A Reconfigurable Architecture for Binary Acceleration of Loops with Memory Accesses

NUNO PAULINO, JOÃO CANAS FERREIRA, and JOÃO M. P. CARDOSO, INESC TEC and Faculty of Engineering, University of Porto, Portugal

This article presents a reconfigurable hardware/software architecture for binary acceleration of embedded applications. A Reconfigurable Processing Unit (RPU) is used as a coprocessor of the General Purpose Processor (GPP) to accelerate the execution of repetitive instruction sequences called *Megablocks*. A toolchain detects Megablocks from instruction traces and generates customized RPU implementations. The implementation of Megablocks with memory accesses uses a memory-sharing mechanism to support concurrent accesses to the entire address space of the GPP's data memory. The scheduling of load/store operations and memory access handling have been optimized to minimize the latency introduced by memory accesses. The system is able to dynamically switch the execution between the GPP and the RPU when executing the original binaries of the input application. Our proof-of-concept prototype achieved geometric mean speedups of 1.60× and 1.18× for, respectively, a set of 37 benchmarks and a subset considering the 9 most complex benchmarks. With respect to a previous version of our approach, we achieved geometric mean speedup improvements from 1.22 to 1.53 for the 10 benchmarks previously used.

Categories and Subject Descriptors: C.1.3 [**Processor Architectures**]: Other Architecture Styles—*Adaptable architectures; Heterogeneous (hybrid) systems*; C.3 [**Special-Purpose and Application-Based Systems**]: Real-Time and Embedded Systems

General Terms: Design, Performance, Validation, Experimentation

Additional Key Words and Phrases: Reconfigurable processor, memory access, Megablock, instruction trace, MicroBlaze, hardware acceleration, FPGA, hardware/software architectures

ACM Reference Format:
Nuno Paulino, João Canas Ferreira, and João M. P. Cardoso. 2014. A reconfigurable architecture for binary acceleration of loops with memory accesses. ACM Trans. Reconfig. Technol. Syst. 7, 4, Article 29 (December 2014), 20 pages.
DOI: http://dx.doi.org/10.1145/2629468

1. INTRODUCTION

Data-intensive embedded applications typically have a number of computational kernels that account for most of the execution time. For such cases, a custom architecture in which the execution of critical portions of the application can be migrated to dedicated

This work was 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 (Portuguese Foundation for Science and Technology) within project FCOMP-01-0124-FEDER-022701. N. Paulino acknowledges the support of FCT through grant SFRH/BD/80225/2011.

Authors' address: N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, INESC TEC, Campus da FEUP, Rua Dr. Roberto Frias, 4200-465 Porto, Portugal; emails: {dee11006, jcf}@fe.up.pt, jmpc@acm.org.

DOI: http://dx.doi.org/10.1145/2629468

hardware units acting as coprocessors might be a more efficient solution than a General Purpose Processor (GPP). Retargeting applications to execute on such heterogeneous platforms requires a significant effort in Hardware/Software (HW/SW) partitioning. Required steps include (1) identifying the portions of the application that are critical and could benefit from a dedicated hardware implementation, (2) designing the hardware itself, (3) modifying the application to use the new hardware, and (4) integrating the hardware into the host system/processor [Wolf 2003]. These are complex design steps, which need to be performed on a per-application basis. Significant development time and hardware expertise are required, as both software and hardware have to be codesigned.

Existing approaches to the problem differ in terms of where the partitioning effort is applied. Typically, HW/SW partitioning is applied to the application's source code (or intermediate representations) in the context of high-level design space exploration methodologies. The code sections selected for hardware implementation are then passed to high-level synthesis tools or are manually converted to RTL (Register-Transfer Level) descriptions for further processing. Changes to source code and modifications to the host system are often necessary. This process also requires tools that are usually unfamiliar to software developers.

Alternatively, binary acceleration relies only on information extracted from the compiled application [Clark et al. 2005; Paek et al. 2011; Lysecky and Vahid 2009; Noori et al. 2008]. By either statically analyzing the application binary or deriving runtime information by instruction stream monitoring (e.g., detection of critical basic blocks), binary acceleration approaches configure or generate dedicated coprocessors. Although binary traces provide less information than high-level code (which may also allow for more powerful optimizations), binary acceleration provides greater portability and decouples hardware and software development, because the application does not need to be retargeted to a specific architecture.

Much of the speedup potential found in intensive computational kernels comes from the parallelism not typically exploited by the GPPs. As critical application kernels typically operate on one or more input/output data arrays, often of size unknown at compile time, with irregular access patterns and data dependencies, it is important to focus on dedicated hardware units able to efficiently perform memory accesses. Specifically, it is important to support concurrent memory accesses to exploit the parallelism on those kernels.

Our previous work Bispo et al. [2013a, 2013b] presented a transparent binary acceleration system based on automatically generated coarse-grained RPUs (Reconfigurable Processing Units). In our system, a GPP is aided by a tailored coprocessor able to transparently accelerate the execution of repetitive sequences of GPP instructions. Those sequences of instructions are migrated at runtime from the GPP to the RPU coprocessor in a manner transparent to the application. Our approach targets a type of repeating pattern of instructions called Megablock [Bispo and Cardoso 2010b]. A Megablock represents a single-path execution through the control flow forming a repeating sequence of instructions with a single entry point and several exit points.

The dynamic partitioning approach of our system is detailed in Bispo et al. [2013b]. The mixed offline/online toolchain has four steps: (1) Megablocks [Bispo and Cardoso 2010a] are identified from execution traces of the application to accelerate; (2) selected Megablocks are translated to an RPU specification and corresponding configurations, producing a tailored reconfigurable accelerator and all information necessary to configure the system, without modifying the application binaries; (3) at runtime, the GPP is monitored to detect the imminent execution of the regions of code translated to hardware; and (4) execution is then migrated transparently to the RPU. Once RPU execution completes, software execution resumes.

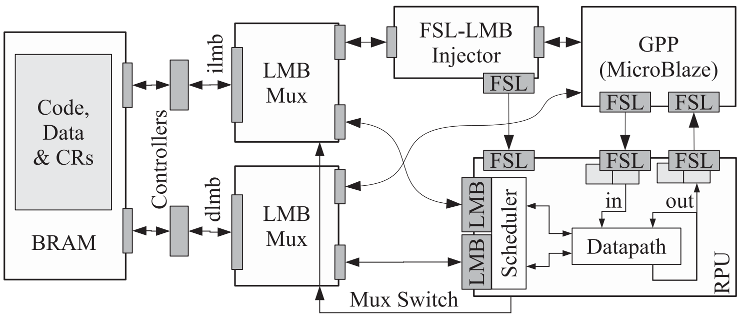


Fig. 1. Architecture overview. The RPU shares BRAM access with the GPP through the LMB Multiplexers.

Although most critical kernels that justify their mapping to hardware involve memory accesses, our original prototype system did not include RPU support to memory accesses. In addition, supporting memory accesses enables larger Megablocks to be considered for mapping, which tends to increase the amount of parallelism available for exploitation. In order to support acceleration of kernels with regular/irregular memory access patterns, this article describes enhancements to our binary acceleration approach. The enhancements described here significantly extend the applicability and the efficiency of the proposed approach. We also include improvements to the RPU generation toolchain for ensuring a better exploitation of memory accesses. A preliminary version of this work is presented in Paulino et al. [2013]. Relative to our previous work [Bispo et al. 2013a, 2013b], this article makes the following contributions:

(1) Design and implementation of a method for two masters to transparently share access to a single-master bus
(2) RPU support for an arbitrary number of load/store Functional Units (FUs)
(3) Extension of the RPU with two memory access ports, allowing for parallel memory accesses through the bus sharing mechanism
(4) Improvement of the hardware generation to save resources and exploit the RPU's memory ports
(5) A static memory access ordering algorithm for load/store operations
(6) Prototype validation with a large set of integer benchmarks

This article is organized as follows. An overview of the proposed system architecture is presented in Section 2. A general overview of the RPU architecture and supporting tool flow is given in Section 3. The handling and scheduling of memory accesses, as well as the module for transparent access to the shared data memory, are explained in Section 4. Section 5 presents and discusses the experimental results obtained with a proof-of-concept implementation. Section 6 summarizes related approaches and compares them with our approach. Finally, Section 7 concludes the article.

2. GENERAL ARCHITECTURE OVERVIEW

The architecture and tools described in this article extend [Bispo et al. 2013a, 2013b] with support for memory accesses and more flexible internal RPU control. The same four-step dynamic partitioning approach, consisting of identification, translation, detection, and migration, is employed. Figure 1 shows the enhanced system architecture, which consists of (1) a MicroBlaze GPP , which executes unmodified input binary code from local Block RAMs (BRAMs); (2) the RPU, which executes selected Megablocks and exchanges operands and results with the GPP via Fast Simplex Links (FSLs), using single-cycle *get*/*put* instructions; (3) the FSL Local Memory Bus (LMB) injector, which interfaces with the instruction bus of the GPP to monitor the instruction stream and controls GPP/RPU execution; and (4) two LMB Multiplexers, which enable shared access to the BRAM ports. The GPP is also connected through the Processor Local Bus to a serial UART and to a peripheral that measures execution clock cycles (not shown in Figure 1).

The injector and the RPU are generated offline from parameterized Hardware Description Language (HDL) descriptions. The parameter values are automatically determined from the Megablocks selected for hardware implementation. Parameters for the RPU include the resources it uses and the corresponding configurations. The injector's parameters include a table of addresses to which it reacts (the start addresses of mapped Megablocks).

At runtime, the injector is responsible for the migration step. This is accomplished by monitoring and modifying the contents of the instruction bus. If the start address of a region of code mapped to the RPU is detected, the injector inserts a branch instruction to a memory location containing an automatically generated Communication Routine (CR), which implements the communication between the GPP and the RPU. By executing the CR, the GPP sends operands from its register file to the RPU via the FSL. The injector simultaneously sends a configuration word to the RPU through its FSL. After the GPP has sent all operands, the injector issues the start signal to the RPU. The GPP then waits for the RPU to complete execution.

The RPU gains control of the LMBs and accesses the BRAM by asserting the switch signals of the LMB multiplexers. Each multiplexer allows for two master devices to access the single-master LMB, sharing the entire address space of a BRAM without any overhead. Access to both memory ports by the RPU allows for the exploitation of data access parallelism, which occurs in a large number of Megablocks, leading to higher speedups, as shown in Section 5.

Once RPU execution ends, control of the LMBs is handed back to the GPP, which executes the remainder of the CR, moving results to the register file and resuming execution of the application. The final state of the GPP register file is the same as it would be if the Megablock had been executed by the GPP. This allows the control flow of the program to resume normally. Thus, the execution of a critical kernel is accelerated transparently, and the RPU can process data residing in memory without incurring costly data transfers between the GPP and RPU.

3. RPU ARCHITECTURE AND TOOL FLOW

Figure 2 shows the RPU architecture, with details omitted for clarity. It consists of a tailored hardware circuit based on FUs (an illustrative example is depicted in Figure 2); two LMB ports that allow for concurrent memory accesses; the Memory Access Manager (MAM), which controls the access of the load/store FUs to the memory ports; a set of selector modules, which establish connections between the FUs and drive their respective *enable* signals according to the active configuration; and an *Iteration Control* module, which controls datapath execution. Interfaces to the GPP and injector are omitted. In general, FUs implement a single 32-bit operation.

Runtime reconfiguration of the RPU is done by selecting one out of a possible set of offline generated configurations prior to execution. A configuration defines the connections between FUs, enables or disables FUs, defines termination conditions, and configures the MAM. Section 3.1 details the structure and execution of the RPU datapath, and Section 3.2 shows how Megablock operations are scheduled.

The MAM, shown on the left side of Figure 2, multiplexes the access of all load/store FUs to the memory ports, which interface with the LMBs through the LMB multiplexers. As we use the dual-port BRAMs present in the FPGA architecture, the number of ports in this implementation is limited to two. For determining which load/store FUs

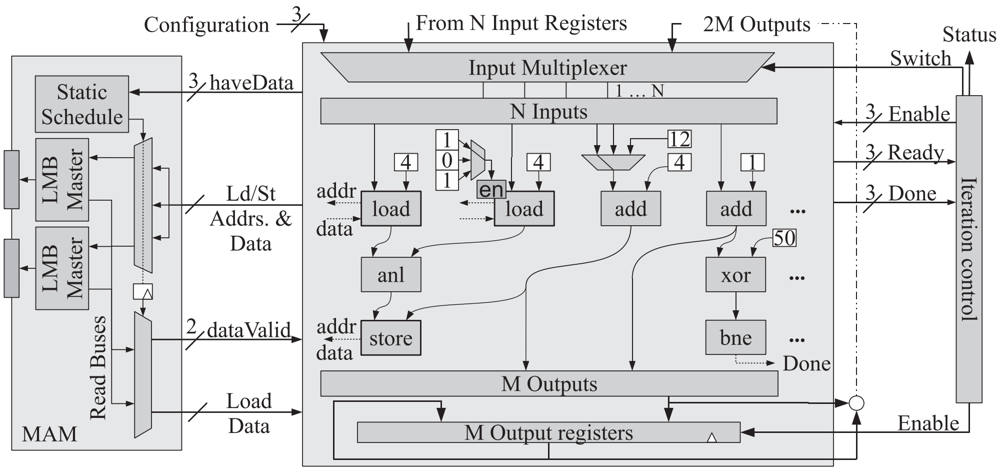


Fig. 2. Simplified conceptual diagram of the RPU's internal architecture, including memory access handling, for a simple datapath.

are connected to the memory ports in a given clock cycle, the MAM contains a set of memory access schedules, one per configuration. Section 4 details the behavior of the load/store FUs, gives an example of a memory access schedule, and explains the LMB multiplexers.

3.1. RPU Datapath

Structure. The RPU's datapath is composed of FUs connected by configurable connections. FUs are associated with computation stages that execute one at a time, as in Bispo et al. [2013a]. This is represented by the conceptual row-based layout in Figure 2. The datapath consists of a number of stages, each one with an associated number of FUs and without restrictions on the number of FUs of a given type.

Selector modules drive FU inputs according to the active configuration. An FU input can be an output of an upstream FU, a constant value, or a value generated in the previous iteration by downstream FUs. Figure 2 omits all but one selector module, which is driving the first input of the *add* FU with one out of three possible inputs. Selectors establish only the necessary connections required by at least one configuration, thus minimizing the required resources when compared to the use of generic interstage crossbars. If only one configuration is specified, all selectors are substituted by wires.

The outputs of the FUs are registered, and direct connections between FUs in different stages can exist. The interstage selectors increase in complexity with the number of configurations, so traversing several selectors may introduce critical delays in some cases. A parameter, which is currently adjusted manually, specifies whether to register these connections at intermediate stages.

Supported operations include all bitwise logical operations, loads and stores, and integer arithmetic operations (including integer division by a constant). The integer division by a constant FU uses a *multiply high* method [Warren 2002]. All operations are single cycle, except for integer division and load operations. The latency of load operations can vary according to the availability of memory ports as set by the MAM schedule; the minimum latency is two clock cycles.

Several load/store FUs can be activated simultaneously. Actual memory accesses are managed by the MAM. Load/store FUs receive three operands: data, address, and byte enable (to support memory accesses involving one, two, or four bytes). Since all operands can be computed by other upstream FUs, general address patterns are supported.

Execution. All the FUs in the same stage are activated simultaneously. Once the execution of all FU operations completes, the next stage of FUs is activated. In contrast to Bispo et al. [2013a], stages can have multicycle latency. The *Iteration Control* module and the RPU‘s datapath communicate using per-stage *ready* and *enable* signals. For each stage, an *enable* is issued by the *Iteration Control* module. The stage asserts its *ready* signal when all activated FUs trigger their individual *ready* signals. The following stage is then activated. This handshaking between the datapath and the control module makes execution control independent of the type, activation stage, and latency of the FUs. Execution on the RPU has a single entry point and multiple exit points. Stages may issue a *done* signal, which indicates that an exit FU (if any), such as the *bne* example in Figure 2, has detected a termination condition.

These exit FUs correspond to conditional branch instructions in the Megablock (unconditional branches are removed when building the Megablocks). By implementing the control flow within the RPU, the number of iterations to be performed does not need to be known at compile or synthesis time. Execution of a Megablock ends when one of the exit FUs evaluates to true. When one exit FU triggers, the current Megablock iteration is discarded and the values of the previous iteration are sent to the GPP. The interrupted final iteration is re-executed from the beginning by the GPP to allow the program to follow the expected control flow.

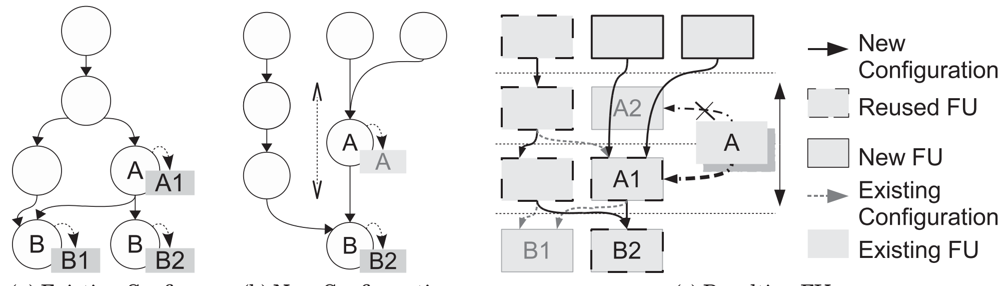
For a given configuration, the number of stages required to complete an iteration of the corresponding Control and Dataflow Graph (CDFG) equals the Critical Path Length (CPL), that is, the number of nodes in the longest path of that graph. The *Iteration Control* module generates the sequence of activation signals that enable successive stages required to compute one Megablock iteration. Once all required stages execute, data is fed back to previous stages for the next Megablock iteration. On the first iteration, the RPU fetches operands from the input registers, whose contents have previously been initialized by the CRs.

3.2. FU Scheduling

To generate the datapath of FUs, each Megablock is first processed to determine the corresponding CDFG representation. In our previous work, the FU scheduling mechanism was straightforward, activating operations according to an as-soon-as-possible (ASAP) policy. In order to improve resource efficiency, our current RPU generator tries to share as many FUs as possible among configurations. The current implementation uses a scheme based on list scheduling to assign CDFG operations to execution steps. This approach exploits the available slack to assign each FU to an activation stage where it can be reused by several RPU datapath configurations.

First, the ALAP (as-late-as-possible) and ASAP values of each FU are computed by analyzing dependencies. A virtual dependency between all store and exit operations is created, so that no store is scheduled before an active exit. That is, evaluation of termination conditions must be done prior to the first store, so that stores are not performed during the final Megablock iteration, which will be aborted and executed in software.

During RPU generation, operations are mapped to FUs one at a time. For each operation, the stages within its slack range are checked for existing FUs not already in use by the configuration being generated. Consider the example given in Figure 3. Assume that the CDFG shown in Figure 3(a) is already mapped to the datapath. A new CDFG, shown in Figure 3(b), is being translated. Gray boxes in Figures 3(a) and



(a) Existing Configuration (b) New Configuration (c) Resulting FUs

Fig. 3. Scheduling example for an FU with available slack.

3(b) indicate the FU to which the operation was mapped. Some of the FUs instantiated due to the translation of the first CDFG can be reused by the second CDFG (e.g., the leftmost three nodes of each CDFG). The operation of type *A* on the new CDFG can be mapped to the existing FU *A1* by exploiting the available slack of the operation, avoiding the instantiation of *A2*. If no reusable FU is found, a new one is mapped according to the ASAP schedule.

Scheduling of load operations is done differently, since only two memory ports are available. Loads require one cycle to strobe the address and a second to read data. Consider two consecutive stages, each with a single load operation. The first stage completes in two cycles, and the second stage also needs two cycles for execution, totaling four cycles. If slack is available, the load operations can be scheduled to the same stage, reducing the total number of cycles to two. Pairing load/store operations in this fashion reduces the number of cycles introduced by memory accesses. Additionally, the LMB accepts a new address while reading data from a previous strobe; that is, strobe/read cycles of several pairs of load/store operations in a stage can overlap. For instance, only three clock cycles are required to execute three or four independent loads.

Our scheduling scheme considers this type of overlapping and tries to schedule a new memory access in a stage that already contains an odd number of loads and stores. This takes advantage of the remaining free memory port without increasing the latency of the stage.

3.3. Tool Flow Summary

Figure 4 summarizes the tool flow, responsible for identifying Megablocks and translating them into RPU specifications and configurations. A previous version of the flow is described in greater detail in Bispo et al. [2013a].

Megablocks are first identified from application execution instruction traces (at the moment, a simulator is used to create the execution traces) using the Megablock Extractor [Bispo and Cardoso 2010b]. Each Megablock is then converted to a CDFG. The RPU generation tool performs the following tasks: (1) assigning the instructions of each CDFGs to FUs, (2) scheduling their activation as explained in Section 3.2, and (3) defining selector modules to drive FU inputs and generate the MAM schedules. Each run of the tool translates one Megablock and saves the generated RPU. The final run outputs the combined information to the files shown in Figure 4.

The parameter-based Verilog headers (*.vh* files) and the base RPU description are inputs to the vendor back-end tools. The parameters specify the number, type, and stage assignments of the instantiated FUs. Some FU types receive individual parameters per instance (e.g., the divider-by-constant has a parameter specifying the divisor). Other

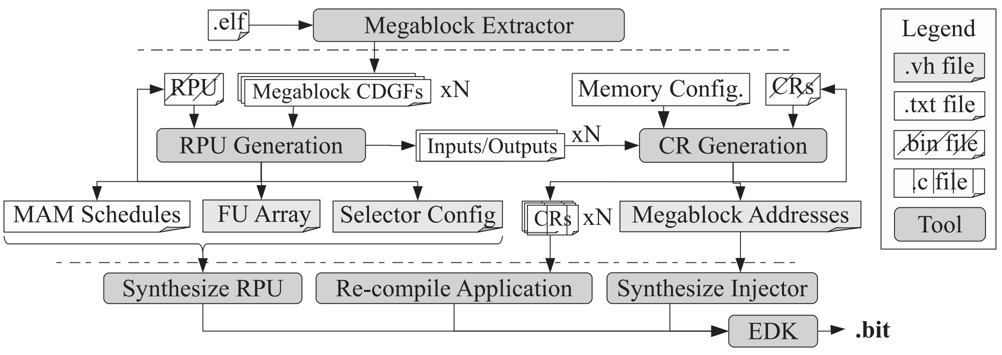


Fig. 4. Tool flow overview. Rectangular boxes indicate file types and rounded boxes indicate tools; *xN* indicates multiple files when considering *N* Megablocks.

parameters are used to define the interconnect capabilities of each selector, to generate the control signals with the correct bit width, and to aggregate the signals of the load/store FUs that connect to the MAM.

Files listing the sequence of GPP register values to transfer to/from the RPU (as operands and results) are input to the CR generation tool, which generates MicroBlaze instructions for the CRs. A table of Megablock start addresses is also generated for configuring the injector. The CRs are packed into C-language containers, which are compiled and stored in known addresses, so that the injector can branch the GPP to their positions. A custom linker script places the CRs in a code section after all contents of the original *ELF*, to avoid changing absolute memory positions of code or data.

4. RPU MEMORY ACCESS

The RPU memory access infrastructure is composed of three elements: the load/store FUs, the control logic of the MAM, and the LMB multiplexers.

4.1. Memory Access Units

Execution of a memory operation on the datapath is decoupled from the memory access itself. The load/store FUs neither have knowledge of the memory interface details nor contain any kind of access control. If enabled, each load/store FU asserts an access request signal to the MAM and waits for a reply.

Load FUs finish executing once data arrives from memory. The BRAMs respond in a single cycle, so in this implementation a load operation can be completed in a minimum of two cycles. As stores produce no data to be used directly in the datapath, the execution may continue immediately after they are issued. If no memory ports are free when a store FU is enabled, it buffers the data and waits for access to be granted while RPU computations proceed. A store FU only delays the processing if its buffered datum has not been handled prior to its next activation.

4.2. Memory Access Management

In our current proof-of-concept prototype, the RPU can perform up to two simultaneous write/read memory accesses by using both ports of the BRAM. There can be, however, more than two load/store FUs per stage; that is, more than two memory accesses can be issued in the same clock cycle. Simultaneous requests might not necessarily originate from FUs in the same stage, due to pending stores. When simultaneous requests occur, the delay introduced in the execution depends on the type of memory operation and

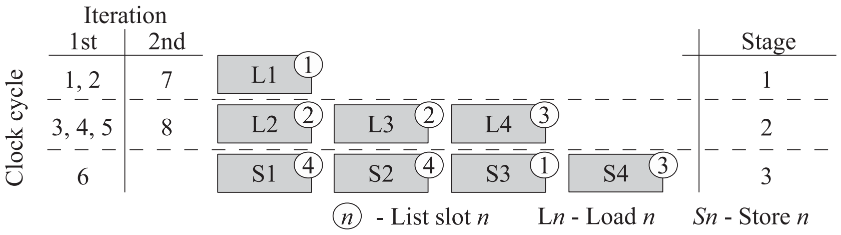


Fig. 5. Load-priority memory access ordering example.

the order in which the multiple concurrent accesses are handled by the selection logic in the MAM.

Distributed memory in the MAM holds one static memory access order list per configuration. Each list slot determines which two (at most) load/store FUs are connected to the memory ports. The MAM advances to the next list slot when the load/store operations for the current slot are handled. With this interface, no restriction on the number or assigned activation stage of memory operations is imposed. The MAM does not need to monitor the status of the datapath or to know stage assignments of load/store FUs. This MAM architecture is also extensible to a larger number of ports.

Memory access order is determined by an iterative algorithm during RPU generation. The algorithm is applied after assigning FUs to stages. For each stage, the algorithm assigns at most two load/store operations to the available ports. If additional loads are present on the same stage, they must be placed next on the list so that the stage can execute to completion. Additional stores may be postponed in favor of load operations in the next stages. In this way, downstream stages may not need to wait until ports are free to execute any load operations they may contain. If a store exists in the same stage as a single load, the store is not postponed, since one port will still be available. For stages without load/store operations, nothing is added to the list, except pending stores, if any.

Figure 5 shows an example of the access order for four load/store FUs of an RPU with three activation stages (other FUs omitted). In the first cycle, *L1* is added to the list. There are no more memory operations in this stage, and the second stage cannot execute before the first completes. So, loads *L2* and *L3* on the following stage are assigned to a new list slot. As the processing cannot advance until all loads complete, *L4* is assigned the third slot. Stores *S1* and *S2* in the last stage are assigned the fourth slot, and two stores remain. Because stores can be postponed, the first stage can be activated again immediately, and the order assignment wraps around to slot one. One of the memory ports had already been assigned to *L1* in slot one, but a port remains free. The pending *S3* is assigned to that port in slot one. Moving to the next stage and advancing to slot two shows that there are no available ports, but slot three has one remaining free port, to which *S4* is assigned. Because all pending stores can execute before their units need to be activated again, they introduce no additional cycles.

The number of clock cycles required to execute one iteration given this ordering is shown on the left in Figure 5. All the load/store operations from the first iteration are completed on the eighth clock cycle, halfway during the second iteration, as a result of postponing stores. This strategy may lead to incorrect results in the presence of data dependencies that flow through memory locations, because stores can be delayed until after the corresponding load. For these cases, a conservative ordering strategy is used, which does not postpone stores.

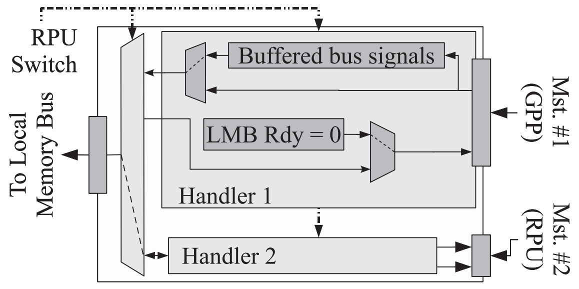


Fig. 6. Two-port LMB Multiplexer. Each master device interfaces with a handler.

4.3. The LMB Multiplexer

The LMB Multiplexer, shown in Figure 6, is a peripheral with three LMB ports. Two ports connect to bus master interfaces and a third connects to the actual LMB. This allows for two masters to access a single LMB and its slave devices. The multiplexer is functionally transparent. It does not add signals to the bus interfaces or clock cycles to data exchanges between the bus and a master. The transaction behavior on the bus is unaltered, and no modifications are required to either the bus or the master devices.

Both ports of the LMB Multiplexer are bidirectional. The module uses the bus signals to perform synchronization and allow for a gracious handover of bus control between the two masters. When a switch is requested, it occurs immediately after the end of the current transaction, if any. When switching, the outputs from the newly selected master are immediately connected to the bus. The bus response signals stay connected to the previous master for an additional clock cycle, so it can receive any pending response.

When a master is not selected, its requests are sent to a handler module, which buffers up to one access request. All downstream master signals are buffered when an address strobe is asserted. This is sufficient for correct operation, since the LMB interface is blocking. The handler module then holds the bus *ready* signal low, halting the master. When the halted master is reselected, the buffered request is sent first followed by any additional transactions. Each multiplexer also includes the same address mask as the memory controller of the bus it interfaces with, ignoring any requests that do not match the address range.

This setup allows the RPU to access the entire address range of the GPP's data memory and avoids data coherency issues between the GPP and the coprocessor. No memory address translation steps are necessary and access to heap-allocated data is directly supported.

5. EXPERIMENTAL RESULTS AND DISCUSSION

The proposed system was tested with 37 integer benchmarks from Texas Instrument's IMGLIB [Texas Instruments 2008], the SNU-RT [Seoul National University 2006] and Powerstone [Scott et al. 1998] suites, and other sources [Warren 2002]. All benchmarks include memory accesses.

We distinguish three sets of benchmarks: (A) simple benchmarks containing one computational kernel, (B) benchmarks whose source code has been manually *if-converted* to remove short *if-else* statements, and (C) a set of *applications* containing more complex benchmarks. Each benchmark from sets A and B contains only one candidate Megablock (unless otherwise noted). The nine benchmarks of set C are larger and contain multiple candidate Megablocks, producing RPUs with multiple configurations. Speedups and RPU characteristics are shown in Tables I–III.

Table I. RPU Characteristics and Results for Benchmark Set A

Benchmark	#Lds/Sts	#FUs	CPL	IPC_{HW}	Speedup (kernel)	Speedup
blit	1/2	13	3.5	2.20	2.48	2.46
bobhash	1/0	11	8	1.11	1.55	1.55
checkbits	2/2	64	16	3.71	3.90	3.89
checksum*	8/0	155	52	2.66	2.62	2.59
fft*	10/8	68	9.5	3.09	0.98	0.93
gouraud	0/1	15	6	2.83	2.99	2.98
lookup2	3/0	48	22	1.92	1.95	1.94
motionEst	2/1	13	7	1.63	1.31	1.29
perlins	4/1	123	29	3.97	3.98	3.97
popArray1	1/0	22	15	1.69	2.06	2.03
popArray2	3/0	46	20	2.55	2.70	2.19
quantize	1/1	11	6	1.57	2.08	2.07
sad16 × 16	2/0	14	8	1.33	1.30	1.29
ycDmxBe16*	4/4	20	3	2.80	3.96	3.94
WCET_fir	2/0	10	5	1.29	1.13	1.05
arithmetic mean	2.93/1.33	42	14	2.29	2.33	2.28
geometric mean	–/–	–	–	–	2.12	2.06

Benchmarks marked with (*) are part of the subset considered for evaluation of memory access optimizations.

Table II. RPU Characteristics and Speedups for Benchmark Set B (*If-Converted* Code)

Benchmark	#Lds/Sts	#FUs	CPL	IPC_{HW}	Speedup (kernel)	Speedup (if-converted)	Speedup
boundary	3/2	24	6	3.13	3.05	3.01	1.56
bubbleSort	2/2	21	10	1.58	1.81	1.80	0.96
chgBrght	1/1	20	12	1.54	1.70	1.68	1.12
compositing	2/1	24	15	1.50	1.97	1.96	1.53
conv3 × 3*	18/0	77	19	2.89	2.99	2.96	2.78
crc32	1/1	80	50	2.57	2.78	2.78	1.57
mad16 × 16	2/0	16	9	1.36	1.31	1.30	1.02
max	1/0	14	10	1.18	1.63	1.63	1.18
milRab16	6/6	52	6	5.90	1.21	0.79	0.65
perimeter	5/1	28	10	1.85	1.91	1.90	1.37
pixSat	1/1	21	14	1.27	1.66	1.66	1.07
rng	6/6	74	18	3.65	7.75	7.73	7.21
sobel	8/1	56	20	2.08	2.15	2.15	1.71
arithmetic mean	4.31/1.77	39	15.31	2.35	2.46	2.41	1.83
geometric mean	–/–	–	–	–	2.15	2.07	1.48

Benchmarks marked with (*) are part of the subset considered for evaluation of memory access optimizations.

Set B was created to improve execution coverage. Conditional constructs (i.e., *ifs* or *switches*) inside loops produce sequences of instructions with multiple execution paths. Since the Megablock is a single-path trace, this may result in several Megablocks, which execute with equal frequency; that is, no single one represents a clearly dominant path. Applying *if-conversion* [Allen et al. 1983] results in fewer but more frequently executed Megablocks [Bispo 2012]. If-conversion is a technique that transforms control dependencies into data dependencies. In our case, if-statements and their branches are transformed into straight-line code by using arithmetic and logical operations.

The benchmarks marked with an asterisk in Tables I–III have a high number of memory accesses and were selected for evaluating the impact of pairing load/store operations (cf. Section 3.2) and prioritizing loads during memory access scheduling (cf. Section 4.2). In Section 5.2, this scheme is compared with the baseline case where all FUs are scheduled using an ASAP scheme together with a conservative memory access ordering.

Table III. RPU Characteristics and Results for Benchmark Set C

Benchmark	#Cfgs.	#Lds/ Sts	#FUs	#Avg. En. FUs	Avg. CPL	IPC_{HW}	Speedup (kernel)	Speed-up	Improv. #clock
powerstone_adpcm	7	8/6	34	11	5.0	1.40	0.78	0.78	1.04
powerstone_bcnt*	1	25/1	98	98	13.0	2.77	2.27	2.21	2.21
powerstone_g3fax	3	2/1	18	8	4.7	1.50	1.12	1.10	1.10
powerstone_jpeg	8	10/11	120	34	7.6	3.45	0.80	0.76	1.52
powerstone_pocsag	2	12/0	62	48	11.0	2.94	2.36	1.15	1.15
WCET_adpcm	13	8/6	39	10	5.4	1.35	1.15	1.10	1.46
WCET_edn	6	7/4	45	18	7.5	1.70	0.99	0.96	1.28
fdct*	2	8/15	136	110	17.0	4.86	2.54	2.46	3.28
SNU_jfdctint	1	9/8	95	95	11.0	4.88	1.80	1.05	1.40
arithmetic mean	9	9.9/5.8	71.89	47.97	9.13	2.76	1.53	1.29	1.60
geometric mean	–	–/–	–	–	–	–	1.39	1.18	1.50

Benchmarks marked with (*) are part of the subset considered for evaluation of memory access optimizations.

The complete system was implemented on a Digilent Atlys board with a Xilinx Spartan-6 LX45 FPGA. Xilinx EDK 12.3 was used for RTL synthesis and bitstream generation. The system clock for the baseline case was set to 66MHz, and the MicroBlaze processor was synthesized for minimum instruction latency. Benchmarks were compiled with *mb-gcc 4.1.2* using the *-O2* flag. For the *rng* and WCET *adpcm* benchmarks, the MicroBlaze includes an integer divider.

To measure execution times, both the injector and the RPU were connected to a custom Processor Local Bus peripheral, which contains five timers. The following measurements were done: number of clock cycles required to execute the entire benchmark; number of clock cycles for execution on the RPU; number of memory access cycles (every cycle required to complete a stage after activation); number of overhead clock cycles introduced by the injector; and total number of cycles required to fully execute the Megablock (i.e., RPU execution plus the last Megablock iteration executed by the MicroBlaze).

5.1. Benchmark Results

Tables I–III summarize the RPU characteristics and the speedups obtained for each benchmark. The *#Lds/Sts* column shows the total number of load/store FUs in the RPU (not necessarily concurrent). The *#FUs* column shows the number of FUs in the RPU, including loads and stores. The average CPL of the datapath configurations is shown in the *Avg. CPL* column. The IPC_{HW} column shows the Instructions per Clock (IPC) achieved by RPU execution. The *Speedup (kernel)* column shows the speedup of the accelerated portions of the application. The rightmost column shows the overall benchmark speedup, including the execution of all the code and all overheads. The overheads are composed of the number of clock cycles required to execute the CRs (i.e., before and after RPU execution) and a few clock cycles (usually three) required by the injector to redirect execution to a CR.

The IPC value is computed as the quotient of the number of instructions per Megablock iteration by the number of clock cycles required to execute them. The average IPC value achieved in software for all benchmarks is $\mu = 0.94$ ($\sigma = 0.09$).

The IPC_{HW} value is also computed using the number of instructions per Megablock iteration. For execution on the RPU, the number of cycles required to complete an iteration equals the CPL of the active configuration plus the cycles added by multicycle FUs, mostly the load/store FUs.

Benchmark set A. Table I contains the results for the 15 benchmarks of set A. Only one Megablock was used for most benchmarks in this set, generating RPUs with one configuration. For *fft* and *blit*, two Megablocks were implemented. The Megablock for the *checksum* benchmark is the largest in all sets, with 149 GPP instructions and a single exit point. For this set, the geometric mean of the speedups is $\mu_g = 2.06$, and the arithmetic mean of IPC_{HW} is 2.29. As the Megablock code for these cases accounts for a large part of the execution time, speedups tend to be high, depending directly on the IPC_{HW} value.

The increase of the IPC_{HW} value depends on several factors. The dominant one is the presence of a high level of Instruction Level Parallelism (ILP) throughout all stages. As only one stage is enabled at a time, a high ILP per stage is required for a high IPC count. Configurations with many stages (i.e., high CPL) and few instructions per stage result in lower IPC. Executing loads concurrently also increases the IPC_{HW} value. As an extreme example, the 18 loads of *conv3x3* (from set B shown in Table II) require at least 19 cycles in software. By allocating four loads to the same activation stage, eight in another one, and four and two in another two, only nine memory access cycles are introduced during RPU execution.

The *ycDmxBe16* benchmark shows how optimized scheduling of memory operations can also increase the IPC. For this benchmark, the RPU's datapath executes four load operations and four stores in three stages. The number of memory access cycles was reduced by two, compared to the baseline case using ASAP scheduling for the memory accesses. This reduction was achieved by relocating loads and postponing stores to a later cycle during the following iteration. The nonoptimized IPC_{HW}value and speedup are 2.00 and 2.83×, while the optimized implementation achieves 2.80 and 3.94×, respectively. These memory access optimizations are more effective for cases where there are many load/store operations for a CDFG of comparatively low CPL.

Benchmark set B. Table II contains information regarding the 13 benchmarks of set B. The RPUs in this set have one configuration. The mean IPC_{HW} is 2.35. Column *Speedup (if-converted)* shows the speedup when compared to the execution of the *if-converted* code, while the last column contains the speedup versus the original (nonconverted) equivalent.

Note that the IPC_{HW} for this set is computed using the number of instructions in the accelerated *if-converted* trace. As the original assembly code differs from the converted version, and execution through it follows multiple paths, it is difficult to measure the number of original instructions that are being accelerated per iteration.

Comparing RPU execution versus the *if-converted* code for this benchmark set, we find that all benchmarks but *milRab16* are accelerated (the geometric mean speedup is $\mu_g = 2.07$). However, when compared to the original code, $\mu_g = 1.48$ and two slowdowns occur. The decrease in speedup occurs because *if-conversion* typically adds instructions to the trace. The RPU's datapath implements these larger traces, but it is effectively accelerating traces that were originally smaller, as nonconverted code, and executed in less time using the MicroBlaze.

Speedups occur if the IPC achieved by the *if-converted* Megablock is still more efficient than execution of the original nonconverted code, despite the instructions added due to *if-conversion*. This is the case for most benchmarks. However, the Megablock from *bubbleSort* has nearly twice the instructions of the nonconverted version. The number of cycles of execution on the datapath exceeds the number required for software execution of the nonconverted version, resulting in a slowdown.

As for *milRab16*, the trace still contains many branch instructions despite the *if-conversion*. Execution often migrates to the RPU and terminates immediately, as a different path through the control flow must be followed, introducing overhead every time the RPU is called, and causing a global slowdown.

The *rng* benchmark is the only instance where integer division by a constant occurs. The selected Megablock contains four such divisions. Each division requires 32 cycles in software, totaling 128 cycles and resulting in a software IPC of 0.40. The RPU implementation allowed for three of the divisions to execute concurrently in two clock cycles. Also, five of the six loads were scheduled on the same stage, leading to only 23 cycles required to complete an iteration. This leads to the highest speedup of 7.75×, despite the fact that the IPC for this case is only the fourth best in sets A and B.

Benchmark set C. Table III contains results for the *application* set, for which $\mu_g = 1.18$ and the mean IPC$_{\mathrm{HW}}$value equals 2.76. All but two of these benchmarks contain several Megablocks. The corresponding number of configurations are shown in column *#Cfgs*. The average number of active FUs is shown in the fifth column and the average CPL in the sixth. The average number of active loads and stores is 15.5 and 8.45, respectively. Depending on the similarity of the instruction sequences used to generate the RPU, there will be a greater utilization of the FUs; that is, there will be fewer FUs disabled per configuration. The IPC$_{\mathrm{HW}}$ value for these benchmarks is computed as the average IPC of all the configurations.

The *fdct* benchmark of set C shows how the latency of many store operations can be hidden when executing on the RPU. The RPU for this case has two configurations; each one issues eight load and eight store operations and has a CPL of 17. All the loads are scheduled in the same stage and complete in four cycles. A sufficient number of cycles is available throughout an iteration so that all stores can be completed without introducing additional cycles. In the worst-case scenario, where concurrent accesses are not allowed and store operations are blocking, the IPC$_{\mathrm{HW}}$ drops to 3.09, versus the attained 4.86.

The RPU with the most configurations occurs for WCET's *adpcm*. The generated RPU accelerates 13 Megablocks with an average of 7.7 instructions, which represents the smallest average number of accelerated instructions per Megablock. Considering all benchmarks, an average of 30 instructions are accelerated per RPU configuration. In comparison, the average number of instructions in the Megablocks presented in Bispo et al. [2013b] was 7.8. In that work, the lack of support for memory accesses prevented the migration of larger Megablocks.

The RPUs for the benchmarks of this set synthesized to lower frequencies, forcing the system to operate at frequencies lower than the 66MHz of the base case. Further details regarding the synthesis frequencies are given in the following subsection. Since speedups are based on execution time, a lower number of clock cycles does not necessarily imply a speedup when considering a lower operation frequency.

Since the focus of this work is to study the effect of supporting memory accesses, we also report the improvement of the clock cycle count, as shown in the *Improvement #clock cycles* column of Table III. In this way, it is possible to observe the potential speedup by exploiting the ILP of the Megablocks for a more frequency-efficient, but functionally equivalent, RPU implementation. The geometric mean speedup in this situation is $\mu_g = 1.50$ for set C, and $\mu_g = 1.74$ for all benchmarks.

5.2. General Aspects

Performance. The global average IPC$_{\mathrm{HW}}$ for all benchmarks is 2.42. The RPUs contain an average of 48 FUs, out of which 31 are enabled per configuration. Considering only cases with multiple configurations, 22 FUs out of 59 are enabled per configuration. The average number of load/store FUs is 5.11 and 2.57, respectively, and 3.29 loads and 1.63 stores occur per configuration. There are on average twice as many loads as stores, which is to be expected, because typically several input items produce one output item.

Memory operations introduce on average 2.50 cycles exclusively for memory accesses. Without the optimization steps, this number would increase to 2.65. Note that memory access optimizations are only significant for benchmarks with large numbers of memory accesses per iteration.

Overall, memory access cycles correspond to 17.5% ($\sigma = 14\%$) of the RPU execution time. For the *gouraud* benchmark, no cycles are spent in memory accesses, since it has only one store operation per iteration. The *bnct* benchmark has the highest number of memory access cycles, which account for 50.25% of the overall execution time. For the 25 loads of this kernel, the minimum possible latency of 13 is indeed achieved by pairing loads.

The overall and kernel speedups for all benchmarks are 1.60× (minimum: 0.65×; maximum: 7.21×) and 1.92× (minimum: 0.78×; maximum: 7.75×), respectively. Considering only cases with more than three configurations, the geometric mean speedup is $\mu_g = 0.92$; for cases were the IPC_{HW} value is above average, $\mu_g = 2.02$. In Paulino et al. [2013], the same system architecture was used, but the RPU was implemented with a higher-overhead bus interface and less optimized support for load/store operations. For the subset of 10 benchmarks evaluated in Paulino et al. [2013], we observe that the geometric mean speedup for the overall execution improves from 1.22 to 1.53.

The benchmarks marked with (*) have an average number of active load FUs greater than three. For these 13 benchmarks, the average IPC_{HW} and speedup for the baseline case are 3.31 and 1.92×, respectively. The optimizations increase these values to 3.41 and 1.99×. However, improvements only occurred for six of these cases. The others remained unaltered, because either the available slack did not allow for a better FU scheduling or the scheduling was already optimal. For the cases where performance did improve, the IPC_{HW} value increased from 3.14 to 3.38, and the speedup from 2.34× to 2.52× (geometric mean).

The GPP/RPU communication overhead accounts for an average of 6.4% of the time required for migration and RPU execution, and 3.6% of the total execution time. Each call of the RPU takes 27 clock cycles on average. The benchmark of set C with the longest execution time is *g3fax*: nearly 10 million clock cycles when using the RPU. Out of these, 13.8% are communication overhead. Despite the low overhead of the FSL interface, there are cases where the number of cycles of RPU execution per call is comparable to the number of cycles required for executing the CR. The impact of the communication overhead diminishes as the number of iterations performed per call of the RPU increases.

Resource usage and operation frequency. In order to save hardware, during RPU generation, the tools attempt to avoid instantiating new FUs by assigning operations to FUs already existing in previously generated configurations.

For benchmarks with multiple configurations, the tools save instantiating an average of 54.2 FUs. Without resorting to list scheduling, the average is 52.5. The number of FUs saved depends on the FU slack and greatly on the similarity of the Megablocks. For the eight configurations of *jpeg*, 150 operations could be mapped to already instantiated FUs. Without sharing FUs, 150 FUs would be added to the 120 actually being used. This corresponds to a reduction of 56%. The RPU for WCET's *adpcm* exhibits the largest reduction. The total of 100 Megablock operations for its 13 configurations are mapped into 39 FUs, a reduction of 70%. The number of saved FUs increases as more Megablocks are mapped to the RPU, because a larger pool of reusable resources becomes available. Figure 7 shows the resource usage and maximum operating frequency of the RPUs.

The speedup measurements were taken by synthesizing the system for a target frequency of 66MHz, except for six out of the nine benchmarks of set C, and for three other cases in the remaining sets. The systems for *adpcm (Powerstone)*, *jfdctint*, *adpcm*

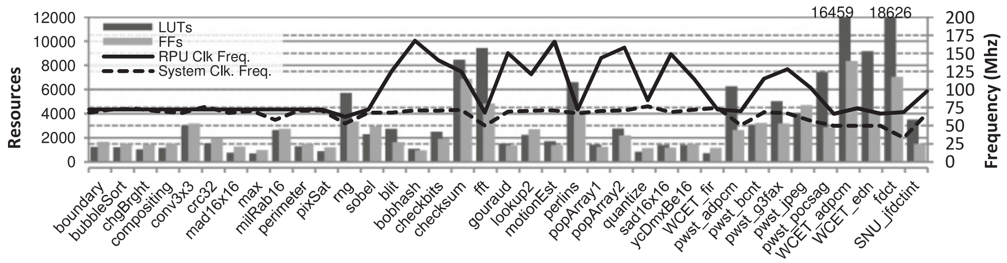


Fig. 7. Required LUTs and FFs and maximum operating clock frequency for the RPUs for the tested benchmarks.

(WCET), *fdct*, *edn*, *milRab16*, *rng,* and *fft* were synthesized for 50MHz, and Powerstone's *jpeg* was synthesized for 33MHz.

The average post-place-and-route maximum clock frequency for the system for all sets is 65.7MHz ($\sigma = 9.6\,\text{MHz}$), and 55MHz for set C alone. Considering only sets A and B, the average system frequency is 69MHz. The full system requires 6,262 Lookup Tables (LUTs) and 4,468 Flip-Flops (FFs) for all sets on average, respectively. This corresponds to 23% and 8% of the available FPGA device resources, respectively. For set C, 10,192 LUTs and 6,037 FFs are required on average. For comparison, a baseline system with a single MicroBlaze, buses, and memory requires a total of 1,850 LUTs and 1,314 FFs, out of which 1,308 LUTs and 986 FFs are for the MicroBlaze.

Despite some cases of system frequency below 66MHz, note that the average synthesis frequency for the RPU for all sets is 97MHz. The maximum is 168MHz (*bobhash*), and the minimum is 62MHz (*rng*). The critical path for this case is due to the constant integer division FU. For *milRab16*, *fft,* and all benchmarks of set C, the achieved RPU frequency is above 66MHz, with an average of 88MHz for set C. However, multiple configurations increase the complexity of the interstage connections, and a higher number of FUs and stages increases the amount of wiring required. The full design becomes too congested to route successfully for a target frequency of 66MHz.

Considering just the RPU, the required resources vary with the number of FUs and number of configurations. The average number of LUTs and FFs is 3,837 and 2,633, respectively; the RPU does not use any BRAMs. For sets A and B, the average number of required resources is 2,441 LUTs and 2,132 FFs, while for set C it is 8,180 and 4,194, respectively.

There is a clear trend between FF and number of FUs. As more FUs are instantiated, more register stages are required to buffer data. The number of required LUTs varies mostly due to the number of stages and configurations of the RPU. The benchmarks *perlins* and Powerstone's *jpeg* have nearly the same number of FUs, but the former uses 6,616 LUTs and the latter 18,626. The eight configurations of *jpeg* increase the complexity of the selectors, and because the RPU for this case has 17 stages, the required resources increase accordingly. WCET's *adpcm* has the most configurations, but the datapath is more compact (fewer FUs and stages), and thus requires only 9,176 LUTs.

In order to gain some insight on the general impact on power consumption, we analyzed the results reported by Xilinx's XPower Analyzer for a number of benchmarks of set C (all benchmarks except *powerstone's jpeg* and *fdct*). Post-place-and-route simulations were used to produce node activity information for both the baseline system and the RPU-enhanced systems. The simulations considered the execution to completion of each benchmark.

On average, the RPU-based system saves 1.9mW ($\sigma^2 = 0.1\text{mW}$). This corresponds to 0.86% of the total power consumption for a software-only system. For these cases, the

Table IV. Characteristics of Related Approaches

Approach	Base Processor	Memory Access	Accelerated Trace	Methodology
Warp	MicroBlaze	Single-port access for regular patterns	Most frequent innermost loops	Online binary disassembly and modification
DIM	MIPS	Concurrent accesses to random addresses	Sequences of basic blocks	Online binary translation
ADEXOR	MIPS	One store operation	Single-entry multiple-exit trace with multipath	Compile time binary modification
CCA	ARM	Not supported	Certain sequences of instructions forming at most 4 input / 2 output graphs	Compile time subgraph detection
This work	MicroBlaze	Two concurrent accesses	Megablocks	Offline discovery and accelerator generation

power required by the additional circuits is slightly more than the GPP power saved. Therefore, the performance improvements do not require additional power.

6. RELATED WORK

This section describes previous work on binary acceleration of applications with reconfigurable architectures. Table IV summarizes the main characteristics of the most relevant approaches.

The WARP processor is a MicroBlaze-based system that is able to migrate the innermost loops to an FPGA-based reconfigurable logic fabric connected to the MicroBlaze [Lysecky and Vahid 2009]. In WARP, program execution is profiled at runtime and in-system to select loops from frequent backward branches. Then, on-chip CAD tools decompile those loops and map them to the FPGA fabric. The reconfigurable fabric is loosely connected to the processor and has access to one port of the data BRAMs. Accelerated kernels cannot contain floating-point instructions, dynamic memory allocation calls, recursive function calls, or random memory access patterns. An RTL model of the custom FPGA was synthesized and simulated to provide functional validation. The rest of the system was implemented in a commercial FPGA and coupled to hardware modules with behavior identical to the generated configurations. An average speedup of $3.3\times$ over a MicroBlaze was reported for six applications of the Powerstone and EEMBC suites. In Stitt and Vahid [2011], the approach is applied to thread acceleration in a scenario with multiple processors and one reconfigurable fabric.

The DIM approach employs a 2-D array of functional units tightly coupled to a MIPS-based processor [Beck et al. 2008]. The array contains homogeneous rows of ALUs that include support for multiplications and loads. Floating-point and division operations are not supported. Inputs for the array are fetched from the processor's register file. The array is configured through binary translation. The instruction stream is transparently monitored and translated into array configurations concurrently with execution. The array is used to accelerate sequences of basic blocks. The authors evaluated the impact of the number of rows of the array and the number of concurrent memory accesses on performance. An average speedup of $2.5\times$ was achieved for 18 benchmarks of the MiBench suite.

ADEXOR is an instruction set extension approach [Noori et al. 2012]. A custom FU is coupled to an MIPS processor pipeline. The custom FU has eight inputs and six outputs, supports conditional execution, and contains 16 heterogeneous ALUs organized in rows. The custom FU supports integer and fixed-point arithmetic and a maximum of one store. Multiplication, division, and load operations are not supported. Frequent execution paths are detected offline by following forward branches, creating a single-entry multiple-exit instruction sequence. For short branches, the taken and non-taken directions can be used to form a multipath trace. These traces are transformed into instructions used to control the custom FU. The binary is then updated with the new instructions. The architecture was synthesized for a 0.18μm CMOS technology, and an average speedup of 1.87× was reported for 16 applications of the MiBench suite.

The CCA is a triangular-shaped array of FUs added to an ARM processor's pipeline and is capable of executing specific sequences of instruction in a single clock cycle [Clark et al. 2004]. Rows of the CCA alternate between two types of FUs, which support arithmetic and logic operations. Crossbar connections are located between neighboring rows. The CCA was designed using a quantitative approach, by selecting the most representative sequences of instructions from 29 applications. Sequences of instructions with branches, multiplications, shifts by statically undetermined amounts, divisions, and memory operations were not considered. Modification of source code is not required. The sequences of instructions are detected at compile time, with candidate regions delimited by special instructions. Delimited regions are transformed into CCA configurations at runtime. A fully online method is also presented. A CCA with depth 4 and a total of 15 FUs was synthesized for a 0.13μm CMOS technology. A mean speedup of 1.26× over a four-issue ARM processor is reported.

In Kim et al. [2011], a general analysis of memory access support issues for a generic family of loosely coupled, coarse-grained accelerators is performed. The accelerators have local single-port memory banks. Concurrent accesses to memories are issued, and an arbiter directs the requests to the correct port. Loops are scheduled to the accelerator under memory access constraints. Operations are scheduled in order to minimize simultaneous accesses to the same bank. To further decrease conflicts, data are split among the memory banks based on a compile-time analysis of access frequency and array sizes.

The presented approaches address the support for memory operations in a manner that is intrinsically different from our work. CCA and ADEXOR approaches are focused on instruction-set extensions. Their tight coupling to the processor pipeline does not provide an obvious method for concurrent memory accesses or for the issuing of multiple accesses by execution of custom instructions. Although the ADEXOR pipeline does support one store instruction, the lack of support for load operations restricts the achieved speedups. In the programs we analyzed, there are often several loads per store, most of them independent of each other. Thus, it is important to have support for simultaneous memory accesses in order to take advantage of the parallelism existent in those programs.

Our system allows for a loosely coupled accelerator, therefore avoiding modifications to the processor pipeline, but without incurring data transfer steps between main memory and local memories of the coprocessor. The WARP processor is similar in this respect as it uses a shared data memory between the accelerator and the main processor, avoiding data transfers and allowing for accesses to the entire address space. However, it is restricted to one access per clock cycle and to regular access patterns.

Each RPU in our approach is specifically tailored to a set of candidate kernels, and therefore includes all necessary FUs. Different sets of kernels require generating different RPUs. In contrast, approaches with a fixed set of FUs may not be able to map some kernels due to resource mismatch. This is an issue for CCA, ADEXOR, and DIM but is avoided by the finer-grained approach of Warp.

7. CONCLUSION

Transparent binary acceleration is a promising strategy for enhancing the performance of data-intensive embedded systems without requiring extensive changes to application development for each target platform. In the architecture discussed in this article, a

GPP is aided by an RPU tailored during synthesis to accelerate repetitive instruction sequences called Megablocks.

In our approach, Megablocks with memory accesses use the direct interface of the RPU to the GPP data memory. Access to the GPP's entire data memory avoids costly data transfer steps and does not introduce any synchronization issues between the GPP and the RPU. Furthermore, there is the need neither for address translation mechanisms nor for distributing data throughout a noncontiguous memory space. The RPU is able to execute many memory operations, and the memory access patterns or array sizes do not need to be known before execution, meaning the RPU can also operate on heap-allocated data.

The current memory-sharing scheme uses on-chip memories to store code and data. Applications requiring the use of external memories are not yet supported. Note, however, that current high-end FPGA devices can contain up to 1.6MB of on-chip memory, making our current approach a viable alternative for many applications.

Our proof-of-concept prototype is able to achieve relevant accelerations with an overall power consumption that is comparable to that of a system with a single GPP. A geometric mean speedup of 1.60× was achieved for a set of 37 benchmarks.

Future work will address support for external memories and more efficient execution models for the RPU. We also plan to extend the Megablock and its detection to support multipath execution. This will allow our system to cover larger sections of the execution traces and thus to possibly increase the overall speedup.

REFERENCES

J. R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. 1983. Conversion of control dependence to data dependence. In *Proceedings of the 10th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*. ACM, 177–189.

Antonio Carlos S. Beck, Mateus B. Rutzig, Georgi Gaydadjiev, and Luigi Carro. 2008. Transparent reconfigurable acceleration for heterogeneous embedded applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE'08)*. ACM, 1208–1213.

João Bispo and João M. P. Cardoso. 2010a. On identifying and optimizing instruction sequences for dynamic compilation. In *Proceedings of the International Conference on Field-Programmable Technology (FPT'10)*. 437–440.

João Bispo and João M. P. Cardoso. 2010b. On identifying segments of traces for dynamic compilation. In *Proceedings of the International Conference Field-Programmable Logic Applications (FPL'10)*. 263–266.

João Bispo, Nuno Paulino, João M. P. Cardoso, and João C. Ferreira. 2013a. Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units. *International Journal of Reconfigurable Computing* (2013), 20. Article ID 340316.

João Bispo, Nuno Paulino, João C. Ferreira, and João M. P. Cardoso. 2013b. Transparent trace-based binary acceleration for reconfigurable HW/SW systems. *IEEE Transactions on Industrial Informatics* 9, 3 (Aug. 2013), 1625–1634.

João Bispo. 2012. *Mapping Runtime-Detected Loops from Microprocessors to Reconfigurable Processing Units*. Ph.D. Dissertation. Instituto Superior susheel – Universidade susheel de Lisboa.

Nathan Clark, Jason Blome, Michael Chu, Scott Mahlke, Stuart Biles, and Krisztian Flautner. 2005. An architecture framework for transparent instruction set customization in embedded processors. In *Proceedings of the 32nd Annual International Symposium on Computer Architecture (ISCA'05)*. IEEE Computer Society, 272–283.

Nathan Clark, Manjunath Kudlur, Hyunchul Park, Scott Mahlke, and Krisztian Flautner. 2004. Application-specific processing on a general-purpose core via transparent instruction set customization. In *Proceedings of the 37th International Symposium on Microarchitecture (MICRO'04)*. 30–40.

Yongjoo Kim, Jongeun Lee, Aviral Shrivastava, and Yunheung Paek. 2011. Memory access optimization in compilation for coarse-grained reconfigurable architectures. *ACM Transactions on Design Automation of Electron. Syst.* 16, 4, Article 42 (Oct. 2011), 27 pages.

Roman L. Lysecky and Frank Vahid. 2009. Design and implementation of a MicroBlaze-based warp processor. *ACM Trans. Embedded Comput. Syst.* 8, 3, Article 22 (April 2009), 22 pages.

Hamid Noori, Farhad Mehdipour, Koji Inoue, and Kazuaki Murakami. 2012. Improving performance and energy efficiency of embedded processors via post-fabrication instruction set customization. *Journal of Supercomputing* 60, 2 (May 2012), 196–222.

Hamid Noori, Farhad Mehdipour, Kazuaki Murakami, Koji Inoue, and Morteza Saheb Zamani. 2008. An architecture framework for an adaptive extensible processor. *Journal of Supercomputing* 45, 3 (Sept. 2008), 313–340.

Jong Kyung Paek, Kiyoung Choi, and Jongeun Lee. 2011. Binary acceleration using coarse-grained reconfigurable architecture. *SIGARCH Computer Architecture News* 38, 4 (Jan. 2011), 33–39.

Nuno Paulino, João C. Ferreira, and João M. P. Cardoso. 2013. Architecture for transparent binary acceleration of loops with memory accesses. In *Proceedings of the 9th International Conference on Reconfigurable Computing: Architectures, Tools, and Applications (ARC'13)*. Springer-Verlag, 122–133.

Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. 1998. Designing the Low-Power M*CORE Architecture. In *Proceedings of the Power Driven Microarchitecture Workshop at the IEEE International Symposium on Circuits and Systems (ISCAS'98)*. Barcelona, Spain.

Seoul National University. 2006. SNU Real-Time Benchmarks. Retrieved from http://www.cprover.org/goto-cc/examples/snu.html.

Greg Stitt and Frank Vahid. 2011. Thread warping: Dynamic and transparent synthesis of thread accelerators. *ACM Transactions on Design Automation of Electronic Systems* 16, 3, Article 32, 21 pages.

Texas Instruments. 2008. TMS320C6000 Image Library (IMGLIB) - SPRC264. Retrieved from http://www.ti.com/tool/sprc264. (2008).

Henry S. Warren. 2002. *Hacker's Delight*. Addison-Wesley Longman.

Wayne Wolf. 2003. A decade of hardware/software codesign. *Computer* 36 (April 2003), 38–43.

Received June 2013; revised September 2013; accepted January 2014