TOWARDS SELF-ADAPTATION IN RECONFIGURABLE NETWORK NODES

Ariane Keller, Daniel Borkmann, Stephan Neuhaus

Communication Systems Group
ETH Zurich
Gloriastrasse 35, 8092 Zurich
email: first.last@tik.ee.ethz.ch

1. INTRODUCTION

Today, computing nodes are everywhere, either visible as laptops and mobile phones or invisible as embedded devices in trains, home appliances or in the backbone of the Internet. The management of those devices becomes more and more difficult as the number of devices and the complexity of the applications increase. In order to cope with this complexity the idea of nodes with *self-**[1] features emerged.

In order to support self-* applications, Field Programmable Gate Arrays (FPGAs) are used as a basis to build (partially) reconfigurable hardware platforms. However, in order to use those platforms optimally, the software also needs to be redesigned. In previous work we developed the *Autonomic Network Architecture (ANA)* [1] which addresses the most important problems in the networking area: scalability, maintainability, and security. We also introduced a node architecture for ANA that is based on partially reconfigurable FPGAs and uses ReconOS [2] as its operating system [3]. The largest remaining challenge is to provide a self-aware algorithm that decides which protocols should be implemented in hardware and which in software.

In this paper we first briefly review the architecture of networking nodes (Section 2), and then tackle this problem by (i) establishing a reasonable adaptation frequency and classifying hardware/software mapping algorithms in the context of networking (Section 3), (ii) developing mechanisms to algorithmically recognize periodic traffic patterns that can be used as input to a mapping algorithm (Section 4); and (iii) developing a simulator for networked hardware/software systems that gives insights into the impact of different parameters (Section 5). The results will be used for further investigation in a self-aware hardware/software mapping algorithm for networking applications.

The research leading to these results has received funding from the European Union Seventh Framework Programme under grant agreement n^o 257906.

[1]Where "*" can be a word such as healing, configuring, adaptation, etc.

2. NETWORKING NODE

For the rest of this paper, we assume that network nodes have the architecture shown in Figure 1; we describe this architecture as well as a prototype implementation more fully in previous work [3]. The whole system is implemented on an FPGA: the operating system runs on a CPU that is configured into the FPGA, and a configurable number of *hardware slots* are connected to the CPU over a shared bus interface. Networking blocks executing in hardware are configured into those slots, and all slots are connected with a dedicated interconnect that offers line rate communication. In hardware, packets are processed in a pipelined fashion. The hardware/software interface is implemented as a combination of shared memory and interrupts.

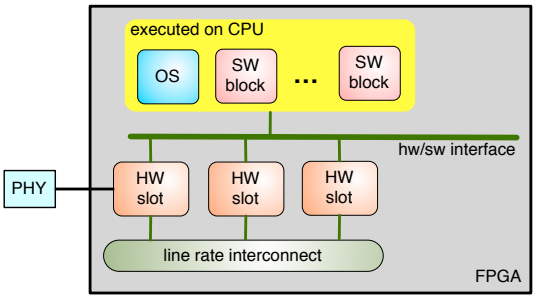

Fig. 1. Simplified node architecture.

3. ADAPTATION FREQUENCY

For the development of an adaptive system, it is crucial to determine an adequate adaptation interval. The underlying hardware limits the the maximum reconfiguration frequency, and hence the minimum time between reconfigurations, which can be computed as follows: if m is the time required to transmit sensor data to the reconfiguration application, c the time to compute the next hardware/software mapping, s_i the state relocation time in block i, r_i the reconfiguration time for block i, and n the number of blocks, the

total adaptation overhead A is then given by $A = m + c + \sum_{i=1}^{n}(s_i + r_i)$.

Adaptation overhead should be a small fraction of overall processing time so that the system spends most time processing traffic. We suggest limiting the adaptation overhead to 1% of the overall processing time. Applying these guidelines to the architecture we presented in [3], we obtain a maximum adaptation frequency of one adaptation per second. On end nodes, where traffic is user-dependent, this maximum frequency should be used so that the node remains responsive to traffic peaks. However, on nodes processing aggregated traffic (e.g., routers), a lower adaptation frequency should be used, in order to cope with seasonal effects [4] or trends [5]. This adaptation frequency could be set to one per hour.

4. ADAPTATION STRATEGIES

Regardless of the adaptation frequency, different strategies can be applied for the actual adaptation algorithm. Those strategies determine whether the actual traffic and/or learned traffic characteristics are taken into account and determine the actual optimization goal.

4.1. Traffic Dependent/Independent Algorithms

Traffic independent algorithms derive a mapping based only on the static properties of the required networking blocks. Such properties include the required FPGA area and memory footprint or the expected average benefit of a hardware implementation. The mapping only changes when new protocols are loaded. **Traffic dependent** algorithms take the actual traffic into account, e.g., by measurement of the number of packets and bytes processed by each block, the number of packets and bytes to be transmitted from hardware to software, and the utilization or energy consumption of the networking blocks.

4.2. Reactive versus Proactive Algorithms

A **reactive algorithm** makes an observation in time slot $t-1$, calculates an optimal mapping for this traffic distribution and uses this mapping in time slot t. This approach is especially useful for traffic changes introduced by a single user where the protocol and traffic mixes depend on user behavior, and which usually contain long periods of inactivity. However, we might also observe peaks in aggregated traffic, e.g., for protocols that are rarely used such as IPSec in the MAWI network [6].

A **proactive algorithm** changes the hardware/software mapping *before* the traffic mix changes. This requires knowledge from past traffic mix changes that can be applied to the current situation. On end nodes this might be a regular pattern for checking emails or storing a backup to a server. On intermediate nodes this might be the variation of network traffic that occurs on a daily basis.

We can distinguish between algorithms that *know the period* and algorithms that *learn the period*. Algorithms that know the period could assume for example that tomorrow's traffic will be much like today's. This approach is particularly useful for aggregated traffic.

End nodes may require a more sophisticated technique such as the *partial autocorrelation function* (PACF). For a given lag ℓ, the PACF is the correlation of the traffic at time $t-\ell$ with the traffic at time t, considering possible linear dependencies due to lesser lags; the significance level of this autocorrelation, i.e., the probability that the observed correlation will be this large by chance, can also be computed.

In Figure 2, a repetitive pattern is clearly visible for email traffic on an end node. In periodic signals, the PACF will be significant at lags that are multiples of the period: in the PACF for this traffic trace, shown in Figure 2, the correlations are higher then the dashed line at lags 60, 120, 180, and 240 seconds ($p = 10^{-4}$), suggesting a period of one minute. Not surprisingly, the email client on the end node checks for new emails every minute.

There are also significant ($p = 0.05$) *negative* autocorrelations just before and after these large peaks. This means that traffic just before and just after a peak is likely to be much less, and resources required for handling the periodic traffic can be freed up immediately after the peak.

In a similar analysis for a protocol without periodic traffic, the PACF did not show any significant periods.

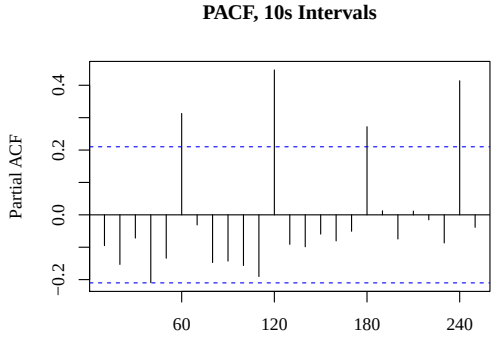

Fig. 2. Partial autocorrelation of IMAP traffic, x is lag, dashed line is significance level 10^{-4}.

4.3. Optimization Goals

Regardless of its type, an algorithm will optimize the mapping for a certain goal. Typical goals are maximizing throughput, minimizing energy, or providing sustainability (e.g., by avoiding mappings that could damage the FPGA over time). For the throughput optimization goal we can distinguish two classes of algorithms.

Maximisation of hardware benefits algorithms put those blocks into hardware that offer the largest benefit of execution in hardware when compared to execution in software. It is assumed that the algorithm has access to information on how long it takes to process a packet of a given size in software and in hardware. For a traffic-independent mapping, the blocks with the largest differences are put in hardware; for a traffic dependent mapping, the differences are multiplied by the number and length of the actual packets.

Protocol graph partitioning algorithms minimize the overhead introduced by transmitting packets between hardware and software. Therefore, they only map *connected subgraphs* of the original protocol graph to hardware. One of those blocks is the block that is physically connected to the network interface. The subgraph mapped to hardware can be selected based on several criteria. For example, we could choose the subgraph with the aggregated largest difference in execution time, or the one that minimizes the load on the hardware/software boundary. Different hardware subgraphs might introduce different loads due to the following reasons:

- one block is a firewall or an intrusion prevention system that drops packets on purpose;
- one block offers routing, sending packets from the receive code path to the transmit code path;
- the packets are dropped due to buffer overflows and hence this subgraph needs more resources.

5. EVALUATION

In order to experiment with different parameters of the underlying hardware we have developed a simulator that can process network traces and execute different algorithms on these traces.

5.1. Simulator

The simulator models the whole system consisting of software and hardware parts. Therefore it needs to simulate real hardware parallelism. We have implemented the simulator in SystemC [7] which is a set of C++ classes and macros that allows the simulation of concurrent processes.

Our simulator models the system described in Section 2, and consists of the following building blocks:

- a CPU, shared among all blocks executed in software;
- several hardware processing units, each capable of hosting a networking block;
- an interconnect between the hardware units, offering line rate communication;
- a communication bus between hardware and software with limited bandwidth;
- a monitoring framework that collects the number of packets and bytes transmitted between the blocks;
- a hardware reconfiguration interface.

The simulator differs from the actual system as follows:

- Instead of receiving real network packets from a physical device, the simulator reads a captured packet trace. For each packet the following information is stored: packet id, arrival time, packet length, protocols. For the packet trace generation we use packet traces captured by libpcap [8] or netsniff [9] that were converted with the help of the pcap decoder provided by yaf [10].
- Since the packets only contain protocol information but not the actual data, the blocks only specify a processing time per packet (to simulate header processing), a processing time per byte (to simulate payload processing), a reply rate (to simulate reliable traffic) and a drop rate (to simulate firewalls etc.).
- The monitoring framework can only obtain packet and byte counters but no physical parameters such as FPGA temperature or energy consumption.
- The reconfiguration overhead is not modelled.

5.2. Results

We have implemented four different mapping algorithms in the simulator that was configured with three hardware slots. To obtain a realistic protocol mix, packet length and packet interarrival times, we captured a packet trace on a notebook in a university network and included all packets that were either broadcast packets or packets addressed to the collecting node into the trace.

We have evaluated the following four algorithms:

- **SW only**: put only those elements in hardware that do not have a software implementation. Put the others into software.
- **Maximum Benefit**: put those elements into hardware that benefit most from a hardware implementation (depending on the number of packets and bytes processed).
- **Minimum Bandwidth**: minimize the number of packets to be sent from hardware to software.
- **Hardware Cluster**: put those blocks in hardware that offer the best hardware benefit and that are connected (in order to avoid sending packets unnecessarily between hardware and software).

We evaluated those algorithms in the following systems:

- **Exp A.** Ethernet is implemented in hardware only, ARP, IPv4, and icmp are implemented in hardware and in software, all the other protocols are implemented in software only. There is no cost associated with sending packets from hardware to software.
- **Exp B.** Additionally, TCP, UDP and tls get a hardware implementation. There is no cost accociated with sending packets from hardware to software.
- **Exp C.** Same as B, but now there is a cost for sending packets between hardware and software, which is proportional to the packet length.

Figure 3 shows one particular hardware/software mapping of the Maximum Benefit algorithm for the Exp B sce-

nario. Nodes represent protocol blocks and edges represent the number of packets sent between the blocks. The colored nodes represent the nodes to be mapped to hardware.

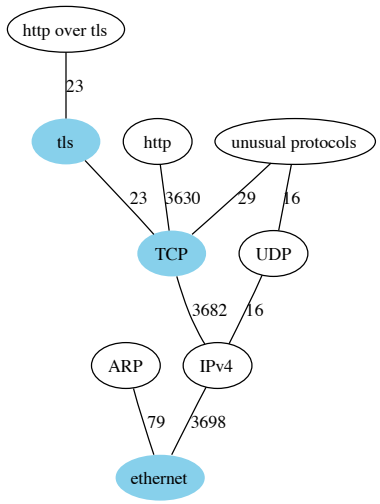

Fig. 3. One particular hardware/software mapping of the Maximum Benefit algorithm for the Exp B scenario.

Table 1 shows the results of the four algorithms in the three different scenarios. However, instead of looking at the individual results, we focus on the *differences* between the algorithms in the different scenarios.

For the SW Only and the Minimum Bandwidth algorithm, the performance does only depend slightly on the scenario. For the Maximum Benefit and the Hardware Cluster algorithm, the performance increases significantly, when more protocols have a hardware implementation. As expected, the Maximum Benefit algorithm offers the best performance. When we introduce a cost for crossing the hardware/software boundary the performance decreases again; however, it decreases less for the Hardware Cluster algorithm so that this algorithm offers now the best performance.

It is also interesting to see that the Hardware Cluster algorithm requires about half the number of reconfigurations than the Maximum Benefit algorithm. This is especially interesting since in the actual hardware, reconfiguration requires time, in which a slot cannot process packets.

From those results we learn that finding the most efficient algorithm heavily depends on the actual parameters and in a real live scenario probably a combination of the different algorithms is required.

6. CONCLUSION AND FUTURE WORK

We have presented an application for self-awareness in reconfigurable computing systems, namely the mapping of network protocols to either hardware or software. To this end an adaptation algorithm is required that takes both network traffic characteristics as well as hardware characteristics into account. We have presented a first classification of such algorithms and developed a simulator that can be used to obtain initial performance results. As a next step we will develop new algorithms for the hardware/software mapping.

Table 1. Performance Comparison

	Exp A	Exp B	Exp C
SW Only Algorithm			
different configurations	1	1	1
reconfigurations	0	0	0
packet drop rate	32.9 %	32.9	33.5%
Maximum Benefit Algorithm			
different configurations	3	9	9
reconfigurations	41	2266	2271
packet drop rate	32.9%	2.7%	12.1%
Minimum Bandwidth Algorithm			
different configurations	2	2	2
reconfigurations	1	1	1
packet drop rate	32.9%	32.9%	33.5%
Hardware Cluster Algorithm			
different configurations	3	4	4
reconfigurations	39	1006	1006
packet drop rate	33.7%	4.8%	11.1%

7. REFERENCES

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

[2] E. Lübbers and M. Platzner, "ReconOS: An RTOS supporting hard- and software threads," *IEEE Int. Conf. on Field Programmable Logic and Applications*, 2007.

[3] A. Keller, B. Plattner, E. Lübbers, M. Platzner, and C. Plessl, "Reconfigurable nodes for future networks," in *GLOBECOM Workshops (GC Wkshps), 2010 IEEE*, dec. 2010, pp. 357 – 361.

[4] K. Thompson, G. J. Miller, and R. Wilder, "Wide-area internet traffic patterns and characteristics," *IEEE Network*, vol. 11, pp. 10–23, 1997.

[5] S. McCreary and kc claffy, "Trends in wide area ip traffic patterns - a view from ames internet exchange," in *ITC Specialist Seminar*, Monterey, CA, Sep 2000.

[6] "MAWI Working Group Traffic Archive," http://mawi.wide.ad.jp/mawi/ (Dec 11).

[7] "SystemC," http://www.systemc.org (Dec 11).

[8] "tcpdump and libpcap," http://www.tcpdump.org/ (Aug 12).

[9] "netsniff-ng," http://www.netsniff-ng.org (Aug 12).

[10] C. M. Inacio and B. Trammell, "Yaf: yet another flowmeter," in *Proceedings of LISA'10*. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–16.