Composable And Persistent-State Application Swapping On FPGAs Using Hardwired Network on Chip

Muhammad Aqeel Wahlah[1] and Kees Goossens[1,2]
[1] Computer Engineering, Delft University of Technology, Delft, The Netherlands
[2] Research, NXP Semiconductors, Eindhoven, The Netherlands
aqeel@ce.et.tudelft.nl and kees.goossens@nxp.com

***Abstract*—We envision that future FPGA will use a hardwired network on chip (HWNoC) [13] as a unified interconnect for functional communications (data and control) as well as configuration (bitstream for soft IPs). In this paper we present a reconfiguration methodology which makes use of such a platform to realize composable inter-application communication and persistent-state intra-application when run-time partial reconfiguration is performed. The proposed methodology also ensures that the required performance constraints of the dynamically swapped in application are fulfilled. We describe the approach and steps required to achieve the above objectives. We model the application dynamic swapping behavior in cycle-accurate transaction-level SystemC which includes bitstream loading, HWNoC programming, clocking, reset, computation.**

I. 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]. These platform-based FPGAs have gained popularity for realizing systems as complex as multi-processor system on chips (MPSoCs) due to their fast time-to-market, low NRE costs and in-field product upgrade benefits. These FPGA [1] based systems, can embody concurrent execution of multiple applications and in different configurations called use-cases.

Before proceeding further we define terminology. A **soft** IP is mapped on FPGA reconfigurable computational blocks (LUTs) whereas an IP is hardwired or **hard** when it is directly implemented in silicon e.g. power pc. 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. A **use-case** is defined as the number of applications that can execute in parallel.

However, inter-application dynamic reconfiguration becomes inevitable when not all the applications in a use-case can co-exist on the FPGA. Also, intra-application reconfiguration is required when an application does not fit in completely due to area constraints. It incorporates facing **critical and non trivial issues** during reconfiguration which include:

1) realizing composablity i.e. the dynamically inserted (sub)application does not effect other (sub)applications, as long as their allocation remains unchanged. Also there must not be any interference among the executing applications and their occupied system resources,
2) implementing persistent-state i.e. the state-information (spread at multiple places in the system) of the sub-application (SubApp) must be saved, when it is swapped out. It is essential to avoid unpredictable behavior of the system,
3) ensuring throughput and latency demands for the dynamically inserted (sub)applications on an FPGA.

Several dynamic reconfiguration studies provide run-time swapping of functionalities without disturbing the existing ones [3], [4], [6], [9] as well as mechanisms [10], [12] to implement safe-state transitions. However, a system level approach is missing that could **concurrently address** them with guaranteed performance constraints on an FPGA.

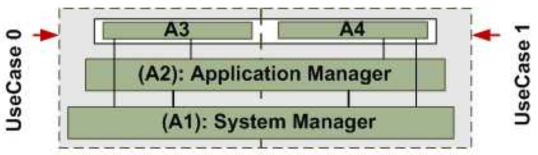


Fig. 1. 3-Tier Infrastructure

We provide an integrated system level methodology which uses a **modular and 3-tier infrastructure** [15] as shown in Figure 1 to achieve the above objectives. It makes use of the the System Manager (SM) as the foundation layer and an Application Manager (AM) per application to ensure composablity across the applications. Services which are related to application loading and resource allocation (i.e. NOC programming) are always provided by the SM to ensure that one application can never affect another. On the other hand an AM provides intra-applications services which include I/O and storage to the clients and enforcement of data integrity between the sub-applications that are dynamically swapped in/out.

The remainder of this paper is structured as follows. Section II positions our methodology with respect to the related work. Section III elaborates the architectural details of the target platform. It is followed by Section IV which describes our methodology by using the 3-tier infrastructure for realizing dynamic composable application swapping. Section V shows a worked example and we conclude in Section VI.

II. Related Work

We discuss run-time reconfiguration researches with respective to (w.r.t.) the features which are provided by our methodology i.e. composability and persistent-state swapping of the modules/applications in parallel with the QoS requirement assurance.

A. Composable Swapping

The authors in [2], [3], [4] place the dynamically inserted module in vertical slots which allows the modules to be attached at any location whereas [5] provides a way to place hardware modules of predetermined size and positions, above each other. To connect the modules [2] uses a reconfigurable multiple bus (RMB), [3] uses an NOC, [4] uses lookup tables and [5] uses bus macros. [6] achieves dynamic on-demand reconfiguration by making use of a run-time system software on MicroBlaze for controlling reconfiguration and message handling. [7] at run time allocates FPGA resources by using a centralized resource manager. [8] uses a reconfigurable system based on square-shaped and arbitrary-sized *swappable logic units* (SLUs) which are arranged in mesh and communicate with each other through a small communication buffer. A dynamic instruction set architecture based approach is used in [9] where authors make use of dynamically rotating instructions for runtime swapping of reconfigurable modules.

In contrast to our application level composability, the above-mentioned approaches [6], [9] take into account single application and realize task level composable behavior of the system. With these approaches guaranteeing that an application's requirements are met is difficult due to possible resource fragmentation over time. [7] takes into account concurrent execution of multiple applications but the mechanism to implement persistent-state and QoS guarantees is missing. The remaining approaches [3], [4] don't explicitly state the level of compsability. Importantly, the above methods e.g. [3], [4], [5] are more concerned about providing the communication among the dynamically placed modules rather than handling the important issues of *stability of prior services*, mechanism to assure *safe-state transition* and *QoS guaranteed resource allocation* with the addition/removal of modules.

B. Persistent-State Swapping

Works in [10], [11], [12] assume that tasks can only migrate at pre-defined execution points. Safe-state task level transition in [10], [11] is achieved by making use of special input queue to collect all unprocessed data whereas in [12] all configuration data is read back and state extraction is performed after reading the configuration data.

With above-mentioned approaches [10], [11], [12] the state information of a task is distributed: a) within the tasks and b) in between the tasks. The state preservation within the task/IP causes complex issues w.r.t. the register states and clock phase to preserve data and timings. In contrast, the IPs in our system are stateless and an AM is the one who triggers the reconfiguration request after the SubApp IP achieve the required execution granularity. However, the inter-IP state exists which is preserved by an AM by providing the persistent storage in between the SubApp swapping. We don't allow cycles during SubApp swapping therefore the existing SubApp's pipeline is completely flushed and preserved in an AM before starting the next SubApp.

We first explain the target platform first presented in [13] and which is extended in this paper. It is used by our reconfiguration methodology to realize structural and communication mapping of the use-case (sub)applications with the objectives of composability and persistency.

III. Target Platform

This section describes how a hardwired NoC is embedded in an FPGA [13]. It expounds the architectural details of the constituting components which include: configuration functional regions (CFR), communication infrastructure consisting of routers and network interfaces (NI) and boot processor (BPro). It concludes by mentioning limitations of the current architecture.

In our HWNoC architecture the FPGA chip comprises number of configuration functional regions (CFRs) [14] whose architecture is illustrated with $CFR1$ in Figure 2. Each CFR has local configuration infrastructure to handle the incoming bitstream. However, the CFRs are not isolated at functional level and therefore do not restrict the placement of an IP spanning fully or partially in multiple regions. The CFR's configuration infrastructure constitutes a newly introduced port and logic e.g. address decoder and dedicated registers to write incoming bitstream onto the desired location. At the moment the CFRs are equal sized and entail multiple 16 CLB columns. Each CLB column is called minimum configuration region (MCR) which is the the least (re)configuration unit in our design. Additionally no single MCR is shared between the two IP cores. Each CFR has a local *Clock generator* which is memory-mapped, i.e. programmable clock frequency for the required MCR can be generated by writing to registers that are accessible over the NoC. Similarly a memory-mapped *Reset generator* is present which as per required enables or disables IP cores from processing input data.

The boot processor is a programmable hardwired IP that bootstraps the system. It contains number of registers to hold bitstream and application related information required to instantiate soft IPs by loading bitstream from the bitstream memory to the appropriate CFRs. The underlying infrastructure which transports IP traffic consists of routers and network interfaces (NIs). The former is hardwired whereas the later is further split into NI kernels and NI shells. NI kernel is hard and its architecture is explained in [16].

Our architecture supports multiple IPs in a single CFR and inter-IP communication only through NoC because then the

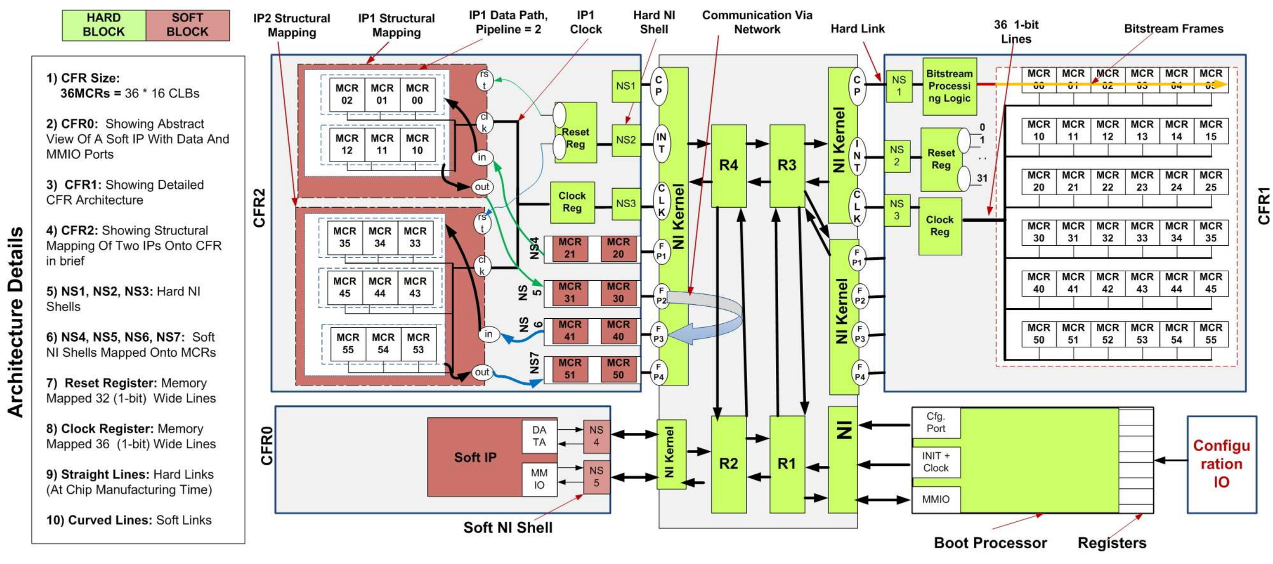

HARD BLOCK
SOFT BLOCK
Architecture Details
1) CFR Size: 36MCRs = 36 * 16 CLBs
2) CFR0: Showing Abstract View Of A Soft IP With Data And MMIO Ports
3) CFR1: Showing Detailed CFR Architecture
4) CFR2: Showing Structural Mapping Of Two IPs Onto CFR in brief
5) NS1, NS2, NS3: Hard NI Shells
6) NS4, NS5, NS6, NS7: Soft NI Shells Mapped Onto MCRs
7) Reset Register: Memory Mapped 32 (1-bit) Wide Lines
8) Clock Register: Memory Mapped 36 (1-bit) Wide Lines
9) Straight Lines: Hard Links (At Chip Manufacturing Time)
10) Curved Lines: Soft Links
IP2 Structural Mapping
IP1 Structural Mapping
IP1 Data Path, Pipeline = 2
IP1 Clock
Hard NI Shell
Communication Via Network
Hard Link
36 1-bit Lines
Bitstream Frames
CFR2
CFR1
CFR0
Reset Reg
Clock Reg
NS1
NS2
NS3
NS4
NS5
NS6
NS7
NI Kernel
R1
R2
R3
R4
NI
Bitstream Processing Logic
Soft IP
DATA
MMIO
Cfg. Port
INIT + Clock
MMIO
Configuration IO
Soft NI Shell
Boot Processor
Registers

Fig. 2. HWNoC Architecture With **Detailed CFR Architecture**, **Structurally Mapped IPs**, **Hard Soft** Partitioning

application manager can observe the state of communication channels which is necessary for safe reconfiguration. However the architecture limits by having: no MCR sharing between the IPs, no read-back port from the CFRs, no relocatable or displaced bit-streams and no IP span in the multiple CFRs.

The next section expounds the proposed methodology which uses the above-mentioned architecture as the target platform to realize the composable and stable-state application reconfiguration.

IV. Dynamic Reconfiguration Methodology

The dynamic reconfiguration methodology first enforces composability across the applications before loading an application and at the end of its execution ensures persistent-state transition particulary among the SubApps of an application. To achieve these objectives it carries out a systematic and close interaction across all the 3-tiers of the HWNOC reconfiguration model as explained below.

A. Composable Inter-Application Enforcement

In a real scenario the individual (sub)applications are combined into use-cases where an application can span in multiple use-cases, Figure 3 (A). Since, each use-case represent different combinations of application(s) therefore could have its own QoS requirements e.g. bandwidth and latency constraints that the communication infrastructure must efficiently accommodate for the required performance constraints.

Composable swapping of an application requires: no impact and conflict with the logic (FPGA) and communication (NOC) plane resources of the existing application(s). The logic plane resources include: CFR(s) locations, in/out memory locations whereas communication plane resources include: data connections and resources associated with each data connection e.g. time division multiplex (TDM) slots and flow control credits. Composable swapping faces another critical challenge when the incoming application triggers the use-case transition i.e. the incoming application posses different use-case from some or all of the already executing application(s). In that situation it becomes essential to avoid use-case conflict among the applications and to preserve glitch less execution for those application(s) which span in both the use-cases.

To cope with above challenges the proposed methodology allocates a virtually isolated platform for each application. It ensures to avoid interference with respect to existing applications and thus allows at-any-time dynamic addition/removal of the functionalities without taking care of system status.

To realize such a platform the reconfiguration methodology first calculates the communication plane resources for each application according to its QoS requirements. It is achieved at compile time by following the principle of [17]. It implies to consider all the use-cases in which the application spans and to allocate the required resources such that not only its QoS requirements are fulfilled but also the use-case transition does not impact its execution. Our methodology reserves NoC resources by 1) choosing the most critical application flow across the use-cases that is not yet assigned. The most critical flow is selected on the basis of its use-case span and the aggregation of throughput and latency demands. Figure 3 (B) and (C) illustrate this concept where *Ap2* and *Ap1* are chosen in prior to *Ap3* because of their execution span in larger number of use-cases. Since, *Ap2* throughput requirements are higher than that of *Ap1* therefore its flow will be chosen in prior to *Ap1*. 2) After selecting the required application flow, the available resources in all the use-cases which are spanned by that particular flow are derived. 3) It is followed by resource reservation according to the Quality of Service demands for that particular application flow and then distributing the reservation across the data structures of the affected use-cases.

The reconfiguration methodology at run time, enforces composability among the applications by making use of the system manager for application-specific actions which include: stream

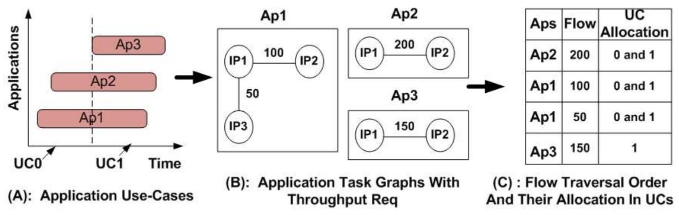


Fig. 3. Application Flow Traversal Order And UC Allocation

bitstream for the application cores and their initialization, stoppage, allocation and deallocation of the NoC resources. The SM keeps record of each application information which include its: use-cases, Ids, corresponding managers, target CFR(s), IP(s) with corresponding frequencies, configuration and functional memory addresses. In prior to load a recently invoked application the system manager first makes a check to avoid a possible use-case and FPGA resource conflict of that incoming application to the existing ones. On finding no such conflict the SM loads the application IV-B and allocates the respective NoC resources after ensuring no TDM slot contention. Currently, the SM waits for the conflicting applications to execute completely before loading the invoked application by using the procedure as explained in section IV-B.

B. Load And Program Application

Loading of an application is illustrated through Figure 4. It initiates by using the SM configuration services which after extracting the bitstream from the configuration memory send it to the destination CFR over a fixed-latency connection. At the destination CFR, circuitry (address decoder and registers) to handle incoming bitstream headers and frames is present which places the incoming bitstream at the correct locations as elaborated in [14]. This way an IP is configured and the bitstream loading process iterates for all the IPs that can be placed in that CFR. Afterwards Initialization is carried out per IP basis which incorporates sending clock and reset information to memory-mapped clock and reset generators. The above phases end up by closing the appropriate connections over the network to release the resources. This way a soft-IP is physically mapped onto the reconfigurable fabric.

Once all the application cores are structurally mapped, TDM based data connections among the candidate IPs are established through the system manager. It accounts for reserving across the network: the path to communicate with peer, time slots to send/receive data at required rate and credit counters to avoid dropping of data with IPs of different clock frequencies and pipeline depths.

The appropriate AM is afterwards programmed with its client application's input/output base addresses, strides, data ranges and number of executions to perform. In case of single application comprising multiple SubApps, the I/O addresses are supplied to an AM after the first SubApp is initialized.

On receiving the application parameters an AM by using its local address generation unit (AGU) calculates the required input/output addresses, fetches the data from input locations and forwards it to the required core on a appropriate data connection. An AM interacts with the executing application both in forward (supplying input) and backward direction (receiving application output). The later also keeps traces of number of executions and dynamically chosen storage location for the intermediate SubApps of the complete application. At the end of the application execution an AM triggers reconfiguration for the next SubApp by sending a notification to the SM which in turn concludes the loading of an application. Once required amount of data is received, the procedure to ensure persistent-state transition is called which is explained below.

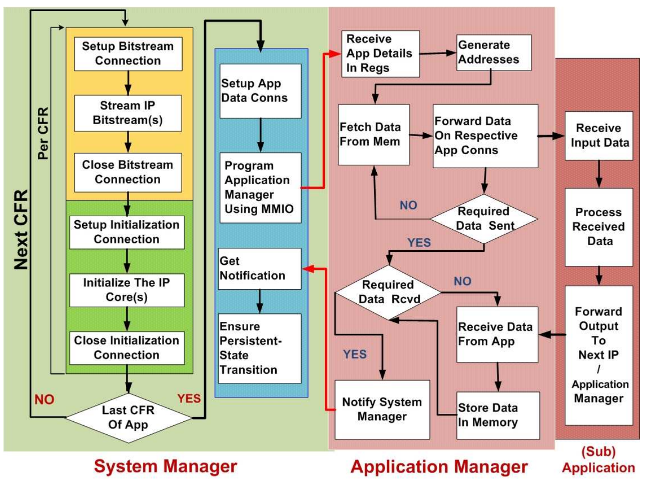


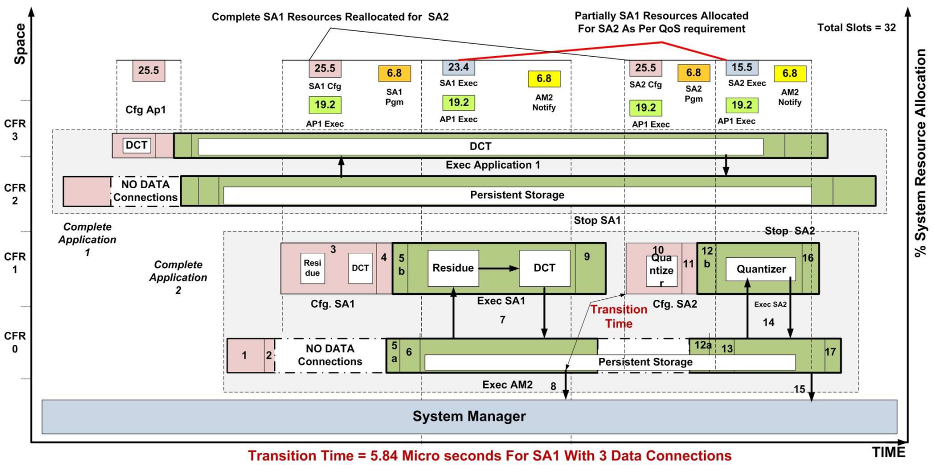
Fig. 4. Loading An Application

C. Persistent-State Intra-Application Assurance

Since, in our methodology the application that is dynamically swapped in and out is split into sub-applications Figure 1, as the data flows through the system. It incorporates that the state between the sub-applications is not lost during reconfiguration.

It is achieved first by tearing-down the application data connections by using a systematic procedure, proposed in [18]. It accounts for blocking the source shell from emitting new transactions, emptying input/output queues, and clearing the slot table entries associate with to-be-shutdown connections. Disabling the application computational resources comes next which starts with opening a reset connection to NI(s) associated with application IPs and afterwards sending a 32-bit reset signal to disable IPs from processing further.

Notably all the soft IP of the application are reconfigured, except the application manager. The reason as explained before an AM sees consistent view over multiple sub-applications through persistent storage in its local memory. The next sub-application that is scheduled then operates on this data. Our current implementation is rather limited: we allow no cycles in the application, and completely empty the sub-application's pipeline before starting the next sub-application. Thus soft IP may be pipelined, but must be able to empty their pipeline when no new data arrives. We evaluate the above discussion in the next section.



Application Independent Timings

Steps	Details	Time (micro s)
(2, 4, 11)	Initialization Of IPs In Their CFRs	0.63
(6, 13)	Program AM For Sub-App data ranges	0.97
(8, 15)	Notification From AM to SM	0.24
(17)	Shutdown Application Manager (AM)	1.25

Sub-Application1 (SA1) Timings

Steps	Details	Time (micro s)
(3)	Bitstream Loading	653
(5a, 5b)	Application Data Connection Setup	2.9
(7)	Application Execution	460.3
(9)	Application Shutdown	5.6

Sub-Application2 (SA2) Timings

Steps	Details	Time (micro s)
(10)	Bitstream Loading	355
(12a, 12b)	Application Data Connection Setup	1.02
(14)	Application Execution	401.5
(16)	Application Shutdown	3.9

Fig. 5. Application **Temporal Analysis** With **% System Resource Reservation** At Different Stages

V. Experiments And Results

We exercised the dynamic reconfiguration methodology in SystemC using the design flow of [19]. We used a simplified H.264 application with behavioral models of the three IPs as shown in Figure 5 to encode Quarter Common Intermediate Format (QCIF) resolution video frames. Synthesis of the VHDL implementations of the Residue and DCT IPs on a Virtex-4 XC4VLX200 chip using Xilinx ISE 8.2 provided their MHz frequencies which were used in SystemC, table I. The size of their bitstreams was estimated from their kLUT areas using the equation ***(IP LUTs * frames per column) / (LUTs per CLB * CLB per column)***. For Virtex-4 [20] a single CLB contains 8 LUTs, and a column contains 22 frames and 16 CLBs. The Quantizer area and frequency were estimated to be between the DCT and the Residue respective values.

TABLE I
Application IP Synthesized Area, Frequency And Reconfiguration Time

IP	Area ($kLUTs$)	Frequency (MHz)	Bitstream (*Frames*)	(Re)config Time (μs)
RESIDUE	**1.68**	**100**	**285**	**273.6**
DCT	**2.36**	**66**	**396**	**380.16**
Quantizer	**2.21**	**75**	**370**	**355.2**

We assume that the application1 (DCT only) and application2 (Residue +DCT + Quant) both run in parallel but with the difference that application2 does not meet the required area constraints and therefore is sub-divided into two Sub-Apps (SA1 and SA2) which are configured and executed in separate use-cases. In addition each use-case comprises the system manager and an application manager per application. The NoC contains 4 routers and NI kernels with respective FIFO sizes of 24 and 41 words. Optimistic average data delay between the two IPs which communicate over the network can be calculated as: ***Total Iterations (FlitTime (Flits Per Iteration + Hops + 1) + FlitTime(Total Slots Allocated Slots))***. Here, Flits Size = $3words$, FlitTime = $0.006\mu s$, Flits Per Iteration = $AllocatedSlots$, Total Iterations = $Ceil(TotalFlitsToSend/FlitsPerIteration)$.

Results provide the evidence and analysis of the application mapping onto the HWNOC platform, persistent-state intra-application swapping and composablity of the system during inter-application reconfiguration, in the next section.

A. Temporal Analysis Of Applications Mapping

Figure 5 expounds the FPGA and NoC resource reservation details for both the applications during the load, program, start and stop phases. In the discussions to follow we will discuss the timing details for the application 2 which comprises two sub-applications (SA1 and SA2).

Each phase is preceded by programming the NoC, as illustrated in [18], in $0.18\mu s$ to $0.24\mu s$, so that the data can reach the required location. This time has been included in the preceding discussions while mentioning individual phase delay. Notably, the source and the destination in our network at the maximum can be three nodes apart and with each extra router an additional delay of $0.006\mu s$ is encountered.

Bitstream loading for the 681 frames of SA1 is carried-out by the SM on a fixed and low latency connection and takes $653\mu s$, Figure 5 step(3). It accounts for the 41-word bitstream frame to be transported the rate of $0.96\mu s$. Afterwards the SM initializes its IPs in $0.63\mu s$ by programming memory mapped reset and clock generator in the destination CFR, Figure 5 step(4). It is followed by setting-up of three data connections for those IPs in approximately $2.9\mu s$, Figure 5 step(5). As the last step before the SA1 execution the respective AM is programmed by the SM in $0.97\mu s$ with 48-words application parameters which comprise full application's I/O addresses and ranges, Figure 5 step(6). Afterwards the SA1 execution is carried out where the single execution of the its IPs process one 4x4 pixel-block, and 16 such pixel blocks constitute single Macro Block (MB). It takes $460.3\mu s$ to process 1QCIF (99MBs) video frame in peer to peer streaming communication fashion, Figure 5 step(7).

Sub-application2 goes through the same phases as shown in Figure 5 steps(10-14) but after achieving a persistent-state

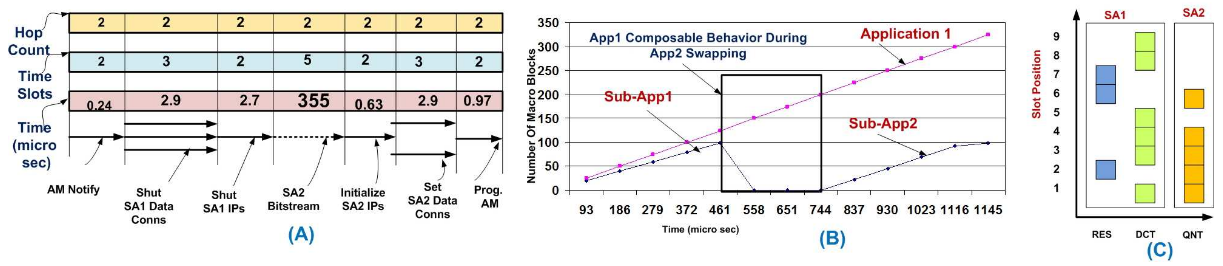


Fig. 6. App2 Micro Details: A)Safe-State Transition, B)Composable Behavior, and C) Allocation

intra-application swapping which is explained in next-section.

B. Persistent-State Intra-Application Swapping

The persistent-state transition initiates with an AM notification to the SM in $0.24\mu s$ on an already established connection, Figure 5 step(8). In prior to load SA2 the transitional delay incorporates various factors for achieving the safe state, as explained with the following equation:

$$(T\_AM_{Notify} + T\_SM_{Response} + \#Conns (T\_Conn_{Shutdown}) + \#CFRs(T\_CFR_{Shutdown})$$

In our case the SM response time is negligible because it is sitting idle and ready to serve at the time of an AM notification. Achieving persistent-state is afterwards triggered by tearing down all the three SA1 connections in $2.9\mu s$ followed by disabling the SA1 IPs in $2.7\mu s$. Figure 6(A) illustrates the resources reserved during safe-switch from SA1 to SA2 with a transitional delay of $5.84\mu s$.

C. Inter-Application Composability

This section explains the composable nature of our dynamic reconfiguration methodology which is evident from the Figure 6(B) where the application1 execution is not affected when an existing application is a) stopped (SA1 after processing the 99MBs) or b) installed (SA2). It is achieved by reserving the TDM slots in the communication plane over the communication channels which ensures that there is not contention of resources at the connection level. These TDM slots realize interleaved yet non-interfering traffics for application bitstream loading and execution over the same network path. The resources which are freed after the removal of an application (SA1) are reallocated fully or partially Figure 5, as per required to the incoming application (SA2). During the resource reallocation SM takes into account the magnitude and positioning of the resources, as shown with Figure 6(C) where Quantizer is allocated with the resources released by Residue and DCT blocks.

VI. Conclusion

In this paper, we presented dynamic reconfiguration methodology which provides composable behavior of the system during the application swapping. It also ensures persistent-state application swapping in addition to run-time allocation for the dynamically reconfigured application. We presented the mechanism to achieve above objectives and modeled the application dynamic behavior in cycle-accurate transaction-level SystemC. The methodology uses SM to provide services of reconfiguration and resource allocation whereas makes use of an AM to ensure run-time memory allocation and persistent-storage for the switching (sub)applications. A detailed analysis is provided in the end to illustrate the composable temporal behavior of our system during the application swapping and safe-state transition in $5.84\mu s$.

References

[1] Xilinx Inc., "Virtex-6 Data Sheets," 2009.

[2] C. Bobda, et al., "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms," in *FPT*, Dec. 2005.

[3] C. Bobda, et al., "A dynamic noc approach for communication in recongurable devices," in *FPL*, Dec. 2005.

[4] M. Hubner, et al., "New 2-Dimensional Partial Dynamic Reconfiguration Techniques for Real-time Adaptive Microelectronic Circuits," in *IEEE Emerging VLSI Technologies and Architectures*, 2006.

[5] P. Sedcole, et al., "Modular dynamic reconfiguration in Virtex FPGAs," in *Proceedings Computers and Digital Techniques*, May. 2006.

[6] M. Ullmann, et al., "An FPGA run-time system for dynamical on-demand reconfiguration," in *IPDPS*, April. 2004.

[7] J. Jean, et al., "Dynamic Reconfiguration to Support Concurrent Applications," in *IEEE Transactions on Computers*, Volume 48 , June. 1999.

[8] G. Brebner, "The Swappable Logic Unit: a Paradigm for Virtual Hardware," in *FCCM*, April. 1997.

[9] L. Bauer, et al.,'Efficient Resource Utilization for an Extensible Processor through Dynamic Instruction Set Adaptation," in *TVLSI*, Volume 16, Oct. 2008.

[10] J-Y. Mignolet, et al., "Infrastructure for design and management of relocatable tasks in a heterogeneous reconfigurable system-on-chip," in *DATE*, 2003.

[11] V. Nollet, et al., "Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles," in *DATE*, Mar. 2005.

[12] H. Simmler, et al., "Multitasking on FPGA Coprocessors," in *FPL*, Springer. 2000.

[13] K. Goossens, et al., "Hardwired networks on chip in FPGAs to unify data and configuration interconnects," in *NoCS*. Apr. 2008.

[14] M.A. Wahlah, et al., "Modeling reconfiguration in a FPGA with a hardwired network on chip," in *RAW*, May. 2009.

[15] M.A. Wahlah, et al., "3-Tier Reconfiguration Model For FPGAs Using Hardwired Network on Chip," in *FPT*, Dec. 2009.

[16] 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.

[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., "Trade-offs in the configuration of a network on chip for multiple use-cases," in *NoCS*, 2007.

[19] 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.

[20] Xilinx Inc., "Virtex-4 Configuration Guide."