Transparent Acceleration of Program Execution Using Reconfigurable Hardware*

Nuno Paulino, João Canas Ferreira, João Bispo, and João M.P. Cardoso
INESC TEC and Faculty of Engineering
University of Porto
Porto, Portugal
{nuno.paulino, jcf, jbispo, jmpc}@fe.up.pt

***Abstract*—The acceleration of applications, running on a general purpose processor (GPP), by mapping parts of their execution to reconfigurable hardware is an approach which does not involve program's source code and still ensures program portability over different target reconfigurable fabrics. However, the problem is very challenging, as suitable sequences of GPP instructions need to be translated/mapped to hardware, possibly at runtime. Thus, all mapping steps, from compiler analysis and optimizations to hardware generation, need to be both efficient and fast. This paper introduces some of the most representative approaches for binary acceleration using reconfigurable hardware, and presents our binary acceleration approach and the latest results. Our approach extends a GPP with a Reconfigurable Processing Unit (RPU), both sharing the data memory. Repeating sequences of GPP instructions are migrated to an RPU composed of functional units and interconnect resources, and able to exploit instruction-level parallelism, e.g., via loop pipelining. Although we envision a fully dynamic system, currently the RPU resources are selected and organized offline using execution trace information. We present implementation prototypes of the system on a Spartan-6 FPGA with a MicroBlaze as GPP and the very encouraging results achieved with a number of benchmarks.**

I. Introduction

Data-intensive embedded applications typically have a number of computational kernels responsible for most of the application execution time. The execution of some of those kernels can be significantly accelerated by using custom hardware. This is the case in architectures coupling a General Purpose Processor (GPP) with reconfigurable hardware (see, e.g., [1]) where those kernels can be migrated to dedicated hardware units on the reconfigurable hardware. Retargeting applications to execute on such heterogeneous platforms requires significant effort in hardware/software partitioning and co-synthesis [2][3] and typically the following per-application basis steps: (i) identifying the kernels, (ii) designing (or mapping to) the hardware accelerator, (iii) modifying the application to use the new hardware, and (iv) integrating the hardware into the host system/processor. The inclusion of custom hardware requires significant design time and hardware expertize, as both software and hardware have to be co-designed. This mapping process harms both code and performance portability.

The common hardware/software partitioning and mapping approaches are specific to the target architecture and rely on the availability of the application source code. When considering the diversity of embedded system architectures, this approach might be too costly, as different implementations may have to be deployed/designed according to each specific product or product version architecture. Alternatively, binary acceleration approaches rely on information extracted from the compiled application [4]. Although some high-level code information is not present in binaries, binary acceleration decouples hardware/software development and solves the problem related to the non-availability of the source code of the software running in the system. Furthermore, by addressing dynamic hardware/software partitioning [5], approaches effectively contribute to code and performance portability as the system can decide about the sections to be migrated to reconfigurable hardware at runtime. By either statically analyzing the binaries or deriving runtime information by instruction stream monitoring, binary acceleration approaches are able to configure or generate dedicated hardware accelerators. Binary acceleration conveys additional acceleration opportunities as it can be applied to instruction traces from operating systems, middleware, libraries, JIT compilers and their generated code. It has the potential to deal with all the code running in the GPP no matter its provenience and the optimizations used to generate the binaries. This approach is also independent of the task allocation over a multicore/manycore architecture and does not limit the acceleration to instruction traces fixed and known *a priori*. Another potential advantage is its adaptability to technology improvements, e.g., due to process scaling.

Our approach consists of a transparent binary acceleration system based on a GPP complemented with automatically-generated, coarse-grained Reconfigurable Processing Units (RPUs) [6][7]. This paper presents our approach and an analysis of its efficiency for different RPUs. We describe recent improvements regarding RPU architecture and loop pipelining. These enhancements have extended the applicability and efficiency of our approach. We also present and discuss the most relevant challenges regarding the runtime translation of binaries to reconfigurable hardware.

This paper is organized as follows. In Section II, we present an overview of our approach. Section III briefly describes the target system architectures. Section IV presents and discusses the experimental results obtained with our prototypes. Section

* This work was partially funded by the European Regional Development Fund, through the COMPETE Programme (Operational Programme for Competitiveness) and by national funds from the FCT–*Fundação para a Ciência e a Tecnologia* within project FCOMP-01-0124-FEDER-022701, and through FEDER/ON2 and by FCT funds within project NORTE-07-124-FEDER-000062. N. Paulino acknowledges the support of FCT through PhD grant FRH/BD/80225/2011.

V summarizes related approaches. Section VI resumes challenges. Finally, Section VII concludes the paper.

II. General Approach Overview

Our approach is based on detecting a type of repeating pattern of executed instructions, named Megablock [8][7], and on transparently migrating the execution of Megablocks to customized hardware accelerators. Although we envision dynamic hardware/software partitioning and mapping, our implementations to date rely on a mixed offline/online approach, which consists of four steps: (1) Megablocks are identified from execution traces of the application, obtained from an instruction set simulator; (2) Selected Megablocks are translated to an RPU specification and corresponding configurations, producing a customized accelerator and all data necessary to configure the system, without modifying the software binaries; (3) At runtime, the GPP is monitored to detect the imminent execution of the regions of code translated to hardware; (4) Execution is then migrated transparently to the RPU. Once RPU execution completes, software execution resumes. Fig. 1 illustrates the main Megablock mapping and RPU generation stages.

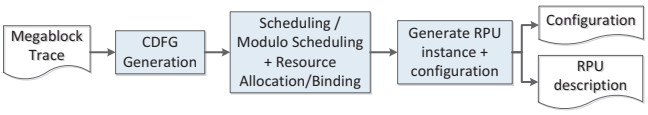


Fig. 1. Main mapping and generation stages.

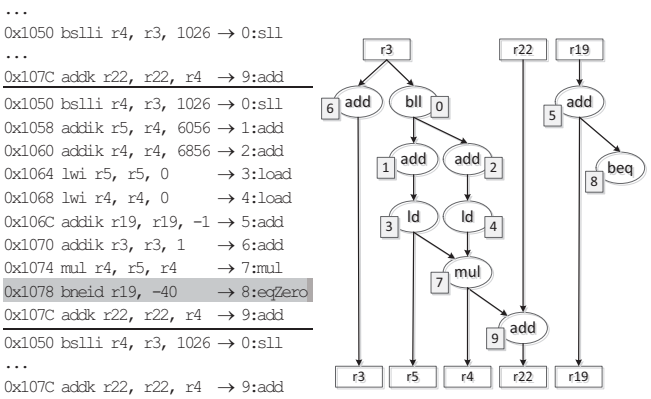
A. About Megablocks

A Megablock [8] represents a single-path control-flow execution forming a repeating instruction sequence with a single-entry and multiple-exit points. Thus, Megablocks are associated to loop behavior (either from loops in the source code and/or from recursive functions). The problem of detecting a Megablock from a trace of GPP instructions is similar to the problem of detecting *squares* (or *tandem repeats*) in strings. Although the detection is currently done offline, the development of the detection algorithm was driven by runtime and hardware implementation suitability. The algorithm finds *squares* in constant time by defining the maximum size of the pattern *a priori*. Thus, streams of GPP instructions can be processed in real-time. We currently use 50 as the minimum number of pattern repetitions, and 1,000 as the maximum pattern size. Instead of individual GPP instructions, the detector uses basic blocks as pattern element (reduces detection complexity). Fig. 2(a) shows a segment of an instruction trace when executing a *dot product* of two vectors and the Megablock detected.

The detection process computes the coverage per Megablock, i.e., percentage of total execution time spent executing Megablock code. Currently, we manually select the Megablocks to be executed by an RPU based on the following properties: the type of instructions in the Megablock (it should not include instructions not supported by the RPU); size of the pattern; coverage value; average number of iterations of the trace; and the number of exit points.

B. Mapping Stages

For each Megablock selected, our tool translates the GPP instructions to architecture independent operations (rightmost side of Fig. 2(a)), builds a CFDG (Fig. 2(b)), and performs a number of optimizations, such as constant propagation and folding, and operator strength reduction. The next mapping/translation steps vary according to the target RPU. For earlier row-oriented 2D RPUs, the translation was based on directly implementing the set of CDFGs as a configurable datapath. We started in [7] without support for pipelining, and added loop pipelining support in [9]. Our recent RPU is a 1D architecture and executes modulo scheduled loops. The translation tools compute the IIs of the CDFGs and generate an architecture with FUs and interconnections capable of executing all CDFGs at the minimum possible II. It is also possible to set an II to reduce resource usage at the cost of performance. The two types of RPUs are briefly described in Section III.



(a) (b)

Fig. 2. Example of a Megablock: (a) the repeating pattern in a trace of MicroBlaze instructions (left), and the corresponding architecture-independent operations (right). The exit condition is highlighted; (b) the CFDG built from the architecture-independent operations of the Megablock (right side of (a)).

C. Megablock Execution on the RPU

Prior to a Megablock execution, the Megablock input values stored in the GPP's register file are moved to the RPU. A Megablock is continuously executed until one of its exit conditions evaluates to true, and each iteration corresponds to an execution of an acyclic dataflow graph. The input registers written during execution (*r3*, *r19*, and *r22* in Fig. 2) are updated in each iteration. An exit condition evaluated to true signals the end of the Megablock execution, after which the system needs to perform the needed operations to continue execution on the GPP (e.g., update the values of the modified registers).

III. System Architecture

Much of the speedup potential in critical kernels comes from the parallelism not typically exploited by GPPs. As critical kernels typically operate on data arrays, often of size unknown at compile time, with irregular access patterns and data dependencies, it is important to add efficient memory accesses to RPUs. This also enables larger sections of code to be migrated, increasing the parallelism available for exploitation.

A. System Level Architecture

Our prototypes mainly consist of a Xilinx MicroBlaze processor (GPP), an RPU, and instruction/data memories. At runtime, a transparent migration mechanism shifts execution to

the RPU. We have considered different system organizations. The system in [9] (Fig. 3(a)) uses local memory for code, external memory for data, and a custom dual-port cache for the RPU, which can access the full range of the GPP's data. The system presented in Fig. 3(b) uses a memory access mechanism allowing the GPP and RPU to share a local data memory. At runtime, the *Injector* module detects the imminent execution of a translated Megablock by watching the address on the instruction bus. The *Injector* replaces the fetched instruction at that address with an absolute jump to a pre-determined memory location containing a tool-generated routine executed by the GPP to send the RPU inputs. The RPU executes the Megablock and afterwards the GPP reads results into its register file. Software execution resumes at the address where the *Injector* intervened.

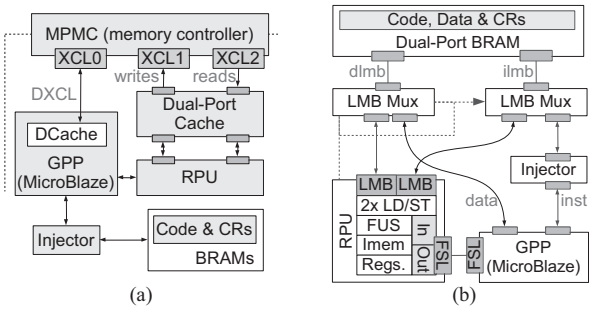


Fig. 3. System organization: (a) with external memory and pipelined RPU; (b) with local memory.

B. RPU Architecture and Generation

The systems shown, and other variants, have been the validation platforms for different RPUs. In general, RPUs consist of FUs, interconnect resources, memory ports and control logic. All our RPUs support MicroBlaze integer arithmetic and logic operations, multicycle operations, two simultaneous load/store operations, and access to a shared data memory.

We have been considering two types of RPUs: 2D RPUs organized in rows, and 1D RPUs. Fig. 4 shows two examples of RPUs implementing the *dot product* Megablock presented in Section II. The 2D RPUs [9] use a single-configuration per Megablock, as we do not yet consider temporal partitioning of Megablocks, and multiple configurations are needed to deal with more than one Megablock. This RPU is obtained by a direct translation of the Megablock CDFGs into (pipelined) datapaths. The result is a row-oriented arrangement of FUs, as exemplified in Fig. 4(a). Omitted for clarity are the specialized multiplexers for the inputs of the FUs, and the per-row control units, which perform inter-row handshaking. Each configuration sets all multiplexers according to the associated Megablock datapath. The latest implementation addresses 1D RPUs (see an example in Fig. 4(b)). It uses multiple single-cycle instructions for implementing a Megablock and each instruction configures all FUs and interconnect resources for a single execution step. There are two fixed *load/store* units per RPU instance and, like in previous approaches, during translation a tool selects the number and type of FUs and specialized multiplexers (omitted). It exploits loop pipelining via modulo scheduling [10]. The bottom half of Fig. 4(b) shows a partial modulo schedule for the *dot product* Megablock. Columns represent resources and rows represent time steps. Numbered circles identify the scheduled CDFG nodes (Fig. 2(b)), and edges represent the dataflow between the FUs through the time steps. For this Megablock, an II of 2 clock cycles is achieved. The light grey nodes indicate operations of the next iteration, which are executed before the current iteration is finished.

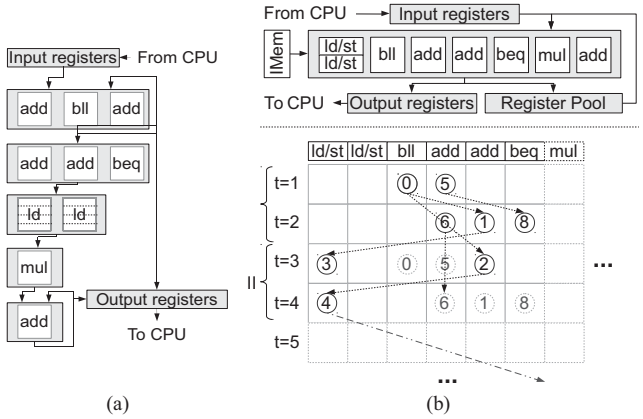


Fig. 4. RPU architecture examples implementing the *dotprod* Megablock: (a) 2D pipelined design; (b) 1D structure (top) and partial modulo schedule (bottom).

IV. Experimental Results

We have implemented and evaluated the three different RPU architectures supported by two mapping approaches. We evaluate the RPU implementations by integrating them into the presented system architectures (see Fig. 3) through FPGA prototyping. We target a Xilinx Spartan-6 FPGA and the RPU's RTL Verilog descriptions generated by our toolchain were synthesized, mapped, placed and routed with Xilinx *ISE 14.7*. For evaluation purposes, we consider 12 benchmarks from signal and image processing. The benchmarks were compiled by *mb-gcc 4.6.4* with the *O2* optimization flag.

While the first 2D RPUs [6][7] do not consider loop pipelining (RPU-C), the 2D RPU used in [8] (RPU-B) and the 1D RPU presented here (RPU-A) have both support for loop pipelining. Fig. 5 presents the speedups achieved for the three approaches. Each benchmark's execution trace resulted in one Megablock (with average/maximum/minimum of 48.7/148/10 GPP instructions, respectively), executed once per benchmark execution, except for *motionestimation* and *quantize* for which there were 4.096 and 16 Megablock calls, respectively. The binary acceleration with the 1D RPU (RPU-A) achieved a geometric mean speedup of 5.44× (from 2.33× for *lookup2* to 28.43× for *checkbits*). On average, 99.3% of the software execution migrated to the RPU. The GPP+RPU execution time consists of 89.60% spent in the RPU, 7.24% spent in GPP↔RPU data communications, and 3.16% in the GPP. Globally, the time spent in transferring data between the GPP and the RPU is very low. There are a few exceptions: e.g., in *motionestimation* the communication is responsible to 45.34% of the execution time. *Checkbits* has the next highest percentage of time spent in communications (8.69%). Four benchmarks have negligible communication overhead (< 1.3%). In a hypothetical scenario without communication overhead, the geometric mean speedup would increase by 8.25%.

The speedups achieved with RPU-A are significantly higher than the ones achieved with RPU-B and RPU-C. For the benchmarks previously used in the experiments with RPU-B (11 out of 12) and with RPU-C (6 out of 12), we achieve a geometric mean speedup of 5.44× (RPU-A) vs 2.25× (RPU-B), and 8.37× (RPU-A) vs. 2.42× (RPU-C), respectively. The main reason for these improvements is the more advanced loop pipelining, when comparing RPU-A to RPU-B. We note that the RPU-C does not use loop pipelining. Both RPU-B and RPU-C suffer from a 2D architecture organized by rows and from the many FUs used for bypassing data between rows. The RPU-A implementations include 14 FUs (from 6 to 33) and 31 registers (from 8 to 84) on average. On average, each RPU used requires about the same number of FPGA Slices as the MicroBlaze processor. This is a marginal improvement, over the resources needed for RPU-B (1.3× more slices than MicroBlaze), but is especially significant considering the higher speedups.

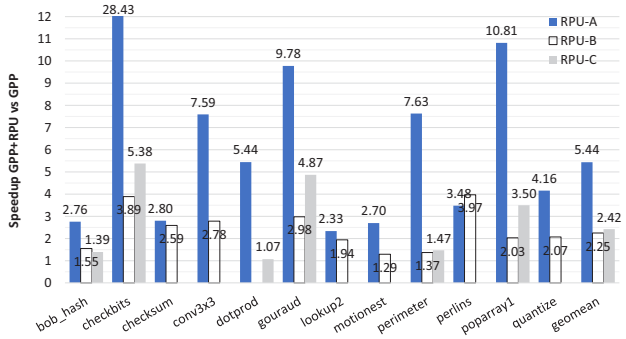


Fig. 5. Speedups achieved with different RPU architectures.

We compare herein the 1D RPUs generated (RPU-A approach) with the use of Xilinx *Vivado HLS* 14.3 when targeting a Xilinx Zynq™-7000 XC7Z020-CLG484-1 FPGA device. We manually translated the C code sections of the benchmarks equivalent to the Megablocks to functions input to *Vivado HLS*. When directing the tool to use the II values achieved for each benchmark implemented with RPU-A, and considering the same clock frequency, the HLS solutions improve the Megablocks' performance by 27.1% on average and requires 15.5% and 25.9% of the total FFs and LUTs used by the RPU implementations, respectively. Note, however, that the FPGA resources for RPU-A implementations consider all the hardware of the RPU (about 60% of which is for FUs, registers and interconnections). The performance improvements with the best II values achieved by HLS are higher as expected, 56.5% average improvement in performance, requiring 10.5% and 25.9% of the FFs and LUTs, respectively. These results are mainly due to higher II values used by our approach, possibly because of the less sophisticated optimizations of our binary translation. Thus, these results strongly make evident the room for further improvements regarding performance.

V. Related Work

The acceleration of critical program sections (hot spots) by using reconfigurable hardware (RPUs) coupled to a GPP has been addressed by many research efforts. There are approaches considering the application source code as input [11] and other approaches dealing with application binaries. We can distinguish two approaches dealing with application binaries: one leveraging the typical approach used with application source code (with the most significant difference being the decompilation step needed) [12][13], and another oriented to runtime (even if currently using a mixed approach) [14]. We can also distinguish approaches by the scheme used to couple the RPU to the GPP. In approaches where the RPU is tightly coupled to the GPP, it is common to support the execution of limited code sections (such as basic blocks) [15][16][17][18]. The work on instruction-set extensions is an example of partitioning usually limited to the migration to custom hardware of acyclic short sequences of instructions (see, e.g., [19]). In approaches where the RPU is loosely coupled to the GPP, as a co-processor, it is common to execute larger code sections (such as entire loops) [20][14][21][22]. We briefly describe next the approaches most relevant to our work and present in TABLE I a summary of the reported speedups.

The Dynamically Specialized Execution (DySE) approach [11] identifies path-trees of basic-blocks from source code, which are then implemented as specialized datapaths in 2D CGRAs, named DySE Resource (DySER) blocks. They use profiling to select the most frequently executed paths, and support pipelined execution. The binary is modified in order to add instructions for configuration and communication from/to DySER blocks. Another approach [22] maps loops in LLVM IR to a Vector Personality softcore for the Convey HC-1. At compile-time, a toolchain (including LLVM and Convey Compiler infrastructures) automatically identifies suitable loops (including outer-loops) for vectorization. With their fully automatic and unguided approach, they achieved higher performance than using a vectorizing compiler in a number of cases.

The WARP processor is a MicroBlaze-based system able to dynamically migrate innermost loops to FPGA-based reconfigurable logic [14]. WARP detects loops from frequently executed backward branches, and on-chip CAD tools decompile and map those loops to the reconfigurable fabric. The fabric is loosely coupled to the GPP, accesses one port of the data BRAMs, and does not support random memory access patterns. In [23] the approach is applied to thread acceleration with multiple processors and one reconfigurable fabric. Beck et al. [24] propose the Dynamic Instruction Merging (DIM) approach, which transparently maps, at runtime, sequences of basic blocks from a MIPS processor to a tightly coupled 2D CGRA. Inputs for the CGRA are fetched from the GPP's register file. The CGRA consists of rows with homogeneous ALUs, multipliers, and a load unit, and supports speculative execution. Ferreira et al [20] dynamically map binary code to a CGRA using modulo scheduling. They use backward jumps to identify innermost loops, and the CGRA supports logic/arithmetic and load/store operations. They evaluate their approach using an architecture with a CGRA tightly coupled to a VLIW softcore. Paek et al [21] map loops (with number of iterations statically known) from a binary file to an heterogeneous mesh-based CGRA, loosely coupled to a GPP. They allow CGRA memory accesses to continuous locations, and communicate data to/from the CGRA via a shared memory mechanism. An offline stage maps the loops and modifies the binaries to use the accelerator.

ADEXOR/AMBER [15] maps basic blocks to custom instructions executed in a triangle-shaped RFU tightly coupled to a MIPS-based GPP. Each RFU's FU supports all integer instructions of the GPP except multiply, divide and loads (supports one store). The RFU is in the same data path as the GPP FUs, and accesses the GPP's register file. An offline phase profiles the applications using an instruction set simulator. This phase detects hot basic blocks (i.e., the ones in loops) and converts each one to a custom instruction. Posterior work [25] added support for conditional execution with single-entry, multiple-exit custom instructions (related to multi-path traces from short branches). The CCA [16] moves a sequence of instructions to a triangular-shaped array of FUs coupled to an ARM processor's pipeline. An offline stage identifies the most representative sequences of instructions and generates the array. This stage does not consider sequences with branches, multiplications, shifts by statically unknown amounts, divisions, and memory operations. The approach delimits candidate regions by special instructions at the binary level, subsequently discovered and transformed into CCA configurations at runtime. A fully online method (i.e., with identification also at runtime) is presented in [26]. The JIT Customizable (JiTC) processor [17] includes a multistage Specialized Functional Unit (SFU), with logic and arithmetic operations, tightly coupled to the processor pipeline. An analysis of 21 applications drove the design of the SFU. A compilation step identifies the custom instructions, maps them to the SFU, and modifies the binary code. Another example is BERET [18] which maps recurring instruction sequences (named as *recurring traces* or *looping traces*) to subgraph execution blocks, which are tightly-coupled to a GPP. Looping traces are single-entry, single-exit program regions with high probability to loop back.

TABLE I. SPEEDUPS FOR MOST REPRESENTATIVE APPROACHES

Approach	Speedups (averages)
WARP [14]	3.3× over a MicroBlaze for 6 benchmarks (Powerstone and EEMBC suites)
DIM [24]	2.7× and 2.35× of energy savings over a MIPS for 18 benchmarks of the MiBench suite
ADEXOR [15]	1.16× vs a 4-issue RISC processor, for 19 benchmarks of the MiBench suite; 1.85× (for single entry multi-exit custom instructions) [25]
CCA [16] [26]	1.26× vs a 4-issue ARM for 29 benchmarks: CCA with depth 4 and 15 FUs for a 0.13 µm CMOS technology
Ferreira et al [20]	2.17× and 2.0× vs two versions, 4 and 8-issue, of a VLIW softcore, respectively
(DySE) approach [11]	2.1× (geometric mean) and energy reduction of 40% (and 60% w/ no speedup): 2 DySER blocks coupled to a dual-issue out-of-order processor for benchmarks of the PAR-SEC, SPEC and Parboil suites
JiTC [17]	1.18× and 1.23× for the profiled benchmarks (i.e., the ones used to derive the SFU design), considering in-order and out-of-order processors, respectively. The out-of-order pipeline uses 4 SFUs. For other benchmarks, achieved 1.1×
Paek et al [21]	9.4× (from 6.4× to 14.8×), 5 kernels from the DSPstone suite and using an ARM7TDMI-S with an 8×8 CGRA; 1.54× for a JPEG decoder
BERET [18]	energy consumption reductions of 3-4× for the migrated portions and 35% for the entire applications; 10% overall execution time reduction

Our work differs from the previous approaches by considering the Megablock as the mapping unit, by proposing an efficient RPU architecture, and by presenting a fully operational prototype implemented on an FPGA board. Furthermore, we support arbitrary accesses to memory, where most work either support limited access patterns or no accesses at all. Our system is based on a loosely coupled accelerator, therefore avoiding modifications to the GPP pipeline. As with other approaches, we use a shared-memory model. Although the WARP processor is similar in this respect, it is restricted to one access per clock cycle and to regular access patterns. We also note that each RPU in our approach is specifically tailored to a set of kernels, and therefore includes only the necessary FUs. In contrast, approaches with a limited fixed set of FUs may not be able to map some kernels or may impose a larger overhead in terms of hardware resources. The former is an issue for CCA, ADEXOR, and BERET, but is avoided by the finer-grained approach of WARP. Although identified at compile time, a similar concept to Megablocks [27][8] are Looping Traces [18]. However, that work considers a more constrained RPU architecture and a different execution model for the loops.

VI. CHALLENGES

The full exploitation of the approach followed in this work still needs to address a number of challenges.

A first challenge involves implementing lightweight cooperation between the GPP and the RPU with shared access to data memory and support for data caches. Furthermore, RPUs might support more complex operations (e.g., floating-point operations). Other challenges deal with extracting more information from the binary traces. The design of an autonomous adaptable system involves addressing the challenge of generation of the RPU on the target system, possibly by exploiting extendable, regular architectures and partial reconfiguration (just-in-time hardware compilation).

Other challenges deal with extracting more information from the binary traces. One way is to aid the runtime-only analysis with compiler-generated meta-information to assist identification of critical sections and data-dependence analysis. Further challenges include the identification of specific memory access patterns, in particular data streaming, to enable the deployment of very efficient, customized memory interfaces; the support for multi-path traces; removal of redundant memory accesses as the RPU can locally hold data. Previous work (see, e.g., [28]) give insights into advanced analysis and optimizations at the binary level. Knowing the possible nature of the binaries, simple data-dependence and alias analysis (e.g., for data partitioning) can be done as in [29]. The possibility to perform this kind of analysis at runtime is still an open issue.

Exploiting the possibilities opened up by having multiple configurations establishes another set of challenges. In particular, reducing the total number of resources and maximizing overall system performance by determining the best RPU to support a specific set of Megablocks is still an open question. Another challenge involves support for value-based specialization. For Megablocks many times invoked with the same value or small subset of values for one of its inputs, it may be worthwhile to consider specialized hardware implementations. Finally, implementing speculative execution may overcome the limited amount of ILP available in binary code. Using resources that might be unused in a specific configuration for speculative execution might improve performance at low cost.

VII. Conclusion

The trend to computing systems based on heterogeneous multicore architectures and hardware accelerators, the existence of legacy code, the variety of the architectures present in embedded systems, highlight the importance to devise efficient schemes to map computations at runtime. Transparent binary acceleration is a promising strategy for enhancing the performance of embedded systems without requiring changes to application development for each target platform. The advantages are manifold, e.g., large sequences of instructions are migrated to hardware configurations providing high-levels of parallelism via, e.g., efficient loop pipelining schemes.

We presented an approach consisting of a GPP aided by an RPU tailored during synthesis to accelerate repetitive instruction sequences named Megablocks. In the proposed system, Megablocks with memory accesses use the direct interface of the RPU to the GPP data memory and are mapped using loop pipelining techniques. The use of single row RPUs, specialized for each Megablock by a modulo-scheduling-based approach, shows a better performance/resources tradeoff than our previous RPUs (based on multiple rows). Our recent, fully operational, system prototype is able to achieve relevant accelerations. It achieved an overall geometric mean speedup of 5.44× for a set of 12 benchmarks. Ongoing work focuses on loop pipelining enhancements. Future work plans include extensions and support for multi-path Megablocks. This will allow our system to cover larger sections of the execution traces and thus to possibly increase the overall speedups.

References

[1] J.E. Carrillo and P. Chow, "The Effect of Reconfigurable Units in Superscalar Processors," in *Proc. ACM 9th Int'l Symposium on Field Programmable Gate Arrays*, New York, NY, USA, 2001, pp. 141–150.

[2] R.K. Gupta and G. De Micheli, "Hardware-software cosynthesis for digital systems," *IEEE Des. Test Comput.*, vol. 10, no. 3, pp. 29–41, Sep. 1993.

[3] R. Ernst, J. Henkel, and T. Benner, "Hardware-Software Cosynthesis for Microcontrollers," *IEEE Test*, vol. 10, no. 4, pp. 64–75, Oct. 1993.

[4] G. Stitt and F. Vahid, "Binary Synthesis," *ACM Trans Autom Electron Syst*, vol. 12, no. 3, pp. 34:1–34:30, May 2008.

[5] G. Stitt and F. Vahid, "Hardware/Software Partitioning of Software Binaries," in *Proc. IEEE/ACM Int'l Conference on Computer-aided Design*, New York, NY, USA, 2002, pp. 164–170.

[6] 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, Feb. 2013.

[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. Inform.*, vol. 9, no. 3, pp. 1625–1634, Aug. 2013.

[8] J. Bispo and J.M.P. Cardoso, "On identifying and optimizing instruction sequences for dynamic compilation," in *Int'l Conference on Field-Programmable Technology (FPT)*, 2010, pp. 437–440.

[9] N. Paulino, J.C. Ferreira, and J.M.P. Cardoso, "Trace-Based Reconfigurable Acceleration with Data Cache and External Memory Support," in *IEEE Int'l Symposium on Parallel and Distributed Processing with Applications (ISPA)*, 2014, pp. 158–165.

[10] B.R. Rau, "Iterative Modulo Scheduling: An Algorithm for Software Pipelining Loops," in *Proc. 27th Annual Int'l Symposium on Microarchitecture*, New York, NY, USA, 1994, pp. 63–74.

[11] V. Govindaraju, C.-H. Ho, and K. Sankaralingam, "Dynamically Specialized Datapaths for energy efficient computing," in *IEEE 17th Int'l Symposium on High Performance Computer Architecture (HPCA)*, 2011, pp. 503–514.

[12] G. Mittal, D. Zaretsky, X. Tang, and P. Banerjee, "An Overview of a Compiler for Mapping Software Binaries to Hardware," *IEEE Trans Very Large Scale Integr Syst*, vol. 15, no. 11, pp. 1177–1190, Nov. 2007.

[12] Y. Ben Asher and N. Rotem, "Binary Synthesis with multiple memory banks targeting array references," in *Int'l Conference on Field Programmable Logic and Applications. FPL 2009*, pp. 600–603.

[14] R. Lysecky and F. Vahid, "Design and Implementation of a MicroBlaze-based Warp Processor," *ACM Trans Embed Comput Syst*, vol. 8, no. 3, pp. 22:1–22:22, Apr. 2009.

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

[16] 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 Annual Int'l Symposium on Computer Architecture*, Washington, DC, USA, 2005, pp. 272–283.

[17] L. Chen, J. Tarango, T. Mitra, and P. Brisk, "A Just-in-Time Customizable processor," in *IEEE/ACM Int'l Conference on Computer-Aided Design (ICCAD)*, 2013, pp. 524–531.

[18] S. Gupta, S. Feng, A. Ansari, S. Mahlke, and D. August, "Bundled Execution of Recurring Traces for Energy-efficient General Purpose Processing," in *Proc. 44th Annual IEEE/ACM Int'l Symposium on Microarchitecture*, New York, NY, USA, 2011, pp. 12–23.

[19] N.T. Clark, H. Zhong, and S.A. Mahlke, "Automated custom instruction generation for domain-specific processor acceleration," *IEEE Trans. Comput.*, vol. 54, no. 10, pp. 1258–1270, Oct. 2005.

[20] 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 *Int'l Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation (SAMOS XIV)*, 2014, pp. 75–82.

[21] J.K. Paek, K. Choi, and J. Lee, "Binary Acceleration Using Coarse-grained Reconfigurable Architecture," *SIGARCH Comput Arch. News*, vol. 38, no. 4, pp. 33–39, Jan. 2011.

[22] T. Kenter, G. Vaz, and C. Plessl, "Partitioning and Vectorizing Binary Applications for a Reconfigurable Vector Computer," in *Reconfigurable Computing: Architectures, Tools, and Applications*, D. Goehringer, M. D. Santambrogio, J. M. P. Cardoso, and K. Bertels, Eds. Springer International Publishing, 2014, pp. 144–155.

[23] G. Stitt and F. Vahid, "Thread Warping: Dynamic and Transparent Synthesis of Thread Accelerators," *ACM Trans Autom Electron Syst*, vol. 16, no. 3, pp. 32:1–32:21, Jun. 2011.

[24] A.C.S. Beck, M.B. Rutzig, and L. Carro, "A transparent and adaptive reconfigurable system," *Microprocess. Microsyst.*, vol. 38, no. 5, pp. 509–524, Jul. 2014.

[25] H. Noori, F. Mehdipour, K. Inoue, and K. Murakami, "A Reconfigurable Functional Unit with Conditional Execution for Multi-Exit Custom Instructions," *IEICE Trans. Electron.*, vol. E91–C, no. 4, pp. 497–508, Apr. 2008.

[26] 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 *37th Int'l Symposium on Microarchitecture, 2004. MICRO-37 2004*, 2004, pp. 30–40.

[27] J. Bispo and J. M. P. Cardoso, "On Identifying Segments of Traces for Dynamic Compilation," in *Int'l Conference on Field Programmable Logic and Applications (FPL)*, 2010, pp. 263–266.

[28] K. Anand, M. Smithson, K. Elwazeer, A. Kotha, J. Gruen, N. Giles, and R. Barua, "A Compiler-level Intermediate Representation Based Binary Analysis and Rewriting System," in *Proc. 8th ACM European Conference on Computer Systems*, New York, NY, USA, 2013, pp. 295–308.

[29] G. Mittal, D.C. Zaretsky, X. Tang, and P. Banerjee, "Automatic Translation of Software Binaries Onto FPGAs," in *Proc. 41st Annual Design Automation Conference*, New York, NY, USA, 2004, pp. 389–394.