FoRTReSS: a flow for design space exploration of partially reconfigurable systems

François Duhem, Fabrice Muller, Robin Bonamy, Sebastien Bilavarn

▶ To cite this version:

François Duhem, Fabrice Muller, Robin Bonamy, Sebastien Bilavarn. FoRTReSS: a flow for design space exploration of partially reconfigurable systems. Design Automation for Embedded Systems, 2015, 19 (3), pp.301-326. 10.1007/s10617-015-9160-2 . hal-01134008v1

HAL Id: hal-01134008
https://hal.science/hal-01134008v1

Submitted on 16 Mar 2016 (v1), last revised 11 Jan 2017 (v2)

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

FoRTReSS: a Flow for Design Space Exploration of Partially Reconfigurable Systems

François Duhem[1], Fabrice Muller[1], Robin Bonamy[1], Sebastien Bilavarn[1] *

Abstract

In this paper, we present a flow enabling design space exploration for partially reconfigurable systems with real-time constraints, called FoRTReSS. FoRTReSS allows estimating mixed hardware/software implementations of an application where the hardware design space, the floorplanning of reconfigurable regions placed on the FPGA, is automatically inferred from application resources information, interface constraints and the target device. Real-time constraints are verified by a highly configurable SystemC simulator, RecoSim, handling applications described as Control Data Flow Graphs (CDFGs). We demonstrate our approach on an H.264 video decoder and an H.265 encoder targeting the latest Zynq-7000 platforms from Xilinx, embedding a Cortex-A9 dual-core processor. We show that an hardware/software implementation of the H.264 decoder using both processor cores and slice decomposition is possible under real-time constraints, effectively achieving a framerate of 30 frames per second while reducing area requirements compared to a static implementation, using 54% less slice resources and 44% less BRAM resources. Additionally we report the ability of the methodology to address very early analysis from high level application specification on the example of an H.265 encoder.

Keywords: Reconfigurable Architecture, Design Space Exploration, Real-Time Systems, Partial Reconfiguration, Field Programmable Gate Arrays

1 Introduction

The last few decades saw the emergence of reconfigurable computing through Field Programmable Gate Arrays (FPGA). These devices provide a high level of parallelism along with programmability, bridging the gap between programmable processors and high performance, but expensive, Application Specific Integrated Circuits (ASIC). Over generations of devices, FPGAs had more and more computing power so that entire systems can now be built into a single device, including processors, hardware accelerators, memory controllers, I/O peripherals and so on. They are called System on Programmable Chip (SoPC). Processors included in the design can be either soft cores (i.e. using the FPGA fabric as resources, for instance MicroBlaze or Nios cores) or hard cores (i.e. integrated within the die, hardwired to the FPGA). In the latter case, the processors are much more powerful than soft cores, hence providing designers with high performance processor-centric architectures like the Xilinx Zynq-7000 devices, based

*[1]LEAT - University of Nice Sophia Antipolis, CNRS

on a dual ARM Cortex-A9 MPCore [36]. However, the hardware part of the Zynq-7000 does not offer many logic cells compared to state-of-the-art Virtex-7 FPGAs built with the same 28 nm technology.

At first, these devices could only be programmed in an all-or-nothing style such that all services and IP cores are stopped during reconfiguration. In the case of communication services, pieces of data might be lost and degrade the Quality-of-Service (QoS). In critical applications, this cannot be tolerated. This led to the development of Partial Reconfiguration (PR), introduced with the Xilinx XC6200 series, which allows modifying the behaviour of pre-defined Reconfigurable Regions (RR) without affecting the remaining logic. The circuit functionality can thus be modified at runtime depending on the application requirements and/or execution. Since not all the resources are present on the FPGA persistently, area requirements are reduced. If a careful study is made at design time, it is possible either to switch to a smaller and cheaper FPGA or to add extra features on the target device [22, 28].

Despite promising features, PR is still not widely spread in the industry [26]. The major issue concerns designing the systems satisfying application requirements as well as technology-specific constraints inherent to PR systems. For instance, mutualising reconfigurable region resources between multiple tasks is possible, but implies PR-compatible scheduling that takes into account reconfiguration times, which cannot be neglected for applications with severe real-time constraints [8, 31]. Moreover, Design Space Exploration (DSE) still remains a great challenge: existing design flows do not allow DSE during early development stages but rather during the late FPGA implementation stage when the cost implied by any architecture modification can be prohibitive. Hence, FPGA engineers prefer relying on existing, well-known and reliable design flows, even if the solution is sub-optimal in terms of logic resources or device cost.

Our contribution in this paper is an extension of previous work described in [12]. The methodology consists of FoRTReSS, a Flow for Reconfigurable archiTectures in Real time SystemS that enables both hardware and software design space exploration for partially reconfigurable applications with real-time constraints, along with its graphical user interface, FoRTReSS Toolbox. Our approach is based on two main steps: first, an architecture is generated in terms of processors and reconfigurable regions. RRs are inferred from synthesis results in the form of netlists and text reports and/or XML (eXtensible Markup Language) task descriptions. The second step consists of the simulation of this architecture using RecoSim, a Reconfigurable simulator written in SystemC. This step automatically verifies whether the application real-time constraints are met with a given QoS.

Since our previous work, a lot of significant improvements have been made on RecoSim and new features have been introduced. First of all, the model of computation changed a lot as RecoSim now handles Control Data Flow Graphs (CDFGs) instead of DFGs. Diagrams may contain cycles and loopbacks, while tasks can be periodic or not. We also added type information to the interfaces (e.g. AXI, PLB, FIFO...) and refined our communication model in order to provide the user with a more accurate description. Another important extension consists in adding software considerations into FoRTReSS. It is now possible to add processors to the design and defining several implementations for one task. This feature is illustrated by new use cases: an H.264 video decoder and H.265 video encoder with multiple hardware and software implementations of

tasks. FoRTReSS provides several Application Programming Interfaces (APIs) to easily modify and/or develop task and scheduling algorithms. The paper also targets a state-of-the-art Zynq device. In fact, any device, already existing or not, can be targeted using a flexible device description using XML.

The remainder of the paper is structured as follows: in Section 2, we discuss works related to FPGA floorplanning and existing PR design flows. Section 3 introduces our methodology. In section 4, our approach is validated using two representative applications with performance and architecture results. Finally, future works and conclusions are outlined respectively in Section 5 and Section 6.

2 Related Work

2.1 FPGA floorplanning

One primordial problem is the task placement on FPGA led by a placement algorithm. Two categories of algorithms are clearly identified: off-line and on-line. In the first category, it is possible to investigate a near-optimal or optimal solution because the off-line scenario is found before the execution of the system. In the second category, the placement decision must be taken quickly because time is very critical. Anyway, RR placement for partially reconfigurable systems is necessarily defined during the design phase. Hence, we only discuss below on off-line floorplanning techniques.

The placement of hardware tasks on an FPGA is an NP-Complete problem and the time to reach a solution depends mainly on the number of the tasks. The work in [5, 6] introduces an exact resolution of scheduling problems. These approaches are very time-consuming and are not scalable. Rather than finding the best solution, it is preferable to find a near-optimal solution in a reasonable amount of time, based on heuristics. For instance, authors in [3] define 3D FPGA templates in time and space dimensions and use simulated annealing and greedy research heuristics to place Reconfigurable Functional Unit Operations (or RFUOPs). Moreover, Lodi et al. propose in [25] and [24] different off-line approaches to resolve hardware task placement as 2D bin-packing problem for instance Floor-Ceiling algorithm and Knapsack packing algorithm. However, these approaches are not adapted for real-time embedded systems because the scheduling of tasks has to be known at compile time.

Authors in [29] introduce an approach based on simulated annealing in order to find out the bigger common area between two reconfigurable zones. This common area will not be modified and the reconfiguration overhead will decrease. Authors also consider traffic congestion in the design brought by placing reconfigurable regions close to each other, which helps to remove infeasible solutions in many designs.

In [27], authors present a resource- and configuration-aware floorplacement framework that uses metrics such as external wire length (total length of wires connecting reconfigurable regions) to qualify a solution. They report an average improvement of 50% when using this metric. Their main objective is to group reconfigurable units together without taking into account the heterogeneity of FPGA resources. Indeed, current FPGAs include different partially reconfigurable resources such as logic blocks (e.g. Configurable Logic Blocks), memories (e.g. Block Random Access Memory) and Digital Signal Processing

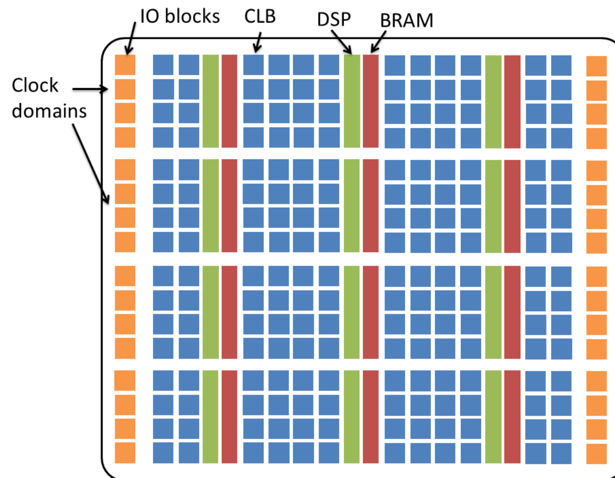

Figure 1: Physical disposition of resources inside an FPGA

blocks. A column includes only one kind of resource. Hence, a flexible model is essential because the structure of FPGAs differs.

A methodology for an architecture-aware and reconfiguration-centric floorplanning is introduced by [34]. To obtain the solution, a cost function is defined by the total wire length and wastage of resources without taking into account the task dependencies or timing constraints for the application. However, the real-time aspect is essential and must be considered for partially reconfigurable systems. A lot of application domains such as robotics, video streaming, automotive, avionics and so on, depend on these constraints.

An approach similar to FoRTReSS has been developed by authors in [23]. This methodology, called FoRSE, (standing for Formulation-level partial Reconfiguration design Space Exploration) performs a mathematical-based exploration of the design space, looking for a Pareto optimum for the application, considerably reducing the exploration time compared to methodologies requiring implementation (implementation-level versus formulation-level). Nevertheless, this formal method does not take into account potential real-time constraints. However, it is worth noting that this methodology relies on Xilinx FPGA models to represent an FPGA device.

According to our knowledge, the works on floorplanning for partial reconfiguration are often treated separately or even are not studied at all. We think that these two problems must be considered together, as the floorplan problem is often meaningless without the communication channels to support it.

2.2 PR design flows

To complete this overview of methods for partial reconfiguration, it is also important to give a clear picture of partial reconfiguration design flows. Xilinx was the first company to introduce such a feature for their FPGAs [37]. Their flow inspired Virginia Tech's open-source tool, OpenPR [30]. Altera also recently unveiled a new version of Quartus enabling dynamic and partial reconfiguration for their state-of-the-art Stratix V FPGAs [1] with a PR flow pretty much

similar to Xilinx's one. Another interesting flow is GoAhead [4], an academic tool providing some new partial reconfiguration features such as module relocation (an extensively adressed subject [16, 9]). GoAhead also allows mapping two reconfigurable modules simultaneously inside the same PR region (which is also possible using only Xilinx tools but with an important extra design effort). We believe that a design methodology should not be (or the least possible) technology-dependent and also extendable to virtual FPGAs and architecture exploration of physical FPGAs.

FoRTReSS addresses these issues by providing designers with a feasible floorplan for state-of-the-art heterogeneous devices and a task scheduling that satisfies the application timing constraints. The FoRTReSS device model is flexible in order to be compliant with future device architectures or virtual FPGA platforms. Finally, FoRTReSS can handle task and RR interfaces to represent various communication models (e.g. point to point, shared bus...).

3 Our approach

3.1 FoRTReSS overview

FoRTReSS is a tool providing the user a way to explore the partial reconfiguration design space and ultimately proposing a set of reconfigurable regions and processors that will ensure a certain Quality-of-Service (QoS) for a given application. The term quality of service refers to the rate of task executions that respected their deadline. For instance, hard real time applications would typically require a QoS of 100% whereas in applications such as video streaming, it is acceptable to lose some packets and have a degraded QoS.

Figure 2 shows an overview of the FoRTReSS flow. It is based upon a Y-chart approach where application and architecture are described separately. The application is described as a Control Data Flow Graph (CDFG) or a set of periodic tasks with dependencies. Each task has some timing characteristics such as a deadline or a period (zero for non-periodic tasks), and a set of possible implementations with different performance, resource and energy trade-offs. These implementations can be hardware (i.e. to be mapped on a reconfigurable region) or software (i.e. to be mapped on a processor core) and share a set of parameters such as the task Best/Worst Case Execution Time (BCET/WCET). Some other parameters are different from one type of implementation to another. Typically, hardware implementations are defined by their resource requirements resulting from synthesis. This information can be extracted from Xilinx synthesis reports or from XML-based files (.tsk extension) developed for compliance with other synthesis tools. Describing task hardware implementation is mandatory in order to determine a reconfigurable region set which might fit the application. FoRTReSS might also require full netlists of each implementation (Xilinx NGC or standard EDF) in cases where compressed bitstreams are used to optimise reconfiguration times (see section 3.3.6 for more details). On the other hand, software implementations are characterised similarly by the time required to load the binary executable into the instruction memory of the processor.

The target architecture is described separately as an FPGA and a set of processor cores that can be either on the same die (Virtex-5 with integrated PowerPC or the latest Xilinx Zynq-7000 SoC with a CortexA9 processor), external

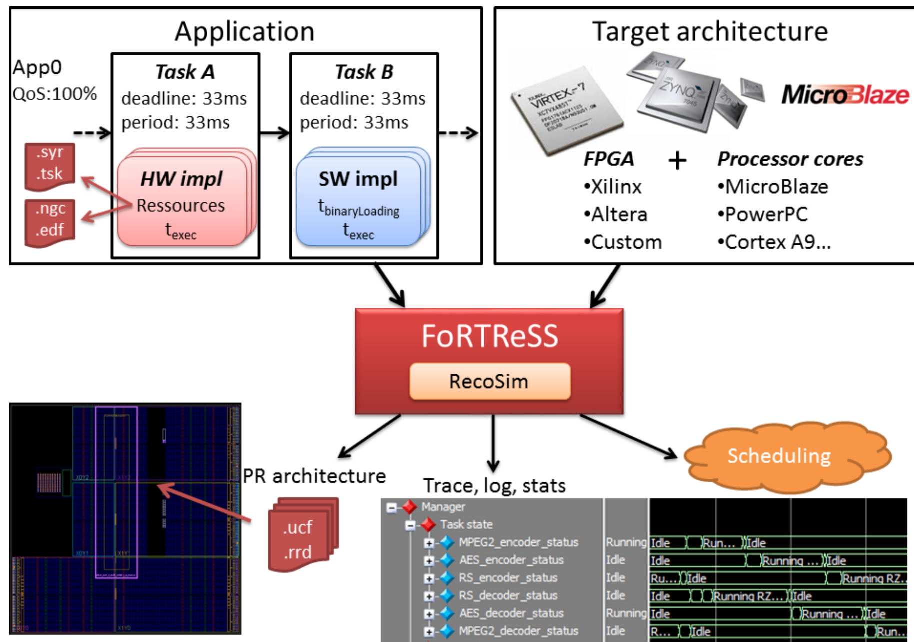

Figure 2: PR flow using FoRTReSS

to the FPGA or soft cores instantiated with its configurable logic resources (e.g. MicroBlaze). The FPGA architecture is also described using an XML-based file format in order to be compatible with existing devices as well as custom, virtual or non-existent (future) architectures.

FoRTReSS uses the application resource requirements and FPGA description to find potential reconfigurable regions. The validation of the application quality-of-service is carried out using a SystemC based simulator called RecoSim. Tasks are scheduled under a standard Earliest-Deadline-First (EDF) policy integrated in this simulator and that can be modified or extended to other scheduling policies due to a dedicated API (Application Programming Interface). RecoSim also generates traces, statistics and log files for every simulation for debug purposes. Finally, FoRTReSS provides an architecture fully defined in terms of RR by an UCF file (User Constraints File, used by Xilinx) and an XML representation of the regions (.rrd extension). The resulting floorplan can be viewed using the Xilinx PlanAhead design tool as shown in Fig. 2.

The SystemC simulator has already been introduced in a previous publication [13]. However, the model of computation has been improved since in order to better process the simulation of real time constraints. The following sections give an update on these features and a description of the underlying FoRTReSS flow in order to better understand the set of parameters impacting the design space exploration process.

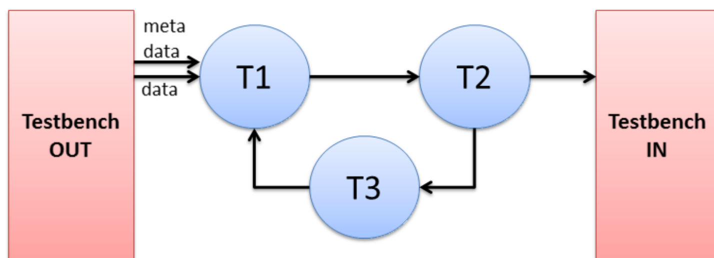

Figure 3: Example of CDFG handled by RecoSim

3.2 RecoSim overview

RecoSim, for Reconfigurable Simulator, is a SystemC/TLM simulator that verifies if an architecture can satisfy real-time constraints of an application. From a description of possible mappings of tasks on the execution units, RecoSim uses Transaction-Level Models of the system to ensure fast simulation while considering abstract communication details, reconfiguration overheads (that can be inferred from cost models such as [14]), context switches, hardware and software preemptions, for a wide range of architectures.

3.2.1 RecoSim model of computation

The main input of RecoSim is an application description given in the form of a Control Data Flow Graph, which means that communications from a module to another can be conditional. An example of specification that can be simulated by RecoSim is shown in Figure 3. Compared to standard Data Flow Graphs (DFGs), CDFGs introduce control at the task level to make conditional communications possible. For instance, task T2 in Figure 3 can send data to Testbench IN and task T3 independently: it is possible to make T2 send n packets to T3 after each execution (i.e. when new data are generated by the task algorithm) while sending data to the testbench once every m executions only. Another feature of this CDFG model is the ability to describe cyclic applications: on the first iteration, no relevant data are provided by T3 and hence T1 only waits for data incoming from the testbench. Finally, it is also possible to simulate several communication channels linking the same two tasks (see connections between testbench and task T1 in Figure 3). This can perfectly describe separate data and metadata channels to evaluate different performance between these connections. For example, a metadata channel would use a low speed bus (AXI Lite bus) while the data channel would need a more efficient bus (AXI bus).

The application must be surrounded by two testbenches as depicted in Figure 3, separating stimuli generation from result verification. Unlike other tasks, we suppose here that testbenches are persistent in the system (i.e. not dynamically reconfigurable). They also have no implementation explicitely defined, however testbench algorithms, which decribes the behavior of data sending or receiving, might be modified using a dedicated API.

Applications can be composed of periodic tasks. The main difference with non-periodic applications resides in the way tasks are started: for non-periodic applications, the task is launched whenever all incoming sockets have sent their data (according to the task algorithm controlling which socket is required for

this task execution). In case of periodic applications, it is also required that a new period has started. If not, the task has to wait for this new period and execution is delayed. However, its absolute deadline is still calculated with regard to the task relative deadline (except whenever incoming data are ready like in full dataflow applications).

Either way, the application has to be simulated long enough to ensure that the architecture maintains the required quality of service. The minimum simulation time for periodic applications is a hyperperiod. For DFGs, it is defined as the Least Common Multiple (LCM) of task periods whereas for CDFGs, the additional control part changes this hyperperiod and it is not possible to predict its value automatically. Moreover, in order for every tasks to be running in the same hyperperiod, the system should be in a steady state. We consider that the upper limit for the time required to enter this steady state can be estimated as the sum of every worst case execution times. Adding this time to the hyperperiod gives an estimation of the minimum simulation time. Note that it is the responsibility of the designer to ensure that this simulation time is respected and can be modified in FoRTReSS flow. The uncertainty brought by CDFGs into the computation of the hyperperiod leads FoRTReSS to notify the user when the simulation is relevant, but the designer might want to manually compute the minimum simulation time (or over estimate it) when designing the testbench.

Let us note $WCET(T_i)$ the Worst Case Execution Time of task i. Equation (1) gives us the minimal simulation time for the system.

$$t_{simulation,min} = LCM\left(T_1..T_n\right) + \sum_{i=1}^{n} WCET\left(T_i\right) \tag{1}$$

3.2.2 Mapping tasks to processing units

Recosim simulates the mapping of application tasks to the processing units and the corresponding execution (i.e. run-time allocation and scheduling). These processing units are either hardware (reconfigurable regions infered by FoRTReSS for the target FPGA) or software (user-defined processor cores). The mapping decision is made by the reconfiguration manager considering the different software and hardware implementations associated with each task as well as the availability of reconfiguration units: the default behaviour consists in getting the most out of the architecture by using As Many units As Possible (AMAP mapping). The choice of an implementation also depends on parameters such as the configuration time, execution time or energy consumption. A dedicated API is defined to help the definition of specific scheduling and allocation techniques.

3.2.3 Task preemption and context switches

We have already presented the management of preemption for hardware tasks in [13]. Our preemption model is based on a request/grant system where preemption points are explicitely defined in the task implementation code as depicted in Fig. 4. The reconfiguration manager does not preempt the task but it is rather the task that issues a preemption request (cooperative multitasking).

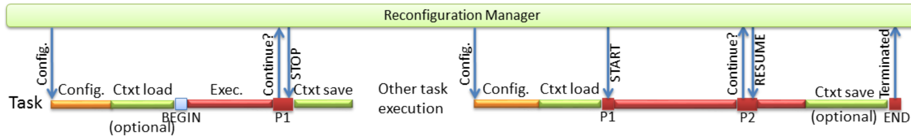

Figure 4: Task preemption within RecoSim

This behaviour fits well with the execution of hardware tasks which preemption is more complex than software tasks. Furthermore, context evolves during task execution as the number of registers used to store important data are changing. Hence, it is understandable that context switches, operated by register save/restore operations, should be performed when it has the fewest impacts.

This approach is also compatible with classic software preemption by defining as many preemption points as instructions in the executable. This way, the task will notify the manager after each instruction, making the task preemptible at every moment of its execution, emulating preemptive multitasking. However, the simulation time overhead inherent to this technique (due to an increase of communication between the module and the reconfiguration manager) can be fairly important but considered reasonable compared to accuracy. For example, we simulated an application composed of 30 tasks with a scenario representing an execution time of 2 seconds on a Core I7-3740QM running at 2.7GHz with 8GB of RAM. With a preemption resolution of 1 us, the execution time on the computer is 810 seconds against 1.53 seconds without preemption. This is a significant increase, but considering the complexity of preemption is at this cost. This overhead will be much closer to 1.53 seconds in real cases as real applications will only require a few preemption points per task. Nevertheless, this overhead is necessary for an accurate modeling of software tasks. In fact, the relationship between these two quantities is shown in Equation 2, where N_i is the number of preemption points for the task i.

$$Overhead\ (in\ seconds)\ = \sum_{i=0}^{TaskNumber} 0.4 \times N_i \tag{2}$$

Using (2), the designer has to perform a trade-off between extreme precision of software preemption, quality of the results (some good solutions might be missed when losing precision in some corner cases) and overall flow execution time.

Figure 4 also illustrates an example of context switch for a task execution. Context save and restore operations are required whenever a task is preempted. There is also an optional context switch at the beginning and at the end of task execution due to the distinction between hardware and software execution. Software context switches are processor dependent whereas for hardware implementations, this context switch depends on the implementation (two implementations of the same task might have different context switch times due to a different number of registers to save/restore, possibly some memories). Some work has been done on context switch for hardware tasks such as [5, 17, 20]. However, context switching is not required after a task configuration, but might

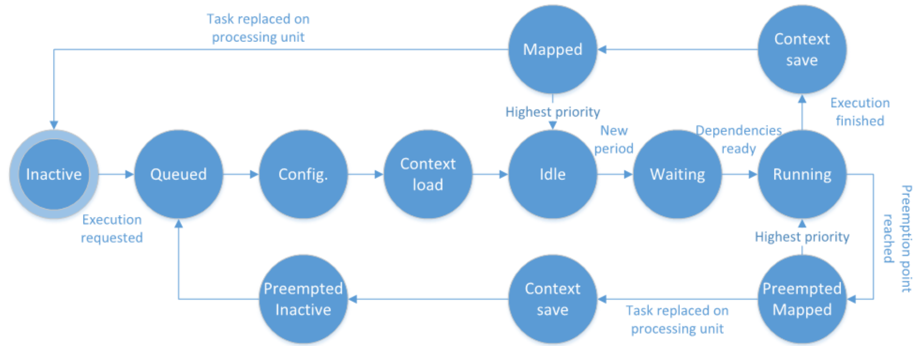

Figure 5: Task finite-state machine

be necessary for proper task execution (for instance, IP configuration). Hence, the designer is given the possibility to enable context switching after configuration for each implementation.

3.2.4 RecoSim task finite-state machine

RecoSim is based on the Finite-State Machine (FSM) depicted in Figure 5. This FSM represents the states and transitions of the application tasks:

- Inactive: task is not running nor placed on the FPGA.
- Queued: after an execution request has been granted by the scheduler and a processing unit has been chosen, the task is queued, waiting for reconfiguration, ordered in a first-come-first-served basis since task priority is handled within the scheduler waiting queue.
- Configuration: task is being configured on the FPGA.
- Context load: loads the task context (for instance after being preempted or if necessary before task execution)
- Idle: task is idle, waiting for the beginning of next period (transitory for non-periodic applications)
- Waiting: task is waiting to receive communications from its predecessors
- Running: task is running
- Preempted/mapped: task reached a preemption point and notified the reconfiguration manager. Task is not preempted yet.
- Context save: saves the task context when the task is being preempted. It may also be needed after task execution.
- Preempted/inactive: task has been actually preempted by the reconfiguration manager and has been replaced on the processing unit. It is brought back into the waiting queue in order to be configured again and resume execution.

- Mapped: task is placed on a reconfigurable region but not running. It can be safely replaced by a higher priority task by the reconfiguration manager.

3.2.5 Interfaces and communications

It is possible to use different types of interface (e.g. AXI, PLB, FIFO...). However, some tasks might have many interfaces (for instance, task 1 from Figure 3 has four interfaces: two interfaces with the testbench, one with task 2 and one with task 3) and it seems rather inappropriate to consider exhaustively all possible types of interface (AXI, PLB, FIFO...) for each reconfigurable region hosting this task (especially in case of several bus interfaces). In order to reduce the overall number of interfaces that should be implemented on a reconfigurable region, we introduce the concept of physical and virtual interfaces. Virtual interfaces are the ones required by the task as described on the diagram (the four interfaces of task 1). On the other hand, physical interfaces are the ones actually used in the implementation. The number of physical interfaces can vary from one implementation to another. A situation can occur when there is less physical than virtual interface in the application diagram. In such a case, since it is not possible to use the same interface for distinct but simultaneous data transfers, accesses to the physical interfaces should be made sequentially, using a first-come, first-served policy. In the case where no physical interfaces have been defined for the task implementation, RecoSim automatically uses the information from the diagram to provide the task with as many interfaces as declared in the diagram, to maintain the best performance.

As a matter of fact, virtual interfaces might require a permanent access to a physical channel to optimize performance. Typically, task T1 from Figure 3 has two virtual input interfaces from Testbench OUT that can represent data and metadata channels. The data channel can be defined as a priority channel: a physical interface is dedicated to this virtual interface while potential other interfaces share the remaining common interfaces. This choice, which is up to the designer, is a mean to reduce data transfer latency, as the cost of an additional resource overhead for the dedicated physical interface.

3.3 FoRTReSS flow

Figure 6 shows the different steps composing the FoRTReSS flow. Since the original flow is described in [12], only major features and enhancements are described here.

3.3.1 RR determination per task

The first step in the architecture definition process is to determine a pool of reconfigurable regions that are able to host one or more tasks of the application in terms of resources. This reference pool is built by browsing the XML-based representation of the target device resources. These RRs are shaped and mapped on the reconfigurable device with regard to the heterogeneity of the device resources, with possible RR overlapping to have a wider choice of RRs later. They are also built to be as close as possible to the resource requirements of the current task, and trying to waste the smallest amount of resources (this is called

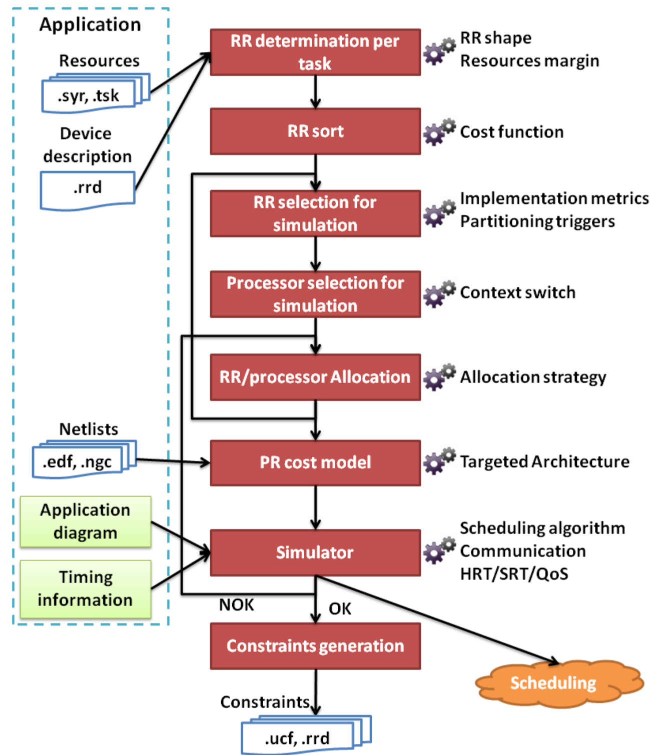

Figure 6: FoRTReSS flow

internal fragmentation). However, experience shows that defining a region based on minimising the number of resources are generally results in a failure during the routing phase. Furthermore, resources information from the synthesis step does not take routing into account. For instance, Xilinx recommends adding 5% extra resources to the reconfigurable region. For a more accurate description, FoRTReSS has a parameter for global routing margin that can be overriden for every hardware implementation that might require more routing ressources.

FoRTReSS also takes care of interfaces: as we already mentioned, it is possible to define a set of interfaces for task implementations. In case of hardware implementations, they are also used to constrain physical task placement on the FPGA. These restrictions are inferred from interfaces placed on the FPGA by the designer. It is also possible to prevent a region from being used by any reconfigurable region (static area). Typically, these features can be used to force the placement of reconfigurable regions around the actual location of interfaces (located for instance between the FPGA and Cortex cores in a Zynq-7000 device).

The search for reconfigurable regions is very useful when starting from scratch, but it is also possible to use FoRTReSS when the placement and number of reconfigurable regions are already defined. For this purpose, an XML-based file format describing predefined regions can be read to avoid being forced to explore the reconfigurable region space.

3.3.2 RR sorting

Once we have determined a pool of compatible RRs for each task, it is sorted according to a cost function in order to select the best regions for the application. The cost function is described in Equation 3.

$$\begin{aligned} Cost_{RR} = k_1 * Cost_{shape} \\ + k_2 * Cost_{compliance} \\ + k_3 * Cost_{fragmentation} \end{aligned} \quad (3)$$

The metric $Cost_{RR}$ is formulated around three components. First, it depends on the shape of the RR with the metric $Cost_{shape}$. As mentioned in previous subsection, regions can have different shapes. However, the more you complexify the shape, the less likely routing is to be efficient, this can possibly lead to failure at the floorplanning step. Therefore, we penalize regions with complex shapes by counting their vertices.

$Cost_{compliance}$ refers to the concept of Application Architecture Adequacy (often noted AAA). It takes into account the number of tasks that can be mapped on the reconfigurable region. Bigger regions might host a more important set of tasks, hence giving more freedom to the scheduler for runtime mapping of the application.

Finally, the last metric $Cost_{fragmentation}$ corresponds to the internal fragmentation and is used to penalize regions that have been built with too many resources compared to the task they host. This cost tries to avoid the waste of resources inside a reconfigurable region. This component actually reflects the percentage of unused resources inside a reconfigurable region.

Since all components have different amplitudes ($Cost_{shape}$ takes values from 4 to 10 vertices, $Cost_{compliance}$ from 0 to n incompatible tasks, n being the number of tasks in the application), it is important to weight them in order to share the same dynamics. On top of that, k_1, k_2 and k_3 in Equation 3 correspond to parameters defined in our flow that can be tuned to promote either the shape, the compliance or the fragmentation component of the cost function. Default values are set to one to give the same importance to all three components.

3.3.3 RR selection for simulation

During this step, the tool picks the RRs that will constitute the system architecture. It searches for the minimum number of RRs that will make the simulation step succeed in order to optimize the partially reconfigurable area. FoRTReSS also addresses the external fragmentation which represents the physical distance between the reconfigurable regions, calculated as the sum of all Manhattan distances between regions. Low external fragmentation reduces the total wire length and thus provides better results during the implementation phase: the regions are packed on a small part of the device, optimizing the remaining area for static logic.

There are two main approaches for the minimisation of fragmentation: reconfigurable regions should be placed as close as possible to each other or there should exist a minimum distance between them. The last option considers congestion at the interconnect level if regions are too close [29], inducing a drop

on the frequency that can be reached by the system. We decided to let this choice up to the designer with a parameter representing the minimum distance between reconfigurable regions.

In order to reduce resource requirements for PR systems, the application is partitioned. If tasks have very different resource needs, i.e. have very different resource needs (which is the case most of the time), small tasks hosted within big regions are in some ways resource inefficient. To prevent this situation, we try to group tasks with similar resource requirements: tasks within the same group will share the same reconfigurable regions. Therefore, tasks with small requirements will not be placed on regions defined for bigger tasks. The first step to partition the application consists in sorting the tasks according to a resource cost function that penalizes the waste of scarce resources. The cost of a reconfigurable resource is inversely proportional to the amount of resources available on the device, while fixing CLB (logic elements) cost to 1.

Then, the tasks are split into three categories according to pre-defined trigger values: *optimum*, *acceptable* and *unacceptable* mapping to reflect adequacy between a task and the biggest RR. An example is given in Figure 7. *Optimum* tasks are the more expensive in terms of resources and their mapping to bigger reconfigurable regions is relevant (tasks t_5 and t_6 in figure 7). *Acceptable* tasks do not waste too much resources within the reconfigurable region (tasks t_3 and t_4) while *unacceptable* tasks represent a clearly bad allocation scheme (tasks t_1 and t_2). Reconfigurable regions from the initial simulation subset (and the biggest ones according to the cost computed in previous section) will host both *optimum* and *acceptable* tasks. New regions are created based on the maximum resource needs of *acceptable* and *unacceptable* tasks, replacing some of the biggest reconfigurable regions in the initial set. These regions cannot physically host the biggest tasks, explaining the area savings that can be obtained by partitioning the application.

Trigger values are user-defined with arbitrary default values of 33% and 66% (percentage of RR resources use). The designer should set the *optimum* trigger value in order to isolate resource demanding tasks from the others. However when many tasks are isolated, more reconfigurable regions should be used and less area optimisation is also expected in this case. The second trigger prevents tasks with low requirements from being mapped to big regions and should be set so that all partitions are balanced (i.e. containing similar number of tasks). The trigger values should then be updated from one exploration to the other depending on the previous results.

3.3.4 Processor selection for simulation

While FoRTReSS focus is on the determination and placement of reconfigurable regions, it is also possible to design mixed systems including one or more software processing units. However, these elements are not processed exactly the same way as hardware reconfigurable regions. Processor cores are added statically at the beginning of the exploration process while reconfigurable regions are dynamically added to the simulation subset. In a future release, we aim to extend exploration to be able to analyse automatically software and hardware implementation opportunities. In the current version, several simulation iterations are needed to consider different number of cores.

A processor type is associated with each software implementation to de-

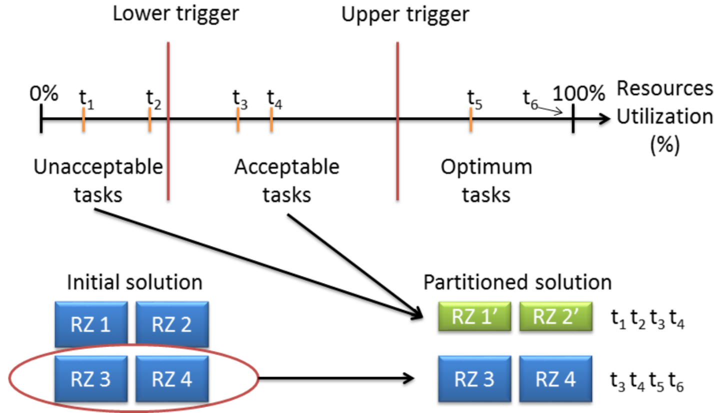

Figure 7: Partitioning step

termine on which core it can be mapped during simulation (e.g. MicroBlaze, PowerPC, Cortex A9 for Zynq platforms). We also defined a context switch parameter to represent the time required to save and restore an execution context. This parameter is processor-dependent.

3.3.5 RR and processor allocation

When an RR is selected for simulation, the tool defines which region is used for each task. This allocation step is the most complex since it is the one giving the greatest degree of freedom to the designer. The most obvious constraint is that the task should fit the RR in terms of resources. However, if the task can be placed on different RRs, this will lead to as many bitstreams stored in memories since bitstream relocation is not currently supported (using the same configuration bitstream for one task on several reconfigurable regions). The memory footprint associated with many configuration bitstreams cannot be neglected in embedded systems where memory can be a scarce resource. Also, larger memories such as DDR have higher access times than FPGA internal memories that can affect reconfiguration times. Hence, it is important not to map tasks to every possible RR but rather limiting the task-RR association to a mimimum for a better memory footprint.

There are two distinct phases in the allocation process. The first phase consists of finding a viable solution, i.e. finding the mimimum number of reconfigurable regions and processors for the application to reach the required QoS. Hence, the first allocation is pretty straightforward and consists in allowing every combination of tasks on RRs to maximise the freedom of the scheduler for on-line task placement. If in spite of this freedom simulation fails, it means that trying to improve the mapping is meaningless and that is it necessary to add another region to the simulation subset. Upon first simulation success, task allocation may be optimized. Furthermore, the first step is greedy in terms of memory usage since there could be a lot of bitstreams to store.

An optimisation based on removing pairs of task/region from the solution is applied using the following strategies:

- Least used allocation: removes the couple task/RR that is the least used

during the simulation (i.e. the association that is most likely to be removed without altering the system performance).

- Highest internal fragmentation: removes the task wasting the most resources on a reconfigurable region.
- Highest memory cost: removes the task having the biggest bitstream.

Because memory footprint improvement differs from one use case to another, there is no best optimization strategy. Therefore, the designer is left with the possibility to select the strategy that best suits to the application needs. Note that for the last strategy, involving bitstream size, we do not consider bitstream relocation (using one bitstream for configuring several regions). We will include bitstream relocation management in a future release of FoRTReSS.

Interface related constraints are not taken into account during this step: we consider it is a result of the allocation step. It is thus possible to find situations where an IP with a FIFO interface and an IP with an AXI interface would share the same reconfigurable region. In such case, the reconfigurable region must access both AXI and FIFO interfaces. A common approach is to separate the IP core from the communication interfaces in order to reduce reconfiguration time and resource overheads. Interfaces are actually connected to their associated IP core by a router which is parameterised at configuration time. As grouping IPs with different interfaces are known to be inefficient, improvements are foreseen on these aspects.

This step allocates software implementations to processor cores as well. Compared to hardware mapping, the difference lies in the optimisation strategies than can not be strictly the same: processor level optimization is based on the *Least used allocation* strategy which is the only one implemented in this case but could be extended by another strategies such as a low power strategy.

3.3.6 PR cost model

Before simulating the solution, it is necessary to calculate the reconfiguration times associated with every task-RR association. For this purpose, we use the cost model developped for an optimised reconfiguration controller called FaRM [14] (Fast Reconfiguration Manager). FaRM also allows for bitstream compression to further reduce the memory footprint without degrading configuration performance.

3.3.7 Simulation with RecoSim

At this point, we can simulate a solution to check whether this architecture is fulfilling the timing constraints of the application. The application is considered frozen and the only parameters that can be modified are those related to the scheduler. For instance, FoRTReSS comes with an Earliest Deadline First (EDF) scheduling strategy and As Many As Possible (AMAP) mapping strategy (using as many processing units as possible), but the designer can define custom strategies using dedicated APIs. The time spent by the scheduler to decide the next move is simulated in order for the simulation to be time-accurate. The main objective is to obtain a realistic schedule and, through measurements or

an accurate cost model, to ensure compliance with real-time constraints. However, this scheduler is currently under implementation based on previous work described in [7].

RecoSim can simulate the simultaneous mapping of several applications on the target FPGA (an application being a sequence of tasks started and ended by a testbench, just like the example of Figure 3). In this case, all applications are controlled by the same reconfiguration manager and the same scheduler that can be implemented either in software and hardware. Either way, schedulers are considered static and placed on a dedicated unit.

3.3.8 Layout constraints generation

When simulation has completed, a User Constraint File (UCF) is produced describing the placement of reconfigurable regions on the device, compliant with Xilinx design tools. For non-Xilinx devices, FoRTReSS also generates an XML file representing the chosen reconfigurable regions.

3.4 About the solution chosen by FoRTReSS

The solution chosen and validated by FoRTReSS is one amongst many other correct solutions in the design space. FoRTReSS mainly focuses on whether or not the architecture satisfies the application real-time constraints. Once the constraints are met, it is not necessary to continue evaluating other solutions and possibly find a better solution since we already found a valuable one. Still, it is possible to manually search for an optimal solution by running several times the FoRTReSS flow while reducing the timing constraints.

In future versions of the flow, we plan on integrating energy minimisation. The same behaviour is expected: there will be an energy consumption constraint that should be respected as well as the timing constraint. We will not be looking for a trade-off between both metrics but FoRTReSS will rather evaluate the system's feasibility. Therefore, the same approach can be preserved.

3.5 FoRTReSS Toolbox

In order to ease the joint use of FoRTReSS and RecoSim, we developed a Graphical User Interface (GUI), called FoRTReSS Toolbox [15] which allows the graphical specification of application diagrams and exploration parameters. The Eclipse Graphical Modeling Framework (GMF) [32] is used to create user interfaces based on Eclipse editor and Eclipse Modeling Framework (EMF). This environment is used to generate the C++ source code required by FoRTReSS and RecoSim using JDOM [19]. Code generation, compilation and simulation are handled within FoRTReSS Toolbox GUI. Interactions with other design tools such as Xilinx PlanAhead or Mentor Graphics ModelSim are possible to examine details of the simulation results. FoRTReSS Toolbox is compatible with Linux-based and Windows operating systems.

4 Application study & results

This section illustrates the application of FoRTReSS methodology for system level exploration of hardware software mappings involving dynamic and partial

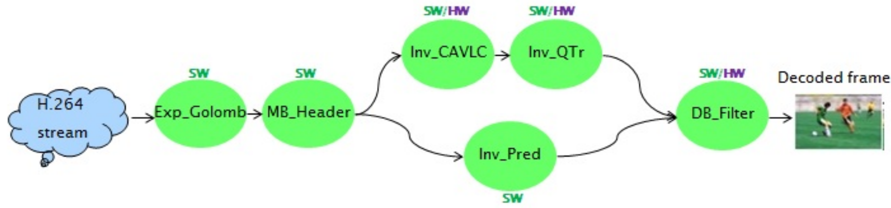

Figure 8: H.264/AVC decoder block diagram

reconfiguration. A reasonable specification assumption can be based on using C/C++ as a high level input code. Since the relevance of RR definition depends greatly on reliable characteristics of hardware accelerators, we rely on the use of High Level Synthesis (HLS) made possible by the use of C/C++ code. This approach is fully illustrated on the analysis example of an H.264/AVC decoder.

The entire system has not been implemented on the Zynq platform. Each hardware block was synthesized to extract resources and was placed and routed separately on the Zynq device to evaluate its performance accurately (i.e. WCET). Moreover, each software block was run on a Cortex-A9 (performed ten times) in order to have the most accurate worst case execution time. About the data transfer, data is located in main memory (e.g. on-board DDR3 memory). A program, running on the Cortex-A9 and using a DMA, moves data to the accelerator. Regarding software IPs, they fetch data directly from memory. We also integrated this data transfer time in the WCET. Therefore, this leads to a very good accuracy for the simulation in our tool in terms of resource and time.

However, in practice C/C++ code requires a lot of time and effort to be actually compliant with HLS rules, sometimes going as far as complete application rewriting. Therefore in a second validation study, we address the mapping exploration of an H.265/HEVC encoder from a reference C++ code released by the x265 open-source project [35]. As this type of code is often not able to comply with HLS requirements, we show how exploration can nevertheless be usefully processed from existing software and relevant hardware implementations reported in the literature as explained in section 4.4, in order to assess early mapping opportunities and their impact on performance.

4.1 H.264/AVC decoder overview

The first application which is considered for this validation study is an H.264/AVC profile video decoder. An Electronic System Level (ESL) design methodology [10] is used here to provide values of cost performance tradeoffs for possible hardware functions, which serve as an entry point to the exploration methodology of FoRTReSS. The H.264 decoder used corresponds to the block diagram of figure 8 which is a version derived from the ITU-T reference code [18] to comply with hardware design constraints and HLS. From the original C++ code, a profiling step identifies four main functionalities for acceleration that are, in order of importance, the deblocking filter (24%), the inverse context-adaptive variable-length coding (Inv. CAVLC 21%), the inverse quantization (Inv. Quant. 19%) and the inverse integer transform (Inv. Transf. 12%). To achieve better re-

Table 1: H264 task parameters on Zynq-7000 EPP (no slice decomposition)

	SW_{ex}	HW_{ex}			
$Task$	WCET(ms)	WCET(ms)	SLICE	DSP	BRAM
Exp_Golomb	1.96	n/a	n/a	n/a	n/a
MB_Header	1.96	n/a	n/a	n/a	n/a
Inv_CAVLC	20.56	5.05	3383	0	6
Inv_QTr	30.35	15.48	1202	3	7
Inv_Pred	8.81	n/a	n/a	n/a	n/a
DB_Filter	23.50	6.50	701	0	5

Table 2: H264 task parameters on Zynq-7000 EPP (two slice decomposition)

	SW_{ex}	HW_{ex}			
$Task$	WCET(ms)	WCET(ms)	SLICE	DSP	BRAM
Exp_Golomb	1.96	n/a	n/a	n/a	n/a
MB_Header	1.96	n/a	n/a	n/a	n/a
Inv_CAVLC	10.28	2.53	3383	0	6
Inv_QTr	15.18	7.74	1202	3	7
Inv_Pred	4.41	n/a	n/a	n/a	n/a
DB_Filter	11.75	3.25	701	0	5

sults, we have merged the inverse quantization and integer transform into a single block (Inv. QTr.). It might be noted here that CAVLC was not added to this block because the HLS tool (Catapult C Synthesis 2009a Release) could not handle the complexity of the resulting C++ code. Therefore, this results in three potential hardware functions representing 76% of the total processing time.

The deblocking filter, inverse CAVLC, and inverse quantization and transform block are the three functionalities of the decoder that can be either implemented in software or in dedicated hardware. In addition to these accelerating oportunities, we also consider a parallelization of the video decoder which exploits the possibility of slice decomposition of frames in the H264/AVC standard. A slice represents an independent zone of a frame, it can reference other slices of previous frames for decoding; therefore decoding one slice (of a frame) is independent from another (slice of the same frame). This way, the decoder can process different slices of a frame in parallel. We have thus considered two versions of the decoder corresponding to i) the original decoder (no slice decomposition), and ii) a two slice decomposition of the image where two streams can be processed in parallel on two halves of a same frame. This will allow considering implementations up to six accelerators and two processors for exploration. The corresponding hardware and software task parameters are reported in Table 1 for the original decoder and Table 2 for a two slice decomposition.

These parameters constitute the inputs for the exploration methodology. Hardware execution parameters (section HW_{ex} in Table 1 and 2) are derived from the full implementation of the three hardware functions identified previously on a Xilinx Zynq-7000 Extensible Processing Platform. Software tasks

(section SW_{ex} in Table 1 and 2) are described over the 667MHz ARM CortexA9 processor. The following section provides exploration results, analysis and discussion relevant to video processing constraints for this application example.

4.2 H.264/AVC decoder exploration results

The aim of this design space exploration is to determine if it is possible to decode an H.264 video stream in real-time, i.e. processing 30 frames per second (fps). For this purpose, we studied different use cases, from full software implementations towards mixed solutions, with or without making use of the slice decomposition possibility. Since FoRTReSS does not automatically explore the number of processor units, it was run with different projects increasing the number of processors. Reconfigurable regions are then automatically adjusted to fit with the configuration.

Performance and area results reported by FoRTReSS are summed up in Table 3. Area results are provided for both static and partially reconfigurable solutions in a way to put forward the relative improvements of dynamic reconfiguration. Raw improvement is computed by comparing resources of a static solution against resources required by reconfigurable regions. A total improvement is also computed considering the additional resources used by the reconfiguration controler (FaRM). For every use case presented here, we took care of choosing the shortest possible deadline, hence achieving the best framerate for the H.264 decoder. When considering a HW/SW implementations, global performance is usually limited by the number of processor units and reconfigurable regions which are directly related to the size of the target device. In this application study, we may use both Cortex-A9 cores of the Zynq-7000 platform.

The second column labelled "Framerate" in Table 3 shows that the first solution complying with a 30fps constraint is HW/SW implementation using two CPUs for a two slice decomposition (34.1 fps). For this solution, the resource improvements from the use of partial reconfiguration are 25.45%, -11.11% and -900% respectively for slice, BRAM and DSP blocks. This means that BRAM and DSP requirements have actually increased while the number of slices decreased in a small fraction. Therefore, this overhead might not promote the use of PR. However, this solution can be improved: the maximum framerate that can be reached with this architecture is more than the targeted framerate of 30 frames per second. Hence, the application deadline can be increased to 33.3 ms: this is the last use case in Table 3. Proceeding this way brings significant improvement with 54% slice improvement (less than half of previous slice requirements, from just reducing the framerate constraint) and even 44% BRAM improvement.

Across all implementations, we notice a important overhead for DSP resources in Table 3. It is due to the limited DSP requirements (only 3 DSP48s) and reconfiguration granularity: each reconfigurable region must span an entire clock domain, hence constraining every resource within the column. For DSP resources, each column is composed of 20 DSP48s, which is the minimum number of DSP48s that can be set for a Zynq-7000 device. Therefore, there will always be an important resource overhead for DSP elements. If this is unacceptable to the designer (for instance, if the remaining static logic is very DSP-consuming), it is always possible to change synthesis parameters in order to prohibit the synthesizer from inferring DSP blocks and use logic elements instead.

Table 3: Performance and area results of H.264 decoder implementations

Implementation	Framerate (fps)	Resource type	Static area	Number of RRs	PR area (columns)	PR area	Improvement (%) Raw	Total
Full SW 1 slice 1 core	11.4	Slice BRAM DSP	n/a	n/a	n/a	n/a	n/a	n/a
Full SW 2 slices 2 cores	21.7	Slice BRAM DSP	n/a	n/a	n/a	n/a	n/a	n/a
HW/SW 1 slice 1 core	23.3	Slice BRAM DSP	5551 18 3	1	72 1 1	3600 10 20	35.15 44.4 - 566.7	29.40 44.4 - 566.7
HW/SW 2 slices 1 core	28.2	Slice BRAM DSP	11102 36 6	4	140 4 3	7000 40 60	36.95 - 11.11 -900	25.45 - 11.11 -900
HW/SW 2 slices 2 cores	34.1	Slice BRAM DSP	11102 36 6	4	140 4 3	7000 40 60	36.95 - 11.11 -900	25.45 - 11.11 -900
HW/SW 2 slices 2 cores	30	Slice BRAM DSP	11102 36 6	2	88 2 1	4400 20 20	60.37 44.44 - 233.3	54.62 44.44 - 233.3

Table 4: Occupation rate results of H.264 decoder implementations

Implementation	Framerate	Core (%)		Reconfigurable Region (%)					
	(fps)	Core1	Core2	RR1	RR2	RR3	RR4	RR5	RR6
HW/SW (2 slices/cores)	34.1	48.42	43.37	26.39	11.08	26.39	11.08	n/a	n/a
HW/SW (2 slices/cores)	30	88.09	73.40	n/a	n/a	n/a	n/a	9.74	30.77

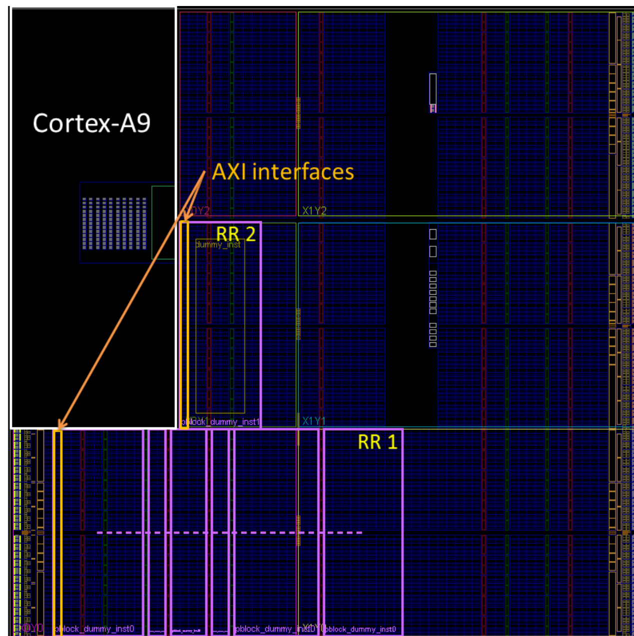

Figure 9: Floorplan with interface constraints on a Zynq-7000 platform

To better understand the results obtained when reducing the targeted framerate, the Table 4 shows the occupation rates of the last two solutions of the Table 3 depending on the framerate and the possible ICAVLC allocation (RR or core). Reducing the framerate gives the scheduler the opportunity to use a slower task implementation (the software one) rather than a faster hardware implementation, but with a resource overhead in return. The 30 fps use case shows much higher occupation rates than the 34.1 use case (73.4 and 88.09% compared to 43.37 and 48.42% respectively for both cores).

4.3 Adding interfaces constraints to FoRTReSS

Solutions found by FoRTReSS did not take into account any location constraints and hence, placement of the reconfigurable regions on the FPGA is arbitrary. However, our application mixes both hardware and software tasks communicating with each other. Hence, task placement can be optimized by placing the reconfigurable regions closer to the Cortex-A9 processors.

Constraining regions is automated within FoRTReSS: for every hardware implementation in the design, the interface requirements are specified so that every region that can host the task must also provide this type of interface. In our case, all accelerators use a 32-Bit AXI high-performance slave port (at 100MHz) between the programmable logic zone and the processing system. Then, the location of potential interfaces has to be specified by the user. Figure 9 shows the location of the AXI interfaces as well as the resulting floorplan of the Zynq-7000 platform for previous use case (two RRs and two processor cores working on two slices at 30 frames per second). In this case, we defined two possible locations

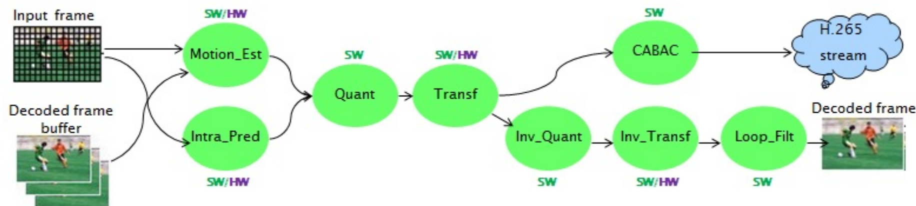

Figure 10: H.265/HEVC encoder flow graph

Table 5: x265 task parameters on Zynq-7000 EPP (no slice decomposition)

$Task$	SW_{ex} WCET(ms)	HW_{ex} WCET(ms)	SLICE	DSP	BRAM
Motion_Est	2418	9.61	14067	0	297
Intra_Pred	126	1.43	472	0	4
Quant	37	n/a	n/a	n/a	n/a
Tranf	21	0.28	3595	0	0
CABAC	139	n/a	n/a	n/a	n/a
Inv_Quant	5	n/a	n/a	n/a	n/a
Inv_Transf	5	0.28	3595	0	0
Loop_Filt	23	n/a	n/a	n/a	n/a

for AXI interfaces, close to the Cortex-A9 processor. In order to comply with the reconfiguration granularity, the possible interface locations span an entire clock domain. We can see that both reconfigurable regions found by FoRTReSS effectively include one AXI interface, ensuring enhanced performance during the FPGA implementation phase.

4.4 H.265/HEVC encoder

In this second application study, we show the ability of the method to help system level mapping analysis from early-stage application specifications. The application considered is a recently released H.265/HEVC encoder from the x265 open source project which software is available since July 2014 [35]. Figure 10 shows the application graph derived from the reference C++ code.

Ideally the characteristics of hardware mappings of tasks needed for exploration can originate from HLS, provided that the C/C++ source can be processed. This is generally not the case for most applications, including x265, as only a subset of C/C++ is usually supported for hardware acceleration. The x265 code makes widespread use of dynamic memory allocation and arbitrary indirection (pointers that are not static arrays) that will require extensive effort to remove. However many works have already been devoted to the implementation of critical processing blocks of H.265/HEVC since the technical content of HEVC was finalized at the beginning of 2013.

Table 5 reports the characteristics of three relevant hardware implementations on Virtex FPGAs that can be used to process an early exploration of the

Table 6: x265 task parameters on Zynq-7000 EPP (two slice decomposition)

$Task$	SW_{ex} WCET(ms)	HW_{ex} WCET(ms)	SLICE	DSP	BRAM
Motion_Est	1209	4.81	14067	0	297
Intra_Pred	63	0.72	472	0	4
Quant	19	n/a	n/a	n/a	n/a
Tranf	11	0.14	3595	0	0
CABAC	70	n/a	n/a	n/a	n/a
Inv_Quant	3	n/a	n/a	n/a	n/a
Inv_Transf	3	0.14	3595	0	0
Loop_Filt	12	n/a	n/a	n/a	n/a

acceleration potential with dynamic reconfiguration: a motion estimation engine [11] which is the most computational part of the encoding process, a high performance intra prediction hardware [21] and an accelerator supporting fast forward and inverse two-dimensional transforms [33]. All implementation and performance results come from Xilinx Virtex devices and have been extrapolated to comply with the same video resolution and a running frequency of 100MHz. We target the same platform (Zynq-7000 EPP) as depicted in section 4.1. As for H.264, H.265 supports slice decomposition to allow frame level parallelism when encoding using multiple cores. Task parameters corresponding to a two slice decomposition are thus also considered and depicted in table 6.

4.5 H.265/HEVC encoder exploration results

Like for the previous H.264 application study, we cover different configurations from full software to hardware software implementations with or without slice decomposition. The corresponding results are reported in table 7.

There are sensitive performance benefits in the three hardware software solutions defined, until 8.82 fps using two cores and three RRs over software execution (respectively 0.36 fps and 0.7 fps for 1 slice/1 core and 2 slices/2 cores). However this use case is very much affected by the presence of a big computation kernel represented by motion estimation which engages 14067 slices, that is 65% of the total accelerator slices. There is no room for sharing and dynamically reconfiguring a single RR able to host all tasks without an important performance penalty (26.8 ms of reconfiguration time). As a result the best solution is based on the use of three RRs, one for each type of function ($Tranf$ and Inv_Transf functions share the same accelerator IP). There is no possible resource improvement from using dynamic reconfiguration in this example. In fact there is even a slight increase of logic resource due to the inevitable oversizing of a Reconfigurable Region to be able to host a task.

Nevertheless this application study shows the ability of the methodology to easily evaluate the relevance of using dynamic reconfiguration or not from very early development stages. In addition, further investigation can permit to propose and analyze a set of possible improvements. For example, the scheduling details of the 8.82 fps solution reported in figure 11 clearly shows that CABAC function becomes a new major bottleneck after global acceleration. Therefore

Table 7: Performance and area results of x265 encoder implementations

Implementation	Framerate (fps)	Resource type	Static area	Number of RRs	PR area (columns)	PR area	Improvement (%) Raw	Total
Full SW 1 slice 1 core	0.36	Slice BRAM DSP	n/a	n/a	n/a	n/a	n/a	n/a
Full SW 2 slices 2 cores	0.7	Slice BRAM DSP	n/a	n/a	n/a	n/a	n/a	n/a
HW/SW 1 slice 1 core	4.58	Slice BRAM DSP	22673 301 0	3	552 32 0	27600 320 0	-21.7 -6.31 0	-26 -6.31 0
HW/SW 1 slice 2 cores	5.33	Slice BRAM DSP	22673 301 0	3	552 32 0	27600 320 0	-21.7 -6.31 0	-26 -6.31 0
HW/SW 2 slices 2 cores	8.82	Slice BRAM DSP	22673 301 0	3	552 32 0	27600 320 0	-21.7 -6.31 0	-26 -6.31 0

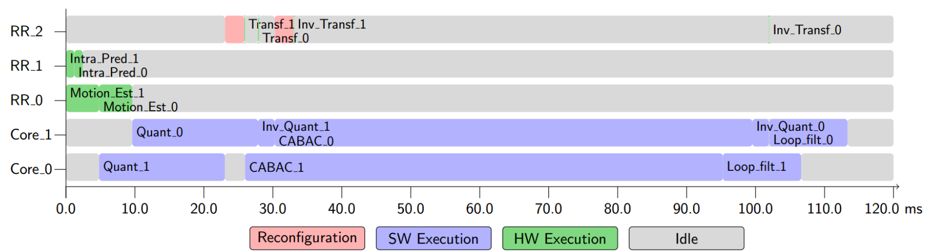

Figure 11: Scheduling of the best X.265 accelerated solution.

a hardware implementation for this function, but also for loop filters (sample adaptive offset, deblocking filter) to a lesser extent, can certainly allow to reach real time processing (e.g. 30 fps) and, depending on their size, change the question of dynamic reconfiguration impact. We did not process these functions as we found no hardware implementation reported in the literature yet.

4.6 Design time with FoRTReSS

The design cycle with FoRTReSS can actually be split in two: application specification using FoRTReSS Toolbox and actual execution of the FoRTReSS flow. The graphical specification of the application diagram and FoRTReSS configuration takes less than an hour, which does not include the time required to obtain the resource requirements for every hardware implementation of the tasks and the timing information. This effort is done just the first time an application is defined: diagrams can be imported and reused into multiple projects. Then, each execution of the FoRTReSS flow can take up to a minute, depending on the complexity of the RecoSim simulation and the unknown number of allocation improvements that are performed during exploration. Therefore, it has also been shown that it was possible to estimate the impact of partial reconfiguration on an application in a short amount of time, which cannot be done with current design flows. The H.265 encoder mapping example was processed from scratch in a matter of three days. To do this, we needed descriptions of hardware implementations or at least an estimation of the required resources and the corresponding hardware and software execution times.

5 Future work

In future versions of FoRTReSS, we would like to add some energy considerations. For now, FoRTReSS highly focuses on real-time constraints to automatically identify performance compliant solution(s) under energy minimisation requirement for example. We believe that combining both temporal and energetic considerations for partially reconfigurable systems into a single tool can be very time-saving for the designer interested in power efficiency. For instance, it is possible to develop and evaluate scheduling algorithms with RecoSim that promote energy consumption when considering selecting task implementations and allocation or using blank reconfigurable regions to minimise the global energy cost while potentially considering DVFS oportunies at the processor level for realistic multicore low power execution. Therefore, we would like to integrate these considerations into the reconfigurable region selection and the allocation optimization process.

6 Conclusion

In this paper, we introduced FoRTReSS, a flow enabling design space exploration for partially reconfigurable applications in real-time systems. FoRTReSS provides the designer with a convenient tool for estimating hardware, software and mixed solutions using partial reconfiguration. The hardware design space exploration is automated by FoRTReSS, which infers a set of reconfigurable regions from task resources information. FoRTReSS relies on a SystemC simulator,

RecoSim, that allows the designer to develop and evaluate its own scheduling algorithms. We described in depth the features and abilities of FoRTReSS on a H.264 video decoding application targeting a framerate of 30 frames per second on a Zynq-7000 platform. Several implementations of the decoder were evaluated, from a full software solution working on the entire stream to a solution using hardware accelerators and working on a two slice decomposition of the stream. We have shown that a first solution was indentified by FoRTReSS to process a framerate of 34.1 frames per second, but with an important resources overhead. Finally, extending slightly the deadlines permitted to reach a 30 fps solution with very interesting area improvements, saving more than half the slice resources compared to a static implementation and using 44% less memory resource. We have additionally shown the usefulness of the framework to help analyzing mapping opportunities from early C++ specifications on a H.265/HEVC encoder. The different explorations performed within this work were facilitated by the use of FoRTReSS Toolbox, a GUI for controlling the FoRTReSS flow.

Acknowledgements

This work was carried out in the framework of project ARDMAHN [2] sponsored by the French National Research Agency under grant ANR-09-SEGI-001, which aims at developing methodologies for home gateways integrating dynamic and partial reconfiguration. This work is also carried out under the BENEFIC project (CA505), a project labelled within the framework of CATRENE, the EUREKA cluster for Application and Technology Research in Europe on NanoElectronics.

References

[1] Altera: Quartus II Handbook Version 12.1 - Volume 1: Design and Synthesis - Design Planning for Partial Reconfiguration (2012)

[2] ARDMAHN consortium: ARDMAHN project. http://ARDMAHN.org/ (2013)

[3] Bazargan, K., Kastner, R., Sarrafzadeh, M.: Fast template placement for reconfigurable computing systems. IEEE Design and Test of Computers **17**(1), 68–83 (2000). DOI http://doi.ieeecomputersociety.org/10.1109/54.825678

[4] Beckhoff, C., Koch, D., Torresen, J.: Go ahead: A partial reconfiguration framework. In: Proceedings of the 2012 IEEE 20th International Symposium on Field-Programmable Custom Computing Machines, FCCM '12, pp. 37–44. IEEE Computer Society, Washington, DC, USA (2012)

[5] Belaid, I., Muller, F., Benjemaa, M.: New three-level resource management enhancing quality of off-line hardware task placement on fpga. EURASIP International Journal of Reconfigurable Computing (IJRC) (2010)

[6] Belaid, I., Muller, F., Benjemaa, M.: Static scheduling of periodic hardware tasks with precedence and deadline constraints on reconfigurable hardware devices. Special Issue in EURASIP International Journal of Reconfigurable Computing (IJRC) (2011)

[7] Bilavarn, S., Khan, J., Belleudy, C., Bhatti, M.K.: Effectiveness of power strategies for video applications: A practical study. Journal of Real-Time Image Processing (2014). DOI 10.1007/s11554-013-0394-6. URL http://link.springer.com/article/10.1007/s11554-013-0394-6

[8] Compton, K., Hauck, S.: Reconfigurable computing: a survey of systems and software. ACM Comput. Surv. **34**, 171–210 (2002)

[9] Corbetta, S., Morandi, M., Novati, M., Santambrogio, M.D., Sciuto, D., Spoletini, P.: Internal and external bitstream relocation for partial dynamic reconfiguration. IEEE Trans. Very Large Scale Integr. Syst. **17**(11), 1650–1654 (2009)

[10] Damak, T., Werda, I., Bilavarn, S., Masmoudi, N.: Fast prototyping h.264 deblocking filter using esl tools. In: Transactions on Systems, Signals & Devices, Shaker Verlag, Vol. 8, No 3, pp.345-362 (2013)

[11] D'huys, T.P.K.C., Momcilovic, S., Pratas, F., Sousa, L.: Reconfigurable data flow engine for hevc motion estimation. In: IEEE International Conference on Image Processing (ICIP) (2014)

[12] Duhem, F., Muller, F., Aubry, W., Le Gal, B., Ngru, D., Lorenzini, P.: Design space exploration for partially reconfigurable architectures in real-time systems. Journal of Systems Architecture (JSA) **59**(8), 571 – 581 (2013). DOI http://dx.doi.org/10.1016/j.sysarc.2013.06.007. URL http://www.sciencedirect.com/science/article/pii/S1383762113001215

[13] Duhem, F., Muller, F., Lorenzini, P.: Methodology for designing partially reconfigurable systems using transaction-level modeling. In: Conference on Design and Architectures for Signal and Image Processing (DASIP), pp. 316–322 (2011)

[14] Duhem, F., Muller, F., Lorenzini, P.: Reconfiguration time overhead on field programmable gate arrays: reduction and cost model. IET Computers & Digital Techniques **6**(2), 105–113 (2012)

[15] Fabrice Muller: FoRTReSS Toolbox (Flow for Reconfigurable architecture in Real-time Systems). https://sites.google.com/site/fortresstoolbox/ (2014)

[16] Flynn, A., Gordon-Ross, A., George, A.D.: Bitstream relocation with local clock domains for partially reconfigurable fpgas. In: Proceedings of the Conference on Design, Automation and Test in Europe, DATE '09, pp. 300–303. European Design and Automation Association, 3001 Leuven, Belgium, Belgium (2009)

[17] Foucher, C., Muller, F., Giulieri, A.: Fast integration of hardware accelerators for dynamically reconfigurable architecture. In: IEEE CAS 7th International Workshop on Reconfigurable Communication-centric Systems-on-Chip (ReCoSoC 2012). IEEE, York, UK (2012)

[18] ITU-T: ISO/IEC 14496-10, Advanced Video Coding for Generic Audiovisual Services, ITU-T Recommendation H.264, Version 4 (2005)

[19] JDOM Project: JDOM. http://www.jdom.org/ (2012)

[20] Jozwik, K., Tomiyama, H., Honda, S., Takada, H.: A novel mechanism for effective hardware task preemption in dynamically reconfigurable systems. In: FPL'10, pp. 352–355 (2010)

[21] Kalali, E., Adibelli, Y., Hamzaoglu, I.: A high performance and low energy intra prediction hardware for high efficiency video coding. In: IEEE 22nd International Conference on Field Programmable Logic and Applications (FPL 2012) (2012)

[22] Kao, C.: Benefits of Partial Reconfiguration. Xcell Journal **55**, 65–67 (2005)

[23] Kumar, R., Gordon-Ross, A.: Formulation-level design space exploration for partially reconfigurable fpgas. In: International Conference on Field-Programmable Technology (FPT), pp. 1–6 (2011)

[24] Lodi, A., Martello, S., Vigo, D.: Heuristic and metaheuristic approaches for a class of two-dimensional bin packing problems. INFORMS Journal on Computing **11**(4), 345–357 (1999)

[25] Lodi, A., Martello, S., Vigo, D.: Neighborhood search algorithm for the guillotine non-oriented two-dimensional bin packing problem. In: Meta-Heuristics, pp. 125–139. Springer US (1999)

[26] Manet, P., Maufroid, D., Tosi, L., Gailliard, G., Mulertt, O., Di Ciano, M., Legat, J.D., Aulagnier, D., Gamrat, C., Liberati, R., La Barba, V., Cuvelier, P., Rousseau, B., Gelineau, P.: An evaluation of dynamic partial reconfiguration for signal and image processing in professional electronics applications. EURASIP J. Embedded Syst. **2008**, 1:1–1:11 (2008)

[27] Montone, A., Santambrogio, M.D., Redaelli, F., Sciuto, D.: Floorplacement for partial reconfigurable fpga-based systems. Int. J. Reconfig. Comput. **2011**, 2:1–2:12 (2011)

[28] Paulsson, K., Hübner, M., Bayar, S., Becker, J.: Exploitation of Run-Time Partial Reconfiguration for Dynamic Power Management in Xilinx Spartan III-based Systems. In: International Conference on Field Programmable Logic and Applications (FPL), pp. 699–700 (2008)

[29] Singhal, L., Bozorgzadeh, E.: Multi-layer floorplanning for reconfigurable designs. IET Computers & Digital Techniques **1**(4), 276–294 (2007)

[30] Sohanghpurwala, A.A., Athanas, P., Frangieh, T., Wood, A.: OpenPR: An Open-Source Partial-Reconfiguration Toolkit for Xilinx FPGAs. In: IPDPS Workshops, pp. 228–235. IEEE (2011)

[31] Tessier, R., Burleson, W.: Reconfigurable computing for digital signal processing: A survey. J. VLSI Signal Process. Syst. **28**, 7–27 (2001)

[32] The Eclipse Foundation: Eclipse Graphical Modeling Framework (GMF). http://www.eclipse.org/modeling/gmp/ (2013)

[33] Tiago Dias, N.R., Sousa, L.: Unified transform architecture for avc, avs, vc-1 and hevc high-performance codecs. In: EURASIP Journal on Advances in Signal Processing (2014)

[34] Vipin, K., Fahmy, S.A.: Architecture-Aware Reconfiguration-Centric Floorplanning for Partial Reconfiguration. In: Proceedings of the 8th International Symposium on Applied Reconfigurable Computing (ARC), pp. 13–25 (2012)

[35] x265 Project: x265. http://x265.org/ (2014)

[36] Xilinx: EPPs: The Ideal Solution for a Wide Range of Embedded Systems (2012)

[37] Xilinx: Partial Reconfiguration User Guide (2012)