

# Selective Flexibility: Breaking the Rigidity of Datapath Merging

Mirjana Stojilović\*, David Novo<sup>†</sup>, Lazar Saranovac<sup>†</sup>, Philip Brisk<sup>§</sup> and Paolo Ienne<sup>‡</sup>

\*University of Belgrade  
Institute Mihailo Pupin  
Volgina 15, 11060 Belgrade, Serbia  
Email: mirjana.stojilovic@pupin.rs

<sup>†</sup>University of Belgrade  
School of Electrical Engineering  
Bulevar Kralja Aleksandra 73, 11120 Belgrade, Serbia  
Email: laza@el.etf.rs

<sup>§</sup>University of California Riverside  
Department of Computer Science and Engineering  
Riverside, CA 92521, U.S.A.  
Email: philip@cs.ucr.edu

<sup>‡</sup>École Polytechnique Fédérale de Lausanne (EPFL)  
School of Computer and Communication Sciences  
CH-1015 Lausanne, Switzerland  
Email: david.novobruna@epfl.ch and paolo.ienne@epfl.ch

**Abstract**—Hardware specialization is often the key to efficiency for programmable embedded systems, but comes at the expense of flexibility. This paper combines flexibility and efficiency in the design and synthesis of domain-specific datapaths. We merge all individual paths from the *Data Flow Graphs (DFGs)* of the target applications, leading to a minimal set of required resources; this set is organized into a column of physical operators and cloned, thus generating a domain-specific rectangular lattice. A bus-based FPGA-style interconnection network is then generated and dimensioned to meet the needs of the applications. Our results demonstrate that the lattice has good flexibility: DFGs that were not used as part of the datapath creation phase can be mapped onto it with high probability. Compared to an ASIC design of a single DFG, the speed of our domain-specific coarse-grained reconfigurable datapath is degraded by a factor up to  $2\times$ , compared to  $3\text{--}4\times$  for an FPGA; similarly, our lattice is up to  $10\times$  larger than an ASIC, compared to  $20\text{--}40\times$  for an FPGA. We estimate that our array is up to  $6\times$  larger than an ASIC accelerator, which is synthesized using datapath merging and has limited or null generality.

## I. INTRODUCTION

Embedded system efficiency is often advanced through specialization, i.e., through the design of application or domain-specific custom computing hardware [10], [13]. Large dedicated datapaths, for example, are common in dedicated signal processing systems; however, their area overhead is significant, and they are inflexible. Area can be reduced by merging several datapaths whose usage at runtime is mutually exclusive.

The problem of achieving some form of flexibility in hardware is more difficult to pin down and evaluate. Ideally, one would desire a fabric that is similar to the target datapath, so as to display only a moderate loss in efficiency compared to an ASIC implementation, but while maintaining sufficient flexibility to accommodate late *design changes* or new applications *in the same domain*. Neither requirement is well-formulated, and yet many system designers and project managers in the semiconductor industry would clearly desire such solutions. *Field Programmable Gate Arrays (FPGAs)* are a solution at one end of the spectrum, where near-ideal flexibility comes at the cost of incredibly poor logic density,

especially for arithmetically-intensive applications. Despite many efforts, no commercially successful product has, to date, embedded FPGAs into ASIC design flows. Datapath merging is at the opposite end of the spectrum, as it achieves high efficiency but offers no flexibility beyond the ability to map the DFGs of the applications that were initially merged [3].

This paper introduces flexibility into the datapath merging process; it presents an algorithm that infers automatically the desired features of a domain-specific reconfigurable datapath from a collection of representative DFGs. Other DFGs that were not used in the datapath generation process are then mapped onto the resulting datapath, thereby establishing its flexibility. 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 are mapped onto it; however, if the DFGs all come from the same general application domain (e.g., filters used in signal processing), then the likelihood of successful mapping is quite high. Our experiments demonstrate that the vast majority of our DFGs can be mapped successfully onto the datapaths that are generated by our algorithm.

## II. FLEXIBILITY, OR LACK THEREOF...

Figures 1a and 1b show two DFGs, which approximately correspond to a *Sum of Absolute Differences (SAD)*, widely used for motion estimation in video compression, and a radix-2 butterfly, which is a building block of the *Fast Fourier Transform (FFT)*. Figure 1c shows the result of merging the two datapaths, with multiplexers inserted; this datapath can be configured to execute either DFG [3], [21].

Now, suppose that we want to execute an image convolution DFG, shown in Figure 1d, which arguably belongs to the same application domain. Although the merged datapath in Figure 1c 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 strives to insert additional flexibility during the process by which the first two DFGs are merged. The cost

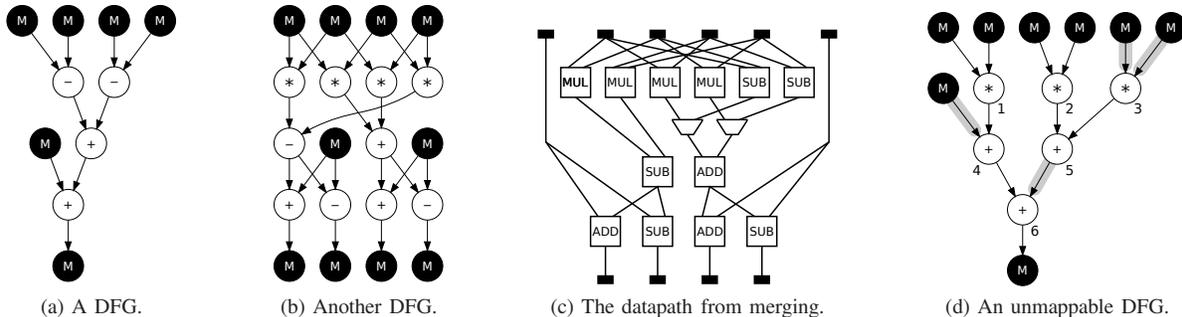


Fig. 1. The limitations of classic datapath merging [3], [21]. Two sample data-flow graphs from (a) a two pixels SAD and (b) radix-2 FFT butterfly. The M nodes represent accesses to some form of memory (register files, stream buffer, etc.) and are not part of the datapaths. A typical result of merging these two graphs would result in (c). Unfortunately, the data-flow graph (d), a  $3 \times 3$  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).

of doing so is the introduction of additional operators and interconnect resources, which increase the area and delay of the resulting datapath; however, doing so also increases the likelihood that additional DFGs 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: 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. It attempts to distill the essential computational structures, and then attempts 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.

Figure 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 *superpath*. The *superpath* is an ordered sequence of operators, which includes all the sequences of operations present in the input applications. Such sequences are assumed to be inherent to the target domain and, once a *superpath* is created, we consider it very likely that all the paths of a new application belonging to the same domain will be already included.

Secondly, we fix the total number of operators implemented in our datapath. The *superpath* is the basic construction block: a column of our rectangular datapath composed by  $N \times M$  operators, where  $N$  is the length of the *superpath* and  $M$  is the number of times that it needs to be replicated for a successful mapping of the applications at hand. The

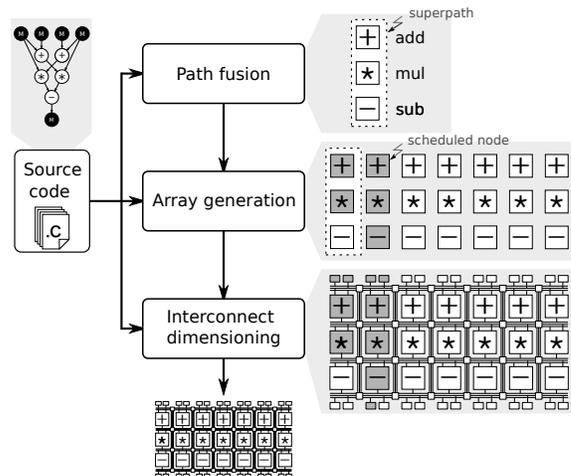


Fig. 2. Proposed flow to synthesize domain-specific datapaths.

interconnect is inspired by FPGAs, but uses bus-based rather than bit-based connections in order to reduce the amount of configuration storage. The reconfiguration of the datapath is achieved by shifting in configuration bits and storing them in the configuration memory cells, and has not been addressed in detail in this paper.

By defining such a datapath, we expect to provide the computational structures that enable a high degree of generality for a particular domain at a reasonably small cost. In the rest of this section, we describe in detail the different parts of our technique to automatically generate the datapath from a collection of DFGs.

#### A. Path Fusion

The basic building block of our reconfigurable datapath is created through *path fusion*. This block defines the type of operations supported by our datapath and how these are sequenced from inputs to outputs. We reuse the *maximum area common subsequence* (MACSeq) metric of other existing graph-merging algorithms [3] to select the building blocks of minimal area; this gives priority in the merging to the paths that have the MACSeq of operators. The input of our path-fusion algorithm consists of all the DFGs of the

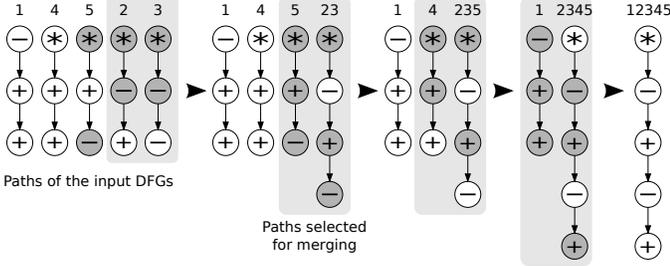


Fig. 3. Steps of our path fusion algorithm. The paths highlighted in gray are selected for merging. The nodes in darker gray form MACseq.

different applications provided by the designer; the process is as follows:

- 1) Enumerate all paths of each DFG.
- 2) Group the paths into multiple sets depending on their length (number of nodes).
- 3) Starting from the set having the longest paths, perform pairwise search for the MACSeq between paths.
- 4) Replace the two paths that maximize the metric by their merged version.
- 5) Repeat 3) and 4) until only one path is left in the set.
- 6) Move the resulting path to the next set containing the paths of smaller length and repeat 3) to 5) until only one path is left: the *superpath*.

In practice, this algorithm converges fast into the *superpath*, as the shorter paths will most likely be already contained in previously merged longer paths.

In Figure 3 we describe the path fusion process implemented on the DFGs illustrated in Figures 1a and 1b as an example: The first DFG has only one distinct path,  $P_1 = \{S, A, A\}$ . The second DFG has 4 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. Our algorithm groups all the paths in the same set as they all have the same length. To find the MACSeq, assume that the areas of the different operators are related as  $M > S > A$ . Accordingly, the MACSeq corresponds to  $\{M, S\}$ , which is contained in  $P_2$ ,  $P_3$ , and  $P_5$ .  $P_2$  is thus merged with  $P_3$ , resulting in  $P_{2,3} = \{M, S, S, A\}$ , which is then merged with  $P_5$ , resulting in  $P_{2,3,5} = \{M, S, A, S, A\}$ . The next path selected for merging is  $P_4$ , which is already included in  $P_{2,3,5} = P_{2,3,5,4}$ . Finally, the only remaining path,  $P_1$ , is selected for merging. Again, the path is already included, and  $P_{2,3,5,4} = P_{1,2,3,5,4}$ , which is the actual *superpath*. Although here we have differentiated between adders and subtractors for illustrating our path fusion algorithm, in actuality our tool assigns them to the same operator, an adder-subtractor, thus increasing the datapath generality. Accordingly, the *superpath* generated by our tool for the example is  $P_{1,2,3,4,5} = \{M, A, A, A\}$ , where the  $A$  now stands for addition-subtraction.

### B. Array Generation

Once the *superpath* is generated, it is replicated  $M$  times to generate the array shaping our datapath. A minimum value of

$M$  is computed so that the datapath includes enough operators to execute all the operations of every input application. To provide generality beyond the size of the input set of applications, our tool can apply an oversizing factor to enlarge the minimum value of  $M$ , which so far is provided by the designer (in our experiments zero or one column is added).

### C. Interconnect Dimensioning

Once the datapath is generated, we need to add routing resources. We use standard FPGA-like interconnections [2] and dimension the number of word-size tracks in each channel between operators. For that, we place and route every DFG, to measure the minimal number of routing tracks needed. Of course, good placement is key since the minimum channel width to guarantee routability depends on it.

To emphasize the regularity of the graphs, we desire a *placement* that mimics effective graph drawing algorithms. These algorithms usually keep graph edges as short as possible, to minimize the number of edge crossings, and emphasize symmetries [20]. We use the algorithm embedded in *dot* [8], which is an open source tool for laying out *hierarchical* drawings of directed graphs. To *dot* we give as inputs the DFG nodes and the sequence of nodes in the *superpath* (the column of the datapath). The DFG nodes belonging to the same row are then assigned the rank of the corresponding node in the *superpath*. Hence, *dot* is forced to place operators only within the rows, without the possibility to move any operator from one row to another (see Figure 4b). The horizontal node coordinates are then scaled and rounded to represent columns. When assigning a node to a row for placement, the tool follows these criteria: (1) If the node is not a part of a binary tree (see Figure 5), it is placed in the first row with the correct operators below the rows assigned to its predecessors. (2) If the node is a part of a binary tree, which is often the case in DSP applications, we first minimize the tree height and then try to place the node as early as possible, thus increasing the overall row utilization. Any rows that are never used in the array as a result of the latter procedure can be removed automatically to conserve area.

To find the minimum channel width necessary to successfully route a placed DFG, we use VPR, a classic open-source FPGA architectural simulator and tool [2] for placement-and-routing. To adapt VPR for our purposes we first represent all DFG nodes as 2-input 1-output configurable logic blocks composed of a single subblock. 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 [18]. Then, for all DFGs, we run VPR with the appropriate placement, netlist, and architecture descriptions. VPR reports the minimum channel width for which a legal routing solution can be found for each benchmark. 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.

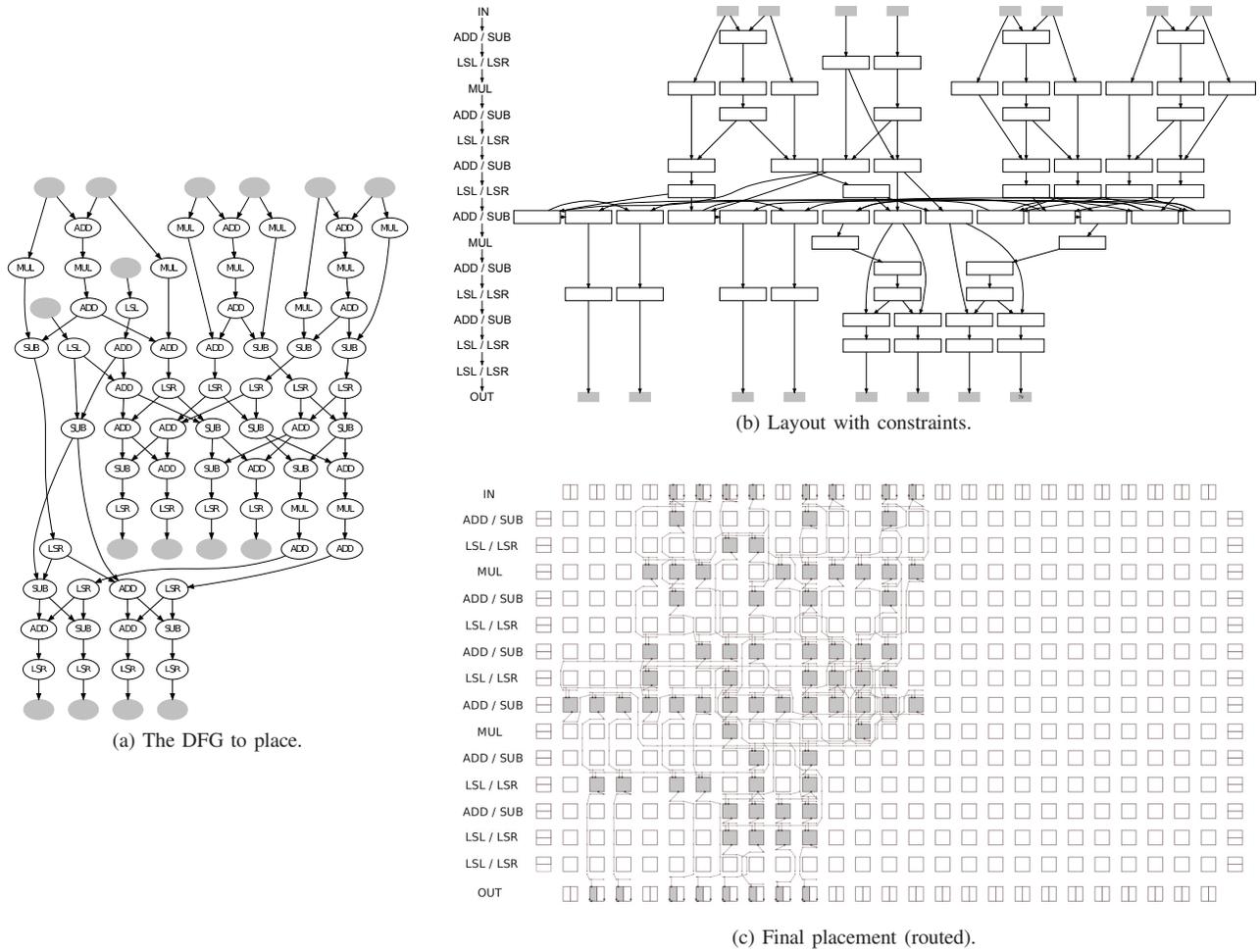


Fig. 4. The placement process. The input DFG shown in (a) is laid out (b) with appropriate constraints and parameters to suggest a detailed placement on the array. After snapping the horizontal placement to the columns, the final placement (c) is achieved (shown after routing by VPR in the figure).

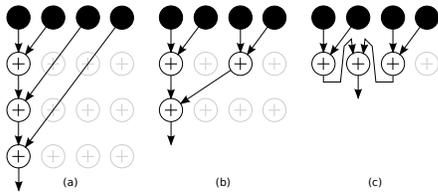


Fig. 5. Optimization of row utilization for binary trees of identical operators. (a) Part of a typical DFG with accumulation of multiple partial results. (b) The chain of operators transformed into a binary tree. (c) Tree nodes assigned to one of the previous rows, to improve row utilization.

#### IV. EXPERIMENTAL RESULTS

In this section we assess the generality achieved by our method, as well as the datapath area and delay with respect to ASIC and FPGA. Additionally, we provide some insights into the drawbacks of the current implementation.

We selected 19 different DFGs from applications available in benchmarks and commercial libraries [7], [14]–[17] covering various classic signal and image processing computations (FFT, DCT, IDCT, FIR, IIR, autocorrelation, etc.). They constitute Group 1 in our experiments. Groups 2A and 2B are

TABLE I  
GENERALITY METRIC FOR VARIOUS GROUPS OF BENCHMARKS.

Group	Generality [%]	Group	Generality [%]
1	87.50	4A	83.33
2A	75.00	4B	50.00
2B	72.70	4C	75.00
3A	90.00	4D	60.00
3B	87.50		

subsets of Group 1, which try to group similar applications. Similarly, Groups 3 $x$  and Groups 4 $x$  regroup all of the DFGs in different and increasingly smaller clusters.

For each of these groups, we measure the *generality* of the resulting 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 III-C) with the exception that the channel width is now known and fixed. The *generality* for group  $G$  is the ratio of the number of successfully mapped excluded DFGs in the  $N$  experiments to the total number of DFGs  $N$ . The results are presented in Table I: Generality is in most of the cases at least 75%.

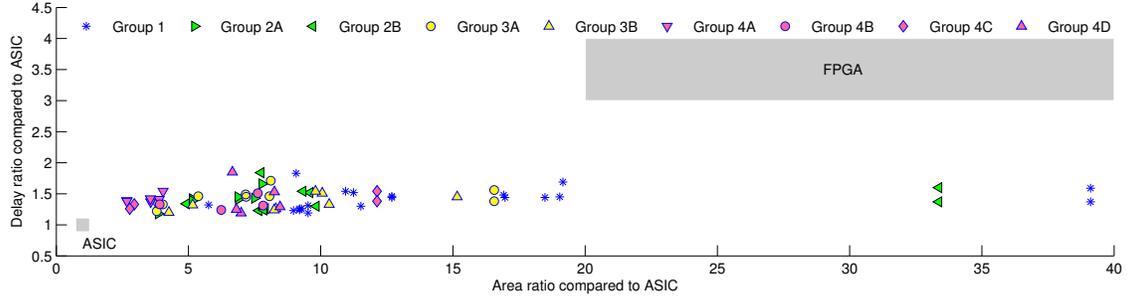


Fig. 6. Area/Delay ratio of the datapaths generated automatically using our tool with respect to an ASIC design of the DFG removed from the group. The datapath is usually up to  $10\times$  larger (with significant deviations in extreme cases) and up to  $2\times$  slower than the corresponding ASIC design. The shaded FPGA zone is as reported in prior studies [12]. Assuming 100% resource utilization, our datapath is conservatively  $2\text{--}6\times$  larger than an ASIC obtained by datapath merging, while ignoring the area overhead of the multiplexers. The results comparing our datapath to datapath merging are not shown on the graph.

For groups having small  $N$ , generality naturally decreases, but remains at least 50%: if an unrealistically small *learning set* is given, it is understandably difficult for the technique to generalize the needs of unknown DFGs. Additionally, since we fix the array width 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.

Next, we estimated the datapath area and delay and compared with a 65nm standard cell ASIC. We synthesized, placed, and routed individually all the logic and arithmetic operations found in the DFGs using the gate implementations of a commercial library. We assumed, to our disadvantage, that the ASIC implementation of a DFG requires no routing area besides the area required for the operators and that all shifts are by a constant value and can be implemented via wiring in the ASIC. Similarly, the delay of the ASIC implementation was assumed as the pure delay through the critical path of the components of the DFG, ignoring routing delays. We used VPR to estimate the routing delay and datapath area. We used an appropriate technology configuration file along with the real number of wires required, as VPR does not natively support bus-based connections. This conservatively overestimates the routing area because VPR controls each wire independently, even though our implementation intends to use a bus-based interconnect. The overall results are shown in Figure 6. The grey area marked as FPGA represents the area/delay space where results would be expected if DFGs were to be mapped on an FPGA [12]—of course, FPGAs achieve perfect generality if the array is large enough. Our results show that the majority of all DFGs result in arrays with an area up to  $10\times$  larger than the corresponding ASIC area, and the delay increase by as much as to  $2\times$ . This indicates that our method succeeds in generating datapaths with a reasonable level of generality at speeds similar to those of a pure ASIC implementation and while not paying the full area/delay cost of FPGAs. However, there are several DFGs that have high area ratio compared to an ASIC implementation. These DFGs do not contain high-area operators, such as multipliers. In Figure 6, which reports the ratio of datapath to ASIC area, the results are skewed because: (1) the denominator (ASIC area) is a very small number; and (2) the numerator (datapath area) is

dominated by the interconnect. Consequently, these data points should be treated as outliers. We also compared the area of our datapath to the area of an ASIC obtained by datapath merging. Assuming 100% resource utilization and ignoring multiplexer overhead, our datapath was conservatively estimated to be  $2\text{--}6\times$  larger.

## V. STATE OF THE ART

Our datapath generation algorithm is motivated in part by the path-based datapath merging algorithm, introduced by Brisk *et al.* [3] and later refined by Zuluaga and Topham [21]. These algorithms decompose each DFG into paths; paths from distinct DFGs are then merged using subsequence and substring matching techniques. Our approach is slightly different: we attempt to generate one single path which is a minimum-cost supersequence of all the paths in all of the input DFGs; we then replicate this path and introduce a flexible interconnect to form our flexible domain-specific datapath.

Yehia *et al.* [19] 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.* [6] 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.

Clark *et al.* [4] introduced a system that automatically identified and synthesized custom instruction set extensions. They introduced two generalizations to enable more effective usage of the hardware units. Firstly, 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. *Preemptive wildcarding* generalized a graph by more versatile operations, e.g., by replacing an ADD or SUB operation with an ADD/SUB unit. We introduce a flexible routing network as an alternative to subsumed subgraphs; our approach to merging exploits preemptive wildcarding as well. An important difference is that we introduce generality into the hardware

datapath itself, while Clark *et al.* introduced generality into the DFG. We believe that this gives our approach more general possibilities for mapping new DFGs onto our datapath.

Ansaloni *et al.* [1] 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.

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 solved by others [9], [11], and any existing technique could be used, depending on the usage context of our datapath.

## VI. CONCLUSIONS

We have presented a novel way to merge several DFGs to create a reconfigurable datapath. Our method is fundamentally different from others in the literature because we try to pay a *reasonable* price to significantly increase the chances that other applications, not considered at design time, can also be accelerated. Our method heuristically distills the essential computational structures of a set of representative DFGs provided by the designers into the datapath, and then provides a standard interconnection network to ensure flexible mappings. Our datapath offers greater flexibility than an ASIC; it is less flexible than an FPGA, but comes at a reduced area overhead, and, for the DFGs that are successfully mapped, a reduced speed penalty as well. We believe that our approach explores a new direction of great importance in a world where heterogeneous spatial systems are likely to emerge as a dominant form of computation, especially for code acceleration in domain-specific embedded systems.

This work can be improved in many future directions, such as specializing the bitwidth of the operators, and composing multiple limited-precision operators to form higher-precision operators. Another possibility is to introduce flexible arithmetic components, e.g., multipliers that can be configured to perform addition/subtraction as well. Lastly, we have observed utilization ratios of the array which are relatively low, hardly above 40%; this could be improved by customizing the rectangular shape of the array to the DFGs in the domain, as some classes of DFGs, especially instruction set extensions, often have the general shape of inverted cones [5].

## ACKNOWLEDGMENTS

This work was partially supported by the research grant TR32043 of the Ministry of Education and Science of the Republic of Serbia and by the SCOPES grant IZ74Z0\_128293/1 of the Swiss National Science Foundation.

## REFERENCES

- [1] G. Ansaloni, P. Bonzini, and L. Pozzi, "Design and architectural exploration of expression-grained reconfigurable arrays," in *Proceedings of the 6th IEEE Symposium on Application Specific Processors*, Anaheim, Calif., Jun. 2008, pp. 26–33.
- [2] V. Betz and J. Rose, "Automatic generation of FPGA routing architectures from high-level descriptions," in *Proceedings of the 8th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2000, pp. 175–84.
- [3] P. Brisk, A. Kaplan, and M. Sarrafzadeh, "Area-efficient instruction set synthesis for reconfigurable system-on-chip designs," in *Proceedings of the 41st Design Automation Conference*, San Diego, Calif., Jun. 2004, pp. 395–400.
- [4] N. Clark, H. Zhong, and S. Mahlke, "Processor acceleration through automated instruction set customisation," in *Proceedings of the 36th Annual International Symposium on Microarchitecture*, San Diego, Calif., Dec. 2003, pp. 129–40.
- [5] J. Cong, Y. Fan, G. Han, and Z. Zhang, "Application-specific instruction generation for configurable processor architectures," in *Proceedings of the 12th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2004, pp. 183–89.
- [6] J. Cong and W. Jiang, "Pattern-based behavior synthesis for FPGA resource reduction," in *Proceedings of the 16th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2008, pp. 107–16.
- [7] EEMBC Consortium, *DENBench Version 1.0, Benchmark Name: MPEG-2 Decode*, Feb. 2006, <http://www.eembc.org/>.
- [8] E. R. Gansner, E. Koutsofios, S. C. North, and K. Phong Vo, "A technique for drawing directed graphs," *IEEE Transactions on Software Engineering*, vol. 19, no. 3, pp. 214–30, Mar. 1993.
- [9] S. Girbal, O. Temam, S. Yehia, H. Berry, and Z. Li, "A memory interface for multi-purpose multi-stream accelerators," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Scottsdale, Ariz., Oct. 2010, pp. 107–16.
- [10] P. Ienne and R. Leupers, Eds., *Customizable Embedded Processors—Design Technologies and Applications*. San Mateo, Calif.: Morgan Kaufmann, 2006.
- [11] 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. Heidelberg, Germany: Springer, 2010, vol. 5952, pp. 126–40.
- [12] I. Kuon and J. Rose, "Measuring the gap between FPGAs and ASICs," in *Proceedings of the 14th ACM/SIGDA International Symposium on Field Programmable Gate Arrays*, Monterey, Calif., Feb. 2006, pp. 21–30.
- [13] M. J. S. Smith, *Application-Specific Integrated Circuits*. Boston, Mass.: Addison-Wesley, 1997.
- [14] *TMS320C64x DSP Library Programmer's Reference*, Texas Instruments, Oct. 2003, lit. no. SPRU565B.
- [15] *TMS320C64x Image/Video Processing Library Programmer's Reference*, Texas Instruments, Oct. 2003, lit. no. SPRU023B.
- [16] *TMS320C67x DSP Library Programmer's Reference*, Texas Instruments, Jan. 2010, lit. no. SPRU657C.
- [17] *ExpressDFG—Instruction Scheduling Benchmarks*, University of California, Santa Barbara, Calif., <http://express.ece.ucsb.edu/benchmark/>.
- [18] A. Ye and J. Rose, "Using bus-based connections to improve field-programmable gate-array density for implementing datapath circuits," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 14, no. 5, pp. 462–73, May 2006.
- [19] S. Yehia, S. Girbal, H. Berry, and O. Temam, "Reconciling specialization and flexibility through compound circuits," in *Proceedings of the 15th International Symposium on High-Performance Computer Architecture*, Raleigh, N.C., Feb. 2009, pp. 277–88.
- [20] J. W. Yoon, A. Shrivastava, S. Park, M. Ahn, R. Jayapaul, and Y. Paek, "SPKM: A novel graph-drawing based algorithm for application mapping onto coarse-grained reconfigurable architectures," in *Proceedings of the Asia and South Pacific Design Automation Conference*, Seoul, Korea, Jan. 2008, pp. 776–82.
- [21] M. Zuluaga and N. Topham, "Design-space exploration of resource-sharing solutions for custom instruction set extensions," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. CAD-28, no. 12, pp. 1788–1801, Dec. 2009.