MIT Libraries | DSpace@MIT

MIT Open Access Articles

Run-time mapping for dynamically-added applications in reconfigurable embedded systems

The MIT Faculty has made this article openly available. ***Please share*** how this access benefits you. Your story matters.

Citation: Beretta, I. et al. "Run-Time mapping for dynamically-added applications in reconfigurable embedded systems." Microelectronics (ICM), 2009 International Conference on. 2009. 157-160. © 2010 Institute of Electrical and Electronics Engineers.

As Published: http://dx.doi.org/10.1109/ICM.2009.5418666

Publisher: Institute of Electrical and Electronics Engineers

Persistent URL: http://hdl.handle.net/1721.1/58896

Version: Final published version: final published article, as it appeared in a journal, conference proceedings, or other formally published context

Terms of Use: Article is made available in accordance with the publisher's policy and may be subject to US copyright law. Please refer to the publisher's site for terms of use.

Massachusetts Institute of Technology

Run-Time Mapping for Dynamically-Added Applications in Reconfigurable Embedded Systems

Ivan Beretta[†], Vincenzo Rana[‖†], David Atienza[†], Marco D. Santambrogio[§‖], Donatella Sciuto[‖]
[†]Ecole Polytechnique Federale de Lausanne, ESL, Lausanne, 1015, Switzerland, {ivan.beretta, david.atienza}@epfl.ch
[‖]Politecnico di Milano, DEI, Milano, 20133, Italy, {rana, sciuto}@elet.polimi.it
[§]Massachusetts Institute of Technology, CSAIL, Cambridge, MA 02139, USA, santambr@mit.edu

***Abstract*— The increasing popularity of multi-core System-on-Chip platforms introduces new challenges, both in terms of hardware platforms and design methodologies. Dynamic reconfiguration can be exploited to increase the flexibility of the system and to implement multiple applications, since it is possible to easily switch between them by reconfiguring part of the device at run-time. Additionally, new applications may be included in the system after the design time synthesis has been completed.**

This paper addresses the problem of mapping new applications on the device area at run-time, by reusing existing components of the system. We propose an heuristic technique that is able to determine how the new application should be mapped in a short time and, thanks to the reuse policy, to immediately deploy the solution on the device. The proposed algorithm also takes into consideration two conflicting performance metrics, in order to generate a good quality result.

I. Introduction

Recent developments in System-on-Chip (SoC) industry point toward multi-core products for highly specialized tasks. Their increasing complexity, as well as their time-to-market and quality requirements, reinforces the need of flexibility both in terms of hardware devices and of methodologies to exploit resource sharing [1]. Field programmable gate arrays (FPGAs) have shown promising results as deployment platforms for large-scale multi-core systems ([2], [3]), providing the required flexibility thanks to their reconfiguration capabilities. Novel FPGA families also support dynamic reconfiguration, which allows the device to be partially reprogrammed at run-time, thus adding another degree of freedom in the design of complex SoC. The hardware resources provided by an FPGA can be used to concurrently execute multiple applications, each one composed by a set of soft cores, such as instruction set processors and highly-optimized special-purpose units. Dynamic reconfiguration allows soft cores to be mapped on the device area when they are required by the currently active application, thus allowing the system to easily switch between applications at run-time. As dynamic reconfiguration introduces a relevant timing overhead (tens of milliseconds)[4], it is necessary to reduce the amount of area that is reconfigured (thus reducing the time required by the reconfiguration process) while loading a new application on the device.

The main contribution of this work is the definition of an algorithm for the run-time mapping of new applications (not known at design-time) on reconfigurable devices. These applications have to be deployed trying to reuse the parts of the system already synthesized at design-time, since a new synthesis process may require a very long time (in the order of several minutes) and forces the interruption of the system. The problem of combining the already synthesized parts of the system in order to allow the execution of the upcoming application is hard, thus we propose a fast greedy approach which aims at finding a feasible solution in a short time. The algorithm considers several performance metrics, such as the minimization of the number of reconfigurations to deploy the application and of the communication latencies among the cores, in order to generate a good solution that satisfies the execution and area constraints of the new application.

II. Related work

The problem of mapping cores on a reconfigurable device has been explored in literature, even though most of the related works aim at finding the mapping at design time. Moreover, dynamic reconfigurations is rarely taken into account, and it is not considered as a mean to switch among different applications. In [5], the authors propose an algorithm to reduce the number of reconfigurations required to switch among different parts of an application. However, they only map a single application, that must satisfy a strict set of assumptions.

In [6], the authors propose a design time mapper to optimize the communication overhead. The work is tailored for a mesh grid Network-on-Chip (NoC) communication infrastructure, and the algorithm uses traffic information in order to generate a mapping that minimizes the communication latency between cores. Since the algorithm is designed for a generic NoC-based system, it does not specifically consider dynamic reconfiguration. The same shortcoming can be found in other mappers for NoC architectures, which aim at minimizing other on-chip performance metrics. For instance, the algorithm proposed in [7] minimizes area requirements and power consumption, while the approach in [8] applies a unified mapping and routing algorithm to reduce the network complexity.

A mapper that exploits dynamic reconfiguration to map multiple applications has been proposed in [9]. The aim of the algorithm is to map application cores into fixed-size slots, which are interconnected by a NoC infrastructure. The mapper is divided into two phases, the first one being in charge of finding similarities among the applications, and the second one mapping the specific cores. The heuristic algorithm is designed to reduce both the number of reconfigurations and the

communication overhead, but its complexity makes it suitable only for design time scenarios.

Also a few works related to run-time mapping can be found in literature. In [10], the authors propose a technique to generate FPGA configurations at run-time using a low amount of resources. A mapping algorithm that incrementally adds new applications to the final solution is proposed in [11]. The mapper takes the incoming application and finds a mapping for it in an incremental way, i.e. without modifying the existing layout. Again, the algorithm does not actively support dynamic reconfiguration for application switching, since all the applications can be concurrently mapped on the device area.

III. Problem definition

A. Reference hardware architecture

The proposed tun-time mapper can be applied to generic dynamically reconfigurable devices, even though it is particularly tailored for modern FPGA families. In particular, the target device should support a fine-grained dynamic reconfiguration, and it should provide a sufficient amount of resources to host complex multi-core applications, such as Xilinx Virtex 4 and Virtex 5 devices.

In order to support complex applications, the communication among the soft cores can be assumed to be implemented using a Network-on-Chip with a mesh topology, which satisfies the flexibility and scalability requirements of the system, even though other communication infrastructures can be used. The reference hardware architecture is shown in Figure 1. The device area is divided into two logical parts: a static communication layer, which implements the backbone of the communication infrastructure and cannot be reconfigured, and a reconfigurable computation layer, which contains all the soft cores. The computation layer can be further divided into fixed-size reconfigurable regions or *slots*, which lay on a regular grid. The slot is an elementary unit of reconfiguration, i.e. the entire slot is reconfigured at once. Each reconfigurable region can contain an islands of one or more soft cores – whose internal communication is resolved inside the slot, without accessing the external backbone – or it may be unused by the active application.

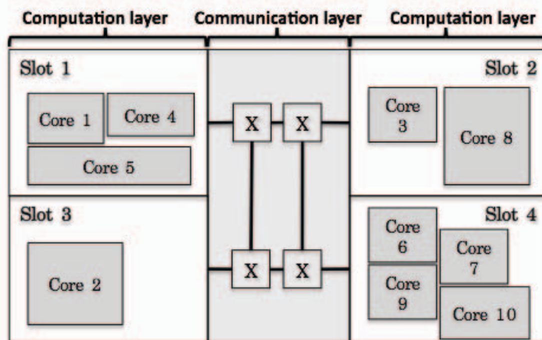

Fig. 1. An example of a reconfigurable target architecture based on a $2x2$ mesh

B. Input applications

An application is a set of cores whose concurrent and coordinated execution solves a complex task. An application can be formally modeled as an undirected graph called *communication graph* [9], whose nodes represent the soft cores, and the edges represent a communication between two cores. A weight can be associated to each edge to specify the communication bandwidth: cores which shares large communication bandwidths should be mapped in the same slot (so the communication is resolved locally) or within a few hops of distance.

C. Design time mapping

The design time algorithm proposed in [9] maps the soft cores of a certain set of input applications onto the reconfigurable regions, minimizing both the average number of reconfigurations and the overall communication overhead. The first metric can be calculated by evaluating the average amount of slots to be reconfigured while switching from an application to another one. On the other side, the communication overhead is evaluated by adding, for each edge of the communication graphs, a value obtained by multiplying the bandwidth with the number of hops that separate the two cores.

The structure of the design time solution is illustrated in Figure 2. The static solution can be divided into two parts: the base mapping and the specific configurations. The base mapping primarily contains shared cores, and it is deployed as the initial configuration of the device. Then, each application is associated to a set of specific configurations, which are loaded on the device when the application has to be executed.

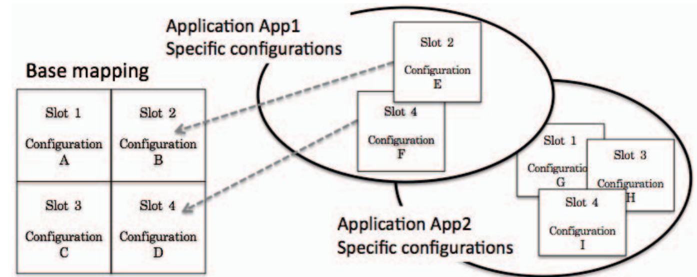

Fig. 2. Base mapping and specific configurations

D. Run-time mapping with configuration reuse

The run-time mapping problem consists in finding a suitable mapping for a new application, without modifying the design time solution, and without introducing new synthesis phases. A brand new execution of the static mapper cannot be performed, since the cores introduced by the new application may force the algorithm to generate a completely different solution, both in terms of base mapping and specific configurations. The new solution would require a time-expensive synthesis process, and moreover a complete reconfiguration would be required to load the new base mapping.

Run-time mapping must be handled by an *ad-hoc* algorithm, which does not modify the base mapping, but simply generates

a set of specific configurations for the new application. Configurations belonging to existing applications can also be used to complete the solution, even though they may contain some redundant cores. Existing configurations are the only mean to deploy the new application without performing a new synthesis process, but it is not always feasible to find a complete solution relying on configuration reuse. A necessary condition for the proposed run-time approach is that the new application does not introduce any core that was not known at design time, so every core of the application can be retrieved in at least one existing configuration. This condition is not sufficient, since it is possible to imagine a scenario in which two cores are synthesized in two different islands targeted to the same slot; in this case, the reuse of one of them precludes the reuse of the other, thus making it impossible to find a feasible solution to the problem. The aim of the run-time mapper proposed in this paper is to estimate (in a very short time) whether a solution exists or not and, if multiple feasible solutions exist, to evaluate which one is the best in terms of minimal number of reconfigurations and communication overhead.

IV. The proposed approach

The run-time mapping problem based on configuration reuse is a combinatorial optimization problem, which primarily aims at finding a feasible solution and, whenever it is possible, tries to optimize two objective functions – minimizing the communication overhead and the average number of reconfigurations. A solution is defined as a combination of existing configurations, and it is feasible if all the cores of the new application can be mapped on the device at the same time.

An exhaustive search among all the possible combinations is not sustainable, since the number of candidate solutions is exponential in the number of slots and also depends on the number of applications mapped at design time. Let us consider a relatively small system with 16 slots and 5 applications mapped statically. In the worst-case scenario, each slot corresponds to one configuration for each application (including the base mapping), and therefore 5^{16} possible solutions should be explored. Then, a heuristic technique is necessary and, given the timing requirements of the run-time scenario, a greedy approach has been chosen. The greedy algorithm can generate a solution in polynomial time with respect to both the number of applications and the number of slots, and it has been designed to minimize the two objective functions.

A. Operation of the greedy algorithm

The pseudocode of the greedy algorithm is shown in Algorithm 1. The rationale is to select a configuration at each iteration of the outermost loop, and therefore the number of iterations is bounded by the number of slots of the device. The choice is performed according to a score associated to each configuration, which include communication and area information, and it is updated at each iteration.

The scores of all the configurations i are computed by function $Compute_Scores$ according to the following equation:

Algorithm 1 Run-time mapper with configuration reuse

$Unmapped \leftarrow New\ application\ cores$
$Conf \leftarrow Design\ time\ configurations$
$Solution \leftarrow \emptyset$
repeat
  $Scores \leftarrow Compute_Scores(Conf, Unmapped)$
  $Scores \leftarrow Force_Scores_Correction(Scores)$
  $Candidate \leftarrow Get_Higher_Score(Scores)$
  $Solution \leftarrow Solution \cup Candidate$
  $Conf \leftarrow Conf \setminus Candidate$
  $Unmapped \leftarrow Unmapped \setminus Cores_In(Candidate)$
until $Solution\ is\ complete$

$$Score_i = \alpha \cdot \frac{Useful_Area}{Slot_Size} + \beta \cdot \frac{Internal_Comm}{Max_Comm} \qquad (1)$$

Two normalized terms appear in the linear combination. The $Useful_Area$ term is defined as the amount of area of a single slot occupied by soft cores that are used by the upcoming application but that are still unmapped. The $Internal_Comm$ is an index that represents the level of the communication inside a single slot, i.e. the total bandwidth among the cores of a slot.

The number of reconfigurations in the system are indirectly reduced by the computation of the useful area. In fact, the algorithm privileges configurations with a high percentage of occupied area, and therefore it is able to find a compact solution that uses a low number of slots. The algorithm also reduces the number of reconfigurations in a direct way, by favoring the configurations belonging to the base mapping when two or more configurations achieve the same score. The communication overhead is also minimized by the algorithm, which privileges the configurations that resolve large amounts of communication within the slot. The influence of each of the two objective functions on the final solution can be tuned by modifying the values of parameters α and β.

The pseudo code shown in Algorithm 1 includes a statement named $Force_Scores_Correction$. This step is used to neutralize the scores related to slots that are already used, since each slot cannot contain multiple configurations. The statement also performs the correction discussed in the following section.

B. Handling single instances

Figure 3 shows a particular scenario that has to be correctly handled in order to find a feasible solution. Core 4 is only included in configuration B, but the greedy algorithm would prefer configuration A for slot 1 because it is better in terms of area usage and internal communication. The algorithm would then realize that no feasible solution can be generated, since core 4 cannot be mapped. However, a solution exists, and it contains configurations B, C and D. This side effect can be easily corrected in the $Force_Scores_Correction$ statement: at each iteration, the algorithm checks whether a core has a single instance, and in this case the score associated to its configuration is set to an arbitrary high value.

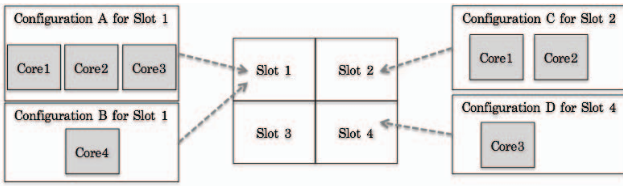

Fig. 3. Example of a particular scenario that must be specifically handled

The example of the single instance is only a special case of a more complex problem (the *single instance chains* problem). Let us assume two cores i and j belongs to two configurations each, called A_i, B_i, A_j and B_j, and let A_i targets the same slot as A_j and B_i the same slot of B_j. Then, the only way to map both i and j is to correct the scores of both A_i and B_j – or equivalently A_j and B_i – because if one of the two configurations is not used, then a feasible solution will not be found. The complexity of detecting these situations rapidly increases with the number of cores involved in the chain. The current implementation of the proposed mapper only detects single instances, in order to keep the overall complexity low.

V. Experimental results

This section presents a set of experimental results to evaluate the quality of the solutions that can be obtained with the proposed approach. The design time approach presented in [9] – which also generates the static mapping used as the starting point of the proposed approach – is employed as a reference.

The quality of the solution is evaluated in terms of communication overhead and average number of reconfigurations. Figure 4 illustrates how the two objective functions are affected by the number of cores in the new application, and it also shows the gap between the static and the run-time mapping. The experiment has been performed by taking 5 applications mapped at design time, with a number of cores between 10 and 35 of different sizes. Each application shares 75% of its cores with at least another application, and this percentage brings an additional advantage to the static mapper, which can exploit the similarities to optimize the result. The results indeed show that the static mapper finds a better solution, since it saves up to two reconfigurations, but it only achieves a 4.8% gain in terms of communication overhead.

Conversely, the run-time mapper can generate the solution in a lower time. Figure 5 shows how the run-time mapper is two orders of magnitude faster than the design time approach. Moreover, the time required to generate a mapping of n applications at design time is higher than the time required to statically map $n - x$ applications, and then complete the solution by adding the remaining x applications at run-time. The gap becomes more relevant as x increases, thus showing that the proposed algorithm scales well with the number of applications to be mapped at run-time. Finally, the data only reflects the time required to compute the mapping, and it does not include the synthesis phase that is always required at the end the static algorithm, whereas the run-time mapper can deploy the solution without performing any synthesis.

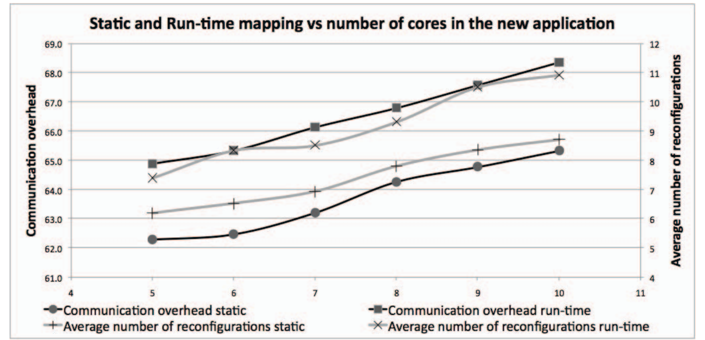

Fig. 4. Quality of the solution computed by the static and the run-time mappers versus the number of cores in the new application

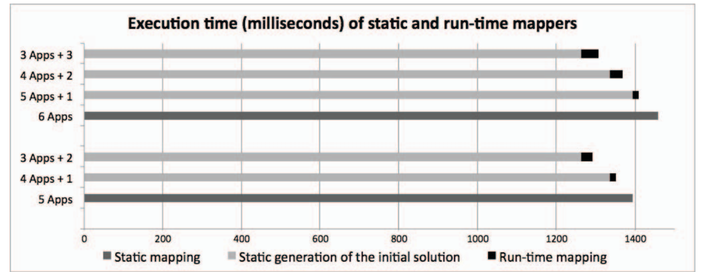

Fig. 5. Execution time of the static and the run-time mappers

VI. Conclusion and future works

This paper addressed the problem of mapping applications on a reconfigurable device at run-time, focusing on a scenario in which existing configurations can be reused to completely deploy the new applications. We proposed a fast heuristic algorithm to find a feasible mapping in a short time, which also avoid the most common traps that prevent it from finding a solution. The algorithm also exploits area and communication-related metrics to improve the quality of the final mapping.

References

[1] E. Flamand, "Strategic directions towards multicore application specific computing," in Proc. of *DATE*, April 2009.

[2] U. Ogras, et al., "Challenges and promising results in noc prototyping using fpgas," *Micro, IEEE*, October 2007.

[3] M. Hubner, et al., "Run-time reconfigurable adaptive multilayer network-on-chip for fpga-based systems," in Proc. of *IPDPS*, April 2008.

[4] V. Rana, et al., "A Reconfigurable Network-on-Chip Architecture for Optimal Multi-Processor SoC Communication," in *VLSI-SoC*, 2009.

[5] S. Ghiasi, et al., "An optimal algorithm for minimizing run-time reconfiguration delay," *ACM Trans. Embed. Comput. Syst.*, 2004.

[6] S. Murali and G. De Micheli, "Bandwidth-constrained mapping of cores onto noc architectures," in Proc. of *DATE*, February 2004.

[7] S. Murali, et al., "A methodology for mapping multiple use-cases onto networks on chips," in Proc. of *DATE*, March 2006.

[8] A. Hansson, et al., "A unified approach to mapping and routing on a network-on-chip for both best-effort and guaranteed service traffic," *VLSI Design*, 2007.

[9] V. Rana, et al., "Minimization of the reconfiguration latency for the mapping of applications on FPGA-based systems," in Proc. of *CODES-ISSS*, 2009.

[10] K. Bruneel, et al., "Automatically mapping applications to a self-reconfiguring platform," in Proc. of *DATE*, April 2009.

[11] C.-L. Chou, et al., "Energy- and performance-aware incremental mapping for networks on chip with multiple voltage levels," in *Computer-Aided Design of Integrated Circuits and Systems*, October 2008.