Heterogeneous Coarse-Grained Processing Elements: a Template Architecture for Embedded Processing Acceleration

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

Abstract

Reconfigurable Architectures are good candidates for application accelerators that cannot be set in stone at production time. FPGAs however, often suffer from the area and performance penalty intrinsic in gate-level reconfigurability. To reduce this overhead, coarse-grained reconfigurable arrays (CGRAs) are reconfigurable at the ALU level, but a successful design needs more than computational power—the main bottleneck usually being memory transfers. Just like the integration of hardwired multiplier and memory blocks enabled FPGAs to efficiently implement digital signal processing applications, in this paper we study a customizable architecture template based on heterogeneous processing elements (multipliers, ALU clusters and memories) that provides enough flexibility to realize fast pipelined implementations of various loop kernels on a CGRA.

1. Introduction

The push toward better performance and higher power efficiency, and the ever-increasing cost of ASIC manufacturing, have highlighted the attractiveness of reconfigurable architectures for applications that benefit from hardware spatial execution and need the flexibility of software implementations. Therefore, the applicability of FPGAs has advanced from glue-logic and custom peripherals, to the implementation of Systems on a Programmable Chip (SoPC) that embed computational cores on reconfigurable fabric.

This achievement has been made possible by technology scaling that allowed to integrate, on a single die, a huge sea of logic elements with hardwired multipliers and memory blocks. Thanks to these special functional units, today's FPGAs can efficiently implement digital signal processing applications [19]. Most other data-intensive applications, however, do not achieve the same satisfying performance results when mapped in traditional, fine-grained reconfigurable architectures, as the bit-level reconfigurability of FPGAs' logic elements comes with a high area and speed penalty.

For this reason, a different class of architectures (*Coarse-grained reconfigurable arrays, CGRAs*) has been proposed [12]; these architectures rely on coarser-grained cells, usually in the order of an ALU [17], as logic elements, thus reducing the area and delay overhead intrinsic in reconfigurability.

Coarse-grained reconfigurable arrays can be integrated with a processor, treating them as accelerators that can implement complex applications-specific functional units. In this way, the CGRA can speed up the execution of frequently occurring pieces of code. This approach is particularly attractive in the embedded systems domain, as software spends most of the time in relatively small code sections; however, memory transfers between the array and the host processor are a bottleneck unless the accelerator is given its own fast internal memory storage [7].

Also because of the small memory footprint of kernels in most embedded applications, including local memory elements into custom functional units has been shown to be beneficial by previous studies on automatic identification of instruction-set extensions [3]. In this paper we investigate the benefits of integrating storage and multiplication units into an existing coarse-grained architecture template, the *EGRA* [2] (Expression Grain Reconfigurable Array). The accelerator proposed here exhibits a very high computation density by achieving high speedups (up to 22x) with a very limited area (up to $1.2\mathrm{mm}^2$).

If the CGRA is able to execute all the operations in a loop (memory loads, arithmetic calculations, and memory stores), it is possible to go beyond the customizable processor model, and to offload execution control to the accelerator in order to aggressively pipeline the loop [14]. Therefore, we also outline the design of a control unit that efficiently supports the execution of modulo-scheduled loops.

In order to get insights on the tradeoffs involved in heterogenous, embedded memory CGRA design, we concentrated our attention not on a particular architecture, but on a *template*—the EGRA [2]—from which a number of archi-

tectural instances were derived, either generic or tailored to implement the loop kernels of a particular benchmark. As detailed in the following sections, these case studies hint that combining scratchpad memories with arithmetic clusters and multipliers offers the flexibility needed to implement a wide variety of kernels. This work can then be seen also as an explorative step towards the design of an efficient, compact accelerator, that can be reconfigured to suit a wide range of applications.

The remainder of the paper is structured as follows. Section 2 describes related work on coarse-grained reconfigurable architectures. Section 3 describes the architectural template structure and features, detailing the *machine description* parameters used to obtain different architecture instances. Section 4 describes the kernels used to investigate the proposed approach, the array instances used to map the kernels and the performance numbers obtained from synthesis (area and critical path) and from kernel mapping (speedup and parallelism). Finally, Section 5 concludes the paper and presents possible future work.

2. Related work

Reconfigurable accelerators, coupling software-like programmability and hardware-like spatial execution abilities, have been proposed as a valid platform for SoC development. However, implementing an entire system on reconfigurable hardware may have disappointing performance and power consumption. For this reason, FPGA vendors have proposed boards where reconfigurable logic coexists with hardwired processors [19].

These advances, however, are often insufficient to accelerate applications substantially, especially in the absence of a fast connection between the host processor and memory. Hence, most available customizable processors still target ASICs [10, 18] or are provided as soft cores for less performance-demanding applications [1].

Attempts to overcome these difficulties resulted in a series of proposals for *coarse-grained* reconfigurable architectures, that sacrifice bit-level reconfigurability of FPGAs to achieve satisfying performance. A common trait of these architectures, which usually are designed as coprocessors, is the presence of local storage connected to the datapath by a high-bandwidth bus. This memory component can be a small general-purpose scratchpad, as in DREAM [7] and in our proposed architecture, or a buffer interface sitting between the computational cells and the system RAM, as in MorphoSys [17].

Our proposal differs from DREAM in that scratchpad memory is scattered around the array, in the form of specialized memory cells. By splitting different arrays on different cells, we limit the I/O requirements (number of ports) of the memories. Furthermore, we can reuse the array interconnect as a data/address bus for memories, removing the need for a separate connection between the computational and storage elements of the array.

Two notable coarse grained architectures that embed RAM within the array are MORA (a homogeneous array in which each cell is composed of a memory *and* an ALU [13]), and PACT-XPP (a heterogeneous architecture where one of the cell templates also includes both memory and an ALU [15]). Our design differs from the above-mentioned ones by decoupling storage and computation, assigning these functionalities to different cells; this choice leads to a more flexible design space.

The design of CGRAs cells varies greatly in the architectures proposed in literature. In some of them, such as ADRES or RAW, cells are close to being general-purpose processors or DSPs. ADRES, for example, has a VLIW instruction set with predicated execution. More commonly, however, cells are purely arithmetic components, possibly including up to 4–6 ALUs or multiply-accumulate units [8]. The homogeneous EGRA [2] belongs to the latter family. In this paper, we augment the EGRA model with heterogeneous processing elements, enabling accelerated execution of a wider range of applications.

The modular nature of CGRAs makes possible for whole families of parameterized architectures to be constructed from a template. Indeed in this paper we consider a family of architectures and, together with a generic accelerator design that applies equally well to different applications, we also study specific instances that are tailored to a particular benchmark. This approach is somehow similar to the exploration of the ADRES template undergone by Bouwens [6], even if the focus of our work is on the cells' structure, as opposed to mesh topology and interconnect.

Effective pipelining of the loop, enabling different parts of multiple iterations to execute simultaneously, is key to extract performance from CGRAs. The mapping technique we present is based on modulo scheduling [16], a software pipelining technique supporting resource constraints, such as the number of I/O ports on a scratchpad memory or the number of multipliers in the cell. The same technique is used for instance in DRESC [14], and can be applied to other arrays that support MIMD execution.

3. Architecture description

In this section we present an extension of the architecture introduced in [2], supporting heterogeneous cells. In addition to detailing the operation of the processing elements, we present a control unit that supports the execution of loops entirely in the array, and overview how the proposed architecture is interfaced with the host processor.

3.1. Cells architecture

The functional units in the array architecture constitute a mesh of cells of three different types: ALU clusters, mem-

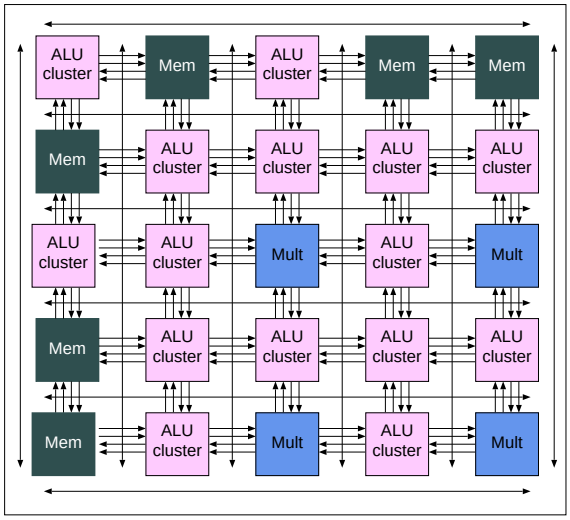

Figure 1. EGRA instance example

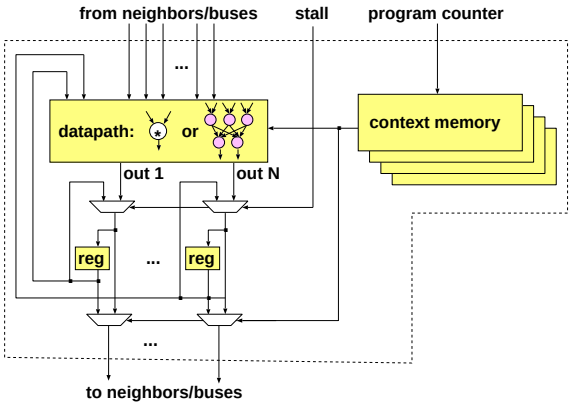

Figure 2. Block scheme of the interface of computational cells (fixed-function multipliers and clusters of ALUs). For memories, the *stall* signal is replaced by a *write-enable* signal.

ories and multipliers. The number and placement of cells for each type is part of the architecture parameter space. As such, it is decided at design time and can vary for different instances of the EGRA. Cells are connected using both nearest-neighbour connections and horizontal-vertical buses, with one such bus per column and row of the array.

The architecture of a cell can be roughly split in three parts: an I/O interface, identical for all cell types, a context memory, and a core, that implements the specific functionality of the cell.

The interface part takes care of the communications between cells, and between each cell and the control unit. It is in charge of connecting the datapath with the outside of the cell. As shown in Figure 2, it sends inputs to the datapath, provides values from the registers and the datapath to the neighbours, and places them on the row-column buses if requested.

The context memory is where cells store their configuration; the word size and format of the context memory varies with the type of cell. For example, multipliers have a fixed-function unit, and consequently the configuration word only has to specify the routing of data for the interface part.

Each cell can store more than one configuration word, so that the entire array can hold several *contexts*; the number of contexts is also part of the architectural specification that is given at design-time. Each context is composed of a configuration word per cell; the control unit can activate a different context on every clock cycle.

The rest of the cell is the actual implementation of the datapath and/or storage. For memory cores, this also include an address generation unit, so that multiple arrays can be mapped in the same memory cell in different areas; different data widths are supported (from 4- to 32-bit accesses) and are selected at reconfiguration time, while memory size is instead specified at design time. For cells that are clusters of ALUs, the arithmetic units in the cluster are organized in rows, with full connectivity between adjacent rows to enable efficient mapping of complex expression on the EGRA [2]. The design space of ALU-cluster cells includes the width and number of ALU rows, and which operations (logical, addition-subtraction, shift-rotation or any combination of these) will be supported by which rows; all ALUs can also act as multiplexers in order to implement conditional execution efficiently. The configuration word of these cells defines the operation executed and how data is routed through the switchboxes.

Non-memory cells also have a small local storage element in the form of output registers. The initial content of the registers can be set at reconfiguration time.

3.2. Control unit

The control unit is explicitly designed to support modulo scheduling [16], a software pipelining technique that increases inter-iteration parallelism and cell utilization by allowing different iterations of the loop to partially overlap.

Modulo-scheduled loops present a prologue part, an iterated steady state part and an epilogue part. The host processor communicates the desired number of iterations to the accelerator. The control unit's program counter is incremented on every clock cycle until it completes the prologue and the first iteration of the steady state, then it loops back at the start of the steady state (Figure 3).

On every clock cycle, one or more input or output values must be exchanged between the processing elements of the accelerator. To this end, several control signals are distributed to the cells at every clock cycle: the id of the context being executed, a per-cell bit to disable cells stuck in a pipeline bubble, a write-enable signal for memory cells. In

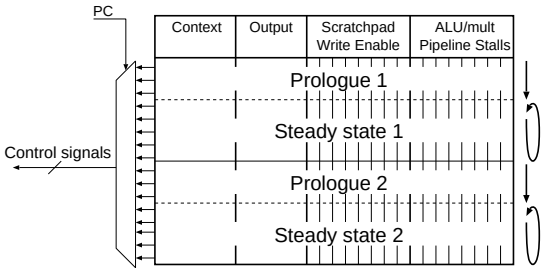

Figure 3. EGRA control unit

addition, each row in the control unit specifies one cell output that is connected to the processor data bus, so that scalar outputs can be communicated back to the host processor.

More than one loop can be programmed on the EGRA by storing control information in different regions of the control unit memory, and possibly using different contexts in the cells. To this end, the host processor communicates the range of rows containing the desired loop when starting the accelerator.

Few conditions must be verified for a loop to be mapped on the array, the most stringent of them is that the number of iterations must be known before the loop starts. This condition leads to a small and efficient control unit, and is usually met for kernels in the embedded systems domain.

3.3. EGRA operation

The EGRA can be set in two operational modes: DMA mode and execution mode. DMA mode is used to transfer data in bursts to the EGRA, and is used both to program the cells (including the output registers) and to read/write from scratchpad memories. These operations take advantage of the full bandwith available between the EGRA and the host processor, resulting in a limited overhead compared to execution time. Scratchpad memory transfers can happen either around a loop, or at program initialization if the scratchpads are used to store read-only look-up tables.

In execution mode, the control unit orchestrates the data flow between the cells as explained earlier in this section. It is possible to interface the EGRA with an extensible host processor, such as the Tensilica Xtensa, using custom instructions. Most such processors support variable-latency custom instructions; in this case, after execution mode is triggered by invoking a special instruction on the host processor, the host processor can stall until the loop is completed and the EGRA asserts a ”done” signal. By embedding input and output vectors entirely in scratchpad memories, the whole kernel can be run with a single special instruction, possibly surrounded by DMA transfers.

4. Experimental results

To analyze the performance of the architecture template we presented in Section 3, we implemented and validated

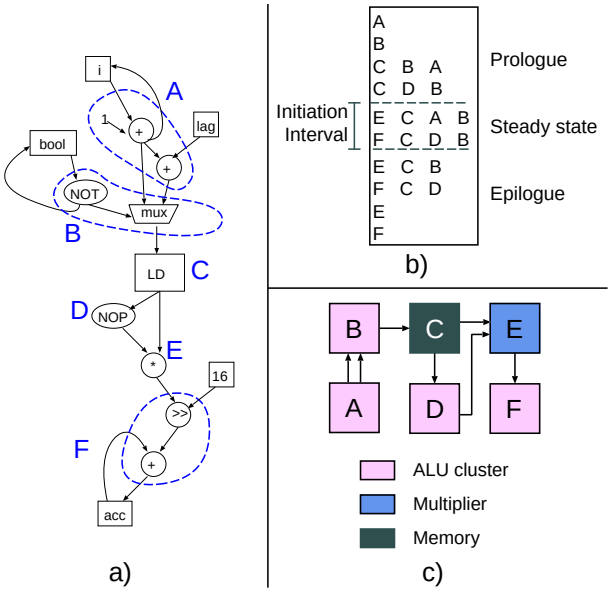

Figure 4. autcor loop kernel DFG: a) clustering; b) scheduling; c) place and route

a design for the cells and control unit. We then explored the performance of various EGRA instances using Synopsys Design Compiler to map them on TSMC 90nm front-end cell libraries.

We used loop kernels from seven benchmarks: aifirf, autcor, fbital, viterb, fft from the EEMBC automotive and telecommunications suites [11], rawdaudio from MiBench [9], life from MIT bitwise benchmarks [4].

We choose loops simple enough to be scheduled by hand, and yet illustrative of applications from different fields and with different memory and computational requirements. For example, fft requires the highest number of multipliers, viterb makes good use of multiple memories, while rawdaudio and life include mostly arithmetic operations.

For each benchmark, a custom-tailored EGRA was designed; Table 1 shows the characteristics of each of these specific architectures: the number of cells in the array, their type (ALU clusters, multipliers or memories), the type of ALU cluster used, the total amount of memory present in the array, and the total area and critical path of the design. These architectures were instrumental in validating the EGRA model and to obtain initial indications on the EGRA capabilities.

The last row of Table 1 shows the characteristics of a *generic* EGRA, i.e., an array that was designed so that all benchmarks kernels analysed could be mapped onto it. Such array configuration is actually the one shown in Figure 1 and consists in a 5x5 mesh, with an area of 1.2mm^2. The difference in performance of generic and specific architectures is quantified and discussed later in this section.

Benchmark	mesh size	cells type[a]			ALU cluster type[b]		memory (bits)	area (μm^2)	crit. path(ns)
aifirf	3x3	6 C	1 MU	2 M	3-AL	2-SL	1 024 x 2	299 095	3.85
autcor	2x3	4 C	1 MU	1 M	3-ASL	2-AL	128	103 695	3.90
fbital	3x3	6 C		3 M	3-AL	2-ASL	4 096 x 3	565 420	1.79
fft	5x3	9 C	4 MU	2 M	2-AL	2-ASL	4 096 x 2	571 331	4.05
life	4x4	15 C		1 M	3-ASL	2-AL	4 096	387 351	1.93
rawdaudio	5x3	14 C		1 M	2-ASL	1-L	2 048	179 616	1.45
viterb	5x3	9 C		6 M	3-ASL	2-ASL	512 x 6	313 382	1.41
generic	5x5	15 C	4 MU	6 M	3-ASL	2-ASL	4 096 x 6	1 199 903	4.05

[a] C: ALU clusters; MU: multipliers; M: scratchpad memories

[b] ALUs in each row and supported operations. A: arithmetic; S: shift-rotate; L: bitwise logical. For example, 2-AL represent a row with 2 ALUs supporting arithmetic and logic operations, but no shifts/rotates.

Table 1. Characteristics of EGRAs optimized for different benchmarks

Benchmark	II	avg. active cells/cycle	avg. ops/cycle
rawdaudio	16	2.6	5.2
fft	3	5	8
life	8	8.25	16.25
fbital	3	3.33	5
autcor	2	4	5.5
aifirf	2	4.5	7.5
viterb	3	5.6	10

Table 2. Parallelism achieved by loop kernels on the EGRA

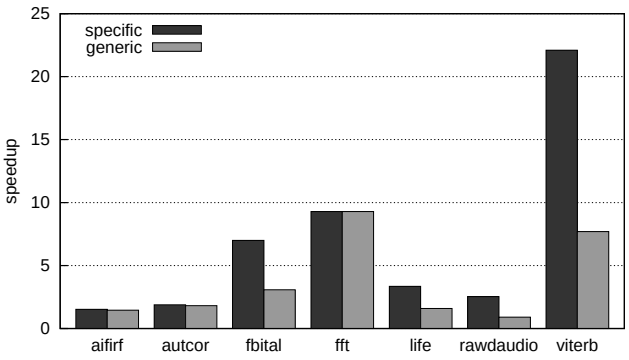

Figure 5. Speedup obtained on the EGRA

To map these kernels into specific and generic architectures, we extracted their most intensive loops; these represented 80% to 99% of the execution time of the whole application in the test cases considered in this paper. We then grouped the arithmetic-logic operations so that every group could be mapped onto one ALU cluster cell, and performed modulo scheduling (Figure 4). While performed manually in this study, algorithms exist for these steps to be automated [5, 14].

The effectiveness of the acceleration is heavily dependent on the degree of parallelism of the benchmarks. This in turn depends on the presence of dependencies in the loop, as well as on the distribution of the different operation types. The amount of parallelism obtained is shown by the average number of cells or operations that are active at any clock cycle. This data, and the initiation interval (II) obtained in each mapping, is summarized in Table 2.

In order to obtain speedups of EGRA-accelerated execution, as opposed to microprocessor-only, we ran each benchmarks through SimpleScalar/ARM, tuned to simulate an XScale processor with a 750 MHz clock[1]. We then compared the number of cycles needed to run the kernels on the XScale, with those needed on various EGRA architectures. Figure 5 shows the speedup obtained by running benchmarks on a processor powered either with the application-specific EGRAs of Table 1 (labeled *specific*) or with the generic one of Figure 1.

As it can be observed, speedups as high as 22x can be obtained on the application-specific architecture. Speedups on the generic architectures are lower for benchmarks that do not need multiplications because of the higher clock cycle. These results indicate that a pipelined multi-cycle multiplier could improve the speedups for this particular experimental setting. This however is not necessarily true if the clock period cannot be arbitrarily high, for example due to power considerations; in this case, the baseline processor's clock frequency will also decrease and expected speedups will be higher.

For three specific kernels, fbital, autcor and aifirf, we additionally compared the architectures of Table1 with limited architectures, in order to assess the advantage provided by the characteristics of the EGRA. In particular, we tried substituting ALU clusters with single-ALU cells, and avoiding the usage of memory cells. The former shows the advantages of evaluating whole expressions in a cell as opposed

[1] An XScale executes a multiplication in 3 cycles, and the latency of a multiplier in our design is approximately 4 ns, which correspond to 250 MHz.

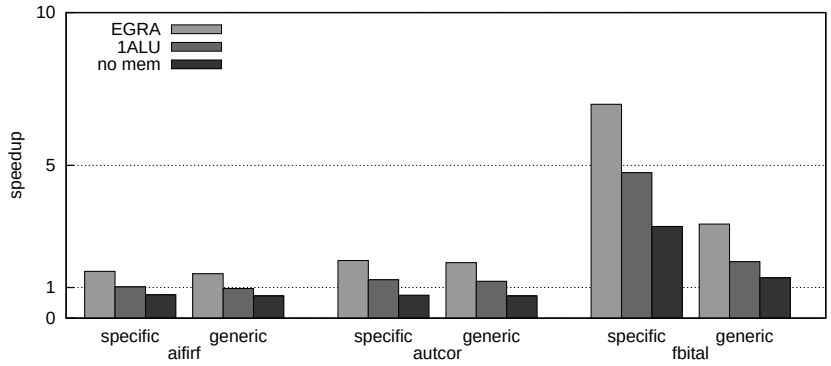

Figure 6. Comparison of speedups obtained by EGRAs with different characteristics

to single operations; the latter causes all communication to go through the host processor and allows to assess the benefit of embedding memory in the array.

The results, plotted in Figure 6, show that either ALU clusters or scratchpad memories are necessary to achieve speedups on most benchmarks. In the absence of those, the net result is a slowdown due to the higher critical path of the EGRA. On the other hand, the two features are substantially orthoghonal. Clustered cells have the advantage of allowing complex operations to be executed in a single clock cycle; complementarily, memory-capable EGRAs fare better compared to memory-less templates thanks to the parallelization of load/store operations on different scratchpads. Therefore, depending on the benchmark, one, the other or both can help improve the speedups obtained by the EGRA. This is true either when kernels are mapped to custom-tailored instances or to the generic one represented in Figure 1.

5. Conclusion

This work describes the architectural template of a heterogeneous coarse-grained reconfigurable array, supporting local storage of data and efficient pipelining of loops. Features such as the size of the mesh, the placement of cells of different kinds, and the complexity of each cell can be defined parametrically; this gives the designer the freedom to derive specialized architecture instances for higher performance, or to design a CGRA that works well across a wide spectrum of applications.

Experimental results over different test cases show that meshes composed by heterogeneous cells can accomodate computational kernels with different characteristics. Thanks to its support for efficient loop pipelining, the EGRA makes it possible to achieve notable parallelism and speedups.

The EGRA template can be further expanded to accomodate other kinds of fixed-function units, and to include layout considerations. For example, irregular mesh topologies could be devised automatically in the presence of different-size processing elements.

The evolution of coarse-grained architectures should not happen in isolation. Together with architectural advancements, research should also focus on developing the compiler infrastructure to allow automated extraction of accelerated kernels from high-level language code.

References

[1] Custom instructions for the Nios embedded processor. App. Note AN-188-1.1, Altera Corporation, San Jose, Calif., Apr. 2002.

[2] G. Ansaloni, P. Bonzini, and L. Pozzi. Design and architectural exploration of expression-grained reconfigurable arrays. In *Proc. of SASP 2008*, pages 26–33, Anaheim, CA, June 2008.

[3] P. Biswas, N. Dutt, P. Ienne, and L. Pozzi. Automatic identification of application-specific functional units with architecturally visible storage. In *Proc. of DATE 2006*, pages 212–217, Munich, Mar. 2006.

[4] Bitwise benchmarks. `www.cag.lcs.mit.edu/bitwise/bitwise_benchmarks.htm`.

[5] P. Bonzini and L. Pozzi. Recurrence-aware instruction set selection for extensible embedded processors. In *IEEE TVLSI*, Oct. 2008.

[6] 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.

[7] F. Campi, A. Deledda, M. Pizzotti, L. Ciccarelli, P. Rolandi, C. Mucci, A. Lodi, A. Vitkovski, and L. Vanzolini. A dynamically adaptive DSP for heterogeneous reconfigurable platforms. In *Proc. of DATE 2007*, pages 1–6, Apr. 2007.

[8] M. Galanis, G. Theodoridis, S. Tragoudas, and C. Goutis. A high-performance data path for synthesizing DSP kernels. *IEEE TCAD*, 25(6):1154–1162, June 2006.

[9] M. Guthaus, J. Ringenberg, D. Ernst, T. Austin, T. Mudge, and R. Brown. MiBench: A free, commercially representative embedded benchmark suite. In *Proc. of WWC 2001*, pages 3–14, Dec. 2001.

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

[11] T. R. Halfhill. EEMBC releases first benchmarks. *Microprocessor Report*, 1 May 2000.

[12] R. Hartenstein. A decade of reconfigurable computing: A visionary retrospective. In *Proc. of DATE 2001*, pages 642–649, Mar. 2001.

[13] M. Lanuzza, S. Perri, and P. Corsonello. MORA: A New Coarse-Grain Reconfigurable Array for High Throughput Multimedia Processing. In *Proc. of SAMOS-VII*, pages 159–168, Samos, Greece, July 2007.

[14] B. Mei, S. Vernalde, D. Verkest, H. De Man, and R. Lauwereins. DRESC: A retargetable compiler for coarse-grained reconfigurable architectures. In *Proc. of FPT 2002*, pages 166–173, Dec. 2002.

[15] PACT XPP Technologies, Inc. XPP-III processor overview, 2006.

[16] B. Ramakrishna Rau. Iterative Modulo Scheduling. *Int. Journal of Parallel Processing*, 24(1):2–64, Feb. 1996.

[17] 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 Trans. Comp.*, 49(5):465–481, May 2000.

[18] J. Turley. Tensilica CPU bends to designers' will. *Microprocessor Report*, 8 Mar. 1999.

[19] Xilinx website. `www.xilinx.com/`.