Programming a Hyper-Programmable Architecture for Networked Systems

Eric Keller and Gordon Brebner
Xilinx Research Labs
Xilinx, Inc., USA
{eric.keller, gordon.brebner}@xilinx.com

Abstract

Modern programmable logic devices have capabilities that are well suited for them to assume a central role in the holistic implementation of networked systems. We have devised a highly flexible soft platform architecture abstracted from such physical devices, which may be viewed as a particularly configurable and programmable type of network processor. In this paper, we discuss a programming model for the architecture, and present an XML-based description language for expressing the programming information. This intermediate language is designed both to be an attractive compilation target for domain-specific languages used for describing networking applications, and also to have efficient mappings to programmable logic devices, harnessing to the full their high degree of concurrency, interconnectivity and programmability. We present a detailed example, where a high-speed Remote Procedure Call (RPC) protocol server for gigabit Ethernet was described directly in the XML-based language, and automatically compiled to a working implementation on a platform FPGA device. The exercise was carried out by a non-hardware expert in only two weeks, thus demonstrating the unlocking of access to programmable logic technology.

1. Introduction

In this paper, we consider the programmed implementation of networked systems. These are systems, now and in the future, that feature communication and networking as a significant activity, alongside computational activity. Examples include simple sensors and actuators, and more complex clients and servers.

We believe that modern field programmable gate arrays (FPGAs) are well suited as a basis for implementing networked systems, through efficiently providing "hyper-programmable architectures". By the term "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 FPGAs can be viewed as the primary, perhaps exclusive, components of networked systems. This means not only demonstrating that FPGAs have apt capabilities, but also devising new use models and tools to unleash these capabilities.

We have designed a particular hyper-programmable architecture for networked systems. In this paper, we consider in detail a programming interface for this model, that allows a user to express system components in a technology-independent way. This description is then automatically compiled to programming information for an FPGA. The intention is to expose a path for harnessing FPGAs that is accessible to those who are not hardware or, more specifically, FPGA experts. This will remove a major deterrent to using FPGAs, by aiming to match the relative ease of writing software for processors.

We demonstrate the programming model through an experiment to create a Remote Procedure Call (RPC) server, speaking the Internet RPC protocol over a stack of the UDP, IP and Ethernet protocols. In addition to providing a substantial speedup of the protocol handling over a conventional software implementation, this experiment also shows that higher level protocols, with a tradition of software implementation, can be conveniently implemented with programmable logic by a non-hardware expert. This points the way to FPGAs playing a major role in the holistic implementation of networked systems.

Moreover, with distributed systems becoming commonplace, this particular example enables progress in allowing heterogeneous technologies to inter-operate. From grid computing, to sensor networks, to the Internet, conventional microprocessor-based systems can interact as peers with FPGA-based systems for applications such as cryptography, image processing and gene matching, where FPGA implementations have been shown to offer speedups several orders of magnitude greater than their software counterparts.

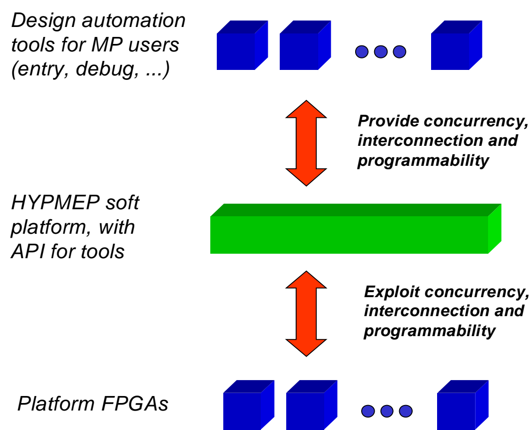

Figure 1. The HYPMEP architectural model

2. Architectural model

2.1. Domain specificity

Various classes of design language are targeted at experts in particular domains, instead of appealing to system designers in general. These domain-specific languages, together with compilers and other tools, provide a design environment tailored to the specific domain. The restriction to a narrower domain offers scope for more realistic mappings to hardware than is possible when attempting to support general-purpose design methodologies. For this reason, we see domain specificity as a promising direction for facilitating access to modern platform FPGA technologies.

We are interested in the domain of networked systems as our focus. There have been various proposed domain-specific languages in this area. One example is the Click modular router [7] developed at MIT, originally targeted at implementation on a Linux workstation. Other recent alternative proposed languages, some targeted at emergent network processor technologies, include CloudShield's RAVE [3], Novilit's Anyware [8] and Teja's Teja C [10]. A more long-standing example is SDL [9], used for almost three decades in telecommunications. However, unlike in the DSP domain, where Matlab and Simulink have wide popularity with domain experts, neither Click nor other proposed languages have yet emerged with the same stature in networking. For this reason, we have focused on a domain-specific hyper-programmable architecture as a target for compilers, rather than a particular domain-specific language.

2.2. Overview

Figure 1 shows the top-level view of our mechanism for 'impedance matching' between domain-specific design automation tools for networked systems and platform FPGAs. The centerpiece is the HYper-Programmable MEssage Processing (HYPMEP) 'soft platform' – a highly programmable microarchitecture. We use 'message' as a generic term for 'packet', 'frame', 'cell', 'data unit', etc., that is, any sort of discrete unit of information being communicated over networks. As indicated in the figure, HYPMEP is designed to exploit, to the full, the concurrency, interconnection and programmability inherent to the FPGA, and deliver this at a higher level of abstraction to the design automation tools. The soft platform is presented as an API that is accessible to the higher-level tools. In essence then, this API defines the programming language that allows programming of the hyper-programmable platform. We thus supply a tool-neutral mapping point for domain-specific tools, abstracting the most applicable properties of an underlying platform FPGA. We also cater for a range of capabilities from underlying programmable logic device (PLD) technologies, down from rich platform FPGAs including embedded processors and memories, through basic FPGAs, down to more constrained programmable logic arrays.

2.3. Soft platform components

Messages enter and leave the soft platform through perimeter interfaces, and are stored within it while they are being processed. All of the major components of the soft platform model are designed to have very efficient mappings to the full capabilities and characteristics of platform FPGAs. Stressing 'full capabilities' is important because we particularly aim to ensure that the attributes of these soft platform components are highly programmable (hyper-programmable).

The soft platform has two types of component concerned with the processing of messages. The first type is the *thread*, a lightweight concurrent entity with programmable structure and function, and an efficient FPGA implementation. It is targeted at the kinds of simple message header processing required for protocol implementation. The second type is the *hook*, a programmable wrapper for an existing functional block with an FPGA implementation. It is targeted at the kinds of more complex algorithmic processing required for message headers (e.g. checksum calculation) or payloads (e.g. encryption).

There are two other types of major component. First, an *interface* is used for moving messages into or out of the system perimeter and, in general, includes both thread- and hook-like functionality.

Second, a *memory* is used for storage of messages, system state or system data. Memories are implemented using apt elements in the memory hierarchy of an FPGA (e.g., flip-flops, registers or block memory).

3. The HAEC API for the soft platform

3.1. Overview

We have designed three experimental APIs – named HIC, HAEC and HOC – at decreasing levels of abstraction. HIC exposes a particular abstract model of message processing, and is suitable for higher-level tools. HOC exposes the basic design components for a platform FPGA, such as HDLs, programming languages and networking IP blocks. In between, HAEC exposes the major soft platform components described above, and is the focus here.

HAEC (HYPMEP Architecture Elaboration Code) is an XML-based programming language. XML is a markup language that allows a formal grammar to be specified through the use of the document type definition (DTD), and here it has allowed very rapid exploration of language provision in tandem with architectural exploration. The API accepts a HAEC description, and we have written a prototype compiler that converts this directly to a hardware description expressed in VHDL (later, the HOC language will be the target). This hardware description is then compiled by standard back-end tools to generate the low-level programming information for the platform FPGA.

3.2. Threads

From a user's perspective, the programming interface provides a coding environment targeting multiple micro-engines that can operate in parallel. These micro-engines are realized as threads. Here, the term 'thread' is used in the sense introduced by Brebner [2] to describe a unit of concurrency targeted at programmable logic, and not implying all of the baggage associated with software threads in standard operating systems. An instruction set is used to program the behavior of a thread; at present, the instruction set is fixed but, in the future, we anticipate this also being programmable. Although these are machine code style instructions, the thread does not operate as a traditional microprocessor, i.e., a fetch, decode and execute pipeline feeding an ALU. Instead, each thread is implemented as a custom finite state machine, with instructions associated with the states, and with each instruction included having its own dedicated hardware implementation. Given this, there is scope for instruction level parallelism, since multiple instructions can be executed simultaneously.

Figure 2 shows a fragment of HAEC code for the definition of a thread. The main definition starts with the variables in lines 4 to 7. Variables can be internal, input or output, and have a defined bit width. Input and output variables are used for inter-thread communication, and are discussed later. The use internal to the thread is the same for each type though, in that variables can be assigned to and read from, as would be expected. After the variables, the thread's controlling state machine is defined from line 8 onwards. The tag that starts this definition indicates which state is the start state, that is, the state that is the first to execute when the thread is started. An optional tag can be used to identify a stop state, for cleanup.

```
 1: <thread name="rx_thread">
 2: <useinterface intname="RX" name="mygmac" port="rx"/>
 3: <usemem intname="PUT" name="ethrecv_buf" port="put"/>
 4: <variables>
 5:   <internal name="len" width="16"/>
 6:   <internal name="addr" width="11"/>
 7: </variables>
 8: <states start="startState" altstart="RX_dataValid">
 9:   <state name="startState">
10:     <operation op="WRITE_DATA" params="PUT, RX_Data, 4"/>
11:     <operation op="ASSIGN" params="addr, 4"/>
12:     <transition next="writeData"/>
13:   </state>
14:   <state name="writeData">
15:     <conditional>
16:       <condition cond="EQUAL" params="RX_dataValid, 1">
17:         <transition next="writeData"/>
18:           <operation op="WRITE_DATA"
19:                 params="PUT,RX_Data,addr"/>
20:         <operation op="ADD" params="addr, addr, 1"/>
21:       </condition>
22:       <condition cond="else" params="">
23:         <operation op="WRITE_DATA" params="PUT, addr, 0"/>
24:         <transition next="commitPacket"/>
25:       </condition>
26:     </conditional>
27:   </state>
…
```

Figure 2. Example HAEC code for a thread.

After the tags defining the thread control, each of the states is defined as shown on lines 9 to 13 and 14 to 27 in Figure 3. Operations, conditionals and transitions can be associated with a state. Operations make use of the instruction set to define functionality. As normal in programming, the instructions can use variables as arguments, or pre-defined symbolic or numeric constants as arguments.

A conditional, as shown on lines 15 through 26 in Figure 3, is a grouping of operations, transitions or other conditionals that depend on a certain condition. This is a standard "if, else if, else" style mechanism. Each branch of the conditional requires a condition, as shown on line 16. These conditions are part of the instruction set and include instructions such as EQUALS and LESS_THAN. Transitions indicate the next state to execute, and are deterministic, i.e., there can only be one transition per possible execution path in a given state.

3.2.1. Inter-thread communication

To provide a useable model, threads must be able to communicate to some extent. The soft platform supports two direct forms of communication between threads, although more indirect communication via shared memory is also possible. One mechanism is very lightweight, and just involves a direct connection between two threads. This allows for the threads to handle any mutual handshaking that is desired. For example, one form of handshaking may involve a data valid signal that tells a receiver of data that the data can be read. Another form may additionally involve an acknowledge signal which tells the sender that the receiver did receive the data. Of course, two threads may also exchange data without any explicit control handshaking if they can assure correct synchronization in some other way.

These explicit connections provide for programmed communication with no restrictions. It is also desirable to support a method of communication where the functionality lies in the communication mechanism rather than in the threads. For this purpose, channels exist. The difference typically comes with the use. Direct connections are more useful for passing fields of a message or results from a calculation between threads. Channels are more useful for streaming a message, or any other ordered data, through the system to each of a number of threads. The streaming may be continuous and have data available every clock cycle, or it may be asynchronous with some extra control.

We now consider how these mechanisms are presented in the HAEC language. As already noted, variables within threads can be defined as input or output. For explicit inter-thread communication, such variables can be connected. When a sending thread assigns a value to an output variable, that value will appear on a receiving thread's input variable and can be used. The value on the receiver's input is held only as long as the sender's output retains that value, so it is necessary for the threads to be accurately synchronized in time. For each connection, a name, a source, and one or more sinks are given. Source and sinks are pairs containing the name of the thread and the variable.

The other form of communication is through explicit channels, allowing for more complex data transfers between threads. A collection of channel types is supplied, each one with a pre-defined FPGA implementation. To use a particular channel, the HAEC description first must declare it, using an "include" mechanism that is also used for interfaces and memories. The threads that make use of the channel, either on the sending or receiving side, include a "usechan" tag within the thread definition.

An example channel type is the AlignedChannel. This channel allows a sender to broadcast a continuous or non-continuous stream of data. It also allows the receivers to access the data in a partly-random manner. The sending thread simply sends data into the channel. Then, each broadcast datum has an address associated with it, the address being calculated by the channel. This address enables receivers to simply wait until the address that the thread wishes to access matches the address of the data being broadcast by the channel. Because of this, the receivers do not have to count cycles or deal with data values that are not valid, thus simplifying the thread circuitry. An additional feature of this channel, as suggested by its name, is that receivers can access data on non-datum boundaries because, in addition to outputting the current datum, the channel also outputs the previous datum.

3.2.2. Inter-thread control flow

A major feature of the threading model is that control flow exists on a per-message basis, instantiated by sequences of threads starting and stopping other threads. A new flow begins with a thread handling the receipt of a message, and ends with a thread handling the departure of a message. The flow can be visualized as a graph of thread activation activity, in order to achieve the overall processing necessary for a particular message.

To implement this, threads can start, stop and query other threads using special control instructions: START(thread name), STOP(thread name) and, for example, the IS_FINISHED(thread name) query instruction. When a thread is stopped, it may either go directly into an idle mode or into a stop state to perform cleanup before going into the idle mode.

3.3. Hooks and blocks

Programmable threads provide a basis for many of the common tasks encountered when processing network protocols. However, there are also algorithms not naturally described by finite state machine model. For example, encrypting a message using direct processing of the data is much more efficient, both in terms of performance and specification. To accommodate such cases, and allow the inclusion of functionality designed outside our flow, we have the capability to include and make use of existing blocks.

We use the term 'hook' to describe the interfacing wrapper used to integrate a block into the overall soft platform based system. These blocks might be in the form of either hardware netlists or software code, though the present version only caters for the former. In the HAEC description, the interface of the block – its inputs and outputs – is defined as shown on lines 1 to 7 in Figure 3. It is assumed that the block will require a clock input as well as a reset signal, which do not need to be defined explicitly. An instance of one of these blocks is introduced as shown on line 8. This specifies the type of the block, and a name by which it will be referred to in the system, to allow its activation by threads and its connection to memories.

```
1: <extern_IP_def type="Multiplier">
2:   <input  name="in1" width="32"/>
3:   <input  name="in2" width="32"/>
4:   <input  name="invalid" width="1"/>
5:   <output name="res" width="32"/>
6:   <output name="outvalid" width="1"/>
7: </ extern_IP_def >
8: < extern_IP type="Multiplier" name="mymult"/>
```

Figure 3. Example HAEC fragment to define and instantiate an externally defined block.

3.4. Interfaces

One or more external interfaces are located at the perimeter of the defined system. These are used to move messages into and out of the system. Note that they are not necessarily restricted to connecting to input or output pins of the underlying FPGA. They can also define an interface between the network processing system defined using the soft platform model and a larger design implemented on the FPGA that incorporates the defined system as a subsystem.

The interfaces are not simply groupings of input and output signals. They also include functionality that enables the exchange of messages between the network processing system and its environment. Typically, the required functionality will exist as a pre-defined block, and this has to be integrated with the system. Here, the block will have an internal interface of the type described in Section 3.3, but also have an external interface describing how it interacts with the enclosing environment. The latter then contributes to the overall external interface of the described system by contributing its list of signals. This explicitly includes clock signals as well as data and control signals.

The internal interface is packaged as a list of ports that are used internally by the network processing system, where each port is a grouping of signals that relate to each other. For example a port of an interface could contain a data bus as well as control signals. A final feature of the specification is the timing requirements of each port as well as of any external clock. Currently, all of the external and internal specification of each type of known interface is built in, and does not feature as a programmable aspect in the HAEC description.

Any built-in interface type can be instantiated within a system description. The "include" mechanism, already seen, is used. After the inclusion of an interface, it must then be associated with a thread by use of the "useinterface" tag within the definition of a thread. Each of the threads with an associated interface is responsible for handling messages entering or exiting the system. As each type of interface will have different protocols to read/write data, there is the need for varying functionality, and this is offered by the programmable threads. To accommodate the functions required, the threads used with interfaces have access to certain 'system' (or 'privileged') instructions that the basic threads do not have. These instructions enable input and output across the external interfaces.

3.5. Memories

Memory is a key component of all systems. In particular, in a network processing system, memory is used for buffering of messages, tables for lookup, and storage for state. Like interface blocks, memory blocks need to be both instantiated and associated with a thread. Each memory can have one or more ports, depending on the type. Instantiating a memory in HAEC involves specifying the name to be used for it as well as the type of memory component. Certain memories can be parameterized and require additional specification. As with interface ports, threads can be associated with a memory port. The ports are accessed by read and write instructions, conditional instructions, and memory specific instructions.

4. Compilation of HAEC to VHDL

Each of the system components instantiated by the HAEC description is mapped to a hardware entity on the FPGA. The hooked blocks, interfaces and memories all exist as predefined netlists. They are included by creating an instance in the VHDL description, linking this to other elements as required.

A major contribution of our compilation process is the automatic generation of clock signals, thus removing a significant hardware-style complication from the software-style HAEC description. Each of the threads, channels, interfaces and memories must have clocks in the VHDL description.

However, it is not necessarily the case that they all operate from the same clock – in fact, it is most likely that they do not. Within the soft platform model, the interfaces play a key role in clock domain determination, since the goal is that the internal clocking of the system is chosen just so that it delivers the required timing at its perimeter interfaces. Each interface can require a different clock frequency, so there will be multiple clock domains and, without careful handling, this can cause problems.

The compiler maps each thread to custom hardware in the VHDL description, driven by the definition of the thread's functionality in the HAEC description. Just as the definition is structured as a finite state machine, so is the definition in the generated VHDL. Operation instructions in the HAEC definition get mapped to VHDL operations, as do conditional instructions. When certain types of inter-thread channels are included, the HAEC description of a receiving thread may specify that the thread has to wait for data to become available on the channel. In the hardware description, this requires the automatic wrapping of the state that the read instruction is being performed in with a conditional statement.

Finally, another necessary set of connections between threads are the inter-thread synchronization signals (e.g. start and stop). These signals are generated automatically by the compiler. In the VHDL implementation, each thread element has a set of output signals that potentially allow it to start each of the other threads. Then, all the signals for starting a particular thread are logically OR-ed together. This may seem like it would create unnecessary extra logic – in practice, every thread will not start every other thread – but we had full confidence that the back-end synthesis tool is capable of optimizing out any unnecessary logic.

It can be seen that the major work of the compiler is in automatic generation of signals between generated elements. This captures the key mapping: from a software-oriented world to a hardware-oriented world.

5. Experiment: an RPC server

5.1. Introduction

We have conducted several experiments in which complete networked systems have been described using HAEC. These descriptions were constructed manually, although ultimately HAEC is intended as a target for higher-level compilers, rather than for human programmers. Here, we demonstrate the capabilities of the HYPMEP soft platform and HAEC itself by the design and implementation of an end system that is a Remote Procedure Call (RPC) server. This example shows the use of an FPGA in an application typically regarded as being in the software domain, but here facilitated by the ease of use of the HYPMEP programming interface and the efficiency of the generated hardware description.

The history of the Remote Procedure Call paradigm [1,4] has been very centered on a workstation-only distributed system model. However, with the increasing use of FPGAs as the basis for systems, it is desirable to extend this model to FPGAs, thus making provision for truly heterogeneous distributed systems. An FPGA-based system for data capture could make use of RPC to provide an NFS-based [5] file system to provide access to its data. Alternatively, with a reverse relationship, an FPGA-based client could log its status every so often to a file on a workstation server. Either direction of relationship allows inclusion of an FPGA-based system without the need for any special software running on workstations.

In our experiment, HAEC was used to define the implementation of an RPC server with a gigabit Ethernet network interface. Over this interface, the IP and UDP protocols are used to underpin the RPC protocol. Thus, including the Ethernet protocol, there is a four-layer protocol stack in use. The server supported the arbitrarily chosen procedures "int add(x, y)" and "int mult(x, y)", to add or multiply two integers, respectively. These simple functions are placeholders, since the main implementation challenge is in the networking; in general, as has been indicated earlier, arbitrary function blocks can be hooked into our system. In addition to the required RPC functionality, port mapper [6] functionality was also implemented. This enables a client to look up the UDP port number of the 'program' that is supporting the callable procedures. (Note that the concept of a program is built in to the terminology although, of course, we have no actual program in our FPGA-based implementation.)

The required functionality of the system is as follows. An RPC client will make a function call that is directed to a remote procedure. This stimulates the creation of an RPC message encapsulated in a UDP segment, within an IP packet, within an Ethernet frame. The RPC message itself contains header information specifying the procedure being called, followed by the parameters to the procedure. When this message arrives at the FPGA-based system via gigabit Ethernet, the various protocol headers must be inspected and stripped off. After the header of the RPC message has been checked and processed, the parameters are passed to the hardware block implementing the procedure. When this is finished, the result is then encapsulated in an RPC message, UDP segment, IP packet and Ethernet frame for transmission back to the client.

As part of our experiment, we also implemented exactly the same system specification on a 2 GHz Linux workstation with a gigabit Ethernet interface, using the built-in operating system support for the Internet protocols. This was to allow measurement of RPC processing times for both the FPGA-based server and the workstation-based server, using a different Linux-based workstation as the client in both cases.

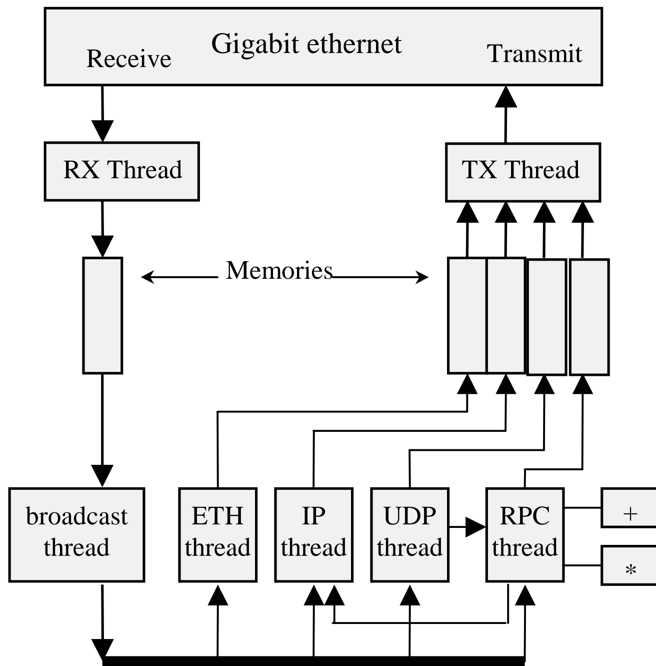

Figure 4. Architecture of the RPC server.

5.2. System architecture

Our system architecture, based on the HYPMEP soft platform, is shown in Figure 4. Each of the labeled components is described in HAEC, and gets mapped to a block of hardware in the FPGA. Only a single interface is needed in this application, for the gigabit Ethernet. The RX thread is responsible for reading data from the interface and placing it in a buffer. In terms of HAEC, one line of code is needed to say that the gigabit Ethernet interface is being used, then one further line of code is needed for the RX thread to connect to this interface. For the buffer, again one line of code is needed to create the memory, and one line of code is needed for the RX thread to connect to this memory. READ and WRITE instructions are used to access the memory.

When a message is fully received into the buffer, the broadcast thread is activated by a non-empty signal from the buffer. The broadcast thread is simply responsible for reading from the buffer and sending 32-bit words through a channel, denoted by the thick link in Figure 4, to each of the other threads. Each thread listening to the words broadcast by the channel just specifies the address that it is waiting for, and then is suspended.

The broadcaster starts the ETH thread when it starts sending data. The ETH thread is then responsible for checking the Ethernet MAC header. If the Ethernet header is valid, then the ETH thread starts the IP thread. Meanwhile the ETH thread continues, to produce a speculative Ethernet header for the outgoing frame containing the RPC result, and place it in a memory. The IP thread is responsible for checking the IP header. When the header is valid, in particular that the protocol type indicates a UDP segment, it will start the UDP thread. It will then generate as much of a speculative IP header for the outgoing return packet as is possible and puts it in a memory. One field is the length of the IP packet, which is unknown at this stage, since it depends on which remote procedure is being called. Thus, the thread waits for the RPC thread to tell it what the result size is. The UDP thread checks the UDP header, which includes examining the UDP destination port to tell which RPC ‘program’, i.e. group of procedures, is to be used. The program number is given to the RPC thread when the UDP thread starts it, as indicated by an arrow between the two threads. Like the ETH and IP thread, the UDP thread generates a speculative outgoing UDP header.

The RPC thread decodes the RPC message and generates the return RPC message. The decoding involves deserializing the parameters and passing them to the relevant function block, which is attached via a hook. When the RPC thread can determine the length of the return message – which may be before the function is completed – it tells the IP thread. This is shown as an arrow between the RPC thread and the IP thread. Finally, when all four threads – ETH, IP, UDP, and RPC – commit their respective parts of the message into the memories, the TX thread then transmits the response.

Note the concurrent execution of the threads in this system, a latency-reducing feature not possible with a pipelined concurrency system architecture.

5.3. Experimental results

We implemented our system on a Xilinx XC2VP7 Virtex-II Pro platform FPGA [11], which includes eight multi-gigabit transceivers. The standard Xilinx ISE 6.1 tools [11] were used to produce the programming bitstream for the FPGA from the VHDL hardware description produced by our compiler. The FPGA was on a Xilinx ML300 board [11], which brings out four optical gigabit ethernet connections from four of the gigabit transceivers.

The ML300 board was connected, by a fiber optic cable, to a Linux workstation fitted with a NetGear GA621 gigabit Ethernet card. RPC calls were made from the workstation to the FPGA, and back, through this physical networking.

The basic required performance of our FPGA-based system was dictated by its external interface, which had to operate at a one gigabit per second line rate. Since the interface block used had an 8-bit interface, this means that the RX and TX threads had to run at 125 MHz. The other threads in the system all used a 32-bit data width, which means that they could operate at one quarter of this speed, namely 31.25 MHz. Both of these required frequencies were comfortably met by our automatically synthesized system.

Given that gigabit line rates could be dealt with, the next metric of interest was the latency of the server, that is, the response time to a remote procedure call. The latency of the FPGA-based system was calculated to be 2.16 µs for each of the RPC calls used in the experiment. To compare this figure to a workstation-based server, the same client workstation made 900 successive calls to a server workstation, which had a 2 GHz Pentium-4 processor and 512 MB of RAM. The local response time of the workstation-based server was measured to be 15 ms total for the 900 calls, compared with just 2 ms total for 900 calls to the FPGA-based system. That represents a 7.5X speedup for the FPGA implementation over a high powered workstation. As the chosen functions (add and mult) are extremely fast with either technology, this processing time essentially only represents the protocol handling time.

For the FPGA-based implementation, the area used on the device is also of interest. Our circuitry required 2600 programmable logic slices and five embedded block RAMs. This area represents less than 2% of the biggest device in the Virtex-II Pro family, and fits within the second smallest family member. It should be noted that half of the area was actually used to implement the gigabit Ethernet MAC interface, which was a predefined netlist.

In terms of the system description, 869 lines of HAEC were required. When processed by the compiler, this generated 2950 lines of VHDL for input to the backend tools. We have noted this approximate 1:3 expansion factor in our other experiments with compiled HAEC descriptions.

A final, and highly important, measurable result was the development time. It took only two weeks for a non-hardware expert to design, implement and fully debug this system. This demonstrates the ease of use factor, even though the abstraction level of HAEC is not really intended for human use.

6. Conclusions and future work

Our overall research aim is to demonstrate that modern programmable logic devices, particularly FPGAs, have capabilities that are well suited for them to be used as first-order components in networked systems. In this paper, we have focused on the programming mechanism for a hyper-programmable architecture for networked systems, which feeds into a compiler that deals automatically with various hardware-like features, notably clock management.

Our RPC experiment reported here demonstrated two important points. First, the functionality of the system itself indicates how FPGA-based systems might be comfortably integrated into heterogeneous distributed systems alongside processor-based systems. Second, the fact that the fully working system was constructed by a non-hardware expert in just two weeks, is an excellent demonstration of the promise of our general approach.

In future work, we plan further refinement, generalization and extension of the HYPMEP architectural model, guided by feedback from various networked system experiments like the one reported in this paper.

References

[1] Birrel, A. and B. Nelson. “Implementing remote procedure calls”, ACM Transactions on Computer Systems **2**(1), Feb 1984, pp.39-59.

[2] Brebner, G., “Multithreading for Logic-Centric Systems”, Proc. 12th International Conference on Field-Programmable Logic and Applications, Sep 2002, pp.5-14.

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

[4] Internet RFC 1057, “RPC: Remote Procedure Call Protocol Specification version 2”, Jun 1988.

[5] Internet RFC 1094, “NFS: Network File System Protocol specification”, Mar 1989.

[6] Internet RFC 1833, “Binding Protocols for ONC RPC Version 2”, Aug 1995.

[7] 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.

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

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

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

[11] Xilinx, Inc., www.xilinx.com.