Selective Flexibility: Creating Domain-Specific Reconfigurable Arrays

Mirjana Stojilović, *Member, IEEE,* David Novo, *Member, IEEE,* Lazar Saranovac, *Member, IEEE,* Philip Brisk, *Member, IEEE,* and Paolo Ienne, *Member, IEEE*

***Abstract*—Historically, hardware acceleration technologies have either been application-specific, therefore lacking in flexibility, or fully programmable, thereby suffering from notable inefficiencies on an application-by-application basis. To address the growing need for domain-specific acceleration technologies, this paper describes a design methodology (i) to automatically generate a domain-specific coarse-grained array from a set of representative applications and (ii) to introduce limited forms of architectural generality to increase the likelihood that additional applications can be successfully mapped onto it. In particular, coarse-grained arrays generated using our approach are intended to be integrated into customizable processors that use application-specific instruction set extensions to accelerate performance and reduce energy; rather than implementing these extensions using application-specific integrated circuit (ASIC) logic, which lacks flexibility, they can be synthesized onto our reconfigurable array instead, allowing the processor to be used for a variety of applications in related domains. Results show that our array is around 2× slower and 15× larger than an ultimately efficient ASIC implementation, and thus far more efficient than field-programmable gate arrays (FPGAs), which are known to be 3–4× slower and 20–40× larger. Additionally, we estimate that our array is usually around 2× larger and 2× slower than an accelerator synthesized using traditional datapath merging, which has, if any, very limited flexibility beyond the design set of DFGs.**

***Index Terms*—Datapaths, domain-specific customization, flexibility, FPGA routing, reconfigurable arrays.**

I. Introduction

EMBEDDED systems often use specialized hardware accelerators to improve performance and reduce energy consumption [3], [4], especially in areas such as signal and video processing, communications, and computer vision, among others. Algorithms can be applied to merge the hardware implementations of accelerators that are always used independently in order to reduce area [1], [2]. This approach can create multioperational datapaths for a fixed set of applications; however, it alone does not provide greater flexibility, as the user of such a system may wish to accelerate additional applications that were not considered at design time.

For a given application, the ideal accelerator minimizes the disparities in terms of performance, energy consumption, and area in comparison to an ASIC implementation of the accelerated functionality; flexibility, which is required to accommodate late design changes or new applications in the same domain, impose unavoidable overheads that must be minimized. The requirements for flexible accelerators are not well formulated; however, it is also clear that many system designers and project managers in the semiconductor industry would desire such solutions. FPGAs, for example, provide high flexibility, but suffer from incredibly poor logic density, even when system designers make good usage of DSP blocks, block RAMs, and transceivers. Thus, it is hardly surprising that despite many efforts, no high-volume commercially successful product, to date, has successfully embedded FPGAs into ASIC design flows. At the other end of the spectrum, datapath merging [1], [2] offers limited overhead in the form of multiplexers inserted into the datapath, but offers no flexibility in terms of being able to accelerate applications that were not conceived at design time. The objective of this paper is to introduce flexibility into the design process of domain-specific accelerators, in a context in which only a subset of the applications that require acceleration are known at design time. We have developed a methodology to generate a domain-specific coarse-grained array from a set of data flow graphs (DFGs) from an application domain, and then to introduce additional flexibility. The approach guarantees that all of the initial set of DFGs can map successfully onto the array, and tries to increase the likelihood that structurally similar DFGs from similar domains can map successfully as well. This problem is loosely defined and poorly articulated in the general case, as there may be little or no similarity between the DFGs used to generate the datapath and the additional DFGs that map onto it. Therefore, it is important that the designers have a good understanding of the properties of the application domain under consideration, as well as the limitations of the approach, when using it to generate a domain-specific array. Our experiments demonstrate that the vast majority of DFGs

Manuscript received May 31, 2012; revised November 11, 2012; accepted November 21, 2012. Date of current version April 17, 2013. This work was supported in part by the Ministry of Education and Science of the Republic of Serbia under Grant TR32043 and by the Swiss National Science Foundation under SCOPES Grant IZ74Z0 128293/1. This paper was recommended by Associate Editor D. Chen.

M. Stojilović is with the Institute Mihailo Pupin, University of Belgrade, Volgina 15, Belgrade 11060, Serbia (e-mail: mirjana.stojilovic@pupin.rs).

L. Saranovac is with the Department of Electronics, School of Electrical Engineering, University of Belgrade, Belgrade 11120, Serbia (e-mail: laza@el.etf.rs).

P. Brisk is with the Department of Computer Science and Engineering, University of California Riverside, Riverside, CA 92521 USA (e-mail: philip@cs.ucr.edu).

D. Novo and P. Ienne are with the School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne (EPFL), CH-1015 Lausanne, Switzerland (e-mail: david.novobruna@epfl.ch; paolo.ienne@epfl.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.2012.2235127

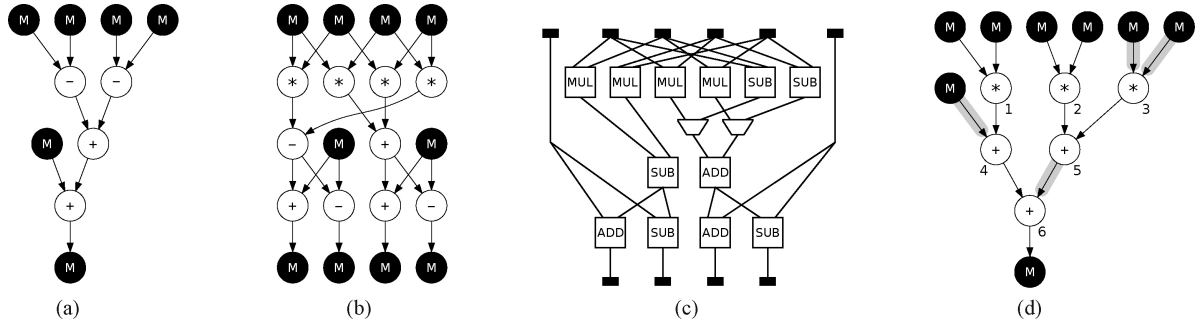

Fig. 1. Limitations of classic datapath merging [1], [2]. Two sample data-flow graphs from (a) a two-pixels SAD and (b) radix-2 FFT butterfly. The M nodes represent accesses to some form of memory (register files, stream buffer, etc.) and are not part of the datapaths. A typical result of merging these two graphs would result in (c). Unfortunately, the data-flow graph (d), a 3×3 image convolution, cannot be mapped onto the merged datapath despite the fact that the resources are almost the same and the connectivity is very similar. The interconnections shaded in (d) are missing in the merged datapath (c).

can be mapped onto the arrays that are generated by our algorithm for the other DFGs of the same application domain. Besides, the results show that our flexible arrays are around $2\times$ larger and $2\times$ slower than an accelerator synthesized using well-known datapath merging technique [1]. At the same time, these domain-specific arrays are only about $2\times$ slower and $15\times$ larger than an ASIC implementation of a single DFG in isolation, and thus far more efficient than FPGAs, which are known to be 3–4$\times$ slower and 20–40$\times$ larger [5].

II. Flexibility, or The Lack Thereof...

In order to illustrate the problem that motivates this paper, Fig. 1(a) and (b) shows two DFGs. The former corresponds to a sum of absolute differences (SAD), widely used for motion estimation in video compression, while the latter corresponds a radix-2 butterfly, which is a building block of the fast Fourier transform (FFT). Fig. 1(c) shows the result of merging the two datapaths, with multiplexers inserted; this datapath can be configured to execute either DFG [1], [2].

Now, suppose that we want to execute an image convolution DFG, shown in Fig. 1(d), which arguably belongs to the same application domain. Although the merged datapath in Fig. 1(c) contains all of the necessary resources, its fixed connectivity is insufficient to implement the desired functionality: the shaded paths cannot be mapped onto the datapath.

This paper, an extension to our previous work [6], strives to introduce additional flexibility during the process by which the first two DFGs are merged. The cost of doing so is the addition of extra operators and interconnect resources, which increase the area and delay of the resulting datapath; however, the additional resources also increase the likelihood that new DFGs, not considered in the datapath design, can be mapped onto it.

III. Selective Flexibility

We understand flexibility as the ability to capture and implement the computational structures that are characteristic of a specific application domain. We call the flexibility selective because these computational structures are characterized, and thus restricted, by 1) the type of operations, 2) their number, and 3) their interconnections. In terms of these parameters, we expect application domains to be relatively homogeneous:

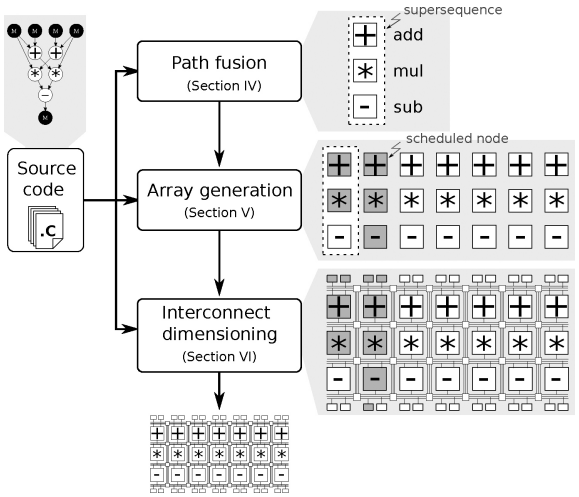

Fig. 2. Our flow to synthesize domain-specific datapaths. First we create from various application DFGs a column of operators appropriate to implement them all. Then we replicate the column to create a regular array to place DFG nodes. Finally, we add an FPGA-like statically configured routing network to route complete DFGs.

FFTs, DCTs, and similar signal-processing primitives use similar operators for essentially similar computations, irrespective of various important implementation choices. In these cases, a high degree of generality can be achieved at a reasonable area overhead and a limited performance cost. Our technique analyzes different applications input by the designer, attempts to distill the essential computational structures and connectivity in a dedicated reconfigurable array, and then makes it possible to map new applications on the datapath. Of course, the generality of the resulting datapath will heavily depend on how well the original applications cover the spectrum of computational structures of the target domain.

Fig. 2 illustrates the fundamental steps in our solution to capture the key features of a number of applications, represented by their DFGs. Firstly, we fix the type and sequence of operations supported in the datapath by defining a *supersequence*. The supersequence is an ordered sequence of operators, which includes all the sequences of operations present in the input applications. Two operators form a sequence if there exist a sequence of vertices connected by edges between these two operator nodes. These sequences are assumed to be inherent to the target domain and, once a supersequence is created, we consider it very likely that all sequences found in new

DFGs belonging to the same domain will be already included. Secondly, we fix the total number of operators implemented in our reconfigurable datapath in the shape of a rectangular array. The supersequence is the basic construction block: our array is composed by $N_r \times N_c$ operators, where each column is composed by the operators in the supersequence (N_r, the number of rows, is thus the length of the supersequence). Then, N_c (the number of columns) is the number of times that the basic column needs to be replicated for a successful mapping of the training set of DFGs. The interconnect among the operators is inspired by FPGAs, but uses 32-bit buses rather than bit-based connections [7] to reduce the amount of configuration storage. The reconfiguration of the datapath is achieved by shifting in configuration bits and storing them in configuration memory cells. It is essentially achieved as in any FPGA and in many coarser grain statically programmed arrays, and is not addressed in detail in this paper. Applications are statically mapped on the datapath, much as in an FPGA; reconfiguration happens only before execution of one of the applications.

By defining such a datapath array, we expect to provide the computational structures that enable a high degree of generality for a particular domain at a reasonably small overhead (unused operators, redundant interconnect, etc.). In the following sections, we describe in detail the different parts of our technique to generate automatically the datapath from a collection of DFGs, before proceeding to an empirical evaluation of its capabilities.

IV. Path Fusion

The basic building block of our regular datapath is the column of the array, which contains all types of operators occurring in the training set of DFGs. We call *path fusion* the process of defining the number of the operators and how they are sequenced in the column. To perform path fusion, we first enumerate all paths of input DFGs, i.e., all sequences of DFG vertices (operators) connected by edges, such that the first node in the sequence is an input port while the last node is an output port. Then, we create a single sequence of nodes (supersequence) such that all those paths are subsequences of it, i.e., they can be derived from the supersequence by deleting some elements, without changing the order of the remaining elements. The described process of designing the datapath column reminds us of one of the classical problems in computer science—finding the shortest common supersequence (SCS). However, to capture generality at a reasonable cost the supersequence should not only be short, but also have small area. We have implemented two algorithms. The first is a modified version of a well-known algorithm for finding a SCS [8], and the second is based on reusing the maximum area common subsequence (MACSeq) metric of existing graph-merging algorithms [1].

A. Algorithm Based on a Modified Weighted Majority Merge Algorithm for Finding the Shortest Common Supersequence

Branke *et al.* [8] formulated the problem of finding the SCS as follows: given a finite set of strings L over an alphabet Σ, find a string of minimal length that is a supersequence of each string in L. A string is a finite sequence of symbols chosen from an alphabet, or, in our case, a path in the application DFG. One of the most well known heuristics for finding a SCS is the majority merge (MM) [9], which builds the supersequence starting from the empty string as follows. It looks at the first symbol in each string in L and chooses the most frequent one, removes it from the strings where it has been found as the first symbol, and appends it to the supersequence. The process is repeated until all strings are emptied. However, MM does not take into account different string lengths, and it is clear that one should focus first on the longer ones. Branke *et al.* [8] propose three new heuristics. Our algorithm builds upon their heuristic H1, namely weighted majority merge (WMM), which makes decisions based on the sum of weights of symbol occurrences at the front of the strings. Originally, the weight of a symbol is the length of the string suffix, excluding the symbol itself. Our heuristic differs in that we strive to find a minimum area supersequence rather than the minimum length one, thus we calculate the weight of the symbol as the sum of the area of the operator implementing that symbol and of the areas of all following symbols in the same path. If there are multiple candidates, the algorithm takes the one with the longer string suffix. The algorithm steps are as follows:

1) initialize the supersequence to the empty sequence and enumerate all paths of each DFG;
2) choose the candidate operator op by computing the sum of weights for all operators at the front of the paths and by finding the one that maximizes this sum, and append op to the supersequence;
3) update all paths by removing op from the front;
4) repeat steps 2) and 3) until all paths are empty.

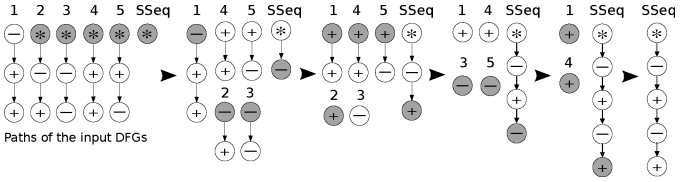

Fig. 3. Steps in our algorithm for finding the shortest common supersequence based on weighted majority merge heuristic. The operators highlighted in gray are selected for appending to the supersequence (SSeq). Once appended, they are removed from the paths.

Fig. 3 illustrates the process of finding the supersequence (SSeq) for the DFGs shown in Fig. 1(a) and (b). Excluding for simplicity one-node paths, the first DFG has one path $P_1 = \{S, A, A\}$, while the second DFG has four different paths, $P_2 = \{M, S, A\}$, $P_3 = \{M, S, S\}$, $P_4 = \{M, A, A\}$, and $P_5 = \{M, A, S\}$. Here, S represents subtraction, A addition, and M multiplication[1]. Assuming that the areas of the different operators are related as $M > S > A$, the maximum weighted sum is found for the multiplier M, which is then inserted into the SSeq and removed from the paths $P_{2,3,4,5}$. Next, the maximum weighted sum is found for the subtractor, and it is appended to the SSeq and removed from $P_{1,2,3}$. Then, the

[1]Additions and subtractions are differentiated here for illustration purposes. In the real implementation, our algorithm assigns them to the same operator, an adder–subtractor, thus increasing datapath flexibility at negligible cost.

adder is selected, appended to the SSeq and removed from $P_{1,2,4,5}$. Next, between an adder and subtractor, the subtractor is selected due to its higher area ($S > A$). Finally, only the adder remains, so the final supersequence generated by the algorithm becomes $SS_{\text{eq}} = \{M, S, A, S, A\}$.

B. Algorithm Based on Reusing the Maximum Area Common Subsequence Metric of Graph-Merging Algorithms

Besides the algorithm presented in the previous subsection, we propose another heuristic, where we reuse the maximum area common subsequence (MACSeq) metric of other existing graph-merging algorithms [1] to give priority in the path fusion to those paths that contain the MACSeq of operators. This way we strive to minimize the total area of the supersequence. The algorithm is as follows.

1) Enumerate all paths of each input DFG and group them into multiple sets depending on their length (number of nodes).
2) Starting from the set having the longest paths, perform pairwise search for the MACSeq between paths, using the algorithm for finding the longest common subsequence (LCS) [10] and calculating the LCS area as the sum of areas of its operators. Since the result of the LCS search algorithm depends on the order of input paths P_i and P_j, $i \neq j$, we run it for both (P_i, P_j) and (P_j, P_i). If there are multiple pairs of paths sharing the MACSeq, this step reports the pair that was found first. Additionally, it always reports the order of paths P_i and P_j used by LCS search algorithm for which the MACSeq was found.
3) Perform path fusion on the ordered pair of paths (P_i, P_j) sharing the MACSeq. To fuse the two paths, we begin by aligning all their respective nodes belonging to the MACSeq (see Fig. 4). Then, the fused path is initialized to the MACSeq, aligned as in paths P_i and P_j. Finally, we add to the fused path all nodes of P_i and P_j not belonging to the MACSeq. For this, we add the nodes into the fused path in the same relative position with respect to the MACSeq as in the original path, adding first the nodes of P_i and, after them, those of P_j. Once the fused path is obtained, it replaces P_i and P_j in the set of paths to fuse.
4) Repeat 2) and 3) until only one path is left in the set.
5) Move the resulting path to the next set containing the paths of smaller length and repeat 2) to 5) until only one path is left: the supersequence.

In practice, the fused path converges quickly into the supersequence, as the shorter paths are most likely already contained in previously fused longer paths.

In Fig. 4, we describe how this algorithm is implemented on the same DFGs used for Fig. 3: As explained in Section IV-A, we find the following five paths from the two DFGs: $P_1 = \{S, A, A\}$, $P_2 = \{M, S, A\}$, $P_3 = \{M, S, S\}$, $P_4 = \{M, A, A\}$, and $P_5 = \{M, A, S\}$. In this case, our algorithm groups all the paths in the same set because they all have the same length. To find the MACSeq, one should remember that the areas of the different operators are related as

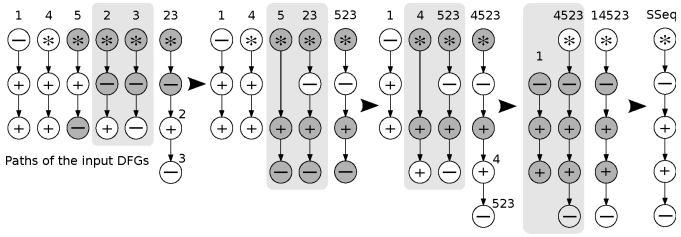

Fig. 4. Steps in the algorithm based on reusing MACSeq metric of graph-merging algorithms. Paths highlighted in gray are selected for merging. Nodes in darker gray form MACseq.

$M > S > A$. Accordingly, the MACSeq corresponds to $\{M, S\}$, which is contained in P_2, P_3, and P_5. The first pair that reports MACSeq is (P_2, P_3). Therefore, we fuse P_2 with P_3, resulting in $P_{23} = \{M, S, A, S\}$. The next found MACSeq is $\{M, A, S\}$ found for the pair of paths (P_5, P_{23}), resulting in fused path $P_{523} = \{M, S, A, S\}$. Similarly, the next path selected for fusing is P_4, so the fused path becomes $P_{4523} = \{M, S, A, A, S\}$. Finally, the only remaining path, P_1, is selected. This path is already included in P_{4523}, so the supersequence SSeq becomes $P_{14523} = \{M, S, A, A, S\}$.

C. Algorithm Complexities

The algorithms for finding the supersequence involve enumerating all paths of the input DFGs. In the worst-case, this may lead to enumerating an exponential number of paths, and thus reaching exponential algorithm complexity. Yet, most DFGs derived from real applications do not exhibit this property, despite having relatively large number of nodes. However, if the number of paths in a graph were to be exponential, one could design a heuristic to limit the number of enumerated paths. For example, one could enumerate the unique paths in a graph using a straightforward application of topological sort, and then insert a thresholding mechanism to decide between proceeding with exhaustive or limited enumeration, depending on the number of paths found. In our experiments, we observe a relatively low (in the order of a millisecond) execution time of both supersequence generation algorithms, regardless of the size of the input DFG. Therefore, we do not constrain the number of enumerated paths.

V. Array Generation

Once the supersequence is generated, a column of corresponding operators is formed and replicated N_c times, to create our datapath array. To find a minimum value of N_c such that all input DFGs can be mapped, we initially assume to have unlimited size of the array. Then we try mapping input DFGs one by one, and finally calculate the minimum number of columns needed to assure successful mapping for all those DFGs. This value is N_c. Of course, to provide generality beyond the size of the input set of DFGs, our algorithm can apply some oversizing factor to enlarge the minimum value of N_c. The amount of oversizing factor can be provided by the designer or devised automatically by the algorithm, based on the characteristics of the input DFGs, as follows. To find a small oversizing factor promising good generality, we measure the minimum oversizing which would have been necessary to

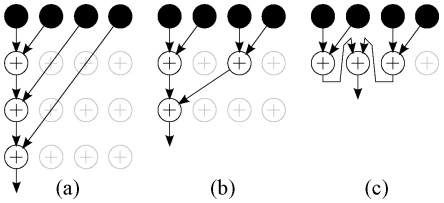

Fig. 5. Optimization of row utilization for binary trees. (a) Part of a typical DFG with accumulation of multiple partial results. (b) Chain of operators transformed into a binary tree. (c) Tree nodes assigned to one of the previous rows to minimize the tree height and thus improve row utilization.

implement any of the given DFGs as if it were not part of the initial set. More specifically, let us assume that the designer has provided a set G containing N DFGs. For each DFG D_i ($1 \leq i \leq N$) from G, we create a set G_i containing all DFGs except D_i. Then we do path fusion on all paths in G_i to find the supersequence. The difference between the number of columns in the array needed to provide enough space to map D_i and the number of columns in the array sufficient to map all DFGs in G_i is the candidate oversize factor $oversize_i$. The maximum of all candidate $oversize_i$ factors, where i is in the range $1 \leq i \leq N$, is the final column oversize factor. Yet, in our experiments, presented in Section VII, we do not oversize the number of columns in the array.

VI. Interconnect Dimensioning

Once the datapath is generated, we need to add routing resources. We use standard FPGA-like interconnections [11] organized in word-size buses [7] inside horizontal and vertical routing channels. To measure the minimal number of routing tracks needed, we place and route every DFG. Of course, good placement is key, since the minimum channel width to guarantee routability heavily depends on it. While developing the algorithm, we first implemented node-to-node connections between operators in the neighboring rows of the array to make the routing network resemble the edges in graph layouts of all input DFGs, and FPGA-like vertical routing channels for connecting nodes placed in distant rows. Then, to make regular routing network between neighboring rows, we would replicate and distribute within the horizontal channel every type (depending on the length and direction of the connection) of connections that were suggested by the graph layout to be used in that channel. However, the possibility to route a new graph on the array was poor, since the routing network was not rich enough. Therefore, we decided to use FPGA-like interconnections in both horizontal and vertical channels.

A. Placement

To leverage on the topological regularity existing in DFGs belonging to the same domain we select a placement algorithm that mimics effective graph drawing algorithms, as other researchers have done before us [12]. These algorithms usually keep graph edges as short as possible, to minimize the number of edge crossings and emphasize symmetries. Such features are particularly important for our array, which is clearly built in such a way as to require a top-down layout of the graphs and with complete horizontal homogeneity. We use the

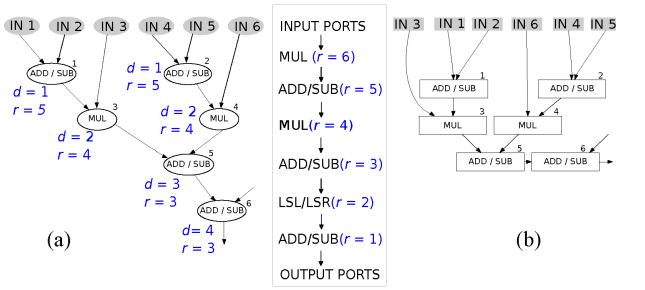

Fig. 6. Assigning nodes of a subgraph for placement. (a) Subgraph with node depths d marked. Nodes having the same depth and performing the same operation are assigned to the same row (rank r). Nodes are always assigned lower rank than their predecessors, unless they are a part of a binary tree (Node 6). (b) Same subgraph after placement by `dot`.

algorithm implemented in `dot` [13], which is an open source tool for laying out hierarchical drawings of directed graphs. For simplicity, we actually use directly `dot` to implement the placement. To `dot`, we give first a description of the supersequence (the target datapath column) and then the DFG nodes grouped by rows to place them in. Each node in the supersequence is assigned a rank, where *rank* refers to the row in the datapath array created by replicating that node: the first node in the SSeq has rank N_r (where N_r equals the number of rows in the datapath), while the last node in the SSeq has rank one. We use the rank to constrain the layout so that nodes are placed in rows with the appropriate operators.

To decide the row in which to place a node, we regroup nodes by their depths. Since nodes can belong to multiple paths, we define the depth of a node as the maximum length of a path from input ports to the node, among all paths containing that node. Then, we apply the following top-down approach, where top refers to the highest-ranked row.

1) Group nodes by depth. Start from the group of nodes having the minimum depth d, i.e., the inputs.
2) For all nodes in the selected group, do the following.
   a) If the node is a part of a binary tree, assign it to the row with the same operator type and the rank equal or lower than predecessor rows (Fig. 5).
   b) If the node is not a part of a binary tree, assign it to the row with the correct operators having lower rank than predecessor rows.
3) Move to the group of nodes having depth $d + 1$.
4) Repeat steps 2) and 3) until all nodes are assigned to a row.

In DSP applications it is often the case that nodes belong to a binary tree, due to accumulation of partial results. Therefore, step 2) of the placement algorithm helps reducing the tree height, thus increasing the overall row utilization. Notice that, by construction of the supersequence and of the array, all nodes can always be placed greedily on the rows of the array as described above. Due to the binary tree optimization, some rows may never be used by any of the input DFGs, and are then automatically removed from the array to conserve area (Fig. 5). Removing rows from the array this way has no effect on its ability to support new sequences of operators belonging to other applications, as long as the resources available are sufficient.

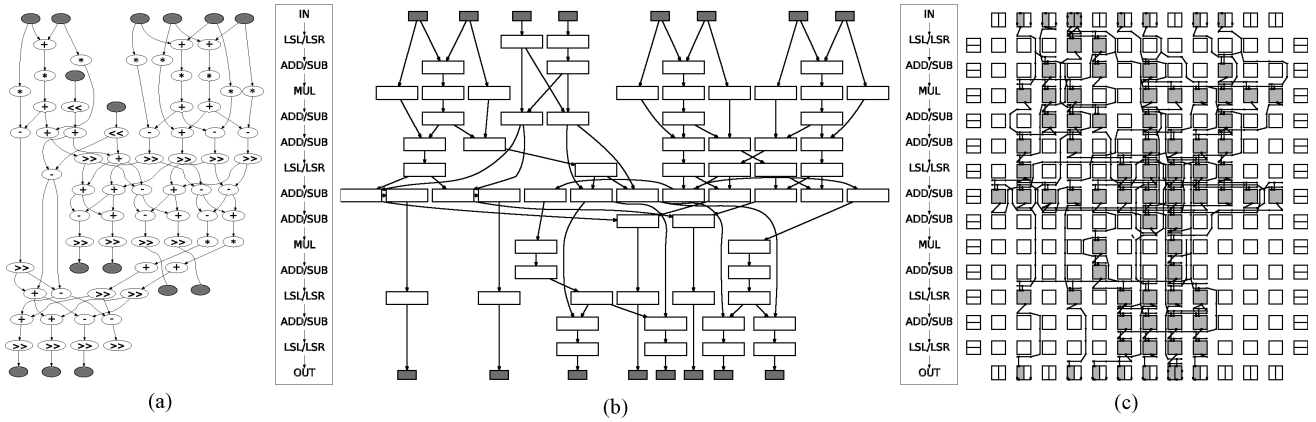

Fig. 7. Placement process. The DFG D_{15} (Table I) in (a) is laid out (b) with appropriate constraints and parameters to suggest a detailed placement on the array. After snapping the horizontal placement to the columns, the final placement (c) is achieved (shown after routing by VPR in the figure).

In Fig. 6, a subgraph of DFG D_7 used in the experimental evaluation (Section VII) is shown. Nodes $1, 2, 5$, and 6 perform addition/subtraction. Nodes 3 and 4 perform multiplication. Let us assume that the supersequence obtained by merging D_7 with other DFGs is the sequence $\{M, AS, M, AS, L, AS\}$ where M stands for multiplication, AS for addition/subtraction, and L for left/right shift. The first node in the supersequence has rank 6 (there are 6 rows in the datapath), while the last node has rank 1. Node depths are marked on the subgraph shown on the left—nodes 1 and 2 have the lowest depth, and perform operation AS. They are assigned the highest ranked row of adders/subtractors, the row with rank $r = 5$. Nodes 3 and 4 have depth $d = 2$. They can be assigned to the row with rank $r = 6$ or the row with rank $r = 4$; however, following the top-down approach, they are assigned to the row with rank $r = 4$ because it is lower than the rank of the rows of predecessor nodes 1 and 2. Similarly, node 5 is assigned the row with rank $r = 3$. Node 6 belongs to a binary tree and thus its predecessor, node 5, performs the same operation. Therefore, to minimize the tree height, node 6 is assigned the row with rank $r = 3$, the same as the predecessor.

Once we give to `dot` the supersequence and DFG nodes with their ranks assigned, `dot` is forced to place operators only within the rows, without the possibility to move any operator from one row to another. After the placement, horizontal node coordinates are further scaled and rounded to represent columns of the reconfigurable array. First, the leftmost node in the graph placed by `dot` is placed into the left most column of the array. All other nodes are placed relative to this node, trying to keep the distances among nodes proportional to those in `dot` placement. The center coordinates of nodes are rounded to the closest integer value, to represent the coordinates of array columns. To avoid two nodes occupying a single column, we rely on the characteristic of `dot`, which is not to overlap nodes. Therefore, we set the rectangular shape of the nodes and the width of nodes such that their centers, when rounded to the closest integer, can never have the same value. However, if the nodes are too distant and thus cannot fit the array after rounding, we perform rescaling of the `dot` placement. Knowing the maximum possible distance between nodes in the array and the distance suggested by `dot`, we calculate the scaling factor and multiply all node coordinates by it, except the left most node coordinate, since this node is always placed in the left most column of the array. Finally, we round the coordinates to the closest integer again. After rescaling, it could happen that two nodes would desire to occupy a single column; our algorithm then redistributes nodes within columns so that there are no conflicts, at the same time trying to keep the distances among nodes as proportional as possible to the distances in the suggested placement by `dot`. An example of placement by `dot` is shown in Fig. 7.

B. Routing

To route the design, and to find the minimum channel width necessary to successfully route a placed DFG, we use VPR, a classic open-source FPGA tool for placement and routing of parameterized architectures [11], which we adapt to our purposes. Since VPR is naturally meant for FPGA bit-based connectivity and our reconfigurable array uses a uniform bitwidth of 32 bits, we assume that each wire in VPR actually represents a 32-bit bus connection, as all bits in the bus will follow the same route [7]. Additionally, we represent all DFG nodes as 2-input 1-output configurable logic blocks composed of a single subblock and assign 2 input/output ports per column of the array [7]. For all DFGs, we run VPR with the appropriate placement calculated through `dot` (see Section VI-A), the netlist, and the appropriate architecture description. Since we expect that our regular and ordered placement requires mostly the short-distance communication, we choose to use only the routing segments of length one. These short segments provide higher chances for successful routing (i.e., favor generality) while keeping the channel width as low as possible. Additionally, we set VPR to use constant width routing channels regardless of the routing direction. After routing all DFGs, VPR reports the minimum channel width for which a legal routing solution can be found for each of them. We take the maximum of these values to be the channel width of the resulting datapath, as it is the smallest value that admits a legal place-and-route solution for all of the benchmarks used in the generation process. At this point, the array is completely specified. Similarly to the array oversizing described in Section V, the designer has the possibility to define a channel-width oversizing factor

to increase the routability of the final datapath beyond the minimum; the cost is the additional area. In our experiments, presented in the next section, we do not oversize the routing channels.

VII. Experimental Results

In this section, we assess the performance of our method. First we compare the two algorithms for finding an area efficient supersequence. Then we measure the generality of our domain-specific arrays and provide an insight into the drawbacks of the current implementation. Additionally, we analyze the effects of grouping the various domains on the generality and on the total area of the arrays. Finally, we compare the area of the array and critical path delay of applications when mapped onto it with respect to ASIC, FPGA, and a well known datapath merging technique [1].

Although we do not perform experimental evaluation of energy consumption of the domain-specific arrays, it is clear that the methodology itself inherently promises good performance and low energy consumption due to two factors. Firstly, at the level of the computational units, we implement them using standard cells and thus they are as high performance and power efficient as they can be in a semicustom design flow, and certainly significantly better than in FPGAs. Secondly, at the level of interconnections among nodes, the methodology strives to achieve short and regular routing, therefore helping shortening the critical path and keeping power consumption to a minimum while using minimal resources.

We selected 19 different DFGs from applications available in benchmarks and commercial libraries [14]–[18] covering various classic signal and image processing computations (FFT, DCT, IDCT, FIR, IIR, autocorrelation, etc.). On all DFGs, we perform loop unrolling using different unrolling factors, to generate DFGs of various size. To achieve the best array utilization, we unroll DFGs so that they require the similar number of columns in the array for high area operators, such as multipliers. Therefore, we unroll until the maximum number of multipliers per depth (Section VI-A) for every DFG is similar. Table I lists all 19 DFGs, their description, the loop unrolling factor, and the number of nodes in every DFG. Selected DFGs were divided into various groups. G_{1A}, G_{1B}, G_{1C}, and G_{1D} include DFGs belonging to similar computational domains. Group G_{1A} contains all correlations, G_{1B} FIR and IIR filters, G_{1C} all FFTs, and G_{1D} all DCTs/IDCTs. Groups G_{2A} to G_{2F} comprise all combinations of any two domains, while groups G_{3A} to G_{3D} comprise all combinations of any three domains. Finally, group G_{4A} comprises all four domains, and thus, all the DFGs. Table II shows the distribution of DFGs D_1 to D_{19} (described in Table I) among the groups.

A. Comparing the Performance of the Algorithms for Finding the Area-Efficient Supersequence

To compare the two algorithms for finding the area-efficient supersequence (Section IV), we run them on all sets of DFGs in Table II, and report the ratio between the area of the supersequence generated using the algorithm based on reusing the MACSeq and the area of the supersequence generated using the modified weighted majority merge algorithm. To estimate the area of every operator, we did synthesis, placement, and routing using a 65 nm standard cell library. The area of the supersequence is the sum of areas of its nodes (operators). The sorted area ratios are shown in Fig. 8. The results show that the algorithm based on reusing the MACSeq metric is superior, since the generated supersequences are at least as area-efficient as those generated by the modified WMM algorithm, for all groups. This may be attributed to the fact that MACSeq heuristic is most directly related to the goal of minimizing area because it performs a greedy selection based on the actual richest mergeable sequences, while WMM bases the same selection on the available opportunity (cumulative area of the units after the merged nodes) without really assessing whether this opportunity will translate in actual merging. Finally, we have decided to use the MACSeq heuristic for the rest of the experimental evaluation of our methodology.

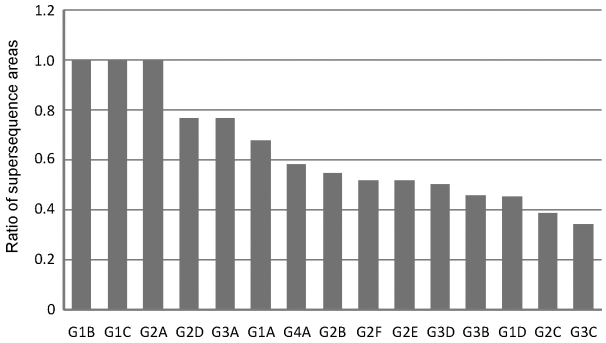

Fig. 8. Ratio between the area of the SSeq generated using the algorithm based on reusing the MACSeq and the area of the SSeq generated using the modified WMM algorithm. The former algorithm achieves superior results, and is thus used in the rest of the experimental evaluation of our methodology.

To estimate the overhead in the column length compared to the longest path in dataflow graphs, we perform the following experiment. For each of the groups G_{1A} to G_{4A}, we generate a domain-specific array using our methodology and the MACSeq algorithm. Then, we measure the supersequence length and compare it to the length of the longest path found in DFGs belonging to a group. The results show that in 6 out of 15 groups (40% of all cases) the supersequence is actually shorter than the longest path by at least 11% up to 50%. Therefore, our algorithm finds supersequences which are not significantly longer than the longest path in the considered DFGs. Additionally, the tree height minimization procedure is the reason why the array column length is shorter than the longest path, because it removes unused rows of the array. However, in the remaining nine cases, the supersequence was longer than the longest path, but only up to 25%.

B. Generality

For each of groups G_{1A} to G_{4A}, we measure the *generality* of the created reconfigurable array as follows: if N is the number of DFGs in group G, we remove in turn each DFG $D \in G$ from G itself and create the array from the remaining $N-1$ DFGs. Then we try to map D onto the datapath following the same place&route flow used for generation (Section VI) with the exception that the channel width is now known and fixed. The generality for group G is thus defined as the ratio of the number of successfully mapped excluded DFGs in the

TABLE I
DATA-FLOW GRAPHS COVERING CLASSICAL SIGNAL AND IMAGE PROCESSING COMPUTATIONS [14]−[18]

DFG	Name	Description	Loop Unrolling factor	Nodes
D_1	DSPLIB_C64_autocor_UNROLL12_BB_2	Autocorrelation	12	24
D_2	DSPLIB_C67_dotp_cplx_UNROLL3_BB_1	Complex Dot Product	3	24
D_3	IMGLIB_C64_corr3x3_UNROLL3_BB_2	3 × 3 Correlation for 8b Data	3	18
D_4	IMGLIB_C64_sobel3x3_UNROLL2_BB_1	16b Sobel 3 × 3	2	28
D_5	DSPLIB_C64_fir_cplx_UNROLL4_BB_2	16b Complex FIR	4	32
D_6	DSPLIB_C64_fir_lms_UNROLL16_BB_1	16b Least Mean Square Adaptive Filter	16	40
D_7	DSPLIB_C64_fir_sym_UNROLL8_BB_2	16b Symmetric FIR Filter	8	24
D_8	DSPLIB_C64_iir_UNROLL4_BB_1	IIR Filter	4	22
D_9	DSPLIB_C64_fftr4_UNROLL2_BB_3	16x16 Radix 4 DIF FFT	2	80
D_{10}	DSPLIB_C67_fftmix_BB_2	Forward FFT with Mixed Radix	1	40
D_{11}	DSPLIB_C67_fftr2_UNROLL4_BB_3	Forward FFT with Radix 2 and DIT	4	40
D_{12}	EEMBC_dct_BB_1	DCT h264 Encoder Library	1	42
D_{13}	EEMBC_idct_BB_1	IDCT h264 Encoder Library	1	42
D_{14}	ExPRESS_mpeg_idct, BB_1	32b, Inverse 2-D DCT	1	55
D_{15}	ExPRESS_mpeg_idct, BB_4	32b, Inverse 2-D DCT	1	64
D_{16}	IMGLIB_C64_dct_BB_2	8×8 Block FDCT with Rounding	1	60
D_{17}	IMGLIB_C64_dct_BB_8	8×8 Block FDCT with Rounding	1	61
D_{18}	IMGLIB_C64_idct_BB_2	IDCT on 8×8 DCT Coeffient Blocks	1	59
D_{19}	IMGLIB_C64_idct_BB_8	IDCT on 8×8 DCT Coeffient Blocks	1	77

TABLE II
GENERALITY, ARRAY SIZE, CHANNEL WIDTH, AND AREA UTILIZATIONS FOR VARIOUS GROUPS OF BENCHMARKS

Group Name	Group Size	DFGs in the Group	Generality (%)	Gen.′ (%)	Gen.″ (%)	Array Size $N_r \times N_c$	Channel Width	Max (Avg) Area Utilization (%)	Routing Area/ Array Area (%)
G_{1A}	4	D_1–D_4	50	50	75	3 × 15	4	58 (51)	28
G_{1B}	4	D_5–D_8	50	50	50	7 × 19	6	71 (30)	40
G_{1C}	3	D_9–D_{11}	67	67	67	5 × 24	6	78 (54)	44
G_{1D}	8	D_{12}–D_{19}	75	88	75	13 × 12	4	48 (45)	38
G_{2A}	8	D_1–D_8	75	75	75	7 × 19	6	71 (29)	40
G_{2B}	7	D_1–D_4, D_9–D_{11}	86	86	86	5 × 24	6	78 (39)	44
G_{2C}	12	D_1–D_4, D_{12}–D_{19}	83	83	83	13 × 15	4	39 (26)	38
G_{2D}	7	D_5–D_8, D_9–D_{11}	71	71	71	8 × 24	6	52 (28)	41
G_{2E}	12	D_5–D_8, D_{12}–D_{19}	83	83	92	15 × 19	6	38 (19)	45
G_{2F}	11	D_9–D_{19}	82	82	82	13 × 24	4	33 (20)	38
G_{3A}	11	D_1–D_{11}	82	82	82	8 × 24	6	52 (24)	41
G_{3B}	16	D_1–D_8, D_{12}–D_{19}	88	88	94	15 × 19	6	38 (17)	45
G_{3C}	15	D_1–D_4, D_9–D_{19}	87	87	87	13 × 24	4	33 (18)	38
G_{3D}	15	D_5–D_{19}	87	93	93	15 × 24	4	30 (16)	35
G_{4A}	19	D_1–D_{19}	89	95	95	15 × 24	4	30 (15)	35

N experiments to the total number of DFGs N. The results are shown in Table II: generality is higher than 75% in most of the cases. The possible reasons a DFGs can fail mapping are the following: insufficient number of columns or ports in the array, failed routing due to insufficient channel width, or limited generality of the supersequence leading to insufficient number of rows in the array for successful mapping. Some of these reasons caused mapping failures in our experimental setup and are discussed below.

1) *Limited Generality of the Supersequence:* In each group at least one DFG has failed the top-down placement (see Section VI-A) due to lack of rows in the datapath. Since the main idea is that the SSeq should capture the computational characteristics of the domain, its generality highly depends on how well the input DFGs represent the domain.

2) *Insufficient Channel Width:* In group G_{4A}, DFG D_6 was unsuccessfully routed. The channel width in the datapath was set to four buses per horizontal/vertical routing channel, which was the minimum needed for successful routing of the remaining DFGs in the group. Our algorithm allows the designer to define the channel width oversizing factor (see Section VI-B), and thus increase the routability at the expense of higher area allocated for routing resources. However, due to the regularity of the routing network, only a small increase in channel width provides high increase in the probability for successful routing. After increasing the channel width for additional two buses beyond the minimum, which is conservative because VPR restricts the number of buses to an even number, DFG D_6 was successfully routed, while the generality for group G_{4A} increased from 89% to 95%. To estimate how often constraining the channel width to a

minimum needed value leads to routing failures, we measure generality when the channel width is not limited. The column labeled Gen.′ in Table II summarizes the results and shows that the generality increased in 20% of all tests.

3) *Insufficient Number of Columns or Ports:* Since we fix the array size fairly tightly based on the input DFGs, if the excluded DFG needs even a little more space to fit into the datapath, mapping may be impossible. One way to avoid this problem is to use the automatic column oversizing introduced in Section V, or to manually define the total number of columns of the final array. By increasing the number of columns beyond the minimum one gets more computational and routing resources, and thus an increased generality, at the cost of increasing the array size. Additionally, increased array size allows DFG mappings to resemble more the `dot` mapping, which is usually scaled horizontally by our algorithm to fit narrow arrays. The problem of insufficient number of ports is analogous, because we assume two input/output ports per column of the array (Section VI-B). To estimate how often constraining the number of columns and ports to a minimum needed value leads to mapping failures, we measure generality when the array size is not limited. The column labeled Gen.″ in Table II summarizes the results and shows that the generality increased in 33% of all tests.

C. Array Size and Utilization

For each group in Table II, we generated a reconfigurable array following our methodology. Table II shows the obtained array dimensions (the number of rows N_r and the number of columns N_c) and minimum channel width. The minimum array size was found for group G_{1A}, storing the DFGs belonging to the same domain: 3 rows × 15 columns. Predictably, the maximum array size was found for group G_{4A}, storing all domains at once: 15 rows × 24 columns. The number of columns in an array is the minimum needed to enable successful mapping of all DFGs in the domain. Therefore, when two or more domains are joined, the new array will have the number of columns equal to the maximum of all values N_c found for each domain separately. For example, groups G_{1B} and G_{1C} need 15 or 24 columns, respectively, for successful mapping of all their DFGs. Hence, the union of those two groups G_{2D} needs at least 24 columns in the array. To find the array area utilization for every group, we estimated the area of the array and the area occupied by DFGs when mapped onto it. For that purpose, we synthesized, placed, and routed individually all the operations found in the DFGs using the gate implementations of a 65 nm standard cell library. For each operation, we provided the possibility to use either the direct output or a registered output. Added pipelining registers are organized as bypassable registers placed after every functional unit, as in FPGAs. To estimate the routing area, we used VPR. Since VPR does not natively support bus-based connections, we used an appropriate technology configuration file along with the real number of wires required. This approach conservatively overestimates the routing area because VPR assumes that each wire can be routed independently, whereas our array uses bus-based interconnects. The maximum and average area utilization per groups are shown in Table II. Maximum area utilization ranges from 30%, for the array generated for all domains at once (G_{4A}), up to 78% for the array generated for group G_{1C}, storing the DFGs belonging to the same domain. Highest values of average area utilizations are found for groups G_{1A} and G_{1C}, each storing the DFGs belonging to a particular domain. The more domains get mixed within a group, the more average area utilization decreases, indicating that our methodology is optimized to provide area-efficient domain-specific arrays.

D. Routing Network Characteristics

The routing network is comprised of wiring segments of length one, distributed among vertical and horizontal routing channels, which all have constant channel width. The ratio of the area dedicated for routing resources to the total array area is shown in Table II, and it is in the range 28–45%, where higher ratios were found for the arrays having higher channel width. Interestingly, these results are significantly better than what is reported in programmable logic devices—according to the paper by Feng and Kaptanogly from Actel Corporation [19] up to 90% of a Programmable Logic Device chip is occupied by the programmable interconnect, including wires, switches, and configuration bits.

Since the array uses only four or six buses per routing channel (Table II), it is not immediately clear if having different segment lengths in routing channels (as in FPGAs) might help increasing the generality and/or decreasing the array area. To get a sense, we reran the experiments, but with a routing network using both segments of the length one and of the length two (note that our regular and ordered placement requires less long-distance communication). When using mixed segment lengths the array requires channel width six to successfully route all DFGs (four is no longer enough). Consequently, increased routing opportunities lead to increased generality (in two groups out of 15), but also to a slight increase in the array area of 5–7% (in 7 out of 15 groups). Additionally, various segment lengths in some cases lead to reduced routing opportunities and thus decreased generality (in two groups out of 15). Only for those arrays where channel width was not affected by introducing segments of the length two the total area of the array was decreased for about 10% (in 8 out of 15 groups). Therefore, it seems that there is no true and clear superiority of using segments of different length.

E. Effects of Grouping the Domains on Generality and Area

The technique we present in this paper aims at generating domain-specific arrays. To verify that it is well tailored for that purpose, we have performed the following tests: we generated the arrays for each individual domain, for all combinations of two or three domains, and for all domains at once. For each of these arrays we measured the generality and area. The results are plotted in Figs. 9 and 10. The former figure shows that increasing the number of domains at the input leads to increased generality, due to larger and thus richer input set of DFGs. Twelve out of 15 groupings achieve generality higher than 70%. Only two groups with four DFGs, G_{1A} and G_{1B} (in Table II), achieve generality equal to 50%, due to a very small set of input DFGs. Fig. 10

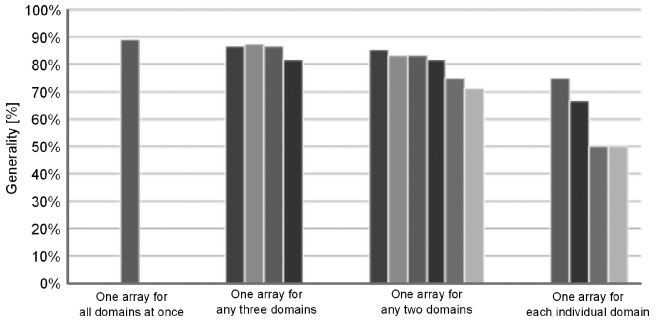

Fig. 9. Generality of the array for different combinations of domains. Increasing the number of domains leads to increased generality, due to increased input set size. However, even for single domains the generality is reasonably high, at least 50%.

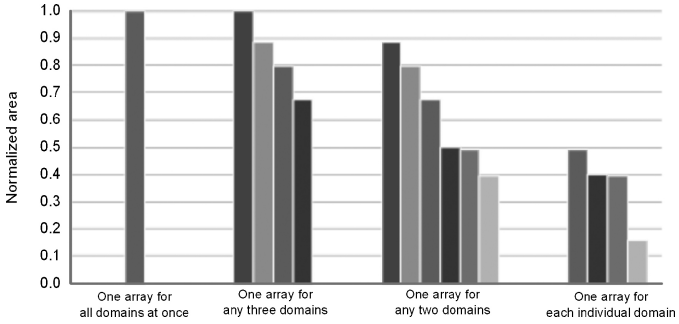

Fig. 10. Area of the array generated for each individual domain and the combinations of any two, three, or four domains, normalized to the area of the array created for all domains at once. Increasing the number of domains per group leads to increased area. But, when generated for individual domains, the array is considerably smaller and thus area efficient, proving that our methodology effectively tailors the array to the domain.

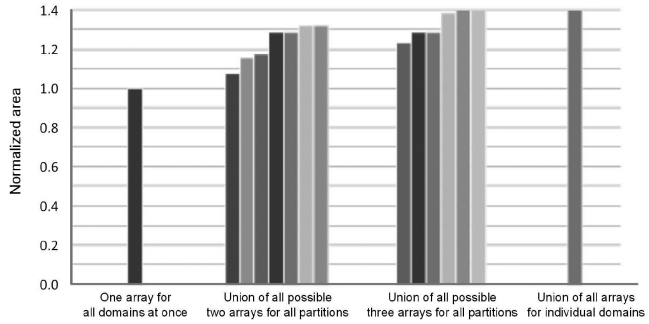

Fig. 11. Effect of dividing the input set of DFGs into two, three, or four sets on the total area needed to accommodate generated arrays. The group G_{4A} storing all DFGs was divided into two, three, or four disjoint groups and the sum of areas of generated arrays was measured. This sum was then normalized to the area of the array created for all four domains at once. The results indicate that the single array generated for all domains at once is the most area efficient solution compared to any multiple-array solution.

shows the array areas normalized to the area of the array generated for all domains at once. Clearly, increasing the number of domains systematically leads to increased area. However, when generated for individual domains, the array is considerably smaller, proving that our methodology is indeed optimized to provide domain-specific arrays.

If input DFGs belong to various domains, one may wonder whether it is more area efficient to generate a single reconfigurable array out of all DFGs, or it is better to divide the input set of DFGs into two or more disjoint sets and generate the same number of arrays. Therefore, we performed the following experiments, relying on the way groupings in Table II were formed. For example, group G_{4A} storing all DFGs can be represented as a union of the following two disjoint sets—$G_{1A} \cup G_{3D}$, or $G_{2A} \cup G_{2F}$, etc. Or it can be represented as a union of the following three disjoint sets—$G_{1B} \cup G_{1D} \cup G_{2B}$, or $G_{1C} \cup G_{1D} \cup G_{2A}$, etc. This way, we divided the input set of four domains into two or three sets, and measured the sum of areas of the obtained arrays. Finally, we normalized the results to the area of the array generated for all domains at once, thus representing a possible solution for a design which needs to cover all domains. The results shown in Fig. 11 indicate that it is more area efficient to create one instead of multiple domain-specific arrays (one per each domain). This evidences the fact that even though the original DFGs belong to four different domains, they actually share many common computation characteristics that can still be exploited by our methodology to build a rather efficient array.

F. Area/Delay Oversize Compared to ASIC

Next, we estimated the reconfigurable array area and delay and compared with a 65 nm standard cell ASIC implementation. For our array, we proceeded as explained in Section VII-C. For all DFGs D_1–D_{19} we assumed, to our disadvantage, that their ASIC implementations require no routing area besides the area required for the operators and pipeline registers, and that all shifts are by a constant value and can be implemented via wiring in the ASIC. Similarly, we assumed that the delay of the ASIC implementation is only the delay through the critical path of the components of the DFG, and ignored routing delays. To estimate the routing delay and the area of the array, we used VPR.

The overall results are shown in Fig. 12. The gray area marked as FPGA represents the area/delay space where results would be expected if DFGs were to be mapped on an FPGA, achieving perfect generality if enough LUTs are present in the architecture. The boundaries of the FPGA area roughly correspond to the data published by Kuon and Rose [5]. Their study provides detailed experimental measurements of the differences between FPGAs and ASICs in terms of logic density, circuits speed, and power consumption for core logic. Kuon and Rose's results [5] show that for circuits containing only look-up table-based logic and flip-flops, the ratio of silicon area required to implement them in FPGAs and ASICs is on average 32× when hard DSP blocks are not used, whereas it decreases to 24× when these blocks are used. These numbers present an optimistic lower bound on the area gap because they assume that all logic array blocks can be fully utilized. Additionally, they report the critical path delay ratio to be on average 3.4× when hard DSP blocks are used, and even a slightly higher ratio, around 3.5×, when these blocks are used in the design. Our results show that the majority of the DFGs result in arrays with an area up to 15× larger than the corresponding ASIC area, and thus significantly more area-efficient than FPGAs with DSP blocks, due to the usage of specialized coarse grain operators. Additionally, the average delay increase compared to ASICs is less than 2×, which is again superior to FPGAs with DSP blocks, due to our efficient

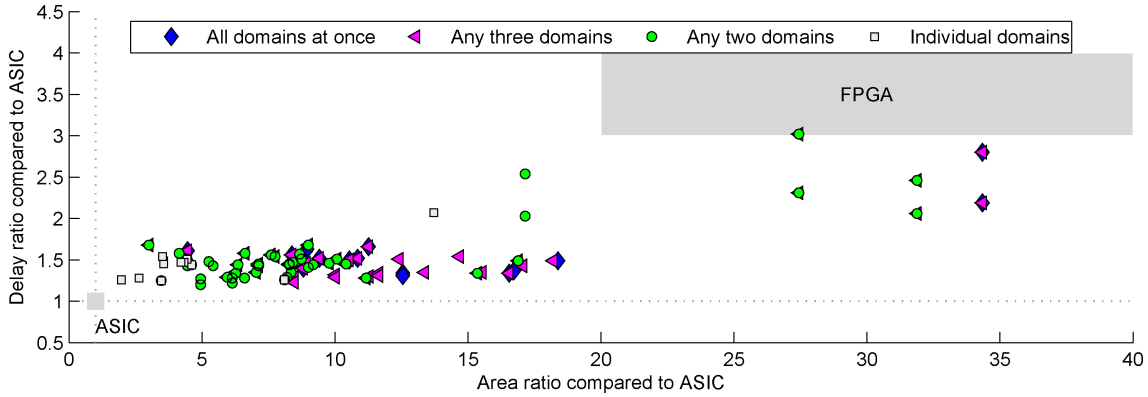

Fig. 12. Area/delay ratio of the arrays generated from all DFGs in the group except the removed DFG, with respect to an ASIC design of the DFG removed from the group. The datapath is usually up to 20× larger and up to 2× slower than the corresponding ASIC design (with some deviations in extreme cases). The results are clustered by the number of domains in the group (Table II). The shaded FPGA zone is as reported in prior studies [5].

word-based communication network. Hence, we succeed in populating the area-delay design space that currently separates ASIC from FPGA implementations while still providing a high generality. However, there are two DFGs that have high area ratio compared to their ASIC implementation—DFGs D_{12} and D_{13}. They do not contain high-area operators, such as multipliers. Consequently, the results in Fig. 12 appear skewed, but these data points can be treated as outliers.

Clearly, our approach of using custom-designed coarse-grained operands as basic building blocks results in both reduced area and improved critical path delay compared to using fine-grained FPGA fabric. Additionally, the amount of configuration storage for both operands and routing network decreases significantly due to our bus-based connectivity, where wires do not need to be configured independently.

G. Area/Delay Oversize Compared to Datapath Merging

To estimate the area and delay oversize of the domain-specific reconfigurable arrays compared with the datapath-merging methodology, we implemented the algorithm introduced by Brisk *et al.* [1]. More recent datapath merging algorithm presented by Zuluaga and Topham [2] is based on the same algorithm, but it introduces new features that are not directly relevant to this paper. Namely, they introduce latency constraints in the merging process to explore the space of possible implementation alternatives instead of trying to find a unique solution, while the datapath merging by Brisk is focusing on maximizing the area savings. Due to high complexity of DFGs under the test, we had to modify the methodology in [1] to improve the runtime.

Datapath merging algorithm [1] assumes as an input a set of directed acyclic graphs (DAGs) $G = \{G_1, G_2, ..., G_n\}$ and has two phases, global and local, that repeat and alternate until all DAGs are not merged, or there are no more candidates for merging. The global phase starts with decomposing each DAG $G_i \in G$ into a set of input-to-output paths P_i, where the set $P = \{P_1, P_2, ..., P_n\}$ stores the sets of paths corresponding to each DAG. Then, it looks for the candidate DAGs G_i and G_j to merge, by finding the pair of paths p_x and p_y, $1 \leq x \leq |P_i|$, $1 \leq y \leq |P_j|$, $1 \leq i, j \leq n$, $i \neq j$, such that they share the maximum area common subsequence MACSeq. Then, it merges G_i and G_j by sharing the nodes in MACSeq. Finally, it replaces G_i and G_j by their merged version G'. The local phase begins with new DAG G', and continues merging nodes inside G', trying to avoid creating cycles in the DAG G'. To accommodate large graphs, enumeration of all paths should be replaced by a pruning heuristic that limits the set P to a reasonable size [20]. Hence, our implementation of the algorithm includes one such heuristic. First, we limit the size of the sets P_i, $1 \leq i \leq n$, to 100 different paths per set, as estimated based on the size of the input DFGs and frequent overlaps among paths in the graphs. Then, to select good candidates for P_i while enumerating the paths, the algorithm checks if a newly found path is a subsequence of the path already present in P_i. If yes, the algorithm ignores it and continues enumerating. This way P_i will contain the paths offering various MACSeq.

Since we already had the areas and delays of all DFG operators, we synthesized, placed, and routed multiplexers of various size using the same 65 nm standard cell library. These multiplexers were inserted in the datapath while merging. Then, we estimated the area of the merged datapath as the sum of areas of all operators and multiplexers, while conservatively still ignoring the area used for routing. To estimate the routing delay and the area of the array, we used the same methodology as in Section VII-F. Ideally, if DFGs within a group are perfectly merged, one would expect that the merged datapath is as large as the DFG reporting the maximum area utilization in Table II, with the addition of the area of the inserted multiplexers, but reduced for the area used for routing in the reconfigurable array. Since the area used for multiplexers is certainly less than the total area used for FPGA-like routing network in our reconfigurable arrays, we expect that for individual domains the area ratio should be less than $100/48 \simeq 2.08$, for any two domains less than $100/33 \simeq 3.03$, and for any three domains and all domains at once less than $100/30 \simeq 3.33$ (Table II). The experimental results presented in Fig. 13 show that the array is up to 3× larger than the merged datapath, while for majority of the groupings this ratio is only up to 2.2. For two groups, G_{2F} and G_{3C}, area of the routing network was considerably higher than the area of the multiplexers in the merged datapath. To understand why, let us look at the size of the array generated for G_{2F}: it equals 13 rows × 24 columns. Group G_{2F} is the

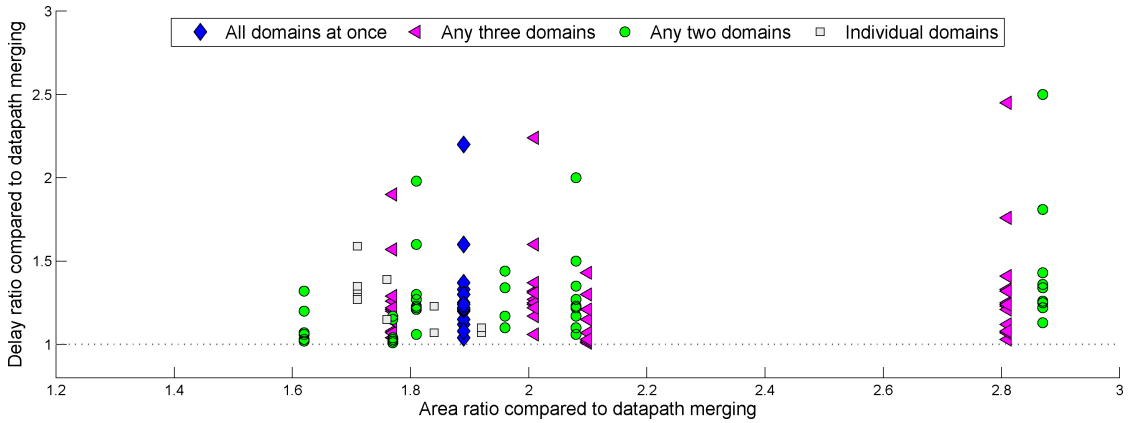

Fig. 13. Area/delay ratio of the arrays generated from all DFGs in the group, with respect to those of the datapath obtained by merging the same DFGs. The array is usually around 2× larger except extreme cases, while the critical path delay of applications mapped onto it is in most cases up to 2× higher than the corresponding delay of the applications when run on merged datapath. The results are clustered by the number of domains in the group (Table II).

union of groups G_{1C} and G_{1D} (Table II), where G_{1C} needs an array of the size 5×24 and G_{1D} of the size 13×12. Obviously, the array created for G_{2F} introduces 50% of unused routing resources, so the area ratio is somewhat higher in this particular case. Fig. 13 also shows that the delay ratio is up to 2.5×, in most of the cases up to 2×. In total, the results indicate that our method succeeds in generating datapaths with a reasonable level of generality at speeds comparable to those of datapaths created by merging the DFGs—designs which have, arguably, practically no generality.

VIII. Related Work

Our algorithm for generating domain-specific reconfigurable arrays is motivated in part by the path-based datapath merging algorithm introduced by Brisk *et al.* [1], which focuses on maximizing the area savings. The work by Zuluaga and Topham [2] differentiate from that of Brisk *et al.* [1] in that the former introduce new features—latency constraints, not directly related to this paper. These datapath-merging algorithms decompose each DFG into paths; paths from distinct DFGs are then merged using subsequence and substring matching techniques. Yet our approach is different; we attempt to generate one single path which is a minimum-cost supersequence of all the paths in all of the input DFGs. Besides, our goals are quite different: traditional datapath merging, although it introduces a form of multiplexer-based configurability in the datapath, aims at reducing the cost. We, on the other hand, aim at a pragmatic increase in flexibility for a moderate cost. Yehia *et al.* [21] presented an approach for graph merging in a wider system context covering both data-flow and control-flow graphs. They introduced additional operations, wires, and multiplexers to increase the similarity of initially dissimilar DFGs. Cong *et al.* [22] considered specific pattern recognition and selection techniques to intelligently select resources to be shared among graphs and produce datapaths with reduced interconnection costs. Like other prior datapath merging techniques, these did not introduce any further generality and the same considerations apply here as with previous work in merging.

Clark *et al.* [23] introduced a system to automatically identify and synthesize custom instruction set extensions, and two generalizations to enable more effective usage of the hardware units. First, they identified *subsumed subgraphs*, which recognized that many operators have an identity element to pass values through unmodified. *Wildcarding* introduced two different operations at the same node in a graph, which increased flexibility in a limited way, and thus *preemptive wildcarding* generalized a graph by more versatile operations—e.g., by replacing an ADD or SUB operation with an ADD/SUB unit. However, we introduce a flexible routing network as a much more general alternative to subsumed subgraphs; also, our approach to merging exploits preemptive wildcarding too.

Ansaloni *et al.* [24] introduced *Expression-Grained Reconfigurable Arrays (EGRAs)* based on combinational processing elements capable of computing entire arithmetic/logic subexpressions using multiple wired ALUs. They built a retargetable mapping toolchain which they used to explore the design space of the elementary cell in an EGRA, thus being able to adapt the architecture to the application domain. Our approach is less general, as our datapath nodes are dedicated operators, rather than more general ALUs; this increases efficiency and moves our solution closer to an ASIC implementation.

Ebeling *et al.* [25] introduced RaPiD, a coarse-grained configurable architecture for executing regular computationally-intensive applications. Similar to our array, RaPiD is composed of word-size computation units, but which are arranged along the 1-D axis. RaPiD limits applications to at most two reads and one write per cycle, while the 2-D nature of our array permits significantly more accesses per cycle, as required by state-of-the-art instruction set extension identification algorithms [4]. Routing in RaPiD is in the form of word-size segmented buses running parallel to the axis, similarly to the routing network between two neighboring rows in our array, but our network provides significantly more routing opportunities due to the introduction of horizontal and vertical routing channels. The RaPiD architecture was manually designed and tuned for a wide variety of circuits within the DSP domain, as well as the other relevant architectures such as PipeRench [26], Pleiades [27], or MorphoSys [28]. Instead, our method enables automated generation of configurable arrays based on the application domains selected by the designer.

Compton and Hauck [29] introduced the Totem tool to generate 1-D architectures similar in style to RaPiD. To

achieve a more customized design, Totem varies the number and order of word-size computation units in a RaPiD-like array, and the length and the number of tracks in the routing channel. To select architectural components, the Totem takes the minimum number of each type of computation unit needed to implement all of the given circuits (one at a time), whereas our tool automatically infers an overhead in the number of components to accommodate larger datapaths not known at design time. Additionally, they constrain computation-unit types to be evenly distributed through the 1-D array, while our tool decides on a good distribution of the units using the path fusion procedure. Finally, Totem reports significantly higher number of word-size tracks in the routing channel (up to 34), while our array uses less tracks per channel, but provides higher routing opportunities thanks to the regular 2-D routing network. To estimate generality, they create the array from multiple synthetic circuits generated by profiling input netlists and add as many computation units as needed to accommodate these input netlists; then, they measure only the routability of the applications when mapped on the Totem array. Instead, we use real domain-specific application DFGs and test the true overall flexibility of our domain-specific arrays.

One important area that this paper does not attempt to address is the interface by which the processor supplies data to the datapath. This issue has been effectively addressed by others [30], [31], and any existing technique could be used, depending on the usage context of our datapath. It is an issue orthogonal to the organization of the array itself, common to practically all acceleration solutions. It is a still partially open problem but one which is complementary to our work.

IX. Conclusion

This paper described a methodology to automatically generate domain-specific coarse-grained reconfigurable arrays from a partially complete set of representative applications. The method automatically introduces generality into the array, which increases the likelihood that additional applications belonging to the same domain, or to a related domain, will map successfully onto it. In particular, this approach supports late design changes to the application software or to a mix of applications that are not fully known at design time. The experiments showed that the approach offers far greater flexibility than traditional ASIC merging, while providing far superior logic density in comparison to an FPGA. An additional advantage is the reduced design effort and time-to-market for future custom hardware accelerators. We strongly believe that current technology trends favor increased heterogeneity and acceleration technologies, which may not have been feasible, either economically or technologically, in prior eras. Thus, we conclude that our approach explores a new direction of great importance that will improve the process by which hardware acceleration technologies are conceived.

Acknowledgments

The authors thank Dr. G. Dimić for his support throughout the project and his comments on earlier versions of the manuscript.

References

[1] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proc. 41st Design Autom. Conf.*, Jun. 2004, pp. 395–400.

[2] M. Zuluaga and N. Topham, "Design-space exploration of resource sharing solutions for custom instruction set extensions," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. CAD-28, no. 12, pp. 1788–1801, Dec. 2009.

[3] M. J. S. Smith, *Application-Specific Integrated Circuits*. Boston, MA: Addison-Wesley, 1997.

[4] P. Ienne and R. Leupers, Eds., *Customizable Embedded Processors—Design Technologies and Applications* (Ser. Systems on Silicon Series). San Mateo, CA: Morgan Kaufmann, 2006.

[5] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," *IEEE Trans. Computer-Aided Design Integr. Circuits Syst.*, vol. CAD-26, no. 2, pp. 203–15, Feb. 2007.

[6] M. Stojilović, D. Novo Bruna, L. Saranovac, P. Brisk, and P. Ienne, "Selective flexibility: Breaking the rigidity of datapath merging," in *Proc. Design, Autom. Test Europe Conf. Exhibition*, Mar. 2012, pp. 1543–48.

[7] A. Ye and J. Rose, "Using bus-based connections to improve field programmable gate-array density for implementing datapath circuits," *IEEE Trans. VLSI Syst.*, vol. 14, no. 5, pp. 462–73, May 2006.

[8] J. Branke, M. Middendorf, and F. Schneider, "Improved heuristics and a genetic algorithm for finding short supersequences," *OR Spectrum*, vol. 20, no. 1, pp. 39–45, Feb. 1998.

[9] T. Jiang and M. Li, "On the approximation of shortest common supersequences and longest common subsequences," *SIAM J. Computing*, vol. 24, no. 5, pp. 1122–39, Oct. 1995.

[10] K.-M. Chao and L. Zhang, *Sequence Comparison: Theory and Methods* (Series Computational Biology). London: Springer, 2009.

[11] V. Betz and J. Rose, "Automatic generation of FPGA routing architectures from high-level descriptions," in *Proc. 8th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Feb. 2000, pp. 175–84.

[12] 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 Design Autom. Conf.*, Jan. 2008, pp. 776–82.

[13] E. R. Gansner, E. Koutsofios, S. C. North, and K. phong Vo, "A technique for drawing directed graphs," *IEEE Trans. Softw. Eng.*, vol. 19, no. 3, pp. 214–30, Mar. 1993.

[14] *TMS320C64x DSP Library Programmer's Reference*, lit. no. SPRU565B, Texas Instruments, Dallas, Oct. 2003.

[15] *TMS320C64x Image/Video Processing Library Programmer's Reference*, lit. no. SPRU023B, Texas Instruments, Dallas, Oct. 2003.

[16] *TMS320C67x DSP Library Programmer's Reference*, lit. no. SPRU657C, Texas Instruments, Dallas, Jan. 2010.

[17] EEMBC Consortium, *DENBench Version 1.0, Benchmark Name: MPEG-2 Decode*, Feb. 2006 [Online]. Available: http://www.eembc.org/

[18] *ExpressDFG—Instruction Scheduling Benchmarks*, University of California, Santa Barbara, CA http://express.ece.ucsb.edu/benchmark/

[19] W. Feng and S. Kaptanoglu, "Designing efficient input interconnect blocks for LUT clusters using counting and entropy," *ACM TRETS*, vol. 1, no. 1, pp. 6:1–6:28, Mar. 2008.

[20] P. Brisk and M. Sarrafzadeh, "Datapath synthesis," in *Customizable Embedded Processors—Design Technologies and Applications* (Series Systems on Silicon Series), P. Ienne and R. Leupers, Eds. San Mateo, CA: Morgan Kaufmann, 2006, ch. 10, pp. 233–55.

[21] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *Proc. 15th Int. Symp. High-Performance Comput. Arch.*, Feb. 2009, pp. 277–88.

[22] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *Proc. 16th ACM/SIGDA Int. Symp. Field Programmable Gate Arrays*, Feb. 2008, pp. 107–16.

[23] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customisation," in *Proc. 36th Annu. Int. Symp. Microarchitecture*, Dec. 2003, pp. 129–40.

[24] G. Ansaloni, P. Bonzini, and L. Pozzi, "Design and architectural exploration of expression-grained reconfigurable arrays," in *Proc. 6th IEEE Symp. Appl. Specific Processors*, Jun. 2008, pp. 26–33.

[25] C. Ebeling, D. C. Cronquist, P. Franklin, J. Secosky, and S. G. Berg, "Mapping applications to the RaPiD configurable architecture," in *Proc. 5th IEEE Symp. Field-Programmable Custom Computing Mach.*, Apr. 1997, pp. 106–15.

[26] S. C. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer, "PipeRench: A co-processor for streaming multimedia acceleration," in *Proc. 26th Annu. Int. Symp. Comput. Arch.*, May 1999, pp. 28–39.

[27] A. Abnous and J. Rabaey, "Ultra-low-power domain-specific multimedia processors," in *Proc. 9th Workshop VLSI Signal Process.*, Oct. 1996, pp. 461–70.

[28] 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. VLSI Signal Process. Syst.*, vol. 24, no. 2–3, pp. 147–64, Mar. 2000.

[29] K. Compton and S. Hauck, "Automatic design of reconfigurable domainspecific flexible cores," *IEEE Trans. VLSI Syst.*, vol. VLSI-16, no. 5, pp. 493–503, May 2008.

[30] S. Girbal, O. Temam, S. Yehia, H. Berry, and Z. Li, "A memory interface for multi-purpose multi-stream accelerators," in *Proc. Int. Conf. Compilers, Architectures, Synthesis Embedded Syst.*, Oct. 2010, pp. 107–16.

[31] T. Kluter, S. Burri, P. Brisk, E. Charbon, and P. Ienne, "Virtual Ways: Efficient coherence for architecturally visible storage in automatic instruction set extensions," in *High Performance Embedded Architectures and Compilers* (Ser. Lecture Notes in Computer Science), X. Martorell, Y. N. Patt, P. Foglia, E. Duesterwald, and P. Faraboschi, Eds. Heidelberg, Germany: Springer, 2010, vol. 5952, pp. 126–40.

Mirjana Stojilović (M'09) received the Dipl. Ing. degree in electrical engineering from the School of Electrical Engineering, the University of Belgrade, Belgrade, Serbia, in 2006. Since then she has been pursuing the Ph.D. degree in electronics from the same university.

She is currently an Embedded Systems Developer with the Institute Mihailo Pupin, Belgrade, Serbia. Since 2010, she has been cooperating with the Processor Architecture Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, visiting periodically as a Guest Researcher. Her current research interests include electronic design automation, reconfigurable computing, and application-specific processors.

Ms. Stojilović serves as a Reviewer of *ACM Transactions on Design Automation of Electronic Systems* journal. In 2010, she was awarded the first prize of the Western Balkan Countries ICT Idea Competition. In 2012, she was a recipient of the Young Author Best Paper Award at the 20th Telecommunication Forum in Belgrade.

David Novo (M'08) received the M.S. degree from the Universitat Autonoma de Barcelona, Barcelona, Spain, in 2005, and the Ph.D. degree in engineering from the Katholieke Universiteit Leuven, Leuven, Belgium, in 2010.

Since November 2010, he has been a Post-Doctoral Scholar with the Processor Architecture Laboratory, Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland. His current research interests include hardware and software techniques for increasing computation efficiency in next-generation computers.

Dr. Novo was a recipient of the Best Paper Award at the 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays in 2012 and nominated at the IEEE Workshop on Signal Processing Systems in 2005. In 2012, he was awarded with the EU Marie Curie Intra-European Fellowship for Career Development. He has been a Guest Editor of a Special Issue on Quantization of VLSI Digital Signal Processing Systems, which appeared in February 2012 on the *EURASIP Journal on Advances in Signal Processing*.

Lazar Saranovac (M'91) was born in Sremska Mitrovica, Serbia in 1961. He received the B.S. degree in electrical engineering, the M.S. degree in electronics, and the Ph.D. degree all from the University of Belgrade, Belgrade, Serbia in 1987, 1993, and 2001, respectively.

He is currently an Assistant Professor of electrical engineering with the University of Belgrade. His current research interests include embedded systems, methods for measuring electric power at power frequencies, digital signal processing, and design of digital systems.

Philip Brisk (M'09) received the B.S., M.S., and Ph.D. degrees, all in computer science, from the University of California, Los Angeles, Los Angeles, CA, in 2002, 2003, and 2006 respectively.

From 2006 to 2009, he was a Post-Doctoral Scholar with the Processor Architecture Laboratory in the School of Computer and Communication Sciences, Ecole Polytechnique Fédérale de Lausanne, Lausanne, Switzerland. He is currently an Assistant Professor with the Department of Computer Science and Engineering, the Bourns College of Engineering, the University of California, Riverside. His current research interests include FPGAs, compilers, and design automation and architecture for application-specific processors.

Dr. Brisk was a recipient of the Best Paper Award at the International Conference on Compilers, Architecture, and Synthesis in 2007, and the International Conference on Field Programmable Logic and Applications in 2009. He has been a member of the program committees of several international conferences and workshops, including Design Automation and Test in Europe, the IEEE Symposium on Application-Specific Processors, the International Workshop on Software and Compilers for Embedded Systems, and the Reconfigurable Architecture Workshop. He has been the General Co-Chair of the 4th IEEE Symposium on Industrial Embedded Systems in 2009, and of the 8th IEEE Symposium on Application Specific Processors in 2010.

Paolo Ienne (S'94–M'96) received the Dottore degree in electronic engineering from the Politecnico di Milano, Milan, Italy, in 1991, and the Ph.D. degree in computer science from the Ecole Polytechnique Fédérale de Lausanne (EPFL), Lausanne, Switzerland, in 1996.

Since 2000 he has been a Professor at the EPFL and heads the Processor Architecture Laboratory. Prior to that, from 1990 to 1991, he was an Undergraduate Researcher with Brunel University, Uxbridge, U.K. From 1992 to 1996, he was a Research Assistant with the Microcomputing Laboratory and at the MANTRA Center for Neuro-Mimetic Systems of the EPFL. In December 1996, he joined the Semiconductors Group of Siemens AG, Munich, Germany (which later became Infineon Technologies AG). After working on datapath generation tools, he became the Head of the embedded memory unit in the Design Libraries division. His research interests include various aspects of computer and processor architecture, electronic design automation, computer arithmetic, FPGAs and reconfigurable computing, and multiprocessor systems-on-chip.

Dr. Ienne was a recipient of the Best Paper Award at the 40th Design Automation Conference (DAC) in 2003, at the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems in 2007, at the 19th International Conference on Field-Programmable Logic and Applications in 2009, and at the 20th ACM/SIGDA International Symposium on Field-Programmable Gate Arrays in 2012. In 2008, he was a General Co-Chair of the 6th IEEE Symposium on Application Specific Processors and a Guest Editor of a Special Section on Application Specific Processors for the IEEE TRANSACTIONS ON VERY LARGE SCALE INTEGRATION SYSTEMS (appeared in October 2008). In 2010, he was the Program Subcommittee Chair of the DAC on High-Level and Logic Synthesis. From 2010 to 2012, he was a Topic Co-Chair of Design Automation and Test in Europe (DATE) for Architectural and High-Level Synthesis topic. In 2011, he was a Program Co-Chair of the 20th IEEE Symposium on Computer Arithmetic (ARITH), Guest Editor of a Special Section on Computer Arithmetic for the IEEE TRANSACTIONS ON COMPUTERS (appeared in August 2012), and a Program Co-Chair of the 22nd IEEE International Conference on Application-Specific Systems, Architectures and Processors. He has been a member of some 50 program committees of international workshops and conferences in the areas of design automation, computer architecture, embedded systems, compilers, FPGAs, and asynchronous design. Since 2011, he has been an Associate Editor of the *ACM Transactions on Design Automation of Electronic Systems*. He is a member of the IEEE Computer Society.