Run-time Reconfiguration of Polyhedral Process Networks Implementations

Hristo Nikolov   Todor Stefanov   Ed Deprettere
Leiden Institute of Advanced Computer Science
Leiden University, The Netherlands
{nikolov, stefanov, edd}@liacs.nl

Abstract

Run-time reconfigurable computing is a novel computing paradigm which offers greater functionality with a simpler hardware design and reduced time-to-market. Although, the reconfigurable technology is constantly advancing, yet reconfigurable computing is hardly employed in real systems due to the difficulties associated with realizing and managing the reconfiguration process. In this paper, we address a particular design challenge, namely, the execution management of the dynamic (reconfigurable) modules. We propose a general and technology independent approach for modeling and implementation of run-time execution management for applications modeled as polyhedral process networks. By exploiting the main characteristics of the polyhedral process networks, the approach guarantees consistent executions of reconfigurable implementations. We do not focus on low-level implementation issues of the reconfiguration process itself since the latter is not (directly) related to the execution management we propose, and therefore, it is out of the scope of this paper.

1 Introduction

When we talk about (re)configurable computing, we usually consider FPGA-based system designs. Such systems retain the execution speed of "fixed" hardware while having a great deal of functional flexibility because the logic within the FPGA can be changed if or when it is necessary. As a result, hardware bug fixes and upgrades can be administered as easily as their software counterparts. For example, in order to support a new version of a network protocol, one can redesign the internal logic of the FPGA, and send the enhancement to the affected customers by email. Once they have downloaded the new logic design to the system and restarted it, they will be able to use the new version of the protocol. Evolving from configurable computing, reconfigurable computing goes one step further by providing manipulation of the logic within the FPGA at rum time. That is, the design of the hardware may change in response to the demands placed upon the system while it is running. Here, the FPGA acts as an execution engine for a variety of different hardware functions much as a CPU acts as an execution engine for a variety of software threads. A particular example of run-time reconfigurable computing is the so called dynamic partial reconfiguration (DPR). Partial reconfiguration is the process of configuring a portion of a field programmable gate array while the other part is still running/operating. DPR allows for critical parts of the design to continue operating while a partial design is loaded into the FPGA.

Reconfigurable computing has two major advantages. First, it is possible to achieve greater functionality with a simpler hardware design. Because not all of the logic must be present in the FPGA at all times, the cost of supporting additional features is reduced to the cost of the memory required to store the logic design. The second advantage is reduced time-to-market. Most importantly, the logic design remains flexible up to, and even after the product is shipped. This allows an incremental design flow, a luxury usually not available to typical hardware designs. One can even ship a product that meets the minimum requirements and add features after deployment. Moreover, in a networked product like a set-top box or cellular telephone, it may even be possible to make such enhancements without customer involvement. In case of run-time reconfigurable computing, a main consideration is the overhead introduced by the reconfiguration process itself. If reconfiguration is performed too often, this overhead can become a bottleneck, limiting system performance. Therefore, the ratio execution-time/reconfiguration-time has to be kept reasonably high.

1.1 Problem Statement

The principal benefits of using dynamic (partial) reconfiguration (DPR) are the ability to execute larger hardware designs with fewer gates and to realize the flexibility of a software-based (multi-threaded) solution while retaining the execution speed of a more traditional, hardware-based approach. However, this comes at the price associated with the difficulties in realizing run-time reconfigurable computing. First, the provided design flows are weak and mostly experimental. It is not possible to model DPR during all the steps of a system development. For instance, SystemC can be used for first high-level steps but then it is difficult to use other tools, e.g., HW/SW partitioning tools, simply because DPR is not integrated by the tool vendors. For the low-level steps, it is (almost) impossible to simulate and validate the designs before the platform is integrated into the final board. As a result, designers are overwhelmed with too many and very low-level details in order to "get it right", making reconfigurable computing a highly error-prone and time-consuming task.

In addition to the lack of a tool support, a major challenge when using dynamic reconfiguration is the execution management of the dynamic (reconfigurable) modules. This includes both spatial and temporal management. The latter is especially important in realizing reconfigurable implementations with consistent run-time behavior. Consistency here means that any reconfigurable implementation and execution generates results equivalent to its non-reconfigurable counterpart for the same application. The challenge in realizing an execution management is further exacerbated by the complexity of today's applications, especially in the domain of multimedia embedded systems. Usually, such systems consist of multiple compute modules that operate in a globally asynchronous fashion. If these modules require reconfiguration, i.e., they are dynamic, it is very easy to violate consistency at rum time. This resembles very much the challenges in software multi-threading: common problems with thread synchronizations include deadlock and the inability to (correctly) compose program fragments that are correct in isolation [3, 6]. In general, it is not known how a programmer can come up with a multi-threaded program with correctness guarantee. The same problems arise in the reconfigurable computing as well, i.e., there is no correctness guarantee for applications demanding and implementing reconfiguration at rum time. We address this issue, and in this paper we present an approach based on conditions defining "save" points when reconfiguration may occur. The main contribution of the proposed approach is that if the defined conditions are respected, consistent system executions are guaranteed while allowing asynchronous reconfiguration of different dynamic modules at rum time.

The remaining part of the paper is organized as follows. In Section 2, we discuss the scope of the approach and the main assumptions it relies on. Section 3 presents the solution approach. Implementation details are discussed in Section 4. Section 5 concludes the paper.

2 Scope of Work

One of the main assumption in our work is that we consider only dataflow dominated applications in the realm of multimedia, imaging, and signal processing that naturally contain tasks communicating via streams of data. Such applications are very well modeled by using the parallel dataflow model of computation (MoC) called Kahn Process Network (KPN) [4]. The KPN model we use is a network of concurrent autonomous processes that communicate data in a point-to-point fashion over bounded FIFO channels, using a blocking read/write on an empty/full FIFO as a synchronization mechanism. Each process in the network performs a sequential computation concurrently with the other processes. A well-known characteristic of KPNs is that their MoC is deterministic. Always for a given input data, one and the same output data is produced. This input/output relation does not depend on the order in which the processes are executed. As the control is incorporated into the processes, no global scheduler is present.

To represent KPNs, we use polyhedral descriptions, therefore, we call our KPNs polyhedral process networks (PPN). The PPNs are specific case of KPNs, i.e., PPNs are static and everything about the execution of the process networks is known at compile time. Moreover, the PPNs execute in finite memory and the amount of data communicated through the FIFO channels is also known. We are interested in this subset of KPNs because they are analyzable, e.g., FIFO buffer sizes and execution schedules are decidable, and SW/HW synthesis from them is possible.

A PPN is implemented as a heterogeneous multiprocessor system on chip (MPSoC) using the DAEDALUS design methodology [1, 10]. In such MPSoCs, the processing components are programmable processors and dedicated HW compute modules (IP cores). The latter may provide run-time reconfiguration. In this paper, we consider fix communication topologies, i.e., a communication topology can not be reconfigured in a target MPSoC. Hence, reconfiguration can be applied only on the dedicated dynamic IP cores.

An IP core implements the main computation of a PPN process which behaves like a function call. Therefore, the computation performed by a reconfigurable IP has to resemble a function call as well. This means that for each input data read by the IP core, the core is executed and it produces output data after an arbitrary delay. In addition, to guarantee seamless integration within the dataflow of the considered heterogeneous systems, an IP core must have unidirectional data interfaces at the input and the output that do not require random access to read and write data from/to memory. Additional information about the IP cores is given further in Section 4.

3 Solution approach

In this section, we discuss the solution approach which allows for run-time reconfiguration of PPN processes in a way that consistent and deterministic PPN executions are guaranteed on the considered MPSoCs. For an illustrative purpose, we use an example presented in Section 3.1. The PPN model is briefly introduced in Section 3.2. It contains parameters which may change values at rum time. The concept of modeling process network containing dynamic parameters was introduced recently in [7]. We use the same approach as in [7] to preserve consistency of PPN executions, and in addition, we use the parameter values to trigger reconfiguration of particular processes (i.e. IP cores) at rum time. In the proposed solution approach, we do not discuss technical details about how FPGA partial reconfiguration is realized since it is highly vendor dependent and it is out of the scope of this work. Instead, we discuss when actually reconfiguration is safe to happen (in terms of consistency). It is based on conditions which have to be respected at rum time. The conditions are discussed in Section 3.4.

3.1 Illustrative example

Below, we present a part of a multi-format video encoding application. Usually, encoding algorithms work on a YUV color space while naturally, the input video information is represented in a RGB color space. Therefore, initial conversion to YUV is required and then, specific processing on the Y, U, and V image components is performed. Figure 1 illustrates this basic scenario which we will use as our illustrative example. Figure 1(a) depicts a high-level view of an MPSoC system in which the input RGB stream is converted by processing component *Conv* to Y, U, and V streams. They are further processed in parallel by processing components $Yproc$, $Uproc$, and $Vproc$, respectively. Figure 1(b) depicts the same YUV-to-RGB conversion and processing, however, implemented on a system with run-time reconfiguration. In this version, there is one dynamic module $Proc$ which is used to process the YUV data. According to the data that need to be processed, $Proc$ is dynamically reconfigured by the $Conv$ component. The implementation of the reconfiguration process must avoid any undetermined behavior. Therefore, explicit handshake logic is required for correct management of the reconfiguration. For brevity, these details are omitted in Figure 1(b).

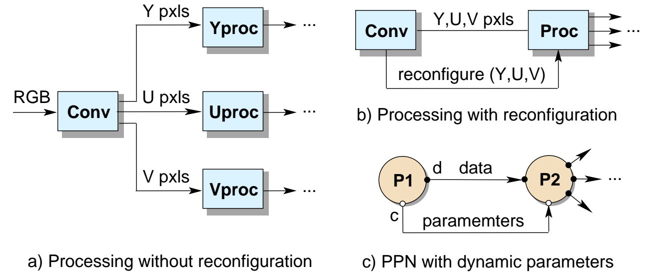

Figure 1. Motivating example.

In our example, we use only the type of the processed data to illustrate a scenario of a reconfigurable computing. However depending on the required level of flexibility, additional information, e.g., frame size: standard or high definition; type of encoding: MJPEG, MPEG4, or DivX; etc., can also be used for reconfiguration of the system at rum time. Moreover, due to performance limitations for example, the quality of the encoding may need to be constrained as well. In our approach to reconfigurable computing, we capture reconfiguration information at application level, i.e., in the polyhedral process network model we use to specify application behavior. More precisely, different configuration possibilities are defined by a set of parameters and their values in a PPN. Our illustrative example is represented as a PPN in Figure 1(c). It consists of two processes, $P1$ and $P2$, connected through one dataflow channel (d). $P1$ implements the RGB-to-YUV conversion and $P2$ realizes the processing of the Y, U, and V components. The information what type of image component is to be processed is specified by a parameter. In order to transfer parameter values between the processes, we use control FIFO channels, i.e., channel c in Figure 1(c). At run time, the parameter values are used to trigger proper reconfigurations.

As is the case with all data-flow models, the main question here is whether the PPNs with dynamic parameters are *consistent*. Consistency has to do with

a balancing of the production and consumption of tokens in the network. When this balancing is dependent on dynamic parameters, consistency conditions may be violated. In the remaining part of the paper, we discuss how we address this problem in order to guarantee consistent executions of applications modeled as PPNs on platforms using run-time partial reconfiguration.

3.2 Polyhedral (Kahn) process networks (PPN)

The parallelism in our PPNs is expressed at the level of the application tasks as a process implements a single application task only. A process of a PPN consists of a *function*, *input ports*, *output ports*, and *control*. The function specifies how data tokens from input streams are transformed to data tokens to output streams. The function also has input and/or output arguments. The input and output ports are used to connect a process to FIFO channels in order to read data tokens, initializing the function input arguments, and to write data generated as a result of the function execution. The control specifies how many times the function is executed and which ports to read/write at every execution, i.e., at every iteration (firing) of the process. The control of a process can be compactly represented mathematically in terms of linearly bounded sets of iterator vectors using the polytope model [2]. A process has a *Process Domain* (DM) which is the set of all iterator vectors. Each iterator vector corresponds to one and only one integral point in a polytope. Formally,

$$DM = \{P(p) \cap \mathbb{Z}^n\},$$

where $P(p)$ is a parametric polytope,

$$P(p) = \{i \in \mathbb{Q}^n, p \in \mathbb{Z}^m \mid Ai \geq Bp + C\},$$

where i is an iteration vector, A, B and C are integral matrices of appropriate dimensions, and p is a parameter vector with an affine range $R(p)$,

$$R(p) = \{p \in \mathbb{Z}^m \mid Dp \geq E\},$$

where D and E are integral matrices of appropriate dimensions. We use the values of the parameter vector's elements to determine different configuration options at rum time.

3.3 Process network instance

In our approach to model dynamic parameters, we introduce a notion of a PPN instance which is defined by the current value of the elements of the parameter vector. Consider the PPN representing

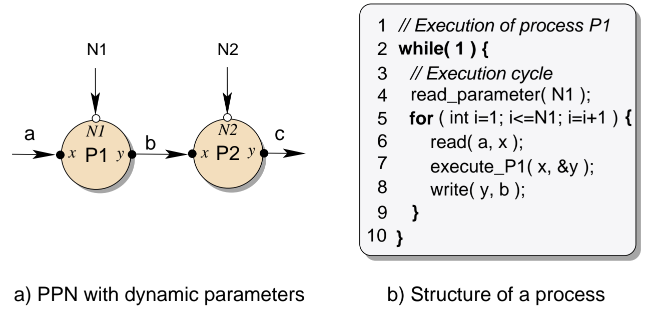

Figure 2. PPN and process execution cycle.

a producer-consumer pair, shown in Figure 2(a). N_1 and N_2 are FIFO channels of the parameters N_1 for process $P1$ and N_2 for process $P2$, respectively. Each parameter can take values within a fixed range. PPN(N_1, N_2) denotes an instance of the PPN. There is generally a relation between the parameters, in this example N_1 and N_2. Therefore, some instances PPN(N_1, N_2) are invalid instances. For the PPN network in Figure 2(a), all different instances are,

Parameters Range:	PPN instances – *PPN(N1,N2)*:
$1 \leq N_1 \leq 3$;	PPN(1,1); PPN(1,2); PPN(1,3)
$1 \leq N_2 \leq 3$;	PPN(2,1); PPN(2,2); PPN(2,3)
$N_2 \geq N_1$;	PPN(3,1); PPN(3,2); PPN(3,3)

Instances $PPN(2,1)$, $PPN(3,1)$, and $PPN(3,2)$ are invalid because they violate the condition $N_2 \geq N_1$. Similarly, instance $PPN(2,4)$ is invalid because N_2 is out of its range. Figure 2(b) shows the structure of a process we propose to deal with dynamic parameters. Network instances are selected by reading parameter values at run time. For this purpose, we add a *read parameters* phase, see line 4, prior to the actual processing at lines 5-9. Because reading parameters and data processing are repeated (possibly an infinite number of times), we call it a process *execution cycle* (lines 3-9). When all processes in a PPN have performed an execution cycle, a network instance has performed an execution.

Definition 3.1 (Consistency of a PN instance) A PN instance is consistent if after an execution, the number of tokens written to any channel is equal to the number of tokens read from it.

3.4 Preserving the consistency

The validity of the PPN instances is a necessary but not a sufficient condition to preserve the PPN consis-

tency when changing parameter values at run time. A valid set of parameters corresponds to a valid (and consistent) PPN instance. However, the transition from a valid instance to another valid instance at an arbitrary point may violate the consistency of the instances and the PPN execution. In order to transfer new values for parameters to a process of the PPN at run time, i.e., to select a new PPN instance, we use control channels with FIFO organization using blocking read/write synchronization mechanism. In addition, we define the following three conditions which are sufficient to preserve consistency when changing parameter values dynamically at run time.

C1: *Parameter sets have to correspond to valid network instances.*

C2: *A valid parameter set has to initiate a network instance execution.*

C3: *Processes may read new parameters from a valid set (corresponding to the selection of a new valid network instance) after they have completed a process execution cycle.*

In other words, parameter values may be changed (reconfiguration may take place) either before or after an execution cycle of the processes. This is taken into account by the proposed execution cycle of a process illustrated in Figure 2(b). Note that the defined conditions are valid only for consistent PPN instances. Therefore, a consistency check of a PPN instance is required, either at design time or at run time. In our approach, a consistency check is performed at design time since everything about the execution of a PPN is known. For more details about the defined conditions and the approach to deal with dynamic parameters at run time, we refer to [7] where the presented approach has been generalized for the SBF MoC [5].

4 Implementation

We consider that reconfiguration is applied on HW IP cores integrated in an MPSoC generated by Espam [8, 9]. To integrate an IP core, Espam generates a HW Module (HM) around an IP core taken from a library. To describe how reconfiguration, based on parameter values, is realized with respect to the previously defined conditions, we explain the structure of a HM, shown in Figure 3. For additional details about HW IP core integration with Espam, we refer to [8]. The processes in our PPNs have always the same structure. It reflects the KPN operational semantics, i.e, read-execute-write using blocking read/write synchronization mechanism. Therefore, a HW Module realizing a process of a PPN has a similar structure,

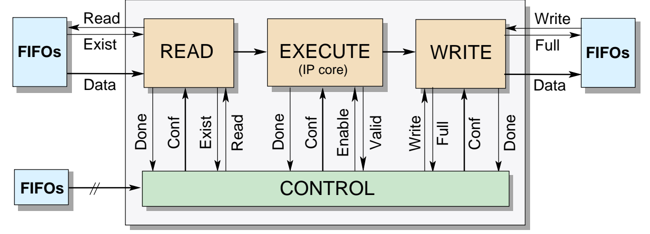

Figure 3. HW Module top-view.

shown in Figure 3, consisting of *READ*, *EXECUTE*, and *WRITE* blocks. The READ and WRITE blocks constitute the communication part of a HM. A set of input data ports belongs to the read unit and a set of output data ports belongs to the write unit. The number of input/output ports is equal to the number of the edges going in (respectively out of) the process of a PPN. The read unit is responsible for getting data from proper channels (FIFOs) at each iteration. The write unit is responsible for writing the result to proper channels (FIFOs) at each iteration. Selecting a proper channel at each iteration means to follow a local schedule incorporated into the read and write units. These local schedules are extracted from the PPN specification automatically by the Espam tool.

The EXECUTE block of a HW Module (HM) is actually a dedicated HW IP core to be integrated. It is not generated by Espam but it is taken from a library. In order to be incorporated into a HW Module, an IP core has to provide Enable/Valid control interface. The Enable signal is a control input to the IP core which allows for running the core when there is data to be processed. If input data is not available, or there is no room to store the output of the IP core to output FIFO channels, then Enable is used to suspend the operation of the IP core. The Valid signal is a control output signal from the IP used to indicate whether the data on the IP outputs is valid and ready to be written to an output FIFO channel. In addition, the IP core has also to provide an interface for accepting configuration information, illustrated by the Conf/Done signals in Figure 3.

A *CONTROL* block is added to capture the process behavior, e.g., the number of process firings, and to synchronize the operation of the other three blocks. Also, *CONTROL* implements the blocking read/write synchronization mechanism using Exist/Read and Full/Write signals. Another function of block CONTROL is to allow the parameter values to be set/modified from outside the HW Module at run time. Below, we present how the CONTROL block

implements the reconfiguration process such that the previously defined conditions are respected.

4.1 Respecting the conditions

Recall that the defined conditions are taken into account by the proposed execution cycle of a PPN process, shown in Figure 2(b). Therefore, to respect the conditions and to preserve consistency of our PPNs, the CONTROL block of a HW Module (see Figure 3) implements this execution cycle.

In the beginning, the CONTROL block reads parameter values from the corresponding control FIFO channels. If data has not been written, the control block stalls waiting for it. The correctness of the parameter values (i.e., the configuration data) has to be guaranteed (condition $C1$) by the module generating them. Thus, the combined writing of parameter values and the reading of these parameters by the control block respects condition $C2$, because only a valid parameter set will cause a PPN process to initiate an execution cycle and, consequently, an execution of a network instance. After reading control data (e.g., iteration domains and information about configuring the IP core), the CONTROL block initiates an execution cycle. First, it performs an IP (re)configuration if it is required as well as setting control information in the READ and WRITE blocks. After IP core (re)configuration is completed (indicated by signal 'Done'), the control block uses the 'Exist/Read', 'Enable/Valid', and 'Full/Write' interfaces (see Figure 3) to control the execution (cycle) of the HW Module. The end of the cycle is reached when the READ and WRITE blocks have performed all required read and write operations. This is indicated by the corresponding 'Done' signals. After that, the control block is free to initiate another execution cycle (respecting condition $C3$), i.e., to read new configuration data from the control channels and to repeat the steps described above.

4.2 Discussion

By using FIFO control channels with blocking synchronization mechanism, we keep the KPN semantics of our polyhedral process networks with dynamic parameters, i.e., we have the capability to control the execution without changing the model. Keeping the KPN model means that the deterministic behavior of our PPNs with dynamic parameters is preserved. The FIFO organization of the control channels and the blocking synchronization mechanism (the KPN semantics) keep the right order of selecting new network instances, i.e., the order in which the parameter sets are generated outside the network and written to the control channels. Since new parameter values are read by the processes after performing an execution cycle, parameter values selecting alternative PPN instances may be written to the control channels while a PPN instance is being executed. In addition, the proposed mechanism allows the processes to read the parameter values independently of each other without violating the conditions defined for preserving the consistency.

Our approach for run-time reconfiguration is applied at two levels, i.e., high-level (no FPGA reconfiguration) by setting control registers and low-level by reconfiguring the FPGA logic. Since we consider fixed communication topology, the READ and WRITE units are reconfigured by just writing data to control registers, e.g., the amount of data to be communicated and particular communication patterns to read/write from/to different FIFO channels. Dynamic partial reconfiguration is applied only on the IP core of a HW Module.

From a design-complexity prospective, the proposed approach to use PPNs with dynamic parameters to capture (rum-time) reconfiguration information and to target reconfigurable MPSoC implementations contributes to a simplified (low-level) design effort because:

1. By using the defined conditions and the control FIFOs, explicit handshaking (between processes) is eliminated. In addition, a reconfigurable IP core has to set only a “Done” signal to the CONTROL block after reconfiguration;

2. During the reconfiguration process, the dataflow FIFOs used for communication between the dynamic modules ensure proper operation of the static portion of the design.

5 Conclusions

In this paper, we proposed a general and technology independent approach for modeling and implementation of run-time execution management for applications modeled as polyhedral process networks (PPNs) and targeting reconfigurable computing. Based on the characteristics of the PPN formal model of computations, we proposed conditions which define “save” points when reconfiguration can occur. The main contribution of the presented work is that it guarantees consistent executions of reconfigurable implementations. In addition, the FIFO communication and synchronization mechanism of the polyhedral process networks simplify design efforts and facilitate automated implementations.

References

[1] Daedalus, a system-level design methodology and toolflow, http://daedalus.liacs.nl/.

[2] P. Feautrier. Automatic parallelization in the polytope model. In *The Data Parallel Programming Model*, volume 1132 of *LNCS*, pages 79–103, 1996.

[3] M. Herlihy. The multicore revolution. In *In 27th FSTTCS: Foundations of Software Technology and Theoretical Computer Science*, pages 1–8, 2007.

[4] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In *Proc. IFIP Congress 74*. North-Holland Publishing Co., 1974.

[5] B. Kienhuis and E. Deprettere. Modeling stream-based applications using the sbf model of computation. *Journal of VLSI Signal Processing*, 34(3), July 2003.

[6] E. A. Lee. The Problem With Threads. *IEEE Computer*, 36(5):33–42, 2006.

[7] H. Nikolov and E. Deprettere. Parameterized Stream-Based Functions Dataflow Model of Computation. In *6th Int. Workshop on Optimizations for DSP and Embedded Systems (ODES-6)*, Boston, USA, Apr. 6 2008.

[8] H. Nikolov, T. Stefanov, and E. Deprettere. Automated Integration of Dedicated Hardwired IP Cores in Heterogeneous MPSoCs Designed with ESPAM. *EURASIP Journal on Embedded Systems*, 2008:Article ID 726096, 15 pages, 2008. doi:10.1155/2008/726096.

[9] H. Nikolov, T. Stefanov, and E. Deprettere. Systematic and automated multiprocessor system design, programming, and implementation. In *IEEE Trans. on CAD of Integrated Circuits and Systems*, volume 27, Mar. 2008.

[10] H. Nikolov, M. Thompson, T. Stefanov, A. Pimentel, S. Polstra, R. Bose, C. Zissulescu, and E. Deprettere. Daedalus: Toward composable multimedia mp-soc design. In *In Proc. 45th ACM/IEEE Int. Design Automation Conference (DAC'08)*, pages 574–579, Anaheim, USA, June 8-13 2008.