Design and Architectural Exploration of Expression-Grained Reconfigurable Arrays

Giovanni Ansaloni, Paolo Bonzini, Laura Pozzi
Faculty of Informatics
University of Lugano
6900 Lugano, Switzerland
{giovanni.ansaloni, paolo.bonzini, laura.pozzi}@lu.unisi.ch

***Abstract*— Reconfigurable Arrays combine the benefit of spatial execution, typical of hardware solutions, with that of programmability, present in microprocessors. When mapping software applications (or parts of them) onto hardware, however, FPGAs often provide more flexibility than is needed, and do not implement coarser-level operations efficiently. Therefore, Coarse Grained Reconfigurable Arrays (CGRAs) have been proposed to this aim. While most CGRA designs feature an array cell of the order of an ALU, this paper proposes a new kind of coarse grained array, called EGRA (Expression-Grained Reconfigurable Array), featuring a cell composed of a cluster of ALUs with flexible interconnect. The EGRA attempts to further close the performance gap between reconfigurable and hardwired logic by implementing an arithmetic/logic expression per cell, rather than a single operation. A mapping methodology is proposed that can retargetably compile to a family of EGRAs, therefore enabling architectural exploration of the granularity of the proposed cell. Performance results on a number of embedded applications show that EGRAs can be used as a reconfigurable fabric for customizable processors, outperforming more traditional CGRA designs.**

I. Introduction

Reconfigurable (or field-programmable) arrays are flexible architectures that can perform execution of applications in a *spatial* way—much like a fully-custom integrated circuit—but retain the flexibility of *programmable* processors by providing the opportunity of reconfiguration.

The ability to exhibit application-specific features that are not "set in stone" at fabrication time would suggest reconfigurable architectures as particularly good candidates for being integrated in customizable processors. Unfortunately, other drawbacks have kept reconfigurable arrays from becoming a largely adopted solution in that field. Among different factors, the performance and area gap that still exists with hardwired logic is certainly one of the most important. The problem of bridging this gap has been the focus of much research in the last decades, and important advances have been made. This paper provides an additional step which goes in the direction of decreasing such gap further.

A walk through related historical background will help stating this paper's aims and contributions. In the earliest examples of reconfigurable architecture such as the PLA (Programmable Logic Array), mapping of "applications" (Boolean formulas in sum-of-product form) is immediate. In fact, each gate in the application is mapped in a *1-to-1* fashion onto a single gate of the architecture (Figure 1a).

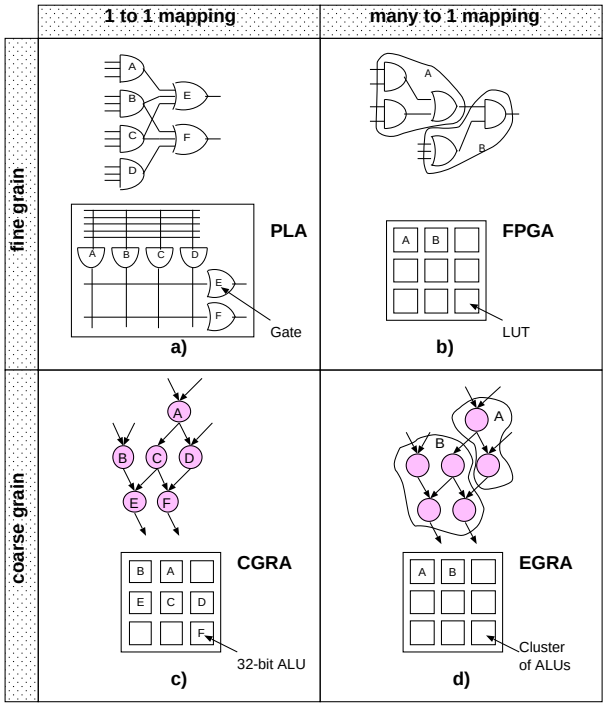


Fig. 1. Parallel between the evolution of fine-grained architectures from simple programmable devices to FPGAs (a and b), and the evolution of CGRAs from simple cells to the EGRA proposed here (c and d).

However, this organization does not scale as applications to be mapped get more complex. For this reason, CPLDs and FPGAs instead use elementary components—PLAs themselves, or look up tables—as building blocks, and glue them with a flexible interconnection network. Then, programming one cell corresponds to identifying *more than one gate* in the Boolean function representation (Figure 1b).

Introducing this additional level is a winning architectural choice in terms of both area and delay, but such innovations cannot be successful unless algorithms are available to efficiently map applications to the new architecture—and indeed efficient algorithms came along to this purpose, e.g., FlowMap [5].

An orthogonal step was the introduction of higher granu-

larity cells (Figure 1c). Fine grain architectures provide high flexibility, but also high inefficiency if input applications can be expressed at a level coarser than boolean (e.g. as 32-bit arithmetic operations). Coarse Grain Reconfigurable Arrays (CGRAs) provide larger elementary blocks that can implement such applications more efficiently, without undergoing gate-level mapping.

A variety of CGRA architectures exist (see Section V) but the process of mapping applications to current CGRAs is usually not very sophisticated: a single node in the application intermediate representation gets mapped onto a single cell in the array (again, *1-to-1* mapping). Instead, the architecture we propose in this paper (Figure 1d) employs an array cell consisting of a group of ALUs with customizable capabilities. We consider this the moral equivalent of the switch from single gates to LUTs that characterizes modern fine grain reconfigurable architectures. We call this cell RAC (Reconfigurable ALU Cluster), and the architecture that embeds it EGRA (Expression-Grain Reconfigurable Architecture).

This allows new and more efficient uses of CGRAs, for example by enabling implementation of application-specific functional units in a customizable processor [11]. However, such a change has to be supported by compilation technology: the proposed architecture would make little sense without a compilation flow able to map efficiently onto it. For this reason, we also show how a compiler can aid in the *architectural exploration* of the granularity of the cell.

To sum up, in this paper we propose a new architecture, the EGRA, which attempts to further close the performance gap between reconfigurable and hardwired logic by providing an effective reconfigurable platform for instruction-set extensible processors. Our contributions are as follows:

- We show a new design for the combinational part of coarse grain reconfigurable arrays. The Reconfigurable ALU Cluster (RAC), the complex cell at the heart of our architecture, supports efficient computation of entire subexpressions, as opposed to single operations. In addition, RACs can be connected either in a combinational or a sequential mode.
- We overview an automated methodology that can effectively map embedded applications on top of RACs. This in turn enables *systematic architectural exploration* of the EGRA design space, and in particular of the RAC structure, to motivate architectural choices with quantitative performance and cost analysis.

The remainder of this paper is structured as follows. Section II details the structure of the EGRA's processing element as well as its control logic, and presents synthesis results for different instances of the architecture. Section III details a mapping methodology that can compile a benchmark to a set of EGRAs whose RACs have different properties, and Section IV shows quantitative evaluation of the architecture's performance.

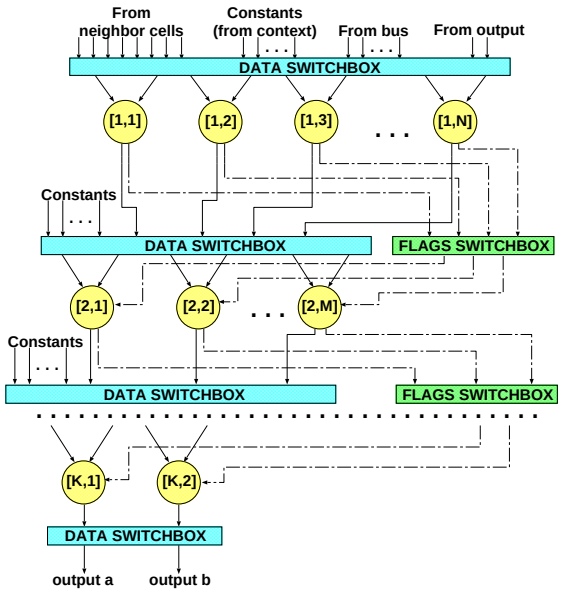


Fig. 2. Datapath of the Reconfigurable ALU Cluster.

II. Proposed Architecture

The architecture framework that we propose can be scaled in two dimensions: at the higher level, deciding the number of cells of the resulting array and the control unit implementation, and at the lower level, determining the implementation template of the RAC.

A. Cell architecture

The RAC datapath consists of a multiplicity of ALUs, with possibly heterogeneous arithmetic and logic capabilities, and can support efficient computation of entire subexpressions, as opposed to single operations. It is inspired by the Configurable Computation Accelerator proposed by Clark. [4] However, we use this structure as a *replicable* element; this has important consequences. First of all, it opens up the possibility to create combinational structures (in Clark's design, a CCA has a fixed multi-cycle latency) using multiple RACs; this favours designs featuring a smaller number of rows. Furthermore, it removes the limit on the number of inputs and outputs, because a pipelining scheme [16] can be used to move data in and out of the array; this allows scheduling of more complex applications and consequently higher gains.

ALUs are organized into rows (see Figure 2) connected by switchboxes. It is important to have flexible routing between ALUs on neighbouring rows, because subexpressions extracted from typical embedded applications often have complex connections that are not captured well by simpler topologies. This organization allows the usage of a simple array topology (we used nearest neighbour) without incurring high penalties on place-and-route.

The inputs of the RAC (see again Figure 2) are taken from the neighbouring cells' outputs, from the outputs of the cell itself, or (optionally) from a set of constants; the inputs of the

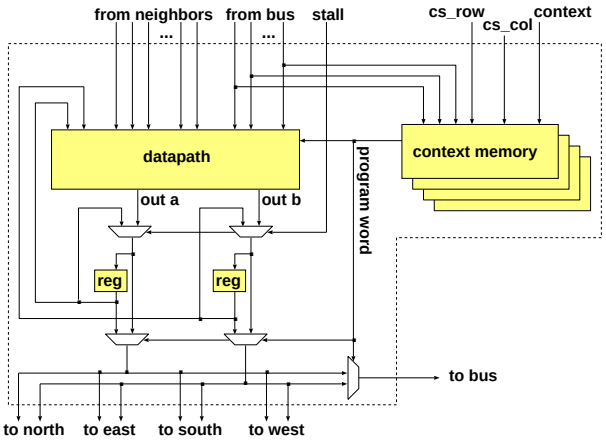
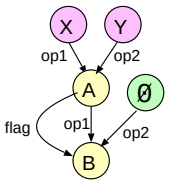


Fig. 3. Block scheme of a RAC composed of datapath, context memory and bypassable registers on the outputs.



node	opcode	op1 source	op2 source	flag source
A	op1 + $\overline{\text{op2}}$ + flag	BUS input 1	BUS input 2	1
B	flag ? op1 : op2	dataout(A)	constant 0	GEU(A)

Fig. 4. Programming a RAC. This example shows how two ALUs can be connected to compute an unsigned subtract with saturation, `(X >= Y) ? X - Y : 0`. The node computing the subtraction also performs the comparison. The multiplexer node B uses both the data output and the *unsigned* $\geq$ flag of the subtraction node A.

ALUs in subsequent rows are routed from the outputs of the previous rows or again from the constants.

The number of rows, the number of ALUs in each row and the functionality of the ALUs is flexible and can be customized by the designer. In fact, they constitute the *exploration level* explained in Section II-D.

The number and size of the constants is also defined at exploration time. If their width is less than the datapath width, their content is zero-extended[1]. The value of the constants, instead, is part of the configuration bitstream and can be different for each cell.

Being a reconfigurable design, the processing element includes not only a datapath, but also a *context memory*—see Figure 3. The context memory stores a number of possible configuration words, and can be programmed according to the desired functionality of the cell at configuration time.

B. ALU design

As in other CGRAs, the basic processing element of our cell design is an ALU. Unlike in the fine grain domain, it is not possible to define a generic component that can

[1]The availability of operations such as $A + \overline{B}$ makes it possible to store negative values even if the constants themselves are zero-extended.

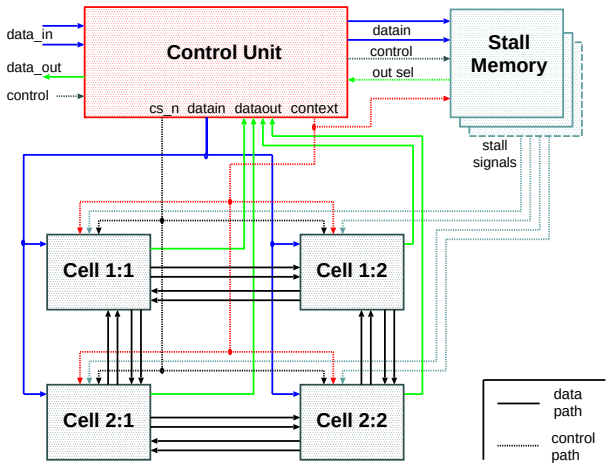


Fig. 5. CGRA functional blocks (2x2 matrix, mesh topology)

data opcodes	flag opcodes
$out = A \;\&\; (B \oplus flag_{sext})$	0
$out = A \mid (B \oplus flag_{sext})$	1
$out = A \oplus (B \oplus flag_{sext})$	$=$
$out = flag\ ?\ A\ :\ B$	$\neq$
$out = A + B + flag_{zext}$	signed $<$
$out = A + \overline{B} + flag_{zext}$	signed $\geq$
$out = A \ll B$	unsigned $<$
$out = A \ll_{rot} B$	unsigned $\geq$
$out = A \gg_{arith} B$	
$out = A \gg_{logical} B$	
$out = A \gg_{rot} B$	

TABLE I

LIST OF SUPPORTED OPCODES. NOTE THAT THE 1-BIT FLAG INPUT WILL BE SIGN- OR ZERO-EXTENDED DEPENDING ON THE OPCODE.

implement arbitrary functions, as is the case of the PLD or the LUT. Therefore, expressions are realized in our architecture by clustering more than one elementary unit (ALU) in one cell.

Four types of ALUs can be instantiated. The simplest one is able to perform bitwise logic operations only; the other three add respectively a barrel shifter (with support for shifts and rotates), an adder/subtractor, and both the shifter and adder. The list of operations in a full-featured unit is in Table I.

Each operation can generate three 1-bit flags: a *zero flag*, an *unsigned* $\geq$ *flag* (equivalent to the carry flag of general-purpose processors), and a *signed* $<$ *flag* (equivalent to $N \oplus V$, where N and V are the sign and overflow flags). Other conditions can be tested by complementing the flag, and/or exchanging the operands of the comparison.

Dually, each operation has *three* operands, two being 32-bit values and the third being a 1-bit value. The third operand can be hardcoded to zero or one, or it can be chosen from the flags that another ALU generated; it can also be complemented, thus giving a total of eight possible *flag opcodes* (also listed in Table I). Figure 4 explains graphically the way this functionality is programmed.

Flags enable efficient implementation of if-conversion—important when automatically mapping software representations onto hardware. In fact, ALUs can act as multiplexers,

# of rows	ALU type	ALUs per row		
		1	2	3
1	log	7 141	1 975	28 926
	log+sh	11 695	30 627	48 029
	log+add	9 125	22 802	35 124
	log+sh+add	12 438	35 105	53 837
3	log	10 586	30 971	57 384
	log+sh	21 740	66 490	113 648
	log+add	14 926	44 054	71 716
	log+sh+add	27 672	77 472	125 552
5	log	12 458	43 793	86 560
	log+sh	32 455	100 165	168 760
	log+add	20 186	65 134	114 034
	log+sh+add	40 583	123 294	202 633

TABLE II
DATAPATH AREA (IN μm^2) FOR DIFFERENT RAC CONFIGURATIONS

# of rows	ALU type	ALUs per row		
		1	2	3
1	log	0.45	0.54	0.55
	log+sh	0.62	0.66	0.71
	log+add	0.63	0.75	0.85
	log+sh+add	0.71	0.76	0.86
3	log	0.75	0.98	1.06
	log+sh	1.29	1.51	1.66
	log+add	1.54	1.88	2.18
	log+sh+add	1.57	1.89	2.32
5	log	1.05	1.49	1.68
	log+sh	1.93	2.43	2.68
	log+add	2.13	2.67	2.97
	log+sh+add	2.37	2.78	3.18

TABLE III
DATAPATH DELAY (IN ns) FOR DIFFERENT RAC CONFIGURATIONS

choosing one of the two 32-bit inputs based on another ALU's flags. This way, cells can evaluate both arms of a conditional, and choose between the two via a multiplexer.

C. Array architecture

The EGRA architecture is composed of a collection of RACs, organized as a mesh with nearest-neighbour connections and input/ouput connections to the external bus. The size (number of rows and columns) of the mesh can be defined at exploration time.

In addition to the cells, the EGRA includes a global control unit. This unit is in charge of managing the transfers to the RACs' context memory, selecting contexts, stalling cells until their output data is consumed, and connecting their outputs to the bus. In order to perform these tasks, the control unit also includes a separate context memory called the *control memory*. Individual control configurations are stored for each context that can be programmed on the EGRA.

Context memory transfers are initiated upon lowering of the chip select signal; the content of the data bus identifies the target of the transfer, either one of the RACs or the control memory. The value of the context control signals identifies which context is being programmed. Data is transferred on the following cycles using full input bandwidth, in order to minimize programming latency.

After configuration, context switches can happen at any clock cycle, and are performed without clock cycle penalties. The number of contexts that can be pre-loaded in the EGRA is defined at exploration time.

Stalling cells and connecting outputs is driven by the control memory. The control unit accesses it in a cyclic fashion; as read pointer advances to the next row on every clock cycle, unless reset by an external signal. The number of rows in the control memory identifies the maximum length (in cycles) of the expressions mapped on the EGRA, and can be decided at exploration time. In order to handle stalls, each RAC is assigned a memory column of this memory, and receives a one-bit stall signal on every clock cycle. Another section of the control memory asserts which RAC output is connected to the processor bus at any clock cycle.

Array size	area (μm^2)
1 x 1	62 622
2 x 1	117 748
2 x 2	237 485
3 x 2	375 526
3 x 3	589 911

TABLE IV
AREA OCCUPATION FOR DIFFERENT EGRA SIZES. ALL CONFIGURATIONS HAVE TWO CONTEXTS, 32 STALL/OUTPUT CONTEXT MEMORY LINES, RACS OF 4 FULL-FEATURED ALUS PER ROW ON 2 ROWS, TWO 8-BIT CONSTANTS PER RAC.

D. Architectural exploration

Choosing the configurable architectural features—RAC granularity, number of constants in a RAC, number of contexts in the array to mention a few—is not at all an obvious task and should be guided by performance evaluation. Therefore we define an *exploration level* where a number of cell and array features can be automatically varied and evaluated in different experiments.

Design-space exploration is made feasible by the availability of a compilation flow that can speedily evaluate many different design choices. Hence, the compilation and synthesis flows share a *machine description* detailing the cell template and the topology of the EGRA.

In order to investigate area and delay figures of the RAC datapath, we synthesized different versions using Synopsys Design Compiler and TSMC 90nm front-end libraries. This has been instrumental in achieving two goals: on one hand, collected data is used by the compiler to compute the performance of Instruction Set Extensions (ISEs) mapped onto the array; additionally, it gives insights on the efficiency of various EGRA configurations as a digital circuit, both in term of occupied silicon area and clock speed.

Tables II and III give area and delay results for different datapath configurations explored. All numbers refer to a datapath without embedded constants and with an equal number of ALUs on every row—neither of these, however, are actual limitations of a RAC template. In the present work, we mostly concern ourselves with the structure of the RAC, because our

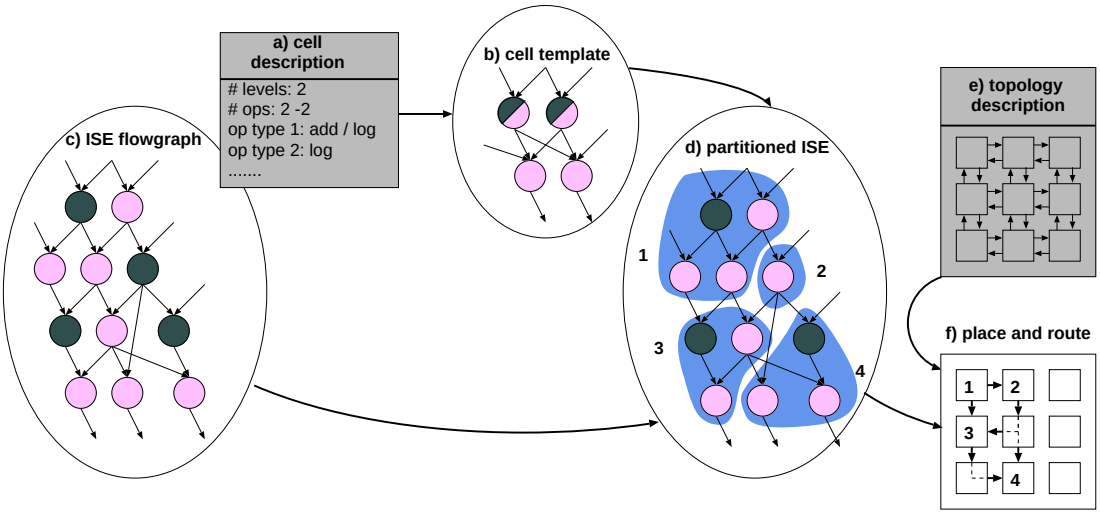


Fig. 6. Overview of the mapping methodology: the configuration to be evaluated and explored is described by a file detailing the cell structure (a, b). The data-flow graph (c) describing the application to be compiled—e.g., an ISE extracted from input code—is partitioned into many subgraphs, each of which to be mapped onto a single RAC (d). In this paper we do not consider placing and routing (f) of cells on the array given a topology description (e).

expression mapping methodology (see Section III) does not yet include place and route of cells on the array. In principle, however, the exploration level may include the mesh size, or even the topology of the array. As a hint of this possibility, Table IV shows examples of the area occupation of a full EGRA (including the control unit and global context memory) for a number of different array sizes.

III. Mapping Methodology

In order to evaluate the points of the design space, a compiler is needed that can map applications onto the array. The compiler's task is 1) to identify parts of the input application (Figures 6c) that profit from being executed onto an EGRA, and 2) to partition them in subgraphs that can each be executed onto a single RAC (Figures 6d).

In this paper, we consider the EGRA as an extension of a customizable processor, and extract portions of applications in the form of ISEs. Therefore, the first part of the compiler task is solved using well-known algorithms for automated ISE identification; we employ an enumeration algorithm similar to the one presented in [19], which extracts from the applications a set of maximal *candidates*.

Before going on with the second part of the task—partitioning—the compiler needs to run a series of *technology mapping* steps repeatedly on each candidate, in order to measure their gain and find a single best-performing one. After this phase, all operations are expressed in terms of the features of our ALU design, obtaining efficient calculations as in the example of Figure 4. In particular, comparisons must be transformed to an operation (typically a subtraction or an exclusive OR) that computes flags, so that users of the comparison can test a particular condition on those flags. In order to improve the utilization of the cells, after technology mapping the compiler reruns common subexpression elimination.

Each ISE identified by the previous step must now be partitioned into different subgraphs that can fit into a single cell of the array (Figure 6d). The process begins by enumerating *clusters* of the ISE graph that can be mapped on a single cell (this is done with a variant of the algorithm in [1]). Partitioning is then performed by picking the cluster that has most nodes on the critical path, and consolidating the nodes that form it into a single node (representing a RAC). After this, a retiming algorithm inserts registers between fragments based on the cycle time requested by the user. Since data-flow graphs are acyclic, we can use a simple, linear time algorithm [3] to do so, as reported in [16]. Finally, I/O operations between cells and register file are scheduled [19].

After these steps are performed for all ISE candidates, the one promising highest individual gain is chosen. This greedy choice is another possible source of non-optimality; nevertheless, it performs well under relatively broad conditions [15].

IV. Results

In order to collect results, we gathered DFGs from four MiBench [9] benchmarks using a GCC-based compiler front-end. The graphs were then placed into our compiler flow, which tested 872 different configurations. These configurations used RACs of one to three rows; the biggest one had 5 ALUs on the first row, 4 ALUs on the second, and 2 on the third. The register file bandwidth is limited to 2 reads and 1 write; higher bandwidth values would yield higher speedups.

The two audio benchmarks rawcaudio and rawdaudio, performing respectively ADPCM encoding and decoding, only use one context because a single ISE is identified by the compiler. The two crypto benchmarks des and sha use four.

Estimated clock cycle savings are plotted in Figures 7 to 10. Speedup is calculated as follows:

$$speedup = \frac{total\ cycles}{total\ cycles - \sum_{all\ ISEs}(cycles_{sw} - cycles_{hw}) \cdot freq}$$

where $freq$ is the number of times the ISE is executed, $cycles_{hw}$ is the latency of the ISE on the EGRA, and $cycles_{sw}$

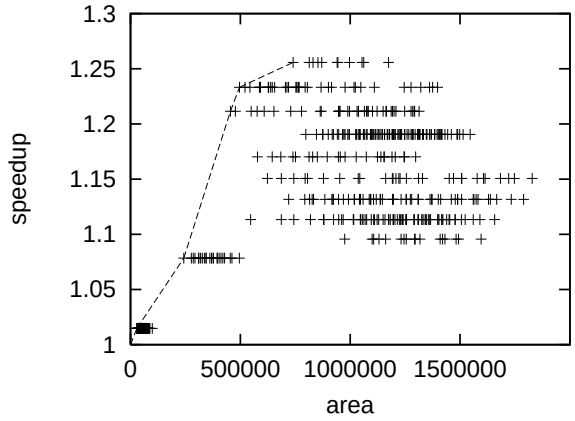


Fig. 7. Speedups obtained by 872 RAC configurations on rawcaudio

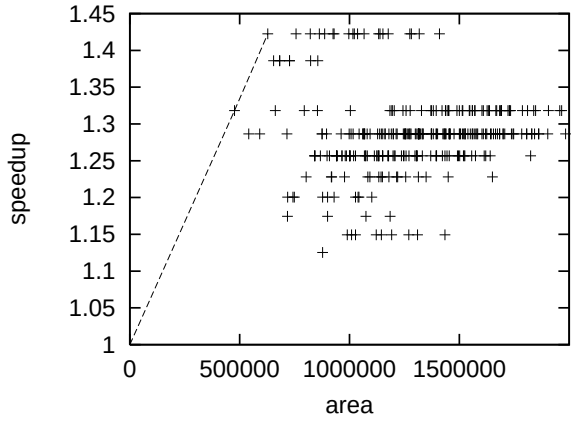


Fig. 8. Speedups obtained by 872 RAC configurations on rawdaudio

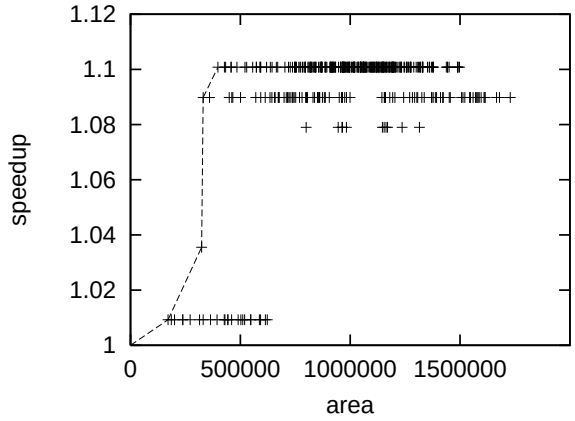


Fig. 9. Speedups obtained by 872 RAC configurations on des

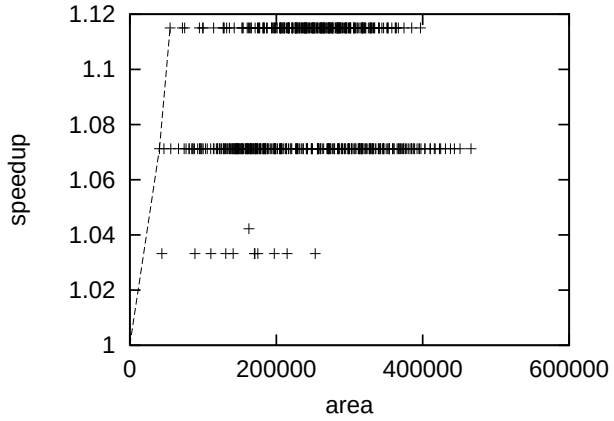


Fig. 10. Speedups obtained by 872 RAC configurations on sha

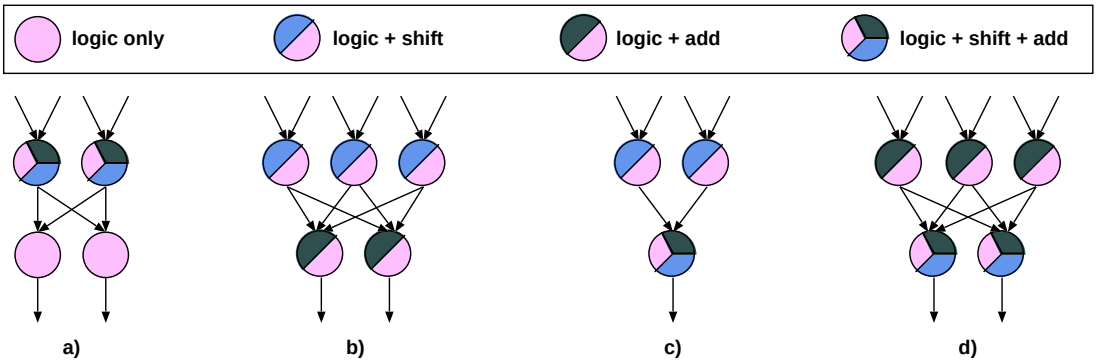


Fig. 11. RAC design of the maximum-speedup Pareto point configuration, for a) rawdaudio; b)rawcaudio; c) crypto benchmarks (des, sha); d) all four benchmarks.

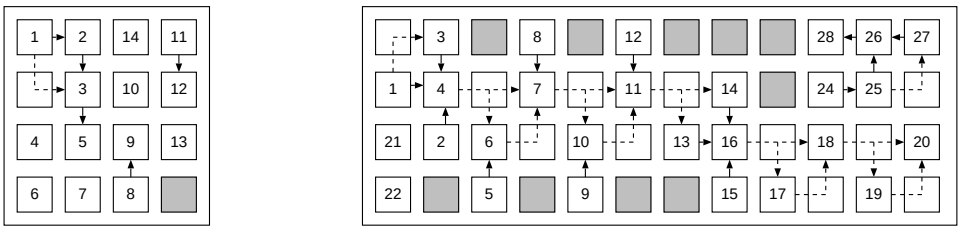


Fig. 12. Manual place-and-route of rawdaudio, for two cell designs. a) RACs as in Figure 11a. b) Each RAC only has one ALU.

	RAC[a]	ALU
Cell area (μm^2)	42 920	13 287
Cell delay (ns)	1.62	1.12
Array size	4x4	12x4
Array area (μm^2)	724 053	690 268
rawdaudio delay (ns)	11.26	22.16
rawdaudio delay (cycles at 150 MHz)	2	4

[a]as in Figure 11a

TABLE V
DATA FOR RAWDAUDIO AFTER PLACE-AND-ROUTE.

is the cost of executing the ISE without custom instructions. Because $cycles_{hw}$ is integer and bounded by $cycles_{sw}$, the plotted speedups can take only a few discrete values, as observed in the figures.

Our experimental setting essentially considers an EGRA as an application-specific functional unit of a customizable processor. The speedup results of this paper show that, in such setting, multi-ALU cells outperform single-ALU cells found in more traditional CGRA designs. In fact, cells consisting of only one row correspond to the low-area points in the plottings, and have barely noticeable speedups.

Figure 11 shows four RAC designs. The first two represent the configuration of the maximum-speedup Pareto point, i.e. achieves the maximum speedup at minimal area cost, for each of the audio benchmarks; the third achieves maximum speedup on both crypto benchmarks; the last finally performs well on all benchmarks but costs noticeably more area than specialized cells. It is important to note that trivially merging the features of the cells in Figures 11a and 11b would use more area than Figure 11d, without improving performance.

All three solutions are 2-row RACs. It is interesting that, despite the apparent similarity between the design of the RAC and the CCA, they are much smaller than the examples of *Configurable Computation Accelerator* presented by Clark [4]. The reason is that RACs can be connected to form combinational structures. This features puts smaller cells to an advantage, since they will usually have higher utilization rates without sacrificing speed.

In order to evaluate fully the gains of the proposed architecture over a simpler CGRA with one ALU per cell, we performed place-and-route by hand on rawdaudio for two RAC configurations: a single-ALU cell, and the optimal RAC of Figure 11a.

The resulting layouts in Figure 12, and the area and delay numbers in Table V, show how the more complex interconnect of the RAC allows more effective routing, so that even a simple nearest neighbour topology is a viable choice for the EGRA. Indeed, the RAC-based design achieves a very compact packing of the computation in the array; 14 used RACs fit in a 4x4 array with only one cell (not on the critical path) used for routing only. Instead, 11 cells are used for routing when each cell can only perform one arithmetic operation, ten of which are on the critical path.

For this reason (see Table V), the two designs occupy roughly the same area after place-and-route, and the single-ALU one is twice as slow—in practice, it would fail to achieve any speedup over a general-purpose processor.

V. RELATED WORK

In the past years, several Coarse Grain Reconfigurable Architectures (CGRAs) have been proposed [12]. The definition is broad, and includes designs that differ widely even in the very *coarseness* of the cell. For example, the cell will usually implement a single execution stage, but may also include an entire execution unit (Rapid [6]) or can even be a general purpose processor (RAW [20]).

The most relevant to our work is probably the *Flexible Computational Component* [7] which, while targeted more specifically to DSP kernels, is similar to the RAC in size and set of allowed operations. However, the authors do not present an exploration methodology to explain quantitively their choices.

ADRES [13], [14] also features a complex VLIW cell. Even though it lacks the ability to perform multi-stage computations within a cell, it features strong instruction-level parallelism capabilities that can be exploited by the compiler through software pipelining [13].

The architectural choices that drove the above-mentioned designs are usually the result of the designer's expertise, more than of systematic, quantitative exploration of the design space. Therefore, the resulting designs have a fixed structure. Even when some flexibility is present (as in ADRES or Rapid), results for exploration are presented only for high-level cycle-base exploration, or not given at all. The work of Bouwens [2] is somehow an exception as it demonstrates design space exploration at the synthesis and place-and-route level for the ADRES architecture. However, it focuses on CGRA high level topology, without investigating the structure and coarseness of the processing elements as done here.

Our HW/SW co-exploration of different EGRA architectures is based on a machine description interface, shared by the synthesis and compilation flow; this concept is independent from the proposed RAC structure and has taken inspiration from recent researches in architectural description languages [17], [10].

Different approaches have been envisioned also for the CGRA level of integration in the architecture hierarchy, ranging from coprocessor (Morphosys) to tight coupling with the processor (ADRES). Our work belongs to the latter family, but the multi-ALU processing unit we presented could in principle be applied to differently integrated architectures. In this sense, the present work constitutes a contribution to the reconfigurable computing field even outside the specific implementation detailed in this paper.

Concerning the proposed design of the EGRA, our cell design is inspired (as mentioned in Section II-A) by the CCA structure proposed by Clark [4]. Besides the idea of replicating this structure, we introduced several other novel aspects, in particular the ability to build combinational functions from multiple CCAs, and the usage of flags to support if-conversion in the compiler. Flags are peculiar in the RAC design, and are

inspired by the program status word of a microprocessor, more than by the carry chains available in many reconfigurable architectures (for example, Stretch [18] or even PipeRench [8]).

VI. Conclusions

In this paper, we have proposed a new coarse-grained architecture, the EGRA. The architecture builds on the well-proven advantages of CGRAs over FPGAs and ASICs (little area and delay overhead due to programmability over FPGAs, adaptivity compared to ASICs) advancing them further through the introduction of a novel, flexible computing element (the RAC) that can evaluate complete subexpressions at once.

In order to analyze the performance of this new reconfigurable fabric, we have identified a set of parameters that identify an *exploration level* and can be varied to evaluate *quantitatively* the performance of the architecture for different benchmarks. Evaluation is aided by automated tools that map benchmarks on an EGRA. The tool uses area and delay estimates, derived from Synopsys synthesis of different RACs, to compile for an entire family of architectures.

Research on the EGRA is still in its infancy, and the initial results presented here suggest multiple directions for future work. For example, the architecture could implement various local memory configurations, so that entire loops can be mapped onto the EGRA and loop pipelining techniques can be applied. Still, our experimental study determined the feasibility of using the EGRA to implement Instruction Set Extensions (ISE) in a custom processor, achieving speedups of up to 1.45x—competitive with other purely arithmetic accelerators—and showing that two-row RACs achieve good performance results and utilization rate.

References

[1] K. Atasu, L. Pozzi, and P. Ienne. Automatic application-specific instruction-set extensions under microarchitectural constraints. In *Proceedings of the 40th Design Automation Conference*, pages 256–61, Anaheim, Calif., June 2003.

[2] F. Bouwens, M. Berekovic, A. Kanstein, and G. Gaydadjiev. Architectural exploration of the ADRES coarse-grained reconfigurable array. In *Reconfigurable Computing: Architectures, Tools and Applications*, volume 4419 of *Lecture Notes in Computer Science*, pages 1–13. Springer, Berlin, June 2007.

[3] P. Y. Calland, A. Mignotte, O. Peyran, Y. Robert, and F. Vivien. Retiming DAG's. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 17(12):1319–25, Dec. 1998.

[4] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner. Application-specific processing on a general-purpose core via transparent instruction set customization. In *MICRO 37: Proceedings of the 37th Annual International Symposium on Microarchitecture*, pages 30–40, Washington, DC, USA, Dec. 2004. IEEE Computer Society.

[5] J. Cong and Y. Ding. Flowmap: An optimal technology mapping algorithm for delay optimization in lookup-table based FPGA designs. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(1):1–12, Jan. 1994.

[6] C. Fisher, K. Rennie, G. Xing, S. G. Berg, K. Bolding, J. H. Naegle, D. Parshall, D. Portnov, A. Sulejmanpasic, and C. Ebeling. An emulator for exploring RaPiD configurable computing architectures. In *Proceedings of the 10th International Conference on Field-Programmable Logic and Applications*, pages 17–26, Jan. 2001.

[7] M. Galanis, G. Theodoridis, S. Tragoudas, and C. Goutis. A high-performance data path for synthesizing DSP kernels. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 25(6):1154–1162, June 2006.

[8] S. Goldstein, H. Schmit, M. Moe, M. Budiu, S. Cadambi, R. R. Taylor, and R. Laufer. PipeRench: A coprocessor for streaming multimedia acceleration. In *Proceedings of the 26th Annual International Symposium on Computer Architecture*, pages 28–39, May 1999.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proceedings of the IEEE 4th Annual Workshop on Workload Characterization*, pages 3–14, Dec. 2001.

[10] A. Halambi, P. Grun, V. Ganesh, A. Khare, N. Dutt, and A. Nicolau. EXPRESSION: a language for architecture exploration through compiler/simulator retargetability. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 485–490, Mar. 1999.

[11] T. R. Halfhill. ARC Cores encourages "plug-ins". *Microprocessor Report*, 19 June 2000.

[12] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 642–649, Mar. 2001.

[13] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proceedings of the IEEE International Conference on Field-Programmable Technology*, pages 166–173, Dec. 2002.

[14] B. Mei, S. Vernalde, D. Verkest, and R. Lauwereins. Design methodology for a tightly coupled VLIW/reconfigurable matrix architecture: A case study. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*, pages 1224–1229, vol.2, Feb. 2004.

[15] L. Pozzi, K. Atasu, and P. Ienne. Exact and approximate algorithms for the extension of embedded processor instruction sets. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, CAD-25(7):1209–29, July 2006.

[16] L. Pozzi and P. Ienne. Exploiting pipelining to relax register-file port constraints of instruction-set extensions. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 2–10, San Francisco, Calif., Sept. 2005.

[17] S. Rigo, G. Araujo, M. Bartholomeu, and R. Azevedo. ArchC: A SystemC-based architecture description language. In *Proceedings of th 16th Symposium on Computer Architecture and High Performance Computing (SBAC-PAD'04)*, pages 66–73, Oct. 2004.

[18] C. R. Rupp. Multi-scale Programmable Array. U.S. Patent 6633181, Oct. 2003.

[19] A. K. Verma, P. Brisk, and P. Ienne. Rethinking custom ISE identification: A new processor-agnostic method. In *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, pages 125–134, Salzburg, Austria, Oct. 2007.

[20] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, and A. Agarwal. Baring it all to software: Raw machines. *Computer*, 30(9):86–93, Sept. 1997.