Run-time Mapping of Applications on FPGA-based Reconfigurable Systems

Ivan Beretta[†], Vincenzo Rana[‖†], David Atienza[†], Donatella Sciuto[‖]
[†]Embedded Systems Laboratory (ESL), EPFL, Lausanne, 1015, Switzerland, {ivan.beretta, david.atienza}@epfl.ch
[‖]Politecnico di Milano, Milano, 20133, Italy, {rana, sciuto}@elet.polimi.it

***Abstract*— The role of Field-Programmable Gate Arrays (FPGAs) in System-on-Chip (SoC) design considerably increased in the last few years. Their established importance is due to the large amount of hardware resources they offer, as well as to their increasing performance, and furthermore to the support for reconfigurability. Even though FPGAs seem to have reached their maturity, there is still a lack of Computer-Aided Design (CAD) tools able to deal with dynamic reconguration. Existing algorithms aim at optimizing the performance of a set of applications, basing the computation on classic metrics (such as communication overhead), while reconfiguration-related issues are not taken into consideration.**

This work proposes a design methodology to map several applications on the FPGA area at run-time. Starting from a basic solution found at design-time for the initial set of applications, the proposed algorithm makes it possible to map a new application (not known at design-time), both minimizing the number of synthesis processes and optimizing the on-chip performance of the new application. Experimental results show that the proposed approach is able to achieve up to a 18% reduction in the number of reconfigurations with respect to an off-line static-mapping approach, while generally preserving the performance of the executed applications on the FPGA.

I. Introduction

Multi-core Systems-on-Chip (SoC) represent a promising opportunity to increase overall system performance [1]. This approach allows the designer to accelerate specific parts of the computation by exploiting special-purpose, high-performance cores, which coordinated execution solves very complex tasks, such as multimedia and graphics problems [2], [3]. Multi-core systems design introduces new challenges, ranging from software development to hardware optimization, such as the creation of scalable inter-core communication infrastructures.

Field Programmable Gate Arrays (FPGAs) showed promising results as deployment fabrics for multi-core systems and new communication paradigms [4], mainly because of their programmability, but also for their dynamic reconfiguration capability, which allows the device to be partially reprogrammed at run-time. Dynamic reconfiguration introduces a tremendous flexibility in hardware design, since it is possible to target different applications on the same device, and to switch among them at run-time by reconfiguring part of the physical device. Furthermore, it is possible to add new applications to the system at run-time, thus increasing the lifetime of the final product. The spreading of dynamic reconfiguration is however limited by its cost in terms of latency, since a reconfiguration process may require hundreds of milliseconds [4]. Still, the reconfiguration overhead can be limited by designing specific CAD tools that aim at reducing the amount of area that is reconfigured during the transition between two applications.

This work addresses the problem of adding a new application to an already-executing system, by mapping its soft cores onto the reconfigurable resources. The main contribution of this work is the definition of a low-complexity algorithm for this run-time mapping problem, which is able to exploit domain-related metrics to find a good quality solution. The goal is to reduce the reconfiguration overhead, while maximizing on-chip performance related to inter-core communication issues. Furthermore, since the new application must be synthesized before being deployed on the device, the proposed approach tries to simplify the synthesis phase by reusing parts of the applications that are already mapped on the device.

II. Related work

Multiple works related to mapping algorithms can be found in literature, but they are generally proposed as design-time algorithms, and only a few of them are specifically targeted for reconfigurable systems. For instance, in [5] the authors propose a mapping algorithm to optimize the communication latency between the cores, which can be applied to a promising communication paradigm called Network-on-Chip (NoC). In [6], the authors combine mapping and routing into a unified problem, which they solve using an algorithm that minimizes the complexity of the network. Finally, in [7] an algorithm that minimizes area and power consumption is proposed.

The previous examples do not explicitly consider dynamic reconfiguration, and therefore they do not aim at reducing the reconfiguration overhead. The algorithm proposed in [8] actually aims at reducing the number of reconfigurations that are used to deploy a partitioned application on the device. Still, the approach is too restrictive, because it considers only one application, which is subject to strict constraints.

In [9], an algorithm has been proposed to map many multi-core applications on a single device by exploiting dynamic reconfiguration. The algorithm maps cores into slots of fixed size, which are connected using a NoC, and it reduces both the communication overhead and the number of reconfigurations required to switch between applications. The first part of the algorithm collects the shared cores and tries to fit them into a base mapping, whereas the second part places the remaining cores. Since the algorithm faces some complex problems such as graph partitioning, it is only suitable for the design-time phase. In [10], a run-time extension has been proposed, but

978-1-4244-5309-2/10/$26.00 ©2010 IEEE

it only aims at reusing existing system components to deploy the new application, whenever it is possible.

Other examples of run-time mappers exist in literature. In [11], the proposed algorithm maps new applications in the device area that is currently unused, and it takes decisions to avoid future area fragmentation. However, the basic assumption is that all the applications should fit on the device area at the same time, and dynamic reconfiguration is not considered.

III. Hardware architecture and basic definitions

The hardware resources of the FPGA can be shared among different multi-core applications. We define an *application* as a set of cores that are required throughout the whole execution, which cooperate to achieve a complex functionality and communicate with each other to exchange data and to synchronize their execution. The structure of an application can be modeled using an undirected graph called *communication graph* [9], whose nodes represent the cores, and whose edges represent a communication. Since the communication among the cores is not uniform, it is possible to assign different weights to the edges, which represent a combination of the bandwidth and the criticality of a communication between two cores.

We divide the dynamically reconfigurable hardware architecture of the target FPGA-based SoC into fixed-size reconfigurable regions or *slots*, as shown in Figure 1. A binary file called *partial bitstream* is used to reconfigure an entire slot during the dynamic reconfiguration process. A partial bitstream encodes the physical implementation of a *slot configuration*, which is defined as the set of cores mapped in the slot, and the required communication infrastructure. The generation of a partial bitstream, also called *bitstream synthesis*, is a time-expensive process, since it requires all the stages from high level synthesis to place and route. Still, it takes a significantly lower time with respect to a synthesis for the whole device.

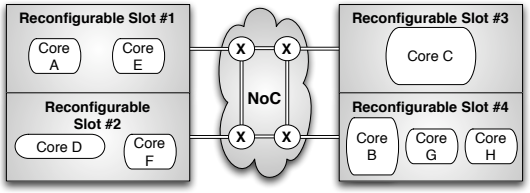

Fig. 1. The proposed NoC-based hardware architecture

The communication infrastructure is divided into two levels. Cores of different slots can communicate using the global inter-region infrastructure, which is provided as a fixed backbone based on a Network-on-Chip (NoC) [4] with an arbitrary topology. Figure 1 shows the mesh grid topology used in this work. Cores of the same slot can communicate using a local intra-region infrastructure, which is also based on a NoC, but it is included in the slot configuration and is reconfigured along with the cores. It is more specific and it is less affected by the global traffic, and therefore it performs more efficiently.

IV. The proposed run-time mapping approach

In this work we propose a low-complexity, multi-stage heuristic algorithm to solve the run-time mapping problem. An *application mapping* is defined as the assignment of all the cores of an input application to a specific slot. Each slot can contain one or more cores, or may be left unused, whereas a core can be mapped in one and only one slot. A *solution* is a set of slot configurations, and it is said to be feasible if all the cores of the applications are mapped. A solution is logically divided into two parts: the base mapping, and the specific configurations. The *base mapping* configures the device at startup with the initial configurations of all the slots and the communication backbone. As it does not contain all the cores, an application may use some configurations included in the base mapping, but it may require one or more *specific configurations* to load the missing cores. Every time the system switches between two applications, a set of specific configurations is loaded, introducing an overhead.

A set of applications can be mapped on the device at design-time using a *static mapper*. If an application is added after the design phase, a new execution of the static mapper cannot be performed because it may modify the base mapping, thus forcing the generation of a set of new bitstreams, a complete reconfiguration of the device, and a temporary interruption of the execution. Therefore, a *run-time mapper* is required to map the new application incrementally, i.e. by generating and synthesizing only the specific configurations. It is also possible to reuse some configurations belonging to other applications: they may contain redundant cores, not used by the new application, but they do not require a new synthesis. Both the static and run-time mappers aim at minimizing some domain-related metrics. The first one is the *average number of reconfigurations* that are required when switching from an application to another. The second one is on-chip performance, which is primarily related to the *communication overhead* in the system, since the algorithm should map highly-communicating cores close to each other (i.e. in the same slot, or within a few hops in the NoC). The run-time mapper also aims at minimizing the *number of partial bitstreams* to be synthesized, since it heavily affects the time required to deploy the new application.

The proposed run-time mapper is able to capture and optimize all the aforementioned metrics, and is divided into three stages. The first stage aims at reusing the existing bitstreams to deploy part of the new application, while later stages aim at mapping the cores that were unknown at design-time, and those not already included in the solution.

A. First stage: configuration reuse

The first stage of the algorithm is designed to reuse the existing configurations to map part of the incoming application. This stage is crucial to identify the components of the solution that do not need to be synthesized from scratch, thus drastically reduce the deployment time. In some cases, it is also possible to deploy the entire application using the existing bitstreams, as described in [10], but in general this approach is not the best option, since it may generate a partial solution with a lot of wasted area. In the worst case, a high number of reused configurations may prevent the algorithm from finding a

solution in later stages, thus a termination condition is required to allow later stages to work with a sufficient area availability.

The first stage adds existing bitstreams to the solution until the termination condition is detected. At every iteration, each existing configuration is associated a score, and the one with the highest score is selected to be included in the solution. The score of a configuration is computed as a linear combination of two metrics: (i) the amount of slot area occupied by useful cores. A core is said to be useful if it is used by the incoming application, but it is not part of the solution yet. This metric detects the configurations where most of the area is not used by the new application; (ii) the amount of communication between useful cores. This metric detects the configurations that resolve a large amount of bandwidth using intra-core communication, which help save traffic on the inter-core NoC.

The choice of whether the candidate configuration should be reused is determined according to two elements. The first one is the ratio between the area required to map the remaining cores and the available area on the FPGA. A high value means that it will be difficult to map the remaining cores on the free slots if the candidate configuration is selected. The second one is related to 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. Then, the candidate configuration *i* is selected at iteration *j* if:

$$Score_i \geq \alpha \cdot \frac{Required\ Area}{Availabe\ Area} + \beta \cdot j \qquad (1)$$

The length of the first stage can be tuned by modifying two parameters, α and β. A conservative tuning assigns them two high values (e.g. 8 to α, and 0.7 to β) and forces the algorithm to reuse a low amount of bitstreams, which guarantees that a feasible solution will be found in later stages. Otherwise, a more aggressive tuning assigns them two low values (e.g. 1 to α, and 0.2 to β), but it increases the probability of unfeasible solutions because of lack of area.

B. Second stage: sorting

In general, the first stage of the algorithm does not map the entire application, thus a set of new specific configurations must be generated to deploy the unmapped cores. The second stage determines the order in which the unmapped cores will be added to the solution: since the algorithm is less constrained at the beginning, it is important to map critical cores first.

The metrics used to detect the critical cores cover both area and communication aspects. In particular, larger cores should be mapped first, because at the end of the algorithm they cannot easily fit into partially-occupied slots. It is also desirable that all the cores that require a large bandwidth are considered early. Therefore, we sorted the cores according to a linear combination of their size and the weights of their incident edges in the communication graph.

C. Third stage: mapping

The third stage concludes the algorithm by mapping the remaining cores, which are considered according to the order computed above. Furthermore, the cores cannot be mapped on slots occupied by configurations selected in the first stage.

For each core, the target slot should be selected according to the communication between the current core and the already-mapped ones, so a local minimum can be found in terms of communication overhead. The target slot should be close enough to all the existing configurations that heavily communicate with the core that is being mapped, and it can be detected by assigning to each slot a score that is computed using a *propagation* technique, as shown in Figure 2. For each slot configuration already included in the solution, the amount of communication between the cores in the slot and the current core is computed according to the communication graph. The resulting value is then propagated from the slot to all the available reconfigurable regions on the FPGA, but the original value is reduced by a factor $\omega < 1$ for each hop. Each slot configuration in the current solution propagates its own value, and all these values are summed. At the end of this process, the slot with the highest value is close to all the slots that heavily communicate with the core to be mapped.

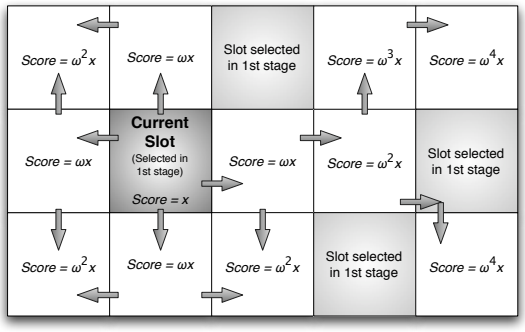

Fig. 2. Example of the propagation technique used in the third stage

The choice of the target slot cannot be driven by communication only, because a large number of slots may be used, thus affecting the number of reconfigurations. Then, it is necessary to favor the slots that have been already used in the third stage, but still provide enough free area to host the current core. This policy can be implemented by adding a value ψ to the existing score if the slot is already partially used.

V. Experimental Results

In our experimental results we compare the proposed run-time approach with the static mapper presented in [9]. The applications used in the tests are synthetic benchmarks, whose cores are characterized by different sizes and communication requirements. Although the results are different according to the various input applications, the relative gap between the run-time and the design-time algorithm is generally preserved.

Figure 3 shows how the number of reused configurations affects the quality of the solution in a 16-slots architecture. A lower number of reused configurations requires a higher number of reconfigurations, but the algorithm is less constrained in the third stage and it is able to generate a good solution in terms of communication overhead. The parameters α and

β have been manually tuned by analyzing the trends of the score and the threshold of equation (1) with different sets of applications: the goal is to stop the reusing procedure for most applications after 30 to 45% of the slots have been occupied in the first stage. According to this criterion, α and β have been set to 5.5 and 0.3 respectively, which leads the algorithm to find a good trade-off between the two metrics, while never failing to find a feasible solution in any test.

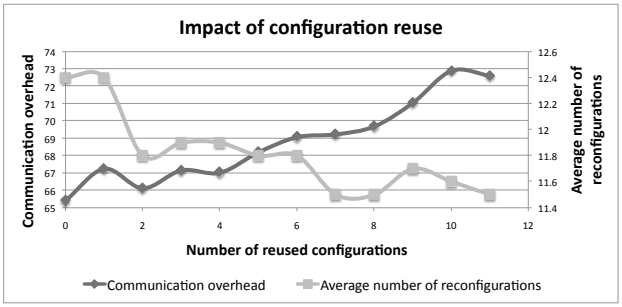

Fig. 3. Effects of configuration reuse on the quality of the final solution

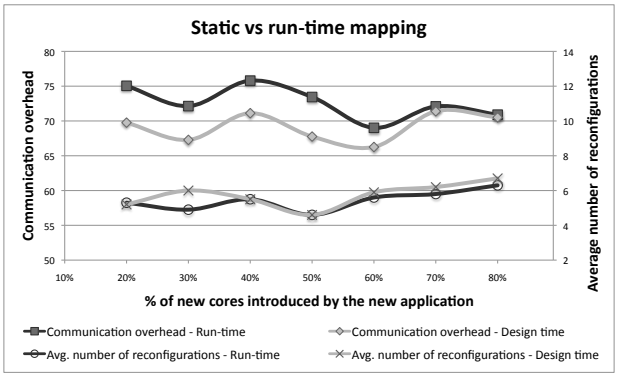

Fig. 4. Comparison w.r.t. communication and average reconfigurations

Figure 4 proposes a comparison between the static and the run-time mappers in terms of communication overhead and average number of reconfigurations. Starting from an initial solution of 6 applications mapped statically, the run-time mapper has to find a mapping for a seventh application, which introduces different percentages of new cores. The result is compared to the best of 25 executions of the design-time algorithm over all the 7 applications. The design-time mapper is expected to outperform the more flexible run-time algorithm, especially if applications are similar to each other. The results actually confirm this behavior, although the gap between the two different solutions is limited, and the run-time approach shows almost the same performance when the new application is not very similar to the ones mapped statically. Moreover, the run-time algorithm also performs at least as well as the static mapper in terms of average number of reconfigurations because in the first stage it efficiently exploits the configurations included in the base mapping, and in the third stage it is able to keep the number of used slots low. The same policies reduce the number of new bitstreams that must be synthesized, especially when the new application is sufficiently similar to the others, as shown in Figure 5.

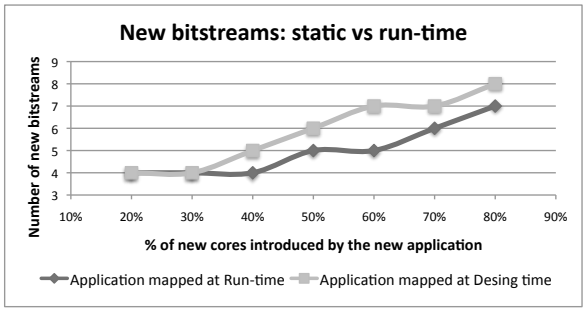

Fig. 5. Number of bitstreams to be synthesized to deploy the new application

Notice that the run-time algorithm generated the previous results more than 20 times faster than the design-time approach. Moreover, the static mapping of 6 applications plus the mapping of a seventh one is 2% faster that the static mapping of 7 applications, although the overall execution time is dominated by the design-time algorithm.

VI. Conclusions

In this paper, we addressed the run-time application mapping problem on dynamically reconfigurable devices. As new applications may be added to the system at any time, we proposed a run-time algorithm which maps all the application cores on the FPGA area in an incremental way, based on the solution computed at design-time. Experimental results show that the proposed approach can generate a good-quality solution, comparable to the one computed at design-time, but in a very short time, making it suitable for run-time scenarios.

Acknowledgments

This research has been partly supported by the HiPEAC network of excellence (www.hipeac.net) and the Swiss National Science Foundation (SNF), grant number 200021-127282.

References

[1] D. Pham, "Key considerations given to the design of a next generation multi-core communications platform," in Proc. of *ICICDT*, June 2008.
[2] P. Kollig, et al., "Heterogeneous multi-core platform for consumer multimedia applications," in Proc. of *DATE*, April 2009.
[3] E. Flamand, "Strategic directions towards multicore application specific computing," in Proc. of *DATE*, April 2009.
[4] V. Rana, et al., "A Reconfigurable Network-on-Chip Architecture for Optimal Multi-Processor SoC Communication," in *VLSI-SoC*, 2009.
[5] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto noc architectures," in Proc. of *DATE*, February 2004.
[6] A. Hansson, et al., "A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic," in *VLSI Design*, 2007.
[7] S. Murali, et al., "A methodology for mapping multiple use-cases onto networks on chips," in Proc. of *DATE*, March 2006.
[8] S. Ghiasi, et al., "An optimal algorithm for minimizing run-time reconfiguration delay," in *ACM Trans. Embed. Comput. Syst.*, 2004.
[9] V. Rana, et al., "Minimization of the reconfiguration latency for the mapping of applications on FPGA-based systems," in Proc. of *CODES-ISSS*, October 2009.
[10] I. Beretta, et al., "Run-Time mapping for Dynamically-Added applications in reconfigurable embedded systems," in Proc. of *ICM*, Dec. 2009.
[11] C.-L. Chou, et al., "Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels," *Computer-Aided Design of Integrated Circuits and Systems*, October 2008.