Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2014, Article ID 692076, 16 pages
http://dx.doi.org/10.1155/2014/692076

*Research Article*

# Self-Awareness in Computer Networks

**Ariane Keller,[1] Daniel Borkmann,[2] Stephan Neuhaus,[1] and Markus Happe[1]**

[1] *Communication Systems Group, ETH Zurich, Gloriastrasse 35, 8092 Zürich, Switzerland*
[2] *Red Hat Switzerland, Europaallee 41, 8004 Zürich, Switzerland*

Correspondence should be addressed to Ariane Keller; ariane.keller@tik.ee.ethz.ch

Received 6 January 2014; Accepted 11 June 2014; Published 25 August 2014

Academic Editor: Tobias Becker



The Internet architecture works well for a wide variety of communication scenarios. However, its flexibility is limited because it was initially designed to provide communication links between a few static nodes in a homogeneous network and did not attempt to solve the challenges of today's dynamic network environments. Although the Internet has evolved to a global system of interconnected computer networks, which links together billions of heterogeneous compute nodes, its static architecture remained more or less the same. Nowadays the diversity in networked devices, communication requirements, and network conditions vary heavily, which makes it difficult for a static set of protocols to provide the required functionality. Therefore, we propose a self-aware network architecture in which protocol stacks can be built dynamically. Those protocol stacks can be optimized continuously during communication according to the current requirements. For this network architecture we propose an FPGA-based execution environment called EmbedNet that allows for a dynamic mapping of network protocols to either hardware or software. We show that our architecture can reduce the communication overhead significantly by adapting the protocol stack and that the dynamic hardware/software mapping of protocols considerably reduces the CPU load introduced by packet processing.

## 1. Introduction

Modern mobile phones change literally dozens of communication parameters several times per second in order to adapt to changing channel conditions, such as distance to cell tower, signal quality, and activity on the channel. Mobile phones (and their communication counterparts) engage in this complicated activity not because of a relish for clever technology, but out of the need to support a growing number of devices with limited physical resources (wireless bandwidth).

The Internet-of-Things will see a similar explosion in Internet-capable devices, of which current estimates predict billions in just a few years. We can therefore expect these devices to compete for resources that are becoming scarcer and scarcer, per device, as device numbers grow, just like mobile phones; we can similarly expect that devices will be exposed to changing operating environments in ways that are comparable to changing channel conditions for mobile phones.

But how does the current Internet architecture support a device's ability to adapt its Internet communication to changing operating conditions? The answer is that it does not: there is a very small choice of protocols, essentially limited to TCP or UDP, and even then it is difficult to switch from one to the other during a conversation. And there are virtually no possibilities to sense operating conditions, such as the number of dropped packets. Yet the benefits of sensing and adapting to changing conditions would be potentially enormous.

For example, if the device has never experienced a dropped packet while talking to a certain server, it might want to switch from a reliable protocol like TCP to a potentially unreliable one like UDP, saving the overhead in processing time and electrical power that is needed for the reliability protocol.

With a bit of effort, this might be achieved using today's Internet architecture. But other features, equally desirable, are more difficult. For example, a medical heart rate monitor could want to talk to a server in the home using unencrypted messages, trusting to the link-layer encryption in the home's wireless network to keep messages secure in transit. When the wearer of the heart-rate monitor leaves the home and is using some public WiFi node, however, messages should be encrypted. Of course, this adds additional load and increases energy consumption, so it should only be done when needed.

Another example is the question of updates. Heart rate monitors do not for the most part expect to be updated at all, and it may even be that its continuous functioning is deemed so critical that it is not updated because that would incur downtime. With a flexible architecture, it is possible to download an update and to switch to the updated version, while the monitor continues to collect data. Should the update turn out to be flawed, it is even possible to switch back to the previous version.

The final example is somewhat related to the previous one. Should the heart rate monitor be based on FPGAs, then another design decision arises because FPGAs support the execution of functions either in hardware or in software. While hardware implementations are generally faster than their software counterparts and thus preferable, FPGA space is limited, which usually makes it impossible to run all functions in hardware. Therefore, the decision which functions to execute in hardware and to execute in software is subject to tradeoffs, which usually depend on the operating conditions. Since these operating conditions may change, so may the optimal assignment of functions to hardware or software. For example, if the heart rate monitor finds itself close to its home server and wants to upload a large number of measurements, it will expect much traffic and thus the transmission protocol should be executed in hardware, at the expense of additional encryption. Outside its home network, privacy is more important than speed, so the encryption module could be in hardware and the transmission protocol in software.

This paper presents a novel clean-slate network architecture that relies on dynamic protocol stacks and therefore fundamentally differs from the Internet architecture that we have learned to cherish over the last decades. In contrast to the static protocol stacks that are used in today's Internet architecture, we propose to split up the networking functionality into functional blocks (FBs), which can be dynamically linked with each other to form arbitrary protocol stacks. At the beginning of a new communication between two peers, the nodes have to agree on a common protocol stack that respects the communication requirements dictated by the application and takes into account the node-specific system constraints and the current network conditions. Changes in the node-specific system constraints or in the network conditions might result in an adaptation of the used protocol stack in order to optimize the communication behavior of the nodes in a resource-efficient manner.

Unfortunately, replacing static standard protocol stacks with dynamic protocol stacks generates the following challenges.

(1) How can application developers program their source code without knowing the protocol stack that will be later used for communication over the network?

(2) How can communication peers set up a common protocol stack when there is no fixed stack?

(3) What methods can be used by nodes to adapt their protocol stacks while maintaining preexisting connections?

(4) How can a node be aware of its resources and the network conditions, and how can it use this information to decide when and how to adapt its protocol stack?

(5) How can we integrate hardware accelerators into the network architecture, even if the used functional blocks are not known before the manufacture of the device?

In this paper we will present a novel FPGA-based self-aware network architecture, called EmbedNet, which supports dynamic protocol stacks and provides solutions to all these challenges.

To answer the first challenge, we have developed a novel Berkeley Software Distribution (BSD) socket type, where the application designer uses key words that specify the communication requirements. These key words might define that the application requires a reliable, secure, and robust communication channel. At run time, the EmbedNet platform composes suitable protocol stacks by matching the application's requirements with the properties of the available functional blocks. The execution environment dynamically selects the best protocol stack for the current situation.

For the second challenge, we assume that the source node initially sends the set of suitable protocol stacks to the destination node using a default protocol, which has to be supported by all nodes. Each functional block will have a unique identifier. A protocol stack can be described as a concatenation of its FB identifiers.

To handle the third challenge, our EmbedNet architecture sends control messages, which can be used by the communication peers to negotiate the next protocol stack. Each packet contains a header which indicates the used protocol stack. Hence, after a protocol stack adaptation the destination node can distinguish between packets processed by the old protocol stack and packets processed by the new protocol stack.

For the fourth challenge, we apply novel concepts and methodologies for self-awareness in networking. Our self-aware network architecture can monitor the system state and network condition using a set of sensors and dedicated models. A self-aware stack builder is part of our EmbedNet architecture that defines at which times the protocol stacks should be adapted. The current implementation of our EmbedNet architecture is aware of the CPU utilization and the workload (in terms of packets to be processed for each communication connection).

For the final challenge, we implement our self-aware network architecture on a field programmable gate array (FPGA). This allows us to dynamically map selected functional blocks to reconfigurable hardware using partial reconfiguration. Hence, EmbedNet can provide hardware acceleration to functional blocks, even if these blocks were unknown at the manufacturing time of the device. However, due to the limited amount of FPGA area, only a limited subset of functional blocks can be mapped to hardware at the same time. Therefore, we apply a self-aware approach such that our system is aware of the workload of all used protocol stacks in order to identify the best hardware/software mapping for all

used functional blocks. This kind of adaptation can be done by each network node individually.

In previous work [1–6] we worked towards such a *self-aware network architecture* and presented individual aspects. While we described in [1] how protocol stacks can be built from individual functional blocks, we described in [6] how such systems can adapt the protocol stack at run time. In [2–4] we presented basic aspects of the implementation in short papers and in [5] we showed how network traffic characteristics can be learned. This paper finally integrates all individual aspects to a fully working system and provides an evaluation of the complete system. To this end, several components described in earlier work were improved, and several new components were added.

Specifically, we provide the following contributions.

(a) We propose EmbedNet, a novel FPGA-based self-aware network node architecture, which supports (i) the autonomous configuration of dynamic protocol stacks and (ii) the dynamic mapping of network functionality to either hardware or software. EmbedNet observes the networking environment using dedicated sensors and internal network models and adapts its protocol stacks when that environment changes. Our EmbedNet prototype does not experience any downtime due to reconfiguration.

(b) We present novel methods to set up and adapt protocol stacks based on the application requirements and the current network conditions.

(c) We show that the self-adaptation of a protocol stack can reduce the communication overhead in terms of sent packets as compared to static stacks in a real-world scenario.

(d) We propose two scheduling algorithms and show how dynamic mapping improves the performance of our EmbedNet platform considerably for different network traffic mixes.

The rest of this paper is structured as follows. After giving an overview of related work (Section 2), we present our self-aware network architecture and show how we adapt the protocol stack and the hardware/software mapping to changing network conditions (Section 3). Next, we describe our FPGA-based execution environment for the self-aware network architecture (Section 4). After that, we demonstrate the efficiency of our approach by comparing our system against static solutions in real-world network scenarios (Section 5). We finish with conclusions and future work (Section 6).

## 2. Related Work

*2.1. From Active Networking to Future Internet Research.* Already, in the early 1990s, as the Internet rapidly grew and became ever more popular, researchers investigated dynamic network architectures. Tennenhouse and Wetherall proposed *Active Networks*, in which users could inject custom code into the network [7]. This code was associated with a set of packets that traversed the network from the source over several routers to the destination, and which was executed on intermediate nodes and modified the packets on the fly as desired. Tennenhouse and Wetherall suggested four different possibilities for active packet processing:

(i) network operators inject code on the intermediate nodes;

(ii) every packet contains the program code to be executed;

(iii) packets can put code into a node and other packets could use that code;

(iv) packets contain a reference to code on an external server and the routers download the code from that server and store it in a local cache.

Based on those four possibilities, many researchers worked on specific Active Networking architectures in the following ten years. This work is summarized by several survey papers [8–10]. However, none of the Active Network architectures found its way into the commercial Internet, some because of performance issues, others because of security concerns, and still others because no hardware supported them.

Still, research on dynamic network architectures continued, driven on the one hand by Internet issues such as poor scalability, extensibility, security, and reliability, and on the other hand by a change from a static, provider-centric network to a mobile and user-centric network. For each of these issues, there exist efforts that tackle them. For example, scalability can be tackled with IPv6 or Network Address Translation (NAT), security problems can sometimes be mitigated by Virtual Private Networks (VPNs), and there is a large number of new routing or transmission protocols for mobile networks, such as Mobile IP or hop by hop transmission.

The umbrella term for this research is *Future Internet*. Under this term, researchers investigate not only additions and patches to the Internet architectures, but also *clean-slate architectures*. These architectures follow from asking the question "given what we know today, how would we have designed the Internet if we had to do it all over again?". Our self-aware network architecture is also a clean-slate Future Internet architecture.

*2.2. FPGAs in Active Networking.* The original work in Active Networking exploited the flexibility of software-only systems. But already in 1998 the first extensions appeared that used FPGAs as hardware accelerators, even though, at that time, FPGAs were small and could only be reconfigured as a whole. For example, Hadzic and Smith introduced the Programmable Protocol Processing Pipeline (P4) architecture [11]. It uses several FPGAs and a switching array that decides which packet will be processed by which FPGA. They can add new functions to the system by reconfiguring an FPGA, and they implement different protocol stacks by changing the path of a packet through the FPGAs. At the same time, Decasper et al. introduced the Active Network Node (ANN) [12]. The central component of their system is

a switch which has, for each input port, a CPU and an FPGA. Performance-critical functions are executed on the FPGA, which can be reconfigured by the CPU. However, they do not discuss how they decide when to reconfigure the FPGA. The Plato architecture is an FPGA-only architecture for Active Networks [13]. They also aim at reconfiguring the FPGA for flexibility but do not discuss how a reconfiguration could be started. Fragkiadakis et al. suggest an architecture with one CPU and one FPGA [14]. They allow only one active application to run at a time, and this application determines the functionality implemented on the FPGA. The currently active application depends on the received packet. There is no discussion on reconfiguration overhead of the FPGA, which seems surprising since the active application could change with every packet.

To summarize, these architectures have realized the potential of FPGAs for active networking but have failed to convincingly address all aspects of using FPGAs. Either they failed to describe the circumstances under which an adaptation is triggered or they failed to discuss adaptation overhead and hence also the maximum adaptation frequency that still leads to an overall performance benefit.

*2.3. FPGA-Based Platforms for Network Processing.* Already in 2000, researchers at the Washington University in St. Louis introduced an FPGA-based system called FPX [15] that enhances the Washington University Gigabit Switch [16] with reprogrammable features. They used two Xilinx FPGAs on a custom board. (Only 85 of these boards were ever produced.) Several research projects were done on the FPX in the areas of IP routing, video processing, and partial FPGA reconfiguration. However, this project is no longer continued, since the hardware is outdated. Similarly, the RiceNIC project [17] developed an open network interface platform for prototyping and educational purposes. In this case, the problem is twofold. First, their underlying hardware is not available anymore and, second, RiceNIC's architecture is tightly coupled with the hardware, which would mean a major redesign when switching to new hardware.

From 2002 to 2009 the German research association DFG [18] funded a priority program entitled *Reconfigurable computing systems* [19]. Within this program two projects were using FPGAs to build more powerful network processors. In the DynaCore project [20] the network processor receives packets and executes the protocols that are computationally cheap. Packets that need processing through computationally expensive protocols are forwarded to an FPGA that is connected to the network processor over a Gigabit Ethernet interface. Partial reconfiguration of the FPGA can be used to change the functionality provided by the FPGA. There is a limited set of protocols that can be executed on the FPGA and the transition condition from one configuration to the next is statically defined.

In contrast to this architecture the FlexPath NP [21] architecture receives packets in the FPGA and processes them completely in hardware if all required protocols are available. Only packets requiring different protocols are forwarded to a CPU cluster, or, with a combination of FlexPath and DynaCore and a corresponding hardware implementation on the DynaCore FPGA, they are sent to the DynaCore FPGA.

Even though DynaCore and FlexPath could be interesting to other researchers, no public version is available. However, if a researcher starts with network programming on FPGAs he can choose from several platforms. The NetFPGA project [22, 23] provides a networking board with four Ethernet interfaces. The community around NetFPGA offers several reference designs that make it easier to start working with the board. A commercial alternative is the COMBO FPGA Board [24] from INVEA-TECH. INVEA-TECH also offers a software suite that facilitates the development of networking programs for their board.

In addition to specialized networking boards, general purpose FPGA evaluation boards can also be used to develop networking applications. They do not have frameworks dedicated to network programming but instead have a wide range of supported peripherals, have professional documentation, and are relatively cheap. This is especially interesting for mixed research groups that share a single hardware platform and that have a diverse range of requirements on their platform.

We have implemented our self-aware network node architecture on a general purpose Xilinx Virtex-6 evaluation board. However, our architecture can also be implemented on specialized networking boards, such as a NetFPGA or a COMBO FPGA board.

*2.4. Runtime Adaptation for Network Processing.* Runtime adaptation was studied in the context of network processors since the beginning of 2004 where Kokku et al. showed that the performance of a network processor can be enhanced by adapting the mapping of networking elements to processing units frequently [25]. The optimization goal is either minimizing power consumption [26, 27] or providing maximum throughput [28–30]. They look at buffer occupancy between processing units and the utilisation of processing units to determine the optimal mapping. These approaches are interesting for our scenario, but they do not fully cover it. First, they have a homogeneous system with several tens of identical processing units, whereas we are dealing with two entirely different kinds of processing units (one is a CPU and the other a hardware accelerator). Second, the mechanisms that connect the execution units are different: whereas they can map several modules to the processing units, we can do this only for the CPU, which has to execute all the modules that are not implemented in hardware.

Kachris and Vassiliadis [31] evaluate run time reconfigurable network processor designs based on FPGAs. Their architecture received packets on an embedded processor and was able to dynamically reconfigure hardware accelerators. Their design used three different hardware accelerators. Moreover, these accelerators could be implemented multiple times. In order to find the best mapping of accelerators, they use Integer Linear Programming (ILP) on five different traffic mixes at design time. At run time, their system analyzes the actual traffic mix and chooses the one configuration corresponding to the closest preanalyzed traffic mix. This

approach is interesting since it moves the computation-intensive task of finding an optimal mapping from run time to design time. However, in our scenario this is unrealistic, since we cannot make any assumptions on traffic mixes and we allow for the introduction of new protocols at run time.

*2.5. Summary.* In contrast to the architectures we reviewed above, ours not only adapts to different distributions of network packets, but also allows us to seamlessly integrate completely new protocols. Additionally, packets can be processed in hardware or in software, depending on the current protocol and performance requirements. Furthermore, we require a methodology, an algorithm that decides which parts should be executed where. None of the algorithms used in related work can be adapted to our scenario.

## 3. Concepts and Methodologies for Self-Awareness in Networking

Dynamic protocol stacks split network functionality into individual *functional blocks* that can be combined at run time to form a protocol stack and that can be changed on the fly, in order to provide a communication channel that is continuously optimized. This section gives some background on packet processing and describes the building blocks and algorithms that enable self-aware networking. Section 4 will then focus on the actual implementation of our self-aware network architecture.

*3.1. Background on Packet Processing.* Packet processing is different from other computing workloads, and in order to understand some of the design choices explained later, it is important to understand how.

(i) Processing a network packet usually requires only simple processing steps at which hardware circuits excel, such as comparing bits, decrementing integers, prepending headers, and looking up data in a hash table. No floating-point arithmetic is required.

(ii) On the network layer, most network packets do not depend on each other. Hence they can be processed independently and in parallel.

(iii) Processing a packet can be split into several sequential steps. For a simple router this could be "verify MAC address to ensure packet is for this node," "determine next hop address," "update time-to-live in IP header," "update checksum," "update MAC header," and "transmit packet." Many packets received on one node require the same processing steps. Together with the previous characteristic this means that a pipelined architecture is appropriate for processing network packets.

(iv) The rate of data to be processed is determined from the outside. If the packet arrival rate is faster than the packet processing rate, packets will be dropped. Hence, the maximum packet rate a system can process is a good system performance indicator, but how long it takes to process a given workload is not. Again, a pipelined architecture is more appropriate for this kind of workload than a sequential one.

*3.2. Self-Aware Network Architecture.* For self-aware networking, we propose the network node architecture shown in Figure 1. The architecture contains the following building blocks.

(i) The *network models* contain a list of supported network protocols, the network characteristics, and predictions on how the networking environment might look like in the future.

(ii) *Sensors* provide information such as the signal-to-noise ratio, remaining battery life, or throughput. Sensors can be passive (just observing) or active (probing the environment and observing the reaction).

(iii) The *sensor daemon* collects data from individual sensors. It offers additional functionality such as sending notifications whenever a monitored value exceeds a specified threshold.

(iv) The *self-adaptation engine* contains a *strategy finder*, which selects the current strategy (minimize power, maximize throughput, etc.), and a *stack builder*, which determines the best stack and adapts the networking core accordingly.

(v) The *networking core* is responsible for processing packets. Functional blocks do not interact directly with each other. This makes it easier to change the protocol stack at run time. Rather, in software there is a *Packet Processing Engine (PPE)* and in hardware there is a dedicated *network on chip (NoC)* that forward packets between the different functional blocks. Each functional block is identified by an *Information Dispatch Point (IDP)*, which is mapped to the function actually processing the packets. The complete protocol stack is identified by the sequence of IDPs of its functional blocks. The details of our networking core are described in [2].

*3.3. Self-Aware Protocol Stack Setup and Adaptation.* The protocol stack setup consists of two parts. First, the local node finds all feasible protocol stacks from the application's requirements and the functional blocks. In the second step, this set of stacks is communicated to the local node's communication partners and they then choose one of the stacks (according to local optimization criteria).

*3.3.1. Local Node.* In the Internet architecture, an application solves the problem of choosing a suitable protocol stack by using a specific BSD socket type and additional libraries as needed, for example, for encryption. In our self-aware networking architecture, the application can instead specify a set of properties that need to be fulfilled for a given communication. The current Application Programming Interface (API) uses simple key words for both protocols and requirements; see Algorithm 1 for an example. The stack builder

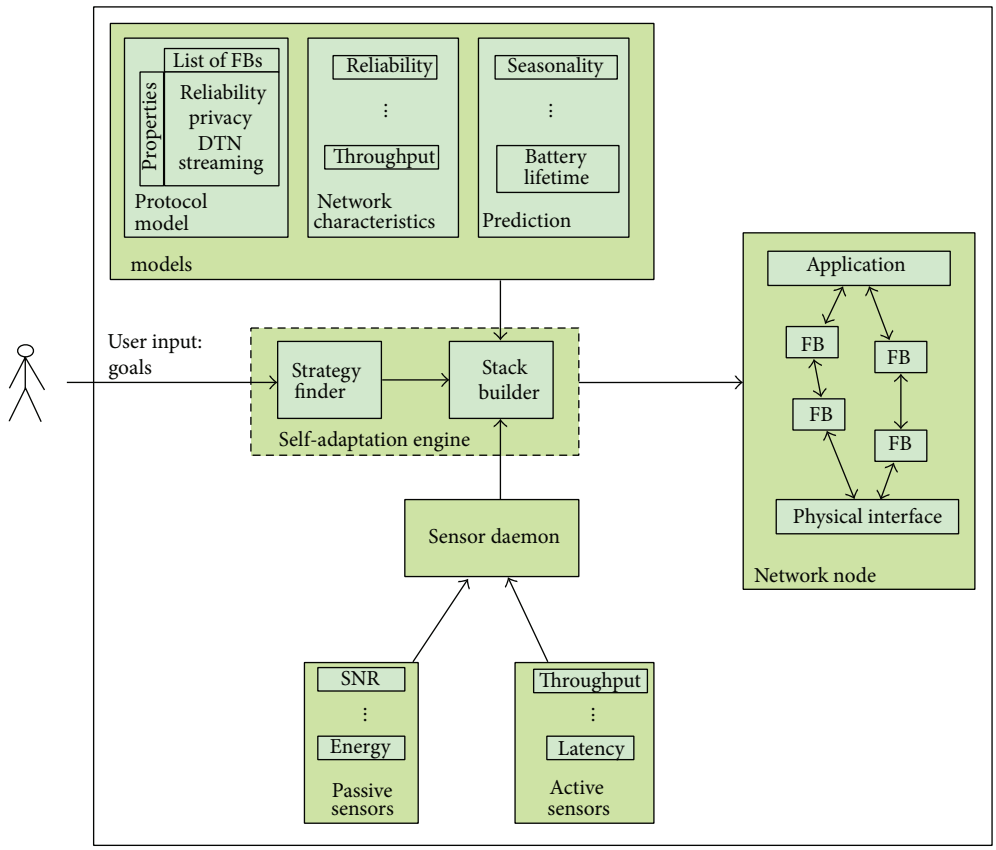


Figure 1: Overview of the self-aware node architecture.

```
struct bind_msg
        char name[FBNAMSIZE];
        char app[FBNAMSIZE];
        char props[MAX_PROPS][10];
        int flags;
;
int main(void)
    ...
    char buff[512];
    memset(buff, 0, sizeof(buff));
    sock = socket(PF_DPS, SOCK_RAW, 0);
    ...
    bmsg = (struct bind_msg *) buff;
    strcpy(bmsg->app, "chat");
    bmsg->props[0] = "RELIABILITY";
    bmsg->props[1] = "PRIVACY";
    bmsg->flags = TYPE_CLIENT;
    ...
    ret = bind_config(bmsg);
    ...
    ret = sendto(sock,data,len,0,NULL,0);
    ...
    ret = close(sock);
    ...
    return 0;
```

Algorithm 1: Dynamic protocol stack architecture API example.

then examines the protocol models and finds all protocol stacks that match the requirements. Local preference of one stack over the other is expressed in additional requirements. This automatically means that all such stacks are equally preferable to the local node and a communication partner cannot choose a protocol from this set that the local node would rather not use.

*3.3.2. Internode Adaptation.* In the second part of the protocol stack setup phase, the local node negotiates a particular protocol stack with the destination node or nodes. First, the local node computes identifiers for all the protocol stacks computed in the first part and sends them to the destination node. The destination node then decides which protocol stack to use, sets up this protocol stack, and sends the identifier of the chosen stack back to the local node.

If the local node never receives a reply from the destination, which could happen on a lossy link, the source resends the configuration message and waits for the confirmation. After the completion of the negotiation phase, actual data transmission starts.

It is important to note that all self-aware network nodes use the same method to compute the identifier of a given protocol stack. In order to distinguish between data messages and control messages (for the stack negotiation) a one byte header is introduced.

There remains the problem of what protocol to choose for the protocol negotiation phase itself, for which we simply assume that all nodes in a given network segment use the Ethernet protocol. Similarly, if a connection to a node in another segment should be established, the intermediate nodes must use the same internetworking protocol, for example, IPv4 or IPv6.

*3.3.3. Stack Identification.* In the Internet architecture, the decision on how to process a packet is based on *next-header fields* that are part of each protocol header. For example, in the next-header field of the Ethernet protocol it is specified whether the protocol encapsulated in this Ethernet frame is IPv4, IPv6, ARP, ICMP, and so forth. If the entire protocol stack is negotiated up front and hence known to the packet processing engines on the source and destination, this step-by-step resolution of the next protocol is not necessary. Rather, we can use one protocol stack identifier per connection.

This identifier is calculated by the stack builder as follows: every functional block has a unique name, derived from the inverted URL associated with its developers. (This is similar to the convention for package names in the Java programming language.) The unique identifier for the overall protocol stack is then obtained by concatenating the individual names and hashing them. If the identical protocol is implemented by several developers and their implementations pass an interoperability test, a special interoperability name should be used. Upon packet reception, the Ethernet functional block checks the hash and forwards the packet to the corresponding stack "pipeline."

*3.3.4. Changing Protocol Stacks.* Negotiating a change in the protocol stack is done just like negotiating a new stack, except that renegotiation is executed over the currently used protocol stack. During the adaptation of the protocol stack, packets might be reordered on their way from source to destination. Therefore, special care has to be taken that packets still belonging to the old stack are not processed by the new stack and vice versa. Since the hash that identifies a given stack will change when the protocol stack is changed, also the packets sent over the new stack will be identified with a different hash. This hash is used to dispatch the packet either to the new or the old protocol stack.

Figure 2 shows what happens during a protocol stack change. First, a source node sends raw packets to a destination node. This "empty" protocol stack is identified with *hash1*. Then, the source nodes wants to renegotiate the protocol stack by sending possible changes to the destination node. The destination node selects a protocol stack, which includes the functional block *FB1*. The destination node sets up the new protocol stack, which is identified with *hash2*, and informs the source node about its selection. Then, the source node also configures the new protocol stack and starts to send packets using the new protocol stack. For some transition time, the destination node supports both protocol stacks. The destination node can therefore correctly process *packet 2*, although it arrives later than the first packet that uses the new protocol stack.

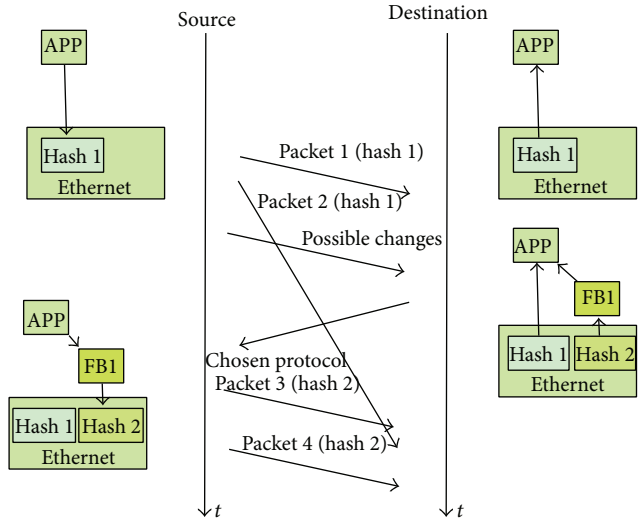


Figure 2: Updating the dynamic protocol stack over time.

*3.4. Dynamic Hardware/Software Mapping.* In order to improve the performance of the system as compared to a software only system, functional blocks can be implemented in hardware accelerators. While for performance reasons it would be desirable to implement as many functional blocks in hardware as possible, we are constrained by the FPGA area and by the requirement to include novel protocols that are unknown at design time. Therefore, a system is required that dynamically decides which functional block should be implemented in hardware and which in software.

Figure 3 shows our approach for self-aware hardware/software mapping. It is a simplified version of the general self-aware node architecture described in Figure 1. The mapping algorithm obtains information from three different sources: *goals*, *sensors*, and *models*. The goals are specified by the user and might be "no packet loss," "minimize CPU load caused by network traffic," and so forth. The sensors collect statistical information such as "packets per second per flow" or "CPU load." The models describe the overall system and can either be known or learned at run time. Examples of models are "a packet is processed faster in hardware than in software," or "packets per second per flow does not change between two measurement intervals." Based on this input, the self-aware scheduler determines the hardware software mapping and also initiates the reconfiguration of the hardware, should that be required. Some specific self-aware mapping algorithms are described in Section 5.2.1.

## 4. The EmbedNet Architecture

In this section we present EmbedNet, which is an FPGA-based system-on-chip implementation of all self-awareness concepts and methodologies, which have been presented in the last section.

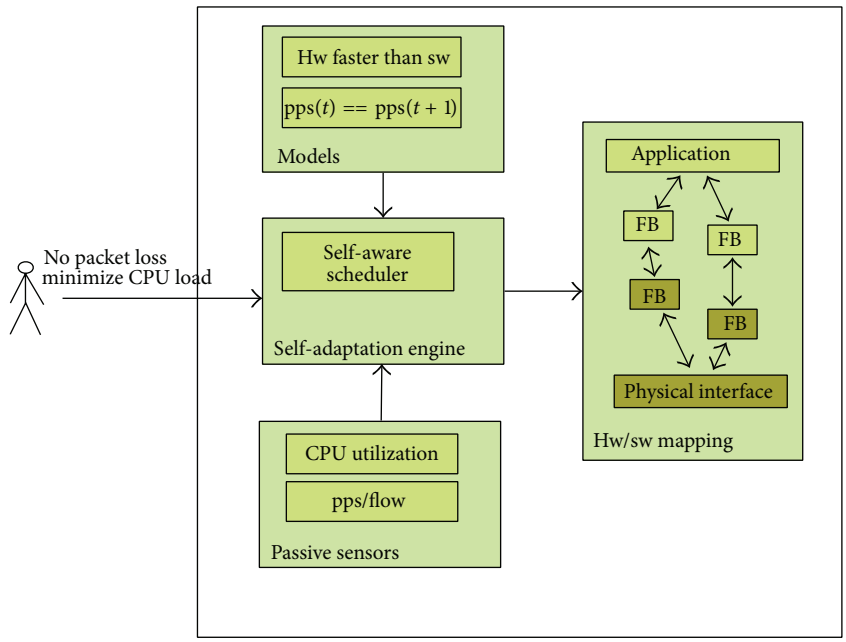


Figure 3: Self-aware hardware/software mapping.

*4.1. System-on-Chip Hardware Design.* The hardware design of our FPGA-based EmbedNet prototype is depicted in Figure 4. We use a System-on-Chip (SoC) architecture that combines a soft-core MicroBlaze CPU [32] with several hardware modules and peripherals on a single device. The Linux operating system runs on the MicroBlaze CPU and the required file system is stored on a compact flash disk. The functional blocks can either be executed on the MicroBlaze or directly in the FPGA. In order to implement our architecture on an FPGA board, we assume that it has external SDRAM, a physical Ethernet interface and a compact flash disk reader.

There are three modules that are always present in the FPGA. One is the Ethernet functional block (ETH) that interfaces with the physical interface (PHY in the figures) and two modules which are responsible for transmitting the packets over the hardware/software boundary (called H2S and S2H, resp.). In addition to these statically configured modules, there are also *dynamic* modules (PR). Those modules can be reconfigured at run time with the functionality of arbitrary functional blocks.

Functional blocks are connected by a network on chip (NoC) that forwards packets between them and also supports pipelined packet processing. The NoC consists of switches in a ring topology, where each switch connects to a configurable number of functional blocks. The total number of modules is a design-time parameter; this allows for the throughput of the NoC to scale appropriately by increasing the bandwidth between the switches and by allowing for more hardware modules to be connected to one switch. Run-time reconfiguration of the modules is done with the help of the Xilinx core XPS HWICAP [33], where the bit files for the partial reconfiguration are stored on an external flash card.

While a given area on the FPGA is reconfigured, it may emit spurious signals. In order to prevent packet processing errors during reconfiguration, we added a dedicated enabling block between the NoC and each dynamic module. During reconfiguration, this block sets the signals that go into the NoC to a known value.

For communication between hardware and software, we use the ReconOS extension to Linux [34]. While the original ReconOS implementation provides transparent communication between Linux user space and the hardware, we have extended it to also support communication between Linux kernel space and the hardware [4]. This extension is required because many software parts of our network architecture run in the kernel space for performance reasons.

To aid implementation of a functional block in hardware and software, we provide wrappers: in hardware, this is a VHDL entity and in software this is a Linux kernel module. The wrapper consists of the code required for receiving and sending packets and configuration data as well as transferring internal state between a hardware and a software module.

Since there is no automatic translation from a functional block in software to one in hardware, it is the responsibility of the functional block's author to make sure that the respective implementations are equivalent and also to provide the state that is required when resuming a hardware block in software or vice versa.

*4.2. Execution Environment.* The complete EmbedNet execution environment is shown in Figure 5. The packet processing framework together with the functional blocks is responsible for packet processing, whereas the self-aware framework is responsible for the adaptation of the system. In the left part, user-space applications send and receive packets over a BSD socket interface. Since we do not use the TCP/IP protocol suite, we implemented a dedicated socket class that offers the

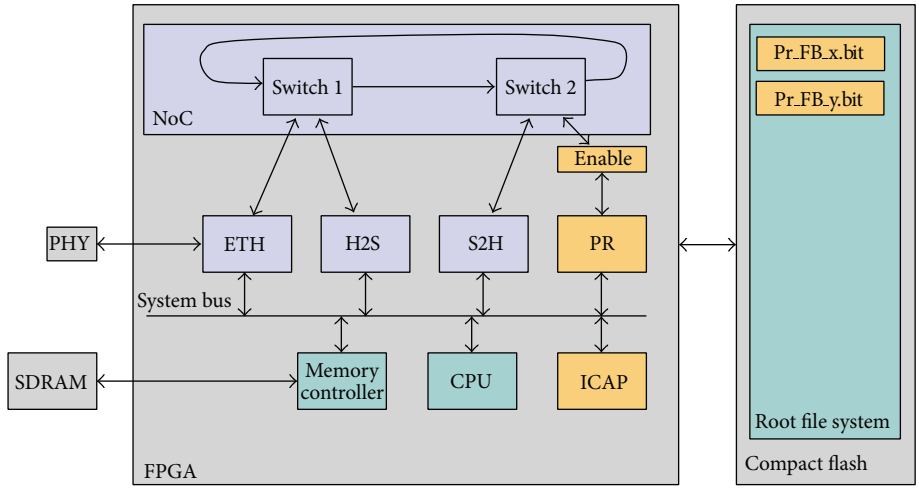


Figure 4: EmbedNet FPGA design.

well-known system calls like *send*() and *recv*(). Sending a packet through our socket inserts it in the packet processing engine, which forwards the packets to the correct functional blocks, no matter whether the block is currently mapped to software or hardware.

The architecture uses a dedicated addressing scheme for the functional blocks. The system updates these addresses on the fly whenever it needs to adapt the route a packet flow takes within a single node. As already described in Section 3 each functional block is identified by an Information Dispatch Point (IDP). An IDP is mapped to the receive function of the corresponding functional block. If the functional block is currently executed in software, a pointer to its function is stored, if the functional block is executed in hardware, the hardware address of the functional block is stored. This address consists of the number of the switches to which the functional block is connected to and the port number on this switch.

If a packet shall be forwarded, first the IDP of the next functional block is looked up. Second, it is checked whether the next block is implemented in hardware, and, if so, the current hardware address is looked up. In software, this decision is done by the packet processing engine. In hardware, however, each functional block has to make this decision by itself. Therefore, each hardware functional block has a dedicated configuration interface. The hardware Ethernet functional block is slightly different, since in addition to making IDP-to-address lookups, it also has to make lookups from the hash identifying a given packet flow to IDPs (and vice versa for sending packets).

When adapting the packet flow due to either a new protocol stack or a new hardware/software mapping, conceptually, only the mapping of the IDP to the actual execution environment needs to be changed. Practically, the procedure is somewhat more complicated, since the FPGA also needs to be reconfigured in order to hold the new functional block.

The self-aware framework in Figure 5 shows the infrastructure that is required for the self-aware mapping of functional blocks to either hardware or software. It is split between Linux user space, Linux kernel space, and the FPGA itself. The central part is the *self-aware scheduler* which is responsible for calculating the hardware/software mapping. While the framework does not specify a given algorithm to be executed, it provides an interface to access the statistics generated by the packet processing engine, such as the number of packets it processed for a given flow. After the self-aware scheduler decides on a new mapping, it performs the reconfiguration.

*4.3. Adapting the Hardware/Software Mapping.* For the explanation of the required steps to adapt the hardware/software mapping of a protocol stack, let us consider the following scenario: there are two protocol stacks $e\text{th} \rightarrow B \rightarrow C$ and $e\text{th} \rightarrow D \rightarrow E$ from which currently $e$th and $B$ are mapped to hardware and the rest is implemented in software. The algorithm decides that in the new mapping $D$ should be implemented in hardware and $B$ in software.

This results in the following steps, which are also described in Figure 6.

*(I) Move B from hardware to software*. The transition of a functional block from hardware to software does not take much time. A functional block that is currently used in a protocol stack is always present in software, regardless of whether it is currently processing packets or not. Therefore, the following three steps need to be done (compare steps 1 to 3 in Figure 6).

(a) *Stop forwarding packets to B*. Functional block $e$th is configured to forward all packets to the software where they are buffered.

(b) *Transfer state*. The state is gathered from the hardware block and transferred to the software block.

(c) *Forward packets to B*. The core forwards the packets again to functional block $B$.

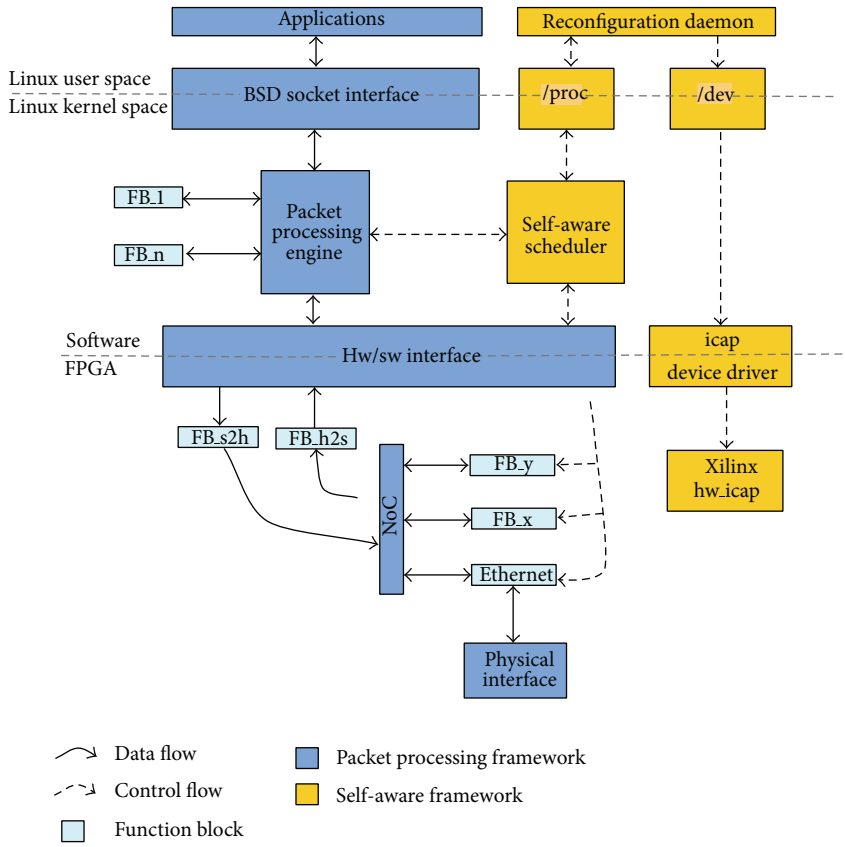


Figure 5: Overall system allowing for a self-aware hardware/software mapping.

This completes the mapping procedure for functional block $B$ and the hardware module is not required anymore and can be reconfigured.

*(II) Move D from software to hardware*. The transition of a functional block from software to hardware requires more time, since the FPGA needs to be reconfigured. However, this reconfiguration can be done, while the functional block is still executing packets in software. The following steps are required.

(a) *Partial reconfiguration*. Perform the partial reconfiguration of the hardware module with the bit file required for functional block $D$ (step 4).

(b) *Stop forwarding packets to D*. In the current implementation this involves buffering all packets in software. However, one could envision an implementation that buffers the packets in hardware (step 5).

(c) *Transfer state*. The state is gathered from the software module and transferred to the hardware module (step 5).

(d) *Forward packets to D*. Functional block *e*th is configured to forward the packets to the hardware module and the packet processing engine forwards the buffered packets to the hardware. In this step the packets might arrive out of order in the hardware module. This could be avoided if the buffering was implemented in hardware (step 6).

This completes the remapping procedure. The functional blocks that are remapped only need to be stopped for a short period of time, where the state is transferred from the software to the hardware or vice versa. The functional blocks not involved in the remapping are running continuously.

When employing such a system it needs to be considered whether for the given situation it is tolerable to have a system that reorders packets (e.g., because the packet order is not important or because the packets get ordered on a higher protocol layer that is always implemented in software) or whether the local node is not allowed to reorder packets (e.g., because some functional blocks that will be executed in hardware require in-order packet processing). For the remainder of this paper, we assume that the packet order does not matter and we therefore can live with the packet reordering that might be introduced during system reconfiguration. Specifically in our experimental setup, we send packets in a way that makes reordering unnecessary.

## 5. Experimental Results

We have performed two different kinds of experiments to analyze our self-aware network architecture. In Section 5.1, we show the benefits of dynamic protocol stacks on software-only systems. We have evaluated our strategies for setting up

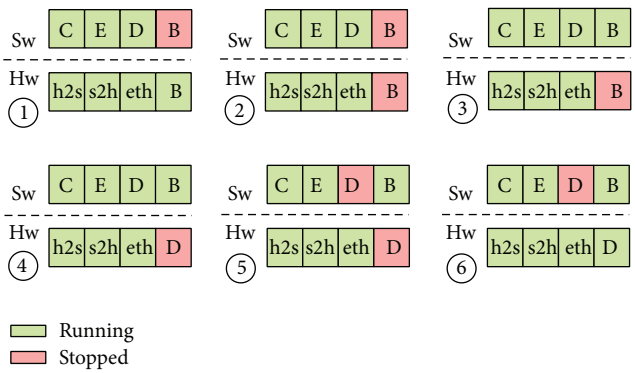


Figure 6: Dynamic hardware/software mapping.

a protocol stack and adapting it over time using commodity laptops that only implement the software part of our self-aware network architecture. In Section 5.2 we evaluate our self-aware hardware/software mapping algorithms for two concurrent protocol stacks, which were implemented on our FPGA-based EmbedNet platform. The self-aware network architecture for both cases is available in github at https://github.com/EPiCS/reconos/tree/v3.0_dev.

*5.1. Self-Aware Protocol Stack Adaptations.* In order to evaluate the benefits of a self-aware network architecture, we show how our system autonomously adapts itself to changing network conditions. We developed a simple application that mimics a sensor that sends measurement data periodically to a server. We argue that transmitting a packet over a wireless interface costs energy and therefore should only be performed when necessary. Therefore, we implemented a stack builder that includes an idle repeat request (IRR) reliability protocol in the protocol stack only when sensors report low link quality. The link quality is determined by a sensor that divides the current with the maximum possible wireless link quality. Our link-quality-aware network architecture [6] is shown in Figure 7.

We evaluated our architecture on commodity notebooks. In order to obtain reproducible results, we used a wired connection between the test machines and used the Linux traffic control tool `tc` with the `netem discipline` [35] to emulate packet loss. We recorded the link quality between two nodes while walking around in our office building; see Figure 8. We have used this recording as realistic input for our emulation. Simultaneously, we measured that packets got lost when the link quality was below 35%.

Our stack builder requests to be notified by the sensor daemon when the signal strength falls below a threshold of 40% or increases beyond 50%; see Figure 8. Upon such an event, it either inserts or removes the reliability module and renegotiates the protocol stack with the neighboring node. The lower threshold for renegotiation ensures that the reliability protocol is inserted to the protocol stack before the link quality reaches the critical value of 35%. The upper threshold is used to avoid frequent adaptations of the protocol stack.

Table 1: Comparison between static and self-aware configurations over 140 seconds [6].

| Config. | Packet loss rate | Overhead | |
|---|---|---|---|
| | | Reliability | Reconfig. |
| Unreliable | 31% | — | — |
| Reliable | 0% | 128% | — |
| Autonomous | 0% | 60% | 40% |

For evaluation purposes, we compared the data loss rate and the total number of packets sent for different configurations; see Table 1. The configuration labeled "unreliable" never uses reliability, "reliable" always uses reliability, and "autonomous" dynamically adapts itself to the link quality. We used these measured values to emulate the network conditions on a machine that connected the two test machines.

The configuration with no reliability lost on average 31% of the packets, whereas we did not observe packet loss in the other two configurations. However, this reliability comes at a price. The overhead (in terms of sent packets) for achieving reliability was 128% for the configuration that was statically configured to use the reliability protocol. The total overhead for the dynamic configuration was 100% split in 60% for sending acknowledgement and retransmission packets and 40% for sending the protocol stack reconfiguration messages. This clearly shows that adaptive protocol stacks can reduce the total communication overhead in dynamic scenarios. However, the adaptation algorithm has to be designed carefully to avoid increasing the total overhead by sending too many stack reconfiguration messages.

We also measured the protocol stack reconfiguration time; that is, the time it takes from an event that triggers a reconfiguration until data can be sent over the new protocol stack. This is the sum of the time to determine and reconfigure the stack on both sides of the communication and the time to send the reconfiguration messages. We measured a protocol stack reconfiguration time of 806 $\mu$s whereof 286 $\mu$s were required for the transmission of the packets (round trip time).

*5.2. Self-Aware Hardware/Software Mapping.* To evaluate the self-aware hardware software mapping, we used two simple protocol stacks that are made up from functional blocks of type *privacy* or *security*. The functional block of type privacy implements an AES (advanced encryption standard [36]) encryption module that uses electronic codebook (ECB) mode. This encryption algorithm requires rather complex operations on the packet and can therefore be seen as a representative of all functional blocks that require heavy packet processing. We do not advocate using ECB mode, because it reveals patterns in plaintext, but it has the same processing characteristics as more advanced modes and is thus well suited for a performance evaluation. The functional block of type security implements a simple intrusion prevention system (IPS) by detecting non-shortest-form UTF8-encoding attacks [37]. This operation requires examining the whole payload once and is therefore representative of all functional

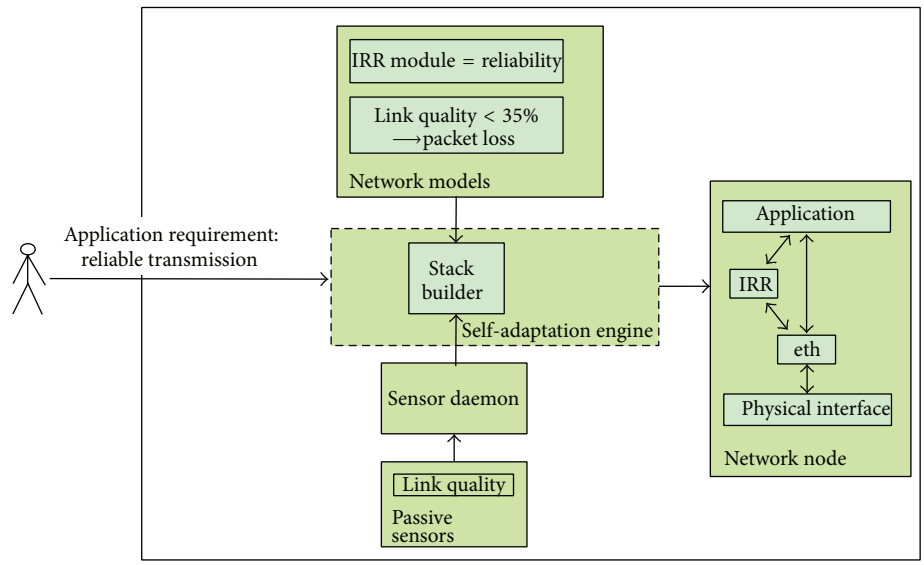


Figure 7: Implementation of the node architecture for the link-quality-aware protocol stack [6].

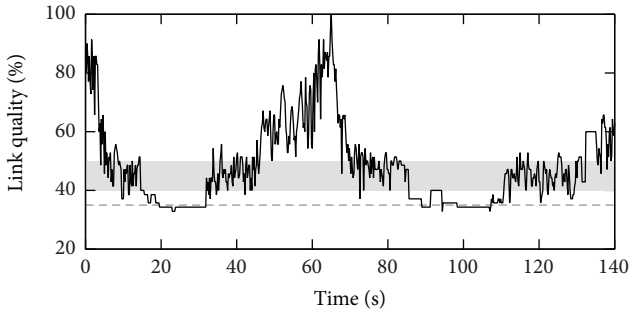


Figure 8: Measured link quality over 140 seconds. Packets got lost when the link quality was below the dashed line. The protocol stack is adapted when the graph crosses the gray bar [6].

blocks that require access to the whole network packet. This is often required, for example, for calculating a checksum.

The overall system consists of two applications. One application receives packets over a protocol stack that is built from the Ethernet and from the AES functional block; the other application receives packets over a protocol stack that is built from the Ethernet and from the IPS functional block. The Ethernet functional block is always mapped to hardware, whereas the applications are always in software in Linux user space. The AES and the IPS blocks can be mapped either to hardware or to software. However, only one of the two blocks can be mapped to hardware at a given time. The system is implemented on a Virtex-6 LX240T FPGA. The Linux operating system runs on a Microblaze CPU with a clock rate of 100 MHz with 16 kB instruction cache and 16 kB data cache. With these performance characteristics, this CPU is by orders of magnitude less capable than current workstation CPUs. The focus of this evaluation is not to show how our architecture compares with other computing platforms, but to show how different levels of self-awareness compare on our platform. To this end, we tested our system with three different mapping algorithms and three different network traffic mixes.

*5.2.1. Mapping Algorithms.* We have implemented the following three mapping algorithms M1–M3 that map functional blocks to either hardware or software.

M1 *Optimal static mapping*: this mapping is obtained by an engineer who has perfect knowledge. He knows how long it takes to process a packet in software and in hardware and he knows the traffic from each scenario. With this knowledge he builds an optimal but static mapping.

M2 *Simple self-aware mapping*: this mapping algorithm measures the number of packets per second that arrive in software for each protocol stack and puts the functional block that has to process more packets in hardware.

M3 *Smart self-aware mapping*: this mapping algorithm senses the number of packets per second for each flow and also senses how long it takes to process a packet for each protocol stack. As a decision basis it uses the moving average of the packet rate weighted by the packet processing time. Additionally it only changes the mapping if this number of one flow exceeds 110% of the other flow. Those measures should help to avoid fluctuations of the mappings during periods in which both applications receive similar amounts of traffic or during time periods with short spikes in the traffic.

For a fair evaluation of our self-aware network architecture, we compare the results of the optimal static mapping with the results obtained by the two self-aware mapping algorithms for varying scenarios.

*5.2.2. Network Traffic Mixes.* We have tested the three mapping algorithms with three different network traffic

mixes T1–T3, which are representative of network traffic seen by an end node (as opposed to an intermediate node such as a network router) in the Internet.

T1 *Nonoverlapping constant bit-rate traffic*: this traffic mix could correspond to a user who first streams a video that requires one protocol stack and after this he makes a phone call that requires the other protocol stack. We sent first 60,000 packets at a rate of 1,000 packets per second for the protocol stack with IPS and then we sent 60,000 packets at a rate of 1,000 packets per second for the protocol stack with AES.

T2 *Overlapping constant bit-rate traffic*: this traffic mix corresponds to a long running application which receives packets over a protocol stack with IPS configured and a short running application that requires packet encryption. We sent 120,000 packets with a rate of 1,000 packets per second to the protocol stack with IPS and in the middle of this transmission we sent 20,000 packets at a rate of 500 packets per second to the protocol stack with AES.

T3 *Congestion-controlled traffic*: in order to avoid overloading in the Internet, several protocols implement congestion control algorithms. Simplified, the traffic that results from those algorithms can be characterized as follows. The traffic rate linearly increases until the algorithm detects congestion. Then the traffic rate is reduced to half of the maximum traffic rate and then increased again until congestion is detected. This results in a typical sawtooth packet rate sequence. We sent 31,275 packets with a maximum packet rate of 200 packets per second to both protocol stacks (AES and IPS). Both traffic flows were shifted, so that the peak traffic rate for both flows alternates.

In all scenarios we send maximum sized Ethernet packets (1500 bytes).

*5.2.3. Mapping Results.* When a packet is processed in hardware, the hardware functional blocks (ETH, AES/IPS, and H2S/S2H) are pipelined and compute in parallel. The hardware/software interface is the slowest functional block in this pipeline. As a result the AES and IPS hardware modules do not affect the maximum packet throughput. Hence, our self-aware scheduler can focus on the software execution times, where the AES block takes 3.5 times as long as the IPS block in our scenario.

To get a rough idea of the speed difference, we have measured the packet processing times in hardware and software for both blocks, AES and IPS. The hardware implementation of the AES block is 150–400 times faster than the corresponding software implementation (depending on the packet size). The hardware implementation of the IPS block is 50–280 times faster than the corresponding software implementation (depending on the packet size).

Figure 9 gives an overview over the resulting hardware software mapping over time for the nine combinations of mapping algorithms and network traffic mixes.

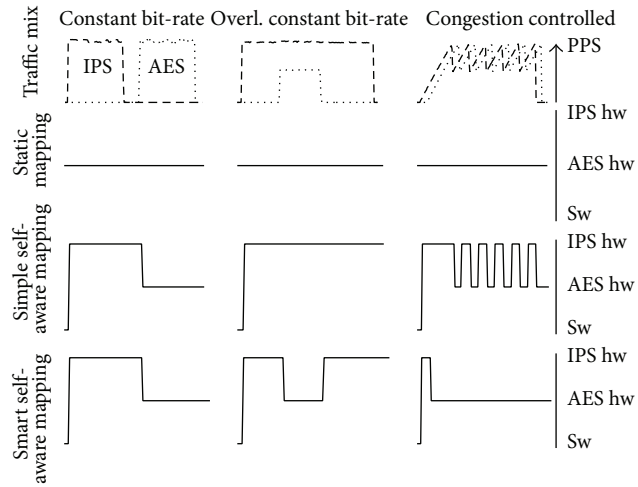


Figure 9: Traffic mixes and resulting hardware/software mappings for the three mapping algorithms over time. The static algorithm always puts the AES block in hardware. The dynamic algorithms start with no functional block mapped to hardware but adapt the mapping to the traffic.

The optimal static mapping implements the AES block in hardware and the IPS block in software. This can be explained by the fact that it takes the AES block 3.5 times longer to process a packet than the IPS block. The simple self-aware algorithm might lead to nonoptimal solutions since it does not take the time required to process a packet into account nor does it prevent frequent changes in the mapping, which lead to increased reconfiguration overhead.

In order to better evaluate the differences in the algorithms, we measured CPU utilization while processing packets (Figure 10) and the packet loss in percent for each traffic mix, for each application separately (Table 2) and also for the overall scenario (Figure 11).

We can see that the self-aware algorithms require the least CPU resources to process the network traffic mixes with constant bit-rate traffic. However, the simple self-aware algorithm requires the most CPU resources for the congestion-controlled traffic. This is because it reconfigures often, which requires CPU resources as well. This behaviour is also reflected in the highest packet loss for the simple self-aware algorithm for the congestion controlled traffic.

If the static algorithm and the smart self-aware algorithm are compared with each other, the self-aware algorithm always performs better. On average it requires 15% less CPU resources and loses 1.7% fewer packets.

In order to better understand how this result could be further improved, we analyzed the overhead introduced by the self-aware network system.

*5.2.4. Overhead Induced by Self-Awareness.* Additional overhead is introduced in order to implement our self-aware network system. For the self-aware hardware/software scheduler the overhead consists of the following parts.

Table 2: Measured packet loss for all combinations of mapping algorithms and network traffic mixes.

| Scheduling algorithm | T1: constant bit-rate | | | T2: overl. constant bit-rate | | | T3: congestion controlled | | |
|---|---|---|---|---|---|---|---|---|---|
| | Total | AES | IPS | Total | AES | IPS | Total | AES | IPS |
| M1: static | **4.5%** | 0.0% | 9.0% | **16.9%** | 14.5% | 17.2% | <**0.1%** | <0.1% | <0.1% |
| M2: simple self-aware | **1.4%** | 1.6% | 1.2% | **22.0%** | 52.0% | 16.8% | <**2.6%** | 2.2% | 0.9% |
| M3: smart self-aware | **1.8%** | 2.2% | 1.3% | **14.6%** | 20.8% | 13.6% | <**0.1%** | <0.1% | <0.1% |

Table 3: Overhead of partial reconfiguration.

| Modules | FPGA area | Bitstream size | Initial reconfiguration | Subsequent reconfigurations |
|---|---|---|---|---|
| IPS | 7% | 0.65 MB | 850 ms | 110 ms |
| AES or IPS | 15% | 1.3 MB | 1600 ms | 220 ms |

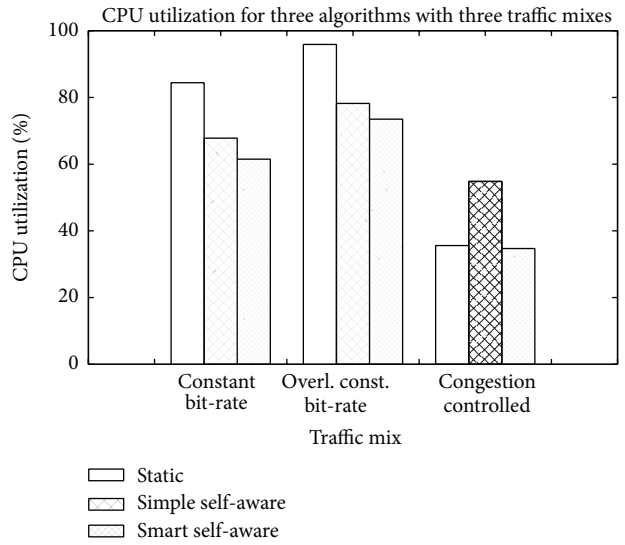


Figure 10: CPU utilization for all combinations of mapping algorithms and network traffic mixes.

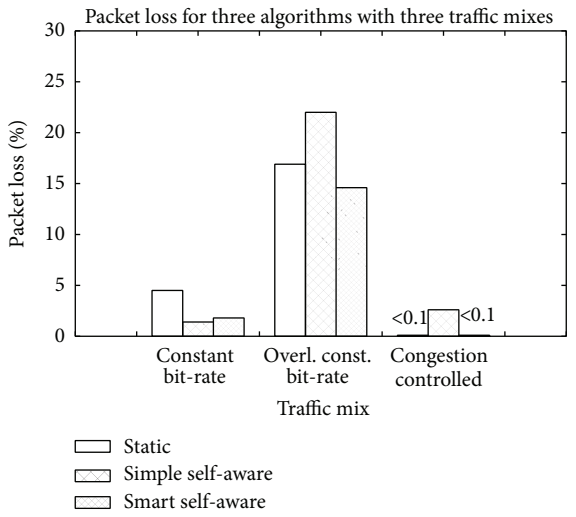


Figure 11: Packet loss for all combinations of mapping algorithms and network traffic mixes.

(i) *Sensors*: sensors need to be implemented which collect statistical data on the network traffic. This results in an enlarged code base which requires more RAM and longer execution time since the statistics need to be updated.

(ii) *Mapping algorithm*: in regular intervals the mapping algorithm needs to evaluate the statistical data and determine the new mapping. This results in an enlarged code base which requires more RAM and additional CPU overhead since the algorithm needs to be executed.

(iii) *Reconfiguration overhead*: If the mapping is changed, a partial reconfiguration of the FPGA has to be performed. During the reconfiguration, no packets can be processed in the area to be reconfigured. Additional time is required to transfer the state from hardware to software or vice versa.

In our system the dominant overhead is the partial reconfiguration of the hardware module (FPGA area) using the Internal Configuration Access Port (ICAP). The reconfiguration times for varying bitstream sizes are shown in Table 3. The partial bitstreams are stored on the compact flash memory. Hence, a bitstream file either has to be loaded from the compact flash disk, or it is already cached in the RAM, in which case the reconfiguration is considerably faster.

The reconfiguration times shown in Table 3 are considerably higher than those shown in related work. This can be attributed to the overhead introduced by the Linux operating system. For instance, in [38] they neither use an operating system nor are the bitstreams stored on an external memory. Additionally, they show that the HWICAP core provided by Xilinx can be optimized to perform an order of magnitude faster while not requiring CPU resources during the reconfiguration. Using their optimized core would also improve the reconfiguration time in our system, as well as the packet loss in our system.

## 6. Conclusion and Future Work

In this paper we have introduced a novel self-aware network architecture that uses dynamic protocol stacks which are composed of functional blocks. In contrast to network

communication in the current Internet with its fixed protocol stacks, we propose to use key words that define the communication requirements for an application. A self-aware stack builder then identifies a set of suitable protocol stacks that fulfill all requirements for specific network conditions. Two self-aware network nodes, which want to communicate, will select a common protocol stack in a negotiation phase, which respects the user-defined application requirements for the current networking condition. This protocol stack can be updated on the fly by both nodes whenever the network conditions change. Our EmbedNet platform implements the proposed self-aware network architecture on an FPGA, where the functional blocks can also be dynamically mapped to either a soft-core CPU or to reconfigurable hardware modules. A self-aware scheduler adapts the hardware/software mapping of the used functional blocks in order to minimize the overall packet loss and the CPU load caused by packet processing.

We have shown that self-awareness in the networking area can improve the performance with regards to several aspects. With the self-aware negotiation of the protocol stack itself, we can adapt the functionality provided by the protocol stack at run time. In a simple scenario with two commodity laptops we could show that we can reduce the number of packets required for data transmission by 28% for a given scenario compared to a static protocol stack (without changing the hardware/software mapping).

With the self-aware mapping of network functionality to hardware and software, we have shown that the CPU resources required to process packets as well as packet loss can be reduced as compared to a static system. Our experiments indicate that smart scheduling algorithms are required in order to obtain better results than optimal static mappings due to the high reconfiguration overhead of our EmbedNet platform. Compared to an optimal static mapping, our smart self-aware mapping strategy could reduce the CPU load by 15% and reduce the packet loss by 1.7% on average for three different traffic mixes for two applications.

In future work, we will develop more advanced self-aware mapping strategies and analyze our self-aware network architecture in scenarios with more concurrent applications, complexer protocol stacks, and reconfigurable hardware modules. The low performance of the soft-core MicroBlaze processor provides a major bottleneck for the overall system performance of our EmbedNet platform. Therefore, we plan to port EmbedNet to the Xilinx Zynq platform [39], which contains a more powerful dual-core ARM processor in addition to an FPGA. Finally, we will investigate techniques to reduce the reconfiguration overhead.

## Conflict of Interests

The authors declare that there is no conflict of interests regarding the publication of this paper.

## Acknowledgments

The research leading to these results has received funding from the European Union Seventh Framework Programme under Grant agreement no. 257906. This work was performed by Daniel Borkmann while affiliated with ETH Zurich.

## References

[1] G. Bouabene, C. Jelger, C. Tschudin, S. Schmid, A. Keller, and M. May, "The autonomic network architecture (ANA)," *IEEE Journal on Selected Areas in Communications*, vol. 28, no. 1, pp. 4–14, 2010.

[2] A. Keller, D. Borkmann, and W. Mühlbauer, "Efficient implementation of dynamic protocol stacks," in *Proceedings of the 7th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '11)*, pp. 83–84, Brooklyn, NY, USA, October 2011.

[3] A. Keller, B. Plattner, E. Lübbers, M. Platzner, and C. Plessl, "Reconfigurable nodes for future networks," in *Proceedings of the IEEE Globecom Workshops (GC '10)*, pp. 357–361, Miami, Fla, USA, December 2010.

[4] A. Keller, D. Borkmann, and S. Neuhaus, "Hardware support for dynamic protocol stacks," in *Proceedings of the 8th ACM/IEEE Symposium on Architectures for Networking and Communications Systems (ANCS '12)*, pp. 75–76, October 2012.

[5] A. Keller, D. Borkmann, and S. Neuhaus, "Towards self-adaptation in reconfigurable network nodes," in *Proceeding of the Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS '12)*, p. 1014, 2012.

[6] A. Keller, D. Borkmann, S. Neuhaus, and M. Happe, "Autonomic configuration of dynamic protocol stacks," in *Proceedings of the Workshop on Self-Awareness in Reconfigurable Computing Systems (SRCS '13)*, pp. 3–7, September 2013.

[7] D. L. Tennenhouse and D. J. Wetherall, "Towards an active network architecture," *Computer Communication Review*, vol. 26, pp. 5–18, 1996.

[8] A. T. Campbell, H. G. de Meer, M. E. Kounavis, K. Miki, J. B. Vicente, and D. Villela, "A survey of programmable networks," *ACM SIGCOMM Computer Communication Review*, vol. 29, no. 2, Article ID 505735, pp. 7–23, 1999.

[9] D. Tennenhouse, J. Smith, W. Sincoskie, D. Wetherall, and G. Minden, "A survey of active network research," *IEEE Communications Magazine*, vol. 35, no. 1, pp. 80–86, 1997.

[10] K. Psounis, "Active networks: applications, security, safety, and architectures," *IEEE Communications Surveys Tutorials*, vol. 2, no. 1, pp. 2–16, 1999.

[11] I. Hadzic, J. M. Smith, and W. S. Marcus, "On-the-fly programmable hardware for networks," in *Proceedings of the Bridge to the Global Integration (GLOBECOM '98)*, pp. 821–826, Sydney, Australia, November 1998.

[12] D. S. Decasper, G. Parulkar, S. Choi, J. Dehart, T. Wolf, and B. Plattner, "A scalable high-performance active network node," *IEEE Network*, vol. 13, no. 1, pp. 8–19, 1999.

[13] A. Dollas, D. Pnevmatikatos, N. Aslanides et al., "Architecture and application of plato, a reconfigurable active network platform," in *Proceedings of the 9th Annual IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '01)*, pp. 101–110, 2001.

[14] A. G. Fragkiadakis, N. G. Bartzoudis, D. J. Parish, M. Sandford, and A. Fragkiadakis, "Hardware support for active networking," in *Proceedings of the International Conference on Security and Management (SAM '03)*, pp. 27–33, Las Vegas, Nev, USA, June 2004.

[15] J. W. Lockwood, J. S. Turner, and D. E. Taylor, "Field Programmable Port Extender (FPX) for distributed routing and queuing," in *Proceedings of the ACM/SIGDA International Symposium on Field Programmable Gate Arrays (FPGA '00)*, pp. 137–144, ACM, New York, NY, USA, February 2000.

[16] T. Chaney, J. A. Fingerhut, M. Flucke, and J. S. Turner, "Design of a gigabit ATM switch," in *Proceedings of the 16th IEEE Annual Conference on Computer Communications (INFOCOM '97)*, pp. 2–11, April 1997.

[17] J. Shafer and S. Rixner, "RiceNIC: a reconfigurable network interface for experimental research and education," in *Proceedings of the Workshop on Experimental Computer Science (ExpCS '07)*, ACM, New York, NY, USA, June 2007.

[18] "DFG—Deutsche Forschungsgemeinschaft," http://www.dfg.de/.

[19] J. Teich, "DFG Schwerpunkt programm 1148 Rekonfigurierbare Rechensysteme," https://www12.informatik.uni-erlangen.de/spprr/.

[20] C. Albrecht, J. Foag, R. Koch, and E. Maehle, "DynaCORE—a dynamically reconfigurable coprocessor architecture for network processors," in *Proceedings of the 14th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing (PDP '06)*, pp. 101–108, February 2006.

[21] M. Meitinger, R. Ohlendorf, T. Wild, and A. Herkersdorf, "FlexPath NP—a network processor architecture with flexible processing paths," in *Proceedings of the International Symposium on System-on-Chip (SOC '08)*, pp. 1–6, Tampere, Finland, November 2008.

[22] J. W. Lockwood, N. McKeown, G. Watson et al., "NetFPGA—an open platform for gigabit-rate network switching and routing," in *Proceedings of the IEEE International Conference on Microelectronic Systems Education (MSE '07)*, pp. 160–161, Washington, DC, USA, June 2007.

[23] NetFPGA, 2011, http://netfpga.org.

[24] INVEA-TECH, "Combo fpga boards," 2011, https://www.invea.com/en/products-and-services/fpga-cards.

[25] R. Kokku, T. L. Riche, A. Kunze, J. Mudigonda, J. Jason, and H. M. Vin, "A case for run-time adaptation in packet processing systems," *ACM SIGCOMM Computer Communication Review*, vol. 34, no. 1, pp. 107–112, 2004.

[26] R. Kokku, U. Shevade, N. Shah, H. M. Vin, and M. Dahlin, "Adaptive processor allocation in packet processing systems," Tech. Rep., 2004.

[27] Y. Luo, J. Yu, J. Yang, and L. N. Bhuyan, "Conserving network processor power consumption by exploiting traffic variability," *ACM Transactions on Architecture and Code Optimization*, vol. 4, no. 1, article 4, 2007.

[28] M. Memik and W. H. Mangione-smith, "Nepal: a framework for efficiently structuring applications for network processors," in *Proceedings of the Network Processor Workshop in conjunction with Ninth International Symposium on High Performance Computer Architecture (HPCA-9)*, 2003.

[29] A. Raghunath, A. Kunze, E. J. Johnson, and V. Balakrishnan, "Framework for supporting multi-service edge packet processing on network processors," in *Proceedings of the Symposium on Architectures for Networking and Communications Systems ( ANCS '05)*, pp. 163–171, New York, NY, USA, October 2006.

[30] Q. Wu and T. Wolf, "Dynamic workload profiling and task allocation in packet processing systems," in *Proceedings of the International Conference on High Performance Switching and Routing (HPSR '08)*, pp. 123–130, Shanghai, China, May 2008.

[31] C. Kachris and S. Vassiliadis, "Analysis of a reconfigurable network processor," in *Proceedings of the 20th International Parallel and Distributed Processing Symposium (IPDPS '06)*, Rhodes Island, Greece, April 2006.

[32] Xilinx, "Microblaze soft processor core," 2013, http://www.xilinx.com/tools/microblaze.htm.

[33] XPS HWICAP, 2013, http://www.xilinx.com/products/intellectual-property/xps_hwicap.htm.

[34] E. Lübbers and M. Platzner, "ReconOS: multithreaded programming for reconfigurable computers," *Transactions on Embedded Computing Systems*, vol. 9, no. 1, article 8, 2009.

[35] "Netem," 2013, http://www.linuxfoundation.org/collaborate/workgroups/networking/netem.

[36] "Announcing the advanced encryption standard (aes)", in Federal Information Processing Standards Publication 197, November 2001.

[37] Unicode, "Unicode security considerations," Unicode Technical Report No. 36, 2013, http://www.unicode.org/reports/tr36.

[38] M. Liu, W. Kuehn, Z. Lu, and A. Jantsch, "Run-time partial reconfiguration speed investigation and architectural design space exploration," in *Proceedings of the 19th International Conference on Field Programmable Logic and Applications (FPL '09)*, pp. 498–502, September 2009.

[39] "Zynq-7000 All Programmable SoC," http://www.xilinx.com/products/silicon-devices/soc/zynq-7000/.