Integrated Kernel Partitioning and Scheduling for Coarse-Grained Reconfigurable Arrays

Giovanni Ansaloni, *Member, IEEE,* Kazuyuki Tanimura, Laura Pozzi, *Member, IEEE,* and Nikil Dutt, *Fellow, IEEE*

***Abstract*—Coarse-grained reconfigurable arrays (CGRAs) are a promising class of architectures conjugating flexibility and efficiency. Devising effective methodologies to map applications onto CGRAs is a challenging task, due to their parallel execution paradigm and constrained hardware resources. In order to handle complex applications, it is important to devise efficient strategies to *partition* a kernel into pieces that obey resource constraint and methodologies to *schedule* them on the underlying hardware. In this paper, we tackle these problems by proposing algorithms to address partitioning based on recursive searches over abstract trees. A novel scheduling strategy is also described that, leveraging differences in delays of various operations, is able to efficiently map operations on CGRA architectures. Experimental evidence on kernels derived from a diverse set of data flow graphs and EEMBC benchmarks demonstrate the efficacy of the described methods, which, when combined, achieve a higher runtime performance on a given mesh size than state-of-the-art approaches (as much as 38% for the benchmark applications considered).**

***Index Terms*—Coarse-grained reconfigurable architectures, partitioning, scheduling.**

I. Introduction

FIELD programmable gate arrays (FPGAs) are a class of integrated circuits where hardware functionality can be dynamically reconfigured down to individual bits. They have enjoyed a growing success in recent years and are present in many application fields, from simple glue-logic replacement up to whole systems on a programmable chip (SoPCs), where even computational cores are implemented on the reconfigurable fabric [1], [2].

Their characteristic flexibility would appear to make FPGAs good candidates to implement reconfigurable accelerators or reconfigurable functional units; nonetheless, the huge performance gap, with respect to fixed ASIC implementations, which FPGAs present (in terms of area, delay, and power consumption) when mapping arithmetic operations, has prevented them from efficiently supporting this class of performance-critical tasks.

To cope with this shortcoming, coarse-grained reconfigurable arrays (CGRAs) [4] waive fine-grained (bit-level) configurability and instead exploit functional-unit-level reconfigurability to boost performance. CGRAs are composed by a mesh of elements (usually comprising one or more arithmetic logic units (ALUs) each [5], [6]) performing arithmetic operations and communicating using a spatial interconnect. Their parallel structure enables coarse-grained meshes to efficiently map and execute data flow graphs (DFGs) describing intensive loops (computational kernels) of applications, at the same time allowing for faster reconfiguration times than those attainable with fine-grained reconfiguration.

CGRAs are able to exploit loop-level parallelism of well-defined loops found in embedded systems and DSP applications, and indeed research efforts have been undertaken to automate the application mapping process [7], [8]. However, scheduling a well-formed kernel onto an aptly sized mesh is only one task of a multistep effort. A kernel to be executed on a CGRA, in fact, must be first transformed according to specific features of the target mesh. In particular, if the target presents complex cells featuring multiple ALUs, nodes of the kernel must be clustered [Fig. 1(b)]. Clustered kernels must then be partitioned into cuts to be executed on constrained resources [Fig. 1(c)] that can be finally scheduled on a CGRA platform [Fig. 1(d)]. This paper proposes innovations in the two latter steps, leveraging the clustering strategy introduced in [3] for the first one.

Regarding scheduling, we noted that most previous works consider time in discrete chunks, assuming that each operation executed on a CGRA tile consumes a full clock cycle.

The first contribution of this paper, in contrast, focuses on exploiting *slack*—the difference between the clock period and the critical path of execution of an operation—to chain computation and routing in each cycle. Judicious utilization of slack time makes it possible to increase routability on a reconfigurable mesh, and lead to higher quality schedules of applications without changing clock frequency.

The intuition behind the approach is presented in Fig. 2. If data communicated between cells must be stored in registers at each hop through a CGRA mesh, operation B must be executed three cycles after operation A; on the other hand, if cycle time allows it and registers can be bypassed during data routing, B can be executed immediately after A. This strategy presents no penalty in maximum clock frequency if operation

Manuscript received January 23, 2012; revised April 26, 2012; accepted June 15, 2012. Date of current version November 21, 2012. This paper was recommended by Associate Editor P. R. Panda.

G. Ansaloni is with the Embedded Systems Laboratory, École Polytechnique Fédérale de Lausanne, Lausanne 1015, Switzerland (e-mail: giovanni.ansaloni@epfl.ch).

K. Tanimura and N. Dutt are with the Donald Bren School of Information and Computer Sciences, University of California at Irvine, Irvine, CA 92617 USA (e-mail: ktanimur@uci.edu; dutt@uci.edu).

L. Pozzi is with the Faculty of Informatics, University of Lugano, Lugano 6900, Switzerland (e-mail: laura.pozzi@usi.ch).

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

Digital Object Identifier 10.1109/TCAD.2012.2209886

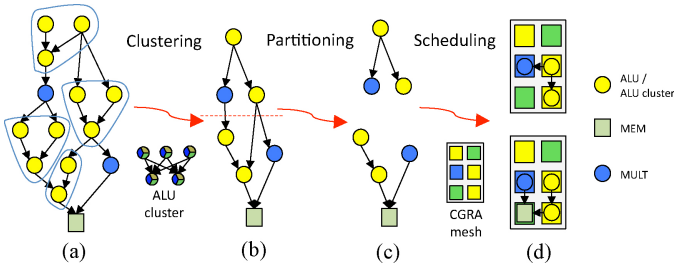


Fig. 1. Kernel scheduling on heterogeneous EGRA. (a) Kernel DFG. (b) ALU operations of the input DFG are clustered according to the methodology described in [3]. (c) The resulting graph is partitioned into cuts. (d) Cuts are scheduled on constrained hardware resources.

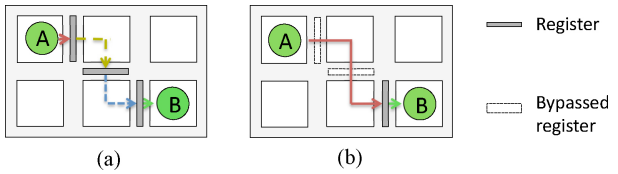


Fig. 2. (a) Registered and (b) unregistered routing through a CGRA mesh.

A, and routing data through cells is fast enough compared to the slowest operation performed on the mesh.

The second contribution of this paper concerns a problem overlooked by proposed CGRA scheduling strategies: how to deal with computational kernels where size exceeds available hardware resources. Resource overuse can result from the limitation of three physical entities in the array: the first, obvious one, is the number of cells performing computation (ALUs, or cluster of ALUs), the second, the number and size of internal memories, and the third, less obvious, is the number of different configurations that the array can hold.

Loop fission [9], a compilation technique developed to improve data locality, is a useful approach in this context. In fact, a well-conceived partition of kernel computation can produce smaller pieces, which a scheduler can then sequentially map on given hardware resources. On the other side of the coin, kernels fission introduces issues of its own, as it may add pressure on the limited memory resources present on CGRA accelerators.

As a simple illustrative example, consider the pseudocode of a kernel to be accelerated, and its DFG representation, in Fig. 3(a). If the computational requirement of the kernel exceeds what is allowed by the underlying platform, the kernel can be partitioned, as shown in this example along the dashed line. The resulting partition includes two sub-kernels [Fig. 3(b)], each of which has a decreased operation count and depth, but an increased memory footprint. In fact, all edges crossing partition boundaries (corresponding to scalar variables c and d in the example) must be promoted to arrays, so that the values produced at each iteration can be stored in internal memory, and later passed on from one sub-kernel to the other.

The storage capacity needed by a sub-kernel is therefore the sum of: 1) the memories already referenced by the computation of the sub-kernel itself, and 2) those created by the partitioning. In the example, the storage capacity needed by the first sub-kernel is the size of $aArr$, plus the size of the two

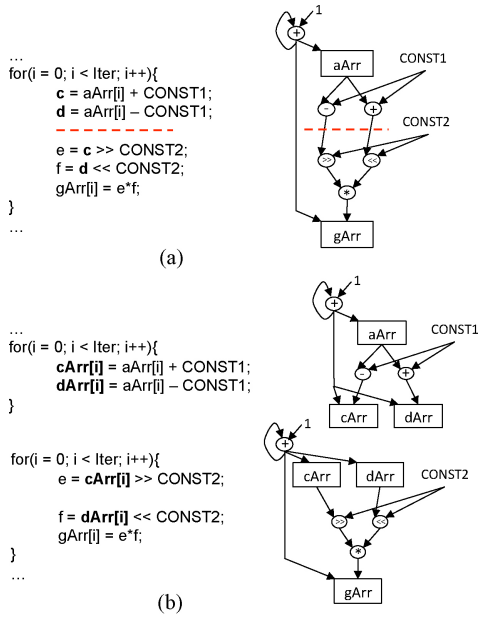


Fig. 3. Pseudocode and related DFGs of a simple illustrative example. (a) Before partitioning. (b) After partitioning. Partitioning causes a decrease in the size and depth of the graphs to be mapped onto hardware, but it increases their memory needs, as each edge crossing partition boundaries requires memory to store data, now passed between sub-kernels.

newly created memories (each needing *Iter* items, *Iter* being the loop iteration count).

Thus, effective partitioning of loops onto CGRAs must consider the tradeoffs between decreased operation count or depth and increased memory footprint. Toward this end, we propose and evaluate an algorithm for partitioning loops to be executed on CGRAs. The inspiration from the proposed algorithm is taken from one published in a different field [that of instruction set extension (ISE) identification] [10], [11]. We modify the algorithm presented there to fit our different needs, apply it to loop partitioning, and obtain an efficiency higher than state-of-the-art methods proposed so far for partitioning.

Partitioning and scheduling are tightly linked together. In fact, the former is a necessary preprocessing step to the latter when considering computational kernels whose total hardware requirements exceed the constraint of the target platform. In this paper, the two methodologies are first described and evaluated separately, investigating their individual benefit. Then, partitioning and scheduling are merged giving a holistic view of the presented framework.

The target platform of the presented scheduling experiments is the expression-grained reconfigurable array (EGRA), described in [12] and [13]. The EGRA is an architectural template, of which widely different instances can be derived parametrically, comprising heterogeneous cells and without restrictions on their arrangement.

In summary, the contributions of this paper are as follows.

1) We present a novel scheduling strategy that considers both registered and unregistered communication among tiles, resulting in an efficient utilization of computational resources, thus allowing the mapping of more complex kernels, and with a better execution performance, than is

done by state-of-the-art slack-oblivious methodologies.

2) We modify an algorithm previously proposed for instruction set extension identification [10], [11] and adapt it to perform kernel partitioning under constraints present in CGRA architectures: limited computation, memory, and configuration resources. We compare obtained results with a state-of-the-art partitioning algorithm, the cluster-based greedy algorithm described in [14]. We show that our algorithm performs partitionings of tangibly better quality than [14], while still scaling gracefully as problem size increases.

This paper proceeds as follows. In Section II, our work is positioned with respect to related efforts in the field of CGRA architectures and application mapping and partitioning techniques. Section III briefly summarizes the EGRA template and its features. Sections IV and V detail the proposed scheduling and partitioning methodologies, respectively. Section VI presents experimental evidence showing the benefits of the proposed approaches. Finally, Section VII concludes this paper.

II. Related Work

A. Architectures

Many designs have been proposed in recent years implementing the coarse-grained reconfigurable paradigm, as partially summarized in [4]. Although these architectures may vary greatly in terms of interconnection topologies, supported functionalities, and even mesh size, two evolutionary patterns can be recognized, based on the increasing complexity of tiles and level of heterogeneity.

Early proposed CGRAs, in fact, used tiles made of single ALUs (Morphosys [5], ReMarc [15]), while later designs employed more complex building blocks, able to evaluate *expressions* (groups of operations), as in PACT-XPP [6] and Montium [16]. The difference in computation time among operations supported by tiles increases with their complexity, as some expressions can be evaluated faster than others.

The transition from homogeneous to heterogeneous structures has been another recent development; notable implementations of heterogeneous CGRAs include RSPA [17] and ADRES [18]. Heterogeneity is another factor increasing differences in computation time, as slower and faster cells have to cohabit on the same array.

Note that our proposed slack-aware approach is not only important in presence of heterogeneous and/or complex cells but also for homogeneous architectures. Indeed, reconfigurable architectures, even when homogeneous, always exhibit heterogeneous computation times in tiles, depending on which operation is executed at runtime on cells. If a tile containing an ALU is configured to perform an addition, while another is configured to perform a Boolean operation, their runtime delay will vary greatly, even though the two tiles are identical (as in homogeneous architectures).

We exploit this imbalance to improve routability by allowing the output register of each tile in a mesh to be by-passable, so that operations can be executed in different cells in the same clock cycle. This mechanism is a feature of the EGRA architectural template [12], [13], described in Section III, and used as a target of the scheduling methodology detailed in Section IV.

TABLE I
CGRA Scheduling Methodologies

Spatial	Modulo
	DRESC [19]
SPKM [20]	Hatanaka [21]
SMP [22]	Graph embedding [23]
	Res. pipeline [17]

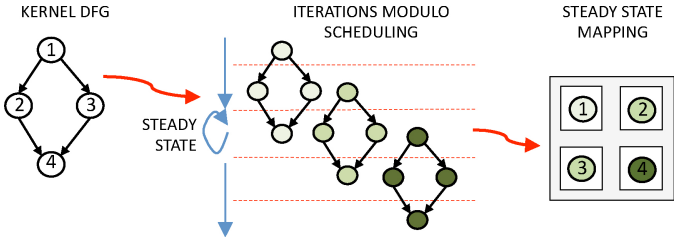


Fig. 4. Modulo scheduling of a simple kernel. DFG representing iterations of the kernel are partially overlapped to increase parallelism. Multiple iterations are concurrently executed on different CGRA cells.

B. Scheduling Techniques

Application scheduling on CGRAs poses a novel challenge, mainly due to their sparse interconnection topology with distributed register files. Research in this field can be broadly categorized into spatial and modulo scheduling approaches as illustrated in Table I and outlined below.

Spatial scheduling employs strategies used in FPGA place and route, with the goal of maximizing execution parallelism. Examples of this strategy are SPKM [20] and SMP [22]. These works acknowledge the dual function (computation and data routing) of CGRA cells but, as opposed to our method, neglect the opportunity to chain routing through cells to speed up execution.

On the other hand, modulo scheduling approaches have been deployed by Mei [19], Hatanaka [21], and Park [23]. In these papers, both spatial and temporal dimensions are considered during mapping, borrowing from techniques originally developed for very long instruction word architectures [24]. Modulo scheduling obtains high parallelism by partially overlapping execution of different kernel iterations. Fig. 4 describes a simple modulo scheduled application, in which, during steady state, operation 1 of iteration i is executed in the same clock cycle of operations 2 and 3 of iteration $(i-1)$ and operation 4 of iteration $(i-2)$.

However, these approaches also overlook critical paths issues by assuming only registered connections between tiles, so that each operation consumes an entire clock cycle. They also adopt less challenging constraints with respect to our paper, by considering only homogeneous meshes and the presence of a register file to hold temporary data in each cell. [1]

[1] Although not investigated in this paper, conceptually, architectures featuring local register files can also be modeled with the proposed methodology.

Another approach is Kim's research on resource pipelining [17], investigating how slow tiles can be pipelined and integrated with faster ones. We go one step further, as our methodology can be applicable even when the divide between "slow" and "fast" tiles is not clear-cut and, again, when the execution time on a given tile depends on the operation scheduled onto it, dictated by configuration.

CGRA schedulers considering the possibility to chain cells are proposed by Toi [25], Parks [26], and Barat [27]. As opposed to our paper, [25] does not investigate the complex interdependency between timing violations and resource overuse, while [26] and [27] only allow chaining in particular cases: pairs of neighboring cells and cells in a fully connected stripe, respectively.

C. Partitioning of Computational Kernels

Application partitioning to cope with limited hardware resources is the focus of many research efforts, as it presents itself in a variety of scenarios.

The first scenario is that of partitioning methodologies targeting FPGAs. Kaul [28] proposed an nonlinear programming formulation to optimally solve temporal partitioning of applications for time-multiplexed FPGAs. This approach assumes that an application is split in well-formed tasks beforehand, and does not scale above a limited number of tasks. The same problem is tackled by Liu [29] by adapting the Kernighan–Lin (KL) network-flow-based algorithm [30] to directed graphs; however, the KL approach cannot directly guarantee the number of edges in each subgraph, and therefore the memory requirements of a subgraph is within given bounds.

In the context of high-level synthesis, Purna [14] proposed a cluster-based heuristic to map DFGs to multi-FPGA boards while minimizing communication bandwidth. The algorithm has a linear complexity but, as highlighted in Section VI, often fails to identify good partitions, especially when dealing with fairly complex computational kernels.

The research in ISE identification [10], [11], [31] aims at identifying groups of operations to be implemented as custom functional units with constrained inputs and outputs. We note that the ISE problem has a high similarity to the partitioning problem tackled in this paper, and we therefore take the route of adapting an efficient ISE algorithm, proposed in [10] and [11], to the scenario of loop partitioning. The algorithm proposed in this paper presents a different set of constraints than in [10] and [11], and for the first time, we apply it to loop partitioning.

Our methodology, proposed in Section V, is based on loop fission, where sub-kernels execute until completion for as many iterations as needed, with reconfiguration happening only at their *boundaries*. An alternative approach is loop disserving, described in [32] as applied to the PACT-XPP CGRA, in which the underlying hardware is reconfigured *inside* loop bodies as many times at each kernel iteration. Loop disserving does not need temporary arrays to store intermediate data, but presents a much higher configuration overhead.

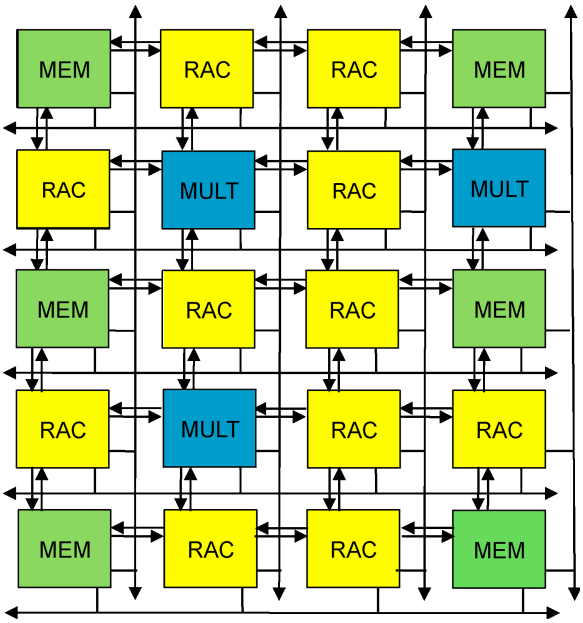


Fig. 5. Example EGRA instance composed of 3 multipliers, 6 memory cells, and 55 ALUs in 11 RACs.

III. Target Architecture

Kernel scheduling on CGRAs is target dependent: the number of computing cells, their type, and the connection scheme of the mesh dictate that operation can be executed on a cell and the routing resources are available to connect them. In this paper, an instance of the EGRA is considered. The template is briefly introduced here; more details on its structure are provided in [33].

Fig. 5 shows a sample EGRA instance that is derived from a generic template. An EGRA is composed of a mesh of cells of parametrically determined size, communicating with nearest neighbor connections, and local horizontal and vertical buses. A by-passable register is present at the output of each cell.

Tiles at each location of a mesh can be one of three basic types: multiplier, cluster of ALUs [reconfigurable ALU cluster (RAC)], or memory, as decided by design parameters dictated by a machine description.

These three types of tiles can accommodate the most commonly used operations present in computational kernels of embedded systems' applications; nonetheless, different types of cells can be integrated in the template as needed by implementing their internal structures, as long as they conform to the common interface used for intertile communication.

Each tile type can be customized at design time to fit intended target applications. As an example, memory cells can be instantiated as single or dual port, with addressing modes ranging from 8 to 32 b. Multipliers can support signed, unsigned, or both types of multiplication.

A. Structure of RAC Tiles

RACs, depicted in Fig. 6, are composed of rows of ALUs connected through switch boxes and support various operations, including if-conversion through the usage of 1-b *flags*. The number of rows in a RAC, the number of ALUs in

TABLE II
CRITICAL PATH DELAY OF DIFFERENT EGRA OPERATIONS

	Route	RAC						mult	mem
		bool-bool	bool-sh	bool-add	sh-sh	sh-add	add-add		
Critical path (ns)	0.31	0.67	0.85	0.98	1.03	1.16	1.29	1.39	0.85
Percentage of a 2-ns clock period	16	34	43	49	52	58	65	70	43
Routing hops	5	4	3	3	3	2	2	1	3

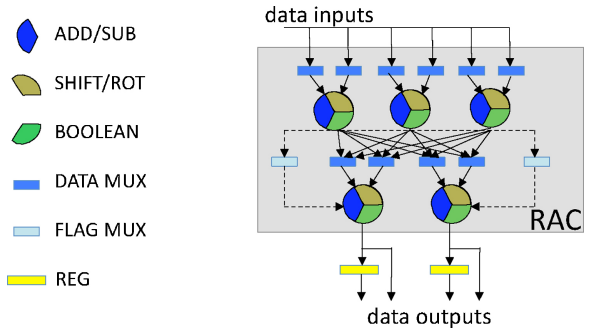


Fig. 6. Structure of a 3-2 reconfigurable ALU cluster.

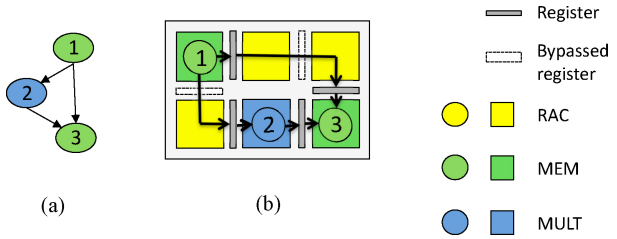


Fig. 7. (a) Example DFG. (b) Its slack-aware mapping on a 3 × 2 heterogeneous CGRA.

each row, and the operations supported by ALUs are again machine description parameters. For this paper, we deploy two-row RACs, as depicted in Fig. 6, since this configuration was shown to achieve good overall performance on several DSP benchmarks through extensive design space exploration [12]. An efficient technique to cluster ALU operations into expressions (groups of operations being placed on a single RAC) is described in [3].

B. Reconfiguration

The machine description dictates design–time instance structure. *Configurations* are then programmed on an instance and executed at runtime, dictating the operations to be dynamically executed. Although the former is statically decided, the latter are stored in a configuration memory local to each cell.

Multiple configuration words are programmed at once in each cell, corresponding to functions to be performed in different clock cycles during kernel execution. This feature makes it possible to perform modulo scheduling; to execute a modulo scheduled kernel, a control unit activates a number of prologue configuration words, then iterates through steady-state configurations for the desired kernel iterations and terminates by triggering the epilogue ones, as illustrated in Fig. 4.

C. Delay of Tiles

Synthesis data shown in the first row of Table II highlights the different critical paths of array tiles in their combinatorial part, depending on their type, and based on the configured operation for RAC tiles. To extract these results, we considered RACs composed of five ALUs in two rows (corresponding to the scheme in Fig. 6), a multiplier capable of both signed and unsigned multiplication, and single-ported 1-kB memory cells with 32-b data addressing. We employed Design Compiler from Synopsys and TSMC 90-nm libraries.

Assuming a working clock period of 2 ns, the second row of Table II shows the percentage of such period taken by each cell performing their supported operations. When this percentage is low, a greater number of routing hops can be accommodated in the same cycle, as shown in the motivational example of Fig. 2(b). The third row of Table II shows how many routing hops can be performed after computation and in the same clock cycle, without violating timing constraints. For example, a RAC configured to execute two shift operations can chain three routing hops before exhausting cycle time, while a multiply operation can chain just one hop.

The data in Table II motivates how heterogeneous computation times can be leveraged to increase schedulability of kernels without increasing clock period, and is used to derive scheduling results presented in Sections VI-A and VI-C.

IV. SLACK-AWARE SCHEDULING

The goal of the scheduler is to modulo-map a DFG, representing an iteration of a computational kernel, onto the architecture, i.e., onto a scheduling space representing a computational mesh. This problem is known to be NP-complete [34], and we devise a nonexact method to solve it.

We start by showing an example DFG to be mapped [Fig. 7(a)] and a valid mapping [Fig. 7(b)] achieved using a slack-aware methodology on a 3 × 2 mesh with nearest-neighbor connections and composed of three RACs, one multiplier, and two memory cells. The graph can be executed in three cycles because it chains routing through two tiles between operations 1 and 3 in a single clock cycle.

Our slack-aware methodology aims at finding a solution that is: 1) valid, and 2) as high performance as possible. A valid schedule does not have resource hazards (no two operations are mapped on the same resource at the same time) and does not violate cycle time (no sequence of chained operations exceeds cycle time). A high-performance schedule takes as few cycles as possible to operate.

In a nutshell, the proposed scheduling algorithm starts from an initial mapping that is high performance but possibly invalid, and iterates in search of a valid solution via a simulated

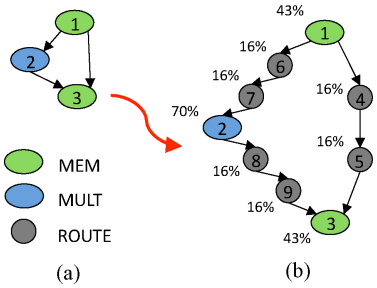


Fig. 8. (a) Routing nodes insertion on a DFG, with the (b) annotation of the critical path length relative to the clock period.

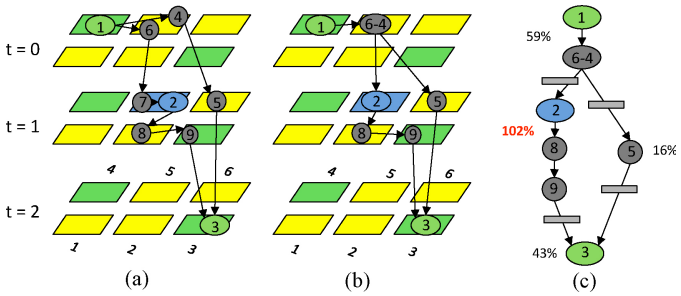


Fig. 9. (a) Expanded DFG mapping on the scheduling space. (b) After redundant nodes deletion. (c) Resulting DFG with annotation of routing times. The chain of nodes 2 and 8 violates the timing constraint.

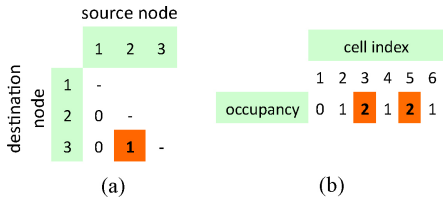


Fig. 10. (a) SVT and (b) MRT derived from the scheduling space in Fig. 9.

annealing strategy. If a valid solution is not found with the currently sought high performance, the target performance is lowered and the iteration starts again. Each step of the proposed algorithm will now be explained.

A. Expansion of the Input DFG

To account for the dual use of CGRA cells (computation and routing), the input DFG is expanded by inserting routing nodes. On each edge, the number of routing nodes must be sufficient to completely traverse the scheduling space, whose time dimension is bounded by the maximum as-late-as-possible (ALAP) among operations to be mapped [24], while its space dimension corresponds to the physical size of the reconfigurable mesh. This approach is an extension of Yoon's work [20] from a spatial to a modulo-constrained environment.

In the case of the considered example, this amounts to two routing nodes and the graph is expanded accordingly (Fig. 8).

B. Generation of an Initial Schedule

The scheduling space is a 3-D graph replicating the bidimensional CGRA structure (size, cells' types, and interconnection scheme) on max(ALAP(op)) time planes (node 3 in the example). The graph edges follow the physical CGRA topology, and connect cells both in the same time plane (representing unregistered connections), as well as from a plane to the following one (representing registered connections).

To generate an initial schedule, three steps are performed [and illustrated in Fig. 9(a) and (b)]: first, operations are placed in the scheduling space on cells that support them and respecting their precedence constraints. Then, routing nodes are mapped to connect such operations, by employing the A* algorithm [35]. Finally, redundant routing nodes are deleted; routing nodes can be redundant either because they carry the same data of a node already scheduled on the same position [node 4 or 6, Fig. 9(a) and (b)] or if they are placed at the position of their successor operation node [node 7, Fig. 9(a) and (b)].

Fig. 9(c) shows the mapped DFG, decorated with registers among planes, and annotated with delays.

In the following, we give details of the second step that maps routing nodes. Mapping routing nodes on the cells between a predecessor cell ($\text{cell}_{\text{pred}}$) and a successor cell ($\text{cell}_{\text{succ}}$) is handled as a problem to find the least costly path between them on the scheduling space considering different candidates. The cost of a routing cell placed on a candidate path ($\text{cell}_{\text{rout}}$) is defined as

$$g(\text{cell}_{\text{rout}}) = \text{distance}(\text{cell}_{\text{pred}}, \text{cell}_{\text{rout}}) \times \#\text{overused\_cells\_in\_path}$$
$$h(\text{cell}_{\text{rout}}) = \text{distance}(\text{cell}_{\text{rout}}, \text{cell}_{\text{succ}})$$
$$\text{cost}(\text{cell}_{\text{rout}}) = g(\text{cell}_{\text{rout}}) + h(\text{cell}_{\text{rout}})$$

where $g()$ calculates the cost of the candidate path between $\text{cell}_{\text{pred}}$ and $\text{cell}_{\text{rout}}$, while $h()$ estimates the cost between $\text{cell}_{\text{rout}}$ and $\text{cell}_{\text{succ}}$. The algorithm has a complexity of $O(n(\log(n)))$, where n is the number of routing nodes from $\text{cell}_{\text{pred}}$ to $\text{cell}_{\text{succ}}$.

C. Calculating the Cost of a Schedule

The initial schedule, constructed in the previous step and shown in Fig. 9, is not valid, as explained in the following. To check whether a schedule is valid or not, a slack violation table (SVT) and a modulo resource table (MRT) are derived, the former keeping track of timing violation, the latter checks resource overuse.

A timing violation occurs when a path from register to register exceeds the cycle time. Delays over paths are calculated, and a table is kept that indicates the amount of violation on each edge. In the example [Figs. 9(c) and 10(a)], the SVT indicates a violation between nodes 2 and 3. Indeed, delay from 2 to 3 accounts to 102% of cycle time, as node 2 is computed at $t = 1$, and its output is routed for two hops before registering the result.

Resource overuse occurs when more than what can be supported by a cell is mapped onto it. This can happen in two cases: 1) when a cell is being used to route more than a single value, and 2) when a cell is being used to compute an operation, and to route a different value.

Information on resource overuse is stored in the MRT, and a note is needed here on modulo scheduling, to explain

the MRT. Modulo scheduling aims at maximizing parallelism by pipelining successive iterations of kernels execution; the distance (in clock cycles) between two iterations is defined as the initiation interval (II). To account for pipelining, the scheduling space must be folded according to the II when considering resource overuse; the resulting MRT is composed of exactly II rows, and contains the usage of each resource added modulo II. Fig. 10(b) illustrates the MRT for the initial placement in Fig. 9 considering II = 1. It can be noted that cells 3 and 5 are overused. Cell 5, e.g., is used both to hosts node 6-4 at $t = 0$ and to execute node 2 at $t = 1$.

This scheme can be easily extended to more complex topologies, including shared communication links, modeled as resources able to accommodate routing cells only. Indeed, the results presented in Section VI consider local buses.

Once the MRT and the SVT are computed, a placement cost can be derived by adding up overuse and timing violations

$$\text{cost} = \sum_{\text{cells,buses}} \max((\text{MRT}_{i,t} - 1), 0) + \alpha * \sum_{\text{edges}} (\text{SVT}_{op})$$

where $\text{MRT}_{i,t}$ are the elements of the MRT, SVT_{op} are the elements of the SVT, and α is a parameter trading off the importance given to each violation type (an $\alpha = 0.3$ was empirically determined as a good balance in the experiments presented in Section VI-A).

D. Iterating in Search of a Valid Solution

If the current schedule is not valid, a new one is created; an operation node is unscheduled together with its successor and predecessor routing nodes, freeing up related resources. The operation node is then remapped and related routing is performed to and from the node. A new cost value is computed and the move is accepted depending on its cost and the current (ever-decreasing) temperature. The process is repeated until a valid mapping is found (with placement_cost = 0) or if the maximum number of tries has been reached.

E. Lowering Performance

If a valid solution has not been found after a number of iterations, a less aggressive mapping, of lower performance, is tried. This can be obtained by either increasing the nodes' mobility by augmenting their ALAP or increasing the II. The former can be beneficial to overcome timing violations, the latter to alleviate resource overuse.

V. Kernels Partitioning

Kernels whose size exceed the capabilities of target architectures must be split into multiple sub-kernels, to be then handled by the scheduler. In this section, we present a formalization of such a partitioning problem. We also describe a state-of-the-art greedy algorithm [14] that we use as an evaluation baseline in Section VI-B.

A. Problem Formalization

Let $G\{V, E\}$ be a direct acyclic graph, where nodes V represent operations executed in an iteration of a computational kernel and edges E represent dependencies among operations. Back edges due to loop carried dependencies are only supported in a limited way, by preassigning all nodes in a circuit to the same subgraph.

Nodes $v \in V$ can be computation nodes or memory-access nodes. In the latter case, nodes have an attribute m_v that has an index unique for each array that the kernel processes. Different memory-access nodes can have the same m_v attribute if they read or write on the same array. We consider the simplified, but realistic, setting in which each array is mapped in a single and distinct memory cell on the architecture.

A cut S is a subgraph of G, where $S \subseteq G$, containing the nodes assigned to a sub-kernel. A partition P of G is a set of nonoverlapping S_i cuts covering all nodes of G. Let $\text{IN}(S_i)$ be the set of predecessor nodes of those edges crossing the cut boundary into S_i, and $\text{OUT}(S_i)$ be the set of predecessor nodes of edges crossing the cut boundary out of S_i.

The goal of partitioning is to assign each node of G to a cut, such that each cut does not violate memory, size, depth, and convexity constraints. A merit function is then used to discern lower- and higher-quality partitions among the valid ones. Here, we explain in detail the constraints and merit function.

1) *Cut Size:* To cope with the limited number of computation elements present in a CGRA mesh, the size of each cut should not exceed a threshold. Schedulability, in fact, sharply decreases as the size of the cut to be mapped increases, as can be seen in Fig. 14, in accordance with results shown in [20] and [36].

2) *Cut Depth:* During execution, CGRAs activate a control word at each clock cycle, so that cells can perform the proper operation at the proper time. CGRAs present a limited number of control words, in turn, limiting the maximum depth of DFGs that can be modulo-mapped onto them (in the simple example shown in Fig. 4, a simple DFG with maxDepth = 3 is executed in five steps, requiring five control words). Contexts are a costly hardware feature, impacting the size of the configuration memory and of the control logic.

3) *Cut Memory Footprint:* As discussed in Section I, after kernel fission a sub-kernel might require excessive data storage with respect to underlying hardware. Given a cut S_i, we define M_{S_i} as the set of all distinct m_v attributes of each $v \in S_i$. The cardinality of M_{S_i} indicates the internal memory requirement of a cut S_i. In addition, $\text{IN}(S_i)$ indicates the number of temporary arrays that are input to the sub-kernel, while $\text{OUT}(S_i)$ is the number of arrays that are generated by executing the sub-kernel. The size of these three elements must not exceed the amount of memory available in the hardware.

4) *Local and Global Convexity:* Local convexity is a property of a cut, imposing that no path through G exists that exits and reenters a valid cut S_i. The constraint ensures that all inputs to a cut can be ready when the cut is to be executed. Fig. 11(a) shows a non locally convex cut.

Global convexity is instead a property of a partition; it states that once all cuts of a partition are collapsed into single nodes, the resulting graph is acyclic. A globally convex partition allows for at least one order in that sub-kernels can be scheduled in sequence, as each cut S_i can be either a

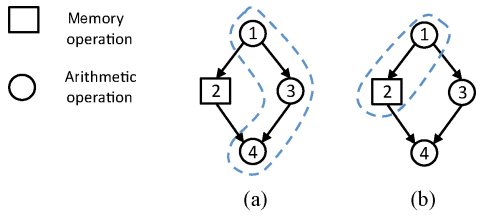


Fig. 11. Single cut identification. (a) Nonconvex cut. (b) Cut with one memory reference and two outputs.

predecessor or a successor of another cut S_j of the partition, but not both.

5) *Merit Function:* Kernel fission forces a sequential barrier in execution of sub-kernels on a CGRA accelerator. To ensure highest performance given the above-mentioned constraints, the number of sub-kernels should be minimized. A solution employing a small number of large sub-kernels also minimizes the overhead due to reconfiguration and transfer of the dataset in and out of the accelerator.

6) *Problem Formulation:* The kernel fission problem can now be formalized as follows.

Given a DFG representation of a kernel $G\{V, E\}$ and indexes of its memory references m_{v_i}, find a partition $P = \{S_1, S_2, ..., S_n\}$ of G such that:

1) $\forall\ S_i$, size of $S_i <$ MaxSize;
2) $\forall\ S_i$, depth of $S_i <$ MaxDepth;
3) $\forall\ S_i$, $|M_{S_i}| + \text{IN}(S_i) + \text{OUT}(S_i) <$ MaxMems;
4) all S_i are locally convex, P is globally convex;
5) $|P|$ is minimized.

B. Recursive Partitioning Algorithm

We use a set of existing algorithms [10], [11], initially proposed for ISE identification, and adapt them to the partitioning problem. As in that paper, we explored two methods: an iterative methodology and an exact one. The methods take in input a kernel DFG and architectural constraints, and output a complete DFG partitioning.

1) *Single cut Identification:* The input graph G is topologically sorted, where a node u precedes v in the order if Depth(u) < Depth(v). Binary recursion is then used to span an abstract search tree, shown in Fig. 12. At each bisection of the tree, two branches are considered, respectively, including or excluding a node $v \in G$ from a cut S. The cut S represents an element of the final partition, and the leaves of the tree represent all possible cuts—some of them invalid, i.e., overusing resources. The algorithm enumerates all valid cuts, and the most performant, according to a given metric (size, in this case), can be selected.

The size of the search tree thus constructed is exponential, but effective pruning can be performed to restrict the search space without sacrificing exactness. Two pruning conditions, specific to this problem formulation, examine the depth and the size of the cut when nodes are added to it: if MaxDepth or MaxSize are exceeded, adding further nodes to S by expanding the underlying search branch cannot result in a valid solution (it violates underlying-platform resources). Pruning related to MaxMems takes into account the numbers of inputs and

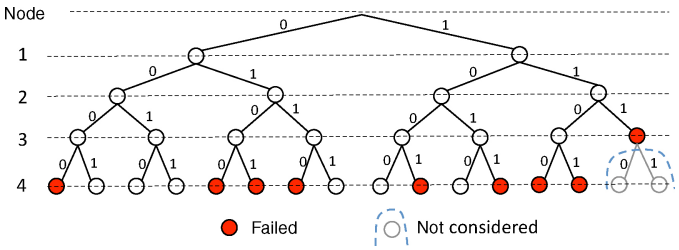


Fig. 12. Single cut identification: abstract search tree of the DFG in Fig. 11, considering MaxSize = 3, MaxMems = MaxDepth = 2. 0→node not included in cut, 1→node included.

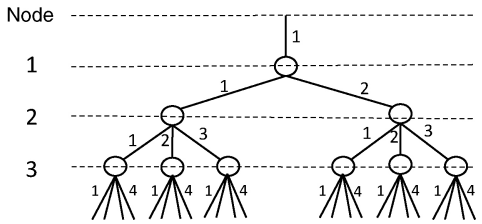


Fig. 13. Abstract search tree for multiple cuts identification.

outputs (as detailed in the original paper [10], [11]) and inclusion of memory nodes in S. Finally, nonconvex candidates cuts are discarded.

Fig. 12 shows the abstract search tree for the simple four-nodes DFG of Fig. 11. When a node of the tree corresponding to an invalid solution (i.e., overusing resources) is reached, the search-tree rooted at that node can be pruned away, thus improving search speed.

2) *Iterative and Exact Partitioning:* The method we propose for performing graph partitioning, called *iterative*, is to iteratively apply the algorithm for single cut identification, seen above. At every iteration, the largest cut within constraints is identified and selected, its nodes correspondingly removed from the input graph, and the algorithm for identifying the next largest cut called again. At every iteration, the size of the remaining graph decreases, and the partition cardinality increases.

A more expensive method for partitioning (called here *exact*) substitutes binary recursion with an N-ary one, where each branch corresponds to assigning a node to one of N cuts, resulting in an abstract search tree of the form shown in Fig. 13.

The lower bound on the number of cuts in a partition is $K = \lceil(|V|/\text{MaxSize})\rceil$, while the upper bound is $|V|$, which corresponds to assigning every node in G to a different cut. As discussed in Section V-A, partitions employing the smallest number of cuts are desirable. Therefore, a search for solutions employing the smallest possible number of cuts, K, is performed first. If no valid partition with cardinality K is found, the algorithm proceeds by seeking a valid partition with $K+1$, $K+2$, ..., $|V|$ cuts.

3) *Comparison:* The exact algorithm is guaranteed to find an optimal solution to the partitioning problem. However, its exponential complexity makes it intractable in some cases. In practice, the iterative methodology identifies very good partitions at a reasonable computational expense. One such

TABLE III
SCHEDULABILITY AND PERFORMANCE OF BENCHMARK DFG KERNELS SCHEDULED USING DIFFERENT METHODS

Benchmark	DFG Nodes[a] Before Clustering	DFG Nodes[a] After Clustering	Avg. Mapped[b]	Scheduling Method	Success (%)	Annealing Steps	Min II	II
conven	6 A – 2 M	3 R – 2 M	5 OP + 6 RT	Slack aware	100	23	1	1.00
				Slack fixed	100	32		1.00
				Slack oblivious	100	2749		1.22
autocorr	4 A – 2 M – 1 MU	2 R – 2 M – 1 MU	5 OP + 6 RT	Slack aware	100	56	1	1.07
				Slack fixed	100	86		1.12
				Slack oblivious	89	6058		1.56
aifirf	5 A – 2 M – 1 MU	3 R – 2 M – 1 MU	6 OP + 5 RT	Slack aware	100	37	1	1.04
				Slack fixed	100	152		1.34
				Slack oblivious	47	12 627		3.43
mpegcorr	9 A – 3 M	5 R – 3 M	8 OP + 13 RT	Slack aware	100	326	2	2.00
				Slack fixed	100	448		2.14
				Slack oblivious	0	–		–
iquant	8 A – 3 M – 2 MU	4 R – 3 M – 2 MU	9 OP + 13 RT	Slack aware	100	364	2	2.01
				Slack fixed	100	472		2.25
				Slack oblivious	0	–		–
fbital	13 A – 3 M	6 R – 3 M	9 OP + 22 RT	Slack aware	100	101	3	3.06
				Slack fixed	100	370		3.79
				Slack oblivious	0	–		–
viterbi_1	8 A – 8 M	3 R – 8 M	11 OP + 18 RT	Slack aware	100	5405	3	3.94
				Slack fixed	94	7155		4.28
				Slack oblivious	0	–		–
idct_1	6 A – 5 M – 4 MU	2 R – 5 M – 4 MU	11 OP + 16 RT	Slack aware	100	5059	2	2.87
				Slack fixed	100	5988		3.03
				Slack oblivious	0	–		–
dct_1	12 A – 6 M – 5 MU	4 R – 6 M – 5 MU	15 OP + 27 RT	Slack aware	100	11 964	2	4.47
				Slack fixed	90	18 320		5.93
				Slack oblivious	0	–		–

[a]A: ALU operations. R: RAC nodes. M: memory operations or nodes. MU: multiplier operations or nodes.
[b]OP: operation nodes (R+M+MU). RT: routing nodes.

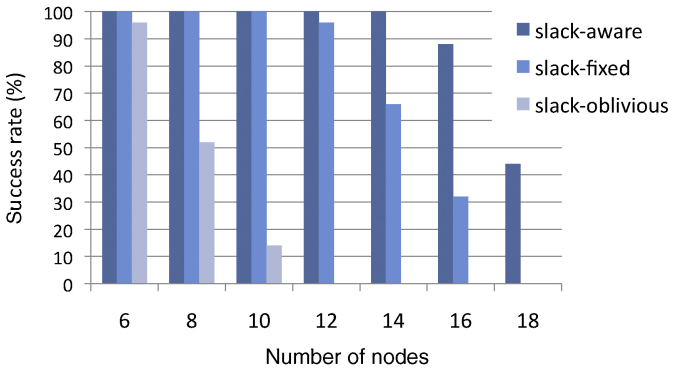


Fig. 14. Success rate of test DFGs using slack-aware, slack-fixed, and slack-oblivious modulo scheduling. Average over 100 graphs for each size.

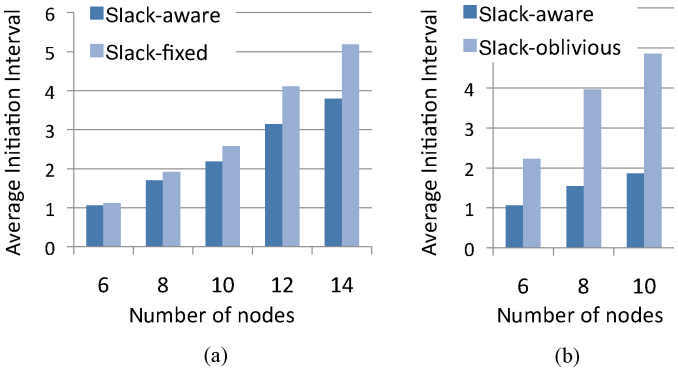


Fig. 15. Achieved II of successfully mapped test DFGs. (a) Slack aware versus slack fixed. (b) Slack aware versus slack oblivious.

partition is depicted in Fig. 20. The performance of both partitioning algorithms is detailed and analyzed in Section VI-B.

C. Greedy Partitioning

Here, we detail a state-of-the-art strategy proposed in the literature for partitioning: a cluster-based partitioning algorithm published in [14]. The paper presents a greedy algorithm of linear complexity: $O(|E| + |V|)$. This algorithm is used as a baseline to measure the efficiency of our proposed recursive partitioning.

Cluster-based partitioning performs a top–down sweep of the application DFG, and schedules to a cut S_i the "ready" node with maximum depth. At each iteration, the ready list is updated, adding those nodes whose predecessors have already been scheduled. The algorithm adds nodes to a cut until constraints are not violated. If a violation occurs, it is resolved by creating a new cut.

VI. EXPERIMENTAL RESULTS

To highlight the benefit of slack-aware scheduling and recursive partitioning, this section proceeds in three phases. First (Section VI-A), slack-aware scheduling is considered, targeting automatically generated and benchmark DFGs that can be directly mapped into a test CGRA mesh. Second (Section VI-B), different partitioning strategies are comparatively evaluated in terms of the quality of achieved results and computational effort required to retrieve them. Finally, in Section VI-C, partitioning and scheduling are joined in a single framework, considering applications whose size exceeds the capability of the target reconfigurable mesh and that must be therefore partitioned before their cuts can be scheduled.

A. Slack-Aware Scheduling

1) *Experimental Settings:* We first evaluate slack-aware scheduling by mapping automatically generated DFGs on the EGRA instance presented in Fig. 5. We considered three

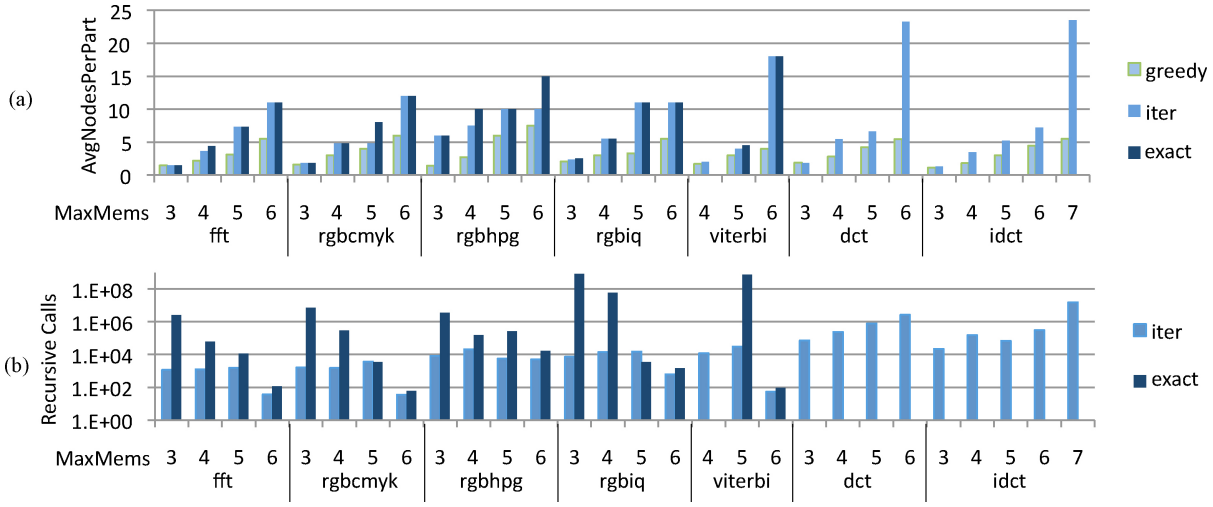


Fig. 16. (a) Partitioning quality using exact, iterative, and greedy algorithms (larger cuts are better). (b) Efforts required to reach solution using iterative and exact algorithms (recursive calls are proportional to runtime). Varying MaxMems with Maxsize = $|V|/2$, MaxDepth = ∞.

TABLE IV
SIMULATED ANNEALING STEPS TO REACH A VALID SLACK-AWARE SCHEDULING (AVERAGE)

No. of nodes	6	8	10	12	14	16	18
SA steps	198	2633	4707	7836	9705	14 781	18 488

scenarios. In the first (slack oblivious), each operation is set to consume an entire clock cycle. In the second (slack fixed), all operations are considered to have the same slack as the slowest one (the multiplier cell, as indicated in Table II); in this intermediate setting, a single hop can be chained after each computation. Finally, slack-aware scheduling exploits the slacks described in Table II, allowing for the corresponding number of chained routing hops.

We considered DFGs with diverse shapes and characteristics: nodes were set to have one or two predecessors, with 50% probability in each case; nodes' types were randomly assigned with a probability matching the composition of the target mesh (15% multiplications, 30% memory operations, 55% RAC operations). Five thousand simulated annealing cycles were performed before increasing the II; application mapping failed when II reached max(ALAP), a situation where loops are not pipelined at all.

2) *Slack-Aware Scheduling Maps More DFGs:* Data plotted in Fig. 14 shows the percentage of successful mappings for each DFG size using slack-aware, slack-fixed, and slack-oblivious strategies when performing modulo scheduling. The added routing flexibility allowed for more complex DFGs to be mapped; for example, all 14-nodes test DFGs were slack-aware scheduled, compared with 66% of test cases using a slack-fixed setting. None of these graphs were successfully mapped with a slack-oblivious strategy. The computation effort to converge to a solution was always reasonable, as shown in Table IV.[2]

[2]Scheduling of 18-nodes DFGs required less than 1 min of computation on a Intel 2.2 GHz core2duo system, which dropped to few seconds for 10-nodes DFGs.

3) *Slack-Aware Scheduling Achieves Better Mappings:* In addition to being able to map more DFGs, slack awareness also improves performance of mapped applications. Fig. 15(a) compares the average II of DFGs that were successfully mapped with slack aware and slack fixed. Similarly, Fig. 15(b) compares slack-aware and slack-oblivious mappings. In both cases, slack-awareness results in smaller II values, which corresponds to faster kernel execution.

4) *Real Benchmarks:* We also considered DFGs of computational kernels extracted from the EEMBC [37] benchmark suite, and scheduled them onto the EGRA instance described in Fig. 5; each kernel was mapped 100 times starting from different initial conditions.

Table III illustrates the number and type of operations in each DFG before and after ALU operations are clustered into RACs (in the second and third column, respectively). In the first six rows, the whole benchmark kernel was mapped onto the EGRA. In the case of viterbi, dct, and idct kernels, which exceeded resources, we used the iterative partitioning strategy detailed in Section V to extract the largest valid sub-kernel, and mapped it. The average size of the mapped DFG, including mapped routing nodes [see, for reference, Fig. 9(b)], is given in column 4. For each scheduling method, the success rate, the average number of simulated annealing steps to reach a valid solution, and the average achieved II is reported and compared to its lower bound minII [24].

Results are in line with the ones obtained for randomly generated DFGs: a slack-oblivious strategy even fails to map the three most complex kernels. Slack-aware scheduling reaches more performing solutions than a simpler slack-fixed strategy, obtaining smaller II (as much as 24% less in the case of the fbital benchmark).

B. Recursive Partitioning

To compare the proposed partitioning methodologies, we again considered kernels from the EEMBC benchmark suite. As opposed to the previous section, we focused on kernels whose size prevented their scheduling on the target EGRA

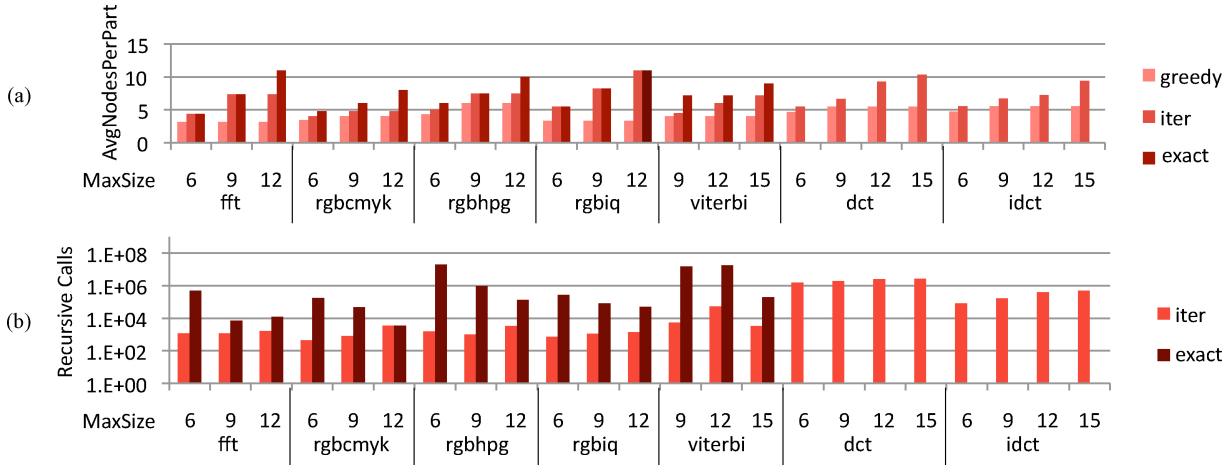


Fig. 17. (a) Partitioning quality using exact, iterative, and greedy algorithms (larger cuts are better). (b) Efforts required to reach solution using iterative and exact algorithms (recursive calls are proportional to runtime). Varying Maxsize with MaxMems = 5 for all benchmarks except viterbi, dct (MaxMems = 6), and idct (MaxMems = 7). MaxDepth = ∞.

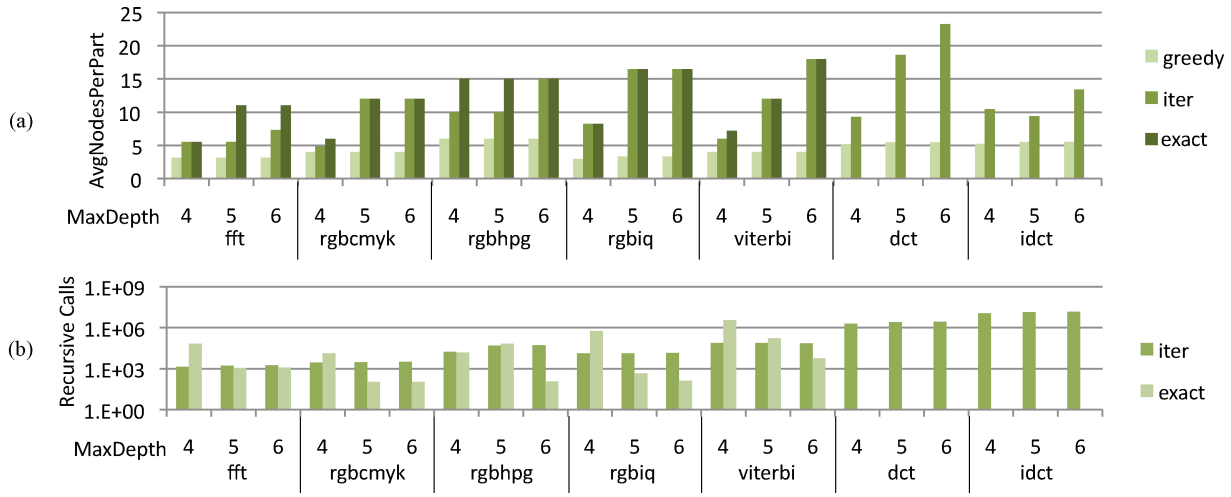


Fig. 18. (a) Partitioning quality using exact, iterative, and greedy algorithms (larger cuts are better). (b) Efforts required to reach solution using iterative and exact algorithms (recursive calls are proportional to runtime). Varying MaxDepth with MaxMems = 5 for all benchmarks except viterbi, dct (MaxMems = 6), and idct (MaxMems = 7). MaxSize = ∞.

TABLE V
CHARACTERISTICS OF BENCHMARK KERNELS INVESTIGATED FOR PARTITIONING

Benchmark	Nodes	Edges	Arrays	Array Accesses
fft	22	31	2	8
rgbcmyk	24	34	2	7
rgbhpg	30	37	2	10
rgbiq	33	40	2	6
viterbi	36	51	5	14
dct	93	126	3	16
idct	94	145	3	24

TABLE VI
RELATIVE CUT SIZE COMPARISON AGGREGATED BY BENCHMARK

Benchmark	Iterative/Greedy (%)	Iterative/Exact (%)
fft	89	−12
rgbcmyk	59	−15
rgbhpg	89	−17
rgbiq	172	−3
viterbi	113	−12
dct	100	–
idct	66	–
Average	98	−12

without partitioning them. Their characteristics are summarized in Table V.

Three rounds of experiments were conducted, varying the requirement relative to: 1) maximum storage; 2) control logic; and 3) computation capability, respectively (Figs. 16–18). Executing its implementation on a standard computer, the iterative algorithm converged at most in a matter of seconds; on the contrary, it was not possible to obtain exact solutions in a reasonable time for the two most complex kernels (dct and idct).

When both exact and iterative did complete, results were similar, in many cases identical; identified partitions were along the lines of solutions an expert programmer would identify, as the dct partition obtained by the iterative algorithm presented in Fig. 20 illustrates.

On the other hand, in all but the simplest cases, the greedy methodology trailed well behind the ones based on recursive searches, resulting in smaller and more numerous sub-kernels. A graphical comparison of the methods, presented in Figs. 20–19 for a partition of the dct kernel, shows how the

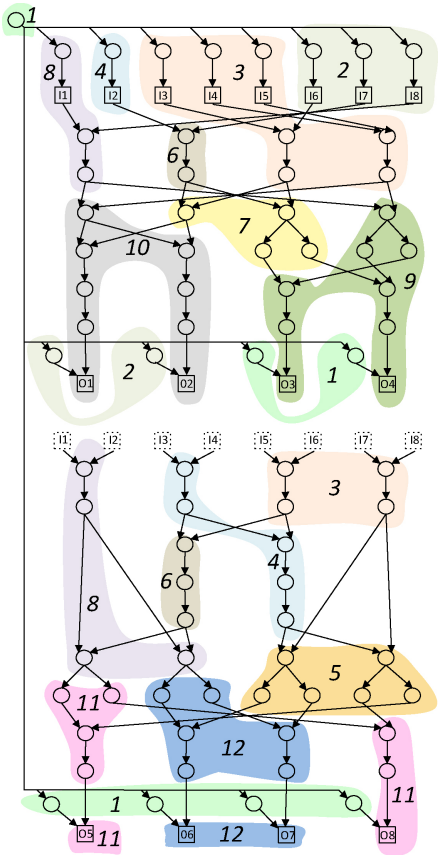


Fig. 19. Partition of dct, greedy strategy, with MaxSize = 30, MaxMems = 7, and MaxDepth = 10, requiring 12 cuts.

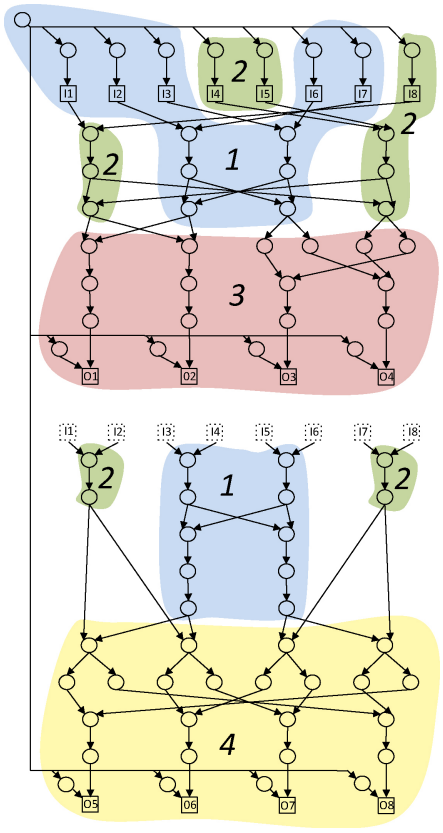


Fig. 20. Partition of dct, iterative strategy, with MaxSize=30, MaxMems=7, and MaxDepth = 10, requiring four cuts.

lack of flexibility of the greedy approach leads to a much worse partition given the same constraints with respect to the iterative and exact solutions. The greedy cluster-based algorithm was particularly ineffective when big, complex cuts could be identified and exploited, as is the case of the viterbi, dct, and idct kernels.

An interesting observation can be made regarding computation time: while exact partitioning converged quite fast in a few selected cases, it was not able to do it consistently, presenting a hugely different required effort in different settings. Particularly demanding were searches presenting a big gap between the upper threshold in number of cuts ($\lceil(|V|/\text{MaxSize})\rceil$) and the actual cuts necessary for a valid solution. In Fig. 16, the experiments relative to rgbiq with MaxMems = 3 exemplifies this effect. An iterative strategy, instead, is able to converge to a solution in few seconds in all cases.

The second column in Table VI compares the relative size of cuts obtained by the greedy and the iterative methodologies, subdivided by benchmark and aggregated on all performed experiments. The metric is computed as

$$(\text{Avg}_{S\_\text{size}}(\text{iter}) - \text{Avg}_{S\_\text{size}}(\text{greedy}))/\text{Avg}_{S\_\text{size}}(\text{greedy}).$$

It can be noted that cuts obtained by iterative partitionings are on average twice the size of the ones identified by a cluster based, and as much as 172% larger in the case of rgbiq. Comparing in a similar way, exact and iterative partitionings result in just 12% difference in average size of cuts (and only 3% in the best case).

C. Mapping of Partitioned Kernels

The three most complex kernels (viterbi, dct, and idct) considered in Section VI-B are here further investigated, to showcase their performance when scheduled on the EGRA instance described in Fig. 5.

In the following experiments, kernels are partitioned adding memory nodes for edges crossing partition boundaries to hold temporary data (as exemplified in Fig. 3). Resulting cuts are then scheduled on the target EGRA instance.

Execution time of whole kernels can then be derived by adding up the execution time of the different sub-kernels constituting them, where each sub-kernel executes for Exec_time = (II $*$ Iter) + Depth clock cycles. The dct and idct kernels iterate eight times, while viterbi has an iteration count of 32.

We employ three different settings: performance of iterative partitioning and slack-aware scheduling compared with two less-capable strategies. In the first case, iterative partitioning is linked to a slack-fixed modulo scheduler, while in the second partitioning, it is performed greedily, resulting cuts being slack-aware scheduled. In all cases, the maximum memory footprint of sub-kernels MaxMems is the number of memory cells embedded in the target reconfigurable mesh, while the maximum number of nodes per sub-kernel MaxSize is limited by the biggest values resulting in a successful scheduling. In case a scheduling fails due to oversized cuts, the application is repartitioned with a less stringent constraint (i.e., a smaller cut size). This process converges rather quickly—few itera-

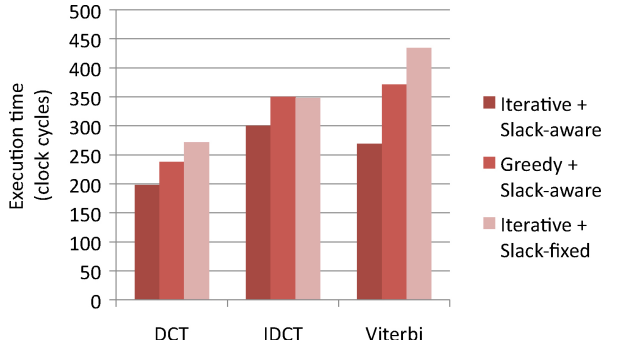


Fig. 21. Execution time of partitioned kernels using different strategies.

TABLE VII
MAPPING PERFORMANCE OF DIFFERENT PARTITIONING OR SCHEDULING STRATEGIES (VITERBI KERNEL)

	Cut	Init. Int.	Depth	Exec. Cycles
Iterative/slack aware	1	3	6	102
	2	3	5	101
	3	2	2	66
	Sum	8	13	269
Greedy/slack aware	1	2	4	68
	2	1	3	35
	3	2	3	67
	4	2	3	67
	5	2	3	67
	6	2	3	67
	Sum	11	19	371
Iterative/slack fixed	1	2	3	67
	2	3	5	101
	3	4	3	131
	4	3	5	101
	5	1	2	34
	Sum	13	18	434

tions are necessary—as schedulability increases sharply when decreasing the size of cuts, as shown in Fig. 14.

As plotted in Fig. 21, the combination of recursive partitioning and slack-aware scheduling is beneficial to increase performance for all three benchmarks. On one side, the increased flexibility offered by slack-awareness makes it possible to map larger sub-kernels for a given mesh size, and to obtain more aggressive mappings. On the other, recursive partitioning leads to better-formed sub-kernels than simpler strategies, in turn resulting in faster kernel execution, with a speedup of up to 27% for the considered benchmarks.

Table VII details the partitioning of the viterbi kernel and the execution performance of the resulting cuts in the three above-mentioned settings. Data shown in Fig. 21 and Table VII only refer to kernel execution, neglecting data transfer overhead. Such overhead would put larger cuts at an even bigger advantage.

VII. CONCLUSION

This paper introduced methodologies to map complex DFGs, extracted from computational kernels, onto CGRAs. The problem was tackled from two points of view. On one side, slack-aware scheduling was introduced to allow for higher utilization of resources, considering registered and unregistered connections among CGRA tiles. On the other, a novel loop fission technique was detailed to partition complex kernels into cuts according to architectural constraints.

Slack-awareness leverages differences in computation times of operations to allow for computation and routing operations to be chained in the same clock cycle, increasing schedulability and execution performance for coarse-grained meshes supporting modulo scheduling. It is particularly beneficial in case of meshes composed of heterogeneous elements and/or complex cells, which most likely present differences in actual critical path, depending on cell type and performed operation.

The proposed partitioning strategy detailed two methodologies, one iterative and one exact, based on recursive searches over abstract trees. The methods were inspired by a previous work on ISEs [11], modified to tackle the different scenario of efficient loop fission of kernels, in the context of systems comprising a CGRA accelerator. Experimental results showed that the proposed iterative partitioning resulted in average sub-kernel size that was only marginally smaller than in the exact case, and twice the size than the one resulting by applying a state-of-the-art cluster-based greedy algorithm [14]. Moreover, the low computational complexity of the iterative partitioning, with respect to the exact one, makes it applicable to more complex cases.

Effective partitioning and scheduling strategies, when used in concert, can maximize runtime performance of complex computational kernels, taking full advantage of the constrained and heterogeneous resources typically present on CGRA meshes.

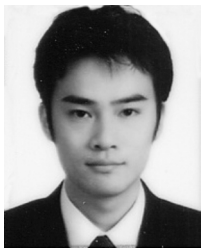
REFERENCES

[1] *Altera*. (2011) [Online]. Available: http://www.altera.com
[2] *Xilinx*. (2011) [Online]. Available: http://www.xilinx.com
[3] P. Bonzini, G. Ansaloni, and L. Pozzi, "Compiling custom instructions onto expression-grained reconfigurable architectures," in *Proc. Int. Conf. Compilers, Architect. Syn. Embedded Syst.*, Oct. 2008, pp. 51–60.
[4] R. Hartenstein, "A decade of reconfigurable computing: A visionary retrospective," in *Proc. Des., Automat. Test Eur. Conf. Exhibit.*, Mar. 2001, pp. 642–649.
[5] M.-H. Lee, H. Singh, G. Lu, N. Bagherzadeh, F. J. Kurdahi, E. M. C. Filho, and V. C. Alves, "Design and implementation of the morphoSys reconfigurable computing processor," *J. Very Large Scale Integr. Signal Process. Syst.*, vol. 24, nos. 2–3, pp. 147–164, Mar. 2000.
[6] V. Baumgarte, G. Elhers, F. May, A. Nuckel, M. Vorback, and M. Weinhardt, "PACT-XPP: A self-reconfigurable data processing architecture," *J. Supercomput.*, vol. 26, no. 2, pp. 167–184, Sep. 2003.
[7] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "Exploiting loop-level parallelism on coarse-grained reconfigurable architectures using modulo scheduling," in *Proc. Des., Automat. Test Eur. Conf. Exhibit.*, Sep. 2003, pp. 255–261.
[8] J. Lee, K. Choi, and N. Dutt, "An algorithm for mapping loops onto coarse-grained reconfigurable architectures," in *Proc. ACM Conf. Languages, Compilers, Tools Embedded Syst.*, Jun. 2003, pp. 183–188.
[9] K. Kennedy and K. Kinley, "Maximizing loop parallelism and improving data locality via loop fusion and distribution," in *Proc. 6th Int. Workshop Languages Compilers Parallel Comput.*, vol. 768. 1994, pp. 301–320.
[10] K. Atasu, L. Pozzi, and P. Ienne, "Automatic application-specific instruction-set extensions under microarchitectural constraints," in *Proc. 40th Des. Automat. Conf.*, Jun. 2003, pp. 256–261.
[11] L. Pozzi, K. Atasu, and P. Ienne, "Exact and approximate algorithms for the extension of embedded processor instruction sets," *IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst.*, vol. CAD-25, no. 7, pp. 1209–1229, Jul. 2006.
[12] G. Ansaloni, P. Bonzini, and L. Pozzi, "Design and architectural exploration of expression-grained reconfigurable arrays," in *Proc. 6th Symp. Applicat. Specific Process.*, Jun. 2008, pp. 26–33.

[13] G. Ansaloni, P. Bonzini, and L. Pozzi, "Heterogeneous coarse-grained processing elements: A template architecture for embedded processing acceleration," in *Proc. Des., Automat. Test Eur. Conf. Exhibit.*, Mar. 2009, pp. 542–547.
[14] K. Purna and D. Bhatia, "Temporal partitioning and scheduling data flow graphs for reconfigurable computers," *IEEE Trans. Comput.*, vol. 48, no. 6, pp. 579–590, Jun. 1999.
[15] T. Miyamori and K. Olukotun, "REMARC: Reconfigurable multimedia array coprocessor," *IEICE Trans. Inform. Syst.*, vol. 82, no. 2, pp. 389–397, 1999.
[16] P. Heysters and G. Smit, "Mapping of DSP algorithms on the MONTIUM architecture," in *Proc. Parallel Distrib. Process. Symp.*, Apr. 2003, pp. 1–6.
[17] Y. Kim, M. Kiemb, C. Park, J. Jung, and K. Choi, "Resource sharing and pipelining in coarse-grained reconfigurable architecture for domain-specific optimization," in *Proc. Des., Automat. Test Eur. Conf. Exhibit.*, 2005, pp. 12–17.
[18] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev, "Architectural exploration of the ADRES coarse-grained reconfigurable array," in *Proc. Reconfigurable Comput.: Architect., Tools Applicat.* (Series Lecture Notes in Computer Science), vol. 4419. Berlin, Germany: Springer, Jun. 2007, pp. 1–13.
[19] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins, "DRESC: A retargetable compiler for coarse-grained reconfigurable architectures," in *Proc. IEEE Int. Conf. Field-Programmable Technol.*, Dec. 2002, pp. 166–173.
[20] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jeyapaul, and Y. Paek, "SPKM: A novel graph-drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," in *Proc. Asia South Pacific Des. Automat. Conf.*, Jan. 2008, pp. 776–782.
[21] A. Hatanaka and N. Bagherzadeh, "A modulo scheduling algorithm for a coarse-grain reconfigurable array template," in *Proc. IEEE Int. Parallel Distrib. Process. Symp.*, Mar. 2007, pp. 1–8.
[22] M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proc. Des., Automat. Test Eur. Conf. Exhibit.*, Mar. 2006, pp. 363–368.
[23] H. Park, K. Fan, M. Kudlur, and S. Mahlke, "Modulo graph embedding: Mapping applications onto coarse-grained reconfigurable architectures," in *Proc. Int. Conf. Compilers, Architect., Syn. Embedded Syst.*, Oct. 2006, pp. 136–146.
[24] R. B. Rau, "Iterative modulo scheduling," *Int. J. Parallel Process.*, vol. 24, no. 1, pp. 2–64, Feb. 1996.
[25] T. Toi, N. Nakamura, Y. Kato, T. Awashima, K. Wakabayashi, and L. Jing, "High-level synthesis challenges and solutions for a dynamically reconfigurable processor," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 2006, pp. 702–708.
[26] Y. Park, H. Park, and S. Mahlke, "CGRA express: Accelerating execution using dynamic operation fusion," in *Proc. Int. Conf. Compilers, Architect. Syn. Embedded Syst.*, Oct. 2009, pp. 271–280.
[27] F. Barat, M. Jayapala, T. Vander Aa, R. Lauwereins, G. Deconinck, and H. Corporaal, "Low power coarse-grained reconfigurable instruction set processor," in *Field Programmable Logic and Application*, vol. 2778, P. Y. K. Cheung and G. Constantinides, Eds. Berlin/Heidelberg, Germany: Springer, 2003, pp. 230–239.
[28] M. Kaul and R. Vemuri, "Optimal temporal partitioning and synthesis for reconfigurable architectures," in *Proc. 35th Des. Automat. Conf.*, Feb. 1998, pp. 389–396.
[29] H. Liu and D. Wong, "Network flow based circuit partitioning for time multiplexed FPGAs," in *Proc. Int. Conf. Comput.-Aided Des.*, Nov. 1998, pp. 497–504.
[30] B. Kernighan and S. Lin, "An efficient heuristic procedure for partitioning graphs," *IEEE Trans. Comput.*, vol. C-27, no. 11, pp. 1064–1068, Nov. 1978.
[31] P. Yu and T. Mitra, "Scalable custom instructions identification for instruction set extensible processors," in *Proc. Int. Conf. Compilers, Architect. Syn. Embedded Syst.*, Sep. 2004, pp. 69–78.
[32] J. M. P. Cardoso and M. Weinhardt, "XPP-VC: A C compiler with temporal partitioning for the PACT-XPP architecture," in *Proc. 12th Int. Conf. Field-Program. Logic Applicat.*, Sep. 2002, pp. 207–226.
[33] G. Ansaloni, P. Bonzini, and L. Pozzi, "EGRA: A coarse grained reconfigurable architectural template," *IEEE Trans. Very Large Scale Integr. Syst.*, vol. 19, no. 6, pp. 1062–1074, Jun. 2011.
[34] C. Shields, "Area efficient layouts of binary trees in grids," Ph.D. dissertation, Dept. Comput. Sci., Univ. Texas Dallas, Dallas, 2001.
[35] P. Hart, N. Nilsson, and B. Raphael, "A formal basis for the heuristic determination of minimum cost paths," *IEEE Trans. Syst. Sci. Cybern.*, vol. 4, no. 2, pp. 100–107, Jul. 1968.
[36] G. Ansaloni, K. Tanimura, L. Pozzi, and N. Dutt, "Slack-aware scheduling on coarse grained reconfigurable arrays," in *Proc. Des., Automat. Test Eur. Conf. Exhibit.*, Mar. 2011, pp. 1–4.
[37] *EEMBC*. (1997) [Online]. Available: http://www.eembc.org

Giovanni Ansaloni (M'08) received the M.S. degree in electronic engineering from the University of Ferrara, Ferrara, Italy, in 2003, the M.A.S. degree in embedded systems design from ALaRI, Lugano, Switzerland, in 2005, and the Ph.D. degree in informatics from the University of Lugano, Lugano, in 2011.

He is currently a Post-Doctoral Researcher with the École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. His current research interests include algorithms for reconfigurable processor customization and real-time signal processing for wireless body sensors.



Kazuyuki Tanimura received the B.E. degree in electronics, information, and communication engineering in 2006 and the M.E. degree in computer science in 2008 from Waseda University, Tokyo, Japan. He is currently pursuing the Ph.D. degree in computer science with the University of California at Irvine, Irvine.

His current research interests include embedded system architectures and design methodologies.

Laura Pozzi (M'01) received the Ph.D. degree in computer engineering from Politecnico di Milano, Milan, Italy, in 2000.

She is currently an Associate Professor with the Faculty of Informatics, University of Lugano, Lugano, Switzerland. She was a Post-Doctoral Researcher with École Polytechnique Fédérale de Lausanne, Lausanne, Switzerland, a Research Engineer with STMicroelectronics, Santa Clara, CA, and an Industrial Visitor with the University of California at Berkeley, Berkeley. Her current research interests include automating embedded processor customization, high-performance compiler techniques, and innovative reconfigurable fabrics.

Prof. Pozzi was the recipient of the Best Paper Award in the embedded systems category at the Design Automation Conference in 2003. She is currently an Associate Editor of the IEEE TRANSACTIONS ON COMPUTER-AIDED DESIGN and is or has been a Technical Program Committee Member of several international conferences in the areas of compilers and architectures for embedded systems.

Nikil Dutt (F'08) received the Ph.D. degree in computer science from the University of Illinois at Urbana-Champaign, Urbana, in 1989.

He is currently a Chancellor's Professor with the University of California at Irvine, Irvine. He is the author of multiple books. His current research interests include embedded systems, electronic design automation, computer architecture, optimizing compilers, system specification techniques, distributed embedded systems, formal methods, and brain-inspired architectures and computing.

Prof. Dutt was the recipient of Best Paper Award at several conferences. He currently is an Associate Editor of the *ACM Transactions on Embedded Computing Systems* and the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS. He was the Editor-in-Chief of the *ACM Transactions on Design Automation of Electronic Systems*. He was also with the steering, organizing, and program committees of several premier computer-aided design and embedded system design conferences and workshops. He is an ACM Distinguished Scientist and an IFIP Silver Core Awardee.