A New Mapping Methodology for Coarse-Grained Programmable Systolic Architectures

Elias Barbudo, Eva Dokladalova, Thierry Grandpierre, Laurent George

A New Mapping Methodology for Coarse-Grained Programmable Systolic Architectures

Elias Barbudo, Eva Dokladalova, Thierry Grandpierre and Laurent George
Université Paris-Est, Laboratoire d'Informatique Gaspard Monge (LIGM), UMR CNRS 8049, ESIEE Paris
Noisy Le Grand, France
{surname.lastname}@esiee.fr

ABSTRACT

Coarse-grained programmable systolic architectures are designed to meet hard time constraints and provide high-performance computing. They consist of a set of programmable hardware resources with directed interconnections between them. The level of complexity of these architectures limits their re-usability. An automated mapping methodology is required to add a re-usability value to these architectures. In this work, we present a new list-scheduling based mapping methodology for coarse-grained programmable systolic architectures. We use a Directed Acyclic Graph to express the tasks and data dependency of the application as well as the hardware resources organization. We demonstrate that our approach can map different applications, provide a latency estimation and generate the configuration context. This approach could be the base for design space exploration and optimization tools for this family of architectures.

CCS CONCEPTS

• **Theory of computation → Scheduling algorithms**; • **Hardware →** *Operations scheduling*;

KEYWORDS

Coarse-grained programmable systolic hardware, DAG, FPGA, mapping and scheduling.

ACM Reference Format:
Elias Barbudo, Eva Dokladalova, Thierry Grandpierre and Laurent George. 2019. A New Mapping Methodology, for Coarse-Grained Programmable Systolic Architectures. In *22nd International Workshop on Software and Compilers for Embedded Systems (SCOPES '19), May 27–28, 2019, Sankt Goar, Germany*. ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3323439.3323982

1 INTRODUCTION

Time-critical applications require computing resources complying with time constraints and providing deterministic and high-performance support. Among the candidate hardware systems, we can find arrays of homogeneous or heterogeneous processors [20], Networks on Chip (NoC) [4] and mainly coarse-grained systolic programmable architectures [1, 5, 16, 18]. In this paper, we concentrate on the latter ones having the advantage to increase the overall performance drastically while decreasing computing latency.

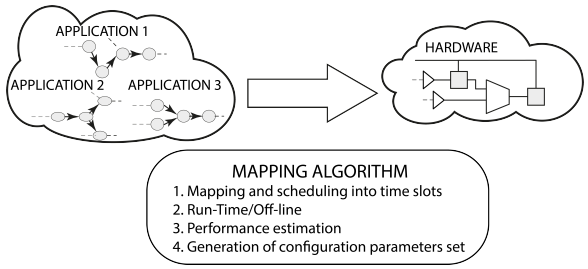


Figure 1: Automated mapping methodology principle.

Obviously, they are designed for a specific applicative field. Coarse-grained programmable systolic architectures are used to implement complex vision algorithms, that includes detection of defects on manufactured surfaces [1], street scene understanding [5], object tracking [18] and feature detection [16].

Generally speaking, this family of architectures consists of a scalable structure, partially configurable before synthesis: number and type of hardware resources, depth of pipelines, parallelism degree; and specifically programmable on run-time: data paths, type of tasks, operational parameters.

Therefore, the mapping and scheduling of various applications require a broad understanding of the internal structure and configuration parameters. It limits the hardware re-usability and their deeper integration in complex heterogeneous systems. So, the need for an automated mapping methodology becomes critical for their practical utilization.

In the past, several application mapping approaches have been explored, based on different task models, for example we can cite: the fork-join model [9], synchronous parallel task model [19] and the Directed Acyclic Graph (DAG) model [7, 11].

The DAG task model is commonly used to describe complex applications. It allows outlining the internal structure of an application, by decomposing it into atomic tasks and data dependence interconnections. Among the DAG-based mapping algorithms, we can find two important families [23], cluster-scheduling [10, 15] and list-scheduling [8, 22]. The latter one produces the scheduling based on a list that highlights a significant feature (topological order, task priority) and is commonly used for architectures with a limited number of resources. Many list-based mapping methodologies have been developed [6, 8, 12, 17, 22]. Nonetheless, these

SCOPES '19, May 27–28, 2019, Sankt Goar, Germany

ACM ISBN 978-1-4503-6762-2/19/05…$15.00
https://doi.org/10.1145/3323439.3323982

solutions do not consider the constraint structure of coarse-grained programmable systolic architectures.

In this paper, we introduce a new approach, allowing to map an application into a coarse-grained programmable systolic architecture (Figure 1). We exploit the DAG task model allowing to describe the inherent data dependency of the tasks and their parameters. This methodology is able to provide the performance estimation and generate the configuration context (parameters for the hardware).

The organization of the remaining sections of the paper is as follows. Section 2 discusses briefly the related work. Section 3 introduces the proposed methodology principles. Section 4 outlines the experimental set-up and the results. Section 5 summarizes the main contributions and future work.

2 RELATED WORK

A considerable amount of approaches brings numerous possibilities for similar mapping problems. Lu et al. [12] present a mapping methodology for coarse-grained reconfigurable architectures (CGRA). They use a directed graph as hardware model. This model allows to represent the directed interconnection between resources. Although, the methodology only consider homogeneous resources. Chin et al. [3] introduce an integer linear programming based mapping algorithm. It uses the modulo routing resource graph (MRRG) [14] as hardware model. The particularity of this model is that it allows to represent heterogeneous resources. Though, it only considers two types of nodes, functional units and routing resources. Chen and Mitra [2] present a graph minor approach for CGRAs mapping. They also use the MRRG as hardware model. They improved the MRRG by integrating a special node representing a register file. This node allows to model register allocation with scheduling, increasing the accuracy of the hardware model. This algorithm transforms the mapping problem into a graph minor problem between the application model and the hardware model.

Possa et al. [18] present a MATLAB-based function library for mapping applications to its targeted hardware, the Programmable Pipeline Image Processor (P^2IP). The P^2IP is a coarse-grained programmable systolic hardware designed specifically for real-time image and video processing. The library consists of a list of possible configurations of the P^2IP. It accepts mnemonics as inputs, and creates an interface object for the configuration of the hardware. This approach is specific to the targeted hardware and can not directly apply to other platforms. To our knowledge, this is the only work in the literature which target a coarse-grained programmable systolic architecture.

Regardless of the extensive work in this field, most methodologies rely on hardware models that do not directly apply to coarse-grained programmable systolic architectures. Thus, the application mapping to coarse-grained programmable systolic architectures is still an open problem. In this paper, we propose a new mapping methodology for coarse-grained programmable systolic architectures. It is based on three dag-based models and a new mapping algorithm. Finally, it is able to generate a performance estimation and the configuration context.

3 METHODOLOGY

Our approach is based on an Application graph (G_{APP}) and an Architecture graph (G_{HW}) as inputs, and an Implementation graph (G_{MAP}) as output, each characterized by DAG formalism

3.1 Basic notions

3.1.1 Application Model. Let $G_{APP}(T, D)$ be a DAG representing a model of an application. The nodes represent the atomic tasks that compose the application and the edges the data dependence between them. Consider T as a set of tasks such as $T = \{t_1, t_2, ..., t_n\}$, where n is equal to the number of tasks in G_{APP}. Consider D as a set of edges. Let $(t_i, t_j) \in D$ represent a data dependence between tasks, where t_i is executed before t_j. Let t_i be further described as ($type_i$, p_i), where $type_i$ corresponds to the transformation applied to the data within the task and p_i is a vector of the input parameters. We assume that, at least one t_i, can be implemented on the resources of the targeted hardware architecture.

3.1.2 Hardware model. Let $G_{HW}(R, K)$ be a DAG representing a model of a hardware architecture. Consider K as a set of edges, where the edge $(r_i, r_j) \in K$ represents a directed interconnection from r_i to r_j. Let R be a set of resources such as $R = \{r_1, r_2, ..., r_m\}$, where m is the number of resources in G_{HW}. We consider that the resources set R is a union of the three main resources classes: $R = R^P \cup R^M \cup R^C$. R^P is a subset of resources dedicated to processing tasks, R^M is a subset of memory access resources and R^C is a subset of data-path control resources. In general principle, resource r_i is characterized by $r_i = (\mathcal{T}_i, \Pi_i, \mathcal{L}_i)$. $\mathcal{T}_i$ is a set of tasks being possible to execute on r_i. Π_i is a set of corresponding working parameters for the hardware and $\mathcal{L}_i$ is a function representing the latency of a node r_i, such that $\mathcal{L}_i : r_i \rightarrow \mathcal{R}^+$.

3.1.3 Time slot. Since the number of resources provided by the hardware architecture may be insufficient, we need to split the application graph into time slots. A time slot is a subset of configured resources in order to execute a subset of tasks in separated time intervals. Each time slot contains a sub-mapping and a sub-scheduling of the application. In order to execute the whole application, we execute the time slots sequentially.

3.1.4 Implementation graph. Let $G_{MAP}(R, K)$ be the DAG obtained by graph transformations of G_{APP} and G_{HW}. As in G_{HW}, R model the resources. Consider $r_i \in R$ be characterized by the fixed properties $r_i = (\tau_k, \pi_k, l_k)$. Let $\tau_k \in \mathcal{T}_i$, $\pi_k \in \Pi_i$ and l_k be an estimated value of the latency according to $l_k = \mathcal{L}_i(\tau_k, \pi_k)$. We define the value of $\tau = idle$ for a not used node.

3.2 Proposed methodology

Figure 2 depicts our method which consists of the 4 following steps.

3.2.1 Topological sorting. The first step of our mapping methodology is a topological sorting of G_{HW} and G_{APP}. We use Kahn's Algorithm [13]. The complexity is $O(|T| + |D|)$. This step produces two lists, L_{HW} and L_{APP}. Each of them contains ordered numbers (indexes) of R and T node sets. The first, L_{HW} represents the resources organization. The second, L_{APP} represents the data dependence between tasks defined by the application model.

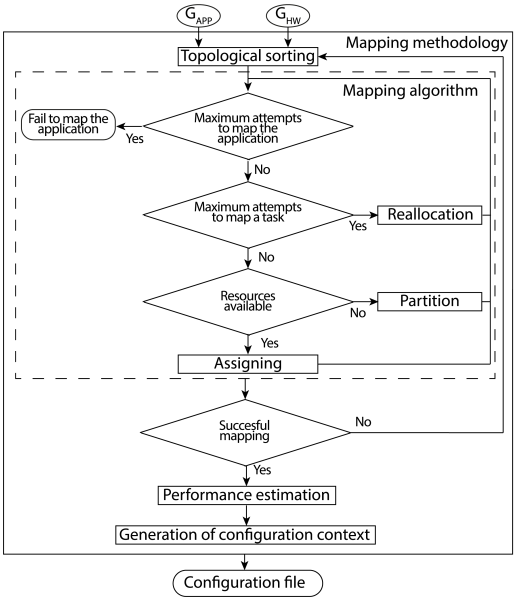


Figure 2: Proposed topological mapping methodology flow.

3.2.2 Mapping algorithm. Algorithm 1, give here under, describes the processing flow. The main part of this algorithm is the function Assigning. It aims to find a matching between a current application task t_i and a resource element from R^P. The function verifies the compatibility between the number of edges (input degree and output degree) of nodes as well as the precedence constraints. Also, it verifies, if any of the successors of the resource element can be used to map the successor of the current task.

During the mapping we have to deal with the following problems: a) *Sub-optimal correspondence between L_{HW} and L_{APP}*. This issue comes from the multiplicity of the topological sorting results and appears as a false lack of resources. b) *Availability of the Hardware resources*. The application mapping requires more resources than the available in the hardware model. These two problems are solved by the Function Partition (lines 9 and 17 for the first problem, and 13 and 15 for the second problem). The function will verify if any data-path is available. For this purpose, the function verifies if there is any datapath without a task mapped. If there is a data-path available, the function re-add its nodes to L_{HW} and continues with the mapping. If the function is not able to find available datapaths, it will proceed to split G_{APP} into sub-graphs. Next, the mapping algorithm will try to schedule them into time slots.

c) *Matching fails*. We observe this issue by an unsuccessful search of a resource for a particular task. We solve this issue using the function Reallocation. The function Reallocation is a modification of the backtracking algorithm presented by Lu et al. [12]. The function removes the mapping of the predecessor of the conflicting task, re-add the task and the resource to their respective list and restart the mapping algorithm. After a second attempt of the reallocation, the algorithm will split the remaining part of G_{APP} into sub-graphs, each sub-graph represents a unique simple path (see function Reallocation and lines 2 to 8 of Function Partition).

Algorithm 1 Mapping algorithm

Input: L_{APP}, G_{APP}, L_{HW}, G_{HW}
Output: L_{MAP}

1: **Initialize:**
  $L_{tmp} = [l_1, l_2, \cdots, l_m]$
  $//List\ of\ temporal\ results,\ where\ l_1 = l_2 = \ldots l_m = 0$
  $cnt\_v\_rea = 0$ $//Reallocation\ calls$
  $cnt\_fails = 0$ $//Failed\ attempts$
  $mapped = False$ $//Flag\ of\ task\ mapped$
  $var\_attempts = number\ of\ datapaths\ in\ G_{HW}$
2: **while** $L_{APP} \neq \{\emptyset\}$ **do**
3: $t_i \leftarrow pop(L_{APP})$
4: $mapped \leftarrow True$
5: **while** *mapped* **do**
6: **if** $cnt\_v\_rea > var\_attempts$ **then**
7: $exit(1)$ // *Fail to map the application*
8: **else**
9: **if** $cnt\_fails == |G_{HW}|$ **then**
10: $cnt\_v\_rea, L_{MAP}, L_{HW}, L_{tmp}, L_{APP} =$ *Reallocation* $(t_i, cnt\_v\_rea, L_{APP}, G_{APP}, L_{HW}, G_{HW}, L_{tmp}, var\_attempts, L_{MAP})$
11: **else**
12: **if** $L_{HW} == \{\emptyset\}$ **then**
13: $L_{MAP}, L_{HW}, L_{tmp}, L_{APP} = Partition$ $(L_{MAP}, cnt\_v\_rea, L_{APP}, G_{APP}, L_{HW}, G_{HW}, L_{tmp}, var\_attempts)$
14: **else**
15: $cnt\_v\_rea, cnt\_fails, mapped, L_{HW}, L_{tmp} = Assigning\ (\ t_i, cnt\_v\_rea, cnt\_fails, mapped, L_{APP}, G_{APP}, L_{HW}, G_{HW}, L_{tmp}\ )$
16: $L_{MAP} \leftarrow L_{tmp}$

1: **function** Assigning(t_i, cnt_v_rea, cnt_fails,*mapped*, L_{APP}, G_{APP}, L_{HW}, G_{HW}, L_{tmp})
2: $r_j \leftarrow pop(L_{HW})$
3: $s\_succe\_HW \leftarrow successors(r_j)$
4: $s\_succe\_APP \leftarrow successors(t_i)$
5: **if** $type(t_i) \in \{\mathcal{T}(r_j)\}$ **then**
6: **if** $p(t_i) \in \{\Pi(r_j)\}$ **then**
7: **if** $input\_degree(t_i) >= input\_degree(r_j)$ **and** $output\_degree(r_j) >= output\_degree(t_i)$ **then**
8: **if** $predecessors(t_i) \in L_{tmp}$ **and** $simple\_path(predecessors(t_i), t_i) \neq \{\emptyset\}$ **then**
9: **if** $type(\{s\_succe\_APP\}) \in type(\{s\_succe\_HW\})$ **then**
10: $L_{tmp}(r_j) \leftarrow t_i$, $cnt\_fails \leftarrow 0$
11: $cnt\_v\_rea \leftarrow 0$, $mapped \leftarrow False$
12: $cnt\_fails \leftarrow cnt\_fails + 1$
13: **return** $cnt\_v\_rea, cnt\_fails, mapped, L_{HW}, L_{tmp}$

The partial results of the mapping (creation of a time slot, see function Partition from line 13 to line 15) and the overall mapping are stored in a list called L_{MAP}. This list contains the parameters assigned to each resource during the mapping. We divide L_{MAP} by time slots. The elements of each time slot is equal to the resources available ($|R|$). The final step of the methodology is the creation of G_{MAP}, which is obtained by parsing L_{MAP}. G_{MAP} will collect all the information contained in L_{MAP}.

```
1: function Partition (L_MAP, cnt_v_rea, L_APP, G_APP, L_HW,
   G_HW, L_tmp, var_attempts )
2:     Initialize:
       nodes_available ← ∅
       paths_app ← unmapped datapaths in G_APP
       paths_hw ← datapaths in G_HW
3:     if cnt_v_rea > var_attempts − 1 then
4:         if ∀{paths_app} == simple paths then
               /*Function to cut a graph into a subgraphs */
5:             RemainingNodes ← cut_graph(paths_app)
6:             L_APP ← topological(RemainingNodes)
7:         else
8:             exit(1) //Fail to map the application
9:     else
10:        for ∀ path ∈ paths_hw do
11:            if available(path) then
                   /*If no node mapped in path, copy
                   nodes to nodes_available */
12:                nodes_available ← nodes ∈ path
13:        if nodes_available == {∅} then
               /*New time slot */
14:            L_MAP.append(L_tmp)
15:            L_tmp = {∅}, L_HW = {∅}
16:        else
17:            L_HW ← topological(nodes_available)
18:    return L_MAP,L_HW,L_tmp,L_APP
```

```
1: function Reallocation(t_i, cnt_v_rea, L_APP, G_APP, L_HW,
   G_HW, L_tmp, L_MAP)
       /*t_i is the conflicted node of G_APP being mapped*/
2:     if cnt_v_rea > var_attempts − 1 then
3:         L_MAP, L_HW, L_tmp, L_APP = Partition (L_MAP,
           cnt_v_rea, L_APP, G_APP, L_HW, G_HW, L_tmp)
4:     else
5:         nodes ← predecessors(t_i)
           /*Mapping removal of t_i predecessors */
6:         L_HW.append(nodes), L_APP.append(nodes)
7:         L_tmp.remove(nodes)
           /*End of the mapping removal */
8:     cnt_v_rea + 1
9:     return cnt_v_rea, L_MAP, L_HW, L_tmp, L_APP
```

3.2.3 Performance estimation . Q represents an estimated value of the total latency of the final mapping. It is obtained from G_{MAP} using the following formula:

$$Q = \sum_{t=1}^{N_s} q_t, \qquad q_t = max(d_1, d_2, ..., d_e) \tag{1}$$

$$\textbf{with } d_e = \sum_{i=1}^{N_{\mathcal{K}_i}} \mathcal{L}_i(\tau_k, \pi_k) \tag{2}$$

where N_s is the number of time slots scheduled by the mapping process, q_t is the critical path of the time slot. Notice that e is the number of paths $\mathcal{P}$ of each time slot of G_{MAP}, and each $\mathcal{K}_i \in \mathcal{P}$ represents a path from a source node to a sink node. Furthermore, $N_{\mathcal{K}_i}$ is the number of nodes in a path $\mathcal{K}_i$, τ_k is the transformation implemented on the resource r_i. The purpose of equation 2 is compute the latency of each data-path, this latency is a function of the task assigned and its parameters. Next, the higher value is considered as the critical latency of the time slot. We perform the summation of the critical latency from each time slot and produce the value of the performance estimation.

3.2.4 Configuration context. The configuration context is obtained from L_{MAP}. It contains the necessary information for the implementation. In this step, we define the parameters for R^M, such as write and read address, and also the size of data, depending on the final mapping and the input of the user respectively. Also, we add the parameters for R^C depending on the data-paths used in the final mapping. We divide the configuration context into sections, each section representing a time slot.

4 VALIDATION AND EXPERIMENTAL RESULTS

The evaluation of the methodology is twofold. First, we compare our mapping algorithm against the state of the art mapping algorithm presented in [12]. Then, we consider the morphological co-processing unit (MCPU) [1] as a candidate for the use of our mapping methodology, on which we experiment two different applications.

4.1 First evaluation and validation

For the evaluation of our approach, we used a modified version of the algorithm presented by Lu et al. [12]. We modify the priority list of the resources, used in their work, to comply with the structure of a coarse-grained programmable systolic architecture. We consider a set of interesting tasks and hardware graphs, highlighting aspects such as parallelism and complex structures. We only consider homogeneous tasks and resources to satisfy the characteristics of the considered algorithm. The purpose of this evaluation is to validate our approach in terms of scheduling length and optimal mapping. From the set of use case examples, we select three that represent special features. In the first example (Figure 3), the application graph represents a linear pipeline of tasks and the hardware graph is composed of two independent data-paths. In the second example (Figure 4), we use the same application graph of the first example and a hardware graph that represents an architecture with two non-independent data-paths and one independent data-path. In the third example (Figure 5) the application graph illustrates a complex algorithm with two different outputs, while the hardware graph

represents an architecture composed of three non-independent data-paths. We consider that the hardware model examples allow re-computation through the system memory block. That is, the sink and sources nodes are connected through the system memory block.

We can notice in Fig. 3 and 4 that our approach achieves a shorter scheduling length (one time slot) than the resulting mapping of [12]. We achieve this result thanks to the function PARTITION. This function searches for any available resource that can be used to map the remaining tasks before the creation of a new time slot. In Fig. 5, we can see that both G_{MAP} and G_{HW} are complex structures. Our approach is able to map correctly the application thanks to the verification of precedence made in the ASSIGNING function.

4.2 Real target

The MCPU is a coarse-grained programmable architecture implemented on an FPGA. It is dedicated to morphological operators such as erosion (ϵ) and dilation (δ) [21], and their deep concatenation and combinations. Figure 6 illustrates the general architecture of the MCPU. The main components of this system are several processing pipelines. Each pipeline is scalable by means of the number of basic stages (Figure 7). The basic stage consists of several processing and data-path control resources. The principal module is the large structuring element (SE) erosion/dilation which is the module that performs the erosion/dilation operations.

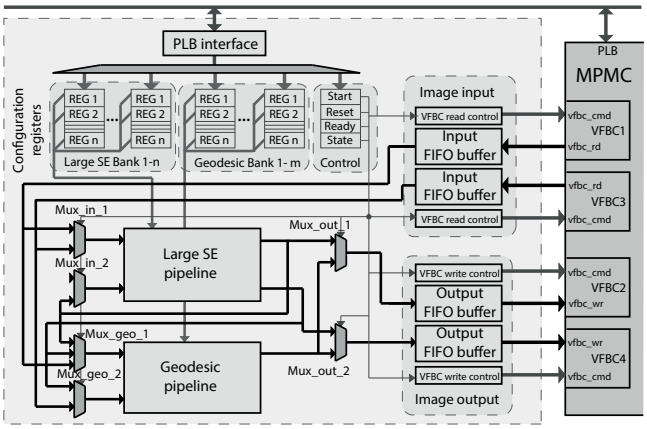


Figure 6: Morphological co-processor unit [1].

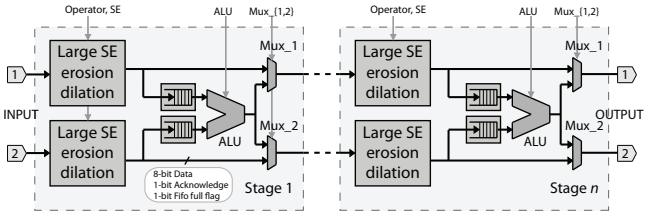


Figure 7: Large SE pipeline basic stage of the MCPU.

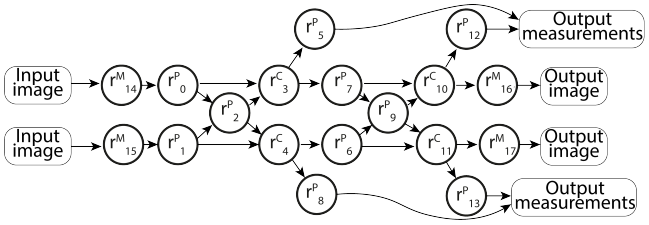


Figure 8: G_{HW} of the morphological co-processor unit.

4.2.1 Application examples. We use two example applications, an Alternated Sequential Filter (ASF) and a road line orientation detection. These applications are considered in the development of the MCPU.

The ASF is extensively used to smooth objects in images, preserving the topology characteristics. In our context, it represents a long linear pipeline of tasks with the possibility to overpass the length of the computing resources (Fig. 9).

$$ASF^{\lambda}(f) = \gamma^{\lambda}\varphi^{\lambda}\ldots\gamma^{1}\varphi^{1}(f) \tag{3}$$

where f denotes the input image, λ the SE size, γ and φ are the operators of opening and closing defined in [21]. The opening is composed of two elementary operations, erosion (ϵ) and dilation (δ). The input parameters for both, erosion and dilation, are: size, shape and angle of the SE.

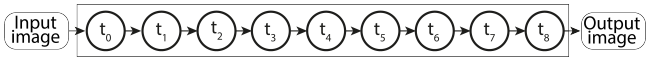


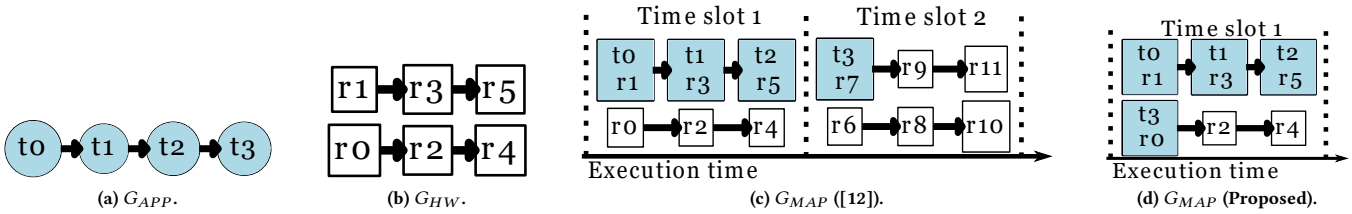
Figure 9: G_{APP} of the ASF^4 application.

The second (road line orientation detection) application represents a highly parallel task organization. The principle is the computing of oriented linear openings of the input (Figure 11).

$$\zeta_{length}(f) = arg \max_{\alpha\in[0,180)} \gamma^{\alpha}_{length}(f) \tag{4}$$

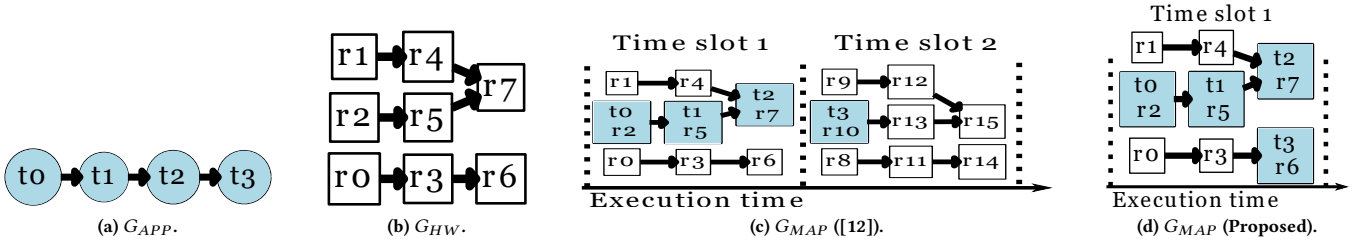
(a) Original Image. **(b) Local orientation of the road.**

Figure 10: Road line orientation detection.



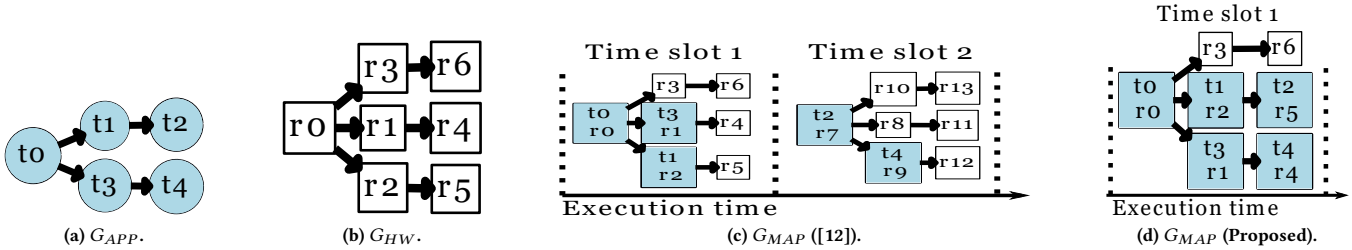
(a) G_{APP}. (b) G_{HW}. (c) G_{MAP} ([12]). (d) G_{MAP} **(Proposed).**

Figure 3: Use case example 1: a) application graph, b) hardware graph, c) and d) resulting mapping according to [12] and the proposed algorithm.



(a) G_{APP}. (b) G_{HW}. (c) G_{MAP} ([12]). (d) G_{MAP} **(Proposed).**

Figure 4: Use case example 2: a) application graph, b) hardware graph, c) and d) resulting mapping according to [12] and the proposed algorithm.



(a) G_{APP}. (b) G_{HW}. (c) G_{MAP} ([12]). (d) G_{MAP} **(Proposed).**

Figure 5: Use case example 3: a) application graph, b) hardware graph, c) and d) resulting mapping according to [12] and the proposed algorithm.

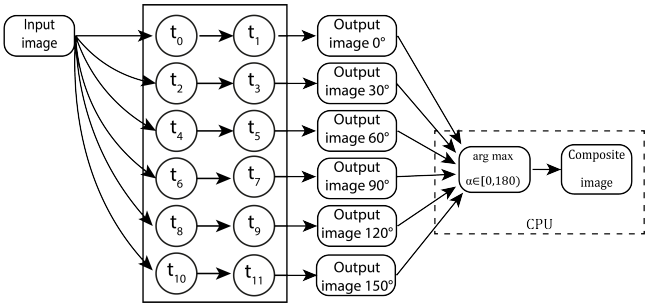


Figure 11: G_{APP} of the road line orientation application.

4.2.2 Special considerations for the mapping. In the case study, from section 3.1.1, we consider the following. For the G_{APP} of the ASF filter application, $type_{\{0,2,4,6,8\}} = \delta$, $type_{\{1,3,5,7\}} = \epsilon$. For the G_{APP} of the road line application, we are only interested in the tasks, denoted by t_i, able to be implemented in the large SE pipeline basic stage of the MCPU. We consider $type_{\{1,3,5,7,9,11\}} = \delta$, $type_{\{0,2,4,6,8,10\}} = \epsilon$. For both applications and all the nodes, $p = [$ *angle, size of structuring element, shape of structuring element, image resolution*$]$.

From section 3.1.2, for G_{HW}, $\mathcal{T}_{\{0,1,7,6\}} = \{\epsilon, \delta, idle\}$, $\mathcal{T}_{\{2,9\}} = \{+, -, min, max, idle\}$, $\mathcal{T}_{\{3,4,10,11\}} = \{0, 1\}$, $\mathcal{T}_{\{5,8,12,13\}} = \{intensity, idle\}$, $\mathcal{T}_{\{15,14,17,16\}} = \{read, write, idle\}$.

Let $\Pi_{\{0,1,7,6\}} = [$ *angle, SE size, SE shape, image resolution*$]$ and $\Pi_{\{15,16,17,18\}} = [$ *address, size of data*$]$ for nodes $r_{\{15,14,17,16\}}$. Figure 8 depicts the G_{HW} model of the MCPU. The total hardware resources of the basic stage of the MCPU are 10 processing resources, 4 data-path control resources and 4 memory access resources. These resources are available per time slot.

4.2.3 Mapping. The optimal implementation of the ASF filter requires the use of the two available data-paths. This implementation creates a data hazard. The output data of one data-path needs to be recomputed in the second data-path.

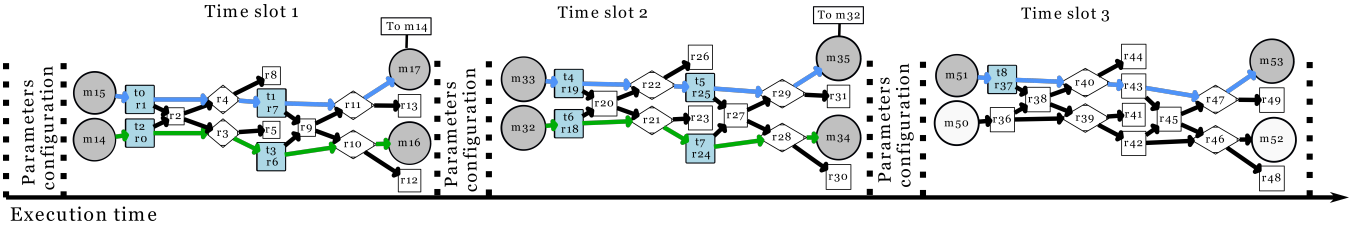


Figure 12: Mapping result and resource occupation per time slots of the ASF application.

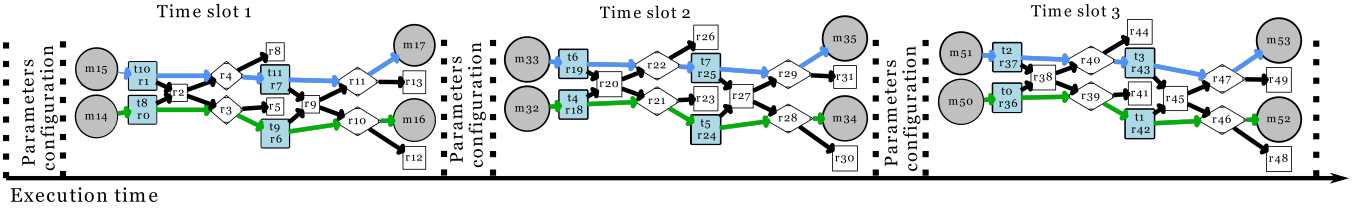


Figure 13: Mapping result and resource occupation per time slots of the road line application.

The mapping methodology handles this situation specifying the necessary parameters in the configuration context file. In the memory access section of the configuration context file, the writing direction for first data-path is the same as the reading direction of second data-path. With this action, we assure the correct re-computation of the data and preserve the data dependence. Figure 14 depicts the state diagram of the ASF filter application, it shows that the beginning of data-path 2 is slightly after the beginning of data-path 1. Figure 12 illustrates the use of resources per time slot. The enabled resources are in light blue for the resources of the subset R^P and grey for the resources of subset R^M. The edges of data-path 1 are in blue. The edges of data-path 2 are in green.

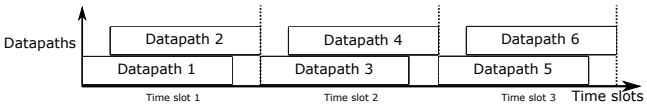


Figure 14: Time slots scheduling of the ASF filter application.

For the road line detection, the optimal implementation requires, also, the use of both data-paths. Figure 15 depicts the timing diagram of the time slots. Figure 13 illustrates the state diagram of the use of resources per time slot.

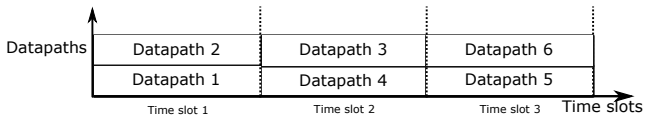


Figure 15: Time slots scheduling of the road line application.

For resources with $\tau \in \{\epsilon, \delta\}$, the worst case of computing latency is defined as three clock cycles per pixel, and the input latency is a function of the size and shape of the SE. The sum of the computing latency and the input latency gives the overall computing latency of one image process. The computing latency for the resources with $\tau \in \{+, -, min, max, 1, 0, intensity\}$ is described as one clock cycle per pixel. For illustration purposes, we fix the value of τ of the memory access resources to one clock per pixel. Also, we fix the value of the parameters configuration to one clock cycle. Table 1 summarizes the resulting timings.

Table 1: Latency estimation per time slot

	Clocks per pixel			Input latency (Images lines)		
	Time slot			Time slot		
	1	2	3	1	2	3
ASF Filter	9	9	3	7	11	4
Road line	6	6	6	7	15	12

For both applications, the results were equal to manual mapping. Table 2 summarizes the use of resources after the mapping. The application requirements represent the number of tasks per application. The subset R^P is used to map these tasks. Notice that only four resources of this subset are able to perform δ/ϵ operations. The subset R^M represents the memory access resources required for each data-path. Each data-path needs two r^M. In addition, the performance evaluation provides an estimate of the time consumed in processing the entire application. Finally, the methodology was able to generate the set of configuration parameters correctly. The results of the mapping algorithm are promising and provide a proof of concept of the proposed methodology.

Table 2: Resources utilization

	Application requirements	Used resources per time slot			Scheduled time slots
	$\|T\|$	$\|R^P\|$	$\|R^C\|$	$\|R^M\|$	
ASF Filter	9	4/4/1	0/0/0	4/4/2	3
Road Line	12	4/4/4	0/0/0	4/4/4	3

5 CONCLUSIONS

In this paper, we presented a new mapping methodology for coarse-grained programmable systolic architectures. It adds reuse possibilities for this family of high-performance architectures. We validated this work for two real applications. Our approach is able to take into account the heterogeneity of the hardware resources and their interconnection. The methodology is suitable for both offline and run-time mapping as it can provide the configuration context set. Our future work will focus on test this methodology in a wider set of architectures and develop an optimization algorithm in order to decrease the execution time of the application.

ACKNOWLEDGMENTS

This research is partially supported by the Mexican National Council for Science and Technology (CONACYT).

REFERENCES

[1] Jan Bartovsky, Petr Dokládal, Matthieu Faessel, Eva Dokladalova, and Michel Bilodeau. 2015. Morphological Co-Processing Unit for Embedded Devices. *Journal of Real-Time Image Processing* (July 2015), pp. 1–12. https://hal-mines-paristech.archives-ouvertes.fr/hal-01117406

[2] L. Chen and T. Mitra. 2012. Graph minor approach for application mapping on CGRAs. In *International Conference on Field-Programmable Technology*. 285–292.

[3] S. Alexander Chin and Jason H. Anderson. 2018. An Architecture-agnostic Integer Linear Programming Approach to CGRA Mapping. In *Proceedings of the 55th Annual Design Automation Conference (DAC '18)*. Article 128, 6 pages.

[4] M. Dridi, S. Rubini, M. Lallali, M. J. S. Florez, F. Singhoff, and J. P. Diguet. 2017. DAS: An Efficient NoC Router for Mixed-Criticality Real-Time Systems. In *IEEE International Conference on Computer Design (ICCD)*. 229–232.

[5] C. Farabet, B. Martini, B. Corda, P. Akselrod, E. Culurciello, and Y. LeCun. 2011. NeuFlow: A runtime reconfigurable dataflow processor for vision. In *CVPR 2011 WORKSHOPS*. 109–116.

[6] O. Feki, T. Grandpierre, M. Akil, N. Masmoudi, and Y. Sorel. 2014. SynDEx-Mix: A hardware/software partitioning CAD tool. In *15th International Conference on Sciences and Techniques of Automatic Control and Computer Engineering (STA)*. 247–252.

[7] N. Frid and V. Sruk. 2014. Critical path method based heuristics for mapping application software onto heterogeneous MPSoCs. In *2014 37th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 1030–1034.

[8] M. Houshmand, E. Soleymanpour, H. Salami, M. Amerian, and H. Deldari. 2010. Efficient Scheduling of Task Graphs to Multiprocessors Using a Combination of Modified Simulated Annealing and List Based Scheduling. In *Third International Symposium on Intelligent Information Technology and Security Informatics*. 350–354.

[9] Linda Kaouane, Mohamed Akil, Thierry Grandpierre, and Yves Sorel. 2004. A Methodology to Implement Real-Time Applications onto Reconfigurable Circuits. *The Journal of Supercomputing* 30, 3 (01 Dec 2004), 283–301.

[10] W. Liu, H. Li, W. Du, and F. Shi. 2011. Energy-Aware Task Clustering Scheduling Algorithm for Heterogeneous Clusters. In *2011 IEEE/ACM International Conference on Green Computing and Communications*. 34–37.

[11] C. Lu, Y. Zhang, and J. Jiang. 2017. A Breadth-First Greedy Mapping Algorithm for Reducing Internal Congestion in NoC. In *4th International Conference on Information Science and Control Engineering (ICISCE)*. 1336–1341.

[12] Lu Ma, Wei Ge, and Zhi Qi. 2012. A Graph-Based Spatial Mapping Algorithm for a Coarse Grained Reconfigurable Architecture Template. In *Informatics in Control, Automation and Robotics*, Dehuai Yang (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 669–678.

[13] Udi Manber. 1989. *Introduction to Algorithms: A Creative Approach*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

[14] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. 2002. DRESC: a retargetable compiler for coarse-grained reconfigurable architectures. In *2002 IEEE International Conference on Field-Programmable Technology, 2002. (FPT). Proceedings*. 166–173.

[15] E. E. Mon, M. M. Thein, and M. T. Aung. 2016. Clustering Based on Task Dependency for Data-Intensive Workflow Scheduling Optimization. In *2016 9th Workshop on Many-Task Computing on Clouds, Grids, and Supercomputers (MTAGS)*. 20–25.

[16] P. Ouyang, S. Yin, L. Liu, Y. Zhang, W. Zhao, and S. Wei. 2018. A Fast and Power-Efficient Hardware Architecture for Visual Feature Detection in Affine-SIFT. *IEEE Transactions on Circuits and Systems I: Regular Papers* 65, 10 (2018), 3362–3375.

[17] M. Pelcat, K. Desnos, J. Heulot, C. Guy, J. F. Nezan, and S. Aridhi. 2014. Preesm: A dataflow-based rapid prototyping framework for simplifying multicore DSP programming. In *2014 6th European Embedded Design in Education and Research Conference (EDERC)*. 36–40.

[18] Paulo Possa, Naim Harb, Eva Dokládalová, and Carlos Valderrama. 2015. P2IP. *Microprocess. Microsyst.* 39, 7 (Oct. 2015), 529–540.

[19] A. Saifullah, K. Agrawal, C. Lu, and C. Gill. 2011. Multi-core Real-Time Scheduling for Generalized Parallel Task Models. In *IEEE 32nd Real-Time Systems Symposium*. 217–226.

[20] M. Schoeberl, L. Pezzarossa, and J. Sparso. 2018. A Multicore Processor for Time-Critical Applications. *IEEE Design Test* 35, 2 (2018), 38–47.

[21] Jean Serra. 1983. *Image Analysis and Mathematical Morphology*. Academic Press, Inc., Orlando, FL, USA.

[22] A. M. Sllame and V. Drabek. 2002. An efficient list-based scheduling algorithm for high-level synthesis. In *Proceedings Euromicro Symposium on Digital System Design. Architectures, Methods and Tools*. 316–323.

[23] H. Wang and O. Sinnen. 2018. List-Scheduling versus Cluster-Scheduling. *IEEE Transactions on Parallel and Distributed Systems* 29, 8 (2018), 1736–1749.