Compiling Custom Instructions onto Expression-Grained Reconfigurable Architectures

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

ABSTRACT

While customizable processors aim at combining the flexibility of general purpose processors with the speed and power advantages of custom circuits, commercially available processors are often limited by the inability to reconfigure the application-specific features after manufacturing. Even though reconfigurable array-based accelerators are available, their performance is often unacceptable, and comes with other disadvantages such as the size of the configuration bitstream. Additionally, compilation support is limited for existing Coarse Grain Reconfigurable Arrays (CGRAs).

We propose to target a different reconfigurable fabric, the EGRA (Expression-Grained Reconfigurable Array), to realize custom instructions in a customizable processor. The EGRA is based on arithmetic processing elements that can compute entire subexpressions in a single cycle and can be connected in both combinational or sequential manners. We present here a compilation flow for this architecture, including novel algorithms for subgraph enumeration and scheduling. The compilation flow proposed is used here to efficiently explore the design space of the EGRA processing element; furthermore, its modularity and flexibility suggest suitability to generic CGRA retargetable compilation.

Categories and Subject Descriptors

B.1.4 [**Hardware**]: Control structures and Microprogramming—*Microprogram design aids*; C.1.3 [**Computer Systems Organization**]: Processor Architectures—*Data-flow architectures*; C.3 [**Computer Systems Organization**]: Special-Purpose and Application-Based Systems—*Embedded systems, Reconfigurable computing*; D.3.4 [**Processors**]: Optimization—*Code generation*

General Terms

Algorithms, Performance, Design

CASES'08, October 19–24, 2006, Atlanta, GA, USA.

Keywords

Coarse-grained reconfigurable architectures, Compilers, Data-flow architectures, Horizontal microprogramming, Instruction set extensions

1. INTRODUCTION

The aim of customizable processors is to combine the small price and the flexibility of general purpose processors, with the speed and power advantages of custom circuits (ASICs). In such processors, a standard machine language can be augmented with *custom instructions* (also known as *instruction set extensions*, or ISEs) that execute on separate application-specific hardware units. Therefore, while retaining the flexibility of processors in terms of programmability, carefully designed hardware accelerators are able to execute applications in a spatial way, much like a fully-custom integrated circuit.

The additional 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. However, many of the processors that are available in the market are not based on reconfigurable architectures.

Currently available customizable processors include some that couple a hard-wired processor with an accelerator based on FPGA technology. However, they often have disappointing results in terms of speedups. In fact, on one end of the spectrum, ASICs provide maximum speed but heavily limit the flexibility of the solution. At the other end, fine-grained reconfigurable fabrics (such as the ones in FPGAs) are extremely flexible, but at a serious performance cost.

Not being able to refine the processor's customization after deployment is a limiting factor for many reasons. For example, even though different applications may be known to run only in different time frames, the area cost for all of them has to be paid. Additionally, firmware upgrades are penalized in their access to the customization features, because they cannot define new application-specific instruction set extensions. These two problems would be felt clearly, for example, in software-defined radios—which would otherwise be an interesting application for customizable processor because of the heavy signal-processing tasks they perform.

If, as is often the case, the accelerated computations need not be expressed as Boolean formulas (e.g. they are 32-bit arithmetic operations), the larger elementary blocks of Coarse-Grained Reconfigurable Arrays could successfully

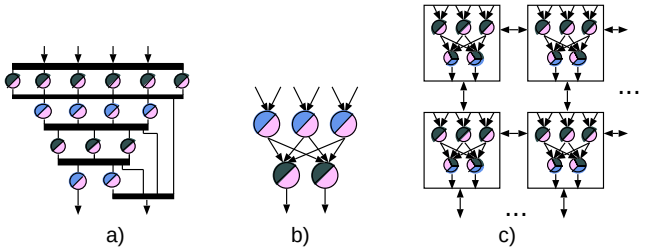


Figure 1: a) The structure of a CCA [2]. b) While smaller in size to limit latency, the RAC has a similar structure. c) The EGRA, consisting of a multiplicity of RACs.

implement such applications more efficiently, without undergoing gate-level mapping. For this reason, in [1] we introduced the Expression-Grained Reconfigurable Array (EGRA), featuring a new design for the processing element and enabling implementation of application-specific functional units in a customizable processor.

Similar to an FPGA, the EGRA is based on combinational processing elements (Reconfigurable ALU Clusters, or RACs) that can be connected either to form a larger combinational structure, or in a sequential mode of operation through a register.

A single RAC can compute entire arithmetic/logic subexpressions using multiple wired ALUs. As opposed to cells consisting of a single ALU, the use of RACs has several advantages. Expression computation will be faster and, even more than in other coarse-grained architectures, the structure of the interconnect can be simplified. Compared to FPGAs, the configuration bitstream is shorter; this shortens reconfiguration times and allows storage of multiple configuration contexts in the array.

However, in order to ascertain the actual gains from this approach, it is necessary to support the EGRA with compilation technology that can map applications efficiently onto the new architecture. While compilation support is of capital importance for all CGRAs in general, albeit often missing, it is particularly so in the case of the EGRA, because an EGRA compiler can enable systematic exploration of the vast RAC design space.

The remainder of this paper is organized as follows. Section 2 overviews the architecture of the EGRA, and Section 3 details the algorithms we used or developed for this task; Section 4 presents experimental results. Section 5 explains how the scheme we developed can be generalized to a framework for CGRA compilation. Finally, section 6 presents related work and section 7 concludes the paper.

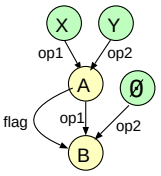
2. THE EGRA ARCHITECTURE

The main feature of the target architecture is its attempt to integrate multiple computational elements in a single cell. The design of the RAC is modelled on the *Configurable Computation Accelerator* (CCA) [2], an acyclic array of arithmetic units that can be used to implement common computations (Figure 1). Like the CCA, the RAC supports efficient computation of entire subexpressions; however, within the EGRA its structure is replicated. This way, the array can better reproduce the *spatial execution* abilities at which custom integrated circuits excel.

Each RAC includes a multiplicity of arithmetic elements,

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 1: List of supported opcodes. Note that the 1-bit flag input will be sign- or zero-extended depending on the opcode.



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)

Figure 2: 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.

organized as rows of ALUs connected by switchboxes. The number of rows, the number of ALUs in each row and their functionality (such as whether they include an adder or a barrel shifter) can be explored in order to obtain best performance with minimal area cost.

Another characteristic of the RAC is the availability of three-input operations, such as word-sized 2:1 multiplexers. This enables efficient implementation of if-conversion, and allows RACs to evaluate in parallel the two arms of a conditional, and use a multiplexer to choose between the two.

In fact, all operations are extended to have three inputs, as in table 1. The third input can be hardcoded to 0 or 1, or can come from one of three 1-bit flags. The design of the flags is inspired by the program status word of most microprocessors. They are a *zero flag*, an *unsigned* $\geq$ *flag* (equivalent to the carry flag of general-purpose processors, such as the ARM), and a *signed* $<$ *flag* (equivalent to $N \oplus V$, where N and V are the sign and overflow flags); they can either be used directly, or complemented. By exchanging the operands of the comparison, or by complementing flag values (see table 1), it is possible to implement all signed and unsigned relational operations.

The array has input/output connection with the processor bus, thus allowing it to read and write back data from/to the

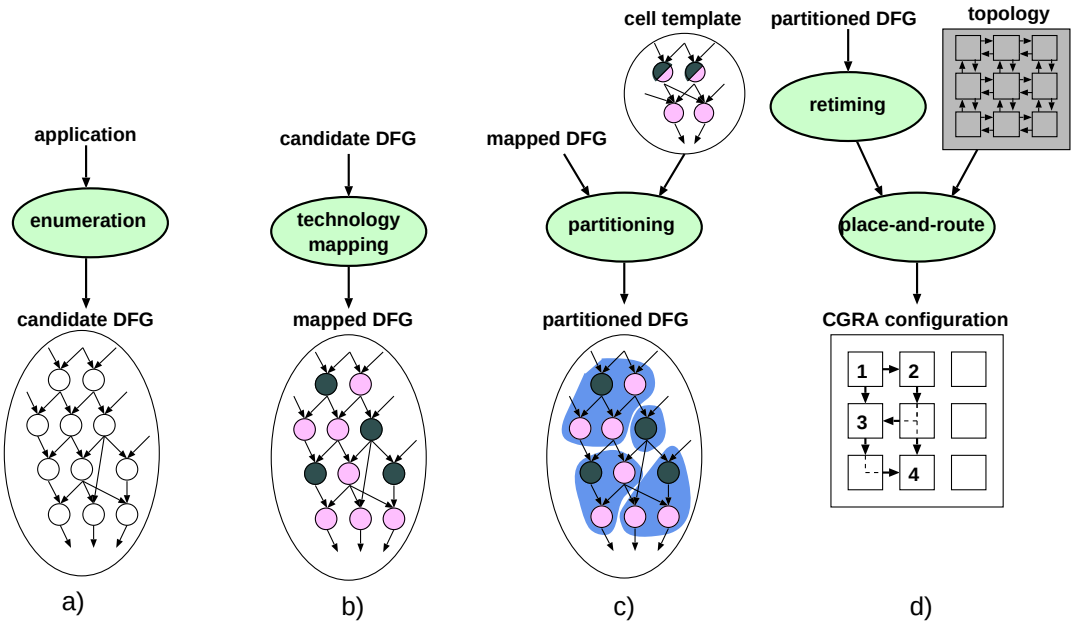


Figure 3: Overview of the mapping methodology. a) Extracting hot parts of the application to be compiled. b) Rewriting the DFG to the set of operations supported by the processing element. c) Partitioning the DFG to allocate the operations to different array cells. d) Placing and routing produces the configuration of the CGRA.

register file, only limited by the bandwidth of the register file itself. We assume the presence of 2 read ports and 1 write port. Therefore, up to two 32-bit values will be distributed to the cells from the register file, and on every clock cycle one cell will be able to send one value back to the register file.

3. MAPPING ON EGRA

The added flexibility of the EGRA cell architecture enables the designer to evaluate different types of RACs. Hundreds of different types can be enumerated by varying the height of the cell, the width of each row, and the type of ALU used on each row. In this context it is very important to perform this evaluation automatically and quickly, using an efficient compiler to map onto a family of architectures and estimate the gains of different cell types.

The methodology we propose borrows from work on the topic of compilation for customizable processors. The traditional compiler is equipped with an additional phase that analyzes the program at a certain, pluggable point of the optimization pipeline and performs hardware-software codesign, i.e., identifies portions of the code to be moved to the CGRA and executed as custom instructions. Up to this step, the compiler treats each operation in the program separately; afterwards, the entire data-flow subgraph identified is collapsed into a single atomic unit.

We split the process of mapping into several phases, as described in Figure 3. The compiler first identifies candidate parts of the program that are suitable for execution on the accelerator. Based on this list of candidates, it runs a series of *technology mapping* tasks repeatedly in order to find a single best-performing one. These tasks include optimizing the subgraph for the chosen accelerator, clustering nodes of the subgraph that can be mapped onto a single array cell, and then transforming these clusters into an actual implementation of the candidate on the accelerator.

After these steps are performed for all the candidates (or, as mentioned before, on a subset thereof), the single best performing subgraph is chosen.

3.1 Maximal subgraphs as candidates

Our mapping flow can extract parts of a basic block to the accelerator and leave the rest to the general purpose processor. Therefore, candidates can be in principle any subgraph of the basic block.

These subgraphs should satisfy two conditions. First of all, they should not include operations that are not supported by the cell. Second, they should be convex; that is, any path between two nodes in the candidate should only include nodes that are themselves part of the candidate. The subgraphs do not need to have any constraint in term of number of inputs and outputs; in case these exceed the bandwidth of the register file, it is possible to serialize them across multiple cycles. In order to limit the search, we add a third condition, namely we restrict our attention to subgraphs that are *maximal*; that is, growing them in any way would violate the two previous conditions.

In order to do so, we employ the enumeration algorithm presented in [3]. As it reduces maximal subgraph enumeration to *clique enumeration*, which is a well-known problem with exponential time complexity, maximal subgraph identification is also worst-case exponential in the number of operations in the basic block; in general, however, it will complete in no more than a few seconds.

The main limitation of the algorithm is that it does not offer any control on the area of the generated subgraphs. This is often not a problem, because the algorithm generates

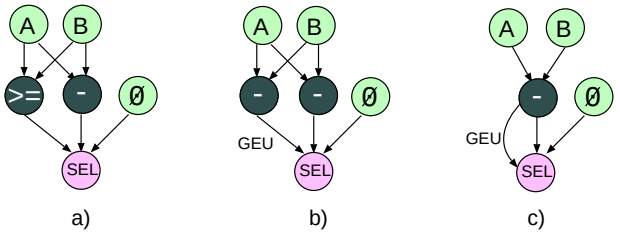


Figure 4: Lowering a data-flow graph to the set of operations supported by the cells. This example shows a saturating unsigned subtraction. a) The DFG representing the C source `A >= B ? A - B : 0`. b) The comparison is transformed into a subtraction and the appropriate flag (unsigned greater-than-or-equal) is tested in the SEL node. c) Common subexpression elimination merges the two subtraction nodes into one.

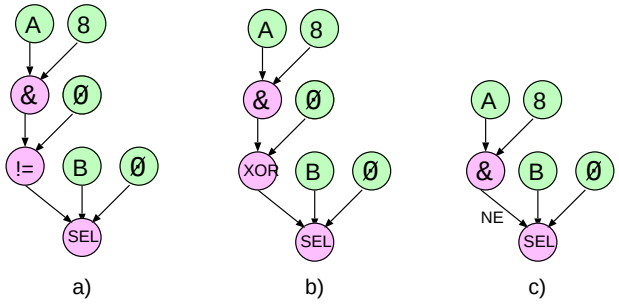


Figure 5: Peephole optimization during lowering. a) Example is taken from the ADPCM benchmark. b) Equality and inequality comparisons are converted to exclusive-ORs. c) Comparisons with zero do not need a separate node, because the AND node is already computing the right value of the flags.

many overlapping candidates and it is rare that none satisfies area constraints; in our experiments, EGRA instruction-set extensions require around 20 cells.

Each candidate needs further processing before its gain can be measured. We call this step *technology mapping*. In principle, each of the candidate subgraphs discovered in the first step could go through the technology mapping step. However, it is better to compute simple lower bounds on the cycles needed to execute the candidate on the accelerator. For the EGRA, for example, the register file bandwidth and the number of inputs and outputs (which must be transferred from/to the processor registers) together provide such a lower bound. Based on this, upper bounds on the gain can be estimated and used to avoid executing the expensive steps for clearly suboptimal subgraphs.

3.2 Operation lowering using graph rewriting

The next step of the compilation flow is operation lowering; its main aim is to effectively use the flags provided by the EGRA architecture. Flags can be used to implement if-conversion (see figures 4 and 5), but they will also be used to implement C logical operations (i.e. using the truth value of a comparison in an arithmetic operation) and to transform special nodes that are included in the intermediate representation, such as absolute value and minimum/maximum.

In all cases, comparisons must be transformed to an operation (a subtraction or an exclusive OR) that will compute flags; users of the comparison can then test a particular condition on those flags. In addition, operands of the comparison can be manipulated in order to support conditions that the ALUs cannot generate, i.e. $>$ and $\leq$.

These transformations are implemented using graph rewriting. In this system, each lowering rule is described by a pair of graphs—a *template* to be matched in the candidate data-flow graph, and a *replacement* graph that will be substituted for it. In [4], the authors use a similar methodology, based on term rewriting. Since our compilation flow is based on data-flow graphs rather than on an imperative intermediate representation, using graph grammars was a natural choice.

The system can also perform common subexpression elimination, as in Figure 4c. Simple peephole optimizations are also performed; for example, Figure 5c shows that comparisons with zero can be merged in the node that computes the other operand of the comparison. The CSE and peephole optimization phases were also described as graph rewrites.

Graph rewriting is a very powerful technique, and several libraries are available to program graph rewriting systems. In particular, we used the GrGen library. [5]. The graph grammar for lowering is small, consisting of only 10 rules, and the rules for CSE and peephole optimization had approximately the same size.

3.3 Partitioning

Technology mapping produces a new DFG, where the set of used operations matches closely the capabilities of a single ALU of the EGRA. In other words, technology mapping defines how the array's ALU will execute each operation in the data-flow graph. The next step is to coarsen the granularity, and find a mapping at the RAC level; to this end, the partitioning phase clusters operations that will execute in the same cell. The output is a new graph that collapses these clusters to a single node; this graph more closely represents the execution of the candidate DFG on the EGRA.

Covering the candidate with a number of clusters is in turn a multi-step process, comprising enumeration of the possible clusters, analyzing their timing characteristics (for example eliminating those that exceed the chosen cycle time), and then selecting, among the enumerated and valid ones, a set of clusters that partitions the candidate.

Clusters can also be seen as data-flow graphs. The most important difference between the candidate subgraphs enumerated as the first step of the process, and the cluster graphs considered here, is that clusters are constrained to the set of computations that a cell can perform. The most important consequence this has for EGRA mapping is that the number of outputs of a cluster is limited by the number of ports of a single cell, and this allows adaptation of efficient algorithms from the state-of-the-art [6].

After a list of clusters is found, it is possible to find a partitioning by using a greedy covering algorithm that tries to place the deepest available cluster on the critical path. Deep clusters have higher utilization rates, and minimize the routing delay on the critical path both within a RAC and in the

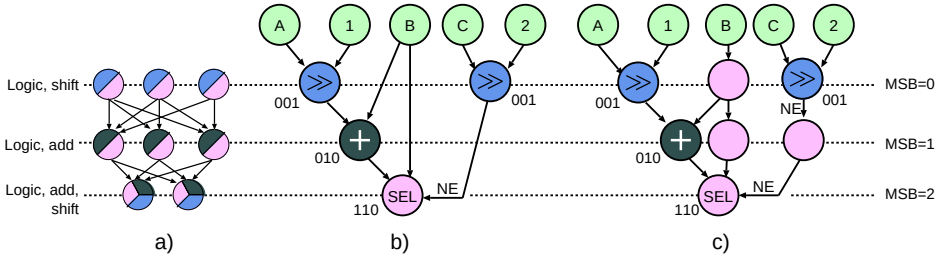


Figure 6: Scheduling operations in a RAC. a) A cell template. b) A cluster that has edges spanning multiple rows. The enumeration algorithm assigns bitmasks to each operation, corresponding to the rows on which the operation can be executed. c) By placing ops on the row indexed by the MSB, one obtains the as-late-as-possible schedule; in this case, 2 ALUs must be configured to pass the value through, and 1 must pass the equality flag.

entire array. This is a comparatively simple problem, thus the rest of this section will concentrate on efficient enumeration and scheduling of the clusters.

3.3.1 Cluster enumeration

Techniques to enumerate subgraphs efficiently are well known. The basic enumeration technique employs a binary search space, where each level of the tree represents a node in the enumerated graph. At each point, two choices are possible, corresponding to including or not the node in the graph. A single subgraph of a DFG corresponds to a leaf. The key to efficient implementation is in bounding the search, avoiding the exploration of branches that correspond to an invalid subgraph, and where no valid subgraph may exist in the branch. Then, the number of search tree points actually visited will grow relatively slowly with n.

In the case of the EGRA, enumeration should consider these constraints:

- clusters should be convex;
- clusters should have no more outputs than the RAC has;
- the maximum depth of the cluster is constrained by the number of rows in the RAC;
- not all rows of the RAC can execute all operations;
- cluster rows should not be wider than RAC rows.

Compared to the I/O-constrained subgraph enumeration problem [6], the most obvious difference is the lack of a constraint on the number of inputs. Instead, the maximum depth poses a strong limit on valid EGRA clusters.

Algorithms such as [6] are not aware of the depth limit. Hence, they will needlessly enumerate very large graphs that cannot fit in an array cell. In fact, Clark [7] reported enumeration times well over 10 minutes for large basic blocks; this would be impractical in the context of architectural exploration. Therefore, it makes sense to develop an alternative enumeration algorithm that can use depth to prune the search. As we report in section 4, execution of our enumeration algorithm took less than one millisecond on the benchmarks we tried.

Enumeration does not need to provide a "perfect" set of clusters. It can be conservative and emit clusters that are invalid (i.e., that cannot map onto an array cell). For this reason, the clusters are subject after enumeration to a separate check that does not have false negatives. In the most general case, the compiler can consider a cluster to be valid, and at the same time obtain its mapping onto a cell, if it is a subgraph of the cell template. This *subgraph isomorphism* test is used for example by Clark [7].

In our case, enumeration filters successfully the first four constraints above. RAC width, instead, is checked after assignment of cluster operations to RAC ALUs (*scheduling*).

As in [6], enumeration visits the tree in reverse topological order. It propagates information to the predecessors of each node, and uses it to trim the search when those nodes are reached. In particular, the algorithm of [6] propagates information on whether adding a node would violate the convexity constraint, and on the number of successors that are already part of the subgraph. The latter piece of information is used as nodes are added or removed from the subgraph, to quickly update the number of inputs and outputs.

Our algorithm propagates an additional value, namely a bitmask of *rows that can implement an operation* in the cell. This bitmask can be computed beforehand for every opcode used by the intermediate representation, and its value can be used to initialize the field when a node is added to the cluster. We will call this per-opcode bitmask the *op-mask*.

Since masks are read like binary numbers, the most significant bit (corresponding to the bottom line of the RAC) is written first. So, for the example RAC of Figure 6a, the op-mask of *SEL* nodes is *110*. In this case, the value *110* includes rows that support logic operations (all three do) and have a flag input (the two bottom rows). The op-mask for addition is also *110* (this time, because the top row does not have an adder) and the op-mask for shifts is *101*.

The bitmask of a node is computed from its op-mask and, if the node has successors in the cluster, from the successors' bitmasks. If b_j is the bitmask of node j, the index $MSB(b_j)$ of the most significant set bit of b_j indicates the last RAC row in which b_j can run; predecessors of j will have to be allocated on a row in the range 0 to $MSB(b_j) - 1$. This corresponds to ANDing the op-mask with $(1 \ll MSB(b_j)) - 1$, and using the result as the bitmask for the newly added node; if the result is zero the cluster is invalid—and adding other nodes will not restore its validity, so that an entire branch of the search tree can be pruned.

If more than one successor is already in the cluster, masking can be applied to all of them—possibly, each of them will further restrict the bitmask of valid rows for the node that

opcode	delay
>>	0.1062
+	0.1329
SEL	0.0764

row	mux delay
logic	0.0697
full	0.0963
other	0.0863

a)

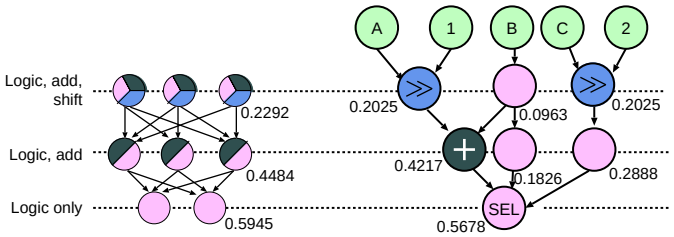


Figure 7: Computing the critical path delay. a) Synopsys Design Compiler is used to compute the critical path delay of the ALU components and of the multiplexer between rows. b) The critical path delay Synopsys gives for the cell template is the worst-case delay. c) Knowing the actual opcodes allows the compiler to produce a better estimate of the critical path delay.

enters the subgraph. Figure 6b shows an example cluster, together with the bitmasks that were attached to each of its nodes.

It is important to notice that the algorithm will never emit clusters whose depth is greater than the cell's, because $MSB((1 \ll MSB(b_j))-1) < MSB(b_j)$. In other words, the bitmasks become smaller and smaller as the cluster's latency increases, and will always be zero if it exceeds the RAC's latency. This makes the algorithm effective in enforcing the third constraint of the above list, too.

This algorithm's application is not limited to cluster enumeration for EGRAs. In fact, because cluster enumeration is very similar to enumeration of valid configurations of a CCA [7], it can be used as a conservative check for CCAs too in order to make the enumeration phase cheaper without sacrificing its optimality.

3.3.2 *Cluster scheduling*

The purpose of scheduling is to ascertain the validity of the cluster and inserting pass-through operations for values and/or flags, whenever they are needed. This process is closely related to retiming, because the way operations are scheduled influences the critical path delay of the RAC, and consequently the latency of the computation on the EGRA.

As shown in Figure 6c, scheduling uses the bitmask as hints to prune the search. The simplest possible scheduling algorithm uses the MSB of the bitmask to allocate each node to a RAC row; this corresponds to as-late-as-possible scheduling (the strategy used in Figure 6c). In order to perform ASAP scheduling, instead, one would visit the cluster in topological order, and pick the least significant bit that a) is set, and b) is placed on a row below all the predecessors. Exhaustive search can also be performed by adding backtracking to the ASAP strategy.

The number of elements needed on each row is computed by summing the number of computation nodes allocated to the row, and the pass-through nodes that are added between the rows. If a_j is the row on which node j is allocated, first of all s_j, the maximum row for all the successors of j, is computed. Then, a pass-through node is needed for rows in the range (a_j, s_j). This can also be expressed as a bitmask, by evaluating $(1 \ll s_j) - (1 \ll (a_j + 1))$.

3.4 Candidate retiming

The EGRA architecture may allow a sequence of cells to execute in the same cycle, as long as the total critical path delay is shorter than the cycle time. This allows creation of relatively complex combinational structures and improves the number of instructions per cycle. However, it also assigns to the compiler the additional task of computing the run-time delays of the EGRA, in order to optimally insert registers.

The delays computed by Synopsys Design Compiler (as in Figure 7b) are worst-case values for a particular RAC design. However, because programming is done in advance, the switching activity of some arithmetic or logic components will not affect the outputs. If these components are on the critical path, a RAC's delay at run-time will be better than the value computed by Design Compiler.

For example, the adder is on the critical path of a full-featured ALU (supporting logic, add/subtract and shift/rotate operations). However, in the specific configuration of Figure 7c, we know that the switching activity on the first row's adders will not influence the outputs, and therefore that the adders' latency will not affect the critical path! Since the compiler knows how the RACs are programmed, it should be able to obtain a refined estimate for each RAC taking part in the computation.

To achieve this objective, we embedded in the compiler a simple model of the RAC's structure. This model splits the delays in *computation delays* (that depend on a node's opcode) and *multiplexer delays* (that only depend on the cell template). These delays can also be computed with Design Compiler, based on the delays for various pieces of the RAC datapath; they can then be tabulated as in Figure 7a and included in the compiler. In the example of Figure 7c, the configuration-specific delay is 4.5% better than the worst-case provided by Synopsys (Figure 7b).

After the candidate graph is partitioned into clusters, this model is applied to each cluster. After the delays are computed, the compiler does not need to know the interconnection within each cluster. Therefore, nodes in the same partition can be collapsed.

A retiming algorithm is then run on the candidate to insert registers between clusters, using a user-provided cycle time. Since data-flow graphs are acyclic, we can use a simple, linear time algorithm [8] to do so.

It is important to note that retiming must be run *after* partitioning, because registers cannot be inserted in the middle of a cluster (i.e., in the middle of a RAC): all cells participating to the computation must execute in less than a single cycle, and their execution must lie within a single cycle. This is easily achieved by running retiming after the

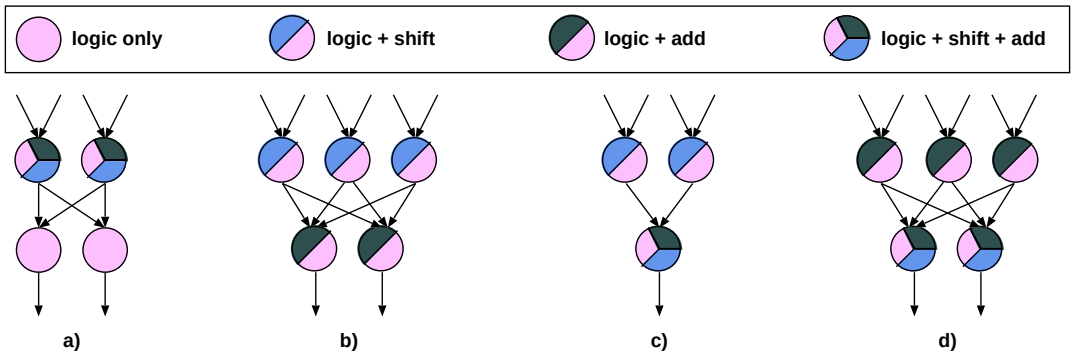


Figure 8: RAC design of the maximum-speedup Pareto point configuration, for a) rawdaudio; b) rawcaudio; c) crypto benchmarks (des, sha); d) all four benchmarks.

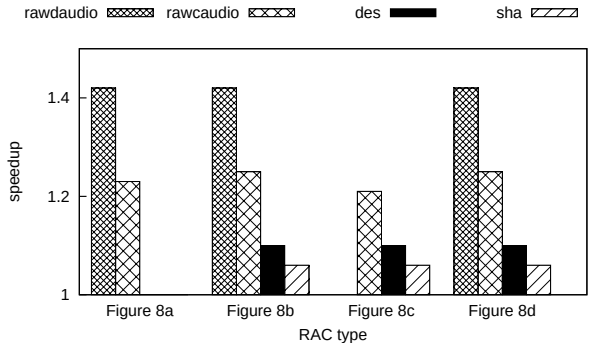


Figure 9: Speedups obtained by four RAC configurations

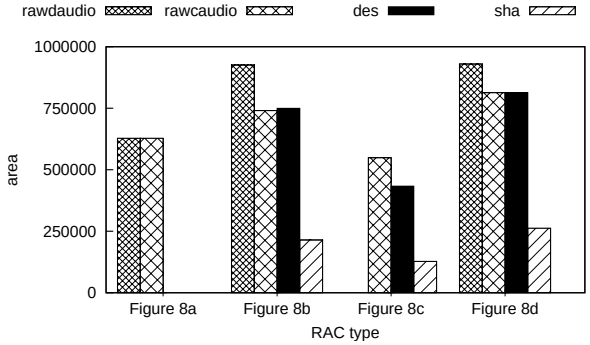


Figure 10: Area needed for the best speedup (in μm^2) on four RAC configurations

clusters have been collapsed to a single node.

4. EXPERIMENTAL RESULTS

In order to assess the computational feasibility of the mapping algorithms presented in section 3, we used them to analyze benchmark performance over a wide range of RAC designs.

The algorithms have two kinds of input. First, benchmark DFGs are used to choose candidates and are later partitioned into RACs; in our experimental platform, the DFGs for each benchmark are extracted with a GCC-based compiler front-end. In addition, the retiming phase needs to know the latency characteristics for the components on a RAC datapath, and uses them to estimate the latency of the programmed RAC configurations (see Section 3.4). These values are the same for all input programs, and were estimated using Synopsys Design Compiler and TSMC 90nm front-end libraries.

We ran our compiler on four MiBench [9] benchmarks, testing 872 different configurations of the RAC for each benchmark; these configurations have a depth varying from one to three rows, and the biggest one has 5 ALUs on the first row, 4 ALUs on the second, and 2 on the third. On a PowerPC G4 processor running at 1.5 GHz, running this procedure took approximately 15 minutes to explore the design space on one program; on average, this corresponds to a run time of one second for each RAC configuration.

We did not observe substantial differences in the run-time for different benchmarks; also, for all benchmarks most of the time was spent performing maximal subgraph enumeration. For bigger benchmarks, it would be beneficial to run it separately from the rest of the flow, because its output only depends on the capabilities of the *most powerful* ALU in the architecture. For the EGRA design, in particular, four runs of the maximal subgraph enumeration algorithm are sufficient to test any number of configurations.

The enumeration algorithm described in section 3.3.1 is extremely fast. By incorporating most of the feasibility checks, it removes the need for a separate checking step (such as *subgraph isomorphism* in [7]), and uses the information to prune the search space effectively. The time needed to perform enumeration was below 1 millisecond in all executions of the algorithm. An important feature of our algorithm is that this time grows very slowly with the size of the basic block and with the range of operations supported by the RAC; it only grows substantially if the depth of the RAC is increased. This however is not a problem because RAC depth is also limited by factors such as clock speed.

We also present estimated speedup results for the four benchmarks. We derived four "optimal" RAC designs from the result of the exploration. These are drawn in Figure 8. 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 8abc would use more area than Figure 8d, without improving performance.

Estimated clock cycle savings and area occupation for the four configurations are found in Figures 9 and 10. Note that not all RACs of Figure 8 will provide speedup on all benchmarks. For some benchmarks, indeed, the designs of Figures 8a and 8c are not well suited; correspondingly, profitable ISEs are not found by the compiler and the EGRA is simply not used. These cases correspond to points with a speedup of 1 and an area of 0 in Figures 9 and 10).

It is interesting to note that all four optimal designs use two rows. This is the same depth used for the data path in [10], but much smaller than the examples of *Configurable Computation Accelerator* presented by Clark [2]—despite the apparent similarity between the design of the RAC and the CCA. The reason is that RACs can be connected to form combinational structures. This feature puts smaller cells to an advantage, since they will usually have higher utilization rates without sacrificing speed.

5. TOWARDS A GENERAL FRAMEWORK

The complexity of designing a compiler for a coarse-grained reconfigurable array has been an obstacle to research on this kind of accelerator hardware. Architectures that were studied present a huge variability, and this makes it hard to develop well-founded building blocks, that compilers could use to analyze programs and retarget them on a CGRA. In turn, the major differences between compilers for different kinds of processing elements and interconnects make it very hard to reproduce published results and to compare architectures.

We believe that the collection of mapping algorithms that we described in Section 3 can be generalized to a framework for CGRA compilation. Of course this generic scheme will need tuning to target all the peculiarities of coarse-grained architectures. On the other hand, studying the possible specializations of this framework will also allow development, evaluation and comparison of different schemes for mapping on the same architecture.

The four steps proposed in this paper's framework are *candidate enumeration*, *technology mapping*, *partitioning* and *place and route*.

Candidate enumeration. The first step for the compiler is to find a set of candidate data-flow subgraphs—see Figure 3a. As explained in Section 3.1, these are the minimal unit that will possibly be chosen for execution on the accelerator. While we used an algorithm from the field of customizable processors [3], in the simplest case the candidates may be hot inner loops whose execution can entirely be moved to the CGRA. While this will be possible only if the array includes local memory elements, or can access the system bus, this simple case is not so uncommon. In particular, if the compiler is to perform software pipelining (including modulo scheduling) [11], all possible candidates are of this kind.

Technology mapping. The technology mapping step takes data-flow graphs, expressed in the compiler's intermediate representation, and lowers them to the exact set of operations that are supported by the target architecture. The implementation of this step is obviously different for every target, but all do share a common gist. In particular, the presented technique based on data-flow graph transformations can be adapted easily.

Partitioning. After technology mapping, the operations in the candidate must be partitioned into different cells of the array (see Figure 3c). This expresses the given data-flow graph in terms of cells (CGRA processing elements). Because of the variety of CGRA designs, the shape of the clusters may be very different depending on the architecture. In the case of a VLIW architecture with n instruction per bundle, for example, the clusters will be convex subgraphs consisting of up to n disconnected nodes.

Inter-cell scheduling. Scheduling the flow of data between the cells that compose the array is the final step in compilation for a coarse-grained architecture. Depending on the interconnect between the cells, this problem can pose very different challenges. For some technologies, inter-cell scheduling is more akin to modulo scheduling [11]. In other architectures, this phase should also allocate shared buses and multipliers; in this case, it can be solved with a combination of heuristics and ILP [12].

6. RELATED WORK

As we mentioned in the introduction, the challenge of mapping applications onto coarse-grained reconfigurable arrays has not been tackled systematically so far. Most of the previous work concentrated on particular architectures, and the potential for generalization was actually pointed out rarely, even if this was the case.

For example, DRESC [11] is a compiler for coarse-grained reconfigurable architectures that performs modulo scheduling and place-and-route for topologies with an arbitrary interconnect. DRESC works on target architectures that use VLIW functional units, and is not easily adaptable to other kinds of processing element. Of particular interest is the representation of the target architecture as a *routing graph* modeling elements such as functional units, register files, multiplexers and buses.

On the other hand, their technique is limited to homogeneous coarse-grained reconfigurable architectures, and the retargetability of the intermediate representation is also not clear. To avoid the latter problem, we propose a specific step in the compiler flow whose purpose is to convert computations to a reduced set of operations supported by the target architecture.

Yoon and Ahn [12, 13] studied different approaches for mapping applications onto possibly heterogeneous CGRAs. Besides providing an ILP formulation of CGRA place-and-route, they propose two efficient algorithms for the same problem: one first groups elements into columns, and then lays out the nodes on the grid; the other uses techniques from planar graph drawing. These strategies are very flexible and can use cells as routing elements—the former [12] only in special cases, while the latter in much more flexible ways [13].

Guo [14] presents an algorithm to extract templates and find them in an application. This technique is not necessarily related to coarse-grained reconfigurable systems, and is subsumed by the *cluster enumeration* step of our technique.

One of the reasons for the lack of a comprehensive base for CGRA compilation is the high variability between architectures, not only in terms of processing element functionality,

but also for the level of integration between the reconfigurable fabric and the rest of the architecture. A common implementation is that of a coprocessor, used for example in Morphosys [15]; other works couple the CGRA and the processor more tightly, as is the case for ADRES [16], the architecture targeted by DRESC. The case study we present uses the reconfigurable fabric in order to implement instruction-set extensions, and as such is closer to the latter family of architectures.

Great differences exist also in the kind of processing element used in the array, and in the topology. Regarding the topology, the choice ranges from the 1D topology of PipeRench to the powerful 8-way interconnection (plus one shared bus for each row and column of the array) of MorphoSys cells [17]. One peculiarity of the MorphoSys is a SIMD machine where all the cells in a row or column share the same *context*. This makes its architecture very complex to program with a compiler; our study is more suitable to MIMD architectures.

Regarding the PEs, there are several notable example of complex cells, i.e. cells containing more than one ALU, in the state-of-the-art. In particular, the PACT-XPP cells [18] (like the RAC we overview in Section 2) support conditional operations within one clock cycle, and the *Flexible Computational Component* [10], while targeted more specifically to DSP kernels, is similar to the RAC in size and set of allowed operations.

Other accelerators that have been proposed are similar to CGRA processing elements, but with only one occurrence of them existing in each processor. The EGRA's processing element is inspired by a stand-alone accelerator, the CCA, that was proposed in [2]. Even though the replication of the structure had important consequences on the design (for example, the ability to build combinational functions from multiple processing elements) some of the techniques presented in Section 3 could indeed be applied also to Clark's accelerator architecture.

7. CONCLUSION

In this paper, we have proposed a set of techniques that can be used for mapping applications to a particular coarse-grained reconfigurable architecture, the EGRA. These include a novel algorithm for subgraph enumeration, that is also suitable for other accelerators such as Clark's Configurable Computation Accelerator.

We also overviewed how the scheme of our EGRA compilation flow can be generalized into a framework that can be adapted to other accelerators. This generalization could fruitfully lead to the development of a framework for CGRA compilation, reducing the amount of work needed to retarget a compiler and making it possible to perform stronger and more reproducible comparisons between architectures.

8. REFERENCES

[1] G. Ansaloni, P. Bonzini, and L. Pozzi, "Design and architectural exploration of expression-grained reconfigurable arrays," in *Proceedings of the 6th Symposium on Application Specific Processors*, Anaheim, CA, June 2008.

[2] 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*. Washington, DC, USA: IEEE Computer Society, Dec. 2004, pp. 30–40.

[3] 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*, Salzburg, Austria, Oct. 2007, pp. 125–134.

[4] C. Morra, J. a. M. P. Cardoso, and J. Becker, "Using rewriting logic to match patterns of instructions from a compiler intermediate form to coarse-grained processing elements," in *Proceedings of the 2007 IEEE International Parallel and Distributed Processing Symposium, 2007*, Mar. 2007, pp. 1–8.

[5] R. Geiß, G. V. Batz, D. Grund, S. Hack, and A. Szalkowski, "GrGen: A fast SPO-based graph rewriting tool," in *Proceedings of the 3rd Internatial Conference on Graph Transformations*, Natal, Brazil, Sept. 2006.

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

[7] N. Clark, A. Hormati, S. Mahlke, and S. Yehia, "Scalable subgraph mapping for acyclic computation accelerators," in *Proceedings of the International Conference on Compilers, Architectures, and Synthesis for Embedded Systems*, Seoul, South Korea, Oct. 2006, pp. 147–157.

[8] 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*, vol. 17, no. 12, pp. 1319–25, Dec. 1998.

[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*, Dec. 2001, pp. 3–14. [Online]. Available: http://www.eecs.umich.edu/mibench/Publications/MiBench.pdf

[10] 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*, vol. 25, no. 6, pp. 1154–1162, June 2006.

[11] 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*, Dec. 2002, pp. 166–173.

[12] M. Ahn, J. Yoon, Y. Paek, Y. Kim, M. Kiemb, and K. Choi, "A spatial mapping algorithm for heterogeneous coarse-grained reconfigurable architectures," in *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition*. European Design and Automation Association 3001 Leuven, Belgium, Belgium, Mar. 2006, pp. 363–368.

[13] 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 *Proceedings of the Asia and South Pacific Design Automation Conference*, Seoul, Korea, Jan. 2008.

[14] Y. Guo, G. J. Smit, H. Broersma, and P. M. Heysters, "A graph covering algorithm for a coarse-grain reconfigurable system," in *Proceedings of the 2003 ACM Conference on Languages, Compilers, and Tools for Embedded Systems*. New York, NY, USA: ACM, July 2003, pp. 199–208.

[15] H. Singh, L. Ming-Hau, L. Guangming, F. J. Kurdahi, N. Bagherzadeh, and E. M. Chaves Filho, "Morphosys: An integrated reconfigurable system for data-parallel computation-intensive applications," *IEEE Transactions on Computers*, vol. 49, no. 5, pp. 465–481, May 2000.

[16] 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*, ser. Lecture Notes in Computer Science. Berlin: Springer, June 2007, vol. 4419, pp. 1–13.

[17] 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," *Journal of VLSI Signal Processing Systems*, vol. 24, no. 2–3, pp. 147–164, Mar. 2000.

[18] PACT XPP Technologies, Inc., "XPP-III processor overview," 2006. [Online]. Available: http://www.pactxpp.com/main/download/XPP-III_overview_WP.pdf