A Mapping-Scheduling Algorithm for Hardware Acceleration on Reconfigurable Platforms

JUAN ANTONIO CLEMENTE, Universidad Complutense de Madrid
IVAN BERETTA, VINCENZO RANA, DAVID ATIENZA,
École Polytechnique Fédérale de Lausanne
DONATELLA SCIUTO, Politecnico di Milano

Reconfigurable platforms are a promising technology that offers an interesting trade-off between flexibility and performance, which many recent embedded system applications demand, especially in fields such as multimedia processing. These applications typically involve multiple ad-hoc tasks for hardware acceleration, which are usually represented using formalisms such as Data Flow Diagrams (DFDs), Data Flow Graphs (DFGs), Control and Data Flow Graphs (CDFGs) or Petri Nets. However, none of these models is able to capture at the same time the pipeline behavior between tasks (that therefore can coexist in order to minimize the application execution time), their communication patterns, and their data dependencies. This article proves that the knowledge of all this information can be effectively exploited to reduce the resource requirements and the timing performance of modern reconfigurable systems, where a set of hardware accelerators is used to support the computation. For this purpose, this article proposes a novel task representation model, named Temporal Constrained Data Flow Diagram (TCDFD), which includes all this information. This article also presents a mapping-scheduling algorithm that is able to take advantage of the new TCDFD model. It aims at minimizing the dynamic reconfiguration overhead while meeting the communication requirements among the tasks. Experimental results show that the presented approach achieves up to 75% of resources saving and up to 89% of reconfiguration overhead reduction with respect to other state-of-the-art techniques for reconfigurable platforms.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable Architectures*; C.3 [**Special-Purpose and Application-Based Systems**]: *Real-time and embedded systems*

General Terms: Design, Algorithms, Performance

Additional Key Words and Phrases: Mapping, task scheduling, reconfiguration overheads, reconfigurable systems, runtime reconfiguration

ACM Reference Format:
Juan Antonio Clemente, Ivan Beretta, Vincenzo Rana, David Atienza, and Donatella Sciuto, 2014. A Mapping-scheduling algorithm for hardware acceleration on reconfigurable platforms. ACM Trans. Reconfig. Technol. Syst. 7, 2, Article 9 (June 2014), 27 pages.
DOI: http://dx.doi.org/10.1145/2611562

This work was supported in part by the EC FP7 FET SCoRPiO project (no. 318013), and the ObeSense RTD project (no. 20NA21_143081) evaluated by the Swiss NSF and funded by Nano-Tera.ch with Swiss Confederation financing; and by the Spanish Ministry of Education, Culture and Sports under grants TIN2009-09806 and AYA2009-13300-C03-02.
Authors' addresses: J. A. Clemente, Computer Architecture Department, Universidad Complutense de Madrid, Madrid 20840, Spain; email: ja.clemente@fdi.ucm.es; I. Beretta, V. Rana, and D. Atienza: ESL-École Polytechnique Fédérale de Lausanne, ESL-IEL-STI-EPFL, Lausanne 1015, Switzerland; D. Sciuto, DEI-Politecnico di Milano, Milan 20113, Italy.

DOI: http://dx.doi.org/10.1145/2611562

1. INTRODUCTION

In the last few years, embedded system platforms have been increasingly employed in fields such as image processing, multimedia applications and artificial vision, which often require complex and computationally intensive algorithms. In order to face these requirements, embedded devices have evolved into complex Systems-on-Chip (SoCs) that are able to guarantee high performance thanks to the combined execution of heterogeneous functional units. Examples of such functional units are general purpose processors, digital signal processors (DSPs), or specific purpose IP cores for hardware acceleration: in the remaining of this article, we will use the term *task* to refer to any of these components.

Given the high number of tasks required by the applications and the limited amount of available resources, new-generation platforms in both industry [Xilinx 2012b] and academia [Chang et al. 2005]; Walder and Platzner [2004] propose a combination of fixed general purpose tasks (CPUs) and reconfigurable resources (such as Field Programmable Gate Arrays (FPGAs)) similar to the one shown in Figure 1, which allows the hardware to adapt itself to different execution contexts. The additional support for dynamic reconfiguration allows tasks to be dynamically added and removed from the system depending on the application being executed. However, a well-known drawback of FPGAs is the time penalty related to their reconfiguration, which can be in the order of hundreds of milliseconds [Xilinx 2010; Kao 2006], which may greatly degrade the system performance. Hence, a good scheduling strategy is needed not only to efficiently distribute the tasks on the available resources, but also to minimize the reconfiguration overhead and to guarantee that the application achieves the required performance.

The scheduling problem in the context of reconfigurable devices consists in finding a sequence of reconfigurations that limits the time penalty and optimizes both resource usage and performance, while guaranteeing that the tasks are configured on the device when they are needed. As a consequence, it is necessary to characterize the application in terms of interdependencies between the tasks, that is, how often they communicate and when two or more tasks constitute a pipeline structure. Typically, scheduling approaches rely on representations such as Data Flow Graphs (DFGs) [Kavi et al. 1986], Control and Data Flow Graphs (CDFGs) [Zaretsky et al. 2005], and Petri Nets [Zurawski and Zhou 1994], which capture data dependencies and may include conditional and synchronization information. However, none of these representations captures the necessity of multiple tasks to coexist at the same time. Another popular representation model is Data Flow Diagrams (DFDs) [Bruza and van der Weide 1993], which is focused on the amount of data exchanged between each pair of tasks, but it does not include any information about data dependencies to determine, for instance, whether two tasks must be executed in sequence or should coexist.

In this article, we propose a hybrid mapping-scheduling algorithm for reconfigurable SoCs based on an innovative representation model that we named *Temporal Constrained Data Flow Diagram* (TCDFD), which combines the data dependencies of DFGs (and consequently of CDFGs and Petri Nets) and the coexisting dependencies captured by DFDs. The proposed algorithm determines an efficient mapping at design time by fully exploiting the information included in the TCDFD, and then performs an event-based scheduling at run time in order to minimize the reconfigurable resources consumption while achieving the required timing performance. The proposed approach aims at mapping and scheduling, without loss of generality, a single TCDFD, which can represent, as explained in Section 2.2, a single application or a set of applications to be executed concurrently.

Our experimental results show that the proposed approach achieves up to 75% of resources saving and up to 89% reduction in terms of reconfiguration overhead with

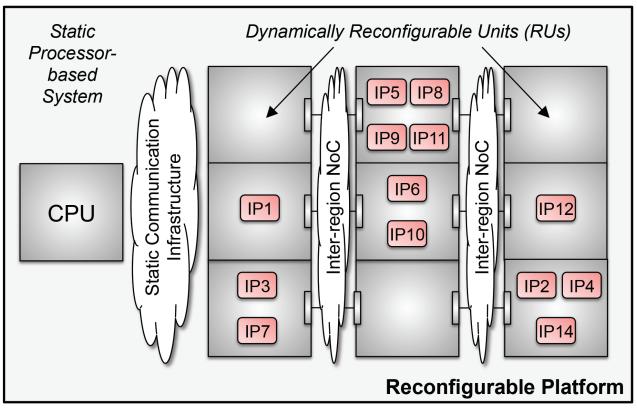

Fig. 1. The target hardware architecture.

respect to other representative state-of-the-art approaches [Murali et al. 2006a; Beretta et al. 2011b; Clemente et al. 2011b]. In addition, we illustrate how our approach can be applied to actual multimedia applications [Theelen et al. 2008; Taghipour et al. 2008; Verderber et al. 2003; Janiaut et al. 2005; Roitzsch 2007; Lindroth et al. 2006; Mei-hua et al. 2007] running on a real system developed on a Xilinx^{TM}Virtex-5 FPGA. With respect to other state-of-the-art approaches [Beretta et al. 2011b; Clemente et al. 2011a], we hereby introduce our contributions.

—We present the novel TCDFD formalism, in order to better capture all the relevant information related to the tasks and their execution.
—We give the definition of the runtime scheduling and the prefetching phases to further reduce the actual reconfiguration overhead.
—We show a set of major improvements in the algorithm presented in Clemente et al. [2011a], including the *Partitioning Reduction* optimization (see Section 5), which lead to a more efficient area usage and a reduced reconfiguration overhead.
—We demonstrate the validation of the proposed approach on new real-world case studies, including the MP3 [Theelen et al. 2008; Taghipour et al. 2008] and the H.264 [Roitzsch 2007] codecs.

The rest of the article is structured as follows. Section 2 provides a detailed definition of the context of this work. Next, Section 3 describes the most relevant state-of-the-art task representation models, namely DFGs, CDFGs, DFDs and Petri Nets, and Section 4 presents the novel TCDFD formalism. Section 5 describes the proposed mapping-scheduling algorithm, which is validated by means of a set of experimental results that are discussed in Section 6. Finally, Section 7 summarizes this work with the final conclusions.

2. CONTEXT DEFINITION

This section presents a description of the mapping-scheduling problem for the target reconfigurable architectures that we consider in this article. We also discuss a real-world application that we use as a reference throughout the description of the proposed algorithm: the MPEG-4 Layer 2 SP (Single Profile) decoder [Theelen et al. 2008; Verderber et al. 2003; Janiaut et al. 2005]. Thanks to the analysis of the case study, we are able to show the main limitations of the current representation models, and to motivate the definition of a new formalism that fulfills the requirements that we identify in Section 2.4.

2.1. The Target Hardware Architecture

The proposed approach is meant to fully exploit the potential of new-generation extensible hardware platforms, which combine a processor-based system for general purpose computation, and dynamically reconfigurable logic, as shown in Figure 1. However, it is also generally applicable to any platform that supports dynamic reconfiguration, as the processor-based part of the architecture can be emulated by means of a soft processor (e.g., MicroBlaze [Xilinx 2012a]) on the programmable logic.

In our work, we organize the programmable logic according to the structure shown in Figure 1. This popular way of managing the reconfigurable part of the architecture consists in dividing it into a grid of homogeneous elementary regions named *reconfigurable units* (RUs), in order to guarantee a high degree of regularity in the architecture and to fully support dynamic reconfiguration and module relocation [Corbetta et al. 2007] (it would be possible to employ RUs of different sizes, but this would prevent relocation and heavily limit the potential of the architecture). A RU is a self-contained region that represents the smallest amount of area that is reconfigured at once, and includes one or more tasks and an intra-region communication infrastructure that connects them (which is not depicted in the figure for the sake of simplicity). Furthermore, each RU contains an interface towards a global or inter-region communication infrastructure that connects all the RUs on the FPGA. All the communication infrastructures are implemented using a Network-on-Chip (NoC) [Benini and De Micheli 2002], which provides good performance as well as good scalability for large numbers of RUs and tasks. However, other techniques (e.g., buses, point-to-point connections) are also possible. The intra-region NoC is specifically designed to connect the tasks in the RU and it is reconfigured along with them, thus it can be assumed to be optimal in terms of bandwidth and traffic distribution. The inter-region NoC is implemented as a fixed backbone, and it is operational even during the reconfiguration of the RUs. In Xilinx™devices, this is achieved by connecting the RUs to the static NoC by means of *bus macros*, which are particular hardware blocks that are instantiated along the edges of the reconfigurable regions and are not affected by the reconfiguration process [Kao 2006].

The inter-region NoC has a fixed structure (in this work, we assume a mesh grid topology with an *XY* routing algorithm) and its efficiency depends on the distance between two RUs as well as on the traffic. As a consequence, it is essential to resolve most of the communications within the same RU in order not to congest the global NoC and not to affect the performance of the system. In this case, the topology of the global NoC can be defined to guarantee a certain minimum throughput between tasks mapped in different RUs, following static design approaches that can be found in the literature [Lukovic and Fiorin 2008].

The dynamic reconfiguration process of the programmable logic is performed by the processor-based system, which is also static during the execution. The processor can access an internal port and write a configuration into a desired RU. The configuration of a RU is stored in a binary file named *partial bitstream*, which reconfigures only the target portion of the FPGA area, and it can be reused to write the same configuration into multiple RUs by means of a technique known as *bitstream relocation* [Corbetta et al. 2007]. On the communication side, the connection between the processor and the programmable logic (which can be implemented by means of either a bus or a NoC), as well as the static inter-region backbone (as mentioned earlier), have been implemented using bus macros [Kao 2006]. These kind of logic blocks have been used to guarantee portability over different platforms, and to effectively allow these infrastructures to be operational even during the reconfiguration of one of the RUs.

We have successfully implemented this platform on a Xilinx™XUPV5-LX110T development board, which features a Virtex-5 FPGA. This system allowed us to

experimentally evaluate its relevant aspects from the point of view of the mapper-scheduler presented in this article: NoC scalability and power consumption due to the dynamic reconfigurations. On the one hand, the NoC showed a good scalability as a moderate number of tasks assigned to the same RU in the system. On the other hand, experimental results on power consumption due to the dynamic reconfigurations are discussed in Section 6.4. More low-level implementation details about this architecture can be found at [Beretta et al. 2011a].

2.2. Problem Statement

In this article, we address the combined design-time mapping and runtime scheduling of tasks on dynamically reconfigurable platforms, as defined as follows.

(1) *Reconfigurable tasks mapping.* The mapping of tasks on reconfigurable devices consists in finding a subset of reconfigurable resources on which these tasks can be placed. Typically, mapping approaches aim at optimizing the utilization of the available reconfigurable resources. In addition, depending on the features of the target reconfigurable architecture, they also minimize the time penalty incurred in the communications among the tasks.
(2) *Reconfigurable tasks scheduling.* The scheduling problem in the context of reconfigurable devices consists in finding a sequence of reconfigurations, as well as in deciding when to trigger the execution of each task in such a way that the data dependencies among them are respected. In our system, an application has a *deadline* associated with it, which is a temporal constraint of the application. It indicates how critical its execution is and it can be determined either by the user, an operating system or a middleware that exists upon the mapper-scheduler that we present in this article. Hence determining it is out of the scope of this work.
(3) *Objectives of the proposed mapping-scheduling algorithm.* The algorithm proposed in this article aims at mapping and scheduling the tasks of a given TCDFD on the hardware dynamically reconfigurable architecture shown in Section 2.1, by reducing the reconfiguration overhead of the applications in order to meet their deadlines, while minimizing the reconfigurable resources consumption. Even though our approach aims at mapping and scheduling a single TCDFD at a time, it can also handle a scenario with multiple concurrent applications. In fact, the concurrent execution of more than one application (let us consider a couple of applications A and B) can always be seen as the execution of a larger application (let us call it application C, including all the cores of both A and B) described by a single TCDFD consisting of distinct subapplications which do not have any data dependencies among them.
(4) *Reconfiguration overhead.* In the remainder of this article, we will refer to *reconfiguration overhead* as the delay introduced in the execution of an application due to the latencies of the runtime reconfigurations. It is computed as follows:

$$rec_{overhead} = Ex\_time - Ex\_time_{without\_rec\_latency}, \tag{1}$$

where Ex_time is the execution time of the application and $Ex\_time_{without\_rec\_latency}$ is the execution time assuming that the latency incurred into the reconfiguration of the tasks is 0.

2.3. Case Study: MPEG-4 Layer 2 Decoder

The popular MPEG-4 Layer 2 decoder is an algorithm for digital video compression [Theelen et al. 2008; Verderber et al. 2003; Janiaut et al. 2005]. An MPEG-4 video is a stream of frames, each of one consisting of *Macro Blocks* of 16×16 pixels. The number of Macro Blocks per frame depends on the resolution of the video to be processed.

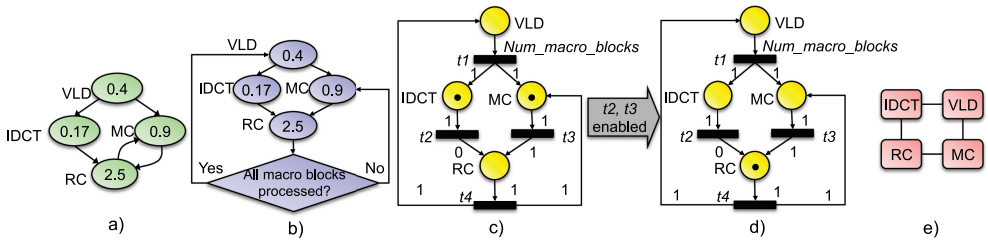

Fig. 2. MPEG-4 Layer 2 application represented as a DFG (a), a CDFG (b), a Petri Net (c and d) and a DFD (e).

A single instance of the MPEG-4 Layer 2 decoder processes one single frame in the video, and it is composed by four tasks (that can concurrently work on different Macro Blocks), namely, Variable Length Decoder (*VLD*), Inverse Discrete Cosine Transform (*IDCT*), Motion Compensation (*MC*) and Reconstruction (*RC*) of the resulting image. These tasks execute in the following order.

—Tasks *IDCT* and *MC* are executed after *VLD* (see Figure 2(a), which will be explained in detail in Section 3).
—Tasks *MC* and *RC* process all the macro blocks contained in that frame in a pipelined fashion (see also Figure 2(a)). In this pipeline, task *RC* executes after *MC*; hence, for the processing of the first block, the pipeline is "filled" during the first execution of task *RC*. The remaining executions of these tasks occur in parallel.

Thus, while there is no restriction on whether the tasks *IDCT* and *MC* must run in sequence or in parallel, tasks *MC* and *RC* constitute a pipeline. They could certainly be executed alternatively, but a considerably more efficient way to run them is to make them coexist in time. This application is executed as many times as necessary in order to process all the frames of an MPEG-4 video.

2.4. Representation Model Requirements

In order to efficiently map and schedule applications, such as the MPEG-4 Layer 2 case study, the aforementioned information about the interdependencies among the tasks during the execution must be taken into account. As a consequence, the application should be specified by means of a proper representation model that is able to capture all of them at the same time. In particular, the representation model should be able to capture:

(1) the *communication requirements* among the tasks that exchange data or control information during the execution;
(2) the *execution flow dependencies* (more usually known as *data dependencies*) among two tasks, that is, a situation in which one or more tasks cannot start their execution before another one completes its own computation. (In the MPEG-4 Layer 2 example, a sequential execution is required between *VLD* and *IDCT*);
(3) the *pipelined behavior information*, that is, a situation where two ore more tasks could be executed in a pipeline fashion. In the MPEG-4 Layer 2 application, this behavior occurs between *MC* and *RC*. It is important to remark that the pipelined behavior does not imply that an actual pipeline must exist in the final architecture, as this choice is up to the design tool.

In the next section, we analyze the existing representation models and we show that no one of them is able to fulfill all these requirements at the same time. This analysis motivates the definition of the new TCDFD formalism that we introduce in Section 4.

3. STATE-OF-THE-ART

Nowadays, researchers are seeking solutions to system level design for embedded systems by investigating new design languages, hardware/software specification environments, and tools. Projects such as Ptolemy [Lee et al.; Eker et al. 2003], Rosetta [Alexander and Kong 2001], and SystemC [Benini et al. 2003] seek system-level specification capabilities that can drive software compilation and hardware synthesis. Although these approaches differ in the scope of their objectives, they all share the common goal of raising the level of abstraction required to design and integrate hardware and software components. Unfortunately, these hybrid computational models are still immature, generally treating FPGAs as computational accelerators that are invoked passively as subroutines. In this context, the use of these approaches for the system level modelling targeted in this work is still too unpractical since they lead to unacceptable performance losses.

The remainder of this section provides an overview of the most relevant existing task representation models available in the literature: Data Flow Graphs (DFGs) [Kavi et al. 1986], Control and Data Flow Graphs (CDFGs) [Zaretsky et al. 2005], Data Flow Diagrams (DFDs) [Bruza and van der Weide 1993] and Petri Nets [Zurawski and Zhou 1994], along with the mapping and scheduling techniques proposed to target these so-represented applications.

3.1. Data Flow Graphs

Figure 2(a) shows the MPEG-4 Layer 2 application represented by means of a Data Flow Graph (DFG). As the figure shows, a DFG is composed of a set of nodes representing computational tasks and a set of directed edges that connect them. Each edge represents a data dependency between them that indicates a precedence constraint existing in the execution of the two involved tasks.

A wide range of algorithms have been proposed in the literature targeting the scheduling problem of DFGs in reconfigurable systems. Many of them are Integer Linear Programming (ILP) formulations [Ghiasi et al. 2004; Cordone et al. 2009], which provide a mathematical formalization for this problem, and are suitable only for static systems. For dynamic scenarios, totally or partially runtime approaches have also been proposed [Noguera and Badía 2004; Haubelt et al. 2005; Wildermann et al. 2011; Resano et al. 2005; Clemente et al. 2011b]. On the one hand, Noguera and Badía [2004], propose a hardware microarchitecture to deal with DFGs applying a list-based scheduling heuristic; Haubelt et al. [2005] present a slack-based list scheduler for time-multiplexed architectures; and Wildermann et al. [2011] propose a design space exploration for reconfigurable embedded systems. These three approaches are purely design-time methodologies. On the other hand, Resano et al. [2005] and Clemente et al. [2011b] propose hybrid design-time/runtime schedulers that exploit task reuse and prefetch in order to reduce the impact of the dynamic reconfigurations, and combine it with a smart task replacement technique.

In addition, the task scheduling problem for DFGs has also been studied in combination with the mapping of the tasks in the available reconfigurable resources [Bender 1996; Ferrandi et al. 2010].

However, the DFG task representation model is not able to specify that two or more tasks should coexist in time on the device for a given period of time. Hence, for the MPEG-4 Layer 2 application, the pipeline existing Tasks *MC* and *RC* cannot be represented with a DFG. Instead, the two arrows connecting *MC* and *RC* represent that these tasks are executed alternatively. A conventional scheduler would use this information in an inefficient way, since it has no knowledge that these two tasks can coexist in order to greatly reduce the execution time of the application.

3.2. Control and Data Flow Graphs

A Control and Data Flow Graph (CDFG) is an extension of a DFG that includes information about the control dependencies among the tasks, which allow controlling the flow of the application via multiway branch. These control dependencies are decisions made based on a condition and they are represented by means of a new type of nodes named *decision nodes*.

Task scheduling for CDFGs on reconfigurable systems has also been studied in the literature. Two interesting discussions about this topic can be found in Anellal and Kaminska [1993] and Memik et al. [2003]. On the one hand, Anellal and Kaminska [1993] proposes an approach to automatically generate a CDFG from a VHDL code, as well as a scheduling algorithm for the generated task graph. On the other hand, Memik et al. [2003] discuss in depth the problem of temporal resources sharing of tasks during the scheduling of CDFGs for FPGAs.

Figure 2(b) shows the MPEG-4 Layer 2 application represented by means of a CDFG, which sole decision node is depicted with a rhombus-shaped box. During the processing of a frame in a video, this node triggers the execution of the right computational node (*VLD* or *MC*) after the completion of *RC* depending on whether all the macro blocks belonging to that frame have already been processed or not. In this case, this CDFG is equivalent to several consecutive executions of the DFG depicted in Figure 2(a). However, as DFGs, CDFGs also lack the information about if two or more tasks constitute a pipeline (in this case, Tasks *MC* and *RC*).

3.3. Petri Nets

Petri nets are a model that can also be used to represent applications in embedded systems. Figures 2(c) and 2(d) show the MPEG-4 Layer 2 application represented by means of a Petri net. It contains three types of objects: *places*, depicted in the figure as circles; *transitions*, depicted as black rectangles; and directed arcs weighed with a non-negative integer value. As in DFGs, the places represent computational tasks and the edges, data dependencies among them. Transitions are unique in Petri nets, and represent synchronization points that trigger the execution of the tasks. The synchronisation is enforced by means of tokens, which are moved from one place to another when a transition triggers. A practical example is shown in where Transitions *t2* and *t3* are enabled; therefore 1 token is deleted from *IDCT* and *MC* and deposited in *RC*.

Temporal scheduling of Petri nets on embedded systems has also been studied in the literature [Zhang and Wu 2009; Eskinazi et al. 2005]. However, similarly as the discussed mapping and scheduling algorithms targeting DFGs and CDFGs, the approaches designed for Petri nets are unable to target applications with pipelines or coexisting tasks.

3.4. Data Flow Diagrams

Figure 2(e) shows the MPEG-4 Layer 2 application represented by means of a Data Flow Diagram (DFD). As the figure shows, a DFD is composed of a set of tasks representing computational tasks and undirected edges among them, representing communication links. Without any further timing information about the tasks, all of them are assumed to coexist throughout the lifetime of the application. DFDs are unable to capture data dependencies among tasks, for instance between *VLD* and *IDCT* in the MPEG-4 Layer 2 application. Hence this makes this model inaccurate to represent this application.

Without any data dependencies among its tasks, the task scheduling problem does not arise when dealing with DFDs. However, task mapping for DFDs has been extensively studied in the literature for complex SoCs. Three relevant related works in the context of reconfigurable systems are Murali et al. [2006b], Hansson [2005], and

Beretta et al. [2011b]. The approaches presented in these first two works aim at minimizing the area requirements and area consumption in the deployment of the tasks of a sole DFD in the target SoC.

Finally, Beretta et al. [2011b] extend the concept of mapping a sole DFD on reconfigurable systems by dealing with several DFDs that are statically defined. However, this approach is unable to exploit any information about the order in which these DFDs will be executed at run time. Hence, unlike [Murali et al. 2006b] and [Hansson 2005], this work considers the possibility of dynamically loading the tasks of the DFD that is mapped each time. Hence this approach also aims at reducing the time penalty incurred between the mapping of two consecutive DFDs.

Although Beretta et al. [2011b] are not able to exploit all the information contained in a TCDFD about the order of execution of the tasks of the applications, it shares the ability of dealing with several DFDs on reconfigurable systems with the proposed approach. Hence, this is the closest compatible work with respect to ours, and it will be used as a reference to compare our work with.

4. TEMPORAL CONSTRAINED DATA FLOW DIAGRAMS

TCDFDs have been defined in order to overcome the limitations existing in the previously discussed task representation models. TCDFDs are visually similar to a *sequence diagram* of UML [Li and Ruan 2011], a well-known model from the software engineering community, which represents the flow of execution of a set of software modules. However, TCDFDs have a different semantics, as they are designed to enhance DFDs by adding the temporal constraints among the tasks that are present in the other representation models discussed in Section 3. Thus, if this information is used properly while taking into account the reconfiguration overhead, a much more efficient usage of the available FPGA resources can be achieved, while reducing the effect of the dynamic reconfigurations.

This section introduces a formal description of the Temporal Constrained Data Flow Diagrams (TCDFDs) and compares them with an equivalent DFG and DFD. For this purpose, we use again the MPEG-4 Layer 2 application as case study.

A TCDFD is represented as a list (T, D, CL), where T is a set of tasks, D is a set of data dependencies between two tasks in T and CL is a set of communication links that exist between two of them. Each task $t_i \in T$ is represented as a pair (s_i, L_i), where s_i is its size in terms of FPGA slices (even though it is possible to specify other resource requirements, such as multipliers, BRAMs, etc.) and L_i is the set of disjoint lifetimes associated with t_i. Each lifetime $j \in L_i$ is, in turn, represented as a pair (b_j, e_j), which are its beginning and finishing instants of time, respectively. Each data dependency in D is represented as a pair (t_i, t_j), indicating that t_j must be executed after the completion of t_i. Finally, each communication link $z \in CL$ between two tasks is represented as a list $(t_k, t_l, z_{start}, z_{finish}, bw)$, where t_k and t_l are the tasks involved in the communication; z_{start} and z_{finish} are the starting and finishing instants of time between which that communication link exists, and bw is the bandwidth, or amount of information that is interchanged between t_k and t_l per time unit. Note that multiple communication links between the same couple of tasks may exist if their communication requirements vary during the execution.

Figure 3 shows the MPEG-4 Layer 2 application represented by means a TCDFD (along with its deadline, which is 8 ms in this case). The figure represents the lifetimes of a task T_i by means of vertical lines delimited by their beginning and ending instants of time; b_i and e_i. The grey non-vertical lines denote data dependencies between the tasks that are relevant for mapping and scheduling purposes. In Figure 3, for instance,

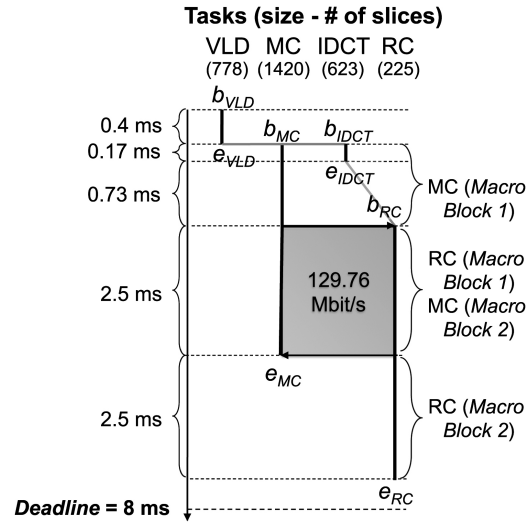

Fig. 3. MPEG-4 Layer 2 application represented as a TCDFD.

a dependency exists between Tasks *VLD* and *MC*, between Tasks *VLD* and *IDCT* and between Tasks *IDCT* and *RC* (see also the DFG in Figure 2(a)).

In addition, the TCDFD in Figure 3 also represents the pipeline between Tasks *MC* and *RC*. In this case, the figure shows the interaction between them when the pipeline completely processes only the first two macro blocks of the frame. Thus, task *MC* runs after *VLD* for $0.17 + 0.73 = 0.9$ ms in order to process *Macro Block 1*. This corresponds to the "filling" of the pipeline. Then, task *RC* is executed in order to finish processing *Macro Block 1*, in parallel with task *MC*, which runs again in order to process *Macro Block 2* (for 2.5 ms). Finally, for another 2.5 ms task *RC* finishes the processing of *Macro Block 2* (this corresponds of the "emptying" of the pipeline). Hence, we can observe that during 2.5 ms, Tasks *MC* and *RC* must coexist in time. On the other hand, the bandwidth of this communication highly depends on the resolution of the frames and the quality of the video (in terms of number of frames/second and bits/pixel). In the example of the figure, we assume that the video to be processed is encoded using the Common Intermediate Format (CIF, 352×288 pixels) [(ITU) 1993], and it displays images at a rate of 40 frames per second. Assuming that a pixel is represented with 32 bits, the bandwidth required in this communication is 129.76 Mbit/s, as indicated in Figure 3.

Thus, the MPEG-4 Layer 2 application is represented as a list (T, D, CL), where:

$$T = \{VLD, IDCT, MC, RC\}, \tag{2}$$

$$D = \{(VLD, MC), (VLD, IDCT), (IDCT, RC)\}, \tag{3}$$

and

$$\begin{aligned} VLD &= (778\ slices, (0, 0.4)) \\ MC &= (1420\ slices, (0.4, 3.8)) \\ IDCT &= (623\ slices, (0.4, 0.57)) \\ RC &= (225\ slices, (1.3, 6.3)). \end{aligned} \tag{4}$$

On the other hand, CL only contains a communication link

$$CL = (MC, RC, 1.3, 3.8, 129.76). \tag{5}$$

This link is indicated in the figure by means of the shaded area.

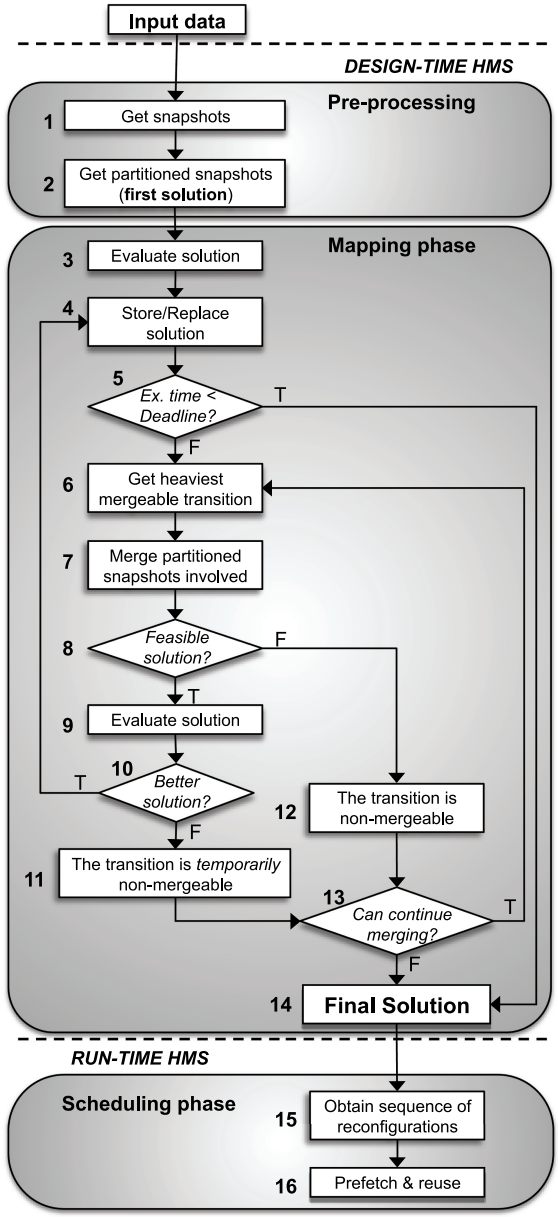

Fig. 4. Flowchart of the proposed HMS algorithm.

5. HYBRID MAPPER-SCHEDULER

The proposed Hybrid Mapper-Scheduler (HMS) receives as input a TCDFD and its deadline, and schedules and maps this TCDFD on the target architecture, aiming at reducing the reconfiguration overhead and using as few hardware resources as possible, while meeting the application deadline and the communication constraints. For this purpose, the algorithm also receives as input the interregion bandwidth provided by the target architecture.

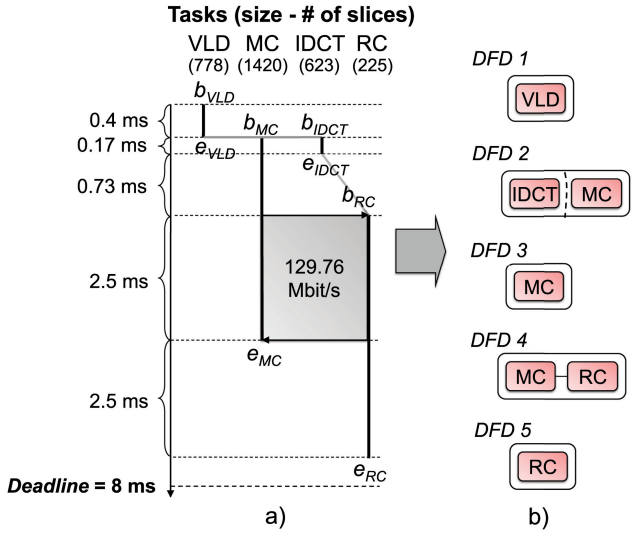

Fig. 5. Snapshot extraction of the TCDFD representing the MPEG-4 Layer 2 applicacion (a), and their corresponding DFDs (b).

As Figure 4 shows, the algorithm consists of three different stages: a *preprocessing* of the input data, a *mapping phase* and a *scheduling phase*. First of all, the pre-processing step extracts information from the input TCDFD about its *snapshots* and *partitioned snapshots* that will be used in the next two phases. Then, the mapping phase is performed in order to obtain a solution. Finally, in the scheduling phase both *bitstream prefetch* and *reuse* scheduling techniques based on partial reconfiguration are applied on this solution in order to further reduce the impact of the reconfiguration overhead. It is important to note that these two techniques have been widely explored in the literature for their use in reconfigurable systems [Resano et al. 2005; Clemente et al. 2011b; Li 2002]. Hence, we are not pioneers of their utilization on reconfigurable systems, but of applying these ideas targeting TCDFDs on complex reconfigurable systems.

Finally, it is important to point out that the preprocessing and mapping phase steps of the proposed HMS are applied at design time, whereas the scheduling phase is carried out at run time. The reason of applying the latter at run time is to make the scheduler adaptable to the execution of a sequence of TCDFDs unknown at design time. This is a common assumption in embedded systems, since even though the set of applications that are to be executed in the system is well-known at design time, the actual sequence of execution of these applications may be known only at run time. In this context, it is advantageous to make the scheduling decisions at run time, since it allows to greatly reduce the impact of the reconfiguration overhead of these bitstreams, which is critical for the system performance. A number of papers in the literature have previously discussed this point [Resano et al. 2005; Clemente et al. 2011b] and succesfully applied runtime scheduling on reconfigurable systems.

5.1. Preprocessing

First of all, the algorithm divides the application in a set of snapshots, that is, intervals of time when the co-existing tasks are always the same (Figure 4, Step 1). For this purpose, it selects all the beginning and finishing instants of time of all the tasks in the TCDFD $\{b_{ti}, e_{ti}, \forall ti \in T\}$, sorts them decreasingly, and creates the snapshots by selecting two consecutive items of this sorted list. Thus, for the MPEG-4 Layer 2 case study

(Figure 5(a)), this sorted list is: {0, 0.4, 0.57, 1.3, 3.8, 6.3}, and the snapshots $Sn1 - Sn5$ are: $(0 - 0.4)$, $(0.4 - 0.57)$, $(0.57 - 1.3)$, $(1.3 - 3.8)$ and $(3.8 - 6.3)$, respectively. Each snapshot is then translated into a traditional DFD that comprises the tasks in the given interval of time and their communication links. The weight of each node in the DFD represents the size of the task, whereas the weight in each communication link represents the bandwidth (measured in Mbit/s) of the involved communication. Figure 5(b) shows an example of the DFDs that are obtained from the snapshots extracted in Figure 5(a). Thus, *DFD 1* contains the *VLD* task, *DFD 2* contains *IDCT* and *MC*, *DFD 3* contains *MC*, *DFD 4* contains *MC* and *RC*, and *DFD 5* contains *RC*. Note that the two tasks in *DFD 4* are connected because a communication link exists between both of them in the initial TCDFD. However, this does not occur between tasks *IDCT* and *MC*, which belong to *DFD 2*.

A communication link between two tasks in these DFDs is said to be *critical* if the bandwidth required in this communication is greater than the inter-region bandwidth that the target architecture guarantees to provide (in the following we will refer to this bandwidth as *threshold bandwidth*). Hence two tasks connected with a critical communication constraint must be mapped together in the same RU; otherwise the communication constraints are not met and the specified system cannot be implemented on the device. For our target architecture, this applies to communications whose bandwidth is greater than 100 Mbit/s, since this is the bandwidth provided by its inter-region NoC. For the case of the MPEG-4 Layer 2 application, the communication link between Tasks *MC* and *RC* is 129.76 Mbit/s. Hence these two tasks must be mapped in the same RU of our architecture in order to meet this communication constraint. In any case, the threshold bandwidth is an input parameter to our mapper-scheduler.

We also assume that there are enough RUs in our system to fit at the same time all the tasks belonging to the snapshot that uses the greatest amount of resources. On the one hand, this assumption is sufficient to guarantee the feasibility of the mapping/scheduling problem, and that the FPGA provides enough area to map a significant part of the application, that is, one of its snapshots. On the other hand, this hypothesis guarantees that two cores that communicate in the same time period can also be configured on the device at the same time.

Once the snapshots have been obtained, the algorithm partitions the DFDs of these snapshots into several groups of tasks or *islands* of tasks (each one of them will eventually be mapped onto a different RU in the system). Thus, the resulting set of islands of the snapshot that has been partitioned constitute a *partitioned snapshot* (Step 2). This set of partitioned snapshots constitutes a first solution, which will be iteratively refined in the rest of the HMS algorithm (Steps 3–13) in order to obtain the final solution (Step 14).

It is important to note that each island will eventually correspond to a bitstream. However, the following two conditions must be met.

(1) The number of islands generated must not exceed the number of available RUs in the system, in order to allow all the islands of the same partitioned snapshot to be mapped on the device at the same time.
(2) The nodes connected through critical communications must be mapped in the same RU (i.e., they must belong to the same island in the generated partitioned snapshot), which is a sufficient condition to maximize the bandwidth between them.

In order to generate the partitioned snapshots while meeting these conditions, we use an external partitioner named *CHACO* [Hendrickson and Leland 1994]. This tool receives as input the DFD associated to a snapshot and the number of islands to generate, and it returns as output a partitioned snapshot of that DFD into the specified number of islands. CHACO always tries to balance the partitioned snapshot that it

generates, so its primary objective is that the sums of the values of the nodes assigned to the same island are as similar as possible. CHACO also offers the possibility of specifying constraints to map in the same islands the tasks connected by critical communication links, which ensures that the communication constraints are always met in the first set of partitioned snapshots that are generated at the end of this step. In our case, HMS uses CHACO to obtain as few islands as possible when generating a partitioned snapshot, starting from the minimum possible number of islands that can accommodate the tasks, and iteratively increasing it until a feasible mapping is found (this is done in Steps 2 and 7 in Figure 4). This strategy leads to better results with respect to spreading the tasks on all the available islands on the device, as shown in the experimental results of Section 6.

After HMS carries out these two steps, the following phase of the HMS algorithm maps the generated partitioned snapshots sequentially, by taking into account the reconfiguration overhead introduced in the transitions between two consecutive mappings.

5.2. Mapping Phase

Once the preprocessing has been performed, the algorithm evaluates the quality of the first solution obtained (Figure 4, Step 3) by calculating its execution time (including the reconfiguration overhead), and stores it (Step 4). In order to obtain the execution time of the solution, in this step the algorithm invokes the module that performs the scheduling phase (further explained in Subsection 5.3), which applies the prefetch and replacement techniques that promote the reuse of the bitstreams to minimize the impact of the dynamic partial reconfigurations.

The same metric (total execution time) is also used in Step 5 to guide the scheduling process: HMS continues iterating until the execution time of the current solution is lower than its deadline. Thus, if this condition is true (Step 5), the current solution is selected as the final one that the mapping phase returns (Step 14). Otherwise, the algorithm applies an iterative process to reduce its execution time (Steps 6-13). In each iteration, the algorithm selects the transition between two consecutive partitioned snapshots that generates the greatest reconfiguration overhead (Step 6) and tries to eliminate it by merging both of them (Step 7). If this process succeeds, the two partitioned snapshots are said to be *mergeable*.

Initially all the transitions between two consecutive partitioned snapshots are *potentially mergeable*; that is, they are not yet proven to be non-mergeable. However, during the execution of the algorithm, if HMS is able to merge them, then it indeed considers them as mergeable. A merging process between two mergeable partitioned snapshots returns as result two *compatible* partitioned snapshots. Let $I(t_i)$ indicate the island to which the task t_i belongs, then two consecutive partitioned snapshots PS_1 and PS_2 are compatible if the following conditions hold true:

$$\forall t_i, t_j \in Tasks_{PS1} \cup Tasks_{PS2}, [[I(t_i) = I(t_j)]_{PS1} \leftrightarrow [I(t_i) = I(t_j)]_{PS2}], \tag{6}$$

$$\forall t_i, t_j \in Tasks_{PS1} \cap Tasks_{PS2}, [[link(t_i, t_j)]_{PS1} \leftrightarrow [link(t_i, t_j)]_{PS2}]. \tag{7}$$

On the one hand, Equation (6) means that, for all the tasks in partitioned snapshots PS_1 and PS_2, if tasks t_i and t_j belong to the same island in PS_1, they also belong to the same island in PS_2 and vice-versa. Hence, if PS_2 and PS_1 are compatible, no extra reconfiguration overhead is introduced to map an island of PS_2, unless that island only contains tasks that do not appear in PS_1, since in that case the island is not loaded for the execution of PS_1. On the other hand, the Equation (7) means that, for

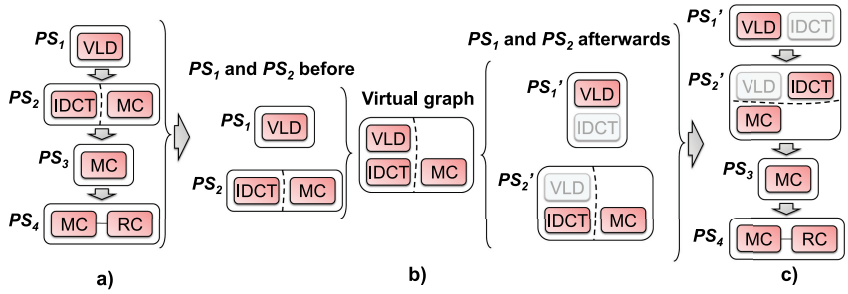

Fig. 6. Merging process of PS_1 and PS_2 in MPEG-4 Layer 2. The dashed lines divide different islands from the same partitioned snapshot PS_i

all the common tasks in PS_1 and PS_2, if tasks t_i and t_j are connected by means of a communication link in PS_1, they are also connected in PS_2, and vice-versa.

Let us now describe the operation of the merging process. At the beginning of the algorithm, each partitioned snapshot is said to belong to a different *compatibility class*. A compatibility class is a set of partitioned snapshots that are all compatible with each other. This means that, as soon as two consecutive partitioned snapshots are merged (e.g., PS_i and PS_{i+1}), then HMS will make them compatible and, therefore, they will belong to the same compatibility class. From the point of view of HMS, this means that any future merging process that involves either PS_i or PS_{i+1} will actually involve both of them and the resulting partitioned snapshots will belong to the same compatibility class again. Thus, if after merging PS_i and PS_{i+1}, the algorithm decides to merge PS_{i+1} and PS_{i+2}, it will have to merge PS_i, PS_{i+1} and PS_{i+2}. This process is applied in this way independently of the size of the compatibility classes involved.

In other words, if HMS designates PS_i and PS_{i+1} to be merged, it will actually merge all the partitioned snapshots that are compatible either with PS_i or with PS_{i+1}; that is,

$$PS\_merged = \{PS_j \in PS : Comp(PS_j, PS_i) \vee Comp(PS_j, PS_{i+1})\}, \tag{8}$$

where PS indicates the set of partitioned snapshots in the application, and $Comp(PS_j, PS_i)$, indicates that the partitioned snapshots PS_j and PS_i are compatible. Note that in this equation, PS_i and PS_{i+1} are assumed to be merged.

The merging operation is described by means of the example in Figure 6, which shows the merging process between PS_1 and PS_2 of the MPEG-4 Layer 2 application. For the sake of simplicity, let us assume that the compatibility classes of PS_1 and PS_2 only contain the partitioned snapshots PS_1 and PS_2, respectively. Hence, this merging process just involves PS_1 and PS_2, not other partitioned snapshots. Figure 6(a) shows an example of initial partitioning for the MPEG-4 Layer 2, Figure 6(b) depicts the merging process of its two first partitioned snapshots and Figure 6(c) shows the final ones.

In the first step of the merging process, a *virtual graph* (VG) is created such that the following two conditions hold true:

$$\forall t_i, t_j \in Tasks, [t_i \in PS_1 \vee t_i \in PS_2 \leftrightarrow t_i \in VG], \tag{9}$$

$$\forall t_i, t_j \in Tasks, \ [[clink(t_i, t_j)]_{P1} \vee [clink(t_i, t_j)]_{PS2} \leftrightarrow [clink(t_i, t_j)]_{VG}]. \tag{10}$$

Equation (9) specifies that, if a given task t_i of the application exists either in PS_1 or in PS_2, it will be included in VG. On the other hand, according to Equation (10),

if tasks t_i and t_j are connected by means of a communication link in either PS_1 or in PS_2, they will also be connected in VG.

Then, the algorithm executes CHACO in order to partition VG, and it finally selects the islands from there to generate the new partitioned snapshots according to the following formula:

$$\forall s_i \in Snapshots, \forall i_j \in Islands_{VG},\ [\exists t_z \in Tasks(i_j) : used(t_z, s_i) \rightarrow i_j \in PS'_i]. \quad (11)$$

This expression means that, for each snapshot (s_i) and for each island (i_j) of the partitioned virtual graph, if i_j contains at least one task (t_z) that is used in the snapshot s_i (according to the initial specification of the application), it will be selected to belong to the new partitioned snapshot associated with that snapshot (PS'_i). In this example, the island that contains the tasks *VLD* and *IDCT* is selected to belong to PS'_1 because *VLD* belongs to S_1, even though *IDCT* does not (according to the initial specification of Figure 2). Note that, in this case, *IDCT* appears shaded, indicating that it is loaded in the system but not used in that snapshot. The same situation, occurs for *VLD* with the island that contains the tasks *VLD* and *IDCT* for PS'_2.

Also, note that the constraints depicted in Equations (6)–(11) ensure that all the tasks that had to be mapped together in the initial partitioned snapshots because of a critical communication link between them, will be assigned again to the same island at the end of the merging process. The reason is that we explicitly ensure that the involved communication link is finally included in both merged (i.e., compatible) partitioned snapshots, as well during all the merging process.

Once the involved partitioned snapshots have been merged, HMS must check whether the new solution is feasible, that is, if the sums of the sizes of the tasks assigned to the same partitioned snapshot do not exceed the size of a single RU (Step 8). If the solution is not feasible, that transition is marked as *non-mergeable* (Step 12), and the algorithm will not try to perform this merging process again. Then, the algorithm checks if it can continue merging partitioned snapshots (Step 13), by determining if there is at least one pair of mergeable partitioned snapshots. If so, at least another iteration of the algorithm is performed, since the solution still does not meet the given temporal constraints. Otherwise, the best solution found is marked as the final solution (Step 14).

On the other hand, if the solution is feasible, it is evaluated (Step 9). If it is better than the best one found so far (Step 10), it is replaced (Step 4), and the algorithm checks if it meets the temporal constraints (Step 5) and continues again as explained earlier. Otherwise, the transition between the two partitioned snapshots is marked as *temporarily non-mergeable* (Step 11), meaning that the algorithm will not try to merge it again, unless one of the two partitioned snapshots involved in the transition (or one of the compatibility classes they belong to) is later part of a successful merging process. For instance, if the transition between PS_1 and PS_2 has been marked as temporarily non-mergeable, and HMS merges PS_2 and PS_3, the transition between PS_1 and PS_2 becomes potentially mergeable again. This process is required in order not to prematurely rule out moves that may improve the solution later during the optimization.

In a nutshell, the mapping phase of HMS applies a set of optimizations that correspond to the merging operations between consecutive partitioned snapshots. The more partitioned snapshots are involved, the more global that optimization is. Each merging process involves the reduction of the reconfiguration overhead between the two involved partitioned snapshots, although more hardware resources are used (for instance, in Figure 6(c), PS'_1 maps the task *IDCT*, whereas PS_1 does not). Hence, the more globally optimized a solution is, the more hardware resources it uses. For this reason, HMS

stops iterating as soon as the temporal constraints are met. Otherwise, the solution would still meet the same constraints, however using more hardware resources.

5.3. Scheduling Phase

Once the mapping phase has been carried out, in this phase HMS schedules at run time the execution of the bitstreams that have just been generated, but taking into account the following three conditions.

(1) There is only one interface to perform dynamic reconfiguration, therefore the bitstreams must be configured sequentially.
(2) If two bitstreams belong to the same partitioned snapshot PS, then they must be executed in parallel.
(3) If two bitstreams belong to consecutive partitioned snapshots PS_1 and PS_2, all the bitstreams that belong to PS_1 must be executed before the ones that belong to PS_2.

For this purpose, the scheduler firstly sorts the bitstreams in a sequence of reconfigurations (Step 15) in such a way that

$$\forall b_1, b_2 \in Bitstreams,\ [PS(b_1) < PS(b_2) \rightarrow pos\_seq(b_1) < pos\_seq(b_2)]. \quad (12)$$

This expression means that, if the partitioned snapshot to which b_1 belongs ($PS(b_1)$) is executed before the partitioned snapshot to which b_2 belongs ($PS(b_2)$), and since two partitioned snapshots cannot be executed in parallel, then b_1 is placed before b_2 in the sequence of reconfigurations.

Once this sequence has been obtained, HMS applies both bitstream prefetch and reuse techniques to further reduce the impact of their reconfigurations (Figure 4, Step 16). For this purpose, and in order to simplify the complexity of the scheduler, it only considers some specific time instants following an event-triggered approach. In other words, when certain events occur the scheduler makes the proper decisions.

Four different events trigger its execution: *new_app*, which is generated when the information of a new application is received; *end_of_reconfiguration*, which is generated when a new bitstream has been loaded, *reused_bitstream*, which is generated when the scheduler detects that a bitstream can be reused since it was already loaded in a previous execution; and *end_of_execution*, which is generated when all the bitstreams belonging to the same partitioned snapshot finish their execution.

Each time an event is captured, the scheduler triggers the reconfiguration of the following bitstream in the sequence of reconfigurations. Since only one reconfiguration can be carried out at a time (because, nowadays, reconfigurable devices feature only one reconfiguration circuitry), the scheduler applies bitstream prefetch in order to load the configurations belonging to a partitioned snapshot in advance, while a previous partitioned snapshot may still be executing. Thus, the reconfiguration overhead between two consecutive partitioned snapshots is further reduced, even in the case they are not compatible.

Algorithm 1 outlines this process. When a new application arrives and the reconfiguration circuitry is idle (Lines 1–3), the system attempts to load the first reconfiguration in the reconfiguration sequence. In case of the *end_of_reconfiguration* or *reused_bitstream* events (associated to a bitstream B), the system checks if all the bitstreams that belong to the same partitioned snapshot PS as B ($PartSshot(B)$ in the figure) have already been loaded in the system, and if $PartSshot(B) - 1$ has finished its execution (Line 4). If so, the scheduler executes $PartSshot(B)$ (Line 5), triggering the execution of all its bitstreams in parallel. In addition, if the reconfiguration circuitry is idle, it tries to load the next bitstream in the reconfiguration sequence (Lines 7–9).

Finally, for the *end_of_execution* event (associated to a partitioned snapshot PS), the scheduler checks again if the reconfiguration circuitry is idle in order to perform a

ALGORITHM 1: The proposed runtime scheduling phase of HMS

```
// RC: Reconfiguration circuitry
CASE event IS
new_app:
 1: if (RC == idle) then
 2:    look_for_reconfiguration (&rec_sequence);
 3: end if;

end_of_reconfiguration or reused_bitstream (bitstream B):
 4: if (all_bitstreams_loaded (PartSshot (B)) and finished_execution (PartSshot (B)-1)) then
 5:    start_execution (PartSshot (B));
 6: end if;
 7: if (RC == idle) then
 8:    look_for_reconfiguration (& rec_sequence);
 9: end if;

end_of_execution (Partitioned Snapshot PS):
10: if (RC == idle) then
11:    look_for_reconfiguration (&rec_sequence);
12: end if;
13: if (all_bitstreams_loaded (PS + 1)) then
14:    start_execution (PS + 1);
15: end if;
end CASE;
```

new reconfiguration (Lines 10–12). Then, if all the bitstreams belonging to partitioned snapshot $PS+1$ have already been loaded, the scheduler triggers their execution (Lines 13–15).

Each time the function *look_for_reconfiguration()* is invoked, the scheduler must decide in which RU to load the next bitstream in the sequence. For this purpose, it firstly checks if the tasks belonging to that bitstream have already been loaded in any of the available RUs. If so, that bitstream is reused, and hence no reconfiguration overhead is generated. Otherwise, a replacement victim is selected according to the *Longest Forward Distance* (LFD) replacement policy [Belady 1966]. We have chosen this technique because it has been proven to be the one that guarantees the optimal reuse rate, which has a direct impact on the reconfiguration overhead reduction. However, any other runtime replacement technique could also be applied instead, such as the one proposed in Clemente et al. [2011b], which has been proved to be very efficient in highly dynamic scenarios.

The benefits of these optimizations can be seen in Figure 7 by means of an example. It shows the execution of the MPEG-4 Layer 2 application on a system with 3 RUs following the mapping that is shown in Figure 6(a), and assuming that each bitstream takes 1 ms to be reconfigured.

Figure 7(a) shows an ideal execution with no reconfiguration overhead. In this case we can observe that the lower-bound execution time is 6.3 ms.

Then, Figure 7(b) shows the execution of this application including the reconfigurations of the bitstreams but without any prefetch and reuse optimizations. In this case, the system carries out 6 reconfigurations, and the total execution time is 12.3 ms. However, when the prefetch and reuse techniques of the scheduling phase in HMS are applied (Figure 7(c)), the execution time is reduced to 9 ms. Indeed, in this case some reconfigurations can be partially or totally overlapped with the execution of other bitstreams (the bitstreams with the tasks *IDCT* and *MC-RC*). In addition, note that in

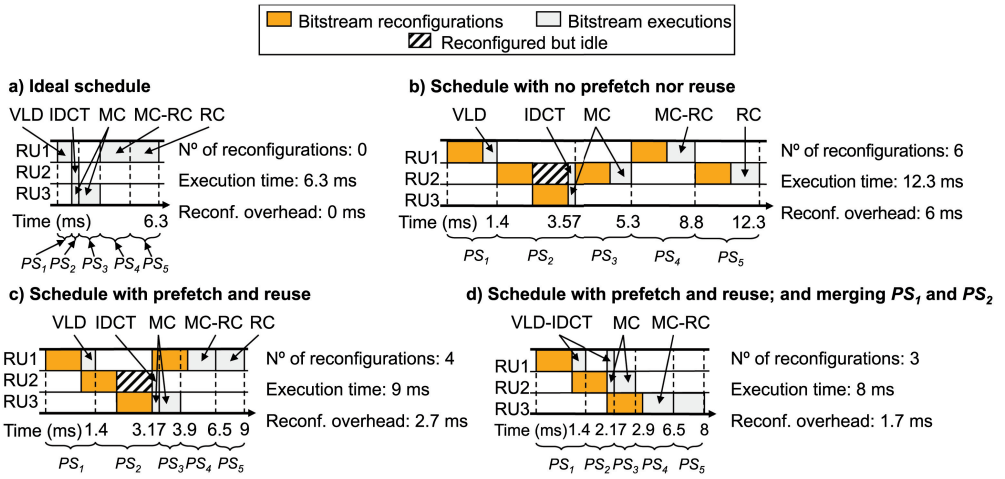

Fig. 7. Example of execution of the MPEG-4 Layer 2 application using our HMS approach.

this case the reconfiguration of the bitstreams that only contain the tasks *MC* (belonging to PS_3) and *RC* (belonging to PS_5) do not generate any overhead, since they can be reused from the previous execution of the same tasks (which also belong to PS_2 and PS_4, respectively).

Finally, Figure 7(d) shows the benefits of applying the optimizations of the mapping and scheduling phases of our HMS algorithm altogether. In this case, we show again the execution of the MPEG-4 Layer 2 application, but this time considering that the merging process between the partitioned snapshots PS_1 and PS_2 depicted in Figure 6 has been executed. As the figure shows, in this case the system only carries out 3 reconfigurations, since a new bitstream *VLD - IDCT* has been created and it is used in PS_1, and reused in PS_2. As a consequence of this, the new execution time is just 8 ms, 1 ms less than the same execution when PS_1 and PS_2 were not merged (Figure 8(d)). In addition, note that the resources consumption does not increase either, since the application is again executed in 3 RUs.

6. EXPERIMENTAL RESULTS

In this section, we evaluate the efficiency of the proposed approach on real reconfigurable platforms, by means of both state-of-the-art multimedia applications and synthetic benchmarks.

6.1. Experimental Setup

As real-world case studies, we have considered the H.264 video coding standard [Roitzsch 2007; Lindroth et al. 2006; Mei-hua et al. 2007] and the MP3 audio decoder [Theelen et al. 2008]; [Taghipour et al. 2008]. The former is a state-of-the-art evolution of the MPEG-4 Layer 2 video coding standard, which is commonly used for the recording, the compression and the distribution of high-definition videos in Blu-ray discs. The codec features a considerably higher computational complexity than the MPEG-4, although from a structural perspective [Roitzsch 2007] it only contains 6 tasks (featuring an average size of 1672 FPGA slices). The MP3 audio decoder, on the other hand, has been chosen because it consists of a considerably higher number of computational tasks (15 tasks with an average size of 343 FPGA slices), thus providing different challenges to the proposed algorithm.

Finally, in order to better analyze how the efficiency of our HMS scales on larger problem sizes, we have also evaluated it with a set of 100 synthetic benchmarks that represent the structure of complex and modular real-world applications that we expect to appear in the upcoming years, and which feature a considerably large structural complexity. These benchmarks are composed of 50 tasks (which can be considered as the execution of 1 application consisting of 50 cores, 2 applications consisting of around 25 cores each one, etc.), each one of which contains from 1 to 5 disjoint lifetimes that span from 0 to 100 ms, in such a way that they all contain at least 10 snapshots. The sizes of the tasks range from 100 to 500 slices. All these parameters are selected by following discrete uniform distributions. Furthermore, for all the snapshots extracted from these synthetic benchmarks, there is a 10% chance for a communication between two tasks to be critical, and the sums of the sizes of all the sets of tasks connected through critical communications is less than the size of a RU. Hence they can always be mapped in the same RU in such a way that their communication constraints are met. This set of parameters has been selected by following the trends of real-world applications. Let us consider the size of the tasks: we observed that applications with more tasks tend to be more specialized, and consequently cores require a lower area. Among the applications we have analyzed, the H.264 includes 6 cores with an average area of 1670 slices, whereas the MP3 includes 15 cores with an average area of 363 slices. As a consequence, we opted for selecting a task area that is closer to the more complex applications, that is, we focused on tasks with an area no larger than 500 slices. Similar considerations led to the determination of the percentage of critical communications, the number of lifetimes, and the number of snapshots.

Even though the proposed tuning is consistent with respect to the trends shown by real applications, we nonetheless did not limit our experimental analysis to a specific set of synthetic benchmarks. In particular, additional experiments were conducted with different number of cores (20, 50 and 100), different core size (maximum 500, 1000 and 2000 slices), and different percentages of critical communications (5%, 10% and 15%), which are not all reported in this manuscript for the sake of simplicity, since we did not observe significant variations in the quality of the final results. In fact, the comparison with other SoA approaches in terms of reconfiguration overhead and number of reconfigurations differ by quantities between 1 and 2% with respect to the results reported in this article.

In our experiments we have compared our HMS with other SoA approaches [Murali et al. 2006a; Beretta et al. 2011b; Clemente et al. 2011b] from the point of view of two metrics: reconfiguration overhead reduction (Section 6.2) and resources saving (Section 6.3). The first one is a conventional SoA mapper that optimizes area usage for a sole DFD through a communication-driven methodology. However it does not take into account the reconfiguration overhead between two consecutive mappings, which clearly leads to suboptimal results. The second one is again a mapper for DFDs, but it also considers reconfiguration overhead, as opposed to other SoA approaches and representation models (see Section 3). However, these two approaches do not take into account the order of execution of the DFDs that correspond to the snapshots of the applications in order to further optimize the execution of the application, contrarily to the presented HMS. In addition, they do not apply any prefetch of the involved configurations either. Finally, the third approach is a conventional scheduler targeting DFGs that aims at minimizing task reconfiguration overhead by means of prefetch and reuse techniques.

The presented HMS also admits comparison with conventional schedulers from the point of view of an additional metric: *total application execution time*. In fact, these approaches feature an important drawback: They do not consider the information regarding potential task co-existency between tasks, hence the resulting schedule

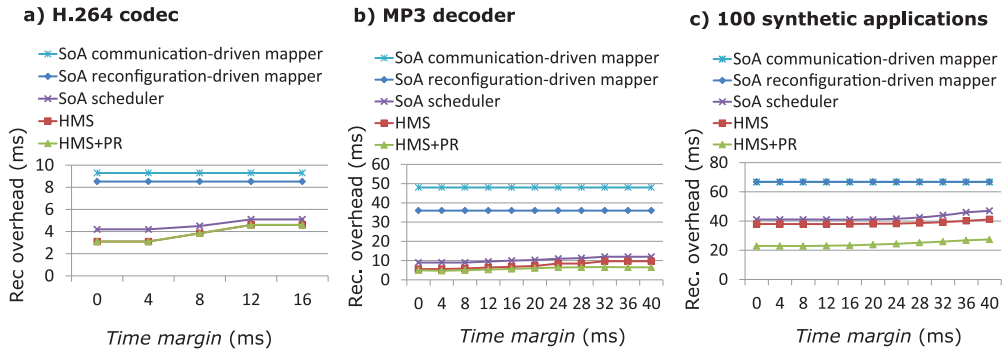

Fig. 8. Reconfiguration overhead of HMS with respect to the mappers in [Murali et al. 2006a] and [Beretta et al. 2011b], and the scheduler in [Clemente et al. 2011b], for different values of the time margin

considerably enlarges the total execution time of the applications. Thus, for instance, according to our measurements, the scheduler in [Clemente et al. 2011b] increases the total execution time of the MP3, H.264 and the synthetic benchmarks by 62%, 37% and 42% respectively, with respect to HMS. Since [Murali et al. 2006a] and [Beretta et al. 2011b] do not lack this problem (as any other conventional mapper), in the next subsection we have decided to compare them using the reconfiguration overhead reduction metric. It is also a more accurate metric that is typically used to evaluate the efficiency of a scheduling approach.

The MP3 case study and the synthetic benchmarks have been evaluated by using a programmable system with 12 RUs implemented in a Xilinx^{TM}XUPV5-LX110T development board, which features a Virtex-5 FPGA. As a consequence, the number of slices per RU is equal to 622, and the reconfiguration of each RU requires approximately 4 ms. On the other hand, for the H.264 application [Lindroth et al. 2006; Mei-hua et al. 2007], we have assumed a target architecture containing 5 RUs of 8200 logic elements, and a reconfiguration overhead of 3.09 ms per RU.

6.2. Reconfiguration Overhead Reduction

Figure 8 shows the reconfiguration overhead generated when using HMS on the evaluated applications with different timing constraints. For that purpose we have used a metric that we have named *time margin*, which is the difference between the optimal execution time of the application and its deadline. In each case we have averaged the single results obtained with different numbers of RUs: from 4 to 12 RUs for the MP3 and the synthetic applications and from 2 to 5 RUs for the H.264 codec, since the latter contains considerably fewer tasks than the other two applications.

As previously mentioned in Section 5.1, during the preprocessing phase, HMS uses the CHACO partitioner [Hendrickson and Leland 1994] to obtain as few islands of tasks as possible when generating the partitioned snapshots. The reason is that this option always performs better than spreading the partitioned snapshots over all the available reconfigurable units (as the results in this section will demonstrate). In the results, we will refer to this optimization as *Partitioning Reduction* or *PR*. In order to evaluate this point, Figure 8 shows how HMS performs when it includes both implementation options: when it spreads the partitioned snapshots (labeled as *HMS* in the figure) and when it does not (labeled as $HMS + PR$).

For the H.264 codec (Figure 8(a)), the SoA mappers generate a constant average reconfiguration overhead of 8.5 and 9.3ms, respectively. These overheads are independent of the value of the time margin of the applications, since these algorithms do not effectively consider any scheduling information.

However, the remaining scheduling approaches clearly outperform these results. On the one hand, the SoA scheduler reduces the average reconfiguration overhead by 4.67 and 3.9ms, respectively, with respect to the SoA mappers. This result may seem very efficient; however, as already discussed in Subsection 6.1, this approach leads to a significant increase in the total execution time of the application (in this case, 37%). For this reason conventional schedulers are completely unpractical to target TCDFDs. On the other hand, both HMS and HMS+PR reduce the reconfiguration overhead by up to 5.4ms with respect to the communication-driven SoA mapper. This means an average reduction of 63.5%. The reason of this speed-up is that the SoA mappers do not exploit the information about the order in which the bitstreams must be mapped, while HMS exploits it to reduce the reconfiguration overhead between two consecutive mappings. In addition, they do not apply any task prefetch to load the reconfigurations in advance. In this case, both *HMS* and *HMS+PR* achieve the same results. However, as the number of tasks in the application grows (Figures 8(b) and 8(c)), the gap in performance between both approaches increases. The reason is that if the application has more tasks, the space solution for our mapping approach is broader, and hence the *HMS+PR* technique has more opportunities to apply the *PR* optimization. This means an additional reduction in the reconfiguration overhead up to 40%. Note also that, for the MP3 case study, the reconfiguration overhead reduction achieved by HMS+PR is significantly higher, up to 89% with respect to the SoA communication-driven mapper, which is a very good result.

In Figure 8 we can also observe that, for both *HMS* and *HMS+PR*, the reconfiguration overhead grows as the value of the time margin increases. The reason is that HMS stops iterating as soon as it finds a solution that meets the given deadline. Note also that, when the time margin is set to 0, the proposed approach still generates solutions with reconfiguration overhead greater than zero. The reason is that the first bitstream of each application can never be reused nor prefetched, since in our experiments we assume that each application is solely executed on the system.

For the synthetic benchmarks, it is also important to mention that our HMS approach always managed to find a feasible mapping for each one of the tested benchmarks. However, that was not possible for the SoA mappers, since they only obtained a feasible solution for the 41% of the cases. The reason is that these mappers do not take into account the criticality of the communications between tasks in order to map them in the same RU and therefore to meet the communication constraints among tasks. This illustrates another important advantage of the proposed approach from the point of view of the feasibility of the generated solutions.

6.3. Resources Saving

Figure 9 shows the Number of RUs used by the *HMS* and *HMS+PR* approaches, comparing them with the SoA mappers for different values of the time margin. For the H.264 codec and the MP3 decoder (Figures 9(a) and 9(b)), both *HMS* and *HMS+PR* save up to 60% and 75% of the available RUs with respect to these approaches. Figure 9(c) shows a similar trend for the synthetic benchmarks. In this case, the RUs saved reach up to 34% and 46% the RUs consumption of the SoA approaches, for both *HMS* and *HMS+PR* respectively, when the time margin is 40 ms.

Furthermore, looking at the results in Figure 9, one can observe that as the value of the time margin increases, the fewer number of RUs are needed. The reason is that, as the value of this parameter increases, the algorithm iterates fewer times. Thus, it returns a solution that may show a degradation in terms of performance with a corresponding area usage reduction, as the HMS iterations (see Section 5.2) aim at reducing the time overhead between two consecutive mappings, at the cost of incurring into additional consumption of RUs. Hence, looking at both Figures 8 and 9, we can

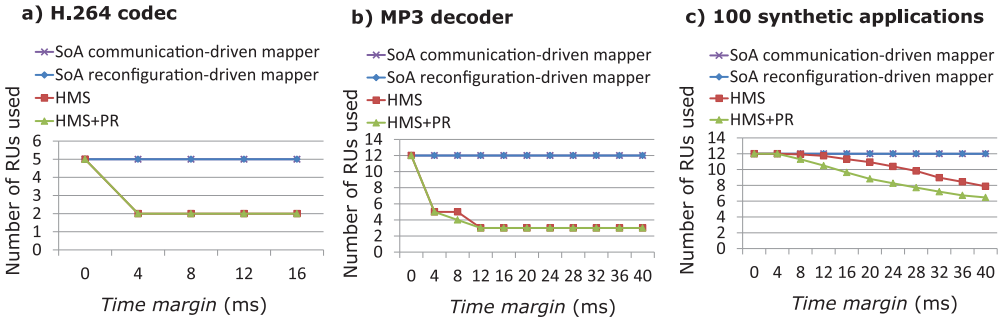

Fig. 9. Number of RUs used by our HMS with respect to the SoA mappers in Murali et al. [2006a] and Beretta et al. [2011b], for different values of the time margin.

observe that all the possible solutions offer different trade-offs between these two optimization objectives. Each one of these can be selected by tuning the value of the time margin. Thus, if this parameter is strict, HMS tries to optimize the performance of the solution, at the cost of using a greater number of RUs, and viceversa.

Finally, note that the RUs savings described in this section can have different impacts on the system performance: Thus, HMS can be used to fit only one application in an embedded system that features an extremely restricted reconfigurable resources area, or in order to maximize the system throughput. Since this is highly dependent on the final users needs, we consider that the management of this is out of the scope of the described methodology.

6.4. Benefits of the Prefetch

This subsection discusses the benefits of the prefetch optimization that is applied during the scheduling phase of *HMS*. For this purpose, we have calculated the reconfiguration overhead and the number of reconfigurations generated when the synthetic benchmarks are executed using the SoA mappers, *HMS* and *HMS+PR* (the MP3 and the H.264 applications show a similar behavior).

For this experiment we have set the time margin to 0, and we have calculated the results when the number of available RUs ranges from 4 to 12. Figure 10 shows that, as the number of available RUs increases, both *HMS* and *HMS+PR* reduce the reconfiguration overhead generated (Figure 10(a)). However, this occurs in spite that the number of performed reconfigurations increases, especially if the algorithm does not spread the partitioned snapshots in the RUs (*HMS* and *HMS+PR* approaches, Figure 10(b). This is due to the prefetch that is applied during the scheduling phase of *HMS*. Indeed, the figure shows that the shape of the plot for the SoA mappers is exactly the same in Figures 10(a) and 10(b), since no prefetch is applied in these cases and hence the reconfiguration overhead is simply the number of reconfigurations multiplied by the overhead generated by one single bitstream. However, for the *HMS* and *HMS+PR* approaches, we can observe that, as the number of RUs in the system increases, the reconfiguration overhead generated decreases (Figure 10(a)) in spite that the number of reconfigurations incrases (Figure 10(b)).

This trend shows that both versions of HMS apply more and more prefetch when a greater number RUs is available in the system, since the reconfiguration overhead of each approach is no longer proportional to the number of reconfigurations that are carried out.

Next, in Figure 10(b) we can also observe that, for 12 RUs, the number of reconfigurations carried out by our *HMS+PR* approach is greater than both SoA mappers, conversely to the remaining cases depicted in the figure. Although this does not have

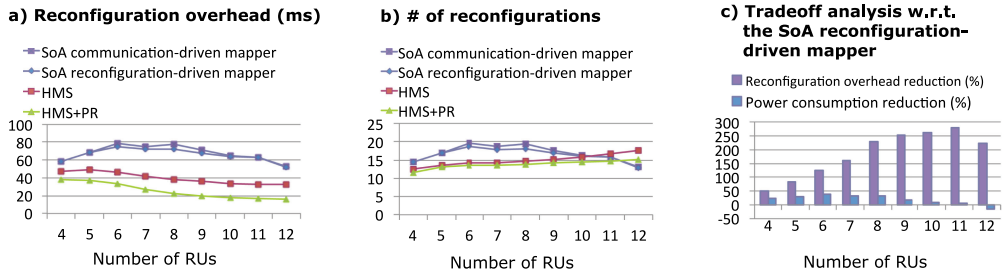

Fig. 10. Performance evaluation of the proposed approach (setting the time margin to 0) on the synthetic benchmarks with respect to the SoA mappers in Murali et al. [2006a] and Beretta et al. [2011b], for different numbers of available RUs (a, b), and the corresponding improvements in terms of communication overhead and power consumption (c)

any negative effect in the reconfiguration overhead of the system (see Figure (a)), it is well known that more reconfigurations lead to a greater power consumption. For instance, according to our experimental measurements taken with a logic analyzer in our hardware platform, the power consumed due to the dynamic reconfigurations by a hardware multi-tasking system deployed on a XUPV5-LX110T development board that uses the ICAP configuration port controlled by a MicroBlaze processor is 0.3W, under the following setup: ICAP read and write FIFO depths to 128 and 64 words (4 bytes each), ICAP block size to 32 words, and partial bitstreams being fetched from the off-chip *compact FLASH* memory. This is a typical setup for carrying out partial reconfigurations through the ICAP port for this board.

Finally, in Figure 10(c) we have compared the proposed HMS+PR approach with the SoA reconfiguration-driven mapper (the results of the comparison with the SoA communication-driven mapper are similar and have been omitted for the sake of brevity), in terms of both reconfiguration overhead reduction and power consumption reduction. This comparison has been performed by taking into account a different number of RUs, ranging from 4 to 12. As shown in this figure, the proposed HMS+PR approach reduces the reconfiguration overhead from 50% to more than 250% (around 185% in the average), while moderately reducing also energy consumption in most of the cases (up to 40%, and around 20% in the average). However, in some cases (such as in the one consisting of 12 RUs), the energy consumption of the SoA reconfiguration-driven mapper is slightly lower (around 15%) than the one of the proposed HMS+PR approach, but this penalty is traded for a considerable increase of the reconfiguration overhead (more than 200%).

Also, note that the SoA reconfiguration-based mapper generates fewer reconfigurations than the communication-driven one, since the former was explicitly optimized to take into account the dynamic reconfigurations.

7. CONCLUSIONS

This article presents an alternative task representation model that enhances current task representation models, in particular Data Flow Graphs (DFGs), Control and Data Flow Graphs (CDFGs), Petri Nets and Data Flow Diagrams (DFDs), by combining their respective features. This new model has been named *Temporal Constrained Data Flow Diagrams* (TCDFDs), and it is able to capture at the same time the data dependencies among the tasks of the applications and whether several tasks constitute a pipeline and thereby should coexist in time.

In addition, since none of the state-of-the-art approaches is able to efficiently map and schedule TCDFDs in reconfigurable systems, this article also proposes

a mapping-scheduling methodology specially designed for TCDFDs. Experimental results show that the proposed approach outperforms other state-of-the-art mapping and scheduling approaches [Murali et al. 2006a; Beretta et al. 2011b; Clemente et al. 2011b], reducing the total application execution time by up to 62%, and saving up to 75% of the available hardware resources for the execution of a set of real-world case studies and synthetic benchmarks. In addition, it also reduces the reconfiguration overhead by up to 89%, at the cost of increasing the number of reconfigurations that are carried out (and hence the power consumption of the system) in some punctual cases.

REFERENCES

P. Alexander and C. Kong. 2001. Rosetta: Semantic support for model-centered systems-level design. *Computer* 34, 11, 64–70. DOI:http://dx.doi.org/10.1109/2.963446

S. Anellal and B. Kaminska. 1993. Scheduling of a control and data flow graph. In *Proceedings of the IEEE International Symposium on Circuits and Systems*. 1666–1669. DOI:http://dx.doi.org/10.1109/ISCAS.1993.394061

L. A. Belady. 1966. A study of replacement algorithms for a virtual-storage computer. *IBM Syst. J.* 5, 2, 78–101. DOI:http://dx.doi.org/10.1147/sj.52.0078

A. Bender. 1996. MILP based task mapping for heterogeneous multiprocessor systems. In *Proceedings of the European Design Automation Conference with EURO-VHDL '96 and Exhibition*. 190–197. DOI:http://dx.doi.org/10.1109/EURDAC.1996.558204

L. Benini, D. Bertozzi, D. Bruni, N. Drago, F. Fummi, and M. Poncino. 2003. SystemC cosimulation and emulation of multiprocessor SoC designs. *Computer* 36, 4, 53–59. DOI:http://dx.doi.org/10.1109/MC.2003.1193229

L. Benini and G. De Micheli. 2002. Networks on chip: A new paradigm for systems on chip design. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 418–419. DOI:http://dx.doi.org/10.1109/DATE.2002.998307

I. Beretta, V. Rana, D. Atienza, and D. Sciuto. 2011a. Island-based adaptable embedded system design. *IEEE Embedded Syst. Lett.* 3, 2, 53–57. DOI:http://dx.doi.org/10.1109/LES.2011.2115991

I. Beretta, V. Rana, D. Atienza, and D. Sciuto. 2011b. A mapping flow for dynamically reconfigurable multicore system-on-chip design. *IEEE Trans. Comput.-Aid. Design Integr. Circuits Syst.* 30, 8, 1211–1224. DOI:http://dx.doi.org/10.1109/TCAD.2011.2138140

P. D. Bruza and Th. P. van der Weide. 1993. The semantics of data flow diagrams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*. 66–78. DOI:http://dx.doi.org/10.1.1.40.9398

C. Chang, J. Wawrzynek, and R.W. Brodersen. 2005. BEE2: A high-end reconfigurable computing system. *IEEE Des. Test Comput.* 22, 2, 114–125. DOI:http://dx.doi.org/10.1109/MDT.2005.30

J. A. Clemente, I. Beretta, V. Rana, D. Atienza, and D. Sciuto. 2011a. A hybrid mapping-scheduling technique for dynamically reconfigurable hardware. In *Proceedings of the 21st International Conference on Field Programmable Logic and Applications*. 177–180. DOI:http://dx.doi.org/10.1109/FPL.2011.40

J. A. Clemente, J. Resano, C. Gonzalez, and D. Mozos. 2011b. A hardware implementation of a runtime scheduler for reconfigurable systems. *IEEE Trans. VLSI Syst.* 19, 7, 1263–1276. DOI:http://dx.doi.org/10.1109/TVLSI.2010.2050158

S. F. Corbetta, M. Morandi, M. Novati, M. D. Santambrogio, and D. Sciuto. 2007. Two novel approaches to online partial bitstream relocation in a dynamically reconfigurable system. In *Proceedings of the IEEE Annual Symposium on VLSI*. 457–458. DOI:http://dx.doi.org/10.1109/ISVLSI.2007.99

R. Cordone, F. Redaelli, M. A. Redaelli, M. D. Santambrogio, and D. Sciuto. 2009. Partitioning and scheduling of task graphs on partially dynamically reconfigurable FPGAs. *IEEE Trans. Comput.-Aid. Design Integr. Circuits Syst.* 28, 5, 662–675. DOI:http://dx.doi.org/10.1109/TCAD.2009.2015739

J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Yuhong Xiong. 2003. Taming heterogeneity: The Ptolemy approach. *Proc. IEEE* 91, 1, 127–144. DOI:http://dx.doi.org/10.1109/JPROC.2002.805829

R. Eskinazi, M. E. Lima, P. R. M. Maciel, C. A. Valderrama, A. G. S. Filho, and P. S. B. Nascimento. 2005. A timed petri net approach for pre-runtime scheduling in partial and dynamic reconfigurable systems. In *Proceedings of the 19th International Parallel and Distributed Processing Symposium*. 154a. DOI:http://dx.doi.org/10.1109/IPDPS.2005.72

F. Ferrandi, C. Pilato, D. Sciuto, and A. Tumeo. 2010. Mapping and scheduling of parallel C applications with ant colony optimization onto heterogeneous reconfigurable MPSoCs. In *Proceedings of the 15th*

Asia and South Pacific Design Automation Conference. 799–804. DOI:http://dx.doi.org/10.1109/ASPDAC.2010.5419782

S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh. 2004. An optimal algorithm for minimizing runtime reconfiguration delay. *ACM Trans. Embed. Comput. Syst.* 3, 237–256. DOI:http://dx.doi.org/10.1145/993396.993398

A. Hansson. 2005. A unified approach to mapping and routing in a combined guaranteed service and best-effort network-on-chip architecture. Tech. Rep., Lund University, Sweden.

C. Haubelt, S. Otto, C. Grabbe, and J. Teich. 2005. A system-level approach to hardware reconfigurable systems. In *Proceedings of the 10th Asia and South Pacific Design Automation Conference*. 298–301. DOI:http://dx.doi.org/10.1109/ASPDAC.2005.1466177

B. Hendrickson and R. Leland. 1994. The Chaco user's guide, Version 2.0. Tech. Rep. Sandia National Laboratories. http://www.cs.sandia.gov/_bahendr/chaco.html

International Telecommunication Union (ITU). 1993. ITU-T Recommendation H.261. (1993). http://www.itu.int/rec/T-REC-H.261/e

M. Janiaut, C. Tanougast, H. Rabah, Y. Berviller, C. Mannino, and S. Weber. 2005. Configurable hardware implementation of a conceptual decoder for a real-time mpeg-2 analysis. In *Proceedings of the 15th International Conference on Field Programmable Logic and Applications*. 386–390. DOI:http://dx.doi.org/10.1109/FPL.2005.1515752

C. Kao. 2006. Benefits of partial reconfiguration. Xilinx.

K. M. Kavi, B. P. Buckles, and U. N. Bhat. 1986. A formal definition of data flow graph models. *IEEE Trans. Comput. C-35*, 11, 940–948. DOI:http://dx.doi.org/10.1109/TC.1986.1676696

E. A. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorffer, S. Sachs M. Stewart, K. Vissers, and P. Whitaker. Overview of the Ptolemy project. Tech. Rep.

M. Li and Y. Ruan. 2011. Approach to formalizing UML sequence diagrams. In *Proceedings of the 3rd International Workshop on Intelligent Systems and Applications*. 1–4. DOI:http://dx.doi.org/10.1109/ISA.2011.5873348

Z. Li. 2002. Configuration prefetching techniques for partial reconfigurable coprocessor with relocation and defragmentation. In *Proceedings of the ACM/SIGDA 10th Symposium on Field-Programmable Gate Arrays*. 187–195. DOI:http://dx.doi.org/10.1145/503048.503076

T. Lindroth, N. Avessta, J. Teuhola, and T. Seceleanu. 2006. Complexity analysis of H.264 decoder for FPGA design. In *Proceedings of the IEEE International Conference on Multimedia and Expo.* 1253–1256. DOI:http://dx.doi.org/10.1109/ICME.2006.262765

S. Lukovic and L. Fiorin. 2008. An automated design flow for NoC-based MPSoCs on FPGA. In *Proceedings of the 19th IEEE/IFIP International Symposium on Rapid System Prototyping*. 58–64. DOI:http://dx.doi.org/10.1109/RSP.2008.31

X. Mei-hua, C. Yu-lan, R. Feng, and C. Zhang-jin. 2007. Optimizing design and FPGA implementation for CABAC decoder. In *Proceedings of the International Symposium on High Density packaging and Microsystem Integration*. 1–5. DOI:http://dx.doi.org/10.1109/HDP.2007.4283645

S. O. Memik, G. Memik, R. Jafari, and E. Kursun. 2003. Global resource sharing for synthesis of control data flow graphs on FPGAs. In *Proceedings of the 50th Design Automation Conference.* 604–609. DOI:http://dx.doi.org/10.1109/DAC.2003.1219090

S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli. 2006a. A methodology for mapping multiple use-cases onto networks on chips. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 118–123. DOI:http://dx.doi.org/10.1109/DATE.2006.244007

S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli. 2006b. A methodology for mapping multiple use-cases onto networks on chips. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition.* 1–6. DOI:http://dx.doi.org/10.1109/DATE.2006.244007

J. Noguera and R. M. Badía. 2004. Multitasking on reconfigurable architectures: microarchitecture support and dynamic scheduling. *ACM Trans. Embed. Comput. Syst.* 3, 2, 385–406. DOI:http://dx.doi.org/10.1145/993396.993404

J. Resano, D. Mozos, D. Verkest, and F. Catthoor. 2005. A reconfiguration manager for dynamically reconfigurable hardware. *IEEE Des. Test Comput.* 22, 5, 452–460. DOI:http://dx.doi.org/10.1109/MDT.2005.100

M. Roitzsch. 2007. Slice-balancing H.264 video encoding for improved scalability of multicore decoding. In *Proceedings of the 7th ACM and IEEE International Conference on Embedded Software*. 269–278. DOI:http://dx.doi.org/10.1145/1289927.1289969

H. Taghipour, J. Frounchi, and M. H. Zarifi. 2008. Design and implementation of MP3 decoder using partial dynamic reconfiguration on Virtex-4 FPGAs. In *Proceedings of the International Conference on Computer and Communication Engineering*. 683–686. DOI:http://dx.doi.org/10.1109/ICCCE.2008.4580691

B. D. Theelen, M. C. W. Geilen, S. Stuijk, S. V. Gheorghita, T. Basten, J. P. M. Voeten, and A. H. Ghamarian. 2008. Scenario-aware data flow. Tech. Rep. Eindhoven University of Technology, Eindhoven, The Netherlands.

M. Verderber, A. Zemva, and D. Lampret. 2003. HW/SW partitioned optimization and VLSI-FPGA implementation of the MPEG-2 video decoder. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. 238–243 suppl. DOI:http://dx.doi.org/10.1109/DATE.2003.1253835

H. Walder and M. Platzner. 2004. A Runtime environment for reconfigurable hardware operating systems. In *Proceedings of the 14th International Conference on Field Programmable Logic and Application*, Lecture Notes in Computer Science, vol. 3203. Springer, 831–835. DOI:http://dx.doi.org/10.1007/978-3-540-30117-284

S. Wildermann, F. Reimann, D. Ziener, and J. Teich. 2011. Symbolic design space exploration for multi-mode reconfigurable systems. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*. 129–138. DOI:http://dx.doi.org/10.1145/2039370.2039393

Xilinx Corporation. 2010. *Virtex-5 FPGA User Guide.*

Xilinx Corporation. 2012a. *MicroBlaze Processor Reference Guide.*

Xilinx Corporation. 2012b. Zynq-7000 extensible processing platform overview.

D. Zaretsky, G. Mittal, R. P. Dick, and P. Banerjee. 2005. Generation of control and data flow graphs from scheduled and pipelined assembly code. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing.* Lecture Notes in Computer Science, vol. 4339, Springer, 76–90. DOI:http://dx.doi.org/10.1007/978-3-540-69330-7_6

H. Zhang and G. Wu. 2009. Petri nets based scheduling modeling for embedded systems. In *Proceedings of the 2nd International Conference on Intelligent Computation Technology and Automation*, Vol. 4. 80–83. DOI:http://dx.doi.org/10.1109/ICICTA.2009.736

R. Zurawski and M. Zhou. 1994. Petri nets and industrial applications: A tutorial. *IEEE Trans. Ind. Electron.* 41, 6, 567–583. DOI:http://dx.doi.org/10.1109/41.334574

Received February 2013; revised August 2013, October 2013; accepted November 2013