A Mapping Flow for Dynamically Reconfigurable Multi-Core System-on-Chip Design

Ivan Beretta, Vincenzo Rana, David Atienza, *Member, IEEE,* and Donatella Sciuto, *Fellow, IEEE*

***Abstract*—Nowadays, multi-core systems-on-chip (SoCs) are typically required to execute multiple complex applications, which demand a large set of heterogeneous hardware cores with different sizes. In this context, the popularity of dynamically reconfigurable platforms is growing, as they increase the ability of the initial design to adapt to future modifications. This paper presents a design flow to efficiently map multiple multi-core applications on a dynamically reconfigurable SoC. The proposed methodology is tailored for a reconfigurable hardware architecture based on a flexible communication infrastructure, and exploits applications similarities to obtain an effective mapping. We also introduce a run-time mapper that is able to introduce new applications that were not known at design-time, preserving the mapping of the original system. We apply our design flow to a real-world multimedia case study and to a set of synthetic benchmarks, showing that it is actually able to extract similarities among the applications, as it achieves an average improvement of 29% in terms of reconfiguration latency with respect to a communication-oriented approach, while preserving the same communication performance.**

***Index Terms*—Field programmable gate arrays, platform-based design, reconfigurable architectures, run-time adaptability.**

I. Introduction

Over the last years, multi-core systems-on-chip (SoCs) have been proposed in many different application scenarios, such as in multimedia applications [1]. In fact, those systems guarantee high performance thanks to the combined execution of many heterogeneous functional units (e.g., general-purpose processors, digital signal processors or DSPs, specific purpose IP-cores), which we refer to as *cores* in the remainder of this paper. In multi-core SoCs design, the definition of a performing and power-efficient communication infrastructure [2], [3], and the mapping of the cores onto it are key aspects in the quality of the final product. Other non-functional requirements, such as high flexibility, have also become an important factor in SoC industry, in order to increase the versatility and the lifetime of the system. A rising trend in advanced platforms [4] is to couple traditional processing elements (CPUs, DSPs, and others) and memories with a large amount of reconfigurable devices, such as field-programmable gate arrays (FPGAs). However, the reconfiguration process has different drawbacks that should be kept under control by the synthesis tool during the design phase, including high latency (in the order of hundreds of milliseconds) and high power consumption. A suitable mapping of cores on the FPGA area can dramatically reduce the number of reconfigurations and improve the efficiency of the communication infrastructure, as cores that communicate frequently can be placed physically close to each other [5].

In this paper, we propose a mapping methodology for multi-core applications on reconfigurable devices. We start from an execution model that allows multiple applications to be executed on the same platform by switching among them at run-time by means of dynamic reconfiguration. We define a versatile hardware architecture, which divides the reconfigurable area into slots that can be connected using different communication paradigms, such as networks-on-chip (NoCs) [3] or buses. Then, we propose a mapping algorithm for a known set of applications, which we underlined in [6], to optimize both the number of reconfigurations and the communication performance of the system. Finally, we propose a run-time mapper flow that allows the addition of new applications at run-time, by possibly reusing parts of the already-deployed applications, as we first discussed in [7] and [8]. With respect to our previous papers, we propose a unique flow for both design-time mapping (DTM) and run-time mapping (RTM), whose main innovative contributions are the following.

1) We provide a formal definition of the DTM and the RTM problems, analyzing their computational complexity and proving their NP-completeness.
2) We include a new RTM algorithm based on a Boolean satisfiability (SAT) solver, which aims finding a mapping for a new application by reusing parts of the existing ones. Unlike the greedy algorithm discussed in [7], the SAT solver always identifies a solution whenever it exists, though its complexity is higher.
3) We extend the RTM flow by introducing the support for a technique known as bitstream relocation [9]. The goal is to increase the solution space of the mapping problem in order to improve the quality of the solution.

Manuscript received March 11, 2011; accepted March 21, 2011. Date of current version July 20, 2011. This work was supported in part by the HiPEAC network of excellence (www.hipeac.net) and the Swiss National Science Foundation, under Grant 200021-127282. This paper was recommended by Associate Editor Y. Xie.

I. Beretta and D. Atienza are with the ESL-Ecole Polytechnique Fédérale de Lausanne, ESL-IEL-STI-EPFL, Lausanne 1015, Switzerland (e-mail: ivan.beretta@epfl.ch; vincenzo.rana@epfl.ch; david.atienza@epfl.ch).

V. Rana is with the ESL-Ecole Polytechnique Fédérale de Lausanne, ESL-IEL-STI-EPFL, Lausanne 1015, Switzerland, and also with the DEI-Politecnico di Milano, Milan 20133, Italy.

D. Sciuto is with the DEI-Politecnico di Milano, Milan 20133, Italy (e-mail: sciuto@elet.polimi.it).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2011.2138140

4) We propose a complete real-world case study based on video encoding/decoding, and we apply our flow to show its benefits, especially in terms reconfiguration latency.

This paper is structured as follows. In Section II, we provide an overview of the related works. We introduce some definitions in Section III, and then we discuss the proposed mapping flow in Section IV. We then propose a DTM and a RTM algorithm in Sections V and VI. Next, we validate the algorithms on a real-world case study in Section VII, and using additional benchmarks in Section VIII. Finally, Section IX concludes the paper with a summary of the main conclusions of this paper.

II. Related work

The problem of mapping cores on a reconfigurable device is widely explored in literature, and many approaches have been designed to solve the mapping problem to enhance performance in many different scenarios, such as in network processors [10], [11]. Most of the related works aim at finding a suitable mapping at design-time, though a few run-time mappers have been proposed. In general, dynamic reconfiguration is rarely taken into account, as most of the mappers are more focused on area [12] or communication [5], and they are not designed to detect core-level similarities among applications.

Over the last years, many design-time approaches have been proposed to map multi-core applications on specific communication infrastructures, with a particular emphasis on the emerging NoCs. The authors of [5], e.g., proposed a design-time mapper to optimize the communication overhead of an application mapped onto a NoC. The algorithm in [12] performs mapping and path selection in order to minimize area requirements and power consumption. In [13], the authors proposed a technique to evaluate and optimize core mapping, path selection, and time-slot assignment at the same time. However, all these approaches are proposed for a design-time scenario and are not suitable to incrementally add other applications at run-time. Furthermore, they are meant for static architectures, thus they do not consider dynamic reconfiguration, whereas our approach can also reduce the number of reconfigurations that are required to configure an application on an FPGA.

Dynamic reconfiguration is explicitly considered in [14], as the authors consider the opportunity of switching among different parts of the same application at run-time, and they proposed a design-time algorithm to reduce the reconfiguration overhead by minimizing the number of required reconfigurations. The main limitation of this approach, in addition to the fact that only a single application is considered, is the assumptions that the application can be divided into smaller tasks with the same area, and whose execution time is negligible with respect to its reconfiguration overhead. Conversely, thanks to the hardware architecture we propose, our approach can efficiently handle cores of different sizes. Moreover, the proposed mapping methodology does not require any assumption about the execution time of each core, thus making it applicable to a larger number of real-world applications.

Even though most of the existing mapping approaches are design-time algorithms, a few works about RTM can be found in the literature. In [15], the authors proposed a technique to generate FPGA configuration files at run-time using a sustainable amount of resources. This approach introduces the so-called parameterizable configurations, which can be easily converted into specific FPGA configuration files with a low computation effort. However, the paper does not propose a complete mapping technique to place applications on the device to optimize a specific performance metric.

A mapping algorithm that incrementally adds new applications at run-time is proposed in [16]. This mapper takes an incoming application and, according to the current device usage, tries to allocate resources in an incremental way, i.e., by exploiting the unused area on the device without modifying the existing layout. The algorithm is tailored for a NoC-based architecture, and it aims at optimizing the power consumption of the on-chip communication. The algorithm proves to be effective because it is able to take local decisions to improve the long-term quality of the mapping. In [17], the approach is extended to consider both internal and external contention of the network, and to implement a machine learning algorithm to better respond to run-time changes. However, both [16] and [17] mainly focused on geometric aspects, i.e., they aimed at determining the best shape for the new application in order to fit it in the existing layout, and reconfiguration-related metrics are not considered. Furthermore, the two works do not detect similarities among the applications, whereas our flow effectively collects shared cores in order to improve both area and reconfiguration-related metrics, such as latency.

The geometric aspects of run-time addition of new applications are also considered in [18], where a placement algorithm is proposed to reduce area fragmentation and to optimize the number of applications that can fit on the FPGA area at the same time. The authors try to combine the benefits of a slot-based placement policy, where a new application is assigned to a fixed-size region of the device, and a flexible placement technique that does not use redundant resources. The proposed algorithm starts from a pre-partitioned layout, and then modifies it whenever the area availability is too low for the incoming application. The placer can perform both split and merging of existing regions, and it also avoids area fragmentation. Still, the approach does not explicitly optimize any communication-related metric and, like most of the previously described approaches, it does not detect similarities among applications.

In this paper, we propose a reconfiguration-aware mapping flow, which includes both a design-time and a run-time dimension. The proposed approach takes advantage of the similarities among the applications to enforce core reuse and reduce the transition time between two applications.

III. Preliminary Concepts

This section introduces some basic notions that are essential to understand the proposed mapping problem. First, we introduce the concept of multi-core application, and we formalize it using a mathematical representation. We also provide an overview of how these applications can be actually deployed on a reconfigurable device, as we introduce the concept of partial dynamic reconfiguration on FPGAs. We then propose a reference hardware architecture that supports dynamic reconfiguration, and guarantees flexibility and good performance. Finally, we define how the applications are executed on the target architecture, introducing the metrics that should be considered during the mapping process.

A. Applications

The proposed approach can map many different applications onto an FPGA device. We define an *application* as a set of heterogeneous cores which cooperate and communicate with each other in order to perform a complex task. An application can be modeled using an undirected graph called communication graph (CG), defined as follows.

Definition 1: (Communication graph): A CG is an undirected graph $G = (V, E)$, where each node $v_i \in V$ represents an application core, and each edge $e_{ij} = (v_i, v_j) \in E$ represents a communication between cores v_i and v_j. A weight $size_i$ is associated with each node $v_i \in V$, and it represents the core size in terms of number of FPGA slices that are required to implement it on the device. Each edge $e_{ij} \in E$ is associated with a weight $comm_{ij}$, representing the bandwidth of the communication between cores v_i and v_j, weighted the criticality of the communication.

The size of a single application should not exceed the device area, otherwise it is not possible to deploy it at once. This assumption is not really strict, since modern FPGAs provide a sufficient amount of resources to host large and complex applications. Moreover, if an application does not fit into the device area, it may be possible to partition it [19] into multiple smaller sub-applications that can be deployed sequentially. Finally, we assume that the application cannot start until all the corresponding cores have been configured on the device.

Different applications can share a certain amount of cores to perform the same operations as part of their overall computation, which is especially true when such applications belong to the same domain (e.g., applications related to signal processing may require to perform a Fourier transform at some point). Intuitively, two applications are *similar* when they share a large percentage of cores, even though their interaction (i.e., their communication requirements) may differ. In order to quantify this idea, we define a metric named *application similarity* as the average percentage of shared cores between each pair of applications in the application set.

B. Dynamic Reconfiguration Support

The proposed mapping flow can be applied to any hardware device that supports partial dynamic reconfiguration, but it is best suited for last-generation FPGA families—such as Xilinx Virtex-4 and Virtex-5 families—because of the large amount of hardware resources they offer and their fine-grained dynamic reconfiguration capabilities [20].

In order to physically configure an FPGA with the desired application, one or more configuration files, called *bitstreams*, have to be used. A bitstream can be either *full* or *partial*, depending on whether it configures the whole device, or only a smaller region. The bitstreams are generated in a process named *bitstream synthesis*, which typically requires several tens of minutes, depending on the complexity of the circuit and on the number of bitstreams to be generated. Since the complexity of the circuit is defined by the hardware designer,

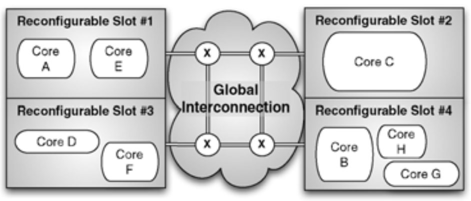

Fig. 1. Example of core mapping with a NoC-based global interconnection.

we can only aim at minimizing the synthesis time by reducing the number of bitstreams that must be generated from scratch.

C. Hardware Architecture Model

The hardware architecture we employ in this paper is logically divided into two layers: the *communication layer*, and the *computation layer*, as proposed in [21]. The computation layer includes the cores of the executed application, while the communication layer guarantees connectivity between them. The division between the two layers makes the reconfiguration process simpler and more efficient, as the same communication infrastructure can serve multiple applications.

We divide the FPGA area into fixed-size reconfigurable regions called *slots*, as shown in Fig. 1. Each slot provides the same amount of resources [22], and it can be filled with one or more cores. The homogenous amount of resources in each slot is a necessary condition to support bitstream relocation, a feature we will discuss in Section IV-B. Instead, the possibility to include multiple cores in one region differentiates this model from other grid-based architectures that can be found in literature [23], where a slot only contains one core, and the mapping problem is only driven by communication-related metrics. Since a core cannot be split among multiple slots, the minimum slot size is bounded by the largest core in the application set. A single slot is the smallest amount of area that can be reconfigured using a partial bitstream; this assumption fits the requirements of modern FPGAs, where reconfigurable modules must have a rectangular shape.

The slot size (or, equivalently, the number of slots) in the hardware architecture is determined at design-time by analyzing the cores that have to be mapped. In particular, if all the cores have approximately the same size, then the slot can be dimensioned to contain only one core, because the amount of unused area inside the slot is guaranteed to be low. In real applications, however, cores may have very different size, and therefore the best strategy is to use larger slots, and let the mapper assign more cores to each slot to reduce the amount of unused area. In practice, a number of slots between 6 and 12 (depending on the device size) is generally a good choice. As we will show in Section VIII-C, the proposed mapping flow is able to optimize the reconfiguration overhead independently from the number of slots, but its improvements with respect to state-of-the-art algorithms increases with slots of large size.

The communication layer is further divided into two levels: the intra-slot, or local, and inter-slot, or global, communication infrastructures. In this paper, we opted to implement the communication infrastructure using a NoC [3], as it provides a good tradeoff between performance and flexibility, but the proposed architecture can be adapted to other communication paradigms. The inter-slot infrastructure connects the different slots, and it is provided as a fixed communication backbone that is never reconfigured during the system execution, thus guaranteeing connectivity even during the reconfiguration process. The intra-slot infrastructure connects cores within the same slot and resolves their communication internally without accessing the global backbone. This local infrastructure is configured in the slot during the reconfiguration process along with the cores. The importance of the intra-slot NoC grows when a low number of large slots is used, as many cores are assigned to the same slot, and the communication among them becomes more complex.

Definition 2: (Slot configuration): A configuration for slot i is composed of the set of cores that are mapped in i at the same time, the local communication infrastructure among the cores, and the access to the global network. A configuration is encoded by a single partial bitstream, and it may be reconfigured in the target slot at any time.

The efficiency of the two communication levels is not uniform, as the intra-slot NoC can be specifically optimized for the cores in the slot, though this is beyond the scope of this paper. Conversely, the inter-slot NoC has a fixed topology, it employs a predetermined routing algorithm (in this case, we assume a deterministic *X-Y routing*), and its efficiency can be optimized by keeping highly communicating cores close to each other to avoid sending a large amount of traffic over the network. To estimate the effect of the traffic, which affects the overall system performance, we define the *communication overhead* metric as follows.

Definition 3: (Communication overhead): Given an application A and an assignment of each core of A to a slot, the communication overhead is defined as follows:

$$Comm\_Overhead_A = \sum_{i,j \in A} comm_{ij} \cdot hops_{ij} \qquad (1)$$

where $comm_{ij}$ is the edge weight specified in the CG, and $hops_{ij}$ is the number of hops that are required to reach core j from core i in the current assignment.

The value of $hops_{ij}$ is related to the time required to deliver a message using the inter-slot NoC; if two cores belong to the same slot, they can communicate using the intra-slot infrastructure, therefore $hops_{ij}$ is set equal to zero. Finally, it is possible to specify the maximum amount of traffic that can be sent on each link of the global NoC, as the algorithm will exclude all the solutions that violate these constraints.

We successfully implemented the proposed hardware architecture model on a Xilinx Virtex-5 FPGA, thus proving it is compliant with the current technology. The implementation includes six slots, which are connected using a global NoC with a 2×3 mesh grid topology. The interfaces between the global NoC and each slot are implemented using hard macros [20], and the infrastructure is proved to be operational even during the reconfiguration of one of the slots. The dynamic reconfiguration process is performed within the device, as it is controlled by a *MicroBlaze* soft processor that can access an internal configuration port (named *ICAP*) to load a partial bitstream in a desired slot.

D. Execution Model

As mentioned above, each application is assumed to fit into the FPGA area, thus it can be deployed at once. Then, depending to the area requirements of each application, we can identify two scenarios as follows:

1) if two or more applications can fit into the device area, it is possible to merge their CGs into a single unconnected graph and configure them at the same time;
2) if it is not feasible to deploy two applications at once because of their area requirements, it is possible to switch among them by reconfiguring part of the device and loading the required cores. This scenario is more general and may occur even when more applications are merged together, and it is more interesting for our approach.

During an application switching, a timing overhead is introduced, which is proportional to the number of slots that are reconfigured. Thus, we define a metric called average number of reconfigurations, which indicates the average number of slots to be reconfigured while switching between any pair of applications, as an index of the reconfiguration overhead.

Note that applications may use some common cores, this they can reuse some of the configurations that were previously loaded for another application, thus simplifying the switching process. To increase the exploration space and to optimize the performance, an application can reuse an existing configuration even if it exploits only some of the cores that are included in the configuration, as the unused cores can be left idle.

IV. Design Flow Definition

We summarize the proposed mapping flow in Fig. 2. We identify a *design-time* mapping phase, which is computed before the system deployment over a known set of applications, and the *run-time* mapping phase, which arises when a new application is added to an existing system at run-time. In fact, the RTM can be considered as a problem on its own because it has to keep the existing solution computed at design-time into account, and it has to guarantee that the new application is available for deployment in a short time, thus making the algorithm more constrained than the design-time one. In this section, we define the DTM and the RTM problems in a formal way, we characterize all the different sub-problems that can be identified in the flow, and we analyze their complexity.

A. Design-Time Mapping

The mapping of a set of applications, which are known before the system deployment, can be computed at design-time. Once the mapping of all the application cores has been determined, all the corresponding bitstreams must be generated from scratch. However, the bitstream synthesis process is performed offline and its length does not affect the run-time behavior of the system, thus the mapper does not need to minimize the number of bitstreams.

More formally, given a set of N_{app} applications $A_i = (V_i, E_i)$, $0 \le i \le N_{app}$, the DTM problem is the assignment of each node $v \in V_i$ of each application A_i, to a reconfigurable slot. A solution S_{DTM} contains the binding between each core and the selected slot, and it should minimize both the average number of reconfigurations and the communication overhead.

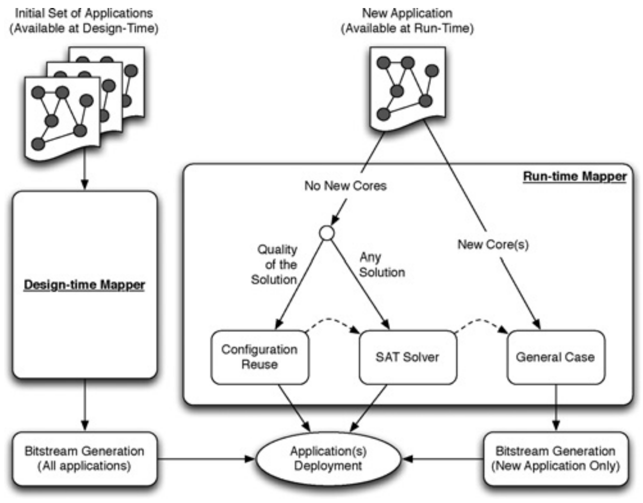
Initial Set of Applications (Available at Design-Time)
New Application (Available at Run-Time)
Run-time Mapper
No New Cores
Quality of the Solution
Any Solution
New Core(s)
Design-time Mapper
Configuration Reuse
SAT Solver
General Case
Bitstream Generation (All applications)
Application(s) Deployment
Bitstream Generation (New Application Only)

Fig. 2. Proposed mapping flow.

B. Run-Time Mapping

The design-time mapper focuses on an initial and well-known set of applications, but new applications may become available at a later time. The incoming application may or may not introduce cores that were not included in the applications mapped at design-time. If the cores are not highly specific, it is more likely that the new application does not include any new core, so its cores can be all found in some configuration computed at design-time. In this case, the fastest way to map the new application is to find a combination of existing configurations that includes all the required cores, which would not require the generation of any new bitstream. Formally, we define the *run-time mapping with configuration reuse* (RTM-CR) as a phase of the RTM where, given an initial mapping S_{DTM} and a new application A (whose cores are used by at least one application known at design-time), the algorithm aims at finding a combination of configurations of S_{DTM}—if it exists—such that all the cores of A are mapped.

We propose two different approaches to handle the configuration reuse problem, as shown in Fig. 2: a greedy algorithm looks for a good-quality solution, though it may not find a feasible solution in some cases, whereas a SAT solver approach finds a solution whenever it exists, though no warranty about its quality is given. According to the designer's needs, one of the two approaches can be selected, or they can be used sequentially to look for any feasible solution whenever the greedy approach cannot find a good-quality one.

The main issue of configuration reuse is that if two configurations have been synthesized for the same slot, they cannot be both selected to map the new application, and a solution might not be found. This problem may be avoided if the existing configurations can be moved to a different slot, thanks to a process called *bitstream relocation*. Relocation has been implemented in older FPGAs [9], such as Xilinx Virtex II-Pro, and has also been proposed for modern devices [24], such as Virtex-4 and Virtex-5. For this reason, we decided to include relocation in the proposed flow, even though we also discuss how the algorithms have been modified to handle a scenario where relocation is not supported.

In general, configuration reuse is not sufficient to find a solution, either because a suitable combination of existing configuration cannot be found, or because the new application introduces one or more new cores. In this case, the generation of at least one new bitstream is required, thus affecting the deployment time of the new application. However, it is possible to reduce this overhead by keeping the number of new bitstreams low, which can be achieved by exploiting configuration reuse to map only a subset of cores, and then creating new configurations for the remaining ones: this problem is named *run-time mapping in the general case* (RTM-G). In particular, given a design-time mapping S_{DTM} and a new application A, the RTM-G problem is the assignment of each core of A to a specific slot, possibly reusing part of the configurations of S_{DTM}, aiming at minimizing the average number of reconfigurations and the communication overhead of all the applications, as well as the number of new bitstreams. Even in the RTM-G phase of the proposed flow, relocation of existing configurations can be exploited, if it is supported.

C. Complexity Analysis

This section analyzes the computational complexity of the different parts of the proposed mapping flow. The complexity of generic mapping problems such as the DTM or the RTM in the general case has already been estimated [12], [25], as they are instances of the *constrained quadratic assignment* problem, which is known to be *NP-Hard*. As a consequence, no deterministic polynomial-time algorithm is known to solve them, and they can only be tackled using heuristic, sub-optimal approaches.

The mapping problem based on configuration reuse (RTM-CR) has never been faced in literature, but it closely recalls other intractable combinatorial problems.

Theorem 1: RTM-CR is an *NP-Complete* problem.

Proof: We prove the theorem by using the polynomial reduction technique [26]. It can be shown that RTM-CR $\in$ *NP*, i.e., a solution for RTM-CR is certifiable in polynomial time. In fact, given a solution S_{RTM-CR}, it is possible to verify it by scanning all the cores and check that they are mapped in some slot, check that the area constraints are met, and possibly evaluate all the objective functions.

In order to complete the proof, we have to reduce an *NP-Complete* problem to RTM-CR. We refer to the SAT problem, which is known to be *NP-Complete* [27], and we show that *SAT* $\propto$ RTM-CR. Let f be a Boolean formula written in conjunctive normal form as follows:

$$f = C_1 \wedge C_2 \wedge \ldots \wedge C_n \tag{2}$$

where each clause C_j is a disjunction of literals as follows:

$$C_j = l_1 \vee l_2 \vee \ldots \vee l_m. \tag{3}$$

A literal l_i may be either a variable x_i, or its negation $\overline{x_i} = \neg x_i$. The SAT problem aims at finding an assignment for each variable x_i such that all the clauses C_j are satisfied.

For each literal l_i, we introduce a slot $slot_i$ in the target architecture. Then, we generate two configurations $Conf_i$ and $\overline{Conf_i}$ for $slot_i$, that are defined as follows:

$$Conf_i = \{C_j \mid x_i \ appears \ in \ C_j\} \tag{4}$$

$$\overline{Conf_i} = \{C_j \mid \overline{x_i} \ appears \ in \ C_j\}. \tag{5}$$

In other words, the configuration $Conf_i$ contains the set of clauses that becomes true if x_i is true, whereas $\overline{Conf_i}$ includes the clauses that are satisfied if x_i is false. As only one configuration can be mapped on $slot_i$, only one of the two set of clauses can be satisfied by the assignment of literal x_i.

The clauses belonging to the configurations represent the cores of the RTM-CR problem, whereas each literal is a slot. If an oracle exists for the RTM-CR problem, it will find a solution S_{RTM-CR} containing a set of configurations, such that all the cores are mapped. Then, we can easily convert this solution into a solution for SAT by assigning

$$x_i = \begin{cases} \text{true}, & \text{if } Conf_i \in S_{RTM-CR} \\ \text{false}, & \text{if } \overline{Conf_i} \in S_{RTM-CR} \\ \text{any value}, & \text{otherwise.} \end{cases} \quad (6)$$

That is, if $Conf_i$ is mapped on $slot_i$, we assign x_i the *true* value, and we satisfy all the clauses included in $Conf_i$. If $\overline{Conf_i}$ is mapped on $slot_i$, the x_i is set to false, whereas if neither $Conf_i$ or $\overline{Conf_i}$ have been selected, then the assignment of x_i does not affect the solution.

The proposed reduction of SAT to a RTM-CR problem is polynomial, and so is the conversion of the solution of RTM-CR into a solution of SAT. Hence, we have shown that $SAT \propto$ RTM-CR and, since $SAT \in$ *NP-Complete*, then also RTM-CR $\in$ *NP-Complete*. ■

V. Proposed Design-Time Mapper

The DTM problem can be effectively tackled using the three-stage algorithm that we initially proposed in [6], consisting of: 1) preprocessing; 2) partitioning; and 3) mapping. The three stages are iterated until a feasible solution is found, or until its quality meets certain requirements.

The basic idea is to divide the solution in two parts: the *base mapping* and the *specific configurations*. The *base mapping* includes a subset of all the cores shared by most of the applications, so its configurations can be employed by most of the applications. Due to its high degree of reusability, the base mapping is initially deployed on the device, and it is considered as the starting point to deploy each application. On the contrary, the *specific configurations* are a set of configurations that cannot be reused from the base mapping. Each specific configuration is implemented using a partial bitstream targeted to a specific slot of the hardware architecture.

A. Preprocessing

The preprocessing phase aims at exploiting similarities among the applications in the input set. This task is accomplished in two steps: the ordering and the selection phases.

1) *Ordering:* The cores the input applications are ordered according to a cost metric that takes into account both core size and the number of times it appears in the set of applications. The core size core is useful to place large cores on the device, as they could be difficult to map in a later stage, while the utilization frequency is needed to maximize core reuse.

2) *Selection:* The algorithm then selects a subset of cores called Γ such that

$$\sum_{c \in \Gamma} size_c \leq \frac{\beta}{100} \cdot Device\_Size. \quad (7)$$

In other words, the set Γ includes a set of cores that can be deployed at the same time on the device, occupying β% of the available area. However, since the total area is divided into slots, it may happen that the cores in Γ do not fit into the device if β is close to the total FPGA area. If the solution generated with a certain value of β is not deployable, the design flow is restarted from the *Selection* stage by decreasing the value of β until a feasible solution is found.

3) *Partitioning:* After the most frequently used cores have been collected into Γ, the set is partitioned into a number of clusters that represents the contents of a slot. An external tool named *Chaco* [28] is used to generate a set of balanced partitions.

As *Chaco* requires an input graph, we generate a set of virtual edges to connect the elements of the Γ set. In particular, the weight of the edge between cores i and j is defined as follows:

$$weight_{i,j} = \alpha \cdot t + (1 - \alpha) \cdot u \quad (8)$$

where u is the number of times cores i and j appear in the same CG. Thus, this variable explicitly keeps similarity among applications into account. Variable t is equal to

$$t = \sum_{z \in Apps} comm^z_{i,j} \quad (9)$$

where $comm^z_{i,j}$ is the communication between i and j in the CG of application z, and α is the parameter that can be used to reach the desired tradeoff between u and t.

The virtual edges keep track of the similarity among applications by connecting the cores of Γ according to their relative frequency. Then, *Chaco* [28] is used to perform a *min-cut* partitioning over the global graph, generating a number of clusters equal to the number of slots.

B. Mapping

The third stage of the algorithm arranges the partitions in the various slots, in order to build the base mapping. Then, the cores that have not been selected in the first stage are mapped into a set of specific configurations.

1) *Primary Mapping:* The assignment of each partition to a physical slot on the device is performed by a genetic algorithm developed for this specific problem. As the partitions are used to generate the base mapping, which is deployed at once at the system startup, the only goal of the genetic algorithm is the minimization of the communication overhead, whereas the average number of reconfigurations is not considered.

Each chromosome of the genetic algorithm has been coded as an array of locations (a single location for each slot) that can be filled up with the identifying marks of the partitions. The initial population is generated through several random permutations of the slots on the locations array. The crossover operation has been coded as an operation between two chromosomes that is able to create a new chromosome in which the position of the slots that are located in the same place in both the parents is preserved, while a random permutation is performed on the positions of all the other slots. The mutation operation has been coded as a random swap between two slots in a single chromosome. Both the crossover and the mutation preserve the validity of all the chromosomes of the population in terms of area utilization, but they may violate the link capacity constraints of the global NoC. Therefore, a feasibility check is necessary to guarantee the consistency of the solution in terms of communication.

2) *Secondary Mapping:* A second mapping is necessary to find a suitable position for all the cores that do not belong to Γ, and that will be deployed using specific configurations.

For each cluster of cores generated during the partitioning phase, we define the following metrics.

a) *Size (S)*, i.e., the percentage of area of the slot that can be set as available in the application being mapped.
b) *Reconfiguration (R)*, i.e., the percentage of times the slot is reconfigured. The average R for the selected slot across all the applications is added to this value to consider the probability that the selected slot is reconfigured by the applications that have not been mapped yet.
c) *Communication (C)*, i.e., the global communication overhead that will be generated if the core is mapped on the selected slot.

A linear combination of S, R, and $\frac{1}{C}$ is then used to select in which slot each core not belonging to Γ is mapped. As shown in [6], it is possible to assign different weights to each metric in order to obtain different tradeoffs between a system in which each application can be deployed after another one with a minimum number of reconfigured regions, and a system in which the communication among the slots is minimal.

VI. Proposed Run-Time Mapper

The design-time mapper discussed in the previous section cannot be employed when a new application is added at run-time, as it may generate a brand new mapping that requires a very long synthesis process. This section addresses the RTM problem by proposing a specific flow according to the guidelines of Fig. 2. The proposed approach is divided into two main subproblems: the first one aims at finding a solution by reusing existing configurations only, while the second one generates a suitable mix of reused and new configurations. The main difference between the two lies in the deployment time, as in the first case the new application can be deployed immediately, whereas in the second case a bitstream generation is required, and this aspect must be explicitly kept into account by the algorithm.

A. Score-Based Approach for Configuration Reuse

The RTM problem based on configuration reuse (RTM-CR) assumes that the incoming application does not introduce any core that was unknown at design-time. Given the timing requirements of the run-time scenario, a polynomial-time greedy approach has been chosen. Because of its greedy nature, the proposed algorithm may not find a global optimum, or it may not even find a feasible solution though a solution actually exists, but it looks for a good solution by performing a best-effort minimization of the two objective functions.

The algorithm is logically divided into two phases: in the first one, it finds a combination of existing configurations, and in the second one it applies bitstream relocation. The details of each phase are discussed in the next section.

Algorithm 1 Score-based selection

```
$Unmapped \leftarrow New\ application\ cores$
$Available \leftarrow S_{DTM}$
$Selected \leftarrow \emptyset$
**repeat**
  $Scores \leftarrow Compute\_Scores(Available, Unmapped)$
  $Candidate \leftarrow Get\_Higher\_Score(Scores)$
  $Selected \leftarrow Selected \cup Candidate$
  $Available \leftarrow Available \setminus Candidate$
  $Unmapped \leftarrow Unmapped \setminus Cores\_In(Candidate)$
**until** $(Unmapped = \emptyset) \vee (No\ Free\ Slot)$
```

1) *Selection Phase:* The first phase aims at finding a set of configurations that includes all the cores of the incoming application. We propose a *score-based* technique [7], which aims at finding a combination of configurations that guarantees a good quality in terms of reconfigurations and communication. The score-based selection procedure is shown in Algorithm 1.

The algorithm selects a configuration at each iteration, thus the number of iterations is bounded by the number of slots of the device. The selection is performed according to a score associated with each configuration, which is computed at each iteration for all the configurations i as follows:

$$Score_i = \zeta \cdot \frac{Useful\_Area}{Slot\_Size} + (1-\zeta) \cdot \frac{Internal\_Comm}{Max\_Comm}. \quad (10)$$

The $Useful\_Area$ term is defined as the amount of area of a configuration occupied by the cores required by the incoming application, and gets a higher value as the new application and the existing ones are more similar to each other. This term indirectly reduces the average number of reconfigurations, as configurations with a high percentage of occupied area are preferred, thus the solution remains compact. Conversely, $Internal\_Comm$ represents the bandwidth among the cores inside the configuration, and helps reducing the communication overhead, as the algorithm focuses on configurations that resolve large amounts of communication within the slot, without accessing the global NoC. The ζ parameter ranges from 0 to 1, and can be tuned to favor the number of reconfigurations (ζ close to 1) or the communication overhead (ζ close to 0); according to our experience, a value close to 0.6 provides the most balanced tradeoff among the two metrics. It should be noted that, as the configurations are selected, the corresponding cores are considered as mapped and do not contribute to the score computation anymore, which allows the algorithm to focus only on the unmapped cores and to increases the probability that a feasible solution will be found.

2) *Mapping Phase:* The mapping phase assigns each selected configuration to a specific slot, and its computational steps are summarized in Algorithm 2. To simplify the assignment while still reducing the average number of reconfigurations, we decided to constrain the configurations of the base mapping (if any) into their original slots.

First, the algorithm identifies the configurations that are more likely to affect the communication overhead and the average number of reconfigurations, so they can be handled first because the algorithm can select the target slot among a large number of alternatives. In particular, the *criticality* of

Algorithm 2 RTM: Configuration reuse

```
$Conf \leftarrow Selection\_Phase(S_{DTM})$
$S_{RTM-CR} \leftarrow In\_Base\_Mapping(Conf)$
$Conf \leftarrow Conf \setminus In\_Base\_Mapping(Conf)$
**repeat**
  $Candidate \leftarrow Get\_Critical\_Configuration(Conf)$
  **for all** $i \in S_{RTM-CR}$ **do**
    $Comm \leftarrow Compute\_Communitation(Candidate, i)$
    $Scores \leftarrow Propagate\_Communitation(Comm)$
  **end for**
  $Target\_Slot \leftarrow Get\_Highest_{S}core(Scores)$
  $Candidate \leftarrow Relocate(Candidate, Target\_Slot)$
  $S_{RTM-CR} \leftarrow S_{RTM-CR} \cup Candidate$
  $Conf \leftarrow Conf \setminus Candidate$
**until** $Conf = \emptyset$
```

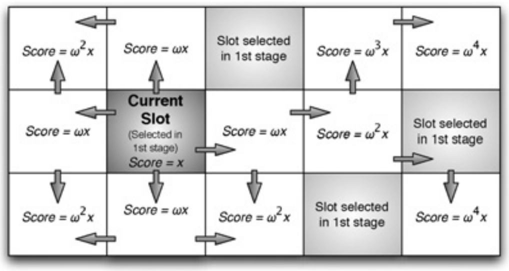

Fig. 3. Example of the propagation technique.

each configuration is computed as a combination of two terms as follows.

a) *Communication*, i.e., the amount of traffic on the global network generated by the cores in the configuration.
b) *Reuse degree*, i.e., the number of already-mapped applications that use the configuration in its original location.

The algorithm then considers one configuration at a time according to its criticality, and finds the best target slot according to the communication requirements between the current configuration and the already-mapped ones, if any. The target slot is computed using a technique called *propagation*, which is shown in Fig. 3. For each slot that has already been assigned, the communication between the cores in that slot and the ones in the current configuration is computed according to the CG. The resulting value is then propagated to all the other slots, but the value is reduced by a factor $\omega \leq 1$ for each hop. Each occupied slot propagates its own value over the mesh grid, and all the values are summed; at the end, the slot with the highest value is close to all the already-allocated slots that frequently communicate with the configuration being mapped.

Finally, the score computed using the propagation technique is corrected to keep the number of reconfigurations into account. The algorithm computes two values as follows:

a) the *reuse degree*, which is the number of applications that use the configuration in its original location;
b) the *reconfiguration* metric, defined in Section V-B2, which counts the number of times the slot is reconfigured while switching from an application to another.

A normalized value of the *reuse degree* is added to the score associated with the original slot of the current configuration to avoid relocation when a configuration is frequently configured in its original location. Also, the normalized value of the *reconfiguration* metric is added to the score of all the unused slots, in order to avoid the reconfiguration of a slot that is rarely overwritten. The slot with the highest score is the best option in terms of communication and reconfiguration issues, thus it is selected to map the current configuration.

B. SAT-Based Selection for Configuration Reuse

As the score-based selection may not find a combination of configurations that includes all the cores of the new application, we now propose an alternative selection method that finds a feasible solution every time it exists. For this purpose, we formulate RTM-CR as a SAT problem, and we consider a class of solvers, known as *complete solvers*, which guarantee that a satisfying assignment will be found if and only if it exists, otherwise the formula is declared as unsatisfiable [29].

We can convert the RTM-CR problem into a single Boolean formula by introducing two kinds of variables.

1) *Core variables*, which represent an instance of a core in a configuration. Given a configuration j, we define a Boolean variable x_{ij} for each core i used by the incoming application and included in the configuration. The variable is true if the instance of core i used in the final solution is the one belonging to configuration j.
2) *Configuration variables*, which are associated with each configuration. For each configuration j in the system, we introduce a variable z_j, which takes the true value if j is selected to be part of the solution.

The variables are then constrained to guarantee the coherence of the model. In particular, we introduce two sets of clauses as follows.

1) *Instance constraints*, which guarantee that each core is mapped in the solution. For each core i required by the incoming application, a new clause is added to the Boolean formula to ensure that, given the set of configurations j that include core i, at least one variable x_{ij} is true.
2) *Inclusion constraints*, which bind core instances to slot configurations. Given a configuration j, a set of clauses including z_j and all the corresponding x_{ij} variables is added, in order to say that if configuration j is selected (i.e., z_j is true), then each core i included in j may or may not be selected. Conversely, if j is not selected, then no variable x_{ij} can be true.

The Boolean formula is then processed by *PrecoSAT* [30], one of the fastest SAT solvers available. If a solution exists, *PrecoSAT* will find it and it will return a satisfying assignment, which is then used to build the solution; each configuration variable z_j assigned to true represents a configuration that is part of the solution, and each core variable x_{ij} that is true means that core i is used in configuration j. The Boolean formula allows a core to appear in multiple configurations, therefore a post-processing of the results may be necessary to select the best instance in terms of communication overhead; this can be performed using an exhaustive search, which is feasible in this situation because of the small solution space.

If bitstream relocation is not supported, the selection of two configurations of the same slot should be forbidden. This requires the introduction of a third set of constraints known as *mutual exclusion constraints*; given a set of configurations

Algorithm 3 RTM: General case

```
// 1) Configuration Reuse
Cores ← New application cores
Conf ← $S_{DTM}$ ; $S_{RTM-G}$ ← ∅
repeat
    Scores ← Compute_Scores(Conf, Cores)
    Candidate ← Get_Higher_Score(Scores)
    if Evaluate(Candidate) ≤ Threshold then
        Exit_Condition ← True
    else
        $S_{RTM-G}$ ← $S_{RTM-G}$ ∪ Candidate
        Conf ← Conf \ Candidate
        Cores ← Cores \ Cores_In(Candidate)
        Exit_Condition ← False
    end if
until Exit_Condition = true
// 2) Sorting
Unmapped ← Sort(Cores, $S_{RTM-G}$)
// 3) Mapping
repeat
    Candidate ← Get_Critical_Element(Unmapped)
    for all $i \in S_{RTM-G}$ do
        Comm ← Compute_Communitation(Core, i)
        Scores ← Propagate_Communitation(Comm, i)
    end for
    Scores ← Correct_Scores(Scores, $S_{DTM}$)
    Target ← Get_Highest_score(Scores)
    $S_{RTM-G}$ ← $S_{RTM-G}$ ∪ Map(Candidate, Target)
    Unmapped ← Unmapped \ Candidate
until Unmapped = ∅
```

represented by variables z_j and referring to the same slot, the clauses make the formula true when at most one z_j is true.

C. RTM in the General Case

We now define a general-case run-time mapper (RTM-G), which is used when at least one unknown core is introduced by the new application, or when a solution cannot be found with the configuration reuse approach, or when such solution is not satisfying in terms of communication or number of reconfigurations. A bitstream generation phase is always required in this scenario, therefore the number of new bitstreams becomes a critical metric as it affects the deployment time.

We tackle the RTM-G problem by means of a multistage, polynomial-time heuristic technique we initially introduced in [8], which is extended to include bitstream relocation, and whose computational steps are summarized in Algorithm 3.

1) *Configuration Reuse:* The first stage of the algorithm exploits configuration reuse to build part of the solution, which does not need the generation of any new bitstream. However, a high number of reused configurations may in the worst case prevent the algorithm from finding a solution in later stages, because they may contain cores that are not used by the incoming application, thus leading to a waste of area. A termination condition is then required to stop the reuse procedure before it can affect the feasibility or the quality of the solution. At the end of this stage, the configurations belonging to the base mapping are considered as fixed and are not relocated, as discussed in Section VI-A2.

Intuitively, the termination condition is related to the amount of area left on the device and to the size the cores that are still unmapped; when the unmapped cores may not fit on the device, the reusing procedure is stopped. We formalize this idea by computing a threshold that is related to two elements. The first one is the ratio between the area required to map the remaining cores and the available area; a high value means that it will be difficult to map the remaining cores if the candidate configuration is selected. The second one is the number of iterations; the reuse of a configuration in the early iterations is not likely to affect the feasibility of the final solution, while the algorithm should be more careful in later iterations. Thus, the candidate configuration i is selected at iteration j if

$$Score_i \geq \mu \cdot \left(\frac{Required\_Area}{Availabe\_Area} + \frac{j}{\#\_of\_Slots} \right). \quad (11)$$

The length of the first stage can be tuned by modifying the value of parameter μ, whose typical values range from 0 to 10. The higher is the value (e.g., 8 or more), the lower is the number of bitstreams that are reused.

2) *Sorting:* As discussed in Section VI-A2 for the reused configurations, the impact of each core on the final solution is not uniform. The algorithm detects the most critical cores by considering both area and communication aspects. In particular, larger cores should be mapped first, because they cannot easily fit into partially occupied slots. Further, the cores that generate a large bandwidth are considered early, because they can be assigned to slots that are close to the already-mapped cores that frequently communicate with them.

Since bitstream relocation is supported in the proposed algorithm, and only the configurations of the base mapping are considered as fixed after the first stage, the remaining reused configurations should be sorted along with the unmapped cores. However, the total amount of area inside a configuration is going to dominate the area of a single core, thus the average size of the cores inside a configuration is used to compute the sorting metric, along with the overall traffic sent by the cores in the configuration toward the global network.

3) *Mapping:* The third stage maps the remaining cores using the propagation technique discussed in Section VI-A2, though in this case single cores are considered along with complete configurations.

VII. Real-World Case Study

We propose a real-world case study related to multimedia (audio/video) compression and decompression, in order to motivate the proposed approach and to show its benefits. We consider a set of video coders and decoders, which share a set of cores to perform some common signal processing operations. In particular, we include two legacy standards, *MPEG-2* [31], [32] and *H.263* [33], and the more recent *MPEG-4* [34], [35] format. The codecs are supposed to be configured on an FPGA device (in this example, we opted for a Xilinx XC4VLX60) to encode or decode a video stream with one of the supported formats, but they cannot fit on the device at the same time because of area and power-related issues. Therefore, the required standard can be loaded at run-time by exploiting dynamic reconfiguration, and by taking advantage of the similarities between the codecs to reduce the deployment time.

The reduction of the deployment time is critical in certain scenarios, e.g., when the user is searching for an interesting stream. In this situation, the user typically switches from

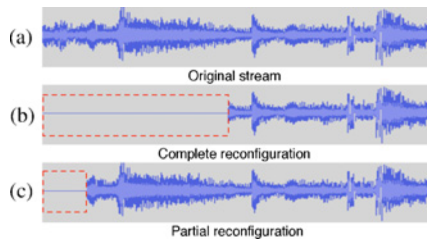

Fig. 4. Impact of (c) partial and (b) complete reconfiguration on a (a) stream of 3 s.

one source to another very quickly, as it takes him just 3–4 s to determine whether the current stream is interesting or not. Fig. 4(a) shows the amount of information that the user perceives when the audio part of the stream is played for 3 s. A complete reconfiguration of the target device requires 1488 ms, introducing a significant delay before the stream is played (Fig. 4(b)), which can be greatly reduced by exploiting the potential of a partial reconfiguration.

We identify two scenarios to validate the proposed approach. In the first one, all the codecs are known when the system is synthesized, and we apply our design-time mapper over all the coders and encoders. In the second one, only the two oldest cores, namely, *MPEG-2* and *H.263*, are all known at design time, while the new *MPEG-4* format is added at run-time, and its cores are handled by the proposed run-time mapper. This scenario simulates a common real-world situation, as standards are continuously created or updated, and it is important for a multimedia platform to remain up-to-date.

A. *Case Study Overview*

We generated the CGs of the selected codecs according to the standard specifications and the related works we found in literature. The starting point to build a CG is a modular description of the codec, which is generally provided as a data flow diagram. Then, some general-purpose units—i.e., a reconfigurable instruction-set processor and a memory unit—are included to perform data acquisition and other general-purpose operations. We show the resulting CGs of the encoders (the corresponding decoders are very similar and are omitted here) in Fig. 5. Some cores are shared among different codecs, such as the *discrete cosine transform* (DCT), the *quantizer* (Q), their inverse operations (IDCT and IQ), or the *Huffman coder*. The weights of the edges in the CGs have been assigned according to the criticality of the communication. In Fig. 5, critical edges are represented by solid lines, whereas the non-critical ones are depicted using dashed lines. The solution should map the cores involved in a critical communication close to each others to maximize the throughput of the system.

We estimate the area requirements of the cores in order to determine the number of slots, and the slot size. In Fig. 5, we provide a visual idea of the relative size of each core, which is proportional to the dimension of the corresponding node in the graph. We can divide the target FPGA into six slots of approximately 4400 slices each, so that the slot is large enough to contain a compact version of the motion compensation block, which is the largest core in the system.

B. *Design-Time Mapping*

In the first scenario, we consider the entire set of codecs and we map them using the design-time mapper. The output of the execution is presented in Fig. 6, where the slots delimited by a double stroke represent a configuration belonging to the base mapping, the ones delimited by a single stroke are specific configurations. A transition between two applications requires 1.53 reconfigurations on average (corresponding to a delay of 380 ms), with respect to the six reconfigurations required to modify the whole FPGA area. Therefore, the proposed approach provides a 74.5% improvement with respect to a complete reconfiguration (which requires 1488 ms), and it reduces the transition time to an acceptable level, as shown in Fig. 4(c). It is important to notice that the mapper achieved this result by isolating cores that are used more frequently (such as the CPU, the storage memory, the motion compensation), so that they are never overwritten, and by focusing the dynamic reconfiguration over only two slots.

The distribution of the communication is also coherent with the expected behavior of the mapper. For instance, the average number of hops that are necessary to deliver all the critical communications is equal to 0.8 for the *MPEG-4* encoder, which is the most complex application in the system. The value is very competitive, because in the proposed architecture a message may require as many as three hops to reach its destination. Conversely, the average number of hops for non-critical communications is 1.4, which again shows the ability of the proposed approach to keep communication into account.

C. *Run-Time Mapping*

In the second scenario, only the two legacy codecs (i.e., *MPEG-2* and *H.263*) are mapped at design time, whereas the *MPEG-4* applications are added later. As the *MPEG-4* codec introduces new cores with respect to the previous ones, the solution cannot be found using configuration reuse only, so the general-case run-time mapper must be used. For sake of simplicity, we show the mapping of the *MPEG-4* encoder only.

Fig. 7(a) shows the base mapping generated for the *MPEG-2* and *H.263* encoders and decoders. The two decoders are fully deployed in slots 1, 3, 4, 5, and 6 of the base mapping, therefore no reconfiguration is required to switch between them. Conversely, each of the two encoders needs to reconfigure slot 6 to map the *quantizer*, the DCT block, and, for the *MPEG-2* only, the *rate control* core.

The *MPEG-4* encoder introduces four new cores, which do not fit into a single slot, so at least two new bitstreams have to be generated. As shown in Fig. 7(b), the run-time mapper reuses four configurations belonging to the base mapping, namely, slots 1, 2, 3, and 5, and to generate two specific configurations. In the process, the mapper decides not to reuse the configuration of slot 4, though the IQ and the IDCT cores are required by the encoder, because otherwise it cannot map the remaining cores on slot 6 only. As a result, the *MPEG-4* encoder is deployed by reconfiguring only two slots, which takes approximately 496 ms with respect to the 1488 ms required by a complete reconfiguration of the device, corresponding to a 66.7% improvement. Furthermore, only two new partial bitstreams have to be generated, thus

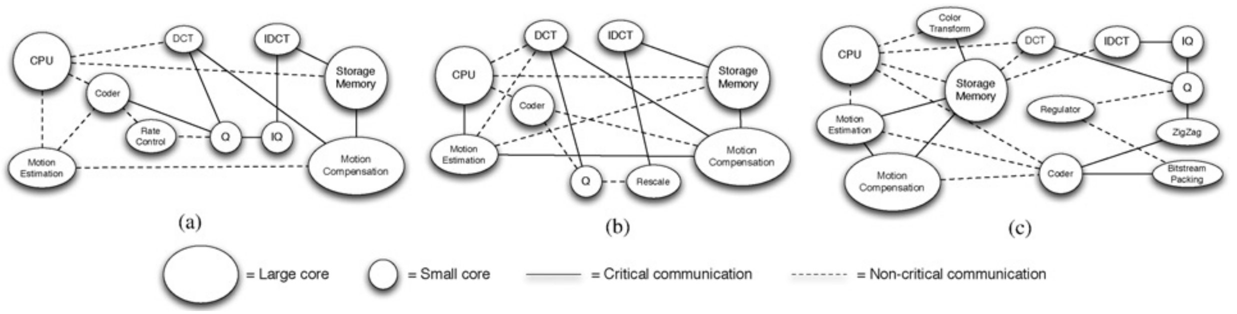

Fig. 5. Communication graphs for the encoder applications. (a) MPEG-2. (b) H.263. (c) MPEG-4.

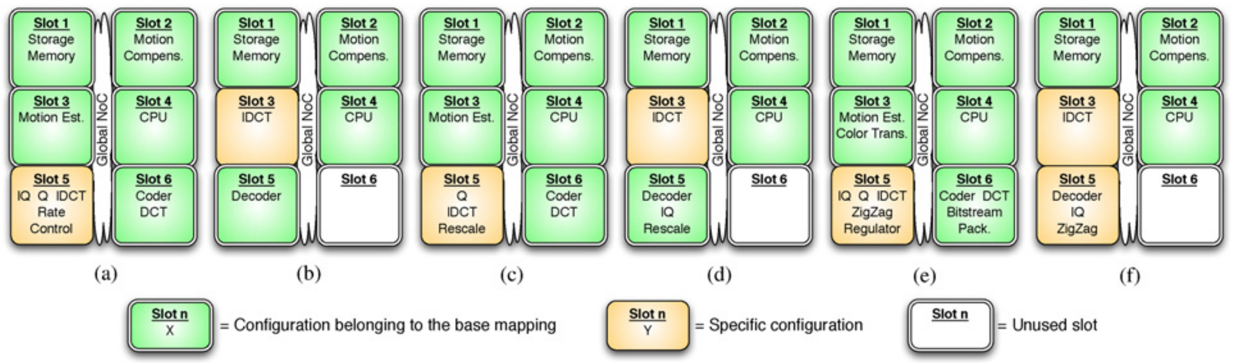
(a) Slot 1 Storage Memory; Slot 2 Motion Compens.; Slot 3 Motion Est.; Slot 4 CPU; Slot 5 IQ Q IDCT Rate Control; Slot 6 Coder DCT
(b) Slot 1 Storage Memory; Slot 2 Motion Compens.; Slot 3 IDCT; Slot 4 CPU; Slot 5 Decoder; Slot 6
(c) Slot 1 Storage Memory; Slot 2 Motion Compens.; Slot 3 Motion Est.; Slot 4 CPU; Slot 5 Q IDCT Rescale; Slot 6 Coder DCT
(d) Slot 1 Storage Memory; Slot 2 Motion Compens.; Slot 3 IDCT; Slot 4 CPU; Slot 5 Decoder IQ Rescale; Slot 6
(e) Slot 1 Storage Memory; Slot 2 Motion Compens.; Slot 3 Motion Est. Color Trans.; Slot 4 CPU; Slot 5 IQ Q IDCT ZigZag Regulator; Slot 6 Coder DCT Bitstream Pack.
(f) Slot 1 Storage Memory; Slot 2 Motion Compens.; Slot 3 IDCT; Slot 4 CPU; Slot 5 Decoder IQ ZigZag; Slot 6
Global NoC
Slot n X = Configuration belonging to the base mapping
Slot n Y = Specific configuration
Slot n = Unused slot

Fig. 6. DTM of the multimedia applications. (a), (b) MPEG-2 encoder and decoder. (c), (d) H.263 encoder and decoder. (e), (f) MPEG-4 encoder and decoder.

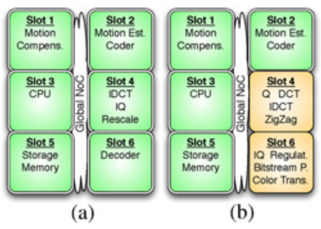
(a) Slot 1 Motion Compens.; Slot 2 Motion Est. Coder; Slot 3 CPU; Slot 4 IDCT IQ Rescale; Slot 5 Storage Memory; Slot 6 Decoder
(b) Slot 1 Motion Compens.; Slot 2 Motion Est. Coder; Slot 3 CPU; Slot 4 Q DCT IDCT ZigZag; Slot 5 Storage Memory; Slot 6 IQ Regulat. Bitstream P. Color Trans.
Global NoC

Fig. 7. RTM of the (b) MPEG-4 encoder application, and corresponding (a) base mapping.

the synthesis phase is heavily simplified with respect to a design for the whole device. As expected, the reuse of existing configurations constrains the run-time mapper in terms of communication overhead, and the average number of hops for critical communication is 1.2 (non-critical communication is also delivered in 1.2 hops). As a conclusion, the results achieved by the design-time mapper over all the applications is 33.3% better in terms of communication, though the run-time algorithm performs well in terms of number of reconfigurations and computes the solution in a much shorter time, as we will show in the next section on a wide set of benchmarks.

VIII. Experimental Results

In this section, we evaluate the proposed mapping flow on a large set of synthetic benchmarks. The goal is to provide a more detailed profiling of its performance, and to show how it scales on larger and more complex applications.

A. Experimental Setup

The experimental results presented in this section, unless otherwise specified, are performed on a hardware architecture divided into 16 slots, which are interconnected using a NoC backbone with a mesh topology. The target FPGA is a Xilinx XC4VLX40, which provides 18 432 slices and supports fine-grained dynamic reconfiguration and bitstream relocation [24]. The estimated reconfiguration time for each slot is 64 ms.

The benchmark applications are larger than the ones of the case study discussed in the previous section, as they contain from 25 to 35 cores of different size and they model a set of highly parallel applications. Each application picks approximately 75% of its cores (unless otherwise specified) from a shared set of cores; this percentage is a conservative approximation of the real amount of shared cores presented in the case study. A set of six to eight applications is assumed to be known at design time and, during the evaluation of the run-time mappers, an additional application is added at run-time.

B. Execution Time

We first evaluate the execution times of the different components of our flow, which are illustrated in Fig. 8, and which have been measured on an Intel Core 2 Duo processor running at 2.8 GHz. An execution of the design-time approach is almost 40 times slower than any component of the run-time mapper, and in general its execution should be iterated multiple times (in practice, tens of times) to get a high-quality solution. Any component of the run-time mapper takes a few tens of milliseconds to complete, and in particular the general-case mapper with relocation support, which is the most complex one, requires 32 ms. Therefore, it is possible to execute multiple algorithms sequentially (e.g., to explore the feasibility of a solution based on configuration reuse with both the score-based and the SAT-based approaches and, if a solution is not found, to build a mapping using the general-case algorithm) in less than a hundred of milliseconds.

C. Evaluation of the Design-Time Mapper

The proposed design-time mapper can be tuned to achieve different tradeoffs between average number of reconfigurations

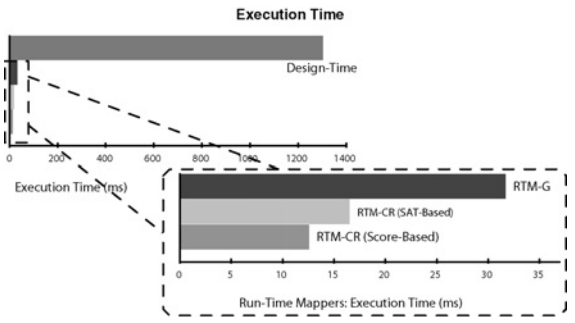

Fig. 8. Execution time of design-time and run-time mappers.

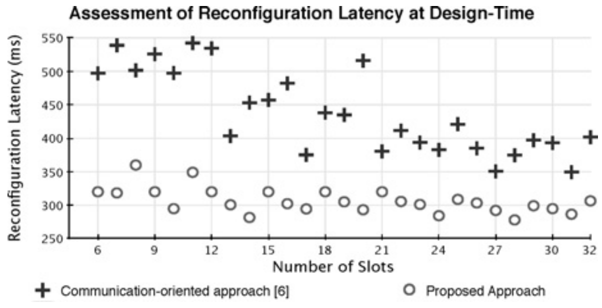

Fig. 9. Performance of the design-time mapper.

and communication overhead. The tuning can be achieved by modifying some of the parameters that affect the selection and the partitioning phases, namely, α and β. Their impact on the algorithm performance has been investigated in [6]. A high value of α—such as 0.4 or higher—leads to a good solution in terms of communication overhead, while a high value of β—such as 0.7 or higher—forces more cores in the base mapping, thus the number of reconfigurations decreases.

We compare the design-time mapper with a communication-oriented approach that maps the cores on the nodes of a NoC to optimize the communication overhead, but does not explicitly consider the reconfiguration costs [6]. A comparison with respect to the few state-of-the-art approaches that consider dynamic reconfiguration [14] is not possible, mainly because they focus on a single application, but also because they rely on a hardware architecture that is not comparable to the one we introduced as a contribution of our work. As shown in Fig. 9, the average reconfiguration latency to switch from an application to another is reduced by 15.9% to 43.2% (29.1% in the average), and it is generally acceptable for most of the application scenarios. This reduction is due to the ability to detect similarities among the applications, and the gap is wider when the number of slots of the system is small, as the proposed approach takes advantage from the packing of shared cores in the same slot. For instance, the proposed approach reduces the reconfiguration latency from 482.2 ms to 302.2 ms (37.3% of reduction) in a 16-slots system, where a complete reconfiguration requires more than 1 s.

D. Evaluation of the Run-Time Mapper

The proposed run-time mapper bases its efficiency on the reuse—either complete or partial—of existing configurations, which heavily reduces the deployment time of the new application, but may also affect the quality and the feasibility of the solution. The effects of the reuse policy on the final solution have been discussed in [8]; a high number or reused slots has been proved to increase the communication overhead, whereas the average number of reconfigurations is reduced. As discussed in Section VI-C, the length of the reuse phase can be tuned by modifying the value of μ in (11). In this paper, we focus on a balanced tuning (i.e., μ equal to 5) to achieve a good tradeoff between the metrics [8].

We first evaluate the proposed run-time flow when all the cores in the new application were known at design-time, so the approaches based on configuration reuse can be included in the comparison. We use the design-time mapper as a reference because the few related works [16]–[18] tackle completely different aspects of run-time addition of new applications; they focus on geometric issues (such as area fragmentation [18]) to maximize the number of applications that can fit on the FPGA at the same time. We run our reference over the set of applications known at design-time plus the incoming one; this approach should exploit similarities in a more efficient way, and it is expected to improve the quality of the solution.

Fig. 10 shows a comparison between the run-time and the design-time mappers in terms of average number of reconfigurations and communication overhead. The general-case algorithm (RTM-G) outperforms the approaches based on configuration reuse (with either a score-based or a SAT-based selection technique) in both the metrics, though the deployment time of the new application is higher, as it requires the generation of 3.5 new bitstreams on average (2.5 new bitstreams are required by the design-time mapper, instead). The results also show that the RTM-G algorithm excels in terms of average number of reconfigurations, because of the reuse of some frequently configured cores combined with the generation of a low number of new configurations.

As expected, the static mapper outperforms all the run-time mappers in terms of overall communication overhead and average number of reconfigurations. However, the design-time approach aims at finding the optimum for a set of applications as a whole, which may not correspond to the best mapping for each single application. Conversely, the run-time algorithm only focuses on the single application and looks for a local optimum, which in turn does not guarantee that a global optimum is achieved over the entire application set. Fig. 10 shows that the RTM-G algorithm can even find a better solution for the incoming application only and in a shorter time, which proves that the proposed run-time algorithm is best suited for incremental addition of new applications.

The decision to include bitstream relocation in the proposed flow has produced two complementary effects. On the one hand, the communication overhead is reduced both in the general-case and in the configuration reuse algorithms, because existing configurations can be assigned to different target slots according to communication needs. On the other hand, bitstream relocation leads to a higher number of reconfigurations, due to the misplacement of the relocated bitstreams. Experimental results show that a general-case run-time mapper without relocation support reduces the average reconfiguration latency by approximately 30 ms, whereas the communication overhead increases by 13.4% on average. Thus, according to the system requirements, the designer may decide not to support relocation to further reduce the reconfiguration latency.

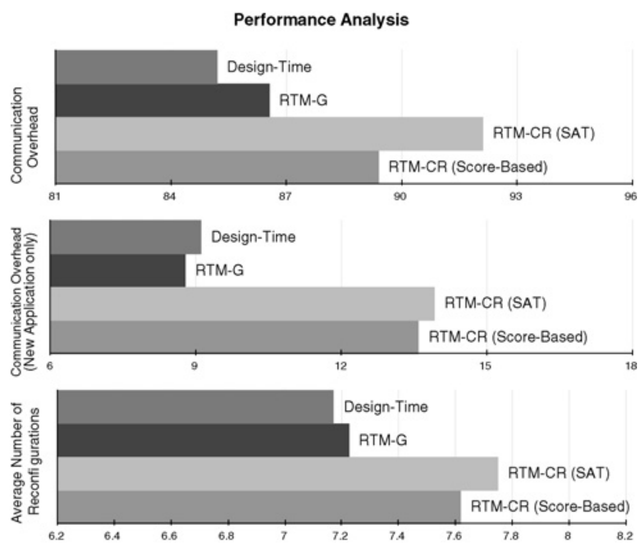

Fig. 10. Performance of the run-time mapper.

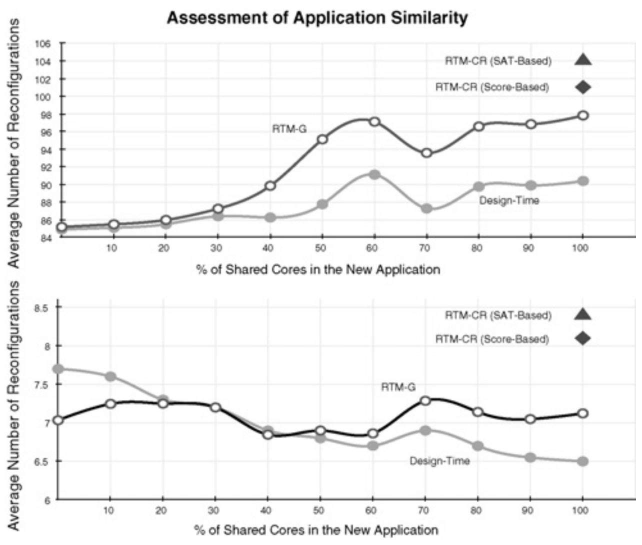

Fig. 11. Results of the run-time mapper with respect to the percentage of shared cores.

E. *Effects of Application Similarities*

As the general-case run-time algorithm can handle applications that introduce new cores, we show how the similarity between the new application and the existing ones can impact the final solution. In particular, Fig. 11 shows how communication overhead and average number of reconfigurations are affected by the percentage of cores that the new application shares with the existing ones, and the design-time mapper is again used as a reference. The design-time mapper improves its performance if the incoming application is similar (more than 60% of shared cores) to the ones mapped at design-time, because it can enforce core reuse more efficiently. Still, the general-case algorithm proves to be very competitive both in terms of communication (with an average gap of just 4.7%), and reconfiguration (the average gap is lower than 7%), since it exploits configurations that are likely to be configured on the device, and adds a minimum amount of specific configurations. Fig. 11 also shows the results of the approaches based on configuration reuse when all the cores of the new application were known at design-time (100% of shared cores), reinforcing the idea that these techniques provide a worse solution, though the application is immediately available.

F. *Evaluation of the SAT-Based Approach*

We introduced the SAT-based selection as a different technique to identify a set of configurations that maps the entire new application using existing configurations only. We evaluated the benefits of this approach with respect to the score-based approach, which aims at finding a good-quality solution, over a set of 300 benchmarks that do not introduce any core that was unknown at design-time. The general-case mapper has not been compared, as it can stop the reuse policy and it always finds a feasible solution.

The SAT-based approach increases the number of solutions that are found by the algorithm by nearly 6% and, since it is a complete solver, the number of feasible solutions found by the SAT-based algorithm is actually the number of feasible solutions over the set of benchmarks. On the contrary, the results show that the score-based approach can still identify approximately 94% of the feasible solutions, which is remarkable considering its greedy nature.

Finally, the support of relocation we included in our flow positively affects the number of solutions found by the algorithms. According to the results of the SAT-based approach, only 74% of the solutions of the previous experiment are still feasible if relocation is not supported. In this scenario, the score-based approach without relocation can identify 86% of those feasible solutions, as it may commit wrong decisions in the selection of different configurations of the same slot.

IX. Conclusion

In this paper, we have presented a novel mapping flow for multi-core applications, tailored for dynamically reconfigurable devices. Unlike the existing approaches, the proposed flow aims at optimizing both reconfiguration-related metrics and on-chip performance, either at design-time or at run-time.

We have applied the proposed flow to a real-world multimedia case study, achieving a 74.5% improvement (66.7% at run-time) in terms of reconfiguration latency, while satisfying the communication requirements among the cores. Additional results have shown that the design-time mapper can effectively handle both communication and reconfiguration-related metrics (with an improvement of 37.3% over a communication-oriented algorithm [6]). Furthermore, we have shown that the proposed run-time mapper can find a good-quality solution in a very short time (40 times faster than a design-time approach), while keeping the number of new bitstreams low.

References

[1] R. Porto, M. Porto, S. Bampi, and L. Agostini, "High throughput architecture for forward transforms module of H.264/AVC video coding standard," in *Proc. Electron., Circuits Syst.*, 2007, pp. 150–153.
[2] D. Pham, "Key considerations given to the design of a next generation multi-core communications platform," in *Proc. ICICDT*, 2008, pp. 253–256.
[3] L. Benini and G. De Micheli, "Networks on chips: A new SoC paradigm," *Comput.*, vol. 35, no. 1, pp. 70–78, Jan. 2002.
[4] E. Flamand, "Strategic directions towards multicore application specific computing," in *Proc. DATE*, 2009, p. 1266.
[5] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto NoC architectures," in *Proc. DATE*, 2004, pp. 896–901.
[6] V. Rana, S. Murali, D. Atienza, M. D. Santambrogio, L. Benini, and D. Sciuto, "Minimization of the reconfiguration latency for the mapping of applications on FPGA-based systems," in *Proc. CODES-ISSS*, 2009, pp. 120–130.

[7] I. Beretta, V. Rana, D. Atienza, M. D. Santambrogio, and D. Sciuto, "Run-time mapping for dynamically-added applications in reconfigurable embedded systems," in *Proc. ICM*, 2009, pp. 157–160.
[8] I. Beretta, V. Rana, D. Atienza, and D. Sciuto, "Run-time mapping of applications on FPGA-based reconfigurable systems," in *Proc. ISCAS*, 2010, pp. 3329–3332.
[9] S. Corbetta, M. Morandi, M. Novati, M. Santambrogio, and D. Sciuto, "Two novel approaches to online partial bitstream relocation in a dynamically reconfigurable system," in *Proc. ISVLSI*, 2007, pp. 457–458.
[10] X. Huang and T. Wolf, "Evaluating dynamic task mapping in network processor runtime systems," *IEEE Trans. Parallel Distribut. Syst.*, vol. 19, no. 8, pp. 1086–1098, Aug. 2008.
[11] T. Wolf and N. Weng, "Runtime support for multicore packet processing systems," *IEEE Netw.*, vol. 21, no. 4, pp. 29–37, Jul. 2007.
[12] S. Murali, M. Coenen, A. Radulescu, K. Goossens, and G. De Micheli, "A methodology for mapping multiple use-cases onto networks on chips," in *Proc. DATE*, Mar. 2006, pp. 1–6.
[13] A. Hansson, K. Goossens, and A. Rădulescu, "A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic," *Very Large Scale Integr. Des.*, vol. 2007, no. 68432, p. 16, 2007.
[14] S. Ghiasi, A. Nahapetian, and M. Sarrafzadeh, "An optimal algorithm for minimizing run-time reconfiguration delay," *ACM Trans. Embedded Comput. Syst.*, vol. 3, no. 2, pp. 237–256, May 2004.
[15] K. Bruneel, F. Abouelella, and D. Stroobandt, "Automatically mapping applications to a self-reconfiguring platform," in *Proc. DATE*, 2009, pp. 964–969.
[16] C.-L. Chou, U. Y. Ogras, and R. Marculescu, "Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels," *IEEE Trans. Comput.-Aided Des.*, vol. 27, no. 10, pp. 1866–1879, Oct. 2008.
[17] C.-L. Chou and R. Marculescu, "Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip," *IEEE Trans. Comput.-Aided Des.*, vol. 29, no. 1, pp. 78–91, Jan. 2010.
[18] Y. Lu, T. Marconi, G. Gaydadjiev, K. Bertels, and R. Meeuws, "A self-adaptive on-line task placement algorithm for partially reconfigurable systems," in *Proc. IPDPS*, Apr. 2008, pp. 1–8.
[19] J. M. P. Cardoso, "On combining temporal partitioning and sharing of functional units in compilation for reconfigurable architectures," *IEEE Trans. Comput.*, vol. 52, no. 10, pp. 1362–1375, Oct. 2003.
[20] *Early Access Partial Reconfiguration User Guide*, Xilinx, Inc., San Jose, CA, Mar. 2006.
[21] V. Rana, D. Atienza, M. D. Santambrogio, D. Sciuto, and G. De Micheli, "A reconfigurable network-on-chip architecture for optimal multi-processor SoC communication," in *Proc. VLSI-SOC*, vol. 1. 2009, pp. 1–20.
[22] V. Rana, M. D. Santambrogio, and A. Meroni, "Design methodologies and mapping algorithms for reconfigurable NoC-based systems," in *Dynamic Reconfigurable Network-on-Chip Design: Innovations for Computational Processing and Communication*. Hershey, PA: IGI Global, 2010, pp. 110–134.
[23] D. Cozzi, C. Farè, A. Meroni, V. Rana, M. D. Santambrogio, and D. Sciuto, "Reconfigurable NoC design flow for multiple applications run-time mapping on FPGA devices," in *Proc. GLSVLSI*, 2009, pp. 421–424.
[24] S. Corbetta, M. Morandi, M. Novati, M. D. Santambrogio, D. Sciuto, and P. Spoletini, "Internal and external bitstream relocation for partial dynamic reconfiguration," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 17, no. 11, pp. 1650–1654, Nov. 2009.
[25] J. Hu and R. Marculescu, "Energy-aware mapping for tile-based NoC architectures under performance constraints," in *Proc. ASP-DAC*, Jan. 2003, pp. 233–239.
[26] R. M. Karp, "Reducibility among combinatorial problems," in *Complexity of Computer Computations*. New York: Plenum Press, 1972, pp. 85–103.
[27] S. A. Cook, "The complexity of theorem-proving procedures," in *Proc. STOC*, 1971, pp. 151–158.
[28] B. Hendrickson and R. Leland, "A multilevel algorithm for partitioning graphs," in *Proc. SC*, 1995, p. 28.
[29] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," in *Proc. DAC*, 2001, pp. 530–535.
[30] M. Järvisalo, A. Biere, and M. Heule, "Blocked clause elimination," in *Proc. TACAS*, 2010, pp. 129–144.
[31] K. Kim and J.-S. Koh, "An area efficient DCT architecture for MPEG-2 video encoder," *IEEE Trans. Consumer Electron.*, vol. 45, no. 1, pp. 62–67, Feb. 1999.
[32] M. Verderber, A. Zemva, and A. Trost, "HW/SW codesign of the MPEG-2 video decoder," in *Proc. PIPDPS*, 2003, p. 7.
[33] M. J. Garrido, C. Sanz, M. Jimenez, and J. M. Menesses, "An FPGA implementation of a flexible architecture for H.263 video coding," *IEEE Trans. Consumer Electron.*, vol. 48, no. 4, pp. 1056–1066, Nov. 2002.
[34] E. B. V. D. Tol and E. G. T. Jaspers, "Mapping of MPEG-4 decoding on a flexible architecture platform," *Proc. SPIE: Media Process.*, vol. 4674, pp. 1–13, Jan. 2002.
[35] F. Casalino, G. di Cagno, and R. Luca, "MPEG-4 video decoder optimization," in *Proc. ICMCS*, Jul. 1999, pp. 363–368.

Ivan Beretta received the M.S. degree in computer science from the University of Illinois at Chicago, Chicago, in 2008, and the Laurea degree in computer engineering from the Politecnico di Milano, Milan, Italy, in 2009. He is currently a Ph.D. student with the Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland.

His current research interests include computer-aided design methodologies for high-performance embedded systems, and runtime adaptability in reconfigurable hardware and wireless sensor networks.

Vincenzo Rana received the Laurea degree in computer engineering in 2006 and the Ph.D. degree in information engineering in 2010, both from the Politecnico di Milano, Milan, Italy.

He is currently a Research Associate with the Embedded Systems Laboratory of the Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, and the Dipartimento di Elettronica e Informazione of Politecnico di Milano. His current research interests include embedded systems design methodologies and architectures, reconfigurable and quantum computing, and networks-on-chip and biological neural networks.

David Atienza (M'05) received the M.S. and Ph.D. degrees in computer science and engineering from the Complutense University of Madrid (UCM), Madrid, Spain, and the Inter-University Microelectronics Center, Leuven, Belgium, in 2001 and 2005, respectively.

He is currently a Professor with the Department of Electrical Engineering and the Director of the Embedded Systems Laboratory at Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, and an Adjunct Professor with the Department of Computer Architecture, UCM. His current research interests include system-level design methodologies for high-performance multiprocessor system-on-chip and embedded systems, including new 2-D/3-D thermal-aware design, wireless sensor networks, HW/SW reconfigurable systems, dynamic memory optimizations, and network-on-chip design. In these fields, he is a co-author of more than 150 publications in peer-reviewed international journals and conferences, one book, several book chapters, and two U.S. patents.

Prof. Atienza was the recipient of the Best Paper Award at the VLSI-SoC Conference in 2009, two Best Paper Award Nominations at the ICCAD 2006 and the DAC 2004 conferences. He is an Associate Editor of IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN OF CIRCUITS AND SYSTEMS, IEEE EMBEDDED SYSTEMS LETTERS, and *Integration* (New York: Elsevier). He has been an Executive Committee Member of the IEEE Council on EDA since 2008, and a member of the Board of Governors of the IEEE Circuits and Systems Society since 2010.

Donatella Sciuto (F'10) received the Laurea degree in electronic engineering from the Politecnico di Milano, Milan, Italy, and the Ph.D. degree in electrical and computer engineering from the University of Colorado, Boulder.

She is currently a Full Professor with the Dipartimento di Elettronica e Informazione of the Politecnico di Milano, and the Vice Rector for International Research. She is a member of the IFIP 10.5 and European Design and Automation Association. She is or has been a member of different program committees of ACM and IEEE Electronic Design Automation (EDA) conferences and workshops. Her current research interests include methodologies for the design of embedded systems and multicore systems, from the specification level down to the implementation of both the hardware and software components, including reconfigurable and adaptive systems. She has published over 200 papers.

Dr. Sciuto was an Associate Editor of the IEEE TRANSACTIONS ON COMPUTERS, and is currently an Associate Editor of the IEEE EMBEDDED SYSTEMS LETTERS for the design methodologies topic area. She is also an Associate Editor of the *Journal of Design Automation of Embedded Systems*. She is the President Elect of the IEEE Council of EDA. She was the Executive Committee Member of DATE for the past ten years. She was also the Technical Program Chair in 2006 and the General Chair in 2008. She was the General Co-Chair for ESWEEK in 2009 and 2010.