The RecoBlock SoC Platform: A Flexible Array of Reusable Run-Time-Reconfigurable IP-Blocks

Byron Navas, Ingo Sander, Johnny Öberg
Dept. of Electronic Systems, KTH Royal Institute of Technology
Stockholm, Sweden
{navas, ingo, johnnyob}@kth.se

Abstract— Run-time reconfigurable (RTR) FPGAs combine the flexibility of software with the high efficiency of hardware. Still, their potential cannot be fully exploited due to increased complexity of the design process. Consequently, to enable an efficient design flow, we devise a set of prerequisites to increase the flexibility and reusability of current FPGA-based RTR architectures. We apply these principles to design and implement the RecoBlock SoC platform, which main characterization is (1) a RTR plug-and-play IP-Core whose functionality is configured at run-time; (2) flexible inter-block communication configured via software, and (3) built-in buffers to support data-driven streams and inter-process communications. We illustrate the potential of our platform by a tutorial case study using an adaptive streaming application to investigate different combinations of reconfigurable arrays and schedules. The experiments underline the benefits of the platform and shows resource utilization.

Keywords—reconfigurable architectures; partial and run-time reconfiguration; system-on-chip; adaptivity; embedded systems

I. INTRODUCTION

Design-and-Reuse has been one of the strategies to overcome the design productivity gap in silicon industry during last decade of the System-on-Chip (SoC) revolution [1] [2]. Now, a new era in reconfigurable computing systems is emerging, where it is possible to port software tasks dynamically into Run-Time-Reconfigurable (RTR) hardware accelerators. This introduces additional complexity to the design space, which makes design task even more challenging.

To cope with system-level design issues in increasingly complex integrated circuits, “orthogonalization of concerns” is important to separate parts of the design process and make them nearly independent, so that complexity can be mastered. Platform based design has been suggested as an essential element to overcome the system design challenges in current embedded systems. It is "important to find common architectures that can support a variety of applications as well as the future evolutions of a given application. To reduce design cost, re-use is a must" [3]. A SoC platform implies HW and SW reusability, which is achieved by a library of Intellectual Property (IP)-Cores and an efficient Hardware Abstraction Layer (HAL). A successful design aims to balance between production cost, development time, and performance.

We are convinced that an approach using high-level models is a good starting point for Design of Embedded Systems. However, most of the design methodologies that exist today are predominantly focused on generating the entire system directly. They fail to consider that a partial reconfiguration can be used to reduce cost and power consumption, by loading critical parts of the design at run-time. However, adoption of partial reconfiguration is limited by the intricacy to generate configurations, which implies time and recursive effort during the layout and physical synthesis stages.

We propose to include high-level models into partial reconfiguration, by creating automatically binding blocks with flexible inter-connections and functionality that can be configured independently during run-time. In this way, they can be stacked in any fashion to replicate process structures modeled and optimized at high-levels of abstraction.

We call this platform the RecoBlock (Reconfigurable-Block) SoC Platform. It introduces the concept of reusable RTR IP-Cores with inter-communication and functionalities reconfigurable at run-time, so that reconfigurable architectures and schedules of the array are not fixed, but defined by software. In addition, we propose a set of requirements for adaptivity, flexibility and reusability of reconfigurable components; which are adopted for the RecoBlock core. A case study that implements a high-level model of function adaptivity is implemented and tested for different combinations of reconfigurable management aspects, which proves the properties of the platform. Our proposal aims to reduce the unnecessary repetitive design steps, allowing high-level designers to rapidly implement and test concepts and concerns regarding reconfigurable computing.

II. RELATED WORK

Some contributions like [4] [5] devise architectures, design methodologies or operating systems for dynamic reconfiguration; conduct performing analysis for key concerns like reconfiguration overhead, dynamic scheduling; or propose methods to efficiently map software tasks to hardware. However, some of them remain as high level models, verified only with custom-made simulators, or implemented in cycle accurate SystemC frameworks; therefore missed implementation constraints would rest validity to those models. For instance, in [6] they present a design methodology and dynamically reconfigurable architectures; which consider arrays of Dynamic Reconfigurable Logic (DRL) blocks, and focuses in dynamic schedule algorithm analysis. However, the model seems very abstract and is not implemented. In the same way, in [7], a dynamic runtime manager for reconfigurable

978-3-9815370-0-0/DATE13/©2013 EDAA

resources that leverages hardware module reuse and inter-module communication is presented. Nevertheless, the concept is proved in a virtual architecture for reconfigurable systems.

In [8], a framework for adapting computing, including layers of software, is implemented on a Virtex-4. It covers issues of communication between partial reconfigurable modules and system. A case study discusses an example that multiplexes two memory controllers (i.e., Flash and SRAM) in one single reconfigurable region. It analyzes context switching overhead between static and reconfigurable implementations. This framework is shown to be functional also in other related works. However, it used Peripheral Local Buses (PLB) and interface like Bus Macros, which are being discontinued by Xilinx. Besides, custom interfaces implemented between partial reconfigurable modules and to external pins seem to be efficient, but reduces reutilization, modularity, and routability. Adaptive computing should be intended for improving processing speed in hardware, not for interfacing external devices. Moreover, reusability, libraries of configurations, or standardization of interfaces is not explicitly addressed.

In [9], authors visualize the importance of a flexible FPGA platform with not-fixed modules, and efficient communication methods in reconfigurable computing. The difficulty of partial configuration design automation is pointed out, mainly because of place and routing constraints when pins and static regions are fixed. The solution is coarse-grained oriented and utilizes two FPGA boards: one for reconfigurable modules and other for a programmable crossbar and external interfaces. This paper obviates implementation details, but presents important concepts towards uniformity and flexibility. The multi-board and granularity approach makes it restrictive.

A method, based in modified Kahn Process Networks, to find optimal templates that fit in a reconfigurable hardware is presented in [10]. The architecture is a complex system consisting in a coarse-grained matrix of processing elements (PE) coupled with a Microblaze (MB) using a Fast Simple Link (FSL) and OPB for other peripherals. Each PE is a fixed functional structure, which includes buffers but communicates only with a fixed number of neighbors, without a global interconnection fabric. Reconfiguring PEs is shown to be faster than partial reconfiguration methods for the same area. To sum up, it is an interesting example of reconfigurable computing system; however it is not a partial RTR one. The array and PEs are fixed. The exclusive FSL is optimal for the high bandwidth required to connect the whole complex array to the system, but each PE cannot access directly to all PEs or system.

III. RECOBLOCK SYSTEM DESCRIPTION

A. *The RecoBlock Concept*

The RecoBlock architecture works like a placeholder for a scalable "block reconfigurable" array architecture [11], with multiple discrete blocks that can be used independently, rather than one large configurable fabric. Each computation process or function can be loaded as needed on any physical RecoBlock at run-time from a library of configurations, as depicted in Fig. 1.

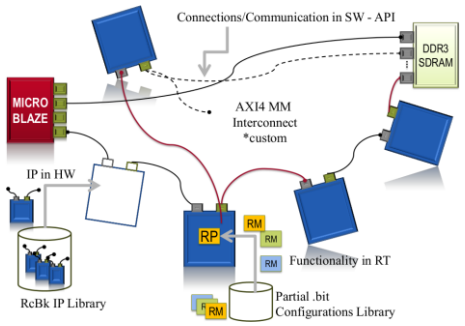

Fig. 1. Simplified RecoBlock concept. RecoBlock-IPs instantiated from IP Library. RP configured from configurations library at run-time. Interconnection fabric (e.g., AXI4 or NoC) support plug-play links configured by software. API functions handles reconfiguration and transactions.

This kind of blocks, or Swappable Logic Units (SLU) [12] , behaves like pages in traditional virtual memory systems. Thus, a set of reconfigurable processes in a processing network represents an abstraction layer of functions that can be implemented over the physical layer of the RecoBlock architecture, taking advantage of intrinsic hardware acceleration. It is important to note here, that no computation or communication link is pre-assigned to any block; therefore the interconnection links are also configurable at run-time. In principle, this means that any pre-compiled function can be downloaded into any existing RecoBlock at all times.

B. *Pre-requisites for Flexibility and Reusability in RTR*

To improve flexibility and reusability with the RecoBlock concept, the following minimum pre-requisites should be met:

1. IP-Core with embedded Reconfigurable Partition (RP).
2. Fixed RP input and output interfaces.
3. Fixed static logic in IP-Core.
4. RTR decoupling logic in static regions.
5. Fixed RP physical layout area.
6. Intercommunication links configurable by software.
7. Pre-generated library of configurations.
8. IP-Core with no direct interface to external pins.

Suggested for data-stream and context switch support:

9. Internal buffer.

C. *The RecoBlock SoC Platform.*

The RecoBlock SoC platform is shown in Fig. 2. In principle it can be viewed as a processor system, where an array of reconfigurable blocks has been hung onto the local bus. In our case, the RecoBlocks are connected through the AXI4 Interconnect of a Xilinx Virtex-6 circuit, and they communicate between each other and with the processor using data-streams implemented as built-in buffers, or traditional memory-buffer schemes.

AXI4 is the Advanced eXtensible Interface (AXI) protocol for IP cores included in the latest Advanced Microcontroller Bus Architecture (AMBA) ARM 4.0 specification. The "AXI Interconnect IP" provides interfaces to connect and route transactions between master (MI) and slave (SI) interfaces of memory-mapped IP cores [13]. To enable array interconnectivity in the RecoBlock platform, it is configured in the Sparse Crossbar Mode, which features a Shared-Address-

Multiple-Data (SAMD) topology, where parallel data pathways connect each MI to all SI they can access; then data transfers can occur independently and concurrently under single-threaded write and read address arbitration.

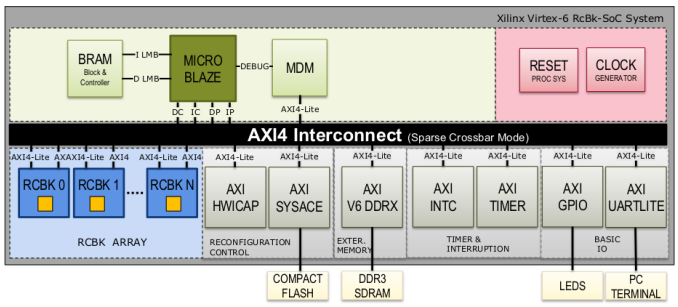

Fig. 2. Abstract diagram of RecoBlock SoC platform. Array of RTR RecoBlock-IPs are on the lower-left corner, with AXI4 and AXI4-Lite interfaces. AXI4 Interconnect details and other connections are omitted.

1) RecoBlock IP Description: A RecoBlock is an AXI4 Memory-Mapped custom component with a Reconfigurable Partition (RP), where several Reconfiguration Modules (RM) will be loaded at run-time [14]. As memory-mapped component, it can be treated as virtual variable/buffer in API functions. Fig. 3 shows a simplified hierarchically structure.

a) Reconfigurable Partition (RP): The RP is the only non-static region in a RecoBlock. Its purpose is to hold a Reconfigurable Module (RM), after its respective partial configuration file has been loaded at run-time. A RM is defined by a HDL description which is separately synthesized into a netlist file (.ngc). One or more RMs represented as partial configuration files constitute the RM library. To facilitate reusability and rapid-integration, the RP has a simple interface consisting of: a) one 32-bit data input, b) one 32-bit data output, and c) clock and reset signals. Slow handshake methods were unnecessary since the logic in the RP must start only in response to software reset [15], and because of the RecoBlock is considered data-driven.

To implement partial reconfiguration the design must be partitioned in static and reconfigurable regions, and a separate netlist synthesized for each partition. When adding a RecoBlock IP in XPS, there is no restriction for the RP size since it is considered a black box without functionality. However, the RP physical size and timing constraints are restricted to the FPGA's space availability and PlanAhead tool's efficiency to generate each configuration run [16].

b) AXI4 IP-Interfaces (IPIF): A RecoBlock instance can interact with the system through 2 AXI4 memory mapped IPIF connected directly to the AXI4 Interconnect IP. First, the AXI4-Lite IPIF is capable of single transactions of 32bit per clock beat. Its main purpose is to perform read/write operations on Slave-Registers. Second, the AXI4 Burst Master is capable of single and burst transactions up to 256 words of 32bits per cycle in a single address phase [15]. It can start a write or read transaction.

c) Slave and Master Registers: The RecoBlock has six 32 bit registers. The Ctrl and Status registers have bits to control the Execution-Decoupling state machine, which basically decouples/isolates the RP during RTR. DataIn is used as data input to be processed by the logic in RP. ExeT will receive the expected execution time (clock beats) needed by the RP to complete a computation after reconfiguration. Result register stores the result (and input context) of the RP function after execution time defined in ExeT. On the other hand, the Master Registers are used along with the Burst-Command-Control logic for AXI4 protocol handling, data interpretation, and single/ burst transactions.

d) Embedded FIFO: The embedded FIFO work as:

- storage for results of RP computations,
- quick access inter-process communication buffer,
- decoupling buffer in streaming SDF models,
- context switching storage during RTR.

It is optimized by using native SRL blocks, and is parameterized to 32-bit wide and 128-bit depth; which is big enough for the current platform purposes.

e) Execution-Decoupling: Because the static regions of the FPGA remain operative when the logic in RP is being modified [14], the Exec-and-Decoupling module guarantee that outputs of RPs are ignored during partial reconfiguration. After software reset, the Exec-Decoupling waits the number of clock beats specified in ExeT register (expected execution time of the RP function), and then asserts its outputs to enable the data channel between RP and FIFO, which stores the new result. Consequently, by having the decoupling and execution time logic separated from the RP guarantees portability and easily conversion of existing HDL designs into RMs.

f) Software Reset: After partial reconfiguration, the initial state of the PR logic is unpredictable [14]. A global reset would reset the static regions as well. For that reason, the Soft-Reset [15] module emits generates a parameterizable local reset pulse, after the "rB_SftRst_Go" function is issued.

g) Reconfiguration Control: Reconfiguration is supported by API and handled in HW by 2 AXI4 components, i.e., HWICAP and SYSACE; and an external Compact Flash (CF) memory. The external memory stores the initial and partial .bit files generated at design time. A software function reads the desired file, unpacks it, and the HWICAP loads the configuration in the PR of the target RecoBlock.

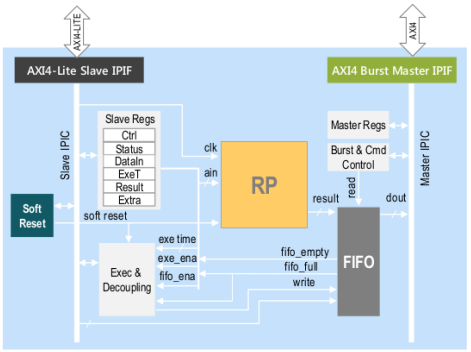

Fig. 3. Simplified RecoBlock-IP diagram. RP is the only reconfigurable region, the rest are static during synthesis.

IV. SOFTWARE VIEW

The RecoBlock platform is oriented to SDF models for streaming applications and is data-driven. A system is modeled as a set of communicating processes. A buffer is needed to decouple the different data rates of the input and output streams, since concurrent process exchange data through unidirectional FIFO channels that carry a 'stream' or sequence of data 'tokens'. Writes to the FIFO channel are non-blocking and reads are blocking [17] [18]. A process fires when: a) firing rules are met, b) enough tokens exist in input-side FIFO, and c) enough space exists in output-side FIFO.

The RecoBlock platform can use two communication mechanisms between processes: a) the classical "buffers in memory", and b) local buffer (FIFO) in RecoBlock. The latter, provides less latency and memory access contention, with additional burst capability, which improves performance.

A. SW Abstraction Layer:

Fig. 4 shows the hierarchical abstract view of HW and SW layers available for the user application designer. Each layer has an exclusive RTR section that is part of the RecoBlock API (i.e., drivers, configurations, API), which handles the underneath RecoBlock array. The Board Support Package (BSP) section is generated in SDK along with basic templates. The RecoBlock Lib is generated externally in PlanAhead.

The RecoBlock API consists of high level functions supported by drivers, macros and definitions of the lower layers. It is divided in: transactions and reconfiguration.

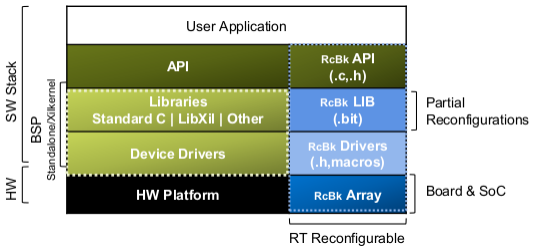

Fig. 4. SW Abstraction Layer. For each layer, RecoBlock Platform components are on the left and exclusive RTR components are on the right. Partial configurations are part of the Library layer.

B. Inter Block/Memory Transactions API:

Depending of the number of instantiated RecoBlock and AXI interconnections in XPS, the definition of the array is completed here, by using the transactions API in Table 1. It summarizes the functions dedicated to control the different type of transactions enabled by hardware from/to a RecoBlock, which can be: a) memory to RecoBlock, b) RecoBlock to RecoBlock, or c) RecoBlock to memory

TABLE I. RECOBLOCK TRANSACTIONS API

RecoBlock Transactions			
Symbolic Transaction	***Type***	***Direction***	***API Function Name***
[Mem] 1 --> 1 [RB_x]	Single	To	rB_MbDataIn( )
[RB_x] 1 --> 1 [RB_{x+1}]	Single	Inter	rB_RcBk2RcBk( )
[RB_x] 1 --> 1 [Mem]	Single	From	rB_SendNoBrst( )
[RB_x] n --> n [Mem]	Burst	From	rB_RdFifo( )

In some cases, a burst transfer of the internal buffer to a memory buffer is allowed, otherwise transfers are single. Functions names are simplified without fields. The format for symbolic transactions is:

[source] #output-tokens → #input-tokens [destination]

Where: Mem is memory variable or buffer, and RB is RecoBlock- IP

C. Reconfiguration API

In the RecoBlock platform, there is no fixed schedule tied to the architecture, and then reconfiguration management is highly flexible allowing exploration of different schemes defined mostly by application software (as shown in the Case Study). Run-time reconfigurations and computed schedules are programmed using functions shown in Table II.

TABLE II. RECOBLOCK RECONFIGURATION API

RecoBlock Reconfiguration		
Symbolic Operation	***Action***	***API Function Name***
[Status] --> [MB]	Read Status register	rB_Status( )
[MB] --> [Enable]	Enable bits in Ctrl register	rB_EnaAll( )
[#cycles] --> [ExeT]	Loads expected execution time in ExeT register.	rB_ExeT( )
[MB] --> [SoftReset]	Reset PR and start execution	rB_SftRst_Go( )
[CF] --> [MB] [MB] --> [ICAP] [ICAP] --> [RP_{RBx}]	Read .bit from library Parse .bit, send to ICAP Reconfigure RP	rB_CF2Icap( )

To illustrate, a basic pseudo-code sequence for a fresh reconfiguration is shown below:

1) Rb_EnaAll() //enable Exec&Decoup. Logic
2) rB_ExeT() //load expected execution time
3) rB_Cf2Icap() //read .bit, parse, reconfigure RP
4) rB_SftRst_Go() //reset RP, start computation, decouple RP

Here, RP is coupled automatically at the end of computation to store the result and is immediately decoupled, without any special instruction. Execution time is pre computed and changed only if different for the new configuration; otherwise only steps 3) and 4) are mandatory. Finally, step 1) is required only after global reset.

V. RECOBLOCK DESIGN FLOW

In Fig. 5, the diagram describes important steps, guidelines, and concepts necessary to use/reuse the RecoBlock platform and methodology. This contribution is product of our experiences and derived from the normal Xilinx flows. Tool domains are represented vertically.

The section on the upper-left corner is not part of the platform, but is recommended. Therefore, after idea conception, the designer should have a model, computed schedule, and reconfigurable architecture to be implemented. Then, if more RecoBlock-IPs is needed, they are instantiated from the IP-Library, otherwise the basic platform is enough to generate the .xml description and export to SDK. Here, a new BSP package is generated if new IPs were added, otherwise the same is reused. Application is developed using the provided RecoBlock API. The FPGA board is programmed with an initial configuration (static + partial) and a boot loader that enables debugging features from SDK.

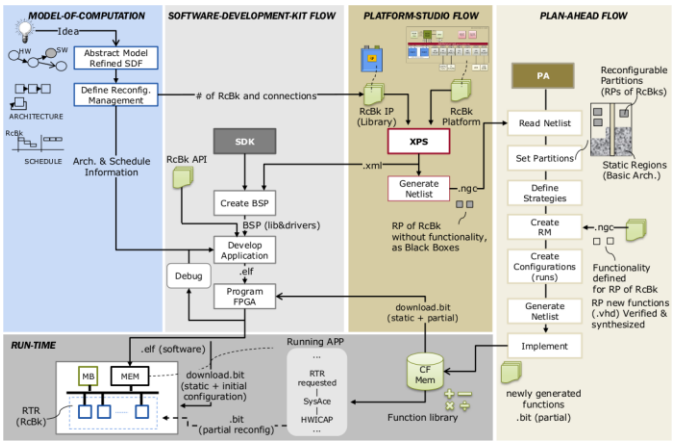

Fig. 5. RecoBlock design flow. Tools domains are vertical. Central and Run-Time sections describe the straightforward flows to reuse the platform. Right section belongs to the offline recurrent flow when configurations are not in library. Upper-left section is a recommended design entry.

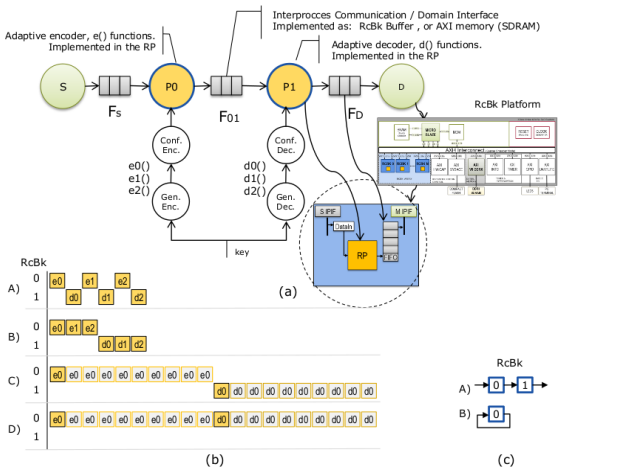

Fig. 6. Experiment setup: (a) typical example model of function adaptivity: encoder/decoder, (b) reconfigurable schedules, (c) reconfigurable architectures: A) spatial, B) phased or sequential (context switch).

During run-time, new partial configurations are loaded from the provided function library. When new configurations are not in library, a netlist with RPs as black boxes is generated from XPS, then functionality is added to RPs from HDL descriptions, and finally a long recursive offline process starts until a new configuration is generated, so it must be avoided.

VI. Case Study

We use a tutorial case study to show the potential of our platform. The case study implements a typical example of an adaptive system model, corresponding to an encoder/decoder streaming application. We run four different experiments (exp.) for the same model, but using different reconfigurable architectures and schedules, without regenerating the platform. Then we evaluate the platform properties based on the results.

By using an additional input signal carrying functions as values, a process can be expanded to cover several types of adaptivity [19]. In that context, Fig. 6 (a) shows a "function" adaptivity model. Here, the main processing network, consisting of processes (S, P0, P1, and D) and decoupling buffers (Fs, F01, and Fd), models a RTR streaming application based on adaptive extensions on SDF semantics [20].

A. Reconfiguration Management: Architecture and Schedule

The SoC platform was generated with 2 RecoBlock IP instances, namely RecoBlock0 and RecoBlock1, representing P0 and P1. Their PR are reconfigured at run-time with encoder "e" and decoder "d" functions, whose bitstreams were generated at design time and are part of the "function library" available during run-time in CF memory.

The e/d functions correspond to Shift Cipher cryptosystems based on modular arithmetic or inverse operations. Thus, in average, all functions utilize 32 Slice Registers, 32 LUTs, and 67 KB (.bit). In that way, we keep algorithm complexity constant and focus our analysis in flexibility, reusability, and configuration management. The physical inter-connections are set during design time in XPS, but the actual destination of each transaction is selected by API functions.

The four schedule variations are described in Fig. 6 (b), colored boxes are "reconfiguration + execution" events, and grayed boxes are just "execution" events of the current configuration. Thus, two scheduling groups are employed:

- Schedules A) and B) use n different e/d function pairs ($e_i d_i$, for $0<i<n-1$) to process a total of n tokens. In B) the whole stream is encoded, stored in internal buffer, and finally decoded. Single and multiple production rates are simulated in A) and B), respectively.
- Schedules B) and C) use only 1 set of e/d functions, to process a large number of tokens compared with the reconfiguration cycles ($t_{running} >> t_{reconfig.}$). In addition, D) is a special case that uses only RecoBlock0 for both functions, according to architecture B) .

The two architectures in Fig. 6 (c) will evaluate at least two concepts, context switching and prefetching. Therefore:

- Architecture A) uses a typical approach (spatial), assigning e0 and d0 to separate RecoBlocks.
- In contrast, B) time-multiplexes e0 and d0 in RecoBlock0. It saves area, but require a "context switch" [21], for which it takes advantage of RecoBlock0's buffer burst to memory.

B. Experiments Setup

According to Fig. 6, Table III summarizes the experiment setup for four combinations of architectures, schedules, number of functions and processed tokens. Time is measured using timer interrupt routines, disregarding unrelated functions. However, it is highly affected by extra software overhead, especially by CF latency, and file processing.

C. Results Analysis

The flexibility and reusability concepts were demonstrated since: a) RecoBlocks were easily instantiated from IP-library, b) different functions were reconfigured at run-time, implementing efficiently the adaptivity model (exp.1-2), and c) communication links were selected by transaction API (Fig. 6 (c) and exp. 4). Besides, produced and consumed tokens match, which validates the HW/SW platform and design flow.

TABLE III. EXPERIMENTS RESULTS - ACCORDING TO FIG. 6.

E [a]	SA [a]	RR [a]		T [a]	TR [a] (ms)		EX [a] (ms)	P [a] (ms)
		0	*1*		*0*	*1*		
1	AA	3	3	3	1067.6	1067.7	0.137	2135.4
2	BA	3	3	3	1067.0	1067.0	0.137	2134.2
3	CA	1	1	10	355.6	355.6	0.463	711.6
4	DB	2	--	10	711.6	---	0.465	712.0

[a.] Experiment (E), Schedule-Architecture (SA), # of reconfigurations per block (RR) , Tokens processed (T) , Sum of reconfiguration time per block (TR), Execution time (Ex), Total processing time (P)

Exp. 1-2 show than the schedule selection (single or multi-rate) does not improve the overall result, given the same architecture. The internal buffer is useful in exp. 2.

Exp. 3-4 clearly illustrate the reconfiguration trade-off between area and time. The overall processing time is similar, but experiment 3 uses only 1 RecoBlock, despite of the context-switch required in experiment 4; which consumed only 42.48 us thanks to the internal buffer burst capability.

In addition, configuration overhead in exp. 3-4 is smaller than 1-2. RTR is attractive when configuration time is small compared to the amount of processed data.

Finally, resource utilization in XPS shows: 9.7% FFs, and 9.3% LUT of total system (no FPGA) for each RecoBlock IP (RP as black boxes). In PlanAhead, the average utilization of e/d functions in RPs is 7%. For each RP pblock, 800 FD, 400 LUT, and 4 DSP48E1 were available. Offline time required to implement each configuration (runs) ranges from 20 to 40 minutes. Supported FMax is 77.918 MHz.

VII. CONCLUSIONS AND FUTURE WORK

The overall impact of the work presented in the paper is summarized as follows. We devised a set of recommendations to promote flexibility and reusability in RTR FPGA-based designs. We applied them to design and implement, compared to [4][5][6][7], the RecoBlock platform and systematize a reliable design flow; which utilizes RTR RecoBlock-IPs. The experiments conducted using the platform demonstrated that those are a valid minimum set of conditions to achieve that goal. The tutorial case study also showed the flexibility and reusability properties of the platform, by replicating a number of reconfigurable arrays and schedules, which were configured at run-time with API functions, without any hardware regeneration. In addition, transactions and configurations are independent and supported by APIs, this feature facilitates design exploration according to the orthogonalization principle. Besides, the platform balances efficiently time, cost, and performance. In contrast to [8] and [10] the system uses the newest AXI4.

Furthermore, we devise some challenges in our platform. Although a library of functions minimizes design time, PlanAhead processes are critical in RTR designs and should be automated if possible or avoided by exploring optimum size and topology arrays. Besides, initializing reconfigurations in DDR3 will improve performance.

Our next step is to exploit and explore the flexibility and reusability of the platform with computation demanding applications and through integration with complex models, compilers, dynamic reconfiguration managers. Our vision is to take the RecoBlock concept to Multi-Core Network-on-Chip environments, where RTR nodes can improve hardware acceleration and fault-tolerance.

REFERENCES

[1] G. Martin and H. Chang, Eds., *Winning the SoC revolution : experiences in real design*. Boston, Mass.: Boston, Mass. : Kluwer Academic Publishers, 2003.

[2] D. D. Gajski, A. C.-H. Wu, V. Chaiyakul, S. Mori, T. Nukiyama, and P. Bricaud, "Essential issues for IP reuse," *Proceedings of the ASP-DAC 2000. Asia and South Pacific*, pp. 37–42, 2000.

[3] K. Keutzer and A. Newton, "System-level design: Orthogonalization of concerns and platform-based design," *... -Aided Design of ...*, vol. 19, no. 12, pp. 1523–1543, 2000.

[4] M. Kuehnle, A. Brito, C. Roth, K. Dagas, and J. Becker, "The Study of a Dynamic Reconfiguration Manager for Systems-on-Chip," *2011 IEEE Computer Society Annual Symposium on VLSI*, pp. 13–18, Jul. 2011.

[5] V. Nollet, P. Coene, D. Verkest, S. Vernalde, and R. Lauwereins, "Designing an Operating System for a Heterogeneous Reconfigurable SoC also Professor at Katholieke Universiteit Leuven," vol. 00, no. C, 2003.

[6] J. Noguera and R. M. Badia, "Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling," *ACM Transactions on Embedded Computing Systems*, vol. 3, no. 2, pp. 385–406, May 2004.

[7] A. Jara-Berrocal and A. Gordon-Ross, "Hardware module reuse and runtime assembly for dynamic management of reconfigurable resources," *2011 International Conference on Field-Programmable Technology*, pp. 1–6, Dec. 2011.

[8] M. Liu, Z. Lu, W. Kuehn, S. Yang, and A. Jantsch, "A Reconfigurable Design Framework for FPGA Adaptive Computing," *2009 International Conference on Reconfigurable Computing and FPGAs*, pp. 439–444, Dec. 2009.

[9] C. Bobda, A. Majer, A. Ahmadinia, T. Haller, A. Linarth, and J. Teich, "The Erlangen slot machine: increasing flexibility in FPGA-based reconfigurable platforms," *Field-Programmable Technology, 2005. Proceedings. 2005 IEEE International Conference on*, pp. 37–42, 2005.

[10] S. Shukla, N. W. Bergmann, and J. Becker, "QUKU: A FPGA Based Flexible Coarse Grain Architecture Design Paradigm using Process Networks," *2007 IEEE International Parallel and Distributed Processing Symposium*, pp. 1–7, 2007.

[11] K. Compton, "Reconfiguration Management," in *Reconfigurable Computing: The Theory and Practice of FPGA-based Computation*, S. Hauck and A. DeHon, Eds. Morgan Kaufmann/Elsevier, 2008.

[12] G. Brebner, "The swappable logic unit: a paradigm for virtual hardware," *Proceedings of the 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines*, pp. 77–86, 1997.

[13] Xilinx, "AXI Reference Guide, UG761 (v13.4)," 2012.

[14] Xilinx, "Partial Reconfiguration User Guide, UG702 (v12.3)," 2010.

[15] Xilinx, "Xilinx: Product Support & Documentation." 2012.

[16] Xilinx, "PlanAhead User Guide, UG632 (v13.4)," 2012.

[17] E. a. Lee and T. M. Parks, "Dataflow process networks," *Proceedings of the IEEE*, vol. 83, no. 5, pp. 773–801, May 1995.

[18] J. P. Barros, A. Costa, and L. Gomes, "Modeling Formalisms for Embedded System Design," in *Embedded Systems Handbook*, CRC Press, 2005, pp. 5–34.

[19] I. Sander and A. Jantsch, "Modelling Adaptive Systems in ForSyDe," *Electronic Notes in Theoretical Computer Science*, vol. 200, no. 2, pp. 39–54, Feb. 2008.

[20] J. Zhu, "Performance Analysis and Implementation of Predictable Streaming Applications on Multiprocessor Systems-on-Chip," KTH, Electronic Systems, 2010.

[21] S. Hauck and A. DeHon, *Reconfigurable Computing: The Theory and Practice of FPGA-Based Computation SE - Systems on Silicon*. Morgan Kaufmann, 2007.