Generation of Customized Accelerators for Loop Pipelining of Binary Instruction Traces

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

***Abstract*— Many embedded applications process large amounts of data using regular computational kernels, amenable to acceleration by specialized hardware coprocessors. To reduce the significant design effort, the dedicated hardware may be automatically generated, usually starting from the application's source or binary code. This paper presents a moduloscheduled loop accelerator capable of executing multiple loops and a supporting toolchain. A generation/scheduling procedure, which fully relies on MicroBlaze instruction traces, produces accelerator instances, customized in terms of functional units and interconnections. The accelerators support integer and single-precision floating-point arithmetic, and exploit instruction-level parallelism, loop pipelining, and memory access parallelism via two read/write ports. A complete implementation of the proposed architecture is evaluated in a Virtex-7 device. Augmenting a MicroBlaze processor with a tailored accelerator achieves a geometric mean speedup, over software-only execution, of 6.61× for 13 floating-point kernels from the Livermore Loops set, and of 4.08× for 11 integer kernels from Texas Instruments' IMGLIB. The proposed customized accelerators are compared with ALU-based ones. The average specialized accelerator requires only 0.47× the number of field-programmable gate array slices of an accelerator with four ALUs. A geometric mean speedup of 1.78× over a four-issue very long instruction word (without floating-point support) was obtained for the integer kernels.**

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

I. Introduction

MODERN embedded systems must meet increasingly stringent requirements on performance and functionality integration. Often, a single embedded system must be able to efficiently perform a heterogeneous set of tasks, limiting the applicability of conventional application-specific circuits. Technology scaling has been the traditional answer to these challenges, but further downscaling faces significant reliability issues and rising fabrication costs. This has driven the adoption of different computing architectures, such as multicore processors, together with the improvements of compiler technology.

For single-threaded tasks, the main solution is to exploit instruction-level parallelism (ILP), either by using superscalar or very long instruction word (VLIW) architectures. Larger performance improvements can be achieved by custom circuits that execute very well-defined tasks (such as audio filters and image processing), as only some portions of the target application are critical to performance. However, even for field-programmable gate arrays (FPGAs), custom hardware design is a lengthy process, which is unfamiliar to many embedded application developers and does not easily accommodate rapidly changing application requirements or revisions. An approach capable of autonomously adjusting the capabilities of the hardware to the needs of the currently executing application, on a per-case basis, would combine performance enhancement with reduced development effort, while adapting naturally to different models of the underlying platform. Several such approaches have relied on compilation flow extensions, proposed new hardware architectures, or both.

One way to create customized circuits is to directly translate the source code of computationally demanding functions to hardware via high-level synthesis [1]. This approach enables a large set of optimizations, but the associated complexity restricts it to offline use. In addition, source code must often be adapted and runtime information remains unexploited by the customization process.

An alternative approach is to use only binary code information. Some approaches detect frequent instruction sequences in order to generate custom instructions that augment the host processor [2]. Others map these instruction sequences onto coprocessors such as coarse grain reconfigurable arrays. All these approaches offer varying degrees of programmability and require specialized architecture-dependent tools, such as resource-constrained schedulers with connectivity awareness [3], [4]. In most cases, changes to the compilation toolchain are required, and application or binary compatibility with nonaugmented systems is compromised.

An attractive solution is to offload all customization to on-chip hardware [5], [6]. This avoids interfering with mature compilation flows, ensures binary compatibility, and eases the adoption of the technology. However, this approach requires the platform to be able to repurpose on-chip hardware at runtime, making high-performance FPGAs a natural choice. In our vision, such a system can be based on acquiring profiling information at runtime, and then using it to drive the generation of customized, coarse-grained modules. The modules

Manuscript received December 30, 2015; revised April 18, 2016; accepted May 19, 2016. Date of publication July 7, 2016; date of current version December 26, 2016. This work was supported in part by the European Regional Development Fund within the Operational Programme for Competitiveness and Internationalisation–COMPETE 2020 Programme through the Fundação para a Ciência e a Tecnologia under Project POCI-01-0145-FEDER-006961, and in part by the National Funds through the Fundação para a Ciência e a Tecnologia under Grant UID/EEA/50014/2013. The work of N. M. C. Paulino was supported by the Fundação para a Ciência e a Tecnologia under Grant SFRH/BD/80225/2011.

The authors are with the Faculty of Engineering, University of Porto, Porto 4200-645, Portugal and also with the Instituto de Engenharia de Sistemas e Computadores–Tecnologia e Ciência, Porto 4200-645, Portugal (e-mail: nmcp@inescporto.pt; jcf@fe.up.pt; jmpc@acm.org).

Digital Object Identifier 10.1109/TVLSI.2016.2573640

are later instantiated into a host accelerator, so that further execution of the application is transparently sped up. Such a system exploits exclusively runtime binary information. Our previous work [7]–[9] has explored steps toward this realization of this dynamic hardware/software codesign concept.

Currently, we automate the hardware design effort by detecting frequently executed instruction sequences called Megablocks [10] (typically corresponding to heavily used single paths through inner loops) via an offline simulation step and generating customized accelerators. The resulting runtime reconfigurable accelerators can speed up multiple loops from one or more applications, and are used transparently without any changes to the application binaries.

Although the possibility of performing extensive analysis of the source code is lost, this approach has the following advantages.

1) It is a postcompilation approach, so no specialized compilers or source-code modifications are required.
2) The application binary remains unmodified and can be used on several target devices, some of which may have no accelerator. Alternatively, in a system with multiple processors and one time-multiplexed accelerator, this would allow the system to fall back to software-only execution if the appropriate accelerator is unavailable.
3) No manual hardware design or intrusive host processor modifications are required.
4) Only performance-relevant portions of the computation are targeted (automatic hardware/software partitioning).

This paper is based on this approach, and focuses on a new accelerator architecture, and on jointly scheduling Megablocks to generate customized reconfigurable instances that efficiently support loop pipelining. One or more control and dataflow graph (CDFG) representations of the instruction sequences are moduloscheduled [11] according to an architecture template, creating a specific reconfigurable accelerator instance. The resulting accelerator implements the calculations of the CDFGs, exploiting operation parallelism and loop pipelining. We experimentally investigate the potential speedups attainable by the architecture and the resources required for deploying it on FPGAs. Preliminary results of an earlier implementation of this approach are briefly presented in [12]. The main contributions of this paper are as follows.

1) A loop accelerator architecture that supports loop pipelining and is built up from pipelined and/or multicycle functional units (FUs) corresponding to native instructions (including integer and single-precision floating-point arithmetic).
2) A scheduler that, given a set of control and dataflow graphs (CDFGs), generates a customized accelerator instance by moduloscheduling operations on pipelined or multicycle FUs.
3) Experimental evaluation of speedup for accelerator-supported execution over software-only execution for a set of floating-point and integer kernels using a fully operational hardware prototype.
4) Performance comparisons between the proposed approach and: a) fixed-resource, ALU-based accelerators; b) VLIW architectures of varying issue widths; and c) the ρ-VEX Very Long Instruction Word (VLIW) processor [13].

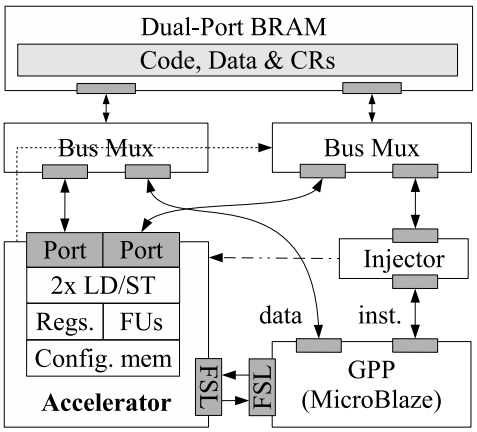


Fig. 1. System architecture overview.

Section II gives a general overview of our transparent binary acceleration approach. Section III contains a description of the accelerator architecture template and an explanation of its execution model. A description of the scheduling process and the resulting hardware is presented in Section IV. Section V presents and discusses the experimental results, while Section VI compares this paper with other related work. Finally, Section VII concludes this paper.

II. General Overview

The overall architecture and the tool flow are summarized briefly in this section. They follow the general approach of [7]–[9], with the necessary adaptations for the new scheduler and accelerator architecture.

A. System Architecture

Fig. 1 shows a complete system with a MicroBlaze general purpose processor (GPP) and our accelerator coupled as a coprocessor. Code and data are stored in a dual-port Block RAM (BRAM). The accelerator is connected to the processor by a point-to-point fast simplex link (FSL). Two bus multiplexers allow the local memory to be shared between the GPP and the accelerator. The injector module monitors the instruction address bus of the general purpose processor (GPP) and triggers the transparent migration of the execution from General Purpose Processor (GPP) to accelerator. This module can be disabled, which causes the unmodified binary to execute normally on the processor. The accelerator is composed of: two load/store units, which are present for every instantiation; FUs, whose number and type vary per instance; a set of registers, whose number and connectivity also vary per instance; and local distributed memory holding configuration words.

B. Tool Flow for Offline Generation of Accelerators

The present implementation generates accelerator instances offline from instruction traces obtained by simulation. An overview of the complete process is shown in Fig. 2.

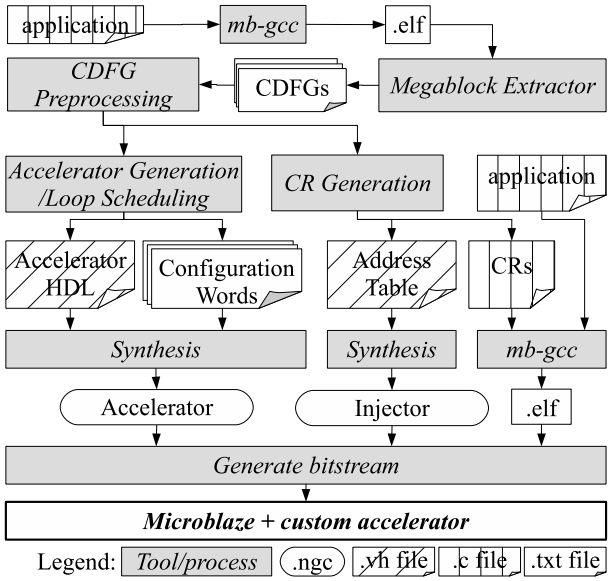


Fig. 2. Toolflow for the generation of customized loop accelerators.

The Megablock Extractor tool [14] performs a cycle-accurate MicroBlaze simulation using the binary file of the application and extracts frequently executed single-path instruction sequences called Megablocks [10]. These are further transformed into Control and Dataflow Graphs (CDFGs), exposing the latent ILP. From the selection of candidate CDFGs, the translation toolchain customizes the accelerator template, shown in Section III, according to the process explained in Section IV. Another output of this process are the communication routines (CRs) to be added to the application code, so that GPP and accelerator can communicate. The CRs are generated directly in MicroBlaze assembly code. They are linked into the application binary. No source-code modification is necessary at this stage either.

C. Transparent Migration at Runtime

Translated loops are accelerated at runtime by the following process. The injector monitors the instruction address bus of the GPP. When the address matches the starting address of one of the accelerated Megablocks, the injector replaces the fetched instruction with an absolute branch to the corresponding CR. By executing the Communication Routine (CR), the MicroBlaze sends operands from its register file to the accelerator. After receiving a known number of operands, the accelerator takes control of the local memory's ports using the bus multiplexers. The MicroBlaze becomes idle by executing a blocking FSL get instruction, which waits for the accelerator results. Thus, it is unaffected by being disconnected from the memory. Execution continues on the accelerator. When an exit condition is triggered, the accelerator relinquishes the control of the memory ports and the MicroBlaze resumes execution. The rest of the CR reads the results into the MicroBlaze's register file. The end of the routine contains a jump back to the point where the injector interfered. Execution resumes with the last iteration of the loop being executed by the GPP.

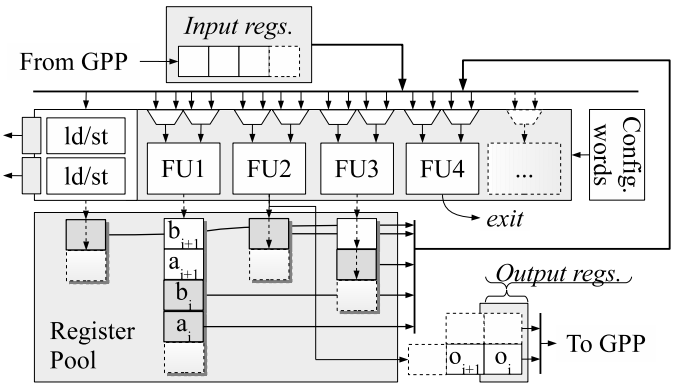


Fig. 3. Accelerator template, showing a variable row of FUs and a possible register pool structure to feedback results to the input multiplexers.

III. Customized Loop Accelerator

The proposed architecture was designed with the acceleration of data-intensive loop kernels in mind. Due to the single-path nature of the Megablocks, the most appropriate loop bodies are the ones with little conditional control flow. The architecture exploits instruction level parallelism (ILP) (intraiteration parallelism), and pipelines loop iterations (interiteration parallelism).

Generating fully customized accelerators has two notable advantages: 1) the amount of required resources (e.g. Functional Units (FUs), interconnects) is reduced to the minimum necessary to execute the chosen loops with the desired initiation interval (II) and 2) moduloscheduling loops onto an architecture with no fixed-resource or interconnect limitations ensure that the minimum II is always achievable. In our case, the only structural limitation is the availability of two memory ports.

A. Accelerator Architecture

The accelerator architecture template is shown in Fig. 3. It includes: input and output registers; a set of 32-bit FUs; two load/store ports present in every accelerator instance; multiplexers to route FU inputs; a register pool of 32-bit registers to hold operands/results; and a configuration memory.

Specialization of this template involves determining: the number of input and output registers, the number and type of FUs, the number of pool registers, and multiplexer connectivity. In addition, the appropriate configuration words are generated.

The input and output registers hold data exchanged with the host processor. Input registers are filled during CR execution, are read only, and can all be fed into any FUs. They hold values valid for the first iteration and values that remain constant throughout all iterations. In a given configuration, each output register is fed by a single FU.

Since moduloscheduling overlaps loop iterations, an FU may produce several values corresponding to the same CDFG node before an iteration is complete. To address this, each output register is preceded by a custom-depth first-input–first-output (FIFO).

For the same reason, each FU drives a chain of registers, which hold the output values of the CDFG operations, and the

FU executes. The chain length is determined during scheduling by computing how long each value needs to live until it is used in all downstream operations. This is exemplified by FU1 in Fig. 3, where results produced by CDFG nodes a and b for iterations $i + 1$ and i are stored in the register pool.

The inputs to the multiplexers are determined on the basis of which pool register holds each computed value at every clock cycle. This is deterministic, since the schedule is static. The required values for each FU input are fetched from the appropriate pool register (shown in dark gray in Fig. 3) according to the CDFG operations to be executed. Multiplexer inputs may also be connected to any input register or to constant values specified at customization time.

The accelerator supports integer and single-precision floating-point arithmetic, comparison operations, and bitwise logical operations. Other operations include conversion from floating point to integer and vice-versa, and a set of units to evaluate termination conditions. Each FU implements one operation, except one which implements both floating-point addition and subtraction. All FUs are pipelined, except for nonconstant integer division and floating-point division. Supporting pipelined FUs avoids the need to raise the initiation interval (II) by effectively increasing resource availability. All integer units have a latency of 1 clock cycle, except the division FU (35 clock cycles). A specialized integer division module provides division by a constant with a latency of 3 clock cycles (via reciprocal multiplication). Like the MicroBlaze processor, the floating-point units (FPUs) do not support denormalized operands or issue denormalized results. The floating-point addition, multiplication, and division units have the latencies of 4, 3, and 32 clock cycles, respectively.

Memory access patterns can be arbitrary, since the loop operations that generate access addresses are also executed on the accelerator. For this implementation, the scheduler considers that the load/store units have a memory access latency of 2 clock cycles, since on-chip memories (BRAMs) are used. In the general case of variable access latency, the accelerator would stall if an access exceeded the expected time. The accelerator design and scheduling allow for pipelining accesses without halting execution.

B. Execution Model

The accelerator is idle until it receives one command word from the injector. The command determines which configuration words to read, which FU will drive each output FIFO, and how many operands to expect from the GPP.

Configuration words define the accelerator's operation for a single cycle: which FUs are active, the multiplexer controls, which pool and output registers will be written to, and when execution is concluded. The length of a configuration word varies per instance, depending on the number of FUs, multiplexer widths, and number of pool registers. One word corresponds to one time step of the respective moduloschedule.

Unlike a VLIW, executing operations in multicycle FUs, such as the division unit, does not halt execution of the other units. For instance, for the nonpipelined integer division with a latency of 31 clock cycles, configuration words continue to be read while that FU is busy. The static schedule considers the FU latency to determine when the result is ready. If the FUs are also pipelined, (e.g., the floating-point adder), both aspects are exploited: the FU can be enabled every cycle and the scheduler knows in which step each result is produced.

In general, operations from the same iteration and without control or data dependences are issued in the same clock cycle (ILP). It is also possible to simultaneously issue operations from two or more successive iterations, depending on the data and control dependences between them (loop pipelining). In this way, a new iteration is initiated (and simultaneously one iteration is completed) in a number of clock cycles lower than the critical path length of the respective CDFG.

The number of iterations is needed neither at synthesis time nor prior to the start of execution. The termination conditions of the executing loop (i.e., loop exits) are evaluated in every iteration. Whenever an iteration triggers an exit, the iteration is discarded. No new iterations are initiated, and the previous ongoing iterations are completed. The output registers contain a full set of values, which correspond to MicroBlaze register file contents. The GPP executes the remainder of the CR to retrieve them.

IV. Accelerator Customization and Loop Scheduling

When performing moduloscheduling [11] for fixed architectures, there are resource and temporal restrictions to consider: 1) given all types of operations in a loop, the target architecture must contain at least one FU capable of implementing each one and 2) operation parallelism and Initiation Interval (II) still depend on the spatial and temporal availability of FUs, which determine performance. In these situations, when a moduloscheduler cannot schedule a loop with a given (minimum) II, the II is increased until scheduling becomes feasible. Increasing the II makes it possible to schedule a loop onto the minimum set of resources, but leads to decreased performance, especially when the minimum possible II is low (e.g., increasing the II from 1 clock cycle to 2 approximately halves the performance). Since the loops we have found typically have low IIs, we instead add the required FUs to the accelerator during scheduling.

A. Scheduling and FU Allocation

Fig. 4 shows an example of the type of CDFG that our scheduler processes to generate a custom accelerator. The nodes represent the GPP instructions, edges represent the data flow between nodes, and the inputs (top) and outputs (bottom) represent the GPP registers. Dotted edges represent the input register values that are read only in the first iteration. The graph itself represents one iteration of the respective Megablock.

The first step of scheduling is to find the II, which can be determined by backward data edges (i.e., data flow across iterations), by resource restrictions (in our case, the two memory ports) or control edges. It is necessary to consider control edges, since the number of iterations to execute is arbitrary. That is, a new iteration can only begin after all exit conditions of the current one are evaluated to false. For this example,

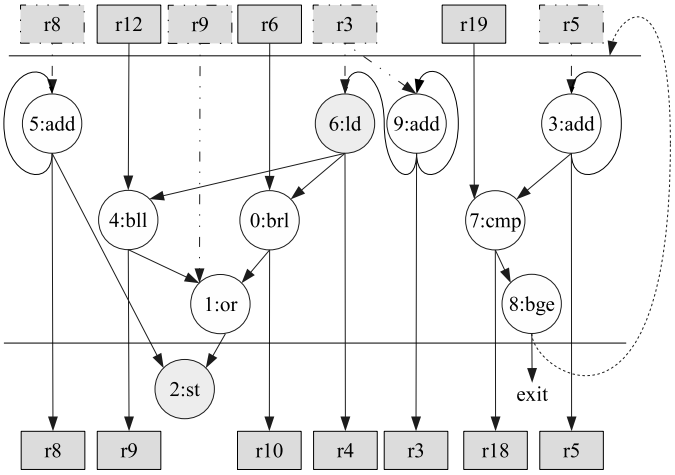


Fig. 4. Example CDFG, showing operations and a cyclical control dependence, which determines the II.

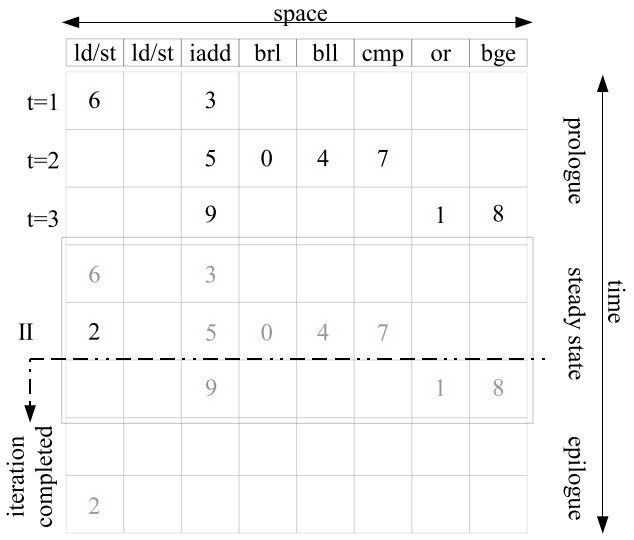


Fig. 5. Modulo schedule for an example loop CDFG.

the bge node sets an II of three clock cycles, since all nodes are implemented by a single-cycle FU.

At the start of the process, no accelerator architectural aspects are defined except the two memory ports. As nodes are scheduled, FUs are added if necessary. Connectivity is assumed to be unlimited. A schedule is composed of prolog, steady state, and epilog. Fig. 5 shows the complete schedule for the nodes of Fig. 4, placed temporally along the vertical axis and spatially along the horizontal axis.

Nodes are list scheduled individually in topological order. For every node n, the earliest (e_c) and the latest (l_c) possible schedule times are computed. The calculation of e_c is based on its predecessor nodes (or input registers), while the l_c is typically unbound. A bounded l_c is only relevant for nodes with outgoing backward edges, since scheduling them too late relative to the predecessor nodes in the cycle would violate the II. In this example, nodes 3, 7, and 8 have no slack; nodes, 6, 0, 4, 1, and 2 can be delayed indefinitely; node 9 can be delayed no later than $t_6 + 3$, where t_6 is the scheduled time for node 6; and node 5 cannot be scheduled past node 2. All nodes are scheduled as early as possible.

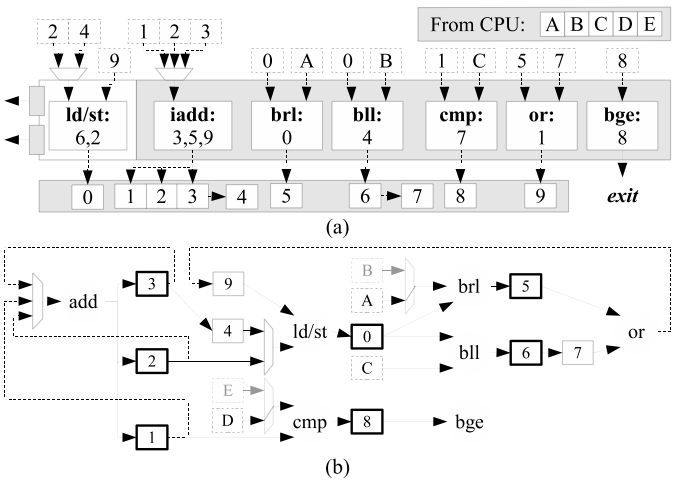


Fig. 6. (a) Operation assignments and register connections for one loop schedule and (b) actual generated datapath for two (similar) CDFGs.

In Fig. 5, node 6 was scheduled first at the earliest possible time, $t = 1$. Since the II is 3, all time slots i for which i mod II $= 1$ are occupied by future executions of node 6 (shown in gray). When scheduling node 3, an add FU is added, and node 3 is scheduled at the earliest time which does not violate the II, $t = 1$. Nodes 5 and 9 can also execute on this FU, since both can be delayed until $t = 3$ at most. If there was another add node and it could not be delayed to a later time, a new FU would have to be added.

The steady state of the schedule has II time steps. It starts at a time t_s, where t_s mod II $= 1$, and contains the time step when the first iteration, started in the prolog, completes. In this case, this happens after only one repetition of the scheduled nodes at $t = 5$. All previous time steps (1–3) belong to the prolog, and time steps after this window belong (7 and 8) to the epilog. The time steps in the epilog only contain operations of outstanding iterations initiated in the steady state.

Additional CDFGs could now be scheduled onto the existing array. As more loops are scheduled, less FUs are added to the accelerator, since FU availability increases. However, there may be an associated increase in the size of the input multiplexers, which depends on the connections between operations. When there is more than one available FU to place a new node, the scheduler attempts to reduce resource consumption by choosing the FU, which will result in the least added inter-FU connectivity (based on the nodes of previously scheduled graphs).

B. Architecture Specification

The later processing stages are concerned with the definition of the register pool, input multiplexers, output FIFOs, and configuration words.

First, the scheduler computes when each value is produced and for how long it must be stored. This information is used to define one chain of registers per FU output. A register is added to the chain whenever a node produces a value before the preceding value is consumed. When the new value is produced, the existing values are shifted down the chain. However, a value in the middle of a chain may be consumed

before the values produced at previous time steps. By having an individual write-enable per register, it is possible to shift data only up until the first expired (i.e., consumed) datum, which allows for shortening the chain lengths. The (simplified) accelerator generated for the CDFG in Fig. 4 according to the schedule in Fig. 5 is shown in Fig. 6(a). The iadd FU requires four registers to hold results, as most will only be read in the following iteration. The bll FU requires two, because the or node will consume a value from the input register for the first iteration, and only afterward, it fetches the values produced by the bll node. In this case, two registers are required to synchronize the data across iterations.

Given the register chains, it is known where each value resides at every time step. This information is used to specify the input multiplexers that route the required values from the appropriate registers. For simplicity, the connectivity is represented by the numbered boxes in Fig. 6(a). If a given FU input receives data from only one pool register, no multiplexer is used. Literal constant inputs are omitted for clarity.

Despite the appearance of a horizontal type layout, the dedicated connectivity between FUs effectively implements circuits with distinct structures. The example in Fig. 6 was generated from two graphs: one shown in Fig. 4 and a second very similar graph. In Fig. 6(b), most of the structure is used for either loop, but some instantiated components are only required for one of them (shown in gray). Registers with a bold outline feed the output registers of the accelerator.

The last step is the generation of configuration words. A configuration word contains FU enable bits, hot-bit encoded multiplexer control bits, and register write-enable bits.

One word is generated for each scheduled time step. The word generation process is simplified by considering only the execution of the first iteration (i.e., the solid black nodes in Fig. 5). The process iterates through every FU, starting from time step $t = 1$ up until when the initial iteration ends, $t = 5$ for this example. If there is a scheduled node, the enable bit is set and the multiplexer bits are set based on the required inputs. The write-enable bits in the respective pool registers are set later according to the FU latency. The resulting set of words, which represents a single iteration, can be used to generate the full configuration word sequence from prolog to epilog by repeating it every II time steps.

Each word also has an address update value used to cycle through the words that belong to the steady-state phase. When an exit condition is triggered, this field is ignored and execution continues through the epilog. The configuration memory holds one sequence of configuration words per scheduled loop.

V. Experimental Evaluation

A fully working hardware prototype of the system was used to perform an experimental evaluation of performance improvements and resource requirements of the customized loop accelerators produced using the proposed approach.

A. Experimental Setup

The proposed accelerator architecture was evaluated with both floating-point and integer kernels. The floating-point kernels are a subset of the Livermore Loops [15]. The integer kernels are from the IMGLIB library [16]. The selected kernels were those with little control flow in the innermost loops. The Livermore Loops kernels operate on single-precision floating-point data. The kernels from IMGLIB operate on integer data with 8, 16, or 32 bit widths.

1) Hardware Setup: Section II described the architecture of the prototype used for the experimental evaluation. There is an additional timer/counter module for execution time measurements and a serial port module, both attached to the MicroBlaze's peripheral bus. The test bed was a Xilinx VC707 board equipped with a Virtex-7 xc7vx485 FPGA. We used ISE Design Suite 14.7 for synthesis and bitstream generation. The synthesis effort policy was set to speed, and the placement and routing effort was set to high. All measurements were run at 110 MHz for both GPP and accelerators.

2) Software Setup: Each kernel from the Livermore Loops and IMGLIB sets is enclosed within a function call. To easily compile, execute, and measure execution times of each, we developed a test harness which can be compiled along with a single C file containing all kernels. The harness' structure is a significantly modified version of the CoreMark code for the Livermore Loops.

The desktop version of the harness accepts parameters that determine which kernels to call, the amount of data to process (N), and the number of times to repeat the kernel code (L). The input data to be processed are generated at runtime by a pseudorandom generator and placed into arrays allocated on the heap. The generator seed is also a variable parameter. Although this implementation is limited to local memories, it is still viable depending on the application, considering that the high-end devices of the Virtex-7 family contain up to 8.46 MB of internal memory [17]. The heap used in these experiments had a total size of 10.5 kB.

The purpose of executing the harness on a desktop machine is to generate reference results for the benchmarks. A checksum of the results is compiled into the harness when targeting the MicroBlaze, to verify the correct functionality of the generated accelerators. The reference data include the used call parameters, so that the embedded versions call the same kernels with the same parameters. The MicroBlaze version reads the execution time (and other information) of the loop accelerator, checks the results, and prints the time measurements.

3) Experimental Procedure: To extract loop traces, we compiled a MicroBlaze version of the harness which executed all kernels. We used mb-gcc 4.6.4 and enabled the use of floating-point, integer multiplication and division, barrel shifter, and pattern-compare operations. Processing the resulting binary file with the Megablock Extractor produced the traces and CDFGs for all kernels.

We then measured the speedups obtained by customized loop accelerators versus a single MicroBlaze (Section V-B), evaluated the cost of a multiloop accelerator versus individual loop accelerators (Section V-C), studied the performance and resource cost of fixed-resource, ALU-based, accelerators (Section V-D), and compared our architecture with several VLIW models (Section V-E).

TABLE I
ACCELERATOR CHARACTERISTICS AND ACHIEVED SPEEDUPS

Id	kernel	II	IPC	# FUs	# RP Regs.	Speedup N=1024	Speedup N=4096
f1	cholesky	3.0	6.7	11	30	7.25	9.80
f2	diffpredict	10.0	4.4	10	52	6.78	7.86
f3	glinrecurrence	3.0	4.3	9	20	3.65	4.58
f4	hydro	3.0	5.7	9	28	12.47	12.84
f5	hydro2d[1]	16.3	3.7	12	71	6.60	(6.60)*
f6	hydro2dimp	6.0	6.2	9	38	11.18	(11.18)*
f7	innerprod (fp)	4.0	2.5	6	9	4.43	4.48
f8	intpredict	10.0	4.1	9	54	7.99	9.46
f9	linrec[2]	11.0	1.0	9	12	2.30	2.31
f10	matmul (fp)	3.0	5.0	10	25	4.84	7.06
f11	pic1d[1]	5.5	3.7	12	30	1.47	(1.47)*
f12	statefrag	5.0	8.4	13	49	18.47	18.98
f13	tridiag	3.0	4.0	8	21	6.81	6.93
	mean	6.4	4.6	9.8	33.8	5.96	6.61
i1	quantize	3.0	3.3	7	15	3.22	3.99
i2	conv3x3	10.0	6.4	12	39	6.75	7.01
i3	perimeter	3.0	7.0	13	34	7.35	7.81
i4	boundary	4.0	6.0	12	25	2.92	3.59
i5	sad16x16	2.0	6.5	9	13	2.31	2.31
i6	mad16x16	2.0	6.5	9	13	2.30	2.30
i7	sobel	5.0	8.0	14	46	7.99	8.14
i8	dilate	20.0	7.1	13	77	5.31	5.23
i9	erode	22.0	6.5	14	72	5.43	5.24
i10	innerprod (int)	3.0	4.0	6	8	3.93	3.98
i11	matmul (int)	3.0	5.0	8	23	3.82	5.44
	mean	7.0	6.0	10.6	33.2	4.27	4.61
	total mean	6.7	5.2	10.2	33.5	6.07	5.61

[1]Three loops accelerated; [2]Two loops accelerated. For all other cases, one loop was accelerated. (*) Values taken from run with $N = 1024$.

B. Customized Loop Accelerator Versus MicroBlaze Processor

Table I shows the characteristics of the accelerators and the corresponding kernel speedups. The results for the floating-point set are on the top half and the ones for the integer kernels on the bottom half. The last two integer kernels (innerprod and matmul) are direct conversions of the corresponding floating-point versions, and not from IMGLIB. The averages shown are geometric for speedup and arithmetic for the remaining metrics. The third column shows the II of the schedules. For the benchmarks that have more than one accelerated loop, the average II is given. The fourth column shows the average executed instructions per clock cycle (IPC), which measures the parallelism exploited by the accelerator.

The next two columns show the number of FUs and the number of registers in the register pool. They are the indicative of the complexity of the accelerator. The last two columns show the speedups obtained when executing each kernel $L = 1000$ times for two different amounts of data ($N = 1024$ and $N = 4096$). All speedup results include the overhead of sending data from the MicroBlaze processor to the accelerator and retrieving the results. Benchmarks *f5*, *f6*, and *f11* were run only for $N = 1024$, because the system does not have sufficient on-chip memory to support the $N = 4096$ case. We used the speedup values for $N = 1024$ as a worst case estimate for the $N = 4096$ case.

1) Speedup: As shown in Table I, we extracted for each kernel only one viable loop trace for acceleration, except for *f5*, *f9*, and *f11*. On average, there are 34 instructions in each trace. The floating-point kernels have on average less instructions than the integer ones: 26 versus 45. However, the average of the integer kernels is inflated by *i8* and *i9*, which have 142 instructions each, the maximum value for all CDFGs. The CDFGs of the floating-point kernels have an average of 7.2 floating-point instructions (27.6% of the total).

The MicroBlaze executes 0.38 instructions per clock cycle (IPC) for the floating-point set and 0.66 IPC for the integer set. Table I shows that the accelerators achieve significantly higher values of IPC, confirming this approach is able to exploit the latent Instruction Level Parallelism (ILP). Note that the average speedup for the integer kernels is lower than that of the floating-point set, although the relationship between IPC values is reversed. The reason for this behavior is that the MicroBlaze requires more cycles to execute floating-point instructions. As an example, consider the matmul kernel, which contains 15 binary instructions in both integer and floating-point cases. One iteration of the integer kernel requires 25 clock cycles on the MicroBlaze, but the floating-point version requires 33 cycles. However, both benchmarks execute on accelerators with the same II, resulting in better speedup for the floating-point version. Execution on the accelerator effectively mitigates the latency of floating-point operations due to the fully pipelined FPUs.

An additional factor that affects speedup is the overhead introduced by the call to the accelerator. If the number of iterations performed per call is low, the fixed (per call) extra time required for communication between host processor and accelerator may offset the acceleration gains. The impact of this factor can be seen by comparing the geometric mean speedup for the two values of N. For some benchmarks, (e.g., *i7* and *i4*), the speedup increases with N, because the constant call overhead is amortized over a larger number of iterations. For other benchmarks (e.g., $f9$, $f12$, $f13$, $i5$, and $i6$), the effect of increasing the value of N is negligible, because the communication time is small compared with the processing time ($f9$, $f12$, and $f13$) or because the communication time also scales with N ($i5$ and $i6$).

In the latter cases, the accelerated loops always perform 16 iterations per call. For these two cases, scaling N does not decrease overhead, since the inner loop processes only 16 data items at a time. The accelerator will be invoked more times with the same overhead per call. This shows a limitation of the proposed approach. A given loop path may be the most frequently executed portion of an application, but, despite a very high number of iterations, it might not be a good target for acceleration in our implementation, if each loop occurrence iterates a small number of times.

The highest overheads occur for *f3*, *i5*, and *i6*. The overheads are 17% for the former case and 51% for the latter two (for $N = 4096$). For *f3*, each accelerator call performs an average of 63 iterations; for *i5* and *i6*, only 15 iterations are

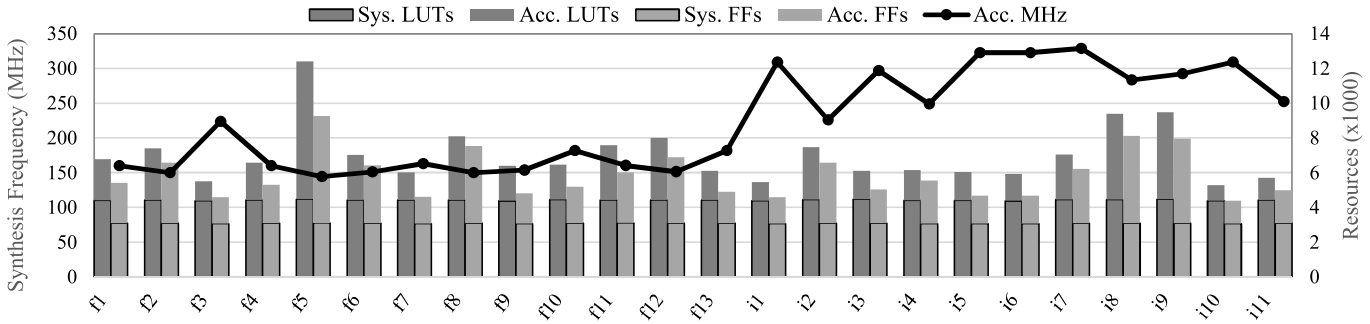


Fig. 7. Resource usage for complete generated systems and accelerator clock frequency reported by the synthesis tool.

executed per call. In contrast, the average overhead is 8% and the average number of iterations per call is 1430.

2) Resources and Operating Frequency: The scheduling approach discussed here favors performance, since it does not impose resource limitations on the architecture beyond the restriction to two memory ports. This choice has impact on the FPGA resource requirements due to more FUs and larger configuration words. Fig. 7 shows the FPGA resources used by each system (after placement and routing). The stacked bars represent the resources used by each accelerator (upper half) and by the remaining system components (lower half). The clock frequency of the accelerator reported after logical synthesis is also shown. In addition, the system uses 32 Block RAMs (BRAMs) of 36 kb each for program code and data, totaling 128 kB.

Using the resource requirements of a MicroBlaze processor (without caches or memory management unit but with an FPU) as a measurement unit, we can say that the average accelerator requires approximately 1.13× more lookup tables (LUTs) and 1.83× more flip-flops (FFs). The average number of required LUTs and Flip Flops (FFs) for the entire system, accelerator included, is 7013 and 5840, respectively.

As a general rule, the accelerators for the integer loops require less resources and also achieve higher operating frequencies than the ones with floating-point support. For instance, the accelerators for the integer and floating-point versions of innerprod require 1259 and 2021 Lookup Tables (LUTs), respectively. The average number of FUs required for the floating-point and integer sets is 9.8 and 10.6, respectively. The average size of the register pool is similar, approximately 33 registers.

The increased complexity of the FPUs justifies the lower synthesis clock frequency for the floating-point set, whose average is 164 MHz, which is significantly lower than the average clock frequency of 290 MHz for the integer-only accelerators. In all cases, the critical path in the accelerators with floating-point operations includes the fadd FU; the sole exception is benchmark *fp3*, where the critical path is determined by the fmul unit (since there are no floating-point additions). Floating-point FUs make up 24% of all FUs per accelerator. For the integer set, the lower frequencies for *i2*, *i4*, and *i11* are due to critical paths between the instruction memory, multiplexers, and integer multiplication FU. Systems *i1*, *i3*, and *i10* also contain an integer multiplier, but do not suffer the same decrease in frequency, since the inputs of the multiplication FU receive operands only from one other FU each and, therefore, have no input multiplexer.

3) Speedup for Maximum Operating Frequency Scenario: We determined that for our test platform, the maximum operating frequency of a system containing a single MicroBlaze (with an FPU) and local instruction memories is 200 MHz. The instantiation of the accelerator and additional hardware required to support the migration approach leads to the decrease in operating frequency, which depends on the benchmark. We determined the maximum operating frequency of each single benchmark: the average is 158.4 MHz for the floating-point set, and 165.1 MHz for the integer set.

Given this, the geometric mean speedup is 5.22× for the former case and 3.81× for the latter case, when comparing software-only execution and accelerated execution, each at their respective maximum operating frequencies.

This paper presents speedups based on the same operating frequency for both systems, since we aim to demonstrate the performance improvements that can be attained by a joint architecture specification and operation scheduling approach that allows for the thorough exploitation of parallelism. Moreover, the decrease in operating frequency does not occur due to the accelerator instance, but instead due to the injector module. In all implementations, the critical path is between the MicroBlaze's instruction port and the local memory. An improvement of this single aspect would address the reduction of operating frequency which occurs for all cases.

4) Power Consumption: We determined the power consumption of the accelerator-based system by retrieving actual power measurements from our evaluation board using Texas Instruments' Fusion Digital Power Designer [18]. We executed the kernels from the floating-point set, with $N = 1024$ and $L = 10\,000$. Setting L to this value increases the runtime, thus capturing more data points and filtering measurement noise.

Measuring the current of the 1 V-rail that powers the FPGA core, we find that the average power consumption for a system containing only a MicroBlaze, local memories, and other required modules was 0.45 W. For the accelerator-based system, the average power consumption was 0.57 W. However, due to the reduced execution time, the total energy required to execute each kernel was, on average, 8.24 J, while the energy required by a MicroBlaze-only system was 31.44 J.

C. Performance and Cost of Multiloop Support

Multiloop accelerators can enhance the execution of several loops (from the same application or from a set of related

TABLE II
MULTILOOP ACCELERATOR CHARACTERISTICS

kernel	# loops	# FUs	# RP Regs.	MRT Occupancy (%)	Frequency (MHz)
f_2_8	2	11	96	48	144.7 (-2.1%)
f_3_9	3	11	25	21	150.9 (-0.0%)
f_4_5_6	5	13	119	41	139.7 (-3.9%)
f_7_10	2	10	28	40	165.4 (-1.5%)
i_5_6	2	8	24	68	309.8 (-8.2%)
i_8_9	2	18	99	35	254.9 (-13.8%)
i_10_11	2	9	13	53	230.7 (-7.3%)
mean	2.57	11	58	44	199.5 (-5.2%)

This table refers to systems with accelerators for loops from several kernels. The benchmark names indicate which kernels are supported.

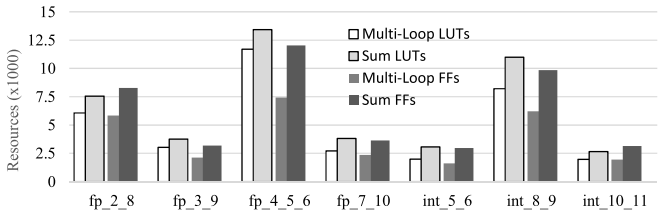


Fig. 8. Resource requirements for multiloop accelerators versus sum of resources for individual loop accelerators.

applications) and also result in resource savings when compared with a group of individual single-loop accelerators. In the results analyzed in Section V-B, there are already three systems with accelerators that support more than one loop (*f5*, *f9*, and *f11*). To further evaluate multiloop support, we chose the combinations of kernels and generated systems with accelerators targeting them. Loop combinations were chosen based on the similarity of calculations performed as well as number of instructions (e.g., creating a single accelerator for dilate and erode). Table II shows the accelerator characteristics for these cases. Since the II of each CDFG remains the same as in the single-loop case, so does the individual speedup.

The resource requirements for the multiloop accelerators are compared with the sum of resources of individual accelerators for the same loops in Fig. 8. Each accelerator requires on average only 76% and 63% of the sum of LUTs and FFs of the individual accelerators, respectively.

The average synthesis frequency of the multiloop cases is only 6% lower than the worst case of the respective single-loop accelerators. As an example of the frequency decrease due to increased connectivity, consider system *f_7_10*. The resource requirements of *f_7_10* are slightly superior to those of *f10*, the largest of the two single-loop accelerators. However, the lowest synthesis frequency occurs for *f7*, due to a path between a pool register and the input of an fadd FU. The same path is found in the accelerator *f_7_10*, but now, it also includes a 3-to-1 multiplexer, leading to a slight decrease in clock frequency.

D. Comparison With Fixed-Resource Scenarios

So far, we presented the results for fully customized accelerators with single-operation FUs. In our approach, this maximized performance, by allowing execution at the lowest possible II. This section compares this approach, referred herein as *a1*, with three other scenarios shown in Table III (*b1*, *b2*, and *b3*) based on scheduling loops onto an accelerator with a fixed number of units. We adjusted the scheduler, so that new FUs are not allocated. Instead, as in typical moduloscheduling approaches, the II is increased until scheduling is possible.

TABLE III
ACCELERATOR GENERATION SCENARIOS

Scenario	Available units
a1	Any number of resources of any supported type
b1	2 ALUs + 1 Multiplier + 1 Branch Unit (+ 1 FPU)
b2	4 ALUs + 1 Multiplier + 1 Branch Unit (+ 1 FPU)
b3	8 ALUs + 1 Multiplier + 1 Branch Unit (+ 1 FPU)

The ALUs used in the evaluation are built from most of the individual integer FUs with the required additional control logic. Supported operations include all integer arithmetic (except division and multiplication), logic, and comparison operations. In order to prevent the logic synthesis tools from optimizing the ALU logic on a per-instance basis (due to constant propagation of the configuration bits or inputs feeding each ALU), we generated a black box instance for the ALU. Each ALU has a fixed cost of 641 LUTs.

The hardware for the new scenarios has a single branch unit (which evaluates all types of exit conditions) and a single integer multiplier, also instantiated as black boxes. For the benchmarks of the Livermore Loops, a single fully fledged FPU is also added, which can execute all floating-point arithmetic operations, comparisons, and float/integer conversions. The ALU has a latency of 1 clock cycle, and the latency of the FPU depends on the issued operation, but it is still possible to pipeline operations. The Floating-Point Unit (FPU) has 1455 LUTs and 526 FFs. Although the number of units is fixed, the interconnections between them are still specialized by the scheduler based on the flow of data between operations.

Fig. 9 shows the speedups for these scenarios. Note that the speedups for the accelerators in scenario *b3* equal those of scenario *a1* for all integer cases. The CDFGs detected by our approach can be executed at the minimum possible II with eight ALUs. In most cases, four ALUs are enough, meaning our fully customized accelerators perform equivalently to generalized accelerators with four to eight ALUs. Only between *b1* and *b2* is there a noticeable difference: the average IIs are 19.6 and 10.3, respectively. Even so, an accelerator with two ALUs achieves a geometric mean speedup of 2.08× for the integer benchmarks.

However, the accelerators in these scenarios contain only one FPU. This means that the speedup decreases for loops with floating-point operations. However, the effect is not uniform. For 6 out of the 13 floating-point benchmarks, the speedup decreases approximately by half (on average); for the other kernels, the decrease is marginal. The average II for *b1*, *b2*, and *b3* increases to about 17 (more than doubling the II of 7.2 of scenario *a1*). The largest speedup decrease occurs for *f12*. This benchmark has 16 floating-point operations, which are scheduled onto four single-operation floating-point FUs in scenario *a1*. Using only one FPU predictably decreases the achieved speedup by approximately four times, from 18.9× to 4.8× regardless of the number of ALUs. Although the accelerator for *f5* contains five floating-point

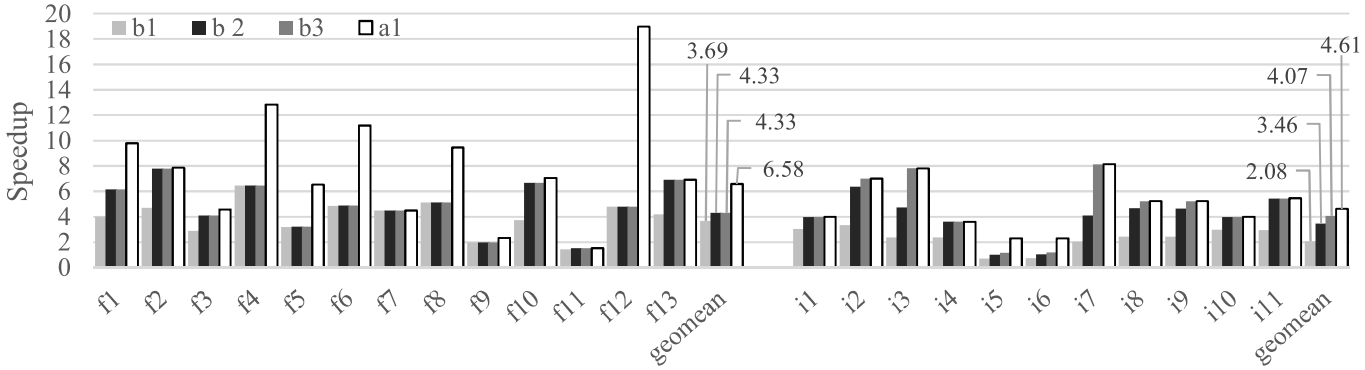


Fig. 9. Kernel speedups using several types of accelerators, versus a single MicroBlaze processor.

TABLE IV
AVERAGE COST OF ACCELERATORS PER SCENARIO

	floating-point			integer		
Scenario	# LUTs	# FFs	# Slices	# LUTs	# FFs	# Slices
a1 (avg)	1.23	1.90	1.28	1.01	1.74	0.93
b1	2.12	1.95	2.12	1.48	1.64	1.42
b2	2.74	2.06	2.70	2.09	1.76	2.01
b3	3.87	2.08	3.79	3.24	1.85	3.26

Values normalized to a single MicroBlaze with 2291 LUTs, 1503 FFs and 860 slices.

FUs in scenario *a1*, the speedup only decreases by half for all other scenarios. This is because the accelerator for this case supports three CDFGs, only one of which uses all five floating-point FUs.

In scenario *a1*, an average of 2.3 floating-point FUs is instantiated to achieve minimum II. This means that for a fixed-resource accelerator, at least two fully fledged FPUs would be required to avoid increasing the II in most cases. An accelerator with four ALUs and two FPUs would need approximately 7700 LUTs and 3700 FFs. In comparison, for scenario *a1*, the accelerators of the floating-point set require an average of 2829 LUTs and 2863 FFs (36.7% and 77.4% less, respectively).

Table IV shows the average required resources for all scenarios, normalized to the resource requirements of a single MicroBlaze. Assuming that the number of slices is a good measure of area on the device, the accelerators in scenario *a1* are roughly 0.62×, 0.47×, and 0.32× smaller than those of cases *b1*, *b2*, and *b3*. The number of required FFs varies little, as the same amount of data needs to be moved between FUs regardless of their type. For the floating-point benchmarks, the reduction in the number of LUTs is more pronounced due to the higher cost of each FPU.

In general, the proposed customized accelerators with single-function units perform no worse than the largest ALU/FPU-based accelerators (which still have customized interconnect), while being significantly smaller. They significantly outperform fixed-resource accelerators while requiring fewer resources.

E. Comparison With VLIW Execution

The MicroBlaze processor is a single-issue processor, and therefore, it is not surprising that considerable speedups can be obtained by exploiting, even if only partially, any latent ILP. However, the FPGA hardware resources may also be used to implement a VLIW processor. This section compares the proposed approach with the well-known embedded VLIW architecture VEX [19] and with ρ-VEX, an implementation (and expansion) of that architecture for FPGAs [13].

The VEX tool suite is a compilation chain, which targets a generic VLIW processor architecture. Architecture parameters include: issue width, the number of processing units, and their latency and type. This is used to generate compiled code simulators that run on a desktop machine.

The ρ-VEX architecture is a synthesis-time parameterizable VLIW processor, which implements the VEX instruction set. We used release 3.3 of ρ-VEX, which includes the open-source ρ-VEX processor itself and also a build of the gcc toolchain. It is also possible to use the VEX compiler for compilation and the ρ-VEX gcc toolchain for assembly and linking.

We used these tools to measure benchmark performance in two situations: 1) simulated execution on three different variants of the VEX architecture model and 2) simulated execution of a four-issue version of the ρ-VEX processor in ModelSim (a Hardware Description Language simulator). Since the VEX architecture has no native support for floating-point operations, the comparisons include only the integer benchmarks.

1) Simulated VLIW Models: The test harness was compiled with the VEX compiler for three different architecture variants (*r1*, *r2*, and *r3*). We then determine the number of clock cycles required to execute the entire kernel function call. To evaluate execution on the ρ-VEX processor, we generate one binary file per kernel, and simulate the system with ModelSim [at the register transfer level (RTL)]. These executables were generated by using the VEX compiler and followed by the ρ-VEX assembler and linker. When measuring the execution cycles for the ρ-VEX, cache stall cycles were not counted, so that the comparison with the systems that use on-chip BRAM memory is fair.

Table V shows the parameters of each VEX variant. The parameter values were chosen to match those of the ρ-VEX architecture, so that we approximately simulate three variants of the ρ-VEX processor which differ only in issue width and number of ALUs. These variants are compared with two versions shown in Section V-D, *a1* and *b2*.

2) Performance Comparison: The speedups for these scenarios versus a single MicroBlaze are shown in Fig. 10.

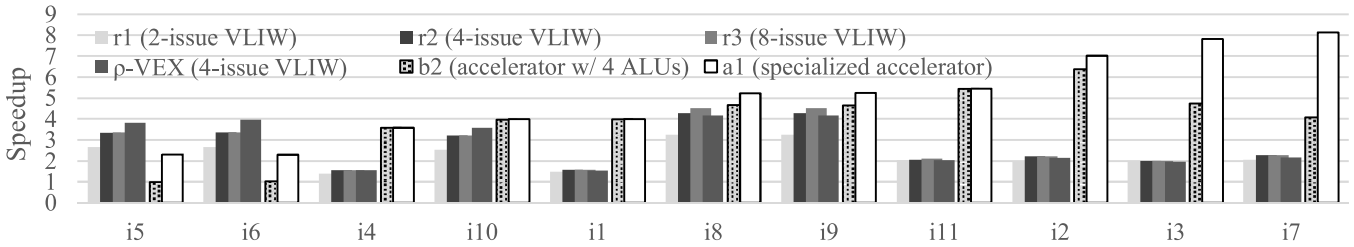


Fig. 10. Kernel speedups for different VLIW models and two of our accelerator scenarios, versus a single MicroBlazer processor.

TABLE V
VEX SIMULATOR MODELS AND ACCELERATOR COMPARISON

Scenario	r1	r2	r3	a1	b2
Issue width	2	4	8	N/A	N/A
#ALUs/FUs	2	4	8	Variable	4
#MULs	2 (16×32)			Variable	1 (32×32)
#Registers	64			Variable	
Memory access	1 Load/Store Unit			2 Memory Ports	
ALU Latency	1			1	
MUL Latency	2			2	
Load Latency	3			2	
Store Latency	1			1	
Memory/Cache	No cache stalls			Local BRAM memory	

Speedups are calculated considering the same operating frequency for all cases and for the baseline. The kernels are sorted according to the speedups for *a1*. The kernels can be classified in three groups: 1) for *i4*, *i1*, *i11*, *i2*, *i3*, and *i7*, our approach outperforms VLIW execution; 2) for *i10*, *i8*, and *i9*, the performance is roughly equivalent; and 3) for *i5* and *i6*, the VLIW processors are more efficient. The geometric mean speedups over a MicroBlaze processor for *r1*, *r2*, and *r3* are 2.22×, 2.58×, and 2.61×, respectively. Under the same conditions, ρ-VEX execution achieves a geometric mean speedup of 2.63×. Customized accelerators (*a1*) are on average 1.78× faster than a four-issue VLIW processor.

Due to different instruction sets, the IPC cannot be directly used as a comparative metric to justify performance differences between our accelerators and the VLIW models on a per-kernel basis. For instance, for *r2*, the loops for *i8* and *i9* contain 38 VEX instructions, and simulation reports an IPC value of 2.4. That is, a total of 97 VEX operations implement the computations of each loop. The respective MicroBlaze loop traces contain 148 instructions, whose execution on the accelerators has a similar performance to *r2*, but requires executing approximately 6.5 IPC. Instead, the following aspects are responsible for these performance differences.

First, the VLIW compiler does not perform loop pipelining, but may perform aggressive loop unrolling to better exploit ILP. We notice for group 1 that an increase in VLIW issue width does not lead to an increase in performance. This means that there is no more ILP to exploit, or the compiler is unable to do so. In contrast, we aggressively exploit loop pipelining. For the same kernels, our scheduler reports that the average number of intraiteration IPC is 1.42. This is similar to the average IPC of the VLIW processors for the same kernels, 1.50. Combining intraiteration ILP and interiteration ILP, our accelerators reach 5.96 IPC for this subgroup of kernels.

Second, our approach performs better when the number of iterations per accelerator call is high, as the communication overhead is better amortized. Section V-B1 shows that this is most critical to the *i5* and *i6* benchmarks (group 3). The potential value of IPC for these both kernels is 6.50, which is similar to the third best performing case, *i2*, whose IPC is 6.40. However, only 15 iterations are executed per call accelerator call for *i5* and *i6*, versus an average of 1364 for *i2*. Execution on a VLIW processor suffers no such effects and can exploit loop optimization techniques, resulting in better performance than the customized accelerators. In these two cases, the VEX compiler partially unrolls the inner loop. Furthermore, they may improve the running time of the sections of the kernel that are not mapped to the accelerators.

In terms of resources, we can compare our approach with the implementation of the ρ-VEX processor reported in [13]. On a Virtex-6 xc6vlx240t, the four-issue version of ρ-VEX requires 1046 FFs, 12 899 LUTs, and 16 BRAMs (36 Kb each). Our specialized accelerators require approximately half as many LUTs and 5× as many FFs. We do not use BRAMs for data storage, which justifies this higher register requirement.

VI. RELATED WORK

The generation of multiloop accelerators for application specified integrated circuits is considered in [24]. Each loop accelerator is produced from a source-code loop by targeting an abstract VLIW-like architecture and first allocating enough FUs to ensure the desired throughput (a predefined II is assumed for each accelerator). After moduloscheduling, the register files of each FU are sized and their connections to the FUs specified. The data width of each FU is adapted to the operation. Three approaches using ILP are used to obtain a single accelerator for multiple functions: 1) instantiation of several single-function accelerators without resource sharing; 2) successive pairwise merging of single-function accelerators via either a heuristic or an ILP formulation; and 3) joint scheduling of all loops via an integer linear programming formulation of moduloscheduling. Unlike our approach, two or more individual FUs are combined into a single multipurpose unit when joining single-function accelerators. The resulting merged unit drives a single-output FIFO, which likewise results from the combination or several FIFOs in terms of width and depth. On a Pentium 4 class processor, the running times took from 20 min to several hours (for benchmarks combining up to six accelerators). The third approach only provided speedup results for five of the ten benchmarks due to complexity issues. When available, the latter results

TABLE VI
COMPARISON OF THE PROPOSED APPROACH WITH RELATED WORK

Approach	CPU	Coupling	Target Trace	Methodology	Source/binary Modification	Accelerator Architecture	Supported Operations	Memory Access	Speedup
CCA [20]	ARM	Tight	Sequences forming graphs w/ 4 inputs and 2 outputs max	Compile time sub-graph detection	Compile time binary instrumentation	Alternating rows of two types of FUs; integrated into processor pipeline; crossbar interconnects	Addition; subtraction; logical operations	Not supported	2.21× (arithmetic)
Warp [5]	MicroBlaze	Loose	Most frequent innermost loops	Runtime trace profiling, binary disassembly and circuit synthesis	Runtime binary modification	Custom fine-grained reconfigurable fabric	Integer arithmetic and logic	Single port access for regular patterns	3.20× (geometric)
ADEXOR [21]	MIPS	Tight	Single-entry multiple-exit trace with multi-path	Offline binary profiling through simulation;	Compile time binary modification	Feed-forward rows of homogeneous FUs; integrated into processor; controllable limited interconnect	Integer and fixed-point arithmetic (except div. and mult.)	At most one store operation	1.87× (arithmetic)
DIM [22], [6]	MIPS	Tight	Sequences of basic blocks	Runtime binary profiling and generation of accelerator configurations	None	Feed-forward rows of heterogenous FUs; rich controllable interconnect	Integer arithmetic and logic	Concurrent accesses to random addresses	2.17× (arithmetic)
ASTRO [23]	MicroBlaze	Loose	Atomic sequences of basic blocks (w/ hazard analysis)	Offline trace detection via simulation; offline synthesis accelerators	None	One-to-one translation of loop CDFGs to pipelined accelerator circuit	Integer arithmetic and logic	Tailored cache for multiple parallel accesses	7.06× (geometric)
This work	MicroBlaze	Loose	Loop trace; single-entry point and multiple exit points	Offline trace discovery and accelerator generation	None	Customized, pipelined, multi-loop accelerator with coarse-grained FUs and dedicated interconnects	Floating-point and integer arithmetic; all bitwise logic; comparisons	Two concurrent accesses to arbitrary addresses	5.61× (geometric)

were just slightly better than the results of approach 2) (i.e., postschedule ILP-based union of single-function accelerators). For a 0.18-μm standard cell technology, we determine that multiple-function accelerators save on average 43% of the resources required by the sum of individual accelerators. We target FPGAs, which makes a direct comparison difficult, but we observed marginally smaller savings for our own multiloop accelerators.

While [24] focuses on the accelerator, other approaches consider the complete embedded system. These approaches differ mostly in the following aspects: whether they are based on high-level source code or binary code; if based on binary code, whether it is a static or dynamic analysis; whether this analysis is done offline or at runtime; the type of integration of the coprocessor modules to the host; the overall architecture of the coprocessor; and the type of operations supported by the coprocessor. Table VI shows the more relevant related approaches and compares them with this paper in terms of these aspects.

The acceleration framework of [20] employs a single-cycle, four-input configurable unit [called the configurable compute accelerator (CCA)] with feedforward data propagation embedded in the CPU's datapath. A compilation step identifies the sequences of instructions for acceleration. The first time the sequence is executed, a translation engine integrated into the processor creates a custom instruction for the CCA instance present in the system. For 22 benchmarks, an average speedup of 2.21× is reported for domain-specific CCA designs coupled to an ARM processor. While we accelerate cyclical graphs, this approach targets smaller acyclical graphs. Our loop acceleration usually attains better results for large graphs and directly supports more complex operations (including multicycle operators). The finer grained CCA approach can accelerate small code regions throughout the entire application, but requires compile-time modifications of the code to isolate the sequences.

The latest version of the Warp processor is presented in [5]. It is based on profiling runtime traces to detect frequent loops. The Warp system augments a MicroBlaze with a loosely coupled, fine-grained, custom reconfigurable fabric designed to be a viable target for on-chip synthesis tools. The on-chip tools run on a second MicroBlaze processor and synthesize a circuit per detected frequent basic block. The binary is modified at runtime to switch execution to the generated circuit. Up to one regular pointer-based memory access is supported. The proposed reconfigurable fabric was modeled behaviorally at RTL and evaluated by postlayout simulation. For six kernels, a geometric mean speedup of approximately

3.2× is achieved by prototyping on a Virtex-II Pro FPGA.

The ADEXOR approach is based on instruction set extension [21]. A reconfigurable FU is coupled to an MIPS processor pipeline. The accelerator contains 16 elementary FUs, and has eight inputs and six outputs. Data are propagated in a feed-forward fashion. The design of the accelerator was determined by a quantitative analysis of the instruction sequences which the approach aims to accelerate. They may contain up to four branch instructions (i.e., multiple exits) as long as these are not function returns, backward branches, or indirect branches. In contrast, we support the former two types of branch instruction, and may support the latter as long as the register used for the indirect branch remains constant throughout all the iterations of the loop. The application binaries are profiled with an instruction set simulator. Each selected sequence is converted into a custom instruction for the accelerator, which is inserted in the second compilation step. The architecture was synthesized for the 0.18-μm technology, and an average arithmetic speedup of 1.87× was reported for 16 applications of the MiBench suite over a single-issue MIPS-based processor.

The approach for inner loop acceleration presented in [6] uses runtime binary profiling by auxiliary hardware that monitors the execution stream for frequent backward branches. Detected loops are moduloscheduled by on-chip translation software onto the target accelerator, with support for hazard detection. The target accelerator is a fixed architecture with a programmable interconnect, containing 16 FUs (ALUs, multipliers, and up to two memory units), each with its own register file. The accelerator is tightly coupled to the main processor and fetches operands from its register file. The work builds on the VEX architecture and tools; the binary translation itself is performed over VEX binaries compiled for a four-issue version. The proposed runtime moduloscheduling-based binary translation is evaluated in a cycle-accurate simulator and compared with offline compilers, which targeted several VLIW models. A single-issue MIPS processor was used as a baseline. The scheduling approach outperforms the VLIWs in terms of exploited parallelism using an accelerator capable of one memory access only. With two memory accesses, the average achieved IPC for 11 loops is 7.99. It requires 13 000 LUTs and 23 BRAMs of 36 kb. As a comparison, our version with eight ALUs requires half as many LUTs and approximately 2800 FFs, achieving an average IPC of 5.2 for 24 benchmarks.

The ASTRO approach [23] is based on the detection of MicroBlaze instruction sequences by profiling with a simulator. ASTRO's accelerators are one-to-one translations of CDFGs to pipelined datapaths, where CDFG nodes are instantiated as hardware primitives. Unlike our approach, each accelerator supports only one loop. For a set of N target loops, N accelerators are instantiated, which suggests less scalability than the approach presented in this paper. The customization effort is focused on maximizing memory access parallelism, using dynamic memory access analysis to create a tailored, BRAM-based, multiported cache for use by the accelerators. In contrast, we provide only two parallel accesses in a straightforward, yet, in our experience, efficient manner. The several accelerators are coupled to a GPP and are invoked by observing the instruction address. The GPP has a data cache augmented with a selective invalidation mechanism used to maintain coherency after an accelerator writes to the shared external data memory. ASTRO detects memory access hazards, while our trace detection step currently does not. An average ASTRO accelerator with a multicache network requires 9786 LUTs, 12 603 FFs, and 26 BRAMs of 36 kb, on a Virtex-5 device. For ten benchmarks from MiBench and SPEC2006, the geometric mean speedup versus software-only execution was 7.06×. For our 24 benchmarks, the average fully customized accelerator requires 7012 LUTs and 5839 FFs, and the geometric mean speedup is 5.61×.

The accelerator architecture and scheduler described in Sections III and IV represent a considerable improvement over our own previous works [9], [25]. Support for floating-point operations increases applicability and performance, especially considering that floating-point operations are particularly penalizing for the MicroBlaze. Fully exploiting loop pipelining improves both the achieved speedups and the resource requirements. The implementation in [25] was capable of partially exploiting loop pipelining: for eight integer kernels, and the geometric mean speedup was 2.57×. The performance was hindered mostly by the unoptimized handling of memory accesses. In contrast, the scheduler and the architecture of the present implementation allow for a geometric mean speedup of 4.61× for 11 integer kernels. Considering FPGA resources, the previous implementations had a much higher cost relative to the present 1-D moduloscheduled accelerator. In [25], an average of 7760 LUTs and 8500 FFs were required for eight single-configuration accelerators. In addition, the cost increased considerable for multiconfiguration cases. The present implementation allows for more compact multiconfiguration accelerators by better utilizing FUs both for the execution of a single loop and across the several supported CDFGs.

VII. Conclusion

This paper has presented a transparent binary acceleration approach for applications in embedded systems. Execution time of unmodified binaries of regular kernels is significantly reduced by customized loop accelerators, which rely on loop pipelining, enhanced by the exploration of data access parallelism. Each accelerator instance is generated automatically from binary execution traces obtained by an offline simulation step. As a result, the accelerated portions of code are the most representative in terms of execution time.

The applicability of our approach has greatly increased due to the support for floating-point operations. The use of specialized and fully pipelined floating-point Functional Units versus fully fledged FPUs consumes fewer resources, and the operations are executed with lower latency compared with the host processor.

Our study on multiloop accelerators showed that supporting multiple loops (up to five) does not lead to a significant resource requirement increase for the proposed architecture. The specialized accelerators perform on par with generalized arrays of eight ALUs, and outperform the arrays of four ALUS, while requiring fewer resources in both cases.

The proposed architecture achieves a geometric mean speedup of 6.61× over a single MicroBlaze for a set of 24 kernels. For the integer kernels, a geometric mean speedup of 1.78× over a four-issue VLIW was measured, which we attribute to the data access parallelism we exploit and the maximization of loop pipelining.

The accelerator architecture implements the equivalent of the complete MicroBlaze instruction set. A customized instance of the accelerator is generated from frequently executing binary instruction traces by a complete supporting toolchain. CRs are also produced and seamlessly integrated in the application to provide fully transparent runtime migration of the execution flow to the accelerator.

ACKNOWLEDGMENT

The authors would like to thank S. Wong from the Delft University of Technology, The Netherlands, for providing the ρ-VEX processor release and tools.

REFERENCES

[1] P. Coussy and A. Morawiec, Eds., *High-Level Synthesis: From Algorithm to Digital Circuit*, 1st ed. The Netherlands: Springer, 2008.

[2] N. R. Miniskar, S. Kohli, H. Park, and D. Yoo, "Retargetable automatic generation of compound instructions for CGRA based reconfigurable processor applications," in *Proc. Int. Conf. Compil., Archit. Synth. Embedded Syst. (CASES)*, 2014, pp. 1–9.

[3] W. Kim, Y. Choi, and H. Park, "Fast modulo scheduler utilizing patternized routes for coarse-grained reconfigurable architectures," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, Dec. 2013, Art. no. 58.

[4] G. Ansaloni, K. Tanimura, L. Pozzi, and N. Dutt, "Integrated kernel partitioning and scheduling for coarse-grained reconfigurable arrays," *IEEE Trans. Comput.-Aided Design Integr. Circuits Syst.*, vol. 31, no. 12, pp. 1803–1816, Dec. 2012.

[5] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *ACM Trans. Embedded Comput. Syst.*, vol. 8, no. 3, Apr. 2009, Art. no. 22.

[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] J. Bispo, N. Paulino, J. M. P. Cardoso, and J. C. Ferreira, "Transparent trace-based binary acceleration for reconfigurable HW/SW systems," *IEEE Trans. Ind. Informat.*, vol. 9, no. 3, pp. 1625–1634, Aug. 2013.

[8] J. Bispo, N. Paulino, J. M. P. Cardoso, and J. C. Ferreira, "Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units," *Int. J. Reconfigurable Comput.*, vol. 2013, 2013, Art. no. 340316.

[9] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "A reconfigurable architecture for binary acceleration of loops with memory accesses," *ACM Trans. Reconfigurable Technol. Syst.*, vol. 7, no. 4, Jan. 2015, Art. no. 29.

[10] J. Bispo and J. M. P. Cardoso, "On identifying segments of traces for dynamic compilation," in *Proc. Int. Conf. Field-Program. Logic Appl. (FPL)*, 2010, pp. 263–266.

[11] B. R. Rau, "Iterative modulo scheduling: An algorithm for software pipelining loops," in *Proc. 27th Annu. Int. Symp. Microarchitecture*, 1994, pp. 63–74.

[12] N. Paulino, J. A. C. Ferreira, J. A. Bispo, and J. M. P. Cardoso, "Transparent acceleration of program execution using reconfigurable hardware," in *Proc. Design, Autom. Test Eur. Conf. Exhibit.*, San Jose, CA, USA, 2015, pp. 1066–1071.

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

[14] J. Bispo. (Feb. 2015). *Megablock Extractor for MicroBlaze v0.7.14*, accessed on Apr. 3, 2015. [Online]. Available: https://sites.google.com/site/specsfeup/

[15] T. Peters. (1992). *Livermore Loops Coded in C*, accessed on Apr. 3, 2015. [Online]. Available: http://www.netlib.org/benchmark/livermorec

[16] Texas Instruments. (2008). *TMS320C6000 Image Library (IMGLIB)—SPRC264*, accessed on Dec. 23, 2012. [Online]. Available: http://www.ti.com/tool/sprc264

[17] Xilinx, Inc. (May 2015). *7 Series FPGAs Overview - DS180 (v1.17)*, accessed on Nov. 28, 2014. [Online]. Available: http://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf

[18] Texas Instruments. (Feb. 2016). *Fusion Digital Power Designer*, accessed on Apr. 14, 2016. [Online]. Available: http://www.ti.com/tool/fusion_digital_power_designer

[19] J. A. Fisher, P. Faraboschi, and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*. San Mateo, CA, USA: Morgan Kaufmann, 2005.

[20] 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 Int. Symp. Comput. Archit.*, Jun. 2005, pp. 272–283.

[21] H. Noori, F. Mehdipour, K. Inoue, and K. Murakami, "Improving performance and energy efficiency of embedded processors via post-fabrication instruction set customization," *J. Supercomput.*, vol. 60, no. 2, pp. 196–222, May 2012.

[22] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proc. Design, Autom. Test Eur.*, Mar. 2008, pp. 1208–1213.

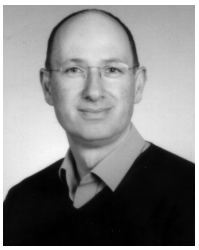
[23] M. Lin, S. Chen, R. F. DeMara, and J. Wawrzynek, "ASTRO: Synthesizing application-specific reconfigurable hardware traces to exploit memory-level parallelism," *Microprocess. Microsyst.*, vol. 39, no. 7, pp. 553–564, 2015.

[24] S. Mahlke, M. Kudlur, H. Park, and K. Fan, "Increasing hardware efficiency with multifunction loop accelerators," in *Proc. 4th Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, Oct. 2006, pp. 276–281.

[25] N. M. C. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Trace-based reconfigurable acceleration with data cache and external memory support," in *Proc. IEEE Int. Symp. Parallel Distrib. Process. Appl. (ISPA)*, Aug. 2014, pp. 158–165.

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

He is currently a Researcher with INESC Technology and Science, University of Porto. His current research interests include run-time reconfigurable systems, embedded systems in FPGAs, co-processor hardware acceleration, and tools for hardware/software co-design automation.



João Canas Ferreira (M'90) 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 with the Faculty of Engineering, University of Porto, and a Senior Researcher with INESC TEC, University of 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 Association for Computing Machinery and Euromicro.

João M. P. Cardoso (M'93) 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 Technical University in Lisbon (IST/UTL), Lisbon, Portugal, in 1997 and 2001, respectively.

He was with the IST/UTL from 2006 to 2008, a Senior Researcher with INESC-ID, Lisbon, from 2001 to 2009, and with the University of Algarve, Faro, Portugal, from 1993 to 2006. In 2001 and 2002, he was with PACT XPP Technologies, Inc., Munich, Germany. He is currently an Associate Professor with the Department of Informatics Engineering, Faculty of Engineering, University of Porto, Porto, Portugal. His current research interests include compilation techniques, domain-specific languages, reconfigurable computing, and application-specific architectures.

Dr. Cardoso is a member of the IEEE Computer Society and a Senior Member of ACM.