2014 IEEE International Symposium on Parallel and Distributed Processing with Applications

Trace-Based Reconfigurable Acceleration with Data Cache and External Memory Support

Nuno Paulino, João Canas Ferreira
INESC TEC and Faculty of Engineering
University of Porto, Portugal
nuno.paulino@fe.up.pt, jcf@fe.up.pt

João M. P. Cardoso
Department of Informatics Engineering
INESC TEC and Faculty of Engineering
University of Porto, Portugal
jmpc@fe.up.pt

***Abstract*—This paper presents a binary acceleration approach based on extending a General Purpose Processor (GPP) with a Reconfigurable Processing Unit (RPU), both sharing an external data memory. In this approach repeating sequences of GPP instructions are migrated to the RPU. The RPU resources are selected and organized off-line using execution trace information. The RPU core is composed of Functional Units (FUs) that correspond to single CPU instructions. The FUs are arranged in stages of mutually independent operations. The RPU can enable several stages in tandem, depending on the data dependencies. External data memory accesses are handled by a configurable dual-port cache. A prototype implementation of the architecture on a Spartan-6 FPGA was validated with 12 benchmarks and achieved an overall geometric mean speedup of 1.91x.**

I. Introduction

Embedded applications with stringent performance requirements and running on reconfigurable systems often need to employ application-specific hardware accelerators to achieve the required performance. These dedicated hardware modules are tailored for the target application, as more efficient designs can be achieved by exploiting the detailed information available in such cases: the amount of data to process, inter-iteration data dependencies in loops, required data representation ranges, etc. However, a design process which requires software/hardware partitioning of the application and the customization of the hardware accelerator incurs long design and verification times, and may lead to an overall reduction of system flexibility.

To avoid this design effort, there are several approaches which automate the design of custom accelerators. Those that depend on the analysis of the run-time behaviour of the binary application code [1]–[5] can exploit information of the actual behaviour of the application (possibly workload-dependent) without requiring any changes to the traditional software development flow or tools. However, they cannot rely on sophisticated static code analyses.

Our previous work presented a binary acceleration approach in which the execution of frequently executed loops is transparently migrated at run-time to a Reconfigurable Processing Unit (RPU), a tailored co-processor [6]–[8]. To generate an application-specific RPU, the binary of the target application is profiled by an Instruction Set Simulator (ISS) to detect several megablock instruction traces [9]. Each is then translated into a configuration for a specific instance of an RPU tailored for the set of translated traces. The approach is intended to accelerate the execution of loops with many iterations, so that the time required for the migration can be amortized over a significant amount of computation.

An accelerator that does not support memory accesses may have its applicability limited, because many data-intensive computational kernels must process significant amounts of data. A previous version of our work presented in [8] efficiently supports on-chip Block RAMs (BRAMs). To keep the migration of the control flow transparent to the executing program and avoid data-consistency issues, the data memory is shared between RPU and General Purpose Processor (GPP). The RPU is capable of two concurrent memory accesses by directly interfacing with the GPP's local data BRAM through a bus sharing mechanism.

The system architecture presented in this paper significantly extends the previous with a more sophisticated control logic for the RPU's stages, and a shared external memory. Specifically, the new version has the following main features:

1) Shared external data memory for GPP and RPU through a dedicated dual-port data cache for the RPU;
2) More sophisticated control logic that allows for simultaneous activation of computing stages.

The complete implementation of the new hardware infrastructure is also evaluated for a set of 12 benchmarks. An overall geometric mean application speedup of $1.91\times$ is achieved and a significant number of memory accesses per megablock is performed through the accelerator to the external data memory.

This paper is organized as follows. The following section summarizes related work. Section III presents an overview of the system architecture and operation. Section IV details the aspects of the RPU. Section V discusses the dual-port cache coupled to the RPU, while Section VI presents and discusses the experimental results. Section VII concludes the paper.

II. Related Work

There have been a number of research efforts considering the migration of binaries to RPUs. The most well known include the DIM [2], the Warp processor [10], and the CCA [11]. Recent approaches include the work presented in [12] and the Dynamically Specialized Execution Resource (DySER) approach [3].

The DIM binary acceleration approach [2] is built around a 2D row-oriented coarse-grained reconfigurable array that is tightly coupled to a MIPS processor. The array is composed

978-1-4799-4293-0/14 $31.00 © 2014 IEEE
DOI 10.1109/ISPA.2014.29

of homogeneous rows, each one containing several Functional Units (FUs) of different types, including load/store units. A transparent runtime translation mechanism detects sequences of processor instructions between special control instructions. A configuration for the array is then generated by determining data dependencies among the instructions and assigning operations to the FU. The DIM binary translation mechanism and coupling to the processor result in a low migration overhead. For a subset of the MiBench benchmark suite, an average speedup of 2.66× was achieved (compared to a MIPS processor).

In [13], the authors propose a loop accelerator architecture. The approach is based on binary acceleration which employs a virtualization layer which abstracts the binary from the accelerator architecture. During runtime, a virtual machine translates unmodifed binary into accelerator control instructions. The accelerator itself is designed to execute modulo schedulable loops, and is composed of 1 CCA [11], 2 integer units, 2 double units, 16 float units and 4 memory address generators for load operations, which are time multiplexed to serve 16 load streams. The loops to map must have an II smaller than 16. The assumption is made that the accelerator can perform memory disambiguation, and the address generators only support regular access patterns. Results are attained by simulation using a modified compiler, and a mean speedup of $2.66\times$ is achieved.

A just-in-time configuration approach is presented in [12]. Short sequences of up to 3 instructions and Instruction Level Parallelism (ILP) of 2 are detected by offline profiling. A compilation step generates configurations for a Specialized Functional Unit (SFU) and patches the binary code to trigger its use. The SFU was designed through analysis of 21 applications from the Mibench [14] and MediaBench [15] benchmark suites. The SFU is tightly coupled to the processor pipeline, executes the hot instruction sequences in 1 clock cycle and supports logic and arithmetic operations. Memory or control flow operations are not supported. An average speedup of $1.10\times$ was achieved when coupling the SFU with a 5-stage, single-issue in-order RISC processor for 14 benchmarks from several suites.

In [3], multiple-path execution trees are detected at compile time. The detected trees are split into memory handling instructions, which are executed on the processor side, and computation instructions, which are moved to DySER blocks, heterogeneous 2D arrays of configurable FUs. The capabilities of the FUs were chosen based on a quantitative analysis. Custom instructions are added to the binary code for configuration and for transferring data from/to the DySER. Multiple DySER blocks can be coupled to the processor, and pipelining execution is supported. A geometric mean speedup of $2.1\times$ is achieved with two DySER blocks coupled to a dual-issue out-of-order processor.

The work presented in this paper is based on binary level detection of repetitively executed GPP instruction sequences. These sequences are then migrated to the RPU, making it a loop accelerator. The RPU is loosely coupled to the main processor, and although this introduces some overhead, it avoids the need for modifications to the base processor, the instruction set or the compiler.

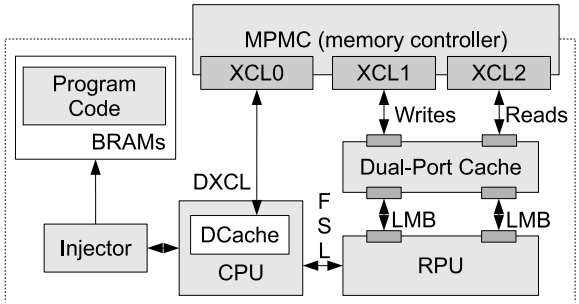


Fig. 1: System architecture overview.

III. Overview

Fig. 1 shows an overview of the core system. It contains: the GPP (a Microblaze processor) executing code located in local BRAMs; the RPU, an accelerator whose computational resources and possible configurations are specified at synthesis time; an injector module which intercepts the GPP's instruction bus; a custom configurable dual-port cache for the RPU; and a Multi-Port Memory Controller (MPMC) to access external data memory (not shown).

The BRAM memory contains only program code; every data element and array is located in external memory. In this prototype, an instruction cache for the GPP is not supported. The local code BRAM also contains communication subroutines. One subroutine per megablock is created by the RPU generation process. These routines control the transfer of operands/results between GPP and RPU. They are loaded into memory without disturbing the original application code.

A. Functional Description

Migration of the execution from GPP to RPU is accomplished by the injector module. It contains a list of predetermined addresses, each corresponding to the start of an identified megablock trace. When the GPP accesses an address in the list, the fetched instruction is replaced with a jump to the communication subroutine for the corresponding megablock. Then the GPP executes the subroutine instead of the original application code.

The subroutine transfers operands from the register file to the RPU via a Fast Simplex Link (FSL). Concurrently, the injector sends a configuration word to the RPU. This sets the data connections between the RPU's computing resources and configures the control logic. The RPU execution begins once all required operands are received. At the same time, the GPP begins invalidating its data cache to preserve data coherency. If the RPU execution time is long enough, this process introduces no additional overhead. GPP data cache invalidation occurs only if the trace being accelerated contains store operations.

The RPU then accelerates the execution of trace iterations by concurrently executing its instructions. The RPU has two Local Memory Bus (LMB) based memory ports that can be used to either connect directly to a local data BRAM or to the dual-port cache shown. The availability of two ports alleviates the memory access bottleneck. In many applications, loops which are good candidates for acceleration contain numerous

memory accesses. Support for concurrent accesses is then particularly important, because it enables the exploitation of data parallelism.

While the RPU is executing, the GPP stalls (with a blocking FSL *get* instruction) waiting for results, which are then placed in the register file by the communication subroutine. The subroutine ends with a jump back to the position where normal execution was interrupted, i.e., the first instruction of the accelerated trace. By doing so, the last iteration of the accelerated loop occurs in software. This allows for the control flow of the program to be followed correctly for cases in which the trace contains several exit points, i.e. branch instructions. Thus, this migration mechanism allows for an application to make use of the accelerator transparently, without need for modification of source code or binary.

B. Clock Domains

The system has two clock domains. The system-wide clock system drives the GPP, code memory, FSL links between GPP, RPU and injector, and MPMC. The RPU and its cache are driven by a different clock signal, whose frequency can be set according to the achievable maximum of each particular RPU. The RPU and cache contain synchronization circuits for data transfer across clock domain boundaries. A separate clock domain for the RPU allows for improved acceleration for those cases in which the RPU's maximum operating frequency exceeds that of the rest of the system (which can be constrained by long buses and other peripherals). When the RPU itself must operate at a lower frequency, the rest of the system need not be similarly constrained: effective acceleration is still possible if the ILP of the Control and Dataflow Graphs (CDFGs) which represent the accelerated traces exceeds the decrease in clock frequency relative to the system clock.

IV. Reconfigurable Processing Unit

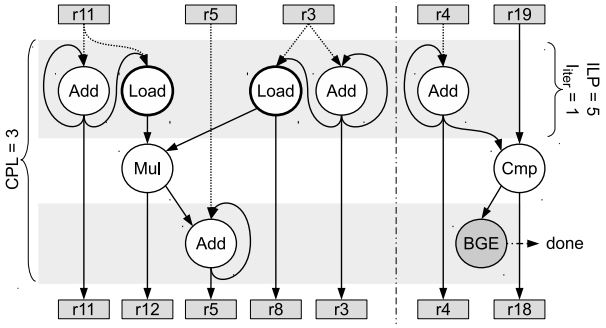
The RPU is an accelerator with a generic structure, which is tailored at synthesis time to accelerate the set of megablocks selected during profiling. A megablock is a dynamic trace representing a repeating pattern of instructions during execution. A specific RPU instance is generated by a set of custom tools using the tool flow described in [8].

A. Control and Dataflow Graphs

The megablocks selected for acceleration are converted to a CDFG representation. The trace shown in Fig. 2a, detected from the execution of an *fir* filter kernel, yields the CDFG shown in Fig. 2b. Since megablocks represent iterative patterns of instructions, the CDFGs contain backwards edges which represent the data and control dependencies between iterations. For the first pass through the CDFG, inputs are fetched from the GPP register file (dotted arrows). Afterwards, data is propagated within the CDFG and results are placed back into the register file. Registers on the top are repeated at the bottom, for simplicity of visualization. The value of register *r19* is never updated during execution, hence the solid arrow feeding the *Cmp* operation. Execution terminates at control nodes (e.g., *BGE* in Fig. 2b), which correspond to conditional branch instructions in the original trace.

```
...
0x300  bltid  r18,   -28          →  9:bge
0x304  addk   r5,    r5,    r12   →  10:add
0x2E4  lwi    r12,   r11,   0     →  1:load
0x2E8  lwi    r8,    r3,    0     →  3:load
0x2EC  addik  r4,    r4,    1     →  4:add
0x2F0  addik  r11,   r11,   4     →  5:add
0x2F4  mul    r12,   r12,   r8    →  6:mul
0x2F8  addik  r3,    r3,    -4    →  7:add
0x2FC  cmp    r18,   r19,   r4    →  8:cmp
0x300  bltid  r18,   -28          →  9:bge
0x304  addk   r5,    r5,    r12   →  10:add
0x2E4  lwi    r12,   r11,   0     →  1:load
0x2E8  lwi    r8,    r3,    0     →  3:load
...
```

(a) Small Megablock trace example



(b) Control and Dataflow Graph derived from the trace in *a)*

Fig. 2: Megablock trace and respective CDFG

The longest backwards connection in a CDFG dictates the iteration interval I_{iter}. When every node represents a single-cycle instruction, this means that an iteration can be completed every I_{iter} cycles. In Fig. 2, the longest backwards connection spans just one level, thus one iteration can be completed every cycle (the execution is fully-pipelined). In this situation, the number of instructions that can be completed per clock cycle, i.e., the Instructions per Cycle (IPC), is dependent not only on the average ILP of the individual stages, but also on the average number of operations completed by executing several CDFG levels simultaneously.

Previous versions of the RPU did not implement the shortest possible backwards connection between FUs and lacked control logic capable of activating several stages concurrently. Instead, the number of cycles to complete an iteration, C_{iter}, was equal to the longest path through the CDFG, at best. Achieved speedups were due only to the ILP of each activation stage. The implementation of the RPU presented here addresses this by supporting multiple simultaneous stage activations, as described next, thereby decreasing C_{iter}.

B. Structure

A simplified view of an RPU instance is shown in Fig. 3. It contains a set of FUs grouped in activation stages. The FUs in a stage execute data-independent operations in parallel. Also shown are the synchronization logic and the memory access control signals originating from stages which have load or store units. These signals are connected to the Memory Access Manager (MAM) (shown in Fig. 4), which controls access to

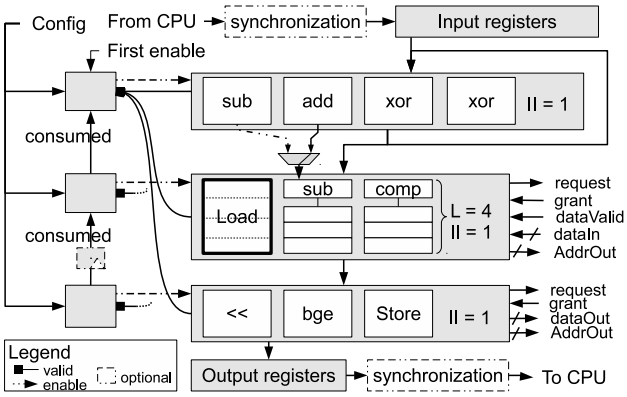


Fig. 3: Simplified RPU view with 3 stages, 2 memory operations and C_{iter}=1

the memory ports. Further details about memory access are given in Section IV-D.

A stage consists of a number of FU (number and type according to the mapping process). FUs correspond to machine code instructions. The supported operations are: loads/stores, integer arithmetic (including integer division by a constant) and logical operations. All FUs are single-cycle operations, except for the integer division by a constant (2 clock cycles), multiplication (3 clock cycles), and load operations (minimum of 4 clock cycles). The actual latency of load and store operations depends on the availability of memory ports. Since stores produce no data back into the array, the item to store may be buffered until access is granted at a later time. That is, a stage may finish executing despite containing outstanding stores. Latency is introduced by stores if the stage they are in needs to be activated again, but cannot be due to the outstanding request.

In order for a stage to be activated, valid data must be present at its inputs. Inputs for a stage can originate from the input registers of the RPU or from any stage of the array, the stage itself included. That is, feedback connections can be established mimicking the CDFG recursion. If any stage is fed data from a downstream stage, it will not be possible to enable that stage every cycle. Fig. 3 shows a synthetic example of a small array for which it is possible to do so. All data flows downwards, except for the first stage, which consumes the data it produced in the previous iteration. Note that the array does not have crossbar-type connectivity between all stages in order to support the different FU interconnection patterns of each configuration. A tailored multiplexer generated per FU input provides the minimum required connectivity for that input for each configuration. Fig. 3 exemplifies this with the multiplexer feeding the first input of the *sub* FU to the second stage. Inputs may also be fed constant values specified at synthesis time.

C. Execution

The RPU is idle until it receives a configuration word from the injector and a known number of operands from the GPP. The contents of the RPU's dual-port cache are invalidated prior to every execution, as the GPP might have made its contents incoherent with the external memory.

As iterations are completed in the array, the assertion of a *done* signal indicates that the execution flow must follow a different path, i.e., a sequence of instructions not part of the accelerated trace. The number of iterations does not need to be known offline. The same RPU configuration can iterate a different number of times per call depending on its input operands and termination conditions. A configuration of the RPU may not use all the existing stages, depending on the megablock being accelerated. For instance, only the first three stages of an RPU with five stages total may need to be activated to complete an iteration of a configuration; the output registers can be fed by any stage.

Per-stage control modules are shown on the left of Fig. 3. They receive a *ready* signal from the corresponding stage, data validity signals from all stages, and a notification signal, *consumed*, from the downstream control module. Each module generates the *enable* signal for its stage based on the presence of required valid data, the *ready* signal of the stage itself, and on whether or not its previously produced data has been consumed.

There are cases where a stage may require several cycles to execute despite it not being fed data from downstream stages. This occurs when the stage contains FUs which require more than one cycle to execute. For instance, the multiplication FU is a pipelined, three-cycle operation. For these cases, intra-stage pipelining is enabled. This is exemplified by the the second stage shown in Fig. 3, which contains a pipelined load FU whose minimum possible latency is four clock cycles. Registers are added to the other FUs to synchronize data and allow the stage to fill, thereby producing data every clock cycle. Intra-stage pipelining is configured on a per-stage basis during synthesis.

However, intra-stage pipelining is only effective if the number of load/store units and memory port availability allow for all accesses from the same activation to complete simultaneously. For instance, a stage with 3 loads would not be able to produce data every cycle due to lack of available ports. A stage with a single load FU could suffer from the same effect if additional accesses from other stages are outstanding. The example in Fig. 3 is a case for which the approach results in a C_{iter} of 1 clock cycle despite the memory access latency (assuming a single-cycle data cache for the RPU).

D. Memory access

Since load/store units receive operands from other FUs in the array, the RPU supports acceleration of loops with arbitrary memory access patterns. The size or location of data arrays, or the memory access patterns do not need to be known off-line. Load/store FUs assert an access request signal and wait until access is granted.

Fig. 4 shows the Memory Access Manager (MAM) module. It receives the *request* signals from all load/store FUs, and asserts back the *grant* signals. For load FUs, the data and the validity bit from the LMB master which performed the access are routed back into the appropriate load FU. Reads or writes to memory have a data width of 32 bits. Operations can use word, half-word or byte addresses. Byte selection is performed by each LMB Master module.

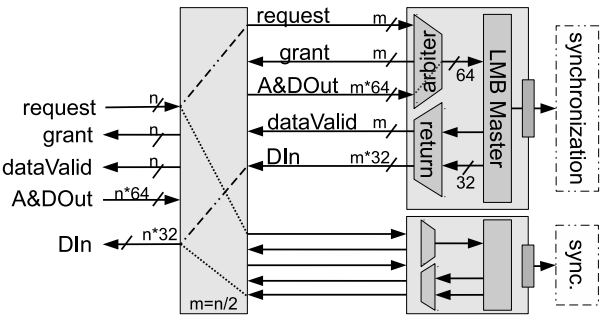


Fig. 4: Memory Access Manager (MAM) with each port handling half of the memory operations

Single-cycle arbitration logic grants access to up to two operations simultaneously. It works as follows: once access is granted to an FU, the next access will be granted only to FUs downstream from that FU. If no access requests are asserted within that range, access is granted to the uppermost FUs. This has the effect of preserving memory access order. Each LMB master is driven by one such arbiter. If more than seven memory operations are present in the array, the control signals are split into two groups, in order to avoid critical paths due to combinational logic in the arbiter, and each arbiter handles half the operations. Otherwise, both arbiters receive all request signals and make mutually exclusive selections.

V. Dual-Port Cache

The RPU can access the data in external memory through a dual-port cache. The two ports on the RPU side are composed of the same signals as an LMB interface: the same RPU instance can be connected either to the cache or directly to a local data BRAM. The memory controller interface of the cache is composed of two Xilinx Cache Links (XCLs). One port is used exclusively for write accesses, and the second for loading blocks into the cache. Requests are issued from the RPU side ports in order. However, the MPMC uses round robin arbitration by default. This caused some accesses to the external memory to be performed out of order. To ensure coherency, the cache uses a write-through no-allocate policy, and the XCL port used for writes is given priority.

The cache can operate in the same clock domain as the RPU. If so, the RPU does not instantiate synchronization logic at its LMB ports. Instead, synchronization is performed at the interface with the MPMC, if it operates at a different clock frequency. Loading data into memory at a higher rate than required by the RPU has the effect of reducing the impact of memory latency on the RPU.

Due to current performance-limiting delays in the connection between the RPU's logic and the cache's internal logic, an additional delay stage has been introduced for the time being at the LMB port interfaces. This increases the access latency to two clock cycles, causing some impact on RPU performance.

The cache is direct mapped, with configurable total and cache block size. Since the RPU side has two ports, one port can issue a load request that initiates a block load into the cache, while the other continues accessing data already present in the cache. Also, both ports can retrieve data from the same block as it is loaded into the cache. Since only one XCL port is used for load accesses, only one block can be loaded from external memory into the cache at a time. Data can also be written by one of the ports into a block being loaded without loss of coherency.

VI. Experimental Validation

The system was tested with 12 integer benchmarks from which appropriate megablocks for implementation could be extracted, i.e., traces with an acceptable number of load/store operations and whose total number of operations is not too large for the FPGA used in the implementation. Eight are simple data-oriented computation kernels; the remaining are applications from the powerstone [16] and WCET [17] benchmark suites. The *gridIterate* benchmark is the kernel of a 3D path planning application [18]. The implementation platform is a Digilent Atlys board, which contains a Spartan-6 LX45 FPGA and 128 MB of DDR2 memory. Xilinx EDK 14.6 was used for bitstream generation and the benchmarks were compiled with *mb-gcc 4.6.4* using the *-O2* flag.

The MicroBlaze is configured with an integer multiplier, a barrel-shifter, and a 256-byte data cache with a line size of 8 words. The RPU's cache has the same organization, and contains the input buffer stage at its LMB ports to avoid the current critical path delays of the RPU/cache interface. The system clock frequency is 83 MHz; the clock frequency of the RPU varies per-implementation. Two additional peripherals are used to measure execution times and provide other metrics, such as the number of iterations completed on the RPU, number of cycles spent on RPU execution, workload on the two memory ports and execution overheads.

A. Results

Table I summarizes the the number of cycles to complete an iteration, C_{iter}, of the accelerated traces for an ideal case, for software only execution, and for RPU execution. Also summarized are the measured speedups. The mean values in the last row of Table I are geometric means for the speedups and arithmetic means for the remaining columns. Column *Avg. #Inst* shows the average number of instructions executed in one iteration. The minimum possible C_{iter} is equal to the I_{iter} derived from the CDFGs. From this we compute the maximum potential IPC as the number of instructions over the I_{iter} of the CDFGs. The *Sw. IPC* is computed by dividing the number of clock cycles spent executing the megablocks in software by the number of instructions executed. Similarly, the *Hw. IPC* is computed from the actual value of C_{iter} for RPU execution. The C_{iter} in this case is equal to the total number of clock cycles required by RPU execution over the total number of iterations completed. Calculations of IPC and C_{iter} in Table I, as well as all clock cycle values given throughout this section are relative to the global system clock (83 MHz).

The last four columns contain: the speedup considering just the kernels S_k, the speedup for the entire application S_o, the application speedup without communication overheads S_{oh} and a speedup upper bound S_{ub}. The overhead excluded in the calculations of S_{oh} covers the time required for the GPP and RPU to exchange operands and results, and the time required

TABLE I: Performance Metrics

Benchmark	Avg. #Inst.	Min. I_{iter}	Max. IPC	Sw. C_{iter}	Sw. IPC	Hw. C_{iter}	Hw. IPC	S_k	S_o	S_{oh}	S_{ub}
blit	10.0	1	10.0	15.0	0.7	7.7	1.3	1.94	1.91	1.92	12.21
bobhash	10.0	5	2.0	15.8	0.6	7.3	1.4	2.14	2.10	2.11	3.03
checkbits	63.0	1	31.5	68.0	0.9	8.2	7.7	8.06	7.58	7.82	26.54
dotprod	12.0	2	12.0	16.6	0.7	9.1	1.3	1.76	1.65	1.72	7.31
fft	39.0	1	19.5	121.2	0.3	100.1	0.4	1.19	0.83	1.04	1.23
gouraud	19.0	1	19.0	21.0	0.9	3.1	6.1	6.65	6.39	6.45	18.42
perimeter	22.0	1	22.0	27.5	0.8	14.2	1.7	1.91	1.87	1.90	16.91
poparray1	27.0	1	27.0	35.7	0.7	6.0	4.5	5.21	3.86	4.24	9.25
gridIterate	120.0	5	24.0	278.9	0.4	245.7	0.5	1.16	1.15	1.24	2.37
powerstone_g3fax	5.4	2	2.7	10.0	0.5	7.1	0.8	1.02	0.93	1.12	1.67
WCET_edn	15.3	2	7.6	23.3	0.6	16.5	0.9	1.22	1.17	1.34	3.97
WCET_fir	9.0	1	9.0	17.8	0.5	17.6	0.5	0.93	0.96	1.03	2.55
mean	29.3	2	15.5	54.0	0.7	37.0	2.24	2.07	1.91	2.05	5.63

S_k (kernel speedup); S_o (overall application speedup); S_{oh} (speedup without communication overhead); S_{ub} (speedup upper bound)

to invalidate the GPP data cache when the RPU is called. The S_{ub} values are computed by comparing the time required for execution of the applications in software with the ideal case when the execution of the targeted traces is reduced to the shortest possible time. The fastest possible time to execute the traces was calculated by measuring how many iterations of the targeted traces are performed. We then took the I_{iter} of the corresponding CDFGs and computed the time required to execute those traces if iterations could be completed at that rate. By replacing the measured trace execution time with these values we arrive at the speedup upper bound S_{ub}.

B. General Considerations

a) IPC: As Tab. I shows, the RPU achieves consistently higher IPC values than the MicroBlaze GPP by completing trace iterations in fewer clock cycles. The ratio between *Hw.* IPC and *Sw.* IPC is indicative of the kernel speedup. If the accelerated traces represent a large part of the program, the overall application speedup increases. The *Hw.* IPC can be increased by exploiting ILP as much as possible. This is greatly restricted, in the cases studied, by memory accesses. For cases where the number of memory accesses in a trace exceed the number of arithmetic and other operations, the memory bottleneck effectively lowers the IPC. The benchmarks for which there are fewer memory accesses are also those for which the *Hw.* IPC is closer to the *Max.* IPC. The purpose of the dual-port data memory is to mitigate the impact of memory accesses. Measuring the number of accesses performed through each port provides insight into the overall memory workload and how it is distributed between both ports. One port issues an average of 502 accesses per call of the RPU, and the second issues 329. This demonstrates that a considerable data access parallelism was exploited through concurrent accesses. The following subsection further discusses the effects of memory accesses on performance.

b) RPU Workload: Over all benchmarks, the RPU executes a mean of 461 iterations per call, and is called an average of 2286 times. The more iterations performed per call, the better the overhead of communication and GPP cache invalidation is amortized. This is detailed in the section VI-D. An average of 830 memory accesses are issued per call (total for both ports), the maximum being 2887 for *perimeter* and the minimum 13 for *fft*.

c) Single clock domain speedup: The average frequency reported by the synthesis tools for the RPU is 128 MHz. The actual average frequency at which the RPU operated was 111 MHz. By knowing the number of clock cycles required for RPU execution, we can compute the RPU's execution time if using the system clock. In this situation, only a geometric mean speedup of $1.13\times$ would be achieved, with slowdowns occurring for 5 benchmarks (*fft* and the last 4 benchmarks of Tab. I). Note that the two-cycle latency of the RPU data-cache now becomes more detrimental to performance, since it is no longer mitigated by a higher RPU clock frequency.

d) Comparison with previous implementation: Most of the benchmarks presented in Tab. I (all except *gridIterate* and *dotprod*), were used to test an earlier implementation, which used local data memories for the RPU and GPP. For that subset, the mean geometric speedup for the overall application was $1.62\times$, versus the $2.03\times$ achieved here for the same subset.

e) Upper bound on speedup: The estimated upper bound (column 10 of Tab. I) is for a scenario with an accelerator capable of completing trace iterations at the highest possible rate. This also implies single-cycle memory accesses and infinite memory bandwidth. For the cases were $S_o > 1$, the system manages to implement an average of 21 % of the maximum potential speedup. The same calculation for all benchmarks using the values of S_{oh} yields 20 %. The *bobhash* benchmark is the case for which the achieved speedup, S_o, is closest to the upper bound: 54 % of the potential speedup of the megablock.

C. Impact of Memory Acesses

Table II summarizes RPU-related metrics. The first column contains the number of RPU configurations. The next column shows the average number of consecutive stages activated by the RPU to complete an iteration. Following are the average number of FUs used per configuration and the next column accounts only for the loads/stores out of those FUs.

TABLE II: RPU and System Characteristics

Benchmark	#Cfgs.	Avg. #Stages	Avg. #FUs.	Avg. #LDs/STs	RPU LUTs	RPU FFs	RPU Synt. Freq (MHz)	RPU Op. Freq (MHz)	$Sys.LUTs$	$Sys.FFs$
blit	2	3.5	12.0	1/2	3940	3798	145.5	125	8639	8094
bobhash	1	8.0	11.0	1/0	1583	2213	178.0	125	6793	6425
checkbits	1	19.0	59.0	1/1	1967	5068	127.1	120	7555	9310
dotprod	1	5.0	10.0	2/0	1333	2113	144.4	125	6637	6324
fft	1	10.0	48.0	6/4	4059	8839	131.7	115	9774	13254
gouraud	1	6.0	16.0	0/1	2417	3297	117.2	110	7382	7426
perimeter	1	10.0	28.0	5/1	2678	4706	113.2	105	8255	9087
poparray1	1	18.0	22.0	1/0	1516	3910	91.5	85	7064	8119
gridIterate	1	16.0	121.0	22/11	7771	16332	100.5	95	14326	20778
powerstone_g3fax	4	3.5	8.5	0.5/0	5029	3959	144.9	125	9942	8199
WCET_edn	2	8.0	22.5	3/0	6212	5745	134.3	100	10443	10025
WCET_fir	1	4.0	11.0	2/0	2342	3401	105.9	100	7543	7622
arithmetic mean	1.4	9.3	30.8	3.7/1.6	3404	5282	127.8	110.8	8696	9555

When connected to single-cycle instruction and data memories, the Microblaze executes approximately 0.98 instructions per clock cycle. Considering the average number of instructions in the accelerated traces, one iteration would be completed in an average of 30 clock cycles in this scenario. Due to the external memory latency, the *Sw.* IPC decreases to 0.7, meaning that 54 clock cycles are is required to complete an iteration. That is, external memory access introduces an additional 24 cycles.

Despite the exploitation of ILP and CDFG I_{iter} by the RPU, the memory accesses still restrict performance. The mean *Hw. IPC* is 2.24, and a mean of 37 clock cycles are required to complete an iteration. When compared to the ideal case, a mean of 35 additional clock cycles are introduced.

Memory accesses performed by the RPU are more costly for two reasons: (i) even though there are two ports between cache and RPU there is only one XCL port through which data is fetched into the cache from external memory; (ii) the 2 clock cycle cache latency makes memory accesses more costly for the RPU. The RPU does not necessarily suffer from twice the memory access latency, because (i) the dual-cache port allows for concurrent accesses, and (ii) the RPU operates at higher clock frequencies relative to the system clock for all cases.

The megablock for the *gridIterate* benchmark contains the largest number of memory operations: 22 loads and 11 stores. As a result, the achievable C_{iter} is 245. This lowers the *Hw. IPC* to 0.49, close to the *Sw. IPC* of 0.43. Each RPU iteration takes 240 more clock cycles than required without memory access constraints; software execution requires an additional 156 clock cycles compared to the base case of execution from local data memories. Despite the potential maximum IPC of 24 derived from the CDFG, the cost imposed by memory accesses prevents higher speedups for cases such as *gridIterate*. On average, there are only 0.80 non-memory operations per memory access for *gridIterate*. The RPU does not require twice as many cycles to handle all memory operations as the GPP because of the dual-port cache and the RPU clock frequency of 95 MHz.

The *fft* benchmark has the second highest number of memory operations. However, the lighter workload on the memory ports when compared to *gridIterate* allowed for completion of 1.16 instructions per memory access. Despite this, the kernel speedups for both cases are very similar, which suggests that a greater number of instructions needs to be executed per memory operation to compensate for the access latency. Although a kernel speedup is achieved for *fft*, the overall application slows down due to false calls to the RPU, as explained in the following subsection.

In contrast, the benchmarks with the fewest memory operations are *bobHash*, *gouraud* and *popArray*, with only either 1 load or 1 store. For these cases a higher speedup would be expected, since the RPU's ports are exclusively assigned to the memory operations. However, the megablock for *bobHash* has a minimum possible I_{iter} of 5 and a low average ILP per stage, achieving a lower speedup when compared to the two other cases. The best *Hw. IPC*, 7.69, occurs for the *checkbits* benchmark. Its megablock contained more instructions than all the other benchmarks except *gridIterate*. An average of 30.5 operations are completed per memory access. The consequent high IPC results in the largest kernel and overall speedups.

D. Overhead

The subroutines used to communicate with the RPU introduce two sources of overhead: the transfer of operands and results, and the invalidation of the GPP's cache.

Whilst the dual-port cache can be invalidated in a single cycle prior to RPU execution, the caches on the Microblaze can only be invalidated one cache line at a time. To invalidate the 256 byte cache, a total of 192 cycles are required. The overhead of cache invalidation can be effectively amortized whenever the RPU accelerates a large enough number of iterations per call. However, there are instances where the most frequent repeating patterns do not repeat consecutively. For example, one megablock of the *g3fax* benchmark iterates a total of 1967 times, but the average number of iterations per call is only 2.2. A low average number of iterations per call leads to significant overall overhead. This would not be the case for systems where the GPP cache can be invalidated in a single instruction.

An additional overhead is introduced by frequent RPU calls which terminate during the first iteration. That is, no useful work is performed on the RPU. This may happen if the number of iterations is dependent on the input data. In this case, any load accesses issued prior to termination could be aborted and an immediate return to the GPP performed. The current

implementation, however, requires that they complete before execution returns to software. The *fft* benchmark suffers from this overhead, as the RPU is called without performing useful work 127 times out of 128 total.

The speedups from column S_{OH} of Tab. I are computed without the overheads mentioned in this section, in order to show the expected gains from minimizing these overhead sources. The average overhead amounts to 4.94% of the total time.

E. Resource Usage

Table II also contains results regarding RPU resources, synthesis and operating frequencies, as well as the resources required for the entire system. The average RPU instance requires 3404 Lookup Tables (LUTs) and 5282 Flip Flops (FFs). These values are calculated from the post-synthesis reports. On average, the RPU requires 2.6× as many LUTs and 5.4× as many FFs as a Microblaze instance. For the entire system, the average number of LUTs and FFs is 8696 and 9555 respectively. These are post-place and route values.

Because the present implementation is capable of simultaneous stage activations, the execution can be fully-pipelined if connections and memory latency allow. To support this, registers are required at every stage to synchronize every item being propagated. Thus, an increase in resource requirements is noticeable when compared to previous work [8]. A subset of the benchmarks presented here (all except *gridIterate* and *dotprod*) was used to test an earlier non-pipelined implementation. The average numbers of system LUTs and FFs were 3341 and 2031, respectively.

VII. Conclusion

This paper presented an FPGA-based embedded system built around a general-purpose processor extended with a Reconfigurable Processing Unit (RPU) that can be transparently used by unmodified applications. The RPU executes iterative CDFGs derived from frequently repeating instruction traces. Specific RPU instances are tailored at synthesis-time to the needs of the target applications.

Relative our to previous work, cached external data memory accesses are fully supported and concurrent stage activation is explored for increased speedups. The actual level of pipelining on the RPU depends on the workload on the memory ports of the RPU, which are the only functional resource constraints. Execution on the RPU results in an average Instructions per Cycle of 2.24, versus 0.66 for software-only execution. For the 12 tested benchmarks, the overall geometric mean speedup is 1.91×. For the subset of the eight kernels, the speedup is 2.57×. For the previous non-pipelined implementation which supported only local memory access [8], the mean geometric speedup for all benchmarks (except *gridIterate* and *dotprod*), is 1.62×, versus the 2.03× reported in this paper for the same subset.

Future work will focus on extending the system with support for instruction caches, so that an application can reside entirely in external memory. The data cache will be improved for smaller access latency and the RPU architecture will be streamlined to reduce resource requirements.

References

[1] G. Stitt and F. Vahid, "Thread warping: Dynamic and transparent synthesis of thread accelerators," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, no. 3, Jun. 2011, article 32, 21 pages.

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

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

[4] R. Koenig, L. Bauer, T. Stripf, M. Shafique, W. Ahmed, J. Becker, and J. Henkel, "Kahrisma: A novel hypermorphic reconfigurable-instruction-set multi-grained-array architecture," in *Design, Automation Test in Europe Conference Exhibition (DATE'10)*, March 2010, pp. 819–824.

[5] G. Ansaloni, P. Bonzini, and L. Pozzi, "Design and architectural exploration of expression-grained reconfigurable arrays," in *Symposium on Application Specific Processors, (SASP'08)*, June 2008, pp. 26–33.

[6] J. Bispo, N. Paulino, J. M. Cardoso, and J. C. Ferreira, "Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units," *International Journal of Reconfigurable Computing*, 2013, article ID 340316.

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

[8] N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "A reconfigurable architecture for binary acceleration of loops with memory accesses (in print)," *ACM Trans. Reconfigurable Technol. Syst.*, 2014.

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

[10] R. Lysecky and F. Vahid, "Design and implementation of a microblaze-based warp processor," *ACM Trans. Embed. Comput. Syst.*, vol. 8, no. 3, Apr. 2009, article 22, 22 pages.

[11] 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. of the 32nd Annual Int'l Symposium on Computer Arch. (ISCA'05)*. IEEE Computer Society, 2005, pp. 272–283.

[12] L. Chen, J. Tarango, T. Mitra, and P. Brisk, "A just-in-time customizable processor," in *2013 IEEE/ACM International Conference on Computer-Aided Design (ICCAD'13)*, Nov 2013, pp. 524–531.

[13] N. Clark, A. Hormati, and S. Mahlke, "VEAL: Virtualized execution accelerator for loops," in *35th International Symposium on Computer Architecture (ISCA'08)*, june 2008, pp. 389–400.

[14] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown, "Mibench: A free, commercially representative embedded benchmark suite," in *Proceedings of the IEEE International Workshop on Workload Characterization (WWC'01)*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 3–14.

[15] C. Lee, M. Potkonjak, and W. Mangione-Smith, "Mediabench: a tool for evaluating and synthesizing multimedia and communications systems," in *Proceedings of the Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture, 1997*, Dec 1997, pp. 330–335.

[16] J. Scott, L. H. Lee, J. Arends, and B. Moyer, "Designing the low-power m*core architecture," in *Power Driven Microarchitecture Workshop at the IEEE Int'l Symp. on Circuits and Systems (ISCAS'98)*, Barcelona, Spain, Jun. 1998.

[17] Jan Gustafsson and Adam Betts and Andreas Ermedahl and Björn Lisper, "The Mälardalen WCET Benchmarks – Past, Present and Future," in *Proc. 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*, B. Lisper, Ed., Brussels, Belgium, Jul. 2010, pp. 137–147.

[18] J. M. P. Cardoso *et al.*, "REFLECT: rendering FPGAs to multi-core embedded computing," in *Reconfigurable Computing: From FPGAs to Hardware/Software Codesign*, J. M. P. Cardoso and M. Hübner, Eds. Springer, 2011.