Hyper-Programmable Architectures for Adaptable Networked Systems

Gordon Brebner, Phil James-Roxby, Eric Keller and Chidamber Kulkarni
Xilinx Research Laboratories
gordon.brebner@xilinx.com

Abstract

We explain how modern programmable logic devices have capabilities that are well suited for them to assume a central role in the implementation of networked systems, now and in the future. To date, such devices have featured largely in ASIC substitution roles within networked systems; this usage has been highly successful, allowing faster times to market and reduced engineering costs. We argue that there are many additional opportunities for productively using these devices. The requirement is exposure of their high inherent computational concurrency matched by concurrent memory accessibility, their rich on-chip interconnectivity and their complete programmability, at a higher level of abstraction that matches the implementation needs of networked systems. We discuss specific examples supporting this view, and present a highly flexible soft platform architecture at an appropriate level of abstraction from physical devices. This may be viewed as a particularly configurable and programmable type of network processor, offering scope both for innovative networked system implementation and for new directions in networking research. In particular, it is aimed at facilitating scalable solutions, matching differently resourced programmable logic devices to differing performance and sophistication requirements of networked systems, from cheap consumer appliances to high-end network switching.

1. Introduction

In this paper, we consider the implementation of what we term “adaptable networked systems”. These are systems that have communication and networking as their dominant activities, in contrast to traditional computer systems, that have computation as their dominant activity. Examples include simple sensors and actuators, simple clients and servers. The adaptability covers both design time, where one may wish to assemble different variants of systems for differing environments, and use time, where one may wish to modify a system’s behavior in response to changes in its environment. The setting for these systems is of ‘ambient intelligence’, the ‘disappearing computer’ and ‘pervasive computing’, to deploy a few futurological cliches. More concretely, such systems fit with Turner’s vision of *extensible networking* [12], where he advocates a shift to make networks ‘application-centric’ rather than ‘technology-centric’, for example, for supporting applications such as near video-on-demand, generalized firewalls and distributed interactive simulation.

We argue that the capabilities of modern programmable logic devices (PLDs) are well suited as a basis for implementing adaptable networked systems, through the efficient delivery of what we term “hyper-programmable architectures”. By “hyper-programmable”, we mean that all aspects of the architecture are configurable, not just that the architecture supports a programming model. In particular, we wish to establish that PLDs can be viewed as the primary, perhaps exclusive, components of adaptable networked systems. This means not only demonstrating that PLDs have apt capabilities, but also devising new use models and tools to unleash these capabilities.

At the present time, there is a conventional spectrum of implementation technologies for networking functions, ranging from custom ASICs, via the use of ASSPs (application specific standard parts) and ASIPs (application specific instruction processors), particularly network processors, to the use of generic processors. Although much used in practical networking, PLDs tend to feature in substitutive or supporting roles. For example, in most application domains, PLDs now feature as substitutes for ASICs, reflecting the increasing capabilities of PLDs, the rapidly increasing non-recurring expenses of ASICs and the need for faster times to market. PLDs also feature as ASSP substitutes in networking, performing specialist data plane functions, notably physical interfacing, and also framing, lookup and scheduling support, for example. Finally, the most historic role of PLDs is for implementing arbitrary glue logic within systems. Notwithstanding all of these current uses, we seek to position PLDs additionally as hyper-programmable alternatives to processors.

Our motivation for advocating the benefits of PLDs is not just for improving performance, though this is the normal reason for their use at present. In line with Turner's proposal for moving the major focus of networking research from improving performance to enabling advanced network services [12], we also seek other benefits. These include latency minimization, reduced power consumption, network and system configurability and adaptability, and simplified design and manufacturing.

In Section 2 of the paper, we introduce and define *message process*ing as the domain of interest for our systems. In Section 3, we give a brief overview of the capabilities of modern PLDs. Then, in Section 4, we give an detailed exposition of our prototype environment for enabling message processing using PLDs. Section 5 contains conclusions and directions for future work.

2. Message processing

We use the term "message" here to describe a discrete block of information, thus generalizing over common terminologies such as "cell", "datagram", "data unit", "frame", "packet", "segment", "slot" and "transfer unit". We do not intend, indeed we hope to avoid, confusion with special-case uses of the word, as in "email message" and "instant message" for example.

In turn, *message processing* (MP) refers to the functions performed by our adaptable networked systems. This phrase generalizes over terminologies such as "network processing" and "packet processing". Reflecting the change of balance from domination by computation to dominance by communication, we see future system architectures as being *message-centric*. That is, control flow follows the arrival of messages at interfaces, through their processing, to their departure from interfaces.

We can categorize types of message processing, loosely following an earlier classification of Shah [10], as:

- Matching and lookup functions, which are read-only on messages, with the results being used for control;
- Simple manipulations (that can be combined), which are read/write on specific message fields;
- Complex pre-canned algorithms, which are read/write on messages or specific message fields; and
- Message marshalling – the movement, queuing and scheduling of messages.

The final category gives a scaled-down networking flavor to the message processing system architecture.

We position MP as a domain that is in various senses intermediate between the well-established domains of digital signal processing (DSP) and data processing (DP).

Table 1. Comparison of DSP, MP and DP

	DSP *Stream-based*	**MP** *Message-based*	**DP** *Processor-based*
Dominant system flow	Synchronous data flow	Asynchronous data flow	Control flow
Raw data complexity	Numerical values	Nested structs, but no iterators	Complex data types
Input/output relationship	Size similar; complex ops	Size similar; simple ops	Size dissimilar; complex ops
Scope for concurrency	High	High-medium	Low
Randomness of data access	Low	Low-medium	High

Table 1 presents a summary comparison between the characteristics of DSP, MP and DP. In presenting this comparison, we wish to point out that MP, as we are viewing it, has many attributes in common with DSP, a domain where PLDs are already employed to great benefit as first-order system components.

Our challenge is to supply efficient support for data with more complex types but with simpler operations, and for rather less structured concurrency and data access. For example, the pipelined architectures that are successful for DSP look less attractive for MP. However, we assert that this can be addressed successfully because we do not have to deal with the more general complications associated with DP, such as arbitrary computations, unpredictable concurrency and unpredictable data access. Thus, it is possible to exploit computational locality and independence to abstract the basic attributes of a PLD to a level that is useful for practical message processing, achieving 'impedance matching' without introducing complex mapping or support mechanisms.

3. Programmable logic devices

3.1. Technology overview

The term "programmable logic devices" encompasses a wide range of chips that can be programmed by the user to perform specific logic functions. The earliest, simplest and cheapest types include programmable array logic devices (PALs) and programmable logic array devices (PLAs), which place restrictions on the structure of implemented logic circuitry. Our main focus here is the more general field programmable gate array (FPGA). The basic device architecture consists of an array of similar programmable logic elements interfaced to interconnection elements. The largest current devices now contain the equivalent of several million programmable logic gates, enough to implement substantial designs.

However, in recent years, the basic logic array architecture has been evolving into a more general programmable system on chip architecture, with the emergence of the *platform FPGA*. Such architectures contain a pre-defined mix of very different types of elements. Today, for example, the Xilinx Virtex-II Pro platform FPGA [14] contains configurable logic blocks as would be expected, plus input/output blocks supporting many different input/output standards, multi-gigabit transceivers, distributed embedded memories, digital clock managers, dedicated multiplier blocks and embedded PowerPC 405 processors. These are embedded in a rich configurable interconnection network.

Modern FPGAs and platform FPGAs present both unique design challenges and unique design opportunities. The prevalent design style tracks the ASIC/SoC (system on chip) design style, with hardware descriptions being synthesized, placed and routed, finally generating FPGA programming data rather than mask data. Where soft or hard embedded processors are included, the prevalent hardware-software co-design style is for scaled down system-on-board architectures, in which a processor is a central component with subordinate functions in logic.

It is possible to harness programmable logic devices for networking applications using these design styles, indeed the vast bulk of PLD penetration has resulted from this approach. PLDs can be used as substitutes for ASICs or ASSPs within board designs. More ambitiously, for example, one could implement particular single-chip network processors on a PLD by using something like the StepNP environment of Paulin et al [8], mapping to an FPGA instead of an ASIC as the physical target.

We believe, however, that many more opportunities lie ahead, by delivering modern PLD attributes directly to networked applications, without the constraints of interposing conventional hardware and/or software design flows. For example, in comparison with a network processor, it is not necessary to pre-define a fixed number of microengines with fixed functions, thus avoiding mapping conundra for higher-level tools. Indeed, it is not necessary to pre-define internal bus and memory architectures either, thus avoiding bottlenecks of the sort identified by Kulkarni et al [5].

To this end, we show how this aspiration can be achieved, by presenting our new highly-programmable message processing design environment for PLDs in the next section. However, before presenting this general framework, we first briefly review three important prior research activities that illustrate how PLDs can be harnessed as first-order components of networked systems, in these cases through special-case implementation using standard tools. In addition, during the period 2002-2004, there has been a significant amount of published research on PLDs used for the specific network application area of firewalling and intrusion detection, where the pattern matching ability of PLDs is particularly exploited.

3.2. Selected prior research on PLDs for programmable networked systems

In 1997, Hadzic and Smith [3] reported a dynamically reconfigurable FPGA-based architecture, the Programmable Protocol Processing Pipeline (P4), a platform for highly flexible implementation of *protocol boosters*, functional elements inserted and deleted from protocol stacks on an as-needed basis. This was a pioneering instance of FPGAs used for networking in a flexible manner that allows for reprogramming at run time.

In 2000, Lockwood et al [6] reported the Field Programmable Port Extender (FPX), a two-FPGA module placed between a line card and a switch fabric, with one FPGA programmable dynamically via control cells sent over the network. Since then, as published variously, the FPX has been used successfully as an open platform supplying reprogrammable hardware for a wide variety of networking functions.

In 2002, Brebner [1] reported a gigabit-rate Multi-version IP Router (MIR) targeting the Xilinx Virtex-II Pro platform FPGA, harnessing all of its on-chip capabilities. This introduced the concept of concurrent threads in logic, allowing generalization beyond the pipeline-style architectures of Smith and Lockwood, and PLD use in general. Also, there was speculative thread execution and speculative packet movement, and a logic-centric approach to harnessing an embedded processor. A particular benefit of the overall approach was enabling very low latency paths for message processing, by exploitation of all opportunities for concurrent execution. This work has been a significant influence on the features supported in our generic design environment.

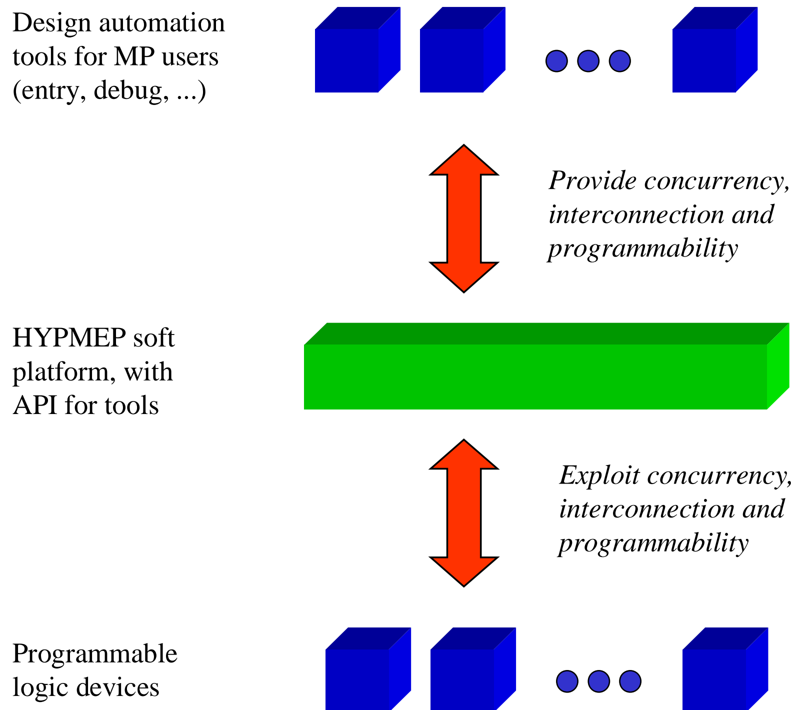

Figure 1. HYPMEP environment, with soft platform as central component

4. The Hyper-programmable MP (HYPMEP) environment

Figure 1 presents the top-level view of our framework for mediation between domain-specific design automation tools for message processing and programmable logic devices (PLDs). The centerpiece is the HYper-Programmable MEssage Processing (HYPMEP) 'soft platform' – a highly programmable microarchitecture. As indicated by the annotations on the figure, HYPMEP is designed to exploit, to the full, the concurrency, interconnection and programmability inherent to the PLD, and deliver this at a higher level of abstraction to the design automation tools.

To derive full benefit from underlying PLD technologies, we envisage that the high-level tools have a domain-specific flavor, beyond conventional design in terms of just hardware description languages and/or high-level programming languages. One long-standing example of a domain-specific description technique with supporting tools is SDL [9], used for almost three decades in telecommunications. Some recent examples, some targeted at emergent network processor technologies, are CloudShield's RAVE [2], MIT's Click [4], Novilit's Anyware [7] and Teja's Teja C [11]. We are supplying a relatively tool-neutral mapping point for such tools, abstracting the most applicable properties of an underlying PLD.

We also cater for a range of capabilities from the underlying PLD technologies, down from rich platform FPGAs that include embedded processors and memories, through basic FPGAs, down to more constrained programmable logic arrays. To remain consistent with the 'impedance matching' role of HYPMEP, we anticipate that, in general, the complexity of a high-level design description will broadly match the complexity of the targeted underlying PLD device.

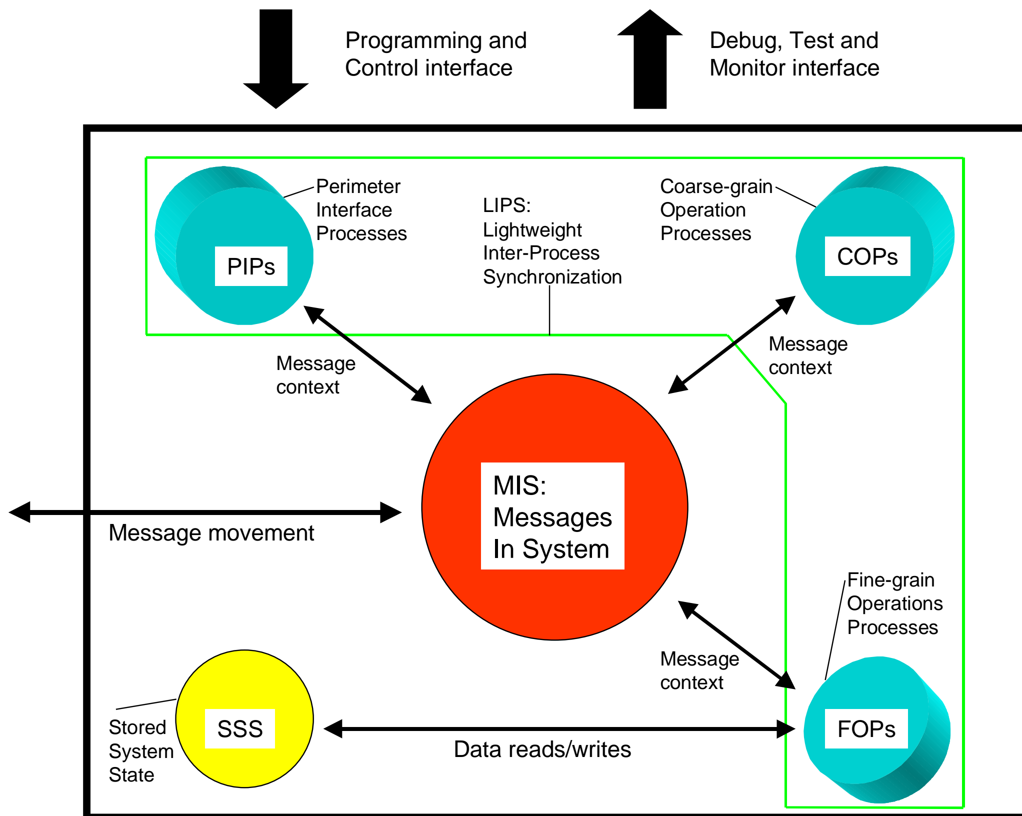

Figure 2. Microarchitecture of the HYPMEP soft platform

4.1. The MIS: Messages in System

The overall microarchitecture of the HYPMEP soft platform is shown in Figure 2. The central feature is the MIS: Messages In System. This is the logical storage point for all messages currently present within the system (or subsystem) implemented using the soft platform. The MIS is interfaced to the enclosing environment through the perimeter of the soft platform, allowing input and output of messages. To take a concrete example, if the system was a packet router, the contents of the MIS would be all packets currently in flight through the router. The external interfaces would be through those ports by which the router is connected to physical networks.

Note that the MIS is a *logical* architectural component, and does not imply physical implementation using a centralized memory device. The availability of large numbers of distributed multi-port memories on chip, combined with vast amounts of interconnect, enables new use models where the typical implementation will involve a complex ensemblage of distributed memories, together with subtle interconnection both between memory elements and between memory elements and input/output interfaces. A particular feature of this soft platform is that the nature of the MIS – that is, in implementation terms, the types, sizes and interconnection of memory elements, as well as the interface to such memory elements – is programmable, giving a highly flexible architectural framework that allows full harnessing of on- or off-chip capabilities to address particular system requirements. In addition, in the logical view of the soft platform, the MIS is highly decoupled from the computational elements, enabling powerful analysis and hence optimization of this particular system component.

In line with the message-centric nature of the soft platform, the model for message processing is that messages arrive at, reside in, and depart from, the MIS, and that different types of processes operate upon these messages, typically concurrently. There are three types of process. Conceptually, the processes come to the messages to do work, in contrast to say a pipeline where messages would come to the processes. Our analogy is that the MIS is similar to the pits at a motor racing circuit, with each resident message corresponding to a racing car. The processes correspond to mechanics working on a car (e.g. refueling or changing tires), with high concurrency and low latency. This basic model is a key feature of the microarchitecture, which distinguishes it from traditional processor/memory architectures on the one hand and pipeline architectures on the other hand. The model provides the smooth linkage needed between a message-centric model of the application space and a model of the PLD-based implementation space that focuses on the external interfaces.

4.2. Implementation of the MIS-based processing model

The actual implementation of the abstract MIS model is likely to have a very different flavor, involving subtle interfacing between MIS memory elements and the elements implementing the processes. As well as scope for concurrency from allowing several processes to operate on one message simultaneously, there is also scope for concurrency from sets of processes operating on several messages at the same time.

One real implementation of the model is as follows. First, note that each single execution of a process from start to finish is associated with one message, as indicated by the arrows labeled "Message context" in the figure. A process may be implemented directly in programmable logic, or in a soft or hard embedded processor. In whichever case, there is a particular register termed the Data Counter (DC) register. The DC points to the current position in the current message that the process is operating upon, and so establishes the 'message context'. In essence, the DC is a message-centric analog of the Program Counter (PC) - as a process executes, the DC is updated, either automatically to advance to the next position in a message, or by execution of programmed 'data jumps'. Thus, the process conceptually moves over the message.

In the interests of avoiding ambiguity, it should be noted that a notion of a 'data counter' has existed prior in the pipelined/systolic style literature, but there it refers to a design-time scheduled address generator for selecting data from a memory as input to a fixed computational fabric. Our notion is different from this. The necessary implementation requirement for our DC mechanism is using the address in the DC to access the correct stored message component. In practice, depending on the optimized memory organization being used for a particular configuration of the soft platform, the required value may be stored very locally (e.g. in a register), more distantly (e.g. in on-chip RAM) or, in multi-chip systems, in external RAM. The address in the DC register is used to drive multiplexers, or otherwise, to form a path between the implementation of the process and the appropriate memory unit, then to select within the memory unit.

4.3. The Lightweight Inter-Process Synchronization (LIPS) mechanism

All three types of processes are intended to be lightweight in nature with a common interface within the soft platform. Importantly, the word "process" here denotes just a concurrent agent, and carries no imputation of the characteristics of a traditional operating system supported process, or indeed a thread for that matter. The Lightweight Inter-Process Synchronization (LIPS) mechanism is provided to allow limited interaction between processes.

Typical interactions might include creating or destroying another process, starting or stopping another process, suspending or interrupting another process, signaling status to another process, or passing a small amount of data to another process. Such a mechanism is adequate to support a degree of control flow in the processing of a message, in addition to the basic underlying concurrency. We discuss the nature of this control flow further in Section 4.7.

4.4. Fine-grain Operations Processes (FOPs)

A Fine-grain Operations Process (FOP) is the basic programmable unit for message processing. Operating on a single message, such a process performs a sequence of steps, each step typically requiring one clock cycle, performing a number of concurrent simple operations at each step. If a Data Counter (DC) mechanism as outlined above is used, then it would be the case that each FOP has its own DC, with the DC normally being incremented at each step, unless it receives a new value through execution of a 'data jump' operation. Examples of operations include inspecting a field (for example, a 16-bit header field) of a message, or performing simple arithmetic (for example, adding one to a 16-bit header field) of a message. The set of allowable operations is targeted at message processing and may be defined statically as a permanent feature or defined dynamically as part of the configuration of the soft platform. The latter represents an addition level of hyper-programmability.

Implementation of a FOP can be in programmable logic, for example as a finite state machine, or on an embedded processor, for example as an operating system thread. Each FOP also has access to the Stored System State (SSS), allowing context to be preserved for a single message, over several messages, or over the system lifetime, for example. One example of a FOP in an Internet Protocol packet router would be to parse the IP header of a packet to classify it for further action, retaining extracted information in the SSS. Another example would be using the SSS to retain per-flow state information for a particular IP packet flow. Note that, like the MIS, the SSS is a logical architecture component, and does not imply physical implementation using a centralized memory device.

4.5. Coarse-grain Operation Processes (COPs)

A Coarse-grain Operation Process (COP) is used to incorporate a function block for some message processing operation, the block being designed using any non-FOP methodology. For example, such a block may already exist as a piece of reusable logic or software, or it may be created to implement a function that is awkward or impossible to describe as a FOP. Examples of COP functions might include compression or encryption of all or part of a message, as well as auxiliary functions such as address lookup or more general message classification.

The essence of a COP implementation is to provide a smart adapter between the interface presented by the functional block and the common interface presented by processes in the soft platform. In particular, this means that a COP may be started, stopped or interrupted by another process; typically, a COP would not actively start, stop or interrupt another process itself. COPs may be in existence permanently, but might also be instantiated or destroyed dynamically, to allow dynamic reconfiguration of functional blocks to reflect an evolving environment enclosing the networked system. If a Data Counter mechanism is used, then it would be the case that each COP has its own DC, which points to the beginning of the message to be processed, but which is not adjusted during execution of the COP.

4.6. Perimeter Interface Processes (PIPs)

A Perimeter Interface Process (PIP) is similar in nature to a COP, in that it is also used to incorporate a function block in an entirely analogous manner. However, functionally, a PIP is specifically concerned with enabling the movement of a message over the perimeter of the soft platform, between the enclosing environment and the MIS. (This is in contrast to a COP, which is concerned with performing an operation on a message that is in the MIS.)

Typical PIP functions are receiving or transmitting successive words of a message over an interface, using the protocol defined for the interface. For example, a PIP may act as the interface to a reusable gigabit ethernet MAC core. However, note that the "enclosing environment" need not necessarily mean direct input/output to the outside world. It may include other system components implemented on the programmable logic device, for example some other type of soft platform targeted at a different domain such as streaming multimedia processing.

4.7. Process targeting, and control flow

Taken together, the three types of process provide a practical basis for the soft platform. The FOPs are intended as the main target for third-party tools, which generally describe message processing in terms of simple, sometimes domain-specific, operations. The COPs provide a means for inclusion of library elements from a multiplicity of sources. The PIPs provide the necessary basic interfacing at the perimeter of the soft platform.

There is control flow associated with individual messages, which arises from one process's execution being followed by another process's execution. The control flow begins with a PIP on the arrival of the message. The PIP then initiates one or more FOPs or COPs, and FOPs in turn can initiate further FOPs or COPs. Ultimately, this web of control flow ends with a PIP on the departure of the message.

To differentiate the overall ensemblage from the product of a standard general-purpose block-based hardware synthesis procedure, three key features are: (a) the existence of FOPs as programmable concurrent elements; (b) the existence of the LIPS as a lightweight synchronization mechanism between the elements; and (c) the presence of the MIS as the central focus of the system.

4.8. Programmability via API

A further key feature, justifying the adjective "soft" in "soft platform", is the interfaces labeled "Programming and Control interface" and "Debug, Test and Monitor interface" in Figure 2. These interfaces correspond to the API for the soft platform. Programming information may include both structural information related to soft platform features, as well as behavioral information such as functional or quality requirements. The level of abstraction of such information affects the level of complexity required in mapping the API to and from the soft platform attributes. An accurate backward mapping is also important, in order to supply a debug, test and monitor interface at the same level of abstraction as the programming and control interface, to ensure that the user experiences a consistent system view.

In terms of implementation, this API might be programmatic, i.e., a third-party tool can interact with the soft platform directly using function calls. Alternatively, the API might be more indirect and involve the transfer of data in a textual or binary intermediate format between a third-party tool and the soft platform.

4.9. Mapping to programmable logic devices

Designing the soft platform as a harmonious, and hence efficient, target of domain-specific tools is only one part of the overall solution. It is also necessary to have efficient mapping of the soft platform to real programmable logic devices (as well as accurate backward mapping). Efficiency issues include basic parameters such as resource use, clock frequencies and power consumption, as well as required message processing parameters such as throughput, latency, reliability and security, versatility, and design and system cost.

A key architectural style that can be used in harnessing a programmable logic device here is an *interface-centric* model (which implies a *logic-centric* system model). Here, the design is driven by the necessary behavior at the system interfaces, in particular, as opposed to a processor-centric model where the design is driven by the behavior of an embedded processor. The interface-centric view can be well matched to the message-centric view of the soft platform. Placement and usage of interfaces, memories and their interconnection dominates the allocation of device features, and then allocation of functional elements (e.g. programmable logic elements, embedded processors, etc.) for the processes can follow as a derivative. This does not rule out other models for harnessing a PLD, but illustrates one particularly good approach for this domain of networked systems, now and in the future.

The actual mapping can be carried out in several ways. In a very static situation where there is only design-time programmability, the programmed features of the soft platform may be described in a standard HDL, and then a standard synthesis, place and route flow followed in order to generate the programming data for a device. In general, a more tailored representation of the programmed features of the soft platform can be used to drive more specialized tools that can perform more efficient mappings to utilize programmable logic device attributes. More specialized tools are also needed to provide features like runtime programmability, as well as debug and monitoring in the reverse direction.

5. Conclusions and future work

We have presented arguments to encourage wider use of programmable logic devices as *primary* components in adaptable networked systems. We see the major requirement as being to make the true capabilities of such devices available to system designers who are not hardware experts. To this end, we presented here a description of our HYPMEP soft platform, architected to mediate between the technology and the communication and computation needs of adaptable networked systems.

We have implemented a first generation of our HYPMEP environment, targeted at the Xilinx Virtex-II Pro platform FPGA family, which includes integrated multi-gigabit speed serial transceivers [14]. The implementation of the soft platform features various novel architectural innovations, and we have devised an XML-based language to describe the programming information for the soft platform. To validate the environment, we have so far carried out four major experimental designs drawn from gigabit rate networking (a protocol bridge, an IPv4/IPv6 router, an RPC server and an automated Click-to-FPGA flow), with successful results in terms of both usability and performance. Each had a focus on a particular subset of HYPMEP soft platform features, for in-depth investigation of this subset to inform future generations of the integrated soft platform. We are reporting full details of these experiments in other publications during 2004.

To highlight just one result, the RPC server experiment involved implementation of an Internet Remote Procedure Call (RPC) protocol server implemented entirely on a single platform FPGA device. The server had a gigabit Ethernet interface, and with the IP and UDP protocols supporting the RPC protocol. The resulting server was capable of handling RPC requests arriving at the gigabit line rate, with the latency caused by protocol handling reduced by a factor of around 10x compared with an equivalent Linux-based implementation running on a 2 GHz processor. Most importantly, from an ease of use point of view, the whole system was implemented – from initial design to working implementation on real hardware – by a non-hardware expert in only two weeks. Note that the system was described in our XML-based language, which ultimately is not intended for human use, rather as a target for higher-level tools, making this speedy work even more impressive.

Beyond these experiments, we are currently exploring the use of the soft platform in applications where the networking is more embedded, for example, in automotive and consumer devices. We are also validating the notion of message processing as an adequate computational style for the activities and functions carried out by such devices. We have developed a precise set of constraints bounding the nature of message processing, and are now exploring these limits for practicality.

Longer term, we desire that this work breaks down obstacles arising from traditional boundaries between hardware and software in networked systems. In tandem, we envisage less rigid boundaries between data planes, control planes and management planes in networked systems. In other words, we see plane integration proceeding in harmony with system integration. Programmable logic devices are a catalyst, matching heterogeneous networks on chip to the functional needs of networked systems.

6. References

[1] Brebner, G., "Single-Chip Gigabit Mixed-Version IP Router on Virtex-II Pro", Proc. 10th IEEE Symposium on Field-Programmable Custom Computing Machines, Apr 2002, pp.35-44.

[2] CloudShield Technologies, Inc., www.cloudshield.com.

[3] Hadzic, I. and J. Smith, "P4: A Platform for FPGA Implementation of Protocol Boosters", Proc. 7th International Workshop on Field Programmable Logic and Applications, Sep 1997, pp.438-447.

[4] Kohler, E., R. Morris, B. Chen, J. Jannotti and M. Kasshoek, "The Click Modular Router", ACM Transactions on Computer Systems **18**(3), Aug 2000, pp.263-297.

[5] Kulkarni, C., M. Gries, C. Sauer and K. Keutzer, "Programming Challenges in Network Processor Deployment", Proc. 2003 International Conference on Compilers, Architectures and Synthesis for Embedded Systems, Oct/Nov 2003, pp.178-187.

[6] Lockwood, J., J. Turner and D. Taylor, "Field Programmable Port Extender (FPX) for Distributed Routing and Queuing", Proc. 8th ACM International Symposium on Field Programmable Gate Arrays, Feb 2000, pp. 137-144.

[7] Novilit, Inc., www.novilit.com.

[8] Paulin, P., C. Pilkington and E. Bensoudane, "StepNP: A System-Level Exploration Platform for Network Processors", IEEE Design & Test of Computers, **19**(6), Nov/Dec 2002, pp.17-26.

[9] SDL Forum, www.sdl-forum.org.

[10] Shah, N., "Understanding Network Processors", Internal Report, U.C.Berkeley, Sep 2001, pp.22/23. Available at www-cad.eecs.berkeley.edu/ ~niraj/papers/UnderstandingNPs.pdf.

[11] Teja Technologies, Inc., www.teja.com.

[12] Turner, J., "Extensible Networking – Transforming the Internet", Keynote address, 2nd Workshop on Network Processors, Feb 2003. Slides at www.cs.washington.edu/NP2/.

[13] Xilinx, Inc., Aurora information, www.xilinx.com/aurora/.

[14] Xilinx, Inc., Virtex II Pro information, www.xilinx.com/virtex2pro/.