mail: trausmuth@fhwn.ac.at).

I. Lee is with the Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA 19104-6389 USA (e-mail: lee@cis.upenn.edu).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TII.2009.2026642

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 or dropped packets in the network communication, which typically results in retransmissions. Since such behavior makes 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. Additionally, the developer must consider properties intrinsic to the protocol and arbitration scheme, and adapt the application to work with or around them.

Ethernet is a widely used network technology in the embedded systems industry besides field bus systems. The market provides many appliances and network components, therefore it is natural to try using Ethernet for real-time communication. Unfortunately, Ethernet's intrinsic nondeterminism caused by the collision detection and binary backoff mechanism for resolving contention make it hard to provide upper bounds for communication delays on this platform. Several 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 [1]–[3] 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 [4]–[7] for traffic shaping and traffic prediction or higher level communication frameworks [8]–[14] 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 [15] and is subject to high jitter.

State-based schedules based on automata [16] or more explicitly state chart like formalisms with conditional transitions [17], [18] represent recent development to improve scheduling of real-time systems. The Network Code language permits developers to express such conditional state-based communication schedules, and while the specification, analysis, and verification are already partially examined, the systems side of how to efficiently realize such schedules is not yet sufficiently explored as we will show in the following section.

A. Motivation

A conditional state-based communication schedule must maintain state information and has guarded transitions between state. This requires the system which executes such a schedule

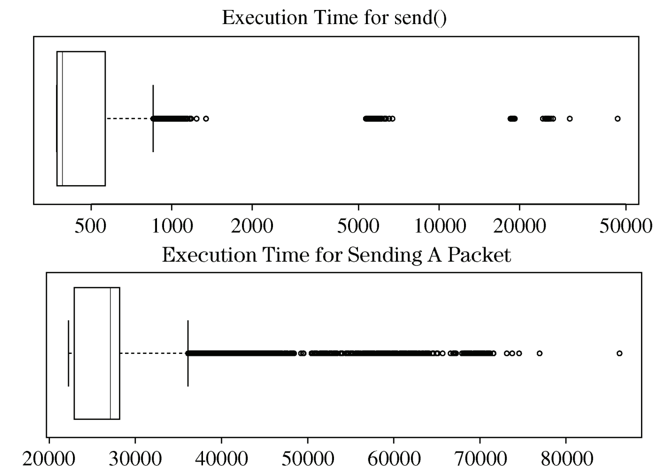


Fig. 1. Execution jitter in [ns].

to have memory and computational resources to evaluate the guards. We can implement such a framework in two ways: in software and in hardware. Commercially available and research systems [8], [10]–[13], [19]–[22] would allow us to implement it in software on top of it—this has already been done [18] and the driver resides in the network driver of a real-time Linux system using standard Ethernet. This provides good flexibility, because the software can easily be changed and extended to accommodate new features. However, although the code sits as close to the hardware as possible considering a full-blown operating system, the system still experiences high jitter which limits its applications in industrial settings. One could envision the same software prototype to use Powerlink Ethernet or EtherCat. However, this will still cause similar execution-time jitter for instructions as with the used software prototype.

Fig. 1 shows two box plots for execution jitter of instructions. In a box-and-whisker plot [23], the central rectangle includes the second and third quartile giving an idea of the distribution's slope. The median divides this box. The two markers to the left and the right of the box mark the smallest and largest values that are no outliers (1.5 times the distance of the interquartile range from the median). All outliers are marked with the symbol "o".

The data in Fig. 1 provides evidence that implementing the framework on top of standard components introduces high jitter in a system—data comes from tests using the software prototype [18]. Let us consider the instruction send() which enqueues a message in the output queue. The statistical mode of this instruction is 372 ns. If we consider the 99th percentile, then the execution time lies between with 371–733 ns. If we increase the percentile and thus increase the timing reliability of our system (a more correct 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 19.090 μs (26 times the original value). Although parts of the software might be optimized by correlating delays and dependencies using for example statistical models [24], the high variance still remains.

This paper describes the Network Code Processor (NCP) which is a hardware implementation of the Network Code framework which enables conditional state-based communication schedules. Specifically to the NCP, we discuss the following items.

1) We present our hardware model and the analysis, which make the framework run at comparable speeds to raw 100 Mbit/s Ethernet. To improve performance: (a) we used an application-specific processor [25], [26] with a superscalar design in which multiple instructs are autonomous execution units, and (b) we used techniques to discover and subsequently exploit concurrent execution as much as possible.
2) We provide measurements to demonstrate that we successfully met our goals to increase throughput and reduce jitter. We also compare the hardware prototype with the software prototype side by side and show the effect of the execution-time jitter.
3) We discuss our lessons learnt when going from the software prototype to the hardware prototype and how the previous work helped us to reduce the space footprint on the FPGA.
4) We show how we provide support for legacy and non real-time applications in our hardware implementation. Specifically, we show how we integrate the standard OS network driver interface and permit running legacy drivers without changes.

The reminder of this paper is organized as follows: Section II introduces the system model, provides an overview of the instruction semantics, and also explains the hardware model. These three elements form the basis for the instruction dependency analysis which we present in Section III. Section IV explains our instruction parallelism control unit used to control the individual execution blocks. Section V shows the measurement results for our system and our comparison with the software prototype. In the discussion part (Section VI), we report our experiences from building the system and discuss the work. Finally, we close this paper with our conclusions in Section VII.

II. System, Semantics, Hardware Model

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 [18], analyzeable [27], and composable [28]. Furthermore, Network Code and its runtime can be seen as a programmable communication layer [29].

A. System Model

Time-division multiple access (TDMA) provides a time-based arbitration method to provide collision-free access to network nodes. Time is partitioned into slices called slots with a duration referred to as slot length. Each network node is allowed to communicate in specific slots. The node-to-slot assignment varies among protocols from dynamic to static. A communication round usually refers to a basic pattern that is then repeated endlessly as the system executes. Nodes must not communicate outside their slots, therefore it is of utmost importance to guarantee that each node's communication terminates prior to the slot boundary.

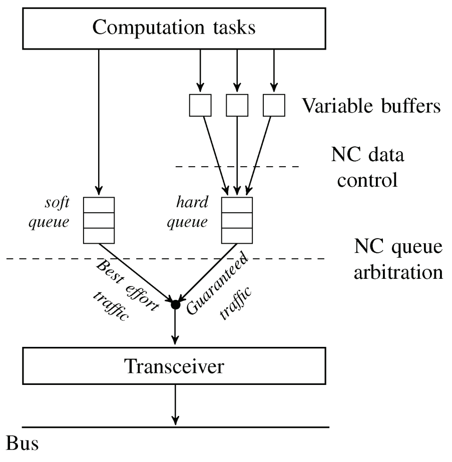


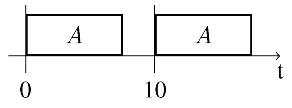
Fig. 2. Overview of the queues and controls.

```
1 L0: create(msg_a, A)
      send(1, msg_a, _)
3     future(10, L0)
      halt()
```

(a)

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

(b)



(c)

Fig. 3. Using guaranteed traffic class communication to transmit variable A. (a) Sender. (b) Receiver. (c) Visual schedule.

Network Code provides two distinct types of quality-of-service (QoS): best effort and guaranteed. Messages sent using the *best effort* quality class do not have a 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 [18] and analysis [27] 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 10 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.

Fig. 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 details beyond this summary, please see the system specification for the initial work on the verification mechanism [18] or the language specification [30].

Let us walk through the system using the best-effort traffic class: the computation tasks implement the application logic and transmit values to other network nodes. The task enqueues this message in the *soft queue.* Whenever the transceiver is placed into soft mode, it will take messages from this soft queue and transmit them. On the receiving node, the transceiver automatically receives such messages and places them into the soft queue for incoming messages. The computation task on the receiving node can dequeue the message and process it.

Let us walk through the system using the guaranteed traffic class: the computation task writes a new value into a dedicated variable buffer; for example the variable temperature. The Network Code program specifies the time when this value will be read from the buffer, turned into a message, and transmitted to receiving nodes via the hard queue. The Network Code program at the receiving node knows the message containing the temperature value will be transmitted and receives it into the local variable buffer. The computation task on the receiving side can read the value from the buffer and process it. In contrast to the best-effort traffic, in the guaranteed traffic everything must be specified offline: the schedule, the buffers, and the timing.

The Network Code language consists of just a few core instructions which control timing, data flow, control flow, and error handling. Derived instructions are like macros that can be represented by core instructions.

The create() instruction creates a message from a variable buffer. The send() instruction issues a transmission of a message on the network. The receive() instruction receives an incoming message into a variable buffer. The if() instruction implements a conditional jump where the branching condition can use values in the variable buffer, history, or the current state of the schedule. The sync() instruction signals a new communication round and synchronizes all nodes. Communication rounds can have different lengths in state-based schedules, because depending on the conditional branching during the round it sometimes might take a branch with a longer or shorter duration. The instruction mode() controls the mode of operation of the runtime 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 system initialization. The instructions future() and halt() implement temporal control through the use of timers which may resume execution at particular program labels.

In the following, we provide two brief examples to demonstrate how Network Code works. Most of the 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 [18].

As an example for virtual circuit-switched communication consider the following programs shown in Fig. 3(a) and (b). Note that for sake of simplicity, we assume that both nodes start simultaneously and there is no clock skew; also wait() is a composite instruction used for instructive purposes and not atomic.

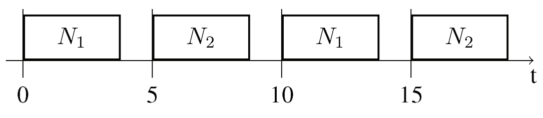
Fig. 3(c) shows the schedule that these programs represent. Note that at time 10, the schedule repeats. The sender first creates a packet from variable A using the alias msg_a. Then, it

```
L0: mode(soft)
    wait(4)
    mode(hard)
    future(11, L0)
    halt()
```

(a)

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

(b)



(c)

Fig. 4. Using best-effort traffic class communication data from the soft queue from N_1 and N_2. (a) Node N_1. (b) Node N_2. (c) Visual schedule.

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 it from channel 1 into the local variable A every ten time units.

As an example for packet-oriented communication, consider the programs in Fig. 4(a) and (b) using the same assumptions as before.

The system guards access to the network through temporal isolation. Fig. 4(c) shows the schedule that these two programs represent. Note that the schedule repeats at time 10. 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 Fig. 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.

B. Operational Semantics Excerpt

The small-step operational semantics of the Network Code language are well defined [30]. In the following, we provide a small excerpt of a few instructions to illustrate how they work and how we used them to detect dependencies among instructions (see Section III).

1) Overview: A program $prgm$ consists of a sequence of instructions where each instruction is stored in a unique address location. The set A contains all valid addresses; a_0 denotes the initial address. Addresses are totally ordered.

- **Node State**: A node state is the 4-tuple

$$s_n = \langle prgm, s_{ncm}, s_t, sto\rangle$$

  consisting of a Network Code program $prgm$, the Network Code Machine state s_{ncm}, time state s_t, and a storage sto.
- **Message**: A *message* is the 3-tuple

$$m = \langle ch, lifetime, cont\rangle$$

  with a channel ch, a relative time span $lifetime$, and a message content $cont$.
- **Storage**: A storage sto contains bindings of identifiers to values. It can be considered as containing tuples $\langle id, n\rangle$. The proposition $id \in sto$ holds, if $\exists\langle x, n\rangle \in sto : x = id$.
- **Network Code Machine State**: The state $s_{ncm} = \langle a, T, M_c, M_{\text{out}}, M_{\text{in}}, mod, t_w\rangle$ consists of
  — a program counter $a \in A \cup \{\perp\}$, where the symbol $\perp$ indicates termination;
  — a set of timed triggers T;
  — a set of created messages M_c;
  — a set of output messages M_{out};
  — a set of input messages M_{in};
  — an operational mode $mod \in \{hard, soft\}$;
  — a time stamp of the last wake up t_w.

2) Auxiliary Operations: $instr(a)$ represents the operation code for the instruction at location a. $next(a)$ represents the successor address of address a. $applySto(sto, id)$ returns the value associated with id in the storage sto. More auxiliary operations are listed in [30].

3) Sample Instructions: The instruction create(msgid, loc) creates a message from a memory location. The parameter msgid identifies the message to be created. The parameter loc identifies the memory location from which the message's values will be taken

$$\begin{aligned}&\langle create(msgid, loc), s_n\rangle \rightarrow \langle skip, s'_n\rangle\\ &\text{with}\quad M'_c = M_c \cup \langle msgid, \_, applySto(sto, loc)\rangle\,. \quad (1)\end{aligned}$$

The semantics of the create instruction is shown in (1). The instruction specifies a state change from s_n to s'_n and (1) specifies how s'_n differs from s_n. For the subsequent instructions, we will use a similar notation.

The instruction send(ch, msgid, lifetime) enqueues a message in the hard output queue. The parameter ch specifies the channel on which messages are to be sent and received. The parameter msgid identifies the message to be communicated. The parameter lifetime specifies the message's relative lifetime. The lifetime is the time span during which the message's packets are alive and valid. After expiry of that value, the message can be cleared from the input buffers. In the normal case, the lifetime of a message is the TDMA slot length. The send instruction neither needs a parameter for message length nor its deadline, because we check offline whether these parameters are satisfied and thus at runtime they serve no purpose. We refer the interested reader to [18] and [30] for further details how message lengths and transit times are specified. Note, that in (2), $m'.cont = m.cont$ with $m \in M_c$ and $m.msgid = msgid$

$$\begin{aligned}&\langle send(ch, msgid, lifetime), s_n\rangle \rightarrow \langle skip, s'_n\rangle\\ &\text{with}\quad M'_{\text{out}} = M_{\text{out}} \cup (m' = \langle ch, lifetime, cont\rangle)\,. \quad (2)\end{aligned}$$

The instruction mode(m)

$$\begin{aligned}&\langle mode(newmode), s_n\rangle \rightarrow \langle skip, s'_n\rangle\\ &\text{with}\quad mod' = newmode. \quad (3)\end{aligned}$$

The instruction future(wakeup, jmp) registers another timed trigger to wake up upon at a specific code location

$$\langle future(wakeup, jmp), s_n\rangle \rightarrow \langle nil, s'_n\rangle$$
$$T' = T \cup \langle wakeup, jmp\rangle. \quad (4)$$

The examples are not to completely describe the semantics, but just to illustrate how we have specified the small step operational semantics of the language elements. A complete description of all instruction is given in the language specification [30]. As explained before, we will use these semantics to investigate potential for parallelism.

C. Hardware Model

Instruction level parallelism is also limited by the underlying hardware. In this section, we describe the hardware and analyze dependencies among instructions.

1) XILINX Virtex 4: We synthesize the NCP on an FPGA. Our choice of hardware is a XILINX Virtex 4 FX12 FPGA. Its main features are a PowerPC core and two Ethernet MAC cores on chip. The PowerPC uses on chip buses according to the IBM CoreConnect specification, namely the Processor Local Bus and the On-Chip Peripheral Bus. All cores used by the PowerPC connect to one of those buses.

The application-specific instruction set processor (ASIP) was designed, optimized and implemented by hand. Although there are several tools available for doing this, namely, MESCAL [31] or commercially available packages like the Tensilica cores [32], we chose this approach to complete control the synthesized hardware. Future research will show whether we can get similar results by using such tools.

The FPGA comprises 36 memory blocks which can be used as dual port random access memory (RAM) or as first-in–first-out (FIFO). We mainly use dual port memory blocks to decouple the NCP and the PowerPC part of the implementation. The chip design is placed into 5472 logic slices which are arranged in a 64 $\times$ 24 matrix.

The implementation target platform is a XILINX ML 403 board with one V4 FX12 chip on it.

2) Core Building Blocks: Fig. 5 shows the functional units and their connection to the FPGA infrastructure. The OnChip RAM contains the computation tasks. The PowerPC runs an operating system and executes the tasks. The PowerPC communicates with the NCP via the on-chip peripheral bus (OPB).

The NCP implements instructions in its own, independent execution units. On the FPGA such an execution unit is a microcode block which is accessible via a well defined interface. In Fig. 5, all such microcode blocks are combined in the "NCP command blocks" element.

To control the independent execution units, we also synthesize the control block "NCP controller." This control block triggers the execution units and manages the instruction level parallelism. We describe the precise rules for concurrent execution (triggering) of instruction in Section IV. The NCP controller also manages buffers. The variable buffers for guaranteed traffic are stored in the dual port RAM inside FPGA (i.e., the Config ROM and the Variable RAM). The Variable RAM contains the

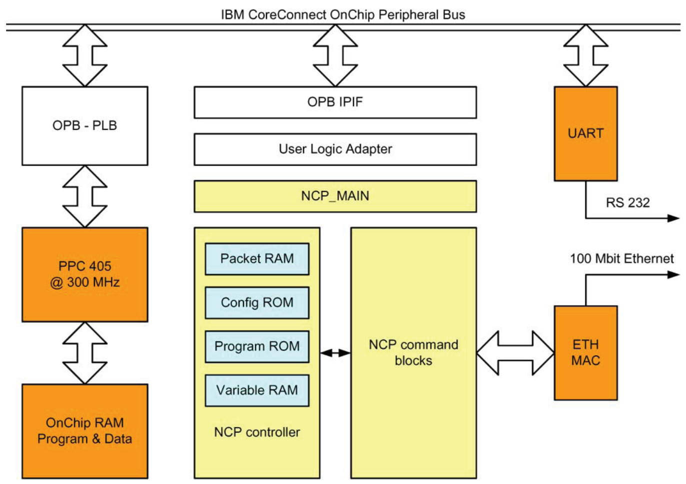


Fig. 5. NCP implementation block diagram.

current values set by the computation tasks. The Config ROM contains the memory layout of the Variable RAM. The Packet RAM contains the next packet to be transmitted in best-effort mode. The dual port RAM is accessible via the on chip peripheral bus by the OPB IPIF and the user logic adapter. Accesses to the memory areas such as the Variable RAM are arbitrated via a synthesized bus in the FPGA area.

3) Interaction and Data Paths: The core building blocks share communication busses and memory resources. We therefore must clearly describe the block interaction and data flows, so we can later analyze data dependencies.

The computation tasks run at the PowerPC. Each task can access the variable buffers in the dual port RAM and the control interface to the NCP. The buffers themselves are memory mapped to a specific address range and the tasks themselves must coordinate access restrictions to these addresses on the computation side. The dual-port access is delayed for synchronization purposes when the NCP is about to create a telegram or receive data.

Each Network Code instruction is encoded in one 32 bit word. The Network Code program resides in the Program ROM area which can be accessed by the PowerPC (for setting up the program) as well as the NCP. The processor uses a classical fetch-decode-execute method for instruction processing. However, branch instructions are preloaded and available for decode whenever needed. The instruction loader uses a preload pipeline and provides the next instruction right after the decoding of the previous one. In case of possible parallel execution, the execution stage of one instruction triggers decoding of the next. The controller takes care of all necessary interlocking mechanisms as described in Section IV-A.

One key requirement is the integration/reuse of legacy software (see one question in Section I-A). We wanted to leave the OS interface to the network controller unchanged. Therefore, the standard network driver in the OS must have access and feedback from the MAC chip on the board. To provide this, the soft queue uses an interface compatible with the XILINX emacLite IP core specification. In case the NCP is set to mode *hard*, the network card appears to be busy to the OS driver. Whenever the NCP switches to mode $soft$, the network card is transparently accessible to the OS driver until the next mode instruction switches back to the $hard$ mode.

The network interface provides two memory areas holding one send and one receive packet. The OS driver sets up the packet to be sent and then signals to the MAC controller to transmit the packet. In the soft mode, the end of transmissions and packet receptions are signalled to the OS driver either by the interrupt or by setting a status bit. The OS driver can transparently access the received packet during the soft mode.

The hard queue is active whenever the NCP is running in hard mode. Messages have a well-defined life cycle and we can map this life cycle to a sequence of instructions to send a packet in hard mode. First, the packet has to be prepared. This is done by the create() instruction. Data is copied into the send FIFO. The send() instruction then assembles a valid Ethernet packet and puts it into the output FIFO which triggers the transmission by the MAC block. The reception of packets is done by an asynchronous receive block which checks the telegram type and unpacks the data of the telegram. The receive() instruction reads data from one of the channel FIFOs and copies the data into the specified variable.

TABLE I
DEPENDENCY SUMMARY

Type	**Dependency**
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	$halt \overset{d}{\longleftrightarrow} if$
	$sync(c, \_) \overset{d}{\longleftrightarrow} if(StatusTest, \_)$
	$receive \overset{d}{\longleftrightarrow} if(G_3, \_)$
	$receive \overset{d}{\longleftrightarrow} create$
	$create \overset{d}{\longleftrightarrow} send$
	$create \overset{d}{\longleftrightarrow} if(\mathit{SendBufferEmpty}, \_)$
	$destroy \overset{d}{\longleftrightarrow} send$
	$destroy \overset{d}{\longleftrightarrow} if(\mathit{SendBufferEmpty}, \_)$
Mode	$mode \overset{m}{\longleftrightarrow} \{sync, receive, create, destroy, send\}$
	$sync(c, \_) \xrightarrow{m} halt$

III. DEPENDENCY ANALYSIS

To maximize the level of concurrency, we must analyze dependencies among instructions. With the full set of dependencies we can construct the control block that controls instruction level parallelism. Without the full analysis, we risk unintended behavior and possible faults during execution which can result in system failures.

A. Instruction Dependency

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

Control Dependency. Given two successive instructions, the second one is 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 nonobvious 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 dependency. Two successive instructions are data dependent, if they access or modify the same resource [33]. 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 dependency. 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 I shows a summary of the dependencies among instructions based on the operational semantics. The symbols $\xrightarrow{c}$, $\xrightarrow{d}$ and $\xrightarrow{m}$ denote a control, data, and mode dependency, respectively. The symbol $a \overset{\star}{\leftrightarrow} b$ denotes a dependency $a \overset{\star}{\rightarrow} b$ and $b \overset{\star}{\rightarrow} a$. The set G_1 consists of all guards except *AlwaysFalse*, the set G_2 contains all guards except *AlwaysTrue*, and set $G_3 := \{\mathrm{TestVar}, \mathrm{GreaterVarVar}, \mathrm{CompareVarVar}, \mathrm{LessVarVar}\}$.

Examples: The dependency $if(G_1, jmp) \xrightarrow{c} (instr(jmp) \setminus \{nop\})$ represents a typical control dependency. Instructions immediately following an if() instruction cannot be executed concurrently with the if(), because it depends on the evaluation of the conditional statement which branch the NCP will follow. The dependency $sync \xrightarrow{c} \{send, receive, halt, mode, if\}$ shows a dependency between the sync() instructions and a number of other instructions. These instructions cannot be concurrently executed with the sync() instruction, because they

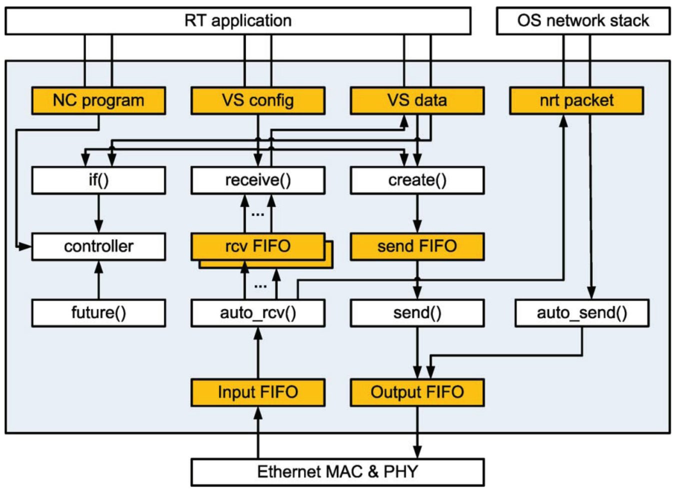


Fig. 6. Block diagram of the NCP.

TABLE II
DEPENDENCIES ORIGINATING FROM THE HARDWARE

Type	Dependency
Hardware	$create \overset{h}{\leftrightarrow} receive$
	$create \overset{h}{\leftrightarrow} if(G_3,\_)$
	$receive \overset{h}{\leftrightarrow} if(G_3,\_)$
	$create \overset{h}{\leftrightarrow} send$
	$send \overset{h}{\leftrightarrow} sync(m,\_)$

result in user-visible actions and must wait for the nodes to finish their synchronization through the sync() instruction.

The dependency $create \overset{d}{\leftrightarrow} send$ represents a typical data dependency between two instructions. The instruction send() uses the message which instruction create() builds and thus must wait for it to finish. The dependency $halt \overset{d}{\leftrightarrow} if$ expresses the data dependency between the two instructions halt() and if(). Both instructions manipulate the program counter and thus cannot execute concurrently.

The dependency $sync(c,\_) \overset{m}{\rightarrow} halt$ is necessary so the halt() instruction does disable the NCP before the sync() operation completed.

B. Dependencies From Hardware

Fig. 6 shows the NCP implementation. Each Network Code instruction (with the exception of the instruction halt()) is encapsulated in a microcode block. The architecture of the NCP follows ideas drawn from the original MIPS architecture [34]. The blocks communicate with the controller using a simple two way handshake protocol.

If we investigate Figs. 5 and 6, we can identify additional dependencies in our system originating from the hardware. For example, at most one execution block may access the shared memory area.

Table II shows the resulting dependencies for our underlying hardware. The instructions create(), receive() and if() access the variable space via a memory bus, so they can only execute one after the other. The instruction if() blocks the memory bus only for variable comparison operations, therefore guards not included in set G_3 as specified in Section III-A can be executed in parallel with instruction create() or receive(). The instructions send() and sync(m, _) both use the output FIFO, thus they can only be executed sequentially. Since the active sync() is a send instruction sending a special telegram, it can be handled by the send block, too. The passive sync() instruction is implemented directly in the controller which handles also the sync timeout. When the auto_rcv block receives a sync telegram, it sends a hardware signal to the controller.

IV. INSTRUCTION PARALLELISM CONTROL

The NCP controller manages concurrent execution of microcode blocks based on dependencies among instructions. In the previous sections, we listed the individual dependencies. In this section, we put them together and optimized the system architecture to reduce the number of dependencies.

A. 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 I, these two instructions must be executed sequentially. However, as they occur frequently, we optimized 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() fills in data, because the internal memory bus is 32 bits wide, whereas the MAC interface only supports 8 bits.

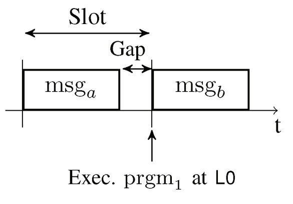
Table III shows the summary of all dependencies for the NCP after optimizations. The meaning of the characters in the table are "w" for wait until finished, "c" for continue with next instruction and "b" wait until the memory bus is available. The table is read the following way: given two sequential 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 III provides a c.

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 III permits concurrent execution, the controller

TABLE III
SUMMARY OF FINAL INSTRUCTION DEPENDENCIES

	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



(a)

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

(b)

Fig. 7. Average case of receiving and transmitting a message. (a) Visual structure. (b) Program $rmprgm_1$.

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 (executing a mode instruction), the also locking condition ensures that there are no packets in transit on the network.

B. Example

Let us consider an illustrative example to show the benefit of our selected architecture. Fig. 7(b) shows one of the snippets representing the average case of a network node receiving a message and transmitting a message. Fig. 7(a) shows how this program fits into the slot structure. The node executing this program first creates a message containing variable B which is then transmitted as message msg_b using channel 1. It also receives a message msg_a from the previous slot and stores its content in variable A.

Fig. 7(b) must be executed within 10 μs, because the instruction future() specifies a delay of 1 time unit which, in our implementation, equals 10 μs. One cycle takes 10 ns as the FPGA runs with a clock speed of 100 MHz. The execution time of an instruction is how long in terms of cycles the block requires to complete its operation. 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 (i.e., 512 bytes), 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 10 μs accommodates exactly 1000 cycles, this program cannot be executed sequentially.

This program executes fast enough to meet the deadline on our architecture with the instruction dependencies as specified in Table III. 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.

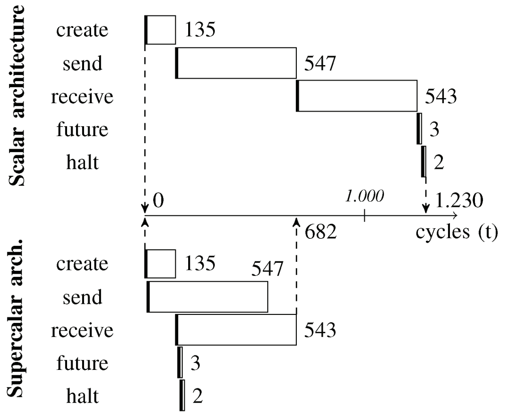


Fig. 8. Scheduling of the example program shown in Fig. 7(b).

Fig. 8 shows the execution trace as a Gantt chart of the NCP for executing Fig. 7(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.

V. 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.

A. Throughput of FPGA Solution

The execution speed of the create(), send() and receive() instructions grows linearly with the size of the data to be transmitted. This 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 based on cycle-accurate information resulting in single-cycle precision, because the hardware is free from jittery influences such as interrupts, cache misses, and page faults. Fig. 9 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/s. Note that the data throughput differs from the actual network utilization: 1) Ethernet messages include a header which introduces overhead and 2) messages have a specific minimum size, so padding must be added until 64 bytes and incurs overhead.

To calculate the throughput of the FPGA implementation, we can use (5) and (6). t_p specifies the computation time of the

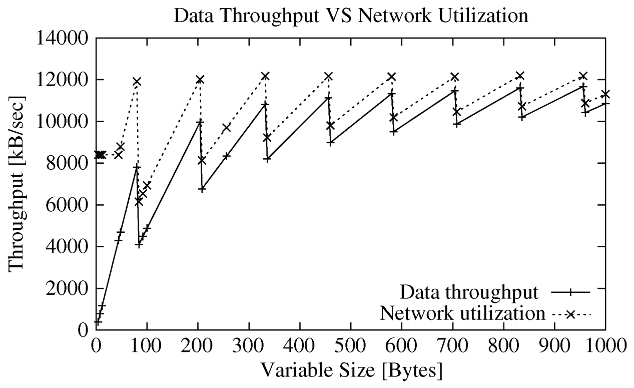


Fig. 9. Throughput of the FPGA implementation.

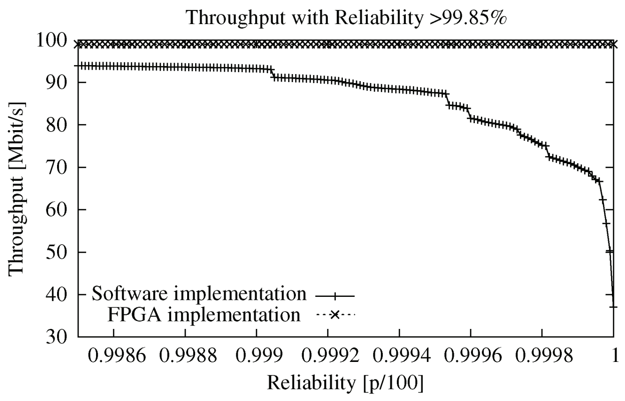


Fig. 10. Throughput of the two prototypes.

NCP, and t_s is the time required by the MAC layer to transmit a message. 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 (signaling of 8, frame accounting for 18, the body with a minimum of 46 bytes, and 12 bytes of transmission time as gap between subsequent slots) times the transmission duration of 80 ns per byte in the MAC layer.

We assume B in bytes

$$t_p(B) = \left(8 + \frac{B}{4} + 5\right) * 0.010\ [\mu\text{s}] \tag{5}$$

$$t_s(B) = (8 + 18 + \max(B, 46) + 12) * 0.08 \quad [\mu\text{s}] \tag{6}$$

Using t_p and t_s we can now compute how many ticks it takes to execute the program and transmit the data. A tick, called t_{tick} is the time duration of future(1, _)

$$\text{ticks} = \left\lceil \frac{t_p + t_s}{t_{\text{tick}}} \right\rceil \tag{7}$$

$$t_{tx} = \text{ticks} * t_{\text{tick}} \quad [\mu\text{s}]. \tag{8}$$

We now calculate how often we can fit this time into 1 s, and then multiply this with the transmitted kilobytes per variable and receive the throughput in [kB/s]

$$f = \frac{1}{t_{tx}} * 10e^5 \quad [\text{Hz}] \tag{9}$$

$$tp = \frac{B}{1,024} * f \quad [\text{kB/s}]. \tag{10}$$

Thus, the throughput $tp(B, t_{\text{tick}})$ of a specific variable size B in bytes and a system tick length of t_{tick} is defined as

$$tp(B, t_{\text{tick}}) = \left(\left\lceil \frac{t_p + t_s}{t_{\text{tick}}} \right\rceil * t_{\text{tick}}\right)^{-1} * \frac{B}{1,024} \quad [\text{kB/s}] \tag{11}$$

Fig. 9 shows the result of (11).

B. Software vs FPGA

As mentioned in the introduction the options to implement a framework for state-based communication scheduling is either on top of an existing communication standard in hardware or in software. We use the presented FPGA core as the hardware implementation and compare it to an implemented software prototype [18]. The selected software prototype used a kernel extension and ran on top of Ethernet.

For the evaluation purposes, we use the same ping-pong program as mentioned before. The software-based system ran as a kernel module of RTLinuxPro 2.2 on an Intel Pentium 4 with 1.5 GHz, 512 MB RAM, and a 3c905C-TX/TX-M [Tornado] (rev 78) with exclusive interrupt access. The hardware system ran on a Xilinx ML403 board. The core of the quantitative evaluation is now to identify that maximum throughput while still obeying the following premises.

1) The sending node must only communicate during its slot, so the ith communication must take place in the time slot $[i \cdot \text{step}, (i+1) \cdot \text{step})$.
2) The input queue must not overflow. The receiver must be fast enough to process the input queue as new messages 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%.

Fig. 10 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 Mb/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.

TABLE IV
SWITCH LATENCY FOR ACTIVE NETWORK COMPONENTS IN [μs]

	Latency 4B var.	Latency 500B var.	Latency 1000B var.
Cisco Catalyst 2950 (LAN Switch)	17	51	99
Cisco Catalyst 3500 (LAN Switch)	25	135	271
Surecom EP-808X-R (Mini Switch)	33	130	270

C. 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 covers 75% of the CLBs on chip. The host operating system (in our case linux) is booted from the FLASH card.

D. 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—a program that specifies used gates and their connections inside the FPGA—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 10 μs. Since we use a 100 MBit Ethernet connection, the quantum is more than the minimum transmission time of an Ethernet message, which is 6.8 μs for 64 bytes plus preamble and interframe gap (IFG) that gives a throughput of 6 MB/s. Note different payload sizes result in more or less throughput (see Fig. 9).

However, active networking components can introduce an additional delay that has to be considered. In one of our experiments, we tested different switches for the introduced delay. Table IV presents the data for all three used switches and shows that the speed varies considerably among brands and models.

E. Robustness

For robustness tests we set up a simple star-form network with two NCP systems connected through a switch. The two NCP systems exchange data (ping-pong) with a communication round of 600 μs. We also connected two workstations to inject rogue traffic into the system. The NCP systems were equipped with counters and indicator flags to measure the correct functioning of the system. The network switch used a store-and-forward principle which introduced some latency in the network communication (see Table IV). The store-and-forward architecture also cause the switch to drop messages. When we present the results and refer to lost messages, these messages have either been corrupted during the transmission—the checksum of the packet did not equal the transmitted checksum—or the switch has dropped the packet.

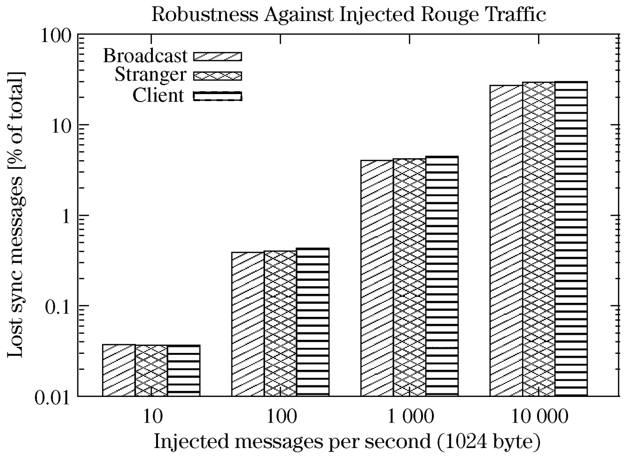


Fig. 11. Robustness test results.

The tests covered: (a) broadcast packets; (b) packets addressed to nonpresent MAC addresses; and (c) packets addressed network clients. We ran these tests multiple times with different injection rates (10, 100, 1.000, and 10.000 packets per second) with in different packet sizes (64, 256, and 1024 bytes) for about 30 s per test. Fig. 11 summarizes the results of the robustness test. The x axis shows the test of 1024 byte packets at different injection rates. The y axis shows the percentage of lost sync messages (indicating a new communication round) relative to the total number of sync messages sent during the test. Note that the y axis uses a logarithmic scale. In these tests, we made the following observations: (a) packets with length of 64 and 256 bytes never affected the system with any given injection rate; (b) the amount of lost packets correlates linearly with the injection rate; and (c) there is no significant difference between the three tested scenarios. The second observation is visible in Fig. 11; as the inject rate increases with an order of magnitude, the packet loss rate increases by an order of magnitude. Fig. 11 also shows the measurements for all three scenarios are about the same.

VI. DISCUSSION

A. Going From Software to Programmable Hardware

General software systems rarely face resource limitations of the storage resource. Even if the developer faces such storage limitations, the typical solution is to either move to a larger chip with more capacity (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 we cut out are message buffers for outgoing messages, and we limit multiple concurrent future() instructions to at most four. 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.

In the software implementation, the developer can code arbitrary branch guards via C functions and execute them on the main processor. The hardware implementation provides no general purpose processor to execute these guards. To overcome this limitation, we analyzed existing programs and guards and now provide a predefined set of frequently used branching functions such as tests of variables and tests of messages and queues. However, the developer also has the 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.

B. Lessons From Using Ethernet COTS Versus 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 create errors as it communicates across its slot boundary. 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.

C. Verification Step Simplifies Software Requirements

As can be easily seen from the examples in Section II-A, such Network Code programs may not necessarily always behave well together. Simply consider one program always transmitting packets and the result will be collisions, scrambled data, and nondeterministic behavior. Finding such bugs in the programs becomes more difficult as Network Code includes the flow control instruction if(), which implements a conditional branch. For this reason, we developed a verification framework [18] that allows checking properties of Network Code programs.

The experience that we got is that by relying on running verified programs. we can significantly reduce the required functionality in the NCP. Without this, the NCP would need to provide functionality for error detection and error recovery. For example, the NCP does not require checks on internal state corruption such as invalid program counters, invalid memory cell accesses, invalid jump locations, tight loops locking up the NCP, 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.

D. Comparison With Commercially Available Systems

Several industrial and research systems enable real-time communication on Ethernet; most notably of the commercial systems are Powerlink Ethernet [13], PROFINET [22], SERCOS III [21], VARAN [19], Modbus [20], and EtherCAT [12].

Each of these systems has a different set of goals in mind, but they try to maximize throughput by for example modifying the Ethernet header while providing bounded communication delays. Some of them also permit transmitting non real-time traffic in a dynamic, optional phase of the communication round. Our approach with conditional state-based schedules differs from what these products and also other research prototypes offer: throughput optimization comes from the application layer by providing a flexible communication framework that can adapt to the application needs. We have shown this in previous work [18] and briefly paraphrase the example here.

Consider a system with one input and temporal triple modular redundancy (TMR) for the input to mask one fault. The traditional setup is that the system uses three sensors to sample that single input, all three measurements get transmitted to a voting controller who then performs a majority vote to determine the final value. A stateless communication schedule without conditions requires three slots per communication round. A conditional state-based schedule can perform a preliminary voting after receiving two samples, and if the voting is already decisive—i.e., the first two slots contain the same value within a specified error bound—then the third slot will be used for other purposes or a new communication round starts immediately. Depending on the fault frequency and the speed of the voting algorithm, this can save up to one third of the bandwidth. However, this assumes that the choices can be made faster than transmission time of additional data. We showed that this is feasible for 100 Mb/s Ethernet in this work.

Another advantage of the presented communication system over commercially available and most research systems is that provides flexibility—conditional branching—but stays verifiable [18]. This is important for safety-critical applications that require evidence-based certification.

VII. Conclusion

In this work, we addressed the problem jitter and long execution times of guards can diminish the benefits of state-based communication schedules. Our approach was to synthesize a soft processor called NCP which is a logic core (intellectual property core) for Network Code programs, and a coprocessor 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.

The measurements showed that high throughput is feasible for systems with state-based communication schedules. More specifically, the NCP meets the design goal to provide a real-time—capable communication system comparable in throughput with standard Ethernet. Finally, we also elaborated on our lessons learnt during the development and described our design choices in the discussion section of this work.

We have already used the NCP in a case study to build a closed-loop medical control system. The selected clinical environment requires support for a dynamic system in which medical devices may be added or removed on the fly. The environment also eventually requires system certification. We found that being able to express and verify the communication behavior of the system in Network Code before testing and deployment was very helpful in building such a system. The demonstration has been showcased at the annual event of the Healthcare Information and Management Systems Society (HIMSS) in 2009. Although related work [35] shows advantages of conditional state-based schedules, we still consider evaluating its benefits on the development cycle as future work.

Acknowledgment

We would like to thank K. Perry for helping with the robustness tests and the reviewers of their excellent feedback on the earlier revisions of this article.

References

[1] R. Court, "Real-time Ethernet," *Comput. Commun.*, vol. 15, no. 3, pp. 198–201, 1992.

[2] N. Malcolm and W. Zhao, "The timed-token protocol for real-time communications," *Computer*, vol. 27, no. 1, pp. 35–41, 1994.

[3] K. Shin and C.-J. Hou, "Analytic evaluation of contention protocols used in distributed real-time systems," *Real-Time Syst.*, vol. 9, no. 1, pp. 69–107, 1995.

[4] S. Kweon, K. Shin, and G. Workman, "Achieving real-time communication over Ethernet with adaptive traffic smoothing," in *Proc. 6th IEEE Real Time Technol. Appl. Symp. (RTAS 2000)*, Washington, DC, 2000, p. 90.

[5] R. Caponetto, L. lo Bello, and O. Mirabella, "Fuzzy traffic smoothing: Another step towards statistical real-time communication over Ethernet networks," in *Proc. 1st Int. Workshop on Real-Time LANS Internet Age (RTLIA)*, 2002, pp. 33–36.

[6] S.-K. Kweon and K. Shin, "Statistical real-time communication over Ethernet," *IEEE Trans. Parallel Distrib. Syst.*, vol. 14, no. 3, pp. 322–335, 2003.

[7] J. Loeser and H. Haertig, "Low-latency hard real-time communication over switched Ethernet," in *Proc. 16th Euromicro Conf. Real-Time Syst. (ECRTS)*, Washington, DC, 2004, pp. 13–22.

[8] C. Venkatramani and T. Chiueh, "Design, implementation, and evaluation of a software-based real-time Ethernet protocol," in *Proc. Conf. Appl., Technol., Architectures, and Protocols for Comput. Commun. (SIGCOMM)*, New York, 1995, pp. 27–37.

[9] P. Pedreiras, L. Almeida, and P. Gai, "The FTT-Ethernet protocol: Merging flexibility, timeliness and efficiency," in *Proc. 14th Euromicro Conf. Real-Time Syst.*, Jun. 2002, pp. 134–142.

[10] 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. Ind. Informat.*, vol. 1, no. 3, pp. 162–172, Aug. 2005.

[11] K. Steinhammer, P. Grillinger, A. Ademaj, and H. Kopetz, "A Time-Triggered Ethernet (TTE) switch," in *Proc. Conf. Des., Autom. Test in Europe (DATE)*, Leuven, Belgium, 2006, pp. 794–799, 3001.

[12] *Real-Time Ethernet Control Automation Technology (EtherCAT)*, IEC/PAS 62407, E. Group, 2008.

[13] Ethernet Powerlink V2.0—Communication Profile Specification, Ethernet Powerlink Standadisation Group (EPSG), 2003.

[14] H. Kopetz*, Real-Time Systems: Design Principles for Distributed Embedded Applications*. Norwell, MA: Kluwer, 1997.

[15] J. Loeser and H. Härtig, "Real time on Ethernet using off-the-shelf hardware," in *Proc. 1st Int. Workshop on Real-Time LANs Internet Age (RTLIA 2002)*, 2002, pp. 59–62.

[16] G. Weiss and R. Alur, "Regular specifications of resource requirements for embedded control software," in *Proc. 14th IEEE Real-Time and Embedded Technol. Appl. Symp. (RTAS)*, 2008, pp. 159–168.

[17] P. Pop, P. Els, and Z. Peng, "Performance estimation for embedded systems with data and control dependencies," in *Proc. 8th Int. Workshop on Hardware/Software Codesign (CODES)*, New York, 2000, pp. 62–66.

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

[19] Varan—Versatile Automation Random Access Network. [Online]. Available: www.varan-bus.net Mar. 2009

[20] "Modbus Application Protocol Specification V1.1b," White paper, Dec. 2006.

[21] *Real-time Ethernet SERCOS III*, IEC/PAS 62410, May 2005.

[22] R. Pigan and M. Metter*, Automating With PROFINET: Industrial Communication Based on Industrial Ethernet*. New York: Wiley, Dec. 2008.

[23] R. E. Walpole, R. H. Myers, S. L. Myers, and K. Ye*, Probability & Statistics for Engineers & Scientists*, 8th ed. Englewood Cliffs, NJ: Prentice-Hall, Mar. 2008.

[24] M. Li, T. V. Achteren, E. Brockmeyer, and F. Catthoor, "Statistical performance analysis and estimation of coarse grain parallel multimedia processing system," in *Proc. 12th IEEE Real-Time and Embedded Technol. Appl. Symp. (RTAS)*, Washington, DC, 2006, pp. 277–288.

[25] P. Ienne and R. Leupers*, Customizable Embedded Processors: Design Technologies and Applications*, 1st ed. San Mateo, CA: Morgan Kaufmann, Jul. 2006.

[26] M. Jacome and G. De Veciana, "Design challenges for new application specific processors," *IEEE Design & Test of Computers*, vol. 17, no. 2, pp. 40–50, 2000.

[27] M. Anand, S. Fischmeister, and I. Lee, "An analysis framework for network-code programs," in *Proc. 6th Annu. ACM Conf. Embedded Softw. (EmSoft)*, Seoul, South Korea, Oct. 2006, pp. 122–131.

[28] M. Anand, S. Fischmeister, and I. Lee, "Composition techniques for tree communication schedules," in *Proc. 19th Euromicro Conf. Real-Time Syst. (ECRTS)*, Pisa, Italy, Jul. 2007, pp. 235–246.

[29] S. Fischmeister and R. Trausmuth, "A programmable arbitration layer for adaptive real-time systems," in *Proc. Int. Workshop on Adaptive and Reconfigurable Embedded Syst. (APRES)*, 2008, pp. 27–31.

[30] S. F. *et al.*, "Network Code Language Specification," Univ. Pennsylvania, Philadelphia, PA, Tech. Rep., 2007, manual & specification.

[31] K. K. M. Gries*, Building ASIPs; the MESCAL Methodology*. Berlin, Germany: Springer-Verlag, 2005.

[32] S. Leibson*, Designing SOCs With Configured Cores: Unleashing the Tensilica Xtensa and Diamond Cores*. San Mateo, CA: Morgan Kaufmann, 2006.

[33] A. V. Aho, R. Sethi, and J. D. Ullman*, Compilers—Principles, Techniques, and Tools*, ser. World Student Series of Computer Science, J. T. DeWolf and M. A. Harrison, Eds. Reading, MA: Addison Wesley, 1986.

[34] J. D. Patterson*, Computer Organization and Design*, 2nd ed. San Mateo, CA: Morgan Kaufmann, 1997.

[35] G. Weiss, S. Fischmeister, M. Anand, and R. Alur, "Specification and analysis of network resource requirements of control systems," in *Proc. 12th Int. Conf. Hybrid Systems: Computation and Control (HSCC)*, San Francisco, CA, Apr. 2009, pp. 381–395.

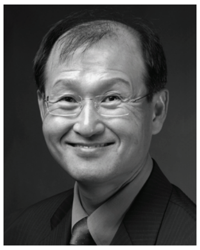
Sebastian Fischmeister (S'97–M'04) received the Dipl.-Ing. degree in computer science from the Vienna University of Technology, Vienna, Austria, in 2000 and the Ph.D. degree in computer science from the University of Salzburg, Salzburg, Austria, in December 2002.

He is an Assistant Professor with the Department of Electrical and Computer Engineering, University of Waterloo, Waterloo, ON, Canada. His primary research interests include software technology and distributed systems for real-time embedded systems.

Robert Trausmuth (M'08) received the Dipl.-Ing. and Ph.D. degrees in technical physics from the Technical University of Vienna, Vienna, Austria, in 1991 and 1996, respectively.

He is a Professor with the Department of Computer Engineering, University of Applied Sciences, Wiener Neustadt, Austria, since 1998. His primary research interests include distributed control systems and real-time communications. Recent projects include the CERN ATLAS Central Detector Control System (profiling and driver development) and the CIMIT Medical Device Plug and Play System (FPGA system implementation).

Prof. Trausmuth is a member of the IEEE Computer Society.



Insup Lee (S'80–M'82–F'01) received the B.S. degree in mathematics from the University of North Carolina, Chapel Hill, in 1977, and the Ph.D. degree in computer science from the University of Wisconsin, Madison, in 1983.

He is the Cecilia Fitler Moore Professor of Computer and Information Science and the Director of PRECISE Center, University of Pennsylvania. His research interests include real-time systems, embedded systems, formal methods and tools, medical device systems, cyberphysical systems, and software engineering. The theme of his research activities has been to assure and improve the correctness, safety, and timeliness of real-time embedded systems.

Prof. Lee has published widely and received the Best Paper Award at RTSS 2003 with I. Shin on compositional schedulability analysis. He received IEEE TC-RTS Technical Achievement Award in 2008. He was Chair of the IEEE Computer Society Technical Committee on Real-Time Systems (2003–2004) and an IEEE CS Distinguished Visitor Speaker (2004–2006). He has served on many program committees and chaired several international conferences and workshops, and also on various steering committees, including the Steering Committee on CPS Week, Embedded Systems Week, and Runtime Verification. He has served on the editorial boards of several scientific journals, including IEEE TRANSACTIONS ON COMPUTERS, *Formal Methods in System Design*, and *Real-Time Systems Journal*. He is a founding Co-Editor-in-Chief of the *KIISE Journal of Computing Science and Engineering* since September 2007. He was a member of the Technical Advisory Group (TAG) of the President's Council of Advisors on Science and Technology (PCAST) Networking and Information Technology (NIT).