Dynamic Partial Reconfiguration of Customized Single-Row Accelerators

Nuno M. C. Paulino, João Canas Ferreira, *Senior Member, IEEE*, and João M. P. Cardoso, *Senior Member, IEEE*

***Abstract*— The use of specialized accelerator circuits is a feasible solution to address performance and energy issues in embedded systems. This paper extends a previous field-programmable gate array-based approach that automatically generates pipelined customized loop accelerators (CLAs) from runtime instruction traces. Despite efficient acceleration, the approach suffered from high area and resource requirements when offloading a large number of kernels from the target application. This paper addresses this by enhancing the CLA with dynamic partial reconfiguration (DPR) support. Each kernel to accelerate is implemented as a variant of a reconfigurable area of the CLA which hosts all functional units and configuration memory. Evaluation of the proposed system is performed on a Virtex-7 device. We show, for a set of 21 kernels, that when comparing two CLAs capable of accelerating the same subset of kernels, the one which benefits from DPR can be up to 4.3× smaller. Resorting to DPR allows for the implementation of CLAs which support numerous kernels without a significant decrease in operating frequency and does not affect the initiation intervals at which kernels are scheduled. Finally, the area required by a CLA instance can be further reduced by increasing the IIs of the scheduled kernels.**

***Index Terms*— Binary acceleration, coprocessor, dynamic partial reconfiguration (DPR), field-programmable gate array (FPGA), loop accelerator, modulo scheduling, very long instruction word (VLIW).**

I. Introduction

COMPLEX embedded applications contain both control-oriented and data-processing sections. Unlike applications executing on desktop or server machines, which benefit from powerful hardware, applications running on embedded devices may be limited to processors that favor power efficiency. Meeting performance demands under these conditions may be difficult, and resorting to a more powerful central processor may make energy consumption goals unreachable.

Manuscript received April 17, 2018; revised July 14, 2018; accepted August 29, 2018. Date of publication October 23, 2018; date of current version December 28, 2018. N. M. C. Paulino and J. M. P. Cardoso acknowledge the support of the Project "TEC4Growth—Pervasive Intelligence, Enhancers and Proofs of Concept With Industrial Impact/NORTE-01-0145-FEDER-000020," financed by the North Portugal Regional Operational Programme (NORTE 2020), under the PORTUGAL 2020 Partnership Agreement, and through the European Regional Development Fund (ERDF) *(Corresponding author: Nuno M. C. Paulino.)*

The authors are with the Faculty of Engineering, University of Porto, 4099-002 Porto, Portugal, and also with the Instituto de Engenharia de Sistemas e Computadores—Tecnologia e Ciência, 4200-465 Porto, Portugal (e-mail: nmcp@fe.up.pt; jcf@fe.up.pt; jmpc@acm.org).

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TVLSI.2018.2874079

An alternative is to resort to hardware/software codesign, building a system which benefits from specialized hardware in order to execute the most demanding portions of applications, thereby maximizing performance while minimizing energy consumption. However, developing HW/SW systems is generally a slow and error-prone process, which requires specific expertise. First, there is the detection and selection of critical regions of code. This may require lengthy profiling and may not be straightforward. Second, a software modification step is required to interface the application with the custom hardware peripherals to be designed. Finally, the selected kernels can benefit greatly from a fully custom hardware design, but hardware design is a slow and error-prone process, which requires specific expertise. Also, the interface between the host processor and accelerator circuits and the architecture of the latter have repercussions on performance, how the software needs to be modified, and on future revisions to both hardware and software.

To address this, considerable research exists on automated generation of specialized hardware targeting computationally intensive portions of applications, avoiding manual design, and/or system integration [1]–[6]. However, given that embedded systems are required to perform not one but a multitude of tasks, this hardware needs to either have some degree of programmability or several different tailored circuits are required. Regardless, as the number of tasks to execute increases, so does the amount of circuit logic. Designers must, therefore, also address the area efficiency of custom hardware via resource reutilization as much as possible [7], [8].

In the previous work, we addressed mostly the issue of custom hardware generation [9], with a customized loop accelerator (CLA) design based on a single row of functional units (FUs) whose interconnectivity was tailored to execute a set of loop traces extracted from a target application. In order to minimize the achievable initiation intervals (IIs), execution on the CLA is pipelined and resources are instantiated as needed by modulo scheduling. However, the higher the number of loops supported, the more likely it is that the required implementation area increases, that instantiated resources are underutilized, and that operating frequency decreases. That is, as reconfiguration capabilities increased, circuit specialization decreased.

The work presented in this paper aims to mitigate this by augmenting the same CLA architecture with dynamic partial reconfiguration (DPR). The CLA presented here is divided into a static area, containing logic which does not significantly

scale with the number of configurations supported, and a partially reconfigurable (PR), area containing logic which without DPR scales significantly and negatively impacts the design's performance and applicability.

For a set of N loop control and dataflow graphs (CDFGs), the approach enables the generation of N or fewer partial bitstreams to configure the PR region. That is, each circuit configuration of the CLA supports one or more loop CDFGs, as opposed to a single circuit supporting the whole set. Implementing one circuit per CDFG maximizes specialization and, therefore, clock frequency, and minimizes resource usage. However, it introduces the need for frequent DPR, increasing overhead. By choosing which CDFGs (from the target set) a single partial bitstream should implement, we can balance overhead and circuit specialization to find better solutions. However, this is not addressed in this paper. We present the maximum area savings and explain what is required to explore the design space in order to minimize overhead.

As our experimental evaluation demonstrates, resorting to DPR in this context has two main advantages relative to the implementation in [9]: 1) the total area required by the CLA is smaller by relying on DPR to switch between configurations, instead of using multiplexer-based logic and 2) we can avoid decreases in operating frequency, since circuit complexity is minimized. In addition, the motivation to implement this system also came from the notion that an architecture with DPR-based reconfiguration capabilities is an important step toward developing a system capable of runtime self-reconfiguration. This paper is organized as follows. Section II summarizes related work; Section III explains the general approach, as well as the system architecture and compilation flow; Section IV presents the CLA architecture, including which components are contained in the PR region and the respective advantages; Section V contains experimental results and discussion; and Section VI presents some final considerations.

II. Related Work

In [6], an approach that translates binary into a modulo scheduled configuration for a heterogeneous coarse grain reconfigurable array (CGRA) is presented. The approach targets inner loops detected by monitoring the instruction stream for backward branches. The scheduling itself is performed in software. The CGRA is tightly integrated with the host processor, exchanging data through the latter's register file. Up to two memory units, logic operations, and integer arithmetic are supported. Relative to an 8-issue very long instruction word (VLIW) processor, which requires 17 490 lookup tables (LUTs) on a Virtex-6 field programmable gate array (FPGA), a CGRA instance with 16 FUs and a crossbar type interconnect is 1.3× smaller and achieves a mean speedup of 2× for 11 test cases.

The ρ-VEX processor is a reconfigurable VLIW architecture, in the sense that it is configurable at instantiation time in terms of number and type of units, and issue width [5]. In addition, custom instructions can be added by manually writing the respective hardware description. The compilation tools are then capable of targeting these custom units. Using a Virtex-6 FPGA, an 4-issue ρ-VEX instance requires 3274 flip–flops (FFs), 19119 LUTs, and operates at 150MHz. For the six application kernels used, speedups range between 1.38× and 3.63× versus a MicroBlaze operating at the same frequency, and the average speedup is 2.72× versus a MicroBlaze operating at 200 MHz.

Like the work presented here, [4] CDFG merges into a single accelerator circuit. For the target CDFGs, the approach searches for the most common shortest subsequence of nodes. This sequence is then used to specify the architecture of a column of heterogeneous FUs which is repeated to create a 2-D array of homogeneous columns. However, in order to attempt to increase the flexibility of the generated circuit, additional hardware is then added to increase programmability so that future CDFGs can be mapped postproduction. For a target technology of 65 nm, the generated instances are about 2.2× larger than fully specialized accelerators.

The BERET approach, presented in [1], focuses on subgraph execution. A compile-time step detects large repeating instruction traces (i.e., loops), generates their CDFG representations, and extracts subgraphs which can be executed by BERET. The approach targets ARM processors, tightly coupling the BERET coprocessor to the processor's pipeline. BERET itself is composed of a set of subgraph execution units, each containing heterogeneous FUs. Memory access is possible via the shared L1 cache. The host processor is stalled while the targeted subgraphs are executed. When compared to an ARM1176, a single-issue in-order processor operating at 800 MHz, BERET achieves average energy savings of 35% for a set of 12 benchmarks, while requiring 5× less area than the ARM1176, for 65-nm technology.

Another system based on VLIW customization is presented in [10]. The approach implements a runtime translation of MIPS-based binary code into VLIW code during an initial translation pass. The free issue slots on the target VLIW are used to transparently instrument the code for profiling. Sequences of basic blocks with up to 256 instructions are then detected, and processed by a second translation pass which accelerates them by reimplementing the code to take advantage of loop pipelining. The binary translation and monitoring modules have been designed in *C* and implemented via high-level synthesis. For a target technology of 65 nm, an operating frequency of 250 MHz, and 10 kernels from the MediaBench suite [11], the loop pipelining step provides an average speed up of 3× over the VLIW code generated during the first pass. Performing the pipelining step with a dedicated hardware module is also 5.5× faster than a software implementation and 11× more energy efficient. The proposed system requires approximately twice the area of a standalone 4-issue VLIW.

In our own previous work [9], an offline trace detection method is used to extract MicroBlaze instruction sequences which represent frequent loop paths capable of crossing control flow boundaries [12]. A postcompilation flow generates a custom CLA instance capable of supporting all desired loop traces. Acceleration is achieved by modulo scheduling the loops, and resource requirements are minimized by tailoring the instance to contain only the required FUs and connectivity. For each accelerated loop, execution proceeds by executing the

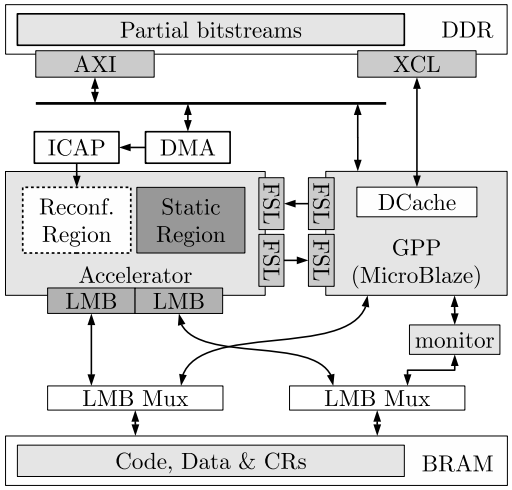


Fig. 1. System architecture overview.

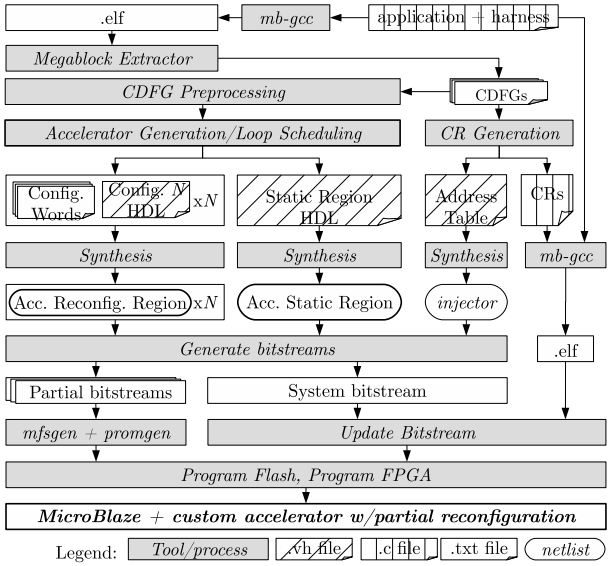


Fig. 2. Flow for generation of MicroBlaze-based system with CLA instance.

steps of the respective modulo schedule, which are contained in a local configuration memory. Thus, the CLA instance scales in size with the number of loops supported, leading to underutilization of resources, decrease in operating frequency, or even placement or routing failures. Despite this, an average geometric speedup of 3.61× was achieved for 24 CLAs supporting a between one and three loops, and the respective area was 1.11× that of the host MicroBlaze processor.

III. General Approach

The approach we present is based on the automated generation of a specialized hardware instance capable of executing selected portions of an application without software modification or manual hardware design. Avoiding modifications to application code alleviates development effort and eliminates the need to maintain different code versions of the same application when deploying onto several targets. We also avoid modifications to the binary that would prevent binary portability, so that the augmented binary generated by our flow is compatible with a nonaccelerated MicroBlaze-based system.

Our system relies on offline detection of critical portions of the application, in the form of megablocks [12]. These are single-entry multiple-exit instruction traces constructed by detecting repeating instruction sequences, i.e., binary trace loops. Each megablock can be expressed as a CDFG, from which latent instruction level parallelism and loop pipelining potential can be exploited by generating custom accelerator instances. In order to support the acceleration of many CDFGs without decreasing circuit specialization and increasing resource requirements prohibitively, and as a step toward a system capable of autonomous runtime self-adaptivity, the CLA is capable of DPR. Sections III-A to III-C describe the system architecture, the compilation flow which generates an instance of the system, and the CLA design.

A. System Architecture

The system architecture is shown in Fig. 1. It consists of a single MicroBlaze processor, one CLA instance, a local block RAM (BRAM) containing application code, data, and automatically generated communication routines (CRs) used to invoke the CLA. The accelerator makes use of both memory ports of the local data memory, which is shared with the MicroBlaze through the auxiliary *local memory bus Mux* modules. The processor and CLA communicate via fast simplex link point-to-point connections. The external memory is used only to store partial bitstreams. The partial reconfiguration of the CLA is performed via the FPGA's internal configuration access port (ICAP) primitive, which is fed by a direct memory access (DMA) module.

At runtime, the *injector* monitors the instruction address bus of the MicroBlaze. If an address corresponding to the start of one of the supported megablocks is detected, the injector replaces the fetched instruction with an unconditional branch to the address containing the respective CR, which was automatically generated and added to the application code. This routine sends operands from the register file of the processor to the CLA, calls a function which drives the DPR step (by configuring the DMA with the start address and amount of data to fetch), waits for CLA execution to complete, and returns to the address at which the injector intervened. Since megablocks may contain multiple exits, the last iteration is executed through software, so that the correct branch can be taken.

The execution of the application is thus accelerated, by pipelining loop iterations on the CLA and exploiting the latent instruction-level and data-parallelism present in the binary code.

B. Tool Flow

Fig. 2 shows the flow from target application to the accelerator-augmented system. The application binary (ELF file) is simulated and profiled in the megablock Extractor [13]. The resulting output is a set of all detected repeating instruction patterns and respective CDFGs. Currently, a manual selection step is used to discard runtime loops which correspond to undesirable portions of the application (e.g., *printf* routines

```
__attribute__ ((section (".CR_seg")))
unsigned int seg_0_opcodes[87] = {

    0xf821fffc, // save r2
    // (omitted) - save RF (32 instructions)
    0x3021ff80,
    0xb0000001,
        // setup "i" operand
    0xb9fcf3b0,
    0x20a00000,
        // call "_wrapCaller" function
    0x30210080,
    0xe821fffc, // retrieve r2
    // (omitted) - retrieve RF (32 instructions)
    // (omitted) - return to software
};

__attribute__ ((section(".pr_seg")))
void _wrapCaller(unsigned int i) {

    doPR(i);            // perform DPR if needed
    putfsl(1, 0);       // soft reset after DPR
    putfsl(1 << i, 0);  // set configuration number
    return;             // return to CR
}
```

Listing 1. Generated *C* source file with one CR and call to DPR function.

and startup subroutines) and to select frequently executed loops.

The CDFGs are then processed by the modulo scheduler. The outputs are HDL parameters for the static side of the CLA, and for all generated configurations of the reconfigurable area, each of which contains the FUs, connections and configuration words to execute one or more CDFGs at the target IIs.

The bitstream for the base system (i.e., all modules and peripherals, plus the static region of the accelerator) is generated first. Afterward, each synthesized version of the PR region imports the base system to generate all partial bitstreams. All bitstreams are placed into an MFS file system, which is a lightweight Xilinx proprietary file system, for which a *C* library is provided. Each partial bitstream file is generated as a binary-only file (i.e., with no header information), and written to the board's flash memory prior to programing the FPGA. The current implementation does not automatically allocate a specific area of the FPGA to use as the PR region. In the future, this could be done based on the largest resource requirements observed in the synthesis reports of each possible configuration (the PR region must be specified before the *map* stage).

For each CDFG supported a CR is also generated, and an address table configures the injector with each routine's position in memory. The CRs are placed into the program code by recompilation using a custom linker script, which also positions the functions related to DPR in specific locations. The code shown in Listing 1 is an excerpt of the output produced by the CR generation step. Each CDFG corresponds to a *_opcodes* array of 32 bit values which encode MicroBlaze instructions.

In the previous work [9], nearly all CR instructions were single-cycle reads/writes from and to the CLA. In the cases where no iterations were executed on the CLA despite its invocation, the contents of the MicroBlaze's register file and the stack would not suffer any modification upon return to software. If any iterations were performed on the CLA, the contents of the register file would match those produced by software-only execution, and the stack would not be altered in any way.

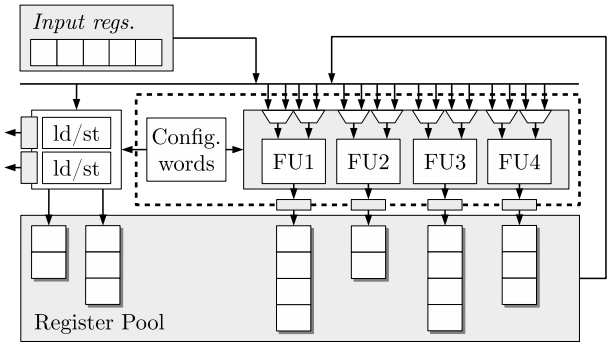


Fig. 3. Simplified architecture for one synthetic instance of the CLA (reconfigurable region delimited by dotted line).

However, the routines in this implementation include the call to the reconfiguration function, *doPR*, and to all child functions therein. As a result, the contents of the MicroBlaze's register file and stack do suffer unwanted modifications whether the CLA executes or not. As an initial approach, this implementation resorts to saving the entire register file to memory and recovering it after the partial reconfiguration function executes.

The *_wrapCaller* auxiliary function is placed at a specific address by the linker script. The function's *i* input argument is set during the CR and determines which partial bitstream will configure the CLA. The doPR function copies the partial bitstreams from flash to DDR (at startup) and drives the DMA engine which feeds the ICAP. When called, it checks if the CLA's reconfigurable area is already configured with the desired partial bitstream. If this is the case, the CR resumes normally. Otherwise, the reconfiguration routine writes the partial bitstream to the reconfigurable area of the CLA through the ICAP peripheral. The *C* function used to control the ICAP is custom written, to attempt to reduce overhead. By having the partial reconfiguration take place as part of the CR the transparency of the approach is not compromised.

IV. Partially Reconfigurable Customized Loop Accelerator

A simplified model of the accelerator is shown in Fig. 3. Its structure is very similar to the version presented in [9], with modifications to deal with the segmentation into static and reconfigurable portions. There is only one row of FUs, customized by the modulo scheduler based on the set of megablocks to accelerate. The CLA supports all 32-bit integer and single-precision floating-point arithmetic, including divisions by nonconstant dividers. All FUs are fully pipelined, with the exception of the nonconstant integer division unit. The *load/store* units are capable of performing byte-addressed operations to arbitrary memory locations since the accelerated traces also implement the address generation operations. When accelerating a megablock, the accelerator executes an arbitrary number of iterations each time it is called. The execution returns to software as a result of evaluating the respective termination conditions (i.e., *branch* instructions).

The modulo scheduler usually generates accelerator instances capable of executing the target CDFGs at their respective minimum II. If this is not possible, generally due to a large number of memory access operations, the II is increased until the CDFG is scheduled. This virtually does not occur, as the only resource limitation in this approach is the two memory ports (all other FUs are instantiated as needed).

A. Static Region

The static region contains the components whose resource requirements we predicted would not scale noticeably or at all with the number of supported CDFGs. This includes the input and the (omitted) output registers, which contain values exchanged directly with the register file of MicroBlaze. Since the data memory is a two-port BRAM shared between the CLA and the processor, all instances have only two memory access modules which are also placed into the static side. Although the number of registers in the register pool and the connections between them vary per instance, the pool was kept in the static region, because we estimated that the amount of resources it requires would not increase severely with the number of supported CDFGs; its size is largely determined by the CDFG which requires the most registers overall.

B. Dynamically Reconfigurable Region

The reconfigurable region contains all FUs required to execute one or more CDFGs, and also implements the connections between them, inserting multiplexers where necessary. Similarly, the configuration word memory contains only words for that CDFGs set. The PR region is implemented essentially in LUT, requiring nearly no FF.

In our implementation of the CLA without DPR support [9], FUs were reutilized as much as possible between supported configurations. However, one configuration may require a large number of FUs, which go unused by the remaining CDFGs. This has no effect on performance but leads to FUs being underutilized, and to greater resource requirements. Also, since more units exist, the width of the configuration word also increases, leading to a larger configuration memory. Finally, as more operations are scheduled onto an FU, multiplexer complexity increases. These two aspects contribute especially to higher resource requirements and longer synthesis times.

By placing these components in the reconfigurable region, the input multiplexers are only as wide as required by the chosen CDFG subset. Regarding FUs, the area required when implementing a set of CDFGs will be determined by the largest one of that set. This area is not necessarily as large as the area required by a multiconfiguration instance supporting all CDFGs. Also, by implementing each (or subsets of) CDFG(s) as a single configuration, the same resources can be used to implement different FUs between configurations.

The configuration memory was placed into this region because it was observed that for a large number of configurations, the memory size increased considerably. Even for a few configurations, the instruction word width (487 bits on average for the 24 benchmarks in [9]) led to a large number of LUTs being required to implement the memory as distributed RAM. By placing the configuration memory in the reconfigurable partition, it only needs to contain the words that implement the CDFGs of the respective configuration. This is an efficient reutilization of LUTs, since the same FPGA resources implement the memory which holds different configuration words, depending on which partial bitstream has been written to the reconfigurable area.

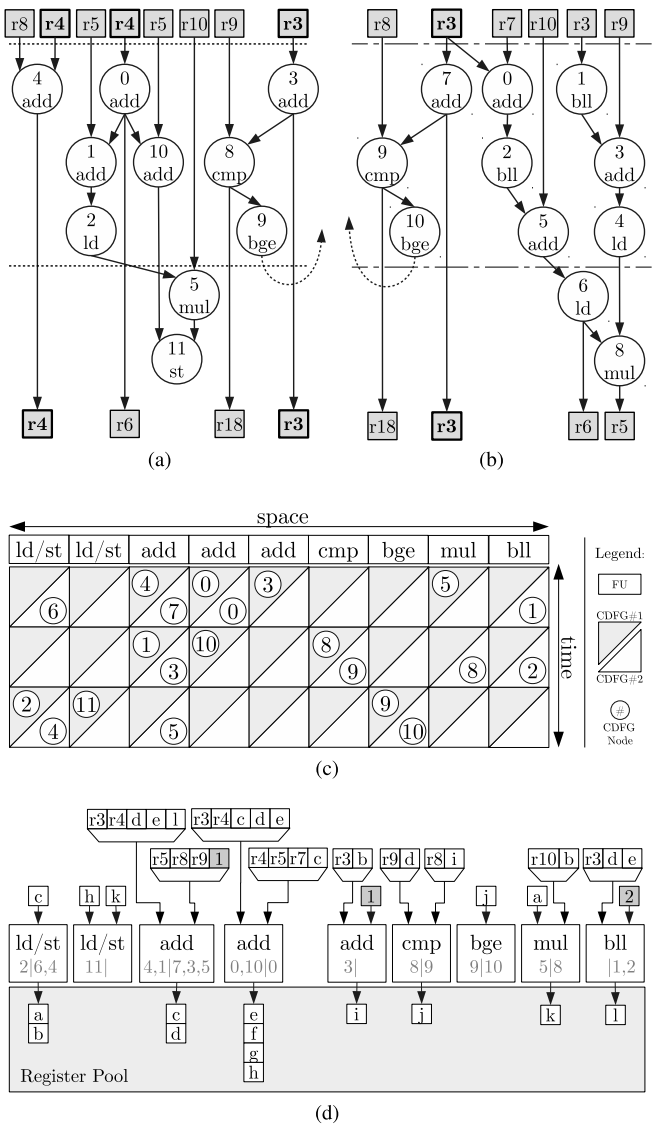


Fig. 4. Simplified example of the generation of a CLA instance capable of executing two CDFGs by reutilizing resources within the same circuit. (a) Example CDFG #1. (b) Example CDFG #2. (c) Combined modulo reservation table. (d) Respective CLA instance, showing multiplexers, register pool, instantiated FUs and which operations of both CDFGs are scheduled on each.

C. Example

Fig. 4 shows a CLA instance which supports the execution of two CDFGs within the same circuit instance, i.e., within the same partial bitstream, by reutilizing resources between configurations via multiplexer logic. The II of the two CDFGs is three, due to the dependence set by the exit node, *bge* (all exits must be evaluated before a new iteration can begin).

Backward connections are omitted but are implied via the input and output registers with bold outlines [e.g., *r3* in Fig. 4(a)]. In Fig. 4(c), the modulo reservation tables of both schedules are combined, showing the three timesteps required to execute all nodes, and demonstrating how resources are reutilized within the same loop and between loops. Fig. 4(d) shows the resulting instance. Inputs to FUs can be values from the input registers (e.g., *r3*), values from the register pool, or constants.

An increase in circuit complexity can already be seen, even for this small example. For instance, the first input of the leftmost *add* unit only requires inputs *r4* and *e* for CDFG #1, but a total width of five is generated. In addition, the rightmost adder is not utilized at all for CDFG #2. Finally, the instruction width is also affected, since it is determined by the number of FUs and multiplexer widths. Scheduling larger and/or more CDFGs exacerbates these effects, so we attempt to avoid these issues resorting to DPR. For this case, having each CDFG implemented as its own circuit would result in two independent reservation tables, and two distinct CLA instances, with reduced complexity relative to Fig. 4(d).

V. Experimental Results and Discussion

The main objective of our experimental evaluation was to determine what area savings were possible by resorting to DPR when deploying a CLA, while also evaluating other beneficial or negative consequences of this strategy.

Software Setup: We relied on a set of 23 kernels, taken from the *Livermore Loops* [14] set and from the TEXAS IMGLIB function library [15]. All kernels were compiled together with a software harness (used in [9]), generating a single binary for all test cases. The harness allowed us to specify which kernels to call, simplifying the evaluation of many different CLA instances targeting different kernels. To validate execution the harness compares CLA-computed results with reference data generated during compilation, generates pseudorandom input data for the kernels which are placed onto the heap, and also retrieves execution times.

Hardware Setup: The system architecture used is shown in Fig. 1. All program code (including the application, harness and partial reconfiguration functions) and data reside in local memory. At boot time, the partial bitstreams are copied from the nonvolatile flash to external memory. The ICAP module is accessed by the MicroBlaze via an AXI (Advanced Extensible Interface) bus. The target platform was a VC707 board, containing a Virtex-7 xc7vx485 FPGA. The tools used for synthesis and bitstream generation are from release 14.7 of Xilinx's ISE Design Suite, running on a machine with an Intel i7-6700K 4-GHz CPU and 32 GB of RAM. In all cases, the MicroBlaze contained a floating-point unit, integer multiplier and divider, barrel shifter, pattern-compare, and integer to/from float conversion operations.

Experimental Setup: With our set of kernels we generated a number of CLA instances, which we categorized as *small*, *medium*, or *large*, as shown in Table I. The eight small CLAs implement between 2 and 3 kernels, four medium CLAs implement between 4 and 6, and the two large instances implement 10 and 11 each. The kernels were grouped so that similar workloads are found within the same application. Each group (e.g., *s1*), was used to generate two CLAs tailored for the execution of the respective kernels. All systems targeted a frequency of 100 MHz, and all bitstream generation steps (e.g., *map*, *par*) were ran with an *effort* flag of *high*3 where applicable.

We first implemented each CLA without resorting to DPR. More specifically, we generated only one partial bitstream. These CLAs have, therefore, single-cycle reconfiguration times when switching between execution of their supported kernels. The DPR overhead is incurred only once when the CLA is first invoked. This case is roughly equivalent to non-DPR capable CLA, similar to the implementation presented in [9], for the purposes of our comparison. These instances are referred to with the prefix *single-*, e.g., *single-s1*. We then implemented the same CLAs, this time generating one partial bitstream per kernel supported, which we refer to with the prefix *multi*.

TABLE I
Kernel Information and Groups for CLA Test Implementations

kernel	# insts.	II	small	medium	big
quantize	10	3	s1	m1	b1
perimeter	21	3			
boundary	24	8			
conv3x3	64	10	s2		
sad16	13	2			
mad16	13	2			
sobel	40	5	s3	m2	
dilate	142	29			
erode	142	29			
innerprod	10	3	s4		
matmul	15	3			
hydro	17	3	s5	m3	b2
hydro2dimp	37	6			
innerprod_fp	10	4	s6		
matmul_fp	15	3			
intpredict	41	10	s7	m4	
diffpredict	44	10			
glinearrec	13	3			
cholesky	20	3	s8		
statefrag	42	5			
tridiag	12	3			

TABLE II
Areas Defined for PR Region of CLA Instances

Name	Slices	LUTs	FFs	DSPs	Circuit Area	Partial Bistream Size (kB)
sml1	882	3528	7056	36	1.06 %	162
sml2	1248	4992	9984	56	1.53 %	295
med1	1568	6272	12544	72	1.93 %	295
med2	2254	9016	18032	108	2.80 %	427
big1	3136	12544	25088	144	3.92 %	588
big2	5928	23712	47424	180	7.49 %	718
big3	9400	37600	75200	400	11.94 %	2573

For both cases, we chose the smallest possible area to implement each CLA. Table II shows the seven different area sizes we preconstructed for our experiment. The area for each case is manually selected, as the scheduling process does not generate a set of constraints that specify this area on a per-case basis. This is mostly because we find that predicting the resources required by the CLA's reconfigurable area via synthesis alone is inaccurate relative to the actual resources required after placement and routing, especially in terms of how many slices are required, due to how logic may be packed. Instead, we try to generate the same CLA instance (e.g., single-*s*1) several times, starting with the smallest possible area, until placement is possible.

Two auxiliary area groups were also used: the first constrains the placement of all static CLA logic to the perimeter of the reconfigurable region, and the second constrains BRAM placement so that it overlaps with the reconfigurable area of the CLA. The latter helps prevent long path delays between the CLA's memory access units and the BRAMs.

The following sections evaluate the results in terms of saved area, the total required bitstream generation time, bitstream storage space, and comment on the effects of both strategies on execution time and overhead. Unless stated otherwise, all kernels are scheduled at their minimum possible II for all cases.

A. Area Savings With DPR

1) Single Partial Bitstream Implementations: In Table III(a), the slice and LUT usage for the implementations resorting to a single partial configuration are shown, along with the chosen area size. Percentages shown are relative to the total number of available resources of the respective type in the chosen area. The average slice usage is 91%, and the average LUT usage is 79%. The number of FFs and digital signal processing unitss (DSPs) used is omitted as they are much lower for all cases, accounting for an average of 1.2% and 4.4%, respectively.

For the *small* set, despite the small number of supported kernels per case, we notice that the required area to implement the CLAs varies between *sml1* and *big1*. Referring back to Table I, we conclude that the main factor in determining the CLA size for these implementations, more so than the number of loop CDFGs supported, is the total number of CDFG nodes (i.e., MicroBlaze instructions). A greater number of nodes leads to more FUs and timesteps required to implement the schedules, which leads to more and wider configuration words, increasing the LUT usage since the configuration words are implemented as distributed memory. We also see this effect in single-*m*3, which requires a smaller area that the rest of the medium set, due to its smaller sum total CDFG nodes.

It is important to note that only the small set was generated successfully for an operating frequency of 100 MHz, which could not be achieved for the medium and large sets. This was due to long paths from the input multiplexers, through an FU (in some cases also the data memory), to the register pool. As a result, these two sets were generated targeting a frequency of 50 MHz. This, coupled with the low FF usage in the PR area, suggests that the register pool should not have been left on the static area. This was done since the amount of registers required in the pool does not scale as noticeably with the number of configurations as other aspects (e.g., multiplexer complexity) and was also an attempt at reducing partial bitstream size. However, due to the way resources are organized on FPGA, the end result is the underutilization of FFs in the PR area.

When attempting to generate these two sets for a target frequency of 100 MHz, the average achieved frequency possible is 91 MHz for set medium and 78 MHz for set *big*. This is only possible by increasing the size of the area used for single-*m*1, -*m*2, and -*m*3 to the next size, relative to the size indicated in Table III(a). In all cases, the critical path begins at either the register pool or instruction memory, passes through the FUs (often a 32-bit adder), ending at the multiplexers which drive the accelerator's output registers. Generating a 100-MHz system was only possible for single-*m*3, with area *big1*, since *med2* does not allow for routing, as mentioned, despite the adequate resource utilization for this area when targeting 50 MHz. It is possible that larger areas or careful floorplanning would improve the chances of achieving frequencies of 100 MHz for the remaining cases, but what we aim to demonstrate is that we may preserve both area and frequency through DPR.

Also, shown on the rightmost column of Table III(a) is the total time required to generate the system, in minutes. This accounts for synthesis time of the static side of the CLA, all of the variants of the reconfigurable region (in this case, only one), and the mapping, placing, and routing times.

2) Multiple Partial Bitstream Implementations: Table III(b) shows the same information for the implementations resorting to multiple partial bitstreams. Immediately, we see a decrease in the required area size in virtually all cases, most noticeably for the medium and large sets. For the small set, there are three cases where area is not reduced: *s3*, *s4*, and *s6*.

For *s4*, the smallest possible area was already being used. For *s6*, we see a reduction in the number of LUTs used, but a smaller reduction in the number of slices. In this case, both implemented kernels require floating-point FUs, which use FFs (unlike most other units). Therefore, despite a decrease in the configuration word memory size, each configuration still requires approximately the same number of FFs, which in this case is dictating the amount of slices. For *s3*, the reduction is simply insufficient to fit in a smaller area.

Note that area savings would still be possible for all cases, if the reserved area was individually specified. The potential area reduction can be inferred from the reduction in slice usage since there is approximately one slice per unit area of the device. Given this, the average potential area reduction would be of 1.42×, 1.61×, and 3.08×, for each set, respectively.

The required implementation time is shown in the rightmost column for comparison with Table IV(a). Each variant of the CLA's reconfigurable area requires mapping, placing, routing, and bitstream generation steps. Thus, the required time increases with the number of configurations. The increases are of 1.48×, 1.93×, and 2.43×, for each set, respectively. For the *big* set, the resulting area has diminished more so than the respective increase in required compilation time. Finally, consider that all systems are successfully implemented at

TABLE III

IMPLEMENTATIONS OF CUSTOMIZED LOOP ACCELERATORS SUPPORTING SEVERAL LOOP CDFGS RESORTING TO (a) A SINGLE PARTIAL BITSTREAM AND TO (b) MULTIPLE PARTIAL BITSTREAMS

(a)

	Area[1]	Slices	%	LUTs	%	Impl. Time (min)
s1	med1	1457	93%	4715	75%	33.1
s2	med1	1557	99%	5744	92%	41.7
s3	big1	3063	98%	11974	95%	48.3
s4	sml1	695	79%	2447	69%	41.5
s5	med1	1441	92%	4825	77%	37.4
s6	sml2	1015	81%	3340	67%	35.5
s7	med2	2084	92%	6724	75%	41.6
s8	med2	2026	90%	6313	70%	38.2
	average	**1667**	**91%**	**5760**	**78%**	**39.7**
m1	big1	2917	93%	9346	75%	42.8
m2	big1	3134	100%	12389	99%	49.2
m3	med1	1531	98%	5973	95%	40.3
m4	big1	3093	99%	11304	90%	46.0
	average	**2669**	**97%**	**9753**	**90%**	**44.6**
b1	big3	7934	84%	22458	60%	85.0
b2	big2	4856	82%	15251	64%	55.2
	average	**6395**	**83%**	**18855**	**62%**	**70.1**

[1]Area allocated for accelerator (see Table II)

(b)

	Area[1]	Slices	%	LUTs	%	Impl. Time (min)
s1	sml1	770	87%	2727	77%	60.7
s2	sml2	1042	83%	3398	68%	59.0
s3	big1	2467	79%	6651	53%	60.1
s4	sml1	607	69%	2034	58%	49.3
s5	sml2	1011	81%	3732	75%	51.7
s6	sml2	1004	80%	2731	55%	48.8
s7	sml2	1180	95%	4298	86%	63.1
s8	med1	1480	94%	5027	80%	63.1
	average	**1195**	**84%**	**3825**	**69%**	**57.0**
m1	med1	1417	90%	3891	62%	98.4
m2	med2	2052	91%	6715	74%	81.5
m3	sml2	1166	93%	3928	79%	73.6
m4	med2	2005	89%	6021	67%	102.8
	average	**1660**	**91%**	**5139**	**70%**	**89.1**
b1	med2	2102	93%	6396	71%	160.9
b2	med2	2043	91%	6414	71%	164.4
	average	**2073**	**92%**	**6405**	**71%**	**162.6**

[1]Area allocated for accelerator (see Table II)

100 MHz (the more demanding timing closure also contributes to longer implementation times).

3) Comparison With MicroBlaze: Table IV shows the average number of slices and LUTs required by the static and reconfigurable areas of the CLA instances, descriminated by the kernel groups, for both the single case and multicase. Values are normalized to the requirements of the MicroBlaze which were reported for each respective system. For reference, the MicroBlaze requires an average of 1260 slices, 1772 LUTs, and 2951 FFs. There is very little variation between instances. Comparing the reconfigurable areas between Table IV(a) and (b), we again observe the resource and area savings through the use of multiple partial bitstreams. Also, although the reconfigurable area of the CLA requires a much higher number of LUTs than the MicroBlaze, the respective FPGA area is much smaller. The placement restriction (absent for the MicroBlaze) is most likely responsible for this behavior.

B. Effects of DPR on Performance

The set of applications kernels used in this paper is the same set used in [9]. Therein, the CLA itself is further described, and focus is given to the scheduling process, the achievable II and number of instructions executed per clock cycle on a per case basis, and resulting speedups. The instances of the CLA presented so far were generated by scheduling the same kernels at their minimum possible IIs, meaning that the reported performance holds, despite the addition of DPR. The only penalty incurred is the initial DPR overhead. The time is required for each partial reconfiguration depends on the partial bitstream size. However, if DPR is performed often with small partial bitstreams, this may result in a greater total overhead, relative to less frequent reconfiguration with larger bitstreams.

The optimal solution (regarding minimization of overhead) is therefore application dependent. It is necessary to consider several aspects: 1) the number of instructions in each kernel; 2) how much of the application execution time the kernel represents; 3) whether existing resources generated when scheduling other kernels can be efficiently reutilized, or if, this not being the case, it is preferable to generate an entirely new variant for the reconfigurable area; and 4) the order in which kernels execute. The latter two aspects are not mutually exclusive, and the desired solution will vary by favoring either area or performance. In addition, the time required for DPR also depends on where in the device the area is located, as well as on the device itself. We do not study these aspects for the presented groups of kernels since both groups and the order in which they execute are artificial. They were designed to demonstrate the area saving potential.

C. Trading Performance for Further Area Savings

The fact that the minimum IIs were used for scheduling means that performance was favored over resource usage. However, reducing the amount of FUs or multiplexer complexity may be possible by increasing the II. In terms of FUs, the smallest possible set which supports a loop kernel is one that contains one and only one FU of each type required (e.g., no multiple addition units). It is not obvious, however, for purposes of area and resource usage prediction, how this translates into the complexity of the connectivity. For instance, with only a single addition FU, all addition operations are scheduled to it, meaning its input multiplexers may become wider, as opposed to more addition FUs with less (or a single) input choices. Increasing the II also tends to result in less wide configuration words but also in more time steps on the resulting schedule.

TABLE IV

AVERAGE SLICE AND LUT USAGE FOR STATIC AND RECONFIGURABLE AREAS OF THE CLA, NORMALIZED TO MICROBLAZE PROCESSOR FOR (a) SINGLE PARTIAL BITSTREAM AND (b) MULTIPLE PARTIAL BITSTREAMS

(a)

	Static Area			Reconfigurable Area		
	Slices	LUTs	FFs	Slices	LUTs	FFs
sml	0.40	0.16	0.87	1.32	3.25	0.05
med	0.77	0.19	1.55	2.01	5.50	0.07
big	1.56	0.45	3.03	4.81	10.64	0.09

(b)

	Static Area			Reconfigurable Area		
	Slices	LUTs	FFs	Slices	LUTs	FFs
sml	0.31	0.14	0.65	0.95	2.16	0.05
med	0.52	0.15	0.96	1.32	2.90	0.06
big	0.63	0.19	1.17	1.64	3.61	0.08

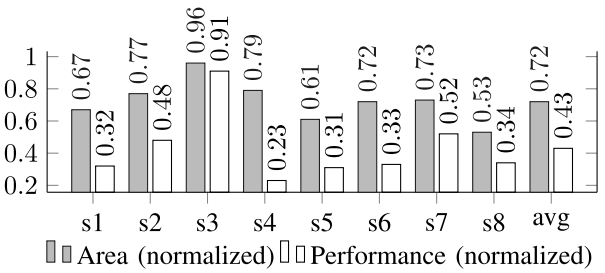


Fig. 5. Smallest area achieved per case, along with resulting performance decrease, normalized to the each case's implementation at its minimum II.

Therefore, although the impact on performance resulting from increasing the II might be easy to predict, the exact resource and area usage is not straightforward to predict analytically. Therefore, in order to gain insight into the potential area savings derived from increasing IIs, we created various implementations of circuits from the *single-small* set. We reimplemented each case, each one with a higher II, stopping when the total area began to increase. We also decreased the reserved area itself when possible (e.g., from *med1* to *sml2*) as the II increased, since doing so results in lower slice usage.

Fig. 5 shows the maximum decrease in slice usage when increasing each case's II between 1 and 10 clock cycles, as well as the respective performance decrease. Both are normalized to the area and performance achieved by the corresponding minimum II implementation. The average decrease in performance is nearly twice the decrease in area, meaning that opting for the minimum-II is the best strategy, assuming their area usage is acceptable. The size of the register pool and the configuration word length inhibits greater area reductions. As improvements, the former should be relocated to the reconfigurable area, and the later could benefit from more efficient encoding.

VI. CONCLUSION

In this paper, we presented a DPR enhancement to an existing CLA used for transparent binary acceleration in embedded systems. By resorting to DPR, we can support multiple loop kernels per CLA instance, while minimizing area usage and maximizing operating frequency. As the number of supported kernels increases, the benefits of resorting to this strategy are more evident, with area savings reaching upward of 1.61×. We also conclude that it would have been even more beneficial to move additional CLA logic into its reconfigurable area, as only LUTs are heavily used within the reserved area.

We explored the case where each kernel is implemented as an individual partial bitstream. The scheduler and CLA, however, support the implementation of several kernels by the same partial bitstream. This means further exploration could be performed to find solutions that prevent reconfiguration overhead incurred by DPR without sacrificing its area saving benefits. As the future work, we plan to explore this mechanism, as well as a prefetching strategy that mitigates the DPR overhead at least partially, since both the DMA engine and the ICAP port can operate in parallel with the host processor.

REFERENCES

[1] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled execution of recurring traces for energy-efficient general purpose processing," in *Proc. 44th Annu. IEEE/ACM Int. Symp. Microarchitecture (MICRO)*, Dec. 2011, pp. 12–23.

[2] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically specialized datapaths for energy efficient computing," in *Proc. IEEE 17th Int. Symp. High Perform. Comput. Archit.*, Feb. 2011, pp. 503–514.

[3] M. Lin, S. Chen, R. F. DeMara, and J. Wawrzynek, "ASTRO: Synthesizing application-specific reconfigurable hardware traces to exploit memory-level parallelism," *Microprocess. Microsyst.*, vol. 39, no. 7, pp. 553–564, 2015.

[4] M. Stojilovic, D. Novo, L. Saranovac, P. Brisk, and P. Ienne, "Selective flexibility: Creating domain-specific reconfigurable arrays," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 32, no. 5, pp. 681–694, May 2013.

[5] R. Seedorf, F. Anjam, A. Brandon, and S. Wong, "Design of a pipelined and parameterized VLIW processor: ρ-VEX v2.0," in *Proc. 6th HiPEAC Workshop Reconfigurable Comput.*, Paris, France, Jan. 2012, p. 12.

[6] R. Ferreira, W. Denver, M. Pereira, J. Quadros, L. Carro, and S. Wong, "A run-time modulo scheduling by using a binary translation mechanism," in *Proc. 14th Int. Conf. Embedded Comput. Syst., Archit., Modeling, Simulation (SAMOS)*, Jul. 2014, pp. 75–82.

[7] M. Fazlali, A. Zakerolhosseini, and G. Gaydadjiev, "A modified merging approach for datapath configuration time reduction," in *Reconfigurable Computing: Architectures, Tools and Applications* (Lecture Notes in Computer Science), vol. 5992, P. Sirisuk, F. Morgan, T. El-Ghazawi, and H. Amano, Eds. Berlin, Germany: Springer, 2010, pp. 318–328.

[8] N. Moreano, E. Borin, C. de Souza, and G. Araujo, "Efficient datapath merging for partially reconfigurable architectures," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 24, no. 7, pp. 969–980, Jul. 2005.

[9] N. M. C. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Generation of customized accelerators for loop pipelining of binary instruction traces," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 25, no. 1, pp. 21–34, Jan. 2017.

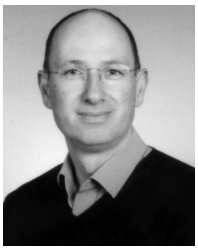
[10] S. Rokicki, E. Rohou, and S. Derrien, "Hardware-accelerated dynamic binary translation," in *Proc. Design, Automat. Test Eur. Conf. Exhib. (DATE)*, Mar. 2017, pp. 1062–1067.

[11] C. Lee, M. Potkonjak, and W. H. Mangione-Smith, "MediaBench: A tool for evaluating and synthesizing multimedia and communications systems," in *Proc. ACM/IEEE 30th Annu. Int. Symp. Microarchitecture*, Dec. 1997, pp. 330–335.

[12] J. Bispo and J. M. P. Cardoso, "On identifying segments of traces for dynamic compilation," in *Proc. Int. Conf. Field Program. Log. Appl. (FPL)*, Aug./Sep. 2010, pp. 263–266.
[13] J. Bispo. (Feb. 2015). *Megablock Extractor for MicroBlaze*. Accessed: Apr. 4, 2018. [Online]. Available: https://sites.google.com/site/specsfeup/
[14] T. Peters. (1992). *Livermore Loops Coded in C*. Accessed: Feb. 23, 2018. [Online]. Available: http://www.netlib.org/benchmark/livermorec
[15] Texas Instruments. (2008). *TMS320C6000 Image Library (IMGLIB)—SPRC264*. Accessed: Dec. 23, 2012. [Online]. Available: http://www.ti.com/tool/sprc264

Nuno M. C. Paulino received the M.Sc. and Ph.D. degrees in electrical and computer engineering from the Faculty of Engineering, University of Porto, Porto, Portugal, in 2011 and 2015, respectively.

He is currently an Assistant Professor at the University of Porto. He is also a Researcher at the Instituto de Engenharia de Sistemas e Computadores—Tecnologia e Ciência, Porto. His current research interests include runtime reconfigurable systems, embedded systems in field programmable gate arrays, coprocessor hardware acceleration, and tools for hardware/software codesign automation.



João Canas Ferreira (SM'09) received the Licenciatura and Ph.D. degrees in electrical and computer engineering from the University of Porto, Porto, Portugal, in 1989 and 2001, respectively.

He has been an Assistant Professor at the Faculty of Engineering, University of Porto. He is also a Senior Researcher at the Instituto de Engenharia de Sistemas e Computadores—Tecnologia e Ciência, Porto. His current research interests include dynamically reconfigurable systems, application-specific architectures for cognitive radio and sensor networks, and adaptive embedded systems.

Dr. Ferreira is a member of ACM and Euromicro.

João P. Cardoso (M'93–SM'09) received the D.Eng. degree from the University of Aveiro, Aveiro, Portugal, in 1993 and the M.Sc. and Ph.D. degrees in electrical and computer engineering from the Instituto Superior Técnico, Technical University of Lisbon (IST/UTL), Lisbon, Portugal, in 1997 and 2001, respectively.

From 1993 to 2006, he was at the University of Algarve, Faro, Portugal. From 2001 to 2002, he was at PACT XPP Technologies, Inc., Munich, Germany. From 2001 to 2009, he was a Senior Researcher at INESC-ID, Porto, Portugal. From 2006 to 2008, he was at IST/UTL. He is currently a Full Professor at the Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal, and a Senior Researcher at the Instituto de Engenharia de Sistemas e Computadores–Tecnologia e Ciência, Porto. His current research interests include compilation techniques, domain-specific languages, reconfigurable computing, and application-specific architectures.

Dr. Cardoso is a Senior Member of ACM.