Transparent Trace-Based Binary Acceleration for Reconfigurable HW/SW Systems

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

***Abstract*—This paper presents a novel approach to accelerate program execution by mapping repetitive traces of executed instructions, called Megablocks, to a runtime reconfigurable array of functional units. An offline tool suite extracts Megablocks from microprocessor instruction traces and generates a Reconfigurable Processing Unit (RPU) tailored for the execution of those Megablocks. The system is able to transparently movebcomputations from the microprocessor to the RPU at runtime. A prototype implementation of the system using a cacheless MicroBlaze microprocessor running code located in external memory reaches speedups from $2.2\times$ to $18.2\times$ for a set of 14 benchmark kernels. For a system setup which maximizes microprocessor performance by having the application code located in internal block RAMs, speedups from $1.4\times$ to $2.8\times$ were estimated.**

***Index Terms*—Binary translation, hardware accelerator, instruction traces, megablock, reconfigurable computing.**

I. Introduction

THE challenging requirements of designing and implementing high-performance and flexible industrial control systems at low cost have made the use of field programmable gate arrays (FPGAs) an attractive option [1]. These modern, high-capacity devices are being used as platforms for the implementation of complete systems-on-chip, including one or more central processing units (CPUs) connected to application-specific accelerators. However, the required design efforts to implement those systems are high. The design-flow combines software development and hardware design, the latter usually starting from a specification in a hardware description language (HDL) such as Verilog [2], and thus requiring hardware design expertise.

Manuscript received September 22, 2011; revised February 13, 2012; accepted November 30, 2012. Date of publication December 21, 2012; date of current version August 16, 2013. This work was supported in part by the ERDF—European Regional Development Fund through the COMPETE Programme (Operational Programme for Competitiveness), in part by National Funds through the FCT—Fundação para a Ciência e a Tecnologia (Portuguese Foundation for Science and Technology) within project "FCOMP—01-0124-FEDER-022701," in part by FCT grants SFRH/BD/36735/2007 and SFRH/BD/80225/2011, and in part by the European Community's Framework Programme 7 (FP7) under contract No. 248976. Paper no. TII-11-562.

J. Bispo is with INESC-ID, Lisboa 1049-001, Portugal, and with the Faculty of Engineering, University of Porto, Porto 4200-465, Portugal (e-mail: joao-bispo@gmail.com).

N. Paulino, J. M. P. Cardoso, and J. C. Ferreira are with Faculty of Engineering, University of Porto, Porto 4200-465, Portugal, and also with INESC-TEC, Instituto de Engenharia de Sistemas e Computadores—Tecnologia e Ciência, Porto 4200-465, Portugal (e-mail: nmcp@inescporto.pt; jmpc@acm.org, jcf@fe.up.pt).

Digital Object Identifier 10.1109/TII.2012.2235844

One way to a faster design process is to use high-level synthesis tools, such as Catapult C, which translate C code to HDL [3]. This often requires rewriting the source code to fit the translator's requirements and limitations. Implementing the interface between the generated hardware and the software is also necessary, a task which might require additional, manually-developed hardware, and further source code modifications.

The work described here explores an alternative approach: loops in an execution trace are automatically identified and mapped to a Reconfigurable Processing Unit (RPU), consisting of a specialized reconfigurable array of Functional Units (FUs), and whenever one of those loops needs to be executed the array is transparently invoked and configured at runtime. The loops chosen for hardware implementation correspond to a special type of repetitive instruction traces called Megablock [4]. The Megablock is a type of loop detected in the execution trace of a program. All iterations of a Megablock type loop have the same execution path. The Megablock has been proposed as a structure to be used for mapping execution traces of CPU instructions to hardware accelerators [4]. The detection of Megablocks is done by identifying repeating patterns of elementary units (e.g., basic blocks) in the streams of instructions forming the execution trace of a program.

The Megablocks identified are transformed into CDFGs (Control-Data Flow Graphs), which are then used as input to an offline tool chain that generates a specialized RPU for accelerated execution of the specific set of loops represented by the Megablocks. The RPU is runtime-reconfigured to execute each Megablock. In this paper, the program execution traces are obtained using a cycle accurate CPU simulator. The synthesis of the RPU is done offline, but the reconfiguration of the RPU occurs at runtime without changes to the executable binary.

A preliminary version of the system prototype, presented in [5], revealed some of the advantages of this approach. The main contributions of the work presented in this paper are:

- Novel approach to the acceleration of binary code by mapping object code blocks (sets of Megablocks) to a runtime reconfigurable array of FUs;
- Completely transparent use of the reconfigurable array of FUs at runtime using an unmodified CPU;
- Implementation of a fully-functional hardware prototype using a MicroBlaze [6] as CPU on a Spartan-6 Xilinx FPGA [7];
- Experimental evaluation of the system showing significant performance improvements.

The remainder of this paper is organized as follows. Section II presents previous work in this area. Section III introduces the Megablock, while Section IV describes the proposed reconfig-

urable architecture and explains some of the design decisions made for the implementation of the system. Section V presents experimental results collected from the implementation of the system on a Spartan-6 FPGA, together with estimates for an improved architecture. Finally, Section VI concludes the paper and briefly introduces future work.

II. Related Work

Considering that FPGAs are becoming more common of-the-shelf components in embedded systems across several fields [1], [8], [9], it has become important to fully exploit their processing capabilities. This also entails integrating FPGA-related development tasks in the product development flow so as to limit toolchain complexity and reduce engineering costs over the whole product lifetime [10], [11]. One particular research topic has been the use of reconfigurable fabrics (e.g., FPGAs) to accelerate execution of general purpose applications in embedded scenarios [12], [13]. Some approaches include the Warp Processor [14], [15], the AMBER architecture [16], [17], the Configurable Compute Array (CCA) [18], [19], the DIM Reconfigurable System [20], [21], and the Megablock approach [4], [22].

The Warp Processor [14], [15] is composed of a MicroBlaze processor with a loosely coupled custom FPGA. In [15] a multi-processor scenario is also analyzed. Basic blocks are detected at runtime by backward branch monitoring, and on-chip tools generate a hardware description for the custom FPGA. The binary of the application is then modified at runtime to use the generated hardware.

The AMBER architecture [16], [17] is based on a tight coupling between the hardware accelerator and a MIPS processor. The accelerator is integrated into the processor's pipeline and reconfigured at runtime using stored configurations. These are generated offline, from applications running on a simulator, by detection of repeated backward branches over a certain threshold. This approach is intrusive due to the tight coupling and requires further development effort to adapt to different processor architectures.

The CCA [18], [19] uses an accelerator tightly coupled to an ARM processor. Their work addresses CDFG detection and mapping. Detection was initially performed online, but was later moved to a compile time step [18], which requires special instructions to delimit the code regions to be mapped to hardware, making the approach non-transparent to software toolchains. In further work, the CCA was integrated as an FU into a different type of loop accelerator, focused on binary portability for hardware accelerated systems, specifically, acceleration of modulo-schedulable loops [23]. Since binary compatibility of applications developed for a given hardware accelerator system is compromised upon further hardware development, [23] proposes a solution where a virtual machine module monitors the processor's instruction stream and generates control and configurations for any given accelerator, decoupling the software from the hardware. Configurations are generated from binary translation, and alternatives implementing some translation steps either online or offline are presented. An average global speedup of 2.66 is achieved for the optimal solution.

The DIM Reconfigurable System [20], [21] is based on a run-time binary translation mechanism that transparently detects groups of up to 3 basic blocks suitable for hardware execution. The accelerator is tightly coupled to a MIPS processor pipeline, having access to its register file. It is composed of heterogeneous coarse grained FUs arranged in homogenous rows. Load/Store FUs allow for memory access to random memory positions.

Paek *et al.* [24] implement a coarse-grained array of homogeneous FUs with a static interconnection scheme. Reconfiguration occurs at runtime according to information extracted from binaries disassembled offline. It is a static code analysis approach that detects suitable loops (e.g., with an iteration count known at compile time; kernels should not be control-dominated). The loops are compiled for the accelerator, and the binary is modified to include mapping and communication routines. Sequential memory accesses are supported, and data are passed to/from the array via a shared memory mechanism. In contrast, the approach proposed in this paper uses information from execution traces, which allows for precise determination of frequent execution paths, and imposes fewer restrictions on the detected loops (e.g., it is not necessary to know the number of iterations). The executed binary is unmodified, and the resulting reconfigurable array uses fixed-functionality processing elements with reconfigurable interconnects, and is tailored for accelerating a specific set of loops.

Bispo and Cardoso [4], [22] propose the mapping of repetitive trace-based patterns of instructions, named Megablocks, in the context of dynamically moving instructions executing on a CPU to reconfigurable arrays. The objective of the Megablock is to address several difficulties related to mapping loops of sequential code to hardware, such as determination of the number of loop iterations, and the handling of jumps and branches in sequential code. It is shown that Megablocks can represent significant portions of the execution of a program, with coverage comparable to or greater than other online loop-detection methods, such as loops identified by backward jumps. As each Megablock considers a single execution path, and is thus suitable for optimizations, it is inappropriate for execution traces with highly irregular paths. Alternative paths are treated as exit points. Although the approach is analyzed and speedup estimations are presented in [4], [22], the current paper presents the first real implementation of a software/hardware system using the Megablock concept. It also shows experimental evidence of the suitability of the approach to accelerate execution traces.

III. Megablocks

A Megablock [22] represents a sequence of executed instructions (trace) forming an iteration *it*, which repeats at least 2 times until one of its exit conditions is true. Fig. 1 shows a portion of the instruction trace from the calculation of the Fibonacci sequence on a MicroBlaze processor. The code has a single loop which, when executed, repeats the same sequence of six instructions. This sequence of six instructions represents the iteration *it* of the Megablock. It contains five arithmetic instructions (using *addk*, *addik*, and *rsubk*) and one branch instruction (*bneid*). Whenever the branch instruction is executed, the value of register $r18$ is compared to zero, and if different from zero,

```
...
0x000001A8 addk r4, r3, r0
0x00000194 addk r3, r4, r7
0x00000198 addik r6, r6, 1
0x0000019C addk r7, r4, r0
0x000001A0 rsubk r18, r6, r5
0x000001A4 bneid r18, -16
0x000001A8 addk r4, r3, r0
0x00000194 addk r3, r4, r7
0x00000198 addik r6, r6, 1
0x0000019C addk r7, r4, r0
0x000001A0 rsubk r18, r6, r5
0x000001A4 bneid r18, -16
0x000001A8 addk r4, r3, r0
0x00000194 addk r3, r4, r7
...
```

Fig. 1. Repeating pattern of instructions in the trace of the *fibonacci* benchmark, running on a MicroBlaze processor.

the control flow jumps back four instructions and repeats iteration *it* (as the branch has a single delay slot, it will execute an additional instruction before jumping). If register $r18$ is equal to zero, the repetitions stop. This branch instruction represents the only exit point of this Megablock.

The number of iterations in the loop of a program might not be known *a priori*. Thus, to widen its applicability, instead of a traditional loop control with a fixed number of iterations, the Megablock exclusively uses "exit points" to determine when computations end. By definition, a Megablock always has at least one exit point.

Control-flow instructions (e.g., branches) make hardware implementation more difficult and prevent optimizations, due to the multiple paths they introduce. Instead of considering multiple paths, each Megablock represents a single path of a loop across several branches. Thus, the Megablock includes an exit point for each instruction able to change the control flow.

Fig. 2 shows C code for a loop which contains an inner loop and a possible infrequent path for the common execution scenario, and Fig. 2 shows the CDFG of the outer loop. Depending on the execution, the trace produced by running the code can form two different Megablocks with the same start address, "ABBD" and "ACD", as shown in Fig. 3(c). If a Megablock is formed for path "ACD", its iteration count may be too low to warrant implementation in hardware. Alternatively, a Megablock may be formed for the repetitive path "ABBD" (with unrolled inner loop), with the branch to the infrequent path considered as an exit point.

Megablocks are useful if they represent a significant part of the execution of a program. Previous work [22] shows that for many benchmarks, Megablocks can have coverage similar to or greater than other runtime detection methods, such as monitoring short backward branches (used by Warp [8]).

A. Detecting Megablocks

Megablocks are found by detecting a repetitive pattern of up to M instructions in the execution trace. Since each individual instruction has an associated address value, the problem is equivalent to detecting patterns in the instruction addresses. Fig. 1 has a pattern of size 6, which corresponds to the sequence of addresses 0x194, 0x198, 0x19C, 0x1A0, 0x1A4 and 0x1A8.

```
for (i=0; i<n; i++) {
   // Infrequent Path
   if(a[i] > max_value) a[i] = max_value;
   else for(j=i; j<i+2; j++) // inner loop
    a[j] = a[j] + 2;
}
```

a)

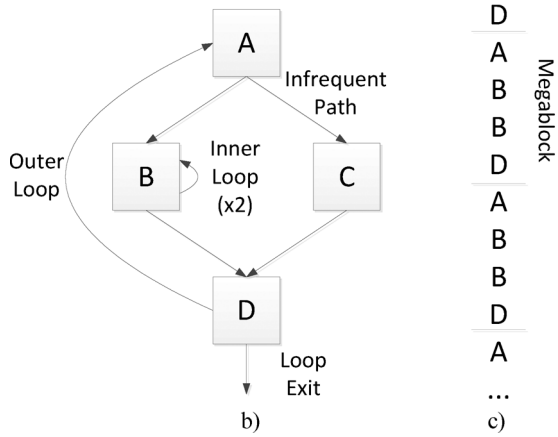


Fig. 2. Example of an inner loop with an infrequent path: (a) C code; (b) CDFG; (c) Execution trace of the CDFG and Megablock formation.

The size of detected patterns can be reduced by considering coarser elements other than instruction addresses [16]. For instance, pattern detection can be done over basic block addresses. In Fig. 1, the list of addresses from 0x194 to 0x1A8 forms a single basic block. Using basic block addresses instead of instruction addresses as pattern elements decreases the size of the detected pattern from 6 to 1. A study using a representative set of benchmarks has shown that to maximize the coverage of detected Megablocks, the maximum size of the pattern can be as high as 32, even when considering coarser elements (e.g., basic blocks) [22].

The problem of detecting a Megablock can be seen as an instance of the problem of detecting repeated substrings, e.g., xx, with x being a substring containing 1 or more elements. This is also known as the *squares*, or tandem repeats detection problem [25]. In our case, substring x represents a single iteration of a loop. Although the objective is to find patterns with many repetitions (a square strictly represents only two repetitions), it was observed that if a sequence of instructions forms a square, it is likely that more x elements will follow (e.g., $xxxx\ldots$). The method adopted here considers that two repetitions are enough to declare the detection of a Megablock.

There are algorithms which can find all tandem repeats in $\mathrm{O}(n \log n + z)$, where n is the length of the string and z is the size of the output [26]; there is also a linear time algorithm which uses suffix trees [27]. However, these algorithms are not suitable if runtime Megablock detection is needed. Since the stream of instructions is usually consumed at a constant rate (i.e., when there are no stalls), we favor streaming oriented algorithms, possibly using a buffer temporarily storing contiguous subsequences of instructions.

Fig. 3 presents the algorithm developed to meet these requirements. The algorithm defines, *a priori*, the maximum size of the sequence (i.e., the maximum number of elements in a pattern). It uses M counters, one for each element of the *FIFO*. The *FIFO*

```
M - maximum number of elements in sequence
MatchingFifo   - size M FIFO of PatternElements
MatchCount     - size M, initialized to zero
PatternElement - unique representation of each
                 instruction or basic block

processElement(PatternElement)
  for idx 1 to M
    if PatternElement equals MatchingFifo[idx]
      if MatchCount[idx] < idx
        MatchCount[idx]++;
      else
        MatchCount[idx] = 0;

  for idx 1 to M
    if MatchCount[idx] equals idx
      Pattern of size idx detected

  shift PatternElement into MatchingFifo
```

Fig. 3. Algorithm for detection of squares up to a maximum size.

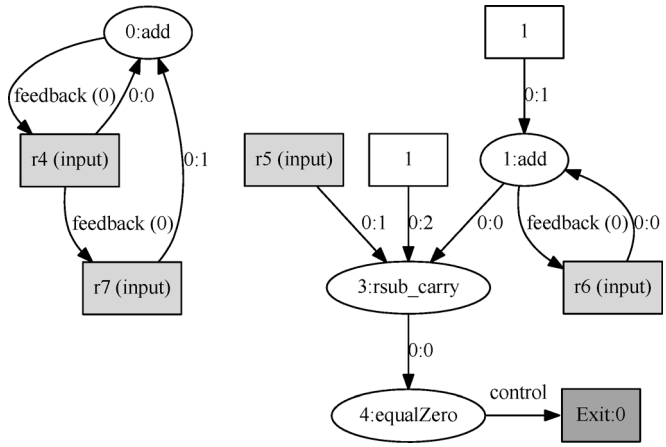


Fig. 4. CDFG of the Megablock found in the *fibonacci* kernel.

stores the previous M elements and gives read access to any element. When a new element arrives, it is compared with the M elements already in the *FIFO*. The position in the *FIFO* determines the size of the pattern being detected (e.g., position 2 detects patterns of size 2). If there is a match in a position of the *FIFO*, the counter corresponding to that position is incremented. If the counter reaches a value equal to the corresponding pattern size, a match for that size is signaled. If there is a mismatch, the counter is reset to zero. Finally, the element is added to the *FIFO*. When the *FIFO* is full, the oldest value is discarded.

When the *FIFO* is full, each following element can signal up to M matches for squares with different sizes. For instance, by feeding the pattern *aaaaaa* to the algorithm, after processing the last element 3 matches are detected, for squares with sizes 1 (a), 2 (aa) and 3 (aaa), respectively. An arbiter must select the most relevant match. For instance, in order to consider only inner loops, priority is given to the match with the smallest substring size; to detect patterns with unrolled inner loops, but only when they appear inside outer loops (e.g., $aabaab\ldots$), priority is given to the match with the longest pattern, but only if there is no match of a shorter pattern simultaneously in the current and in the previous set of matches (to prevent unrolling in cases such as $aaaa$).

Because of the repetitive nature of the Megablock, any address in the pattern can be taken as the start address. As the start address can influence single-pass optimizations, we chose the lowest address of the Megablock which appears only once as the start address. For the example in Fig. 1, the start address according to this heuristic is 0x194.

B. Megablock Graph

In the mapping approach described in this paper, Megablocks are transformed and then mapped into a dataflow graph (DFG) representation. Fig. 4 represents the graph obtained from the execution trace of Fig. 1.

The DFG is built by taking the start address of the Megablock (i.e., 0x194 in Fig. 1) and adding nodes to the graph according to the sequence of instructions of the Megablock. Each CPU instruction is transformed into one or more platform-independent instructions. When building the graph, additional information—not represented in Fig. 4—is generated, including a counter for each type of graph node, tables which map each output of the graph to its last definition in the graph, and a table with all non-data dependencies between operations. This information is needed for implementing Megablocks, for instance, to indicate which nodes produce the output values that must be sent to the CPU on termination of the Megablock, which CPU registers are the destinations of these values and what restrictions need to be imposed when scheduling operations. Optimizations such as constant folding and propagation, and algebraic simplifications are then applied to the DFG.

The DFG represents one iteration (the kernel) of the Megablock and contains operations (oval nodes), constants (white square nodes), input values ("LiveIn" square nodes) and exit points ("Exit" square nodes). All connections in the DFG represent dataflow between the nodes. For illustrative reasons, the labels in the connections in Fig. 4 indicate output and input indexes, respectively. For instance, "0:1" means "output 0 of source node connects to input 1 of destination node". The connection labeled "control", between source node `4:EqualZero` and destination node `Exit:0` establishes a 1-to-1 relationship between the Boolean output of an operation (e.g., true or false) and an exit point. The output determines if an exit is activated or not, according to the exit rule of the destination node.

The results from each Megablock iteration are passed to the following iteration via feedback connections that feed data back to the "LiveIns", as dictated by the detected dataflow.

IV. Hardware Architecture and Support Tools

The architecture of a hardware prototype supporting transparent hardware acceleration is shown in Fig. 5. In order to avoid modifications of the CPU or its interfaces, or of the software toolchain, as in other approaches [16], [19], [20], the implementation uses standard interfaces such as the PLB (Processor Local Bus) [6]. Modifications of the CPU instruction streams are made at runtime without rewriting instruction memory and thus ensuring full transparency.

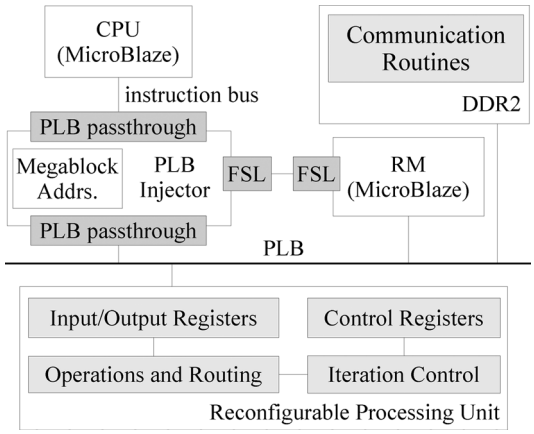


Fig. 5. Target System Architecture.

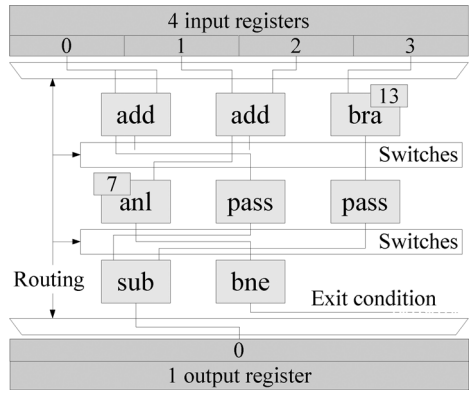


Fig. 6. Synthetic example of a reconfigurable processing unit (RPU).

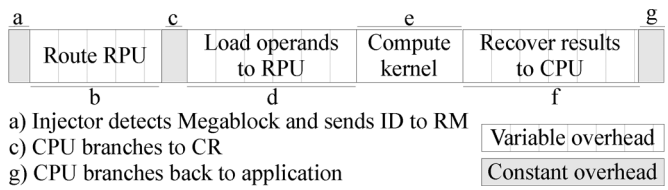


Fig. 7. System communication model.

The hardware prototype system consists of five modules as shown in Fig. 5. The Reconfigurable Processing Unit (RPU) is a loosely coupled hardware accelerator connected to the PLB. An injector module monitors the instruction address bus and triggers the use of the RPU by modifying the contents of the instruction stream. The injector acts as a pass-through for all the instructions to be executed by the CPU. In this prototype, the CPU loads program instructions from external memory (DDR2). The Reconfiguration Module (RM) is responsible for managing the runtime configuration of the RPU.

The system was conceived for an FPGA environment: instead of using a single all-purpose RPU, the tool chain generates the HDL description of an RPU tailored for a single or a set of programs to be executed in the system. This step is done offline and automatically, as detailed further in this section.

A. RPU Architecture

Fig. 6 shows an example of a possible arrangement of the array of FUs in the RPU. The RPU is organized as a set of rows with a variable number of single-operation FUs. If there is an operation with one constant input, the RPU generation process tailors the FU to that input, as is the case with the *bra* FU in Fig. 6, which represents an arithmetic shift right by 13 operation. The current implementation supports arithmetic and logic operations with integers, including carry operations. Floating point operations, integer divisions, and memory accesses are currently not supported. Crossbar connections are included between adjacent rows, and are runtime reconfigurable. Connections spanning more than one row are established by pass-through FUs (labeled *pass* in Fig. 6). Runtime reconfiguration changes the connections established by the inter-row crossbar switches. If the RPU is already configured for the Megablock to be executed, reconfiguration is skipped, reducing its overhead. Fig. 6 also depicts one possible interconnection arrangement for a Megablock. According to its configuration, the RPU executes any one member of the set of Megablocks it was designed to support. Configurations applied at runtime are generated offline along with the RPU description.

The architecture of the RPU was heavily influenced by the structure of the Megablocks. Specifically, the RPU was designed to execute loops with one path and multiple-exits. The RPU does not need to know the number of iterations of the loop before execution; instead, it keeps track of the possible exit conditions of the loop and detects when an exit situation occurs (the *bne* FU in Fig. 6 performs exit detection in this example). For an RPU supporting multiple Megablocks, the configuration also determines which exit conditions are to be considered to terminate execution. The results of the iteration that triggers an exit, i.e., the last iteration of the loop, are discarded, and the last iteration of the loop is re-executed in software. The CPU resumes execution at the Megablock address so it can follow the correct control-flow path, which leads to the execution of the branch corresponding to the triggered exit.

In the current implementation of the RPU, all operations complete within one clock cycle. Each loop iteration takes as many clock cycles as the number of rows in the RPU, as intermediate results are registered at the outputs of the FUs. The RPU acts as a bus slave communicating by means of a standard PLB interface. Data transfer of operands and results is done through memory mapped registers. Configuration is performed by writing to a set of configuration registers, whose number depends on the size of the array. These registers control the routing of the operands through the RPU and define which exit conditions are active.

B. Communication Scheme

The transparent execution of computations on the RPU entails several steps, summarized in Fig. 7. The injector is responsible for interfacing the CPU with the rest of the system, as well as for starting the reconfiguration process.

This approach changes the execution flow of the CPU without overwriting the original instructions of the program or interfering with the original software toolchain. The execution of the Communication Routine (CR) may introduce a significant overhead (slots d and f). Each CR contains 20 fixed MicroBlaze instructions, plus one instruction per input or output value.

The tasks performed by the CR include loading values into the RPU (in slot d), polling the RPU for completion (during slot e), checking the exit status and recovering values (slot f).

C. Prototype Considerations

As mentioned before, the transfer of control flow performed by the injector is triggered by the detection of an instruction address that corresponds to the start of a Megablock. Although two or more Megablocks may start at the same memory address, the current approach considers for Megablock implementation the one with the highest code coverage measured during profiling.

Situations may occur where the CPU fetches instructions which will never execute. This occurs if a Megablock starts after a mispredicted branch instruction. The implementation correctly identifies this situation: if the starting Megablock address is followed by the next address in the Megablock, this means the CPU did not discard the instruction, and is attempting to execute the code region mapped to the RPU.

In the current implementation, the feedback routing of results to the next iteration is configured through a single register. This limits the maximum number of input/outputs of the RPU. It depends on a combination of the number inputs and outputs, e.g., a maximum of 10 inputs for 8 outputs. A similar limitation imposes a maximum number of 32 exit conditions. However, these limitations were not an obstacle for implementing the Megablocks found in the benchmarks being used.

D. Tool Flow

Starting with an executable file in ELF (Executable and Linkable Format), the offline tool suite extracts Megablocks, maps them to the RPU, and generates the configurations. The Graph Extractor tool [28] uses a cycle-accurate simulator of the CPU to obtain execution traces.

The set of selected Megablocks is processed by two tools. One generates the HDL (Verilog) descriptions for the RPU and routing information to be used at runtime by the RM. The other generates the CPU instructions for the communication routines and a Verilog header file with the Megablock addresses for the Injector. Both the routing information and the CRs are included into the program memory of the RM. The RM copies the CRs to DDR2, so that they can be executed by the CPU. The RPU description generation tool processes Megablock information provided by the Graph Extractor tool [28], determines FU sharing across Megablock DFGs, assigns FUs to rows, adds pass-through units, and generates Verilog header files that characterize the placement of FUs. As only one Megablock will be executing on the RPU at any given time, the tool generates a description which reuses FUs between Megablocks.

V. Experimental Evaluation

The proposed architecture and tools were tested and evaluated with 14 integer benchmarks from embedded computing, as well as commonly used algorithms [29]. Each benchmark contains a kernel that operates on 32-bit integer values. In order to ensure a fair comparison, each benchmark executes the corresponding kernel N times, where N is a compile-time constant. Nine of the kernels have a fixed number of loop iterations: *count*, *even_ones*, *ham_dist*, *pop_cnt*, *reverse*, *divlu*, *isqrt2*, *sqrt* and *usqrt*. The number of iterations of the remaining kernels depends on the values of the inputs.

Two additional benchmarks were considered, each one combining a set of 6 kernels: *merge1* consists of *count*, *even_ones*, *fibonacci*, *ham_dist*, *pop_cnt* and *reverse*; *merge2* consists of *compress1, divlu, expand, gcd2, isqrt2* and *maxstr1*. These benchmarks are used to validate the transparent support of multiple Megablocks with different characteristics on a single RPU. In addition, they are used to verify that the tools correctly identify resources that can be reused between Megablock DFGs.

A. Experimental Setup

The Graph Extractor tool [28] was used to do an offline extraction of the Megablocks from execution traces. For the detection, inner loop unrolling was disabled, basic blocks were used as the pattern element, and the maximum pattern size considered was 32. For each kernel (except the *merge* ones), the Megablock with the highest coverage was implemented. On average, the selected Megablocks cover 90% of the executed instructions.

For the evaluation, parameter N was set to 500, and each kernel is compiled with *mb-gcc* 4.1.2, using the $-$O2 flag and additional flags which enable specific units of the MicroBlaze processor (e.g., -mxl-barrel-shift for barrel shifter instructions).

The prototype uses a Digilent Atlys board with a Xilinx Spartan-6 LX45 FPGA [7] and 128 Mbyte DDR2 memory. The CPU is a MicroBlaze processor optimized for speed, clocked at 66.7 MHz. The same clock signal is used for all modules, including the RPU. Two kernels (*mpegcrc* and *usqrt*) required lowering the clock frequency of the system to 33.3 MHz, due to delays in the generated RPU. The RM was implemented as another MicroBlaze with the reconfiguration data in its local memory. This additional MicroBlaze can also be used for monitoring purposes.

B. Results

Table I summarizes the characteristics of the Megablocks used in the evaluation. Most kernels present similar values for the number of instructions executed per Megablock call; the most notable exception is *fibonacci*, which executes from 3 to 16 times more instructions per call than the other benchmarks. Included in Table I are values for maximum Instruction Level Parallelism (ILP), percentage of instructions covered by the Megablocks (column *Cov.*) over the total executed instructions, and average number of Instructions per Cycle (IPC), assuming each instruction takes one clock cycle to execute.

Table I also summarizes the characteristics of the RPUs generated for each kernel. Due to the interconnection scheme used, most of the FUs are pass-throughs. The FU column indicates the total number of FUs, and *OP Ratio* gives the percentage of FUs that implement actual operations (as opposed to pass-throughs). The RPU depth (i.e., number of rows) ranges from 3 to 8. The number of bits required for configuration of the inter-row switches grows with the number of FUs in each row and with RPU depth: the largest individual benchmark *(usqrt)* requires 352 bits (44 bytes).

The RPUs for the merged benchmarks exploit resource sharing. Therefore, the total number of resources used is less

TABLE I
CHARACTERISTICS OF THE EXTRACTED MEGABLOCKS AND GENERATED RPUS

Kernels	Megablocks				RPUs				
	Avg. Inst. p/call	ILP	Cov (%)	IPC	#FU	FR	OP Ratio (%)	Depth	#Conf. Bits
count	192	2	94.9	2.0	12	4	50.0	3	72
even_ones	192	3	94.0	2.0	16	6	37.5	3	87
fibonacci	1497	3	99.4	2.0	15	6	40.0	3	87
ham_dist	192	3	94.0	2.0	15	6	40.0	3	81
pop_cnt	256	3	97.2	2.7	15	6	53.3	3	84
reverse	224	3	95.6	2.3	14	6	50.0	3	81
compress	138	3	89.7	2.0	26	8	30.8	4	160
divlu	155	2	90.5	1.7	9	4	55.6	3	38
expand	138	3	89.7	2.0	26	8	30.8	4	126
gcd	330	2	98.8	1.3	21	5	38.1	6	99
isqrt	90	3	84.0	2.0	13	5	46.2	3	72
maxstr	120	2	88.1	1.3	10	4	40.0	3	54
mpegcrc	434	4	87.6	2.0	40	7	35.0	7	189
usqrt	255	6	84.9	2.1	59	9	28.8	8	352
merge1	426	3	98.3	2.1	28	8	64.3	3	204
merge2	162	3	93.2	1.1	59	15	44.4	6	402

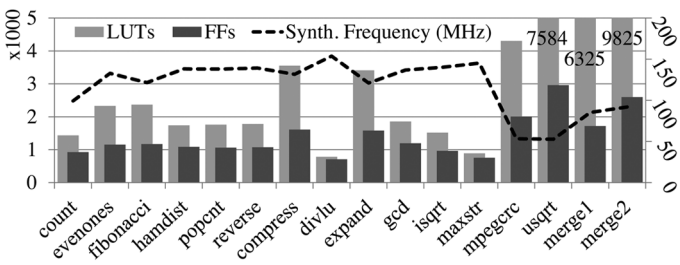


Fig. 8. Synthesis results for system implementation: Lookup Tables (LUTs), Flip-flops (FFs).

than the total sum of the resources of the individual benchmarks. Due to FU sharing, the *merge1* and *merge2* implementations use 32% and 51% of the total number of FUs (operations and pass-throughs), respectively.

The *merge2* benchmark demonstrates an RPU that supports Megablock DFGs of different depths. The RPU has a depth equal to the maximum depth of the individual DFG kernels, and smaller DFGs are supported by using pass-throughs in the additional rows.

Fig. 8 characterizes the FPGA implementation of the RPUs. The maximum RPU clock frequencies are in the 52.1—153.7 MHz range (as reported by the synthesis tools). Most RPUs work above the MicroBlaze clock frequency (66.7 MHz). Since the current implementation uses a single clock for the RPU and the MicroBlaze, most benchmarks were run with a 66.7 MHz clock; the exceptions are *mpegcrc* and *usqrt*, which used a 33.3 MHz clock.

Resource utilization for the Spartan-6 LX45 FPGA ranges from 5.3% to 27.8% for LUTs, and from 1.3% to 5.4% for flip-flops. The implementation of *merge1* requires 55% of the LUTs and 27% of the FFs that would be needed if the RPU were generated with no sharing of FUs. For *merge2*, these values are 81% and 38%, respectively.

Table II summarizes the results measured with the prototype. The execution times were measured using dedicated timers (counting clock cycles). For each benchmark, the table includes

TABLE II
PERFORMANCE RESULTS FOR DDR-BASED PROTOTYPE

Kernels	SW (ms)	SW+HW (ms)	**Speedup**	HW no OH (ms)	**OH (%)**
count	37.7	9.6	**3.9**	0.73	**92.4**
even_ones	38.0	9.1	**4.2**	0.73	**92.1**
fibonacci	280.8	15.4	**18.2**	5.67	**63.2**
ham_dist	38.0	8.8	**4.3**	0.73	**91.8**
pop_cnt	47.1	9.1	**5.2**	0.73	**92.1**
reverse	42.8	9.5	**4.5**	0.73	**92.3**
compress	27.9	10.4	**2.7**	0.52	**95.0**
divlu	32.9	9.3	**3.5**	0.73	**92.2**
expand	27.9	10.4	**2.7**	0.52	**95.0**
gcd	43.5	8.3	**5.3**	1.88	**77.3**
isqrt	20.1	9.0	**2.2**	0.34	**96.2**
maxstr	27.0	9.1	**3.0**	0.68	**92.5**
mpegcrc	137.3	22.7	**6.0**	3.29	**85.5**
usqrt	73.2	19.3	**3.8**	1.94	**89.9**
merge1	456.1	71.5	**6.38**	9.30	**87.0**
merge2	175.4	66.5	**2.64**	6.95	**89.6**

the execution time for the software version (SW), for the hardware-accelerated version (SW+HW), and the associated speedup. Also included is the running time of the hardware version without the communication overhead (HW no OH).

The last column shows the communication overhead (OH) relative to total execution time. The effective speedups measured (including communication overhead) range between $2.2\times$ and $18.2\times$ ($4.96\times$ on average). For the *merge* benchmarks the speedup is lower than the average of the individual benchmarks they merge ($6.7\times$ and $3.2\times$ for the benchmarks in *merge1* and *merge2*, respectively) due to the overhead of RPU reconfiguration.

Since the CPU has no cache, the MicroBlaze requires 23 clock cycles to fetch each instruction from the external DDR2 memory. For this scenario, the observed speedup is mainly due to avoiding this instruction fetch delay by executing operations on the RPU instead. Each CR is also in external memory. Therefore, the time required for executing the CR overwhelms the execution time of the RPU. The high communication overhead is also aggravated by the relatively low number of average instructions executed per call (see Table I).

In order to evaluate the performance difference between specialized hardware implementations of each kernel and our approach, all benchmarks except *merge1* and *merge2* were synthesized using the high-level synthesis tool, Catapult C Synthesis v2010a [30]. This approach generates one dedicated hardware module per kernel. For the synthesis we used loop pipelining with an iteration interval of 1 for all kernels. For the specialized architectures, on average only 0.53% of LUTs and 0.21% of the FPGA registers are needed. We consider here that the hardware cores generated with Catapult C replace the FU array and the same data communication scheme is used. The average estimated speedup obtained over the software solution is $5.7\times$ (from a minimum of $2.3\times$ and a maximum of $24.1\times$) for a 66.7 MHz system clock. Although the generated kernels are individually faster than the FU array (average clock frequency is 237 MHz), system performance is constrained by the communication overhead. Therefore, only a modest speedup over the RPU version is achieved (less than 3%).

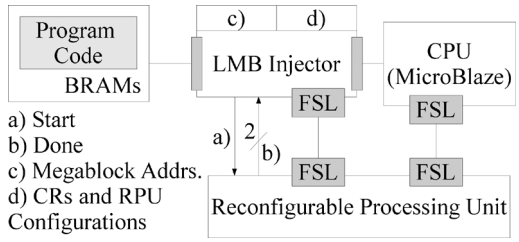


Fig. 9. System architecture with support for code execution from BRAM.

C. Speedup Estimation for BRAM+FSL-Based Execution

Using an external memory for CPU code without cache support imposes a significant overhead. An alternative is to consider an implementation which executes the kernels from on-chip memory implemented with BRAMs (Block RAMs), and use a point-to-point interface between the CPU and RPU. The injector interfaces with the RPU in the same manner. This scenario ensures the fastest possible execution of the software version. The injector is attached to the dedicated bus (Local Memory Bus, LMB) that connects the CPU to the local memory (see Fig. 9).

Communication with the RPU is done directly via a Fast Simplex Link (FSL) interface [6]. When a Megablock start address is detected, the injector directly inserts the stream of instructions, which are adapted CRs held in its own memories. The load/store instructions previously used by the CPU to communicate with the RPU via the PLB bus are now replaced with instructions (get/put) that send/receive a single value over the FSL. Once all operands are sent to the RPU, the injector sends a start signal to the RPU. The CPU then stalls until the RPU delivers the results as FSL *get* instructions are blocking. The RPU sends back a done signal which reports the exit status once calculations are done. The injector then continues the CR, recovering results. CRs are now fetched at the same speed as regular BRAM accesses, which is 1 instruction per cycle.

In order to estimate the speedups for this scenario, it is assumed that a CR now contains only 4 fixed instructions versus the previous 20. These are absolute branches back to the starting address of the Megablock and take 4 cycles to execute. In addition, there are as many put/get instructions as inputs/outputs. Each completes in one clock cycle instead of the 12 cycles required to write successive values over the PLB.

In this alternative, it is the injector that performs the RPU reconfiguration, instead of the RM, via its own FSL connection. This can be done at the same time as the CPU sends the operands to the RPU. Table III shows the estimates calculated for this model. The expression $n \times d$ was used to calculate the number of clock cycles for execution in the RPU alone (HW no OH), where n is the number of iterations and d is the depth of the RPU. The total execution time with hardware acceleration (SW+HW) includes the communication overhead, given by the expression $\mathrm{CR}_{\mathrm{cycles}} + N_{\mathrm{regs}} \times L_B$ to calculate the number of overhead clock-cycles, where $\mathrm{CR}_{\mathrm{cycles}}$ is the fixed number of cycles needed to execute the CR (4, in this case), N_{regs} is the number of RPU registers (inputs and outputs) to communicate, and L_B is the BRAM access delay (1 clock cycle, in this case). Since all benchmarks (except *merge1* and *merge2*) implement a single Megablock, we did not include the RPU reconfiguration overhead, which is only incurred once. For *merge1* and *merge2*, this overhead was considered. The estimates indicate an average overhead of 11% in individual benchmarks when using the FSL interface, versus 89% measured for the DDR case.

TABLE III
ESTIMATED PERFORMANCE FOR BRAM-BASED CODE EXECUTION

Kernels	SW (ms)	SW+HW (ms)	**Speedup**	HW no OH (ms)	**OH (%)**
count	2.0	0.9	**2.2**	0.7	**10.3**
even_ones	1.8	0.9	**1.9**	0.7	**11.9**
fibonacci	13.3	5.8	**2.3**	5.7	**1.7**
ham_dist	1.8	0.9	**1.9**	0.7	**11.1**
pop_cnt	2.5	0.9	**2.8**	0.7	**11.1**
reverse	2.0	0.9	**2.2**	0.7	**11.1**
compress	1.5	0.8	**1.9**	0.5	**16.9**
divlu	1.6	0.9	**1.6**	0.7	**8.6**
expand	1.6	0.8	**2.0**	0.5	**16.9**
gcd	3.2	2.0	**1.6**	1.9	**3.9**
isqrt	1.0	0.6	**1.7**	0.3	**19.6**
maxstr	1.3	0.9	**1.4**	0.7	**9.1**
mpegcrc	9.5	4.7	**2.0**	3.3	**6.5**
usqrt	6.3	3.1	**2.0**	1.9	**11.7**
merge1	23.1	10.3	**2.2**	9.2	**7.2**
merge2	9.7	8.6	**1.1**	6.9	**13.1**

As the software execution from BRAMs represents the best possible case for the MicroBlaze processor, hardware speedup is significantly reduced, because there is no longer a high penalty for accessing memory. However, estimations indicate that speedups (including all overheads) for individual benchmarks between $1.4\times$ and $2.8\times$ ($1.96\times$ on average) are achievable.

In this scenario, replacing the FU array by the kernels generated with Catapult C achieves higher speedups, since the communication overhead has a lower impact on the overall performance. The average estimated speedup is $4.6\times$, with speedups for individual benchmarks between $2.6\times$ and $7.1\times$. On average, the implementations generated by Catapult C are $2.4\times$ faster than our proof-of-concept implementation, while using about one-tenth of the resources. Note that in a system with specialized hardware versions, the injector module can still be used to transparently move the computation from the CPU to the dedicated hardware.

We applied the same estimation approach to an extended set of 62 integer benchmarks (including image processing algorithms [31] and commonly used algorithms [29]), which produce Megablocks with memory operations. The estimations indicate an average speedup of $2.7\times$. Those results assume the use of dual-port BRAMs, which support up to 2 memory operations per clock cycle [7]. This extended set of benchmarks includes more complex examples than the ones presented earlier in this section. Those benchmarks and their individual speedups include: *md5* ($1.8\times$), *bilinear* ($4.8\times$), *fft* ($1.2\times$), *fir* ($7.3\times$), and *fdct* ($6.7\times$). Using the same approach, we applied our mapping technique to an airborne collision avoidance application, known as 3D Path Planning, provided by Honeywell. It consists of 841 lines of C code, distributed over 10 files and 48 functions. A step of the application requires 50 601 067 MicroBlaze clock cycles. We use our target architecture with Megablocks mapped to a 2D CGRA based on the structure of the RPU depicted in Fig. 6, extended with 2 load/store units per row. This implementation

considers 9 Megablocks responsible for almost 87% of the total execution time. The speedup achieved by the execution of the Megablock sections of code in the RPU is around $2.34\times$, resulting in an estimated overall application speedup of $2\times$.

VI. Conclusion

This paper presented a novel approach for transparently moving computations from a CPU to reconfigurable processing units (RPUs). Execution traces are analyzed offline to identify loops that can be valuably moved to specialized RPUs implemented in an FPGA. The loop identification process targets Megablocks, structures representing repeating patterns of instructions. An RPU is then synthesized to support the selected set of Megablocks. At runtime, the execution the Megablocks is moved transparently from the CPU to the RPU, which is reconfigured to carry out the computations of each Megablock. The implemented system prototype shows that this is a feasible and promising approach for transparently accelerating the execution of embedded programs.

The implemented system is fully functional, runtime-reconfigurable and complete. The support tools successfully generate a hardware description that combines several Megablocks with DFGs of varying depths along with the configuration and communication information required at runtime. The loose coupling between the injector and the remaining system modules allows the system to be easily adapted to other CPUs, as well as other types of memory interface. The use of the injector module avoids the need to recompile the programs, while ensuring controlled behavior modification of the CPU by altering its instruction stream.

Future work will address the support of caches memory and floating-point operations, as well as the reduction of the amount of resources needed to implement an RPU. Possible further developments include the implementation of online Megablock extraction and RPU synthesis, in order to have an autonomously adaptable embedded system.

References

[1] E. Monmasson, L. Idkhajine, M. N. Cirstea, I. Bahri, A. Tisan, and M. W. Naouar, "FPGAs in industrial control applications," *IEEE Trans. Ind. Informat.*, vol. 7, no. 2, pp. 224–243, May 2011.

[2] L. Józwiak, N. Nedjah, and M. Figueroa, "Modern development methods and tools for embedded reconfigurable systems: A survey," *VLSI J. Integr.*, vol. 43, no. 1, pp. 1–33, Jan. 2010.

[3] S. Ben Othman, A. Ben Salem, and S. Ben Saoud, "Hw acceleration for FPGA-based drive controllers," in *Proc. ISIE'10*, Jul. 4–7, 2010, pp. 196–201.

[4] J. Bispo and J. M. P. Cardoso, "On identifying and optimizing instruction sequences for dynamic compilation," in *Proc. Int. Conf. Field-Programmable Tech. (FPT'10)*, Beijing, China, Dec. 8–10, 2010, pp. 437–440.

[5] J. Bispo, N. Paulino, J. M. P. Cardoso, and J. C. Ferreira, "From instruction traces to specialized reconfigurable arrays," in *Proc. Int. Conf. Reconfigurable Comp. FPGAs (ReConFig'2011)*, Cancun, Mexico, Nov. 30–Dec. 2 2011, pp. 386–391.

[6] Xilinx Inc., Microblaze Processor Reference Guide v13.4, 2011.

[7] P. Alfke, *Xilinx Spartan-6 FPGA User Guide Lite*, 2009, ed. EE Times—Design: UBM Electronics.

[8] S. Jin *et al.*, "Design and implementation of a pipelined datapath for high-speed face detection using FPGA," *IEEE Trans. Ind. Informat.*, vol. 8, no. 1, pp. 158–167, Feb. 2012.

[9] T. Atalik *et al.*, "Multi-DSP and -FPGA-based fully-digital control system for cascaded multilevel converters used in facts applications," *IEEE Trans. Ind. Informat.*, vol. 8, no. 3, pp. 511–527, Aug. 2012.

[10] M. Gora, A. Maiti, and P. Schaumont, "A flexible design flow for software IP binding in FPGA," *IEEE Trans. Ind. Informat.*, vol. 6, no. 4, pp. 719–728, Nov. 2010.

[11] C.-H. Huang and P.-A. Hsiung, "Model-based verification and estimation framework for dynamically partially reconfigurable systems," *IEEE Trans. Ind. Informat.*, vol. 7, no. 2, pp. 287–301, May 2011.

[12] J. O. Filho, S. Masekowsky, T. Schweizer, and W. Rosenstiel, "CGADL: An architecture description language for coarse-grained reconfigurable arrays," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 17, no. 9, pp. 1247–1259, Sep. 2009.

[13] M. Sima, M. McGuire, and J. Lamoureux, "Coarse-grain reconfigurable architectures—Taxonomy-," in *Proc. IEEE Pacific Rim Conf. Communications, Computers Signal Processing (PacRim'09)*, Aug. 23–26, 2009, pp. 975–978.

[14] F. Vahid, G. Stitt, and R. Lysecky, "Warp processing: Dynamic translation of binaries to FPGA circuits," *Computer*, vol. 41, no. 7, pp. 40–46, July 2008.

[15] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *Trans. Embedded Computing Syst.*, vol. 8, no. 3, pp. 1–22, 2009.

[16] H. Noori, F. Mehdipour, K. Murakami, K. Inoue, and M. S. Zamani, "An architecture framework for an adaptive extensible processor," *J. Supercomput.*, vol. 45, no. 3, pp. 313–340, Sept. 2008.

[17] A. Mehdizadeh, B. Ghavami, M. S. Zamani, H. Pedram, and F. Mehdipour, "An efficient heterogeneous reconfigurable functional unit for an adaptive dynamic extensible processor," in *Proc. IFIP Intl. Conf. Very Large Scale Integration*, Oct. 15–17, 2007, pp. 151–156.

[18] 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 *Proc. 37th Ann. IEEE/ACM Intl. Symp. Microarch.*, 2004, pp. 30–40.

[19] N. Clark, J. Blome, M. Chu, S. Mahlke, S. Biles, and K. Flautner, "An architecture framework for transparent instruction set customization in embedded processors," in *Proc. 32nd Ann. Intl. Symp. Computer Architecture (ISCA'05)*, 2005, pp. 272–283.

[20] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proc. Conf. Design, Automation Test in Europe (DATE'08)*, Munich, Germany, 2008, pp. 1208–1213.

[21] A. C. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Run-time adaptable architectures for heterogeneous behavior embedded sys-tems," in *Proc. 4th Intl. Workshop Appl. Reconfigurable Computing (ARC'08)*, London, UK.., Mar. 26–28, 2008, pp. 111–124, LNCS 4943.

[22] J. Bispo and J. M. P. Cardoso, "On identifying segments of traces for dynamic compilation," in *Proc. Intl. Conf. Field Programmable Logic and Appl. (FPL'10)*, Milano, Italy, 2010, pp. 263–266.

[23] N. Clark, A. Hormati, and S. Mahlke, "VEAL: Virtualized execution accelerator for loops," in *Proc. 35nd Ann. Int. Symp. Computer Architecture (ISCA'08)*, Jun. 21–25, 2008, pp. 389–400.

[24] J. K. Paek, K. Choi, and J. Lee, "Binary acceleration using coarse-grained reconfigurable architecture," in *ACM SIGARCH Computer Architecture News*, 2011, vol. 38, pp. 33–39.

[25] J. V. Leeuwen, *Handbook of Theoretical Computer Science: Algorithms and Complexity*. Cambridge, MA, USA: MIT Press, 1990.

[26] M. G. Main and R. J. Lorentz, "An O(n log n) algorithm for finding all repetitions in a string," *J. Algorithms*, vol. 5, no. 3, pp. 422–432, 1984.

[27] D. Gusfield and J. Stoye, "Linear time algorithms for finding and representing all the tandem repeats in a string," *J. Comput. Syst. Sci.*, vol. 69, pp. 525–546, 2004.

[28] J. Bispo, Megablock Tool Suite—Graph Extractor v0.19. Lisboa, Portugal, 2011, ed. IST/UTL.

[29] H. S. Warren, *Hacker's Delight*. : Addison-Wesley Longman, 2002 [Online]. Available: http://www.hackersdelight.org/HDcode.htm, Code available at

[30] M. Graphics, Catapult C Synthesis 2008 [Online]. Available: http://www.mentor.com

[31] TMS320C64x Image/Video Processing Library—Programmer's Reference, Texas Instruments, 2003.

João Bispo received the Licenciature and the Ph.D. degree in computer systems and informatics from University of Algarve, Faro, Portugal, in July 2006 and Instituto Superior Técnico, Lisboa, Portugal, in July 2012, respectively.

He is currently a researcher of the SPeCS research group at the Faculty of Engineering, University of Porto, Portugal. His current research interests include embedded systems in FPGAs, co-processor hardware acceleration and compilation from high-level descriptions to embedded targets.

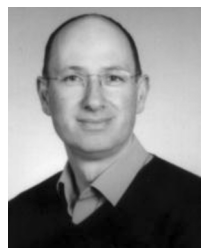
Nuno Paulino was born in Lisbon, Portugal, in 1988. He received the M.Sc. degree in electrical and computer engineering from from the Faculty of Engineering of the University of Porto, Porto, Portugal, in 2011, where he is currently pursuing the Ph.D. degree in electrical and computer engineering at the Faculty of Engineering.

His current research interests include run-time reconfigurable systems, embedded systems in FPGAs and co-processor hardware acceleration.

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

He is Associate Professor at the Department of Informatics Engineering, Faculty of Engineering of the University of Porto. Before, he was with the IST/UTL (2006–2008), a Senior Researcher at INESC-ID (2001–2009), and with the University of Algarve (1993–2006). In 2001/2002, he worked for PACT XPP Technologies, Inc., Munich, Germany. He serves as a Program Committee member for various international conferences. He has (co-)authored over 100 scientific publications (including books, journal/conference papers and patents) on subjects related to compilers, embedded systems, and reconfigurable computing. He is currently one of the scientific coordinators of the EU-funded REFLECT research project.

Dr. Cardoso is a member of IEEE Computer Society and a senior member of ACM. His research interests include compilation techniques, domain-specific languages, reconfigurable computing, and application-specific architectures.



João Canas Ferreira received the Licenciature and the Ph.D. degrees in electrical and computer engineering from the University of Porto, Porto, Portugal, in 1989 and 2001, respectively.

He is currently an Assistant Professor with the Faculty of Engineering, University of Porto, and a research manager at INESC Technology and Science. He is a member of IEEE and ACM. His research interests include dynamically reconfigurable systems, application-specific digital system architectures, and digital ECAD tools and algorithms.