

# Modeling Reconfiguration in a FPGA with a Hardwired Network on Chip

Muhammad Aqeel Wahlah\* and Kees Goossens\*<sup>†</sup>

\*Computer Engineering, Delft University of Technology, [aqeel@ce.et.tudelft.nl](mailto:aqeel@ce.et.tudelft.nl)

<sup>†</sup>NXP Semiconductors, The Netherlands, [kees.goossens@nxp.com](mailto:kees.goossens@nxp.com)

## Abstract

We propose that FPGAs use a hardwired network on chip (HWNOC) as a unified interconnect for functional communications (data and control) as well as configuration (bitstreams for soft IP). In this paper we model such a platform. Using the HWNOC applications mapped on hard or soft IPs are set up and removed using memory-mapped communications. Peer-to-peer streaming data is used to communicate data between IPs, and also to transport configuration bitstreams. The composable nature of the HWNOC ensures that applications can be dynamically configured, programmed, and can operate, without affecting other running (real-time) applications. We describe this platform and the steps required for dynamic reconfiguration of IPs. We then model the hardware, i.e. HWNOC and hard and soft IPs, in cycle-accurate transaction-level SystemC. Next, we model its dynamic behavior, including bitstream loading, HWNOC programming, dynamic (re)configuration, clocking, reset, and computation.

## 1. Introduction

Advancements in chip-making technology during the last two decades have fueled the field programmable gate arrays (FPGAs) transformation from a simple PLD to multi-million gate chips [1], [2]. These FPGAs are used to prototype as well to realize systems as complex as multiprocessor system on chip (MPSOC). However with MPSOCs systems, constituting hundreds of processing elements and executing large number of applications with heterogeneous characteristics, issues related to IP integration, system verification and validation become prominent. The situation becomes even more critical with real time applications which demand hard guarantees in terms of throughput and latency from the underlying FPGA data interconnect.

Before proceeding further we define terminology. A *soft IP* is mapped on FPGA reconfigurable computational blocks (LUTs) which are connected to each

other using FPGA programmable interconnect. An IP is *hardwired* or *hard* when it is directly implemented in silicon. We define *(re)configuration* as the installation of new functionality (of soft IPs) in the FPGA by sending a bitstream to a reconfiguration region. An IP is *programmed* after it is configured, if necessary, which entails changing the state of its registers when it is in functional mode.

Soft interconnects are often used interconnect soft IPs. However, timing closure problems may arise because the interconnect physically spans the FPGA. Moreover, dynamic partial reconfiguration may be restricted if the soft interconnect is interspersed with the soft IPs in the same FPGA regions. To overcome these limitations we have proposed to hardwire an inter-IP interconnect [3]. Soft IPs can be (re)loaded without being restricted by the interconnect. Additionally, we use a network on a chip (NOC) to ensure that deep sub-micron and timing-closure problems can be solved. Our *Æthereal HWNOC* [4] offers *composable* communication to ensure that different applications do not affect each other, as they are configured and started, run, and are stopped dynamically [5].

In this paper we describe the design-time synthesis and configuration design flow, and the definition of the run-time dynamic starting and stopping of applications without disruption to other concurrent applications. We model the HWNOC and soft IPs cycle-accurately in SystemC, including bitstream loading, clocking and reset, programming and running, and show a detailed example of the platform in action.

The remainder of this paper is structured as follows. Section 2 positions our platform with respect to related work. Section 3 defines the platform, and Section 4 describes the design-time tool flow. Section 5 defines the run-time dynamic starting and stopping of soft IPs, and how it is modelled in SystemC. Section 6 extends this to dynamic partial reconfiguration of applications. Section 7 shows a worked example, and compares the performance of the system with a soft and hard interconnect. We conclude in Section 8.

## 2. Related Work

Our platform uses a hardwired composable real-time NOC to guarantee uninterrupted simultaneous configuration (soft IP bitstream loading) and functional communications between IPs. We therefore discuss research on soft and hard interconnects in FPGAs.

*Soft* interconnects include single-hop buses [6] and cross-bars [7], and multi-hop networks [8], [10], [11]. The former are not scalable in the number of attached IPs, and cope less well with timing closure problems. The latter, do not suffer from these restrictions but are expensive in terms of area and performance because they are soft, i.e. implemented using the LUTs of the FPGA. They therefore also interfere with the placement and routing of soft IPs in the FPGA, requiring restrictions on IP and soft interconnect placements if dynamic partial reconfiguration is required. [9] offers real-time guarantees when applications run, but implements configuration and functional communications on separate interconnects.

*Hardwired* NOCs are described in [12], [13] as functional interconnects for future FPGA chips. Our previous work [3] works these concepts out in detail. More importantly, it 1) unifies the transport of all kinds of data: configuration (bitstreams) and functional (programming or control, and “normal” data). And 2) ensures that concurrent applications do not affect each other at all during configuration or functional mode, by using the real-time (i.e. composable) *Æ*thereal NOC. In this paper we model the proposed platform and demonstrate its run-time behavior.

The architecture in [14] also proposes the unification of functional and configuration data by introducing so-called cells. Although the architecture is described in detail, no simulation results are shown of it in action, as we do here.

## 3. Hardwired NOC Platform

In this section we describe how a hardwired NOC is embedded and used in a FPGA [3]. Figure 1 illustrates the different components: configuration and functional regions (CFR), NOC consisting of routers and network interfaces (NI), configuration memory, and boot processor (BPro).

A CFR is a collection of LUTs that are configured from one port on the NOC, i.e. some IP connected to the NOC sends a bitstream to the CFR, where it is instantiated. Normally, a real-time connection with fixed latency (and guaranteed bandwidth) would be used for this. Note that in our case CFRs are isolated

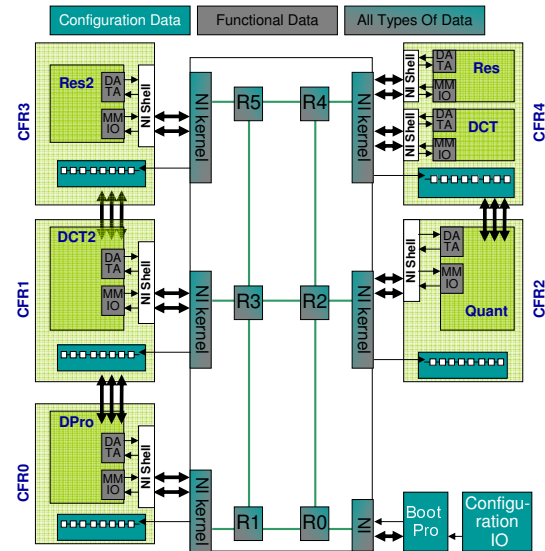


Figure 1. FPGA with Embedded NOC [3].

configuration regions, but they are not isolated functionally. In other words, a soft IP may span multiple CFRs (not shown in the figure), and multiple soft IPs may be placed in the same CFR (e.g. CFR4). This removes restrictions on IP placement, although for dynamic partial reconfiguration some restrictions persist, as described later. NIs are split in hard NI kernels and soft NI shells [15]. A shell is soft because its configuration (number of ports, depth of FIFOs, etc.) depends on the soft IP that is attached to it. This is reflected in how the platform is modeled, as described in Section 5.2.

The boot processor (BPro) is a programmable or fixed-function hardwired IP that instantiates soft IPs by loading bitstreams from a bitstream memory to the appropriate CFRs. It does so by first setting up a connection from a port on its NI to the configuration port on the NI of the CFR. This entails programming the NOC using memory-mapped IO (MMIO), as described in detail in [16], [3]. For our purposes, this is performed by an abstract `open_connection` function. After the bitstream has been sent to the CFR, the bitstream connection is removed. Following this, the IP is programmed by the boot processor using connections over the NOC; e.g. filter coefficients are written in memory-mapped registers. Finally, the IP starts operating, which includes communication via connections over the NOC to other IPs.

The *Æ*thereal NOC can use any topology that fits with the particular layout of CFRs and hard IP on the FPGA. It offers guaranteed (real-time) performance on its connections, such as minimum bandwidth and maximum latency [4]. In particular, it offers *composable* communication [17], [5], which means that different

connections do not influence each other at all. Hence, an application can be started and stopped (configured, programmed, run, etc.) without disturbing other concurrently operating applications. An application contains multiple connections between its constituent IPs, and a set of concurrent applications is called a use-case.

#### 4. Design Time

In this section we describe the design-time activities. Ideally, the soft IPs are independently synthesized into relocatable units (bitstreams occupying (parts of) CFRs). The size and shape of these units should then be passed to UMARS [18]. In general, the UMARS tool maps ports of IPs to appropriate NI ports, to minimize the bandwidth on the NOC, and to minimize the latency for the IPs. It routes connections on the topology and reserves time slots in a time-division-multiplexing (TDM) table to ensure that required latencies and bandwidths are met for each connection. However, unlike [19], it is not aware of the area of IPs. Moreover, current tools do not permit us to generate relocatable bitstreams per soft IP. In our experimental set-up we therefore use a synthetic bitstream per soft IP, and manually map soft IPs on fixed locations on the HWNOG, like the hard IPs. We currently do not support bitstream relocation, i.e. the ability to load a bitstream in different CFRs. The bitstreams are stored in the (off-chip) bitstream memory.

#### 5. Run Time

Our platform contains a boot processor (BPro) to bootstrap the system and to manage the dynamic starting and stopping of applications. The CFRs are passive in the sense that they react to the commands of the boot processor. In this section we describe in detail all the steps required to start and stop an application, and also how this is modeled in SystemC. From a high-level perspective, to start a single soft IP operating, the following steps take place, also illustrated in Figure 2. 1) The boot processor reads a bitstream from the bitstream memory. 2) It checks if the bitstream would overwrite active regions of the CFR. 3) assuming not, then the clock of the IP is switched off, 4) the bitstream is sent to the CFR. Then 5) the clock of the IP is switched on, and 6) the IP is reset. After 7) programming the IP, 8) the IP runs.

An application normally consists of multiple (hard and soft) IPs. Steps 1-6 are performed first, for all IPs, and then the remaining steps. (Configuration is omitted for hard IPs.) The NOC is reconfigured several times:

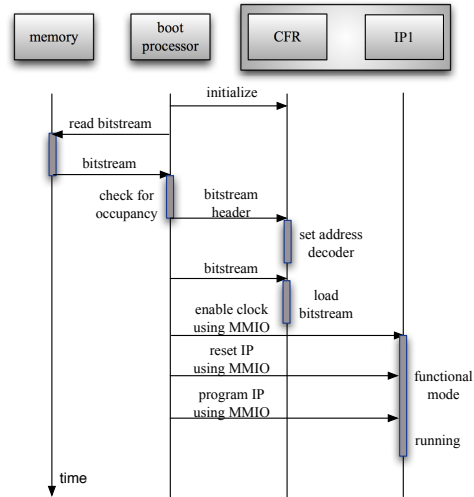


Figure 2. Starting a soft IP.

to transport the bitstream to the CFR (boot processor to CFR configuration port), to program the IP (clock, reset, functional MMIO registers), and to enable the IP to communicate with other IPs (boot processor sets up connections between the IPs).

The application that we model, described in more detail in Section 7, comprises a pipeline of soft IPs that operate on and produce addressless streaming data (e.g. DCT, Quantizer). The data at the start and end of the pipeline is read from and written to memories that are part of a data processor (DPro). In the description of the run-time process, we assume that the data processor is set up once and persists over different applications, while the application (pipelines) are dynamically started and stopped. Figure 3 explains the run-time flow to configure and execute a basic pipeline, starting with the reset of the system.

Our SystemC model includes: 1) A bit and cycle accurate model of the NOC, including its MMIO registers, its programming, and transport of bitstreams, programming, and functional data of IPs. 2) Accurate modeling of bitstreams consisting of multiple frames with headers and specific location in CFRs, etc. 3) A behavioral model of the boot processor, i.e. how it manages applications and their transitions, including bitstreams, clock, reset, and programmable registers of IPs. 4) A bit and cycle accurate behavioral model of CFRs, including exactly when which frames are occupied and/or in use by soft IP. It also switches between the behavioral models of all soft IPs that can be mapped to this CFR, depending on the bitstream that has been loaded. The clock, reset, and programming of the soft IPs is also modeled. 5) Models of all memories, including the bitstream memory. The

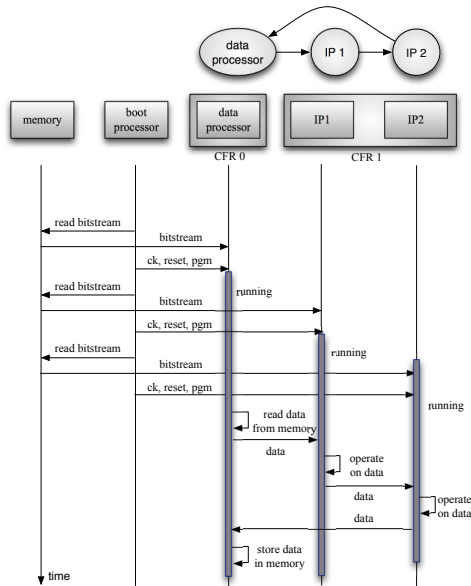


Figure 3. Starting an Application.

following subsections further detail the boot processor, the CFR, data processor, and their models.

## 5.1. Boot Processor

The boot processor is the backbone of the platform: it starts the applications consisting data processors and IP pipelines. It knows which applications are stored in the bitstream memory, and which ones are running in which CFRs at what location. Bitstreams are modeled accurately, with registers such as system frame length (FLR), start frame address (SFAR), total bitstream frames (TFR) and data frame register (DFR). Similarly, information about applications includes their status (ASR), which allows it to avoid configuring a CFR for a new application when it still in use by another running application. Additionally, the regions for their input (IBR) and output (OBR) data at the start and end of the pipeline are also provided.

The following pseudo-code algorithm illustrates how bitstreams are loaded, frame by frame, into CFRs.

### BootPro Functioning Per UseCase

```

1 InitializeCFRs ();
2 // Configuration Phase
3 For Each application,
4   For Each ip in application
5     buf = retrieve (Config_Memory);
6     For Each bitstream in buf
7       IF bitstrheader
8         SFAR = getStFrame (buf);
9         TFR = getFrameCnt (buf);
10        CFId = getCFRID (SFAR);
11        // Check IP Conflict
12        IF IPScan (SFAR, TFR)
13          StopIP (RST, CFId);

```

```

14      EndIF;
15      // Save CFId for first ip of each app;
16      // it is sent to data proc. to provide input data
17      IF First App IP
18        ACF = CFId;
19      EndIF
20      // Send Header and Update IP tables
21      Sndbitstream (SFAR, TFR, CFId);
22      UpdateIPTable (SFAR, TFR, CFId);
23      EndIF;
24      IF bitstrframe
25        For all frames <= TFR
26          DFR = getBitstrData (buf);
27          Sndbitstream (DFR, CFId);
28        EndFor;
29      EndIF;
30      IF last bitstream
31        InitializeIP (Clk, RST, CFId);
32      EndIF;
33    EndIf;
34  EndFor bitstream;
35 EndFor ip;
36 Endfor application;
37
38 // Execution Phase
39 For Each application
40   buf2 = getdata (Applicationdata);
41   IBR = getInBaseMemoryAddr (buf2);
42   OBR = getOutBaseMemoryAddr (buf2);
43   IMD = getInDataRegs (buf2);
44   OMD = getOutDataRegs (buf2);
45   Frm = getTotalFrames (buf2);
46   SndDPro (IBR, OBR, IMD, OMD, Frm, ACF, CFR0);

```

The boot processor retrieves the bitstream for each IP of each application, which is combination of frame headers and data. The header contains data such as the location of the start frame, number of frames, and CFR identifier. By keeping track of all frames of running applications in the system, the boot processor can spot if an active frame would be overwritten. After this check, the CFR is informed of the start frame location (SFAR), the number of frames (TFR), and the actual bitstream frames.

Note that the first IP of an application to be set up is the data processor, which remains active for the duration of the application because it provides manages transitions between sub-applications (described later).

The clock generator for a soft IP is also modeled. Clocks can be programmed and switched on/off by writing to memory-mapped (MMIO) registers via the NOC. Similarly, the soft IP can be reset by MMIO.

After configuring and initializing all the IPs, the required (design-time) input and output address ranges (IMD, OMD) are retrieved from the bitstream memory, and are stored in boot processor. Finally, the data processor is notified about this information, and fetches the required data from memory and forwards it to the first IP in the pipeline. In due course it will receive output data from the last IP in the pipeline. This concludes the configuration and execution phases for a single application. The data processor knows when the application finishes by monitoring the amount of data it receives from the pipeline, and comparing it

with OMD. When the application has finished, the data processor notifies the boot processor, who then updates its data structures on application, CFR, and frame activity, and stops the IPs (by resetting them, and switching off their clocks).

Application transitions are quite fast (see Section 7.1). As long as independent applications use different CFRs, the configuration, starting, and stopping of applications does not affect other running applications because the HWNOG is composable. The role of the boot processor lies in the creation and destruction of individual IPs by interacting with CFRs, described next.

## 5.2. Configuration and Functional Region

The CFR contains a two-dimensional array of configuration frames that determine its logical function. It also contains a clock controller that is memory-mapped, i.e. multiple clocks with programmable frequencies can be enabled or disabled by writing to registers that are accessible over the NOC. Similarly, a reset controller is present. The CFR also contains circuitry to handle incoming configurations. A configuration starts at a given location in the CFR and contains a number of frames. An address decoder is used to place frames at the correct location.

A CFR is configured by its bitstream with (one or more) soft IPs and their accompanying soft NI shells. Each NI shell receives data of 32 bits from the NI kernel at the NOC frequency. It converts this data to the appropriate data width and frequency for the soft IP. Similarly, other NI shells receive data from the soft IP and convert it to NI kernel format and speed. The use of credit-based end-to-end flow control between NI kernels, and link-level hand-shakes between NI kernel, NI shell, and IP ensures safe data transfer even when IPs (and the data processor) operate at different frequencies [15].

A CFR can be programmed with different soft functions at different points in time, which must be modelled in SystemC. This is done by modeling the loading of bitstreams, inspecting the bitstream, and based on this selecting the behavioral model of the soft IP that has been configured. In this manner, it is not necessary to model the behavior of the LUTs and so on of the CFR in a bit-and-cycle accurate manner, which would be prohibitively complex and slow. The clock and reset behavior is modelled, as is the programming of the IPs by the data processor. The CFR architecture and its SystemC model are shown in Figure 4.

The boot processor and the CFR functionality are fundamental to our hardwired NOC platform. The data

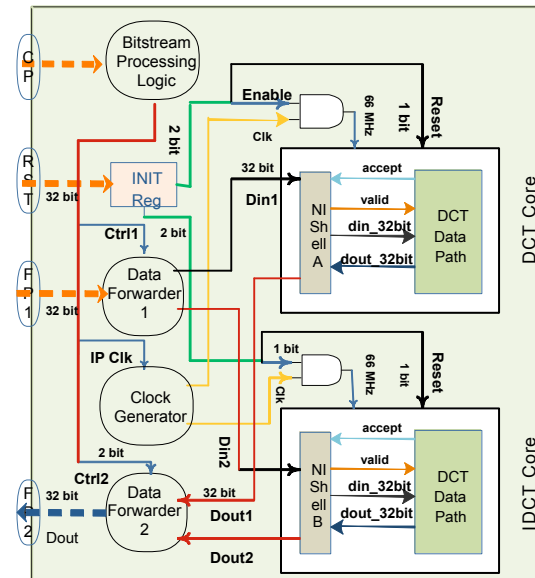


Figure 4. SystemC Model of a CFR.

processor, described next, illustrates one particular way to use the platform: dynamic partial reconfiguration of applications.

## 6. Dynamic Applications

The data processor, illustrated in Figure 5, is essentially just another soft IP. However, it serves two important functions. First, it acts as a *source and sink* for the processing pipeline, i.e. supplies the first IP in the pipeline with data, and receives the results from the final IP in the pipeline. This data is read from and stored in memory, which can be internal or external to the data processor. In our experiments it is internal. In a real system, the input/output data could come from the FPGA I/O, and would be handled in the same manner.

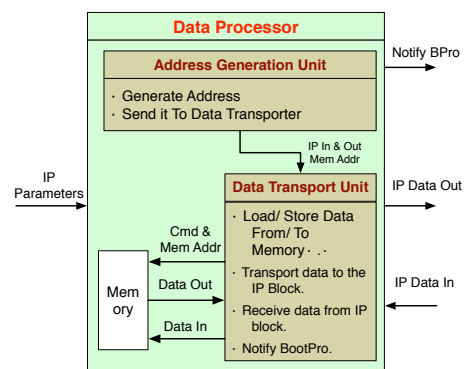


Figure 5. Data Processor.

Second, the DPro is aware of the *progress* of the

pipeline, i.e. knows when all source data has been processed (if ever), and when all result data has been received (and hence the pipeline is empty). This information is application specific, and is communicated to the BPro. Section 5 defined how the boot processor sets up and removes an individual soft IP. In this section we describe how the DPro uses this to tear down (part of) the application.

As mentioned, the DPro contains one or more memories that act as sources and sinks to the rest of the pipeline. DPro uses an address generator shown in Figure 5 to generate the required input/output addresses and forwards it to data transport unit (DTU). The DTU has information about the application pipeline such as the number of input/output channels of the application, where the data of these channels is stored in the memories, and how much information must be sent or received. This information is stored in local memory and has been received from the BPro. It also allows the data processor to know when the application pipeline is empty and hence finished. It signals this to the boot processor, which can then reconfigure the system. This infrastructure enables dynamic starting and stopping of entire applications, as already described in Section 5.

## 7. Experiments and Results

We modeled our HWNOC platform in SystemC using the design flow of [20]. We use a simplified H.264 application encoding with Quarter Common Intermediate Format (QCIF) resolution, with behavioral models of the six IPs as shown in Figure 6. Synthesis of the VHDL implementations of the residue and DCT IPs on a Virtex-4 XC4VLX200 chip using Xilinx ISE 8.2 provided their frequencies, which were used in SystemC. The size of their bitstreams was estimated from their LUT areas, using the equation  $(IP\ LUTs * frames\ per\ column) / (LUTs\ per\ CLB * CLB\ per\ column)$ . For Virtex-4 [21] a single CLB contains 8 LUTs, and a column contains 22 frames and 16 CLBs. The IDCT and reconstruction IPs are similar to the DCT and residue, and we used the same numbers for them. The quantizer IP speed and area were estimated to be between those of the DCT and residue.

We run a single use-case containing two applications (A1, A2). We assume that A1 does not fit in the available reconfigurable area, and hence is partitioned in A1a and A1b. Figure 6 shows the IPs that constitute the applications. There are two masters (Bpro, Dpro) and eight slaves that form A1 and A2 (five of which are shown in Figure 1). The NOC contains six routers and six NI kernels.

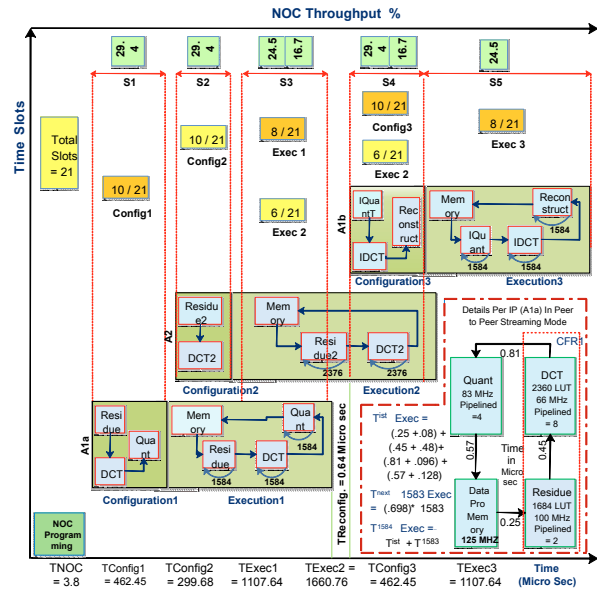


Figure 6. Application Timing Analysis.

Our simulations aim to show the correct functioning of application starting and stopping, system performance, and evidence that applications do not interfere.

### 7.1. Undisrupted Configuration And Execution

This section provides experimental evidence of undisrupted bitstream and functional data transportation. A single execution of all the A1a and A1b processes one  $4 \times 4$  pixel-block, and 16 such pixel-blocks constitute single Macro Block (MB). For A1 to process a complete QCIF frame of 99 macro block, each IP executes 1584 times. In our example, A2 computes 1.5 QCIF frames to ensure it executes while A1a and A1b are configured and executed.

The phases the DCT IP goes through, like the other IPs, are shown in Figure 7. The DCT IP has a pipeline depth of 8 as shown with vertical lines and runs at  $66\text{MHz}$ . Prior to bitstream loading, the HWNOC is programmed by setting up connections to all 5 CFRs, as described in Section 3. It takes approximately  $3.8\mu\text{s}$  before A1a's configuration data appears on the network, see Figure 6.

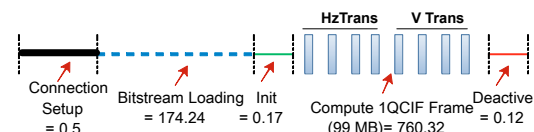


Figure 7. DCT Phases ( $\mu\text{s}$ ).

Execution of A1a starts after programming the NOC, the data processor send data to the residue IP. The residual data is forwarded to DCT block on  $4 \times 4$  sub-block level, which applies integer pixel transformation on the input data. The DCT block forwards its output to the data processor that stores the intermediate results in its local memory, which serves as the starting input for A1b, when A1a has finished. The IPs of A1a (and A1b, and A2) operate as a true pipeline, i.e. they process their data and communicate over the NOC concurrently and in a pipelined fashion. The IPs run at different clock speeds, and flow control ensures that no data is lost or overwritten. It takes approx.  $1100\mu s$  to encode a QCIF frame encoding by A1a, as indicated in Figure 6.

In this example, 21 time-division multiplexing (TDM) time slots in the NOC are used to divide link bandwidth between different connections [4]. Phases S1-S5 in Figure 6 indicate the number of slots allocated to different phases (configuration, function) for each application. The NOC throughput line shows the percentage utilization of the NOC. A1a and A2 configuration each use 29.4%, concurrent execution of A1a and A2 (41.2%), concurrent configuration of A1b and execution of A1b (46.1%), and A1b execution only (24.5%).

The loading of A2's bitstreams take place in parallel with A1a's execution. Figure 8 illustrates that the loading of the bitstream of A2's residue IP is not affected by A1a because the bitstream latency is constant.

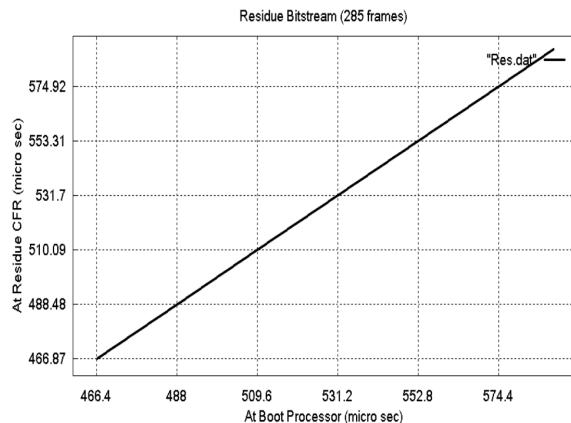


Figure 8. Bitstream Loading with Fixed Latency.

The data processor of A1 observes when A1a has finished executing. It notifies the boot processor, which configures A1b and starts it. The entire transition takes  $0.64\mu s$ . As explained before, this requires reprogramming the HWNOC, loading bitstreams, enabling the clock, resetting and programming the IPs of A1b. During all this, A2 operates in parallel without any

interference. Note that the resources that were allocated to A1a, such as NOC time slots and CFRs, are re-allocated to A1b. This can be observed in Figure 6, where the 10 time slots of A1a in phase S4 are re-allocated to 8 time slots of A1b in phase S5.

## 7.2. Configuration And Functional Performance Comparison

We compared the latency of configuration using our platform with a HWNOC, with that in a conventional FPGA. It takes  $0.165\mu s = 173.4\mu s/1051$  to transport a single bitstream frame of application A1a when no other applications are present in the system, and we use the full capacity of the platform. It takes  $0.44\mu s$  if we reserve capacity for A2 (whether it is present or not then makes no difference). In a traditional FPGA with a 32-bit 60 MHz SelectMap interface, a frame requires  $0.7\mu s$  to configure the destination region.

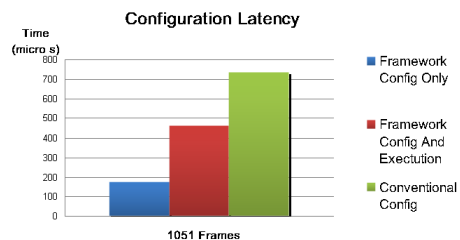


Figure 9. Configuration Comparison.

To compare the functional performance, we consider the transport of the residue IP's data through the NOC. We compare our HWNOC that runs at  $500MHz$  with several soft implementations of it on a Virtex-4 FPGA. Our measure is the delay, i.e. the time in the router network, of the connection with the longest path (boot processor to residue IP). Table 1 shows the time to transport half of the current and predicted Macro block (MB) data, each comprising 128 bytes, to the residue IP, for a number of NOCs. The times are computed by:  $((Flithoptime * (Total Flits + Number of hops + 1))$ . In the equation  $Flithoptime$  accounts for time taken by a flit of 3 words in each router. It is equal to  $0.0250$ ,  $0.0333$  and  $0.0462\mu s$  for synthesized  $2 \times 2$ ,  $3 \times 2$  and  $3 \times 3$  NOCs, respectively. The last term represents the output queue at the destination. Compared to a conventional soft NOC, our HWNOC can transport data approximately 5.5 times faster, for  $3 \times 2$  network.

The experiments in this section illustrate that multiple applications can be started and stopped concurrently, without any interference. Reconfiguring an

Table 1. Functional Comparison ( $\mu s$ ).

Network	#Hops	soft NOC	HWNOC
2x2	3	0.650	0.156
3x2	4	0.899	0.162
3x3	5	1.29	0.168

application takes a minimum time of  $0.64\mu s$ . Reconfiguration using the HWNOC is faster than a soft NOC, even when not all of its bandwidth is available, e.g. because other applications operate concurrently.

## 8. Conclusions

In this paper, we modeled a FPGA architecture that uses a hardwired NOC (HWNOC) to transport configuration (bitstream) and functional data. We describe the run-time procedures to configure, program, and run soft IPs. This basic infrastructure is then used to dynamically start and stop entire applications, and also sub-applications. The latter is useful when an application does not fit in the resources (configuration regions) allocated to it. Dynamically swapping sub-applications then enables the entire application to execute anyway, although at a lower speed. Our platform is composable, which means that starting and stopping of applications does not affect concurrently operating applications (and vice versa).

We modeled the platform in cycle-accurate transaction-level SystemC, together with soft IP blocks. In particular, we model bitstream loading and frame placement of soft IPs, soft IP clock management and reset, the programming of HWNOC and IPs, and their functional operation. We simulated the concurrent configuration and execution of two small applications, one of which was split in two sub-applications.

We compared the performance of a conventional FPGA with a soft NOC and dedicated configuration interconnect with our HWNOC platform. Bitstream loading is faster in our platform.

## References

- [1] Xilinx Inc., "Virtex-4 Data Sheets," 2005.
- [2] Altera Inc., "Stratix Data Sheet," 2005.
- [3] K. Goossens, et al., "Hardwired networks on chip in FPGAs to unify data and configuration interconnects," in *NOCS*, Apr. 2008.
- [4] K. Goossens, et al., "The  $\mathcal{A}$ ethereal network on chip: Concepts, architectures, and implementations," *IEEE Des. and Test of Comp.*, vol. 22, no. 5, 2005.
- [5] A. Hansson, et al., "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM TODAES*, vol. 14, no. 1, 2009.
- [6] M. L. Silva, et al., "Support for partial run-time reconfiguration of platform FPGAs," *J. of Sys. Arch.: the EUROMICRO Journal*, vol. 52, Dec. 2006.
- [7] H. Nikolov, et al., "Efficient automated synthesis, programming, and implementation of multi-processor platforms on FPGA chips," in *FPL*, Aug. 2006.
- [8] C. Bobda, et al., "A dynamic NOC approach for communication in reconfigurable devices," in *FPL*, 2004.
- [9] A. Kumar, et al., "An FPGA design flow for reconfigurable network-based multi-processor systems on chip," in *DATE*, Apr. 2007.
- [10] S. Lukovic, et al., "An automated design flow for noc-based MPSoCs on FPGA," in *RSP*, Jun. 2008.
- [11] E. Carvalho, et al., "PaDReH - a framework for the design and implementation of dynamically and partially reconfigurable systems," in *SBCCI*, 2004.
- [12] R. Hecht, et al., "Dynamic Reconfiguration with hardwired Networks-on-Chip on future FPGAs," in *FPL*, Aug. 2005.
- [13] R. Gindin, et al., "NoC-Based FPGA: Architecture and Routing," in *NOCS*, May 2007.
- [14] H. Ito, et al., "The plastic cell architecture for dynamic reconfigurable computing," in *RSP*, 1998.
- [15] A. Rădulescu, et al., "An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming," *IEEE TCAD*, vol. 24, no. 1, Jan. 2005.
- [16] A. Hansson, et al., "Trade-offs in the configuration of a network on chip for multiple use-cases," in *NOCS*, 2007.
- [17] A. Hansson, et al., "Undisrupted quality-of-service during reconfiguration of multiple applications in networks on chip," in *DATE*, Apr. 2007.
- [18] A. Hansson, et al., "A unified approach to mapping and routing on a network on chip for both best-effort and guaranteed service traffic," *VLSI Design*, May 2007, Hindawi Publishing Corporation.
- [19] S. Murali, et al., "SUNMAP: A tool for automatic topology selection and generation for NOCs," in *DAC*, Jun. 2003.
- [20] K. Goossens, et al., "A design flow for application-specific networks on chip with guaranteed performance to accelerate SOC design and verification," in *DATE*, Mar. 2005.
- [21] Xilinx Inc., "Virtex-4 Configuration Guide."