Architecture for Transparent Binary Acceleration of Loops with Memory Accesses

Nuno Paulino[1], João Canas Ferreira[1], and João M. P. Cardoso[2]

[1] INESC TEC and Faculty of Engineering, University of Porto, Portugal
nuno.paulino@fe.up.pt, jcf@fe.up.pt
[2] INESC TEC and Department of Informatics Engineering, Faculty of Engineering, University of Porto, Portugal
jmpc@fe.up.pt

Abstract. This paper presents an extension to a hardware/software system architecture in which repetitive instruction traces, called Megablocks, are accelerated by a Reconfigurable Processing Unit (RPU). This scheme is supported by a custom toolchain able to automatically generate a RPU tailored for the execution of one or more Megablocks detected offline. Switching between hardware and software execution is done transparently, without modifications to source code or executable binaries. Our approach has been evaluated using an architecture with a MicroBlaze General Purpose Processor (GPP) softcore. By using a memory sharing mechanism, the RPU can access the GPP's data memory, allowing the acceleration of Megablocks with load/store operations. For a set of 21 embedded benchmarks, an average speedup of 1.43× is achieved, and a potential speedup of 2.09× is predicted for an implementation using a low overhead interface for communication between GPP and RPU.

Keywords: reconfigurable processor, memory access, Megablock, instruction trace, MicroBlaze, hardware acceleration, FPGA

1 Introduction

The use of dedicated hardware co-processors is an often-adopted solution to accelerate demanding computational kernels. However, the hardware/software (HW/SW) partitioning steps required to implement such co-processor based systems are time consuming, requiring hardware expertise and integration with a host system. Runtime reconfigurable coprocessor-based systems aim to resolve these issues by automatically and transparently accelerating demanding software kernels [1]. As a vast majority of such kernels operate on one or several input/output data arrays, often of unknown size at compile time, which may have random access patterns and data dependencies, it is important to focus on co-processors capable of performing memory access efficiently.

HW/SW partitioning approaches differ in terms of where the partitioning effort is applied. Typically, HW/SW partitioning applies high-level synthesis techniques to source code, e.g. analysis/modification, to exploit more powerful

optimizations to generate HW components. We, however, address a low level approach, usually referred to as binary acceleration, which attempts to find (either online or offline) suitable candidate instruction traces for acceleration when mapped to reconfigurable co-processors [2–5].

Several approaches have considered the use of RPUs acting as co-processors and providing memory operations. Kim et al. use an RPU with local memories and address operation scheduling to reduce access conflicts to the available ports [6]. Data arrays can be distributed among memories to optimize accesses. This requires a code analysis step to determine access patterns. Other authors specifically focus on binary acceleration. Beck et al. propose an approach to transparently map at runtime basic blocks of instruction traces into an RPU [7]. There can be as many concurrent memory accesses as available memory ports. Data access patterns can be random and known only at runtime. Paek et al. propose an offline binary dissassembly step to generate RPU configurations [3]. Acceleration is considered for data-dominant loops with the number of iterations known at compile time. Sequential memory accesses are supported, and data are passed to/from the RPU via a shared memory mechanism.

As an extension of previous work [8], this paper proposes an architecture for transparent binary acceleration which allows for an RPU to have transparent access to a shared main memory. In our approach, kernels to be mapped to the RPU are automatically identified from program execution traces, and are used to generate a dedicated RPU supporting up to two concurrent memory accesses, and including the Functional Units (FUs) needed to exploit the operation-level parallelism of the kernels. This dedicated RPU is then used to accelerate program execution transparently at runtime.

This paper is organized as follows. An overview of the proposed architecture is presented in Section 2. Section 3 details the RPU architecture and the handling of memory operations. Section 4 describes the module for transparent access to data memory by the RPU. Section 5 explains the tool flow, the experimental setup, and presents experimental results. Finally, Section 6 concludes the paper.

2 General Architecture Overview

The architecture and tools described in this paper are extensions of the ones presented in [8] in order to provide RPU support for memory accesses. They support the same four-step dynamic partitioning approach: (1) Loops within computational kernels are identified from execution traces and represented as Megablocks [9]; (2) Selected Megablocks (repeating code patterns) are transformed into a RPU specification and corresponding configurations by a custom toolchain, resulting in a specialized reconfigurable accelerator; (3) during runtime, mapped Megablocks are identified at the start of their execution when the GPP reaches the Megablock code; (4) execution of the mapped Megablocks is migrated transparently to the RPU.

The present work addresses the lack of support for memory accesses by our previous RPUs. Fig. 1 shows the enhanced system architecture, which is com-

posed of a program/data dual-port Block RAM (BRAM) connected to Local Memory Busses (LMBs), a GPP, an RPU connected to the GPP via a Processor Local Bus (PLB), two Local Memory Bus (LMB) Multiplexer modules, each connected to an LMB, and an LMB Injector module attached to the GPP's instruction bus.

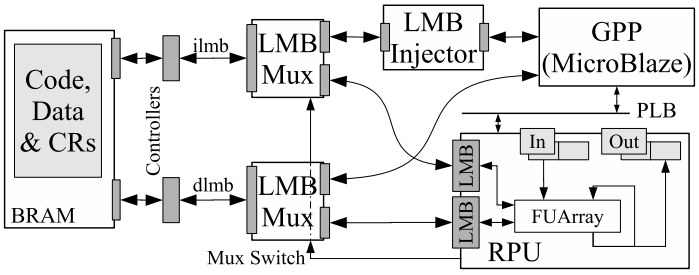

Fig. 1. Architecture overview. The RPU shares BRAM access with the GPP through the LMB Multiplexer

As in the previous version of our architecture, the MicroBlaze executes unmodified program code from local memories. The LMB Injector is responsible for the migration step, which 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 branches the GPP to a special subroutine that handles the communication between the GPP and the RPU.

The Communication Routine (CR) sends operands from the GPP's register file to the RPU through the peripheral bus, followed by a start signal. The RPU then gains control of the Local Memory Bus and accesses the BRAM by asserting the switch signals of the LMB Multiplexers. Each such module allows for two master devices to access a single-master LMB, sharing the entire address space of a BRAM without incurring any overhead, and without introducing data coherency issues. No memory address translation steps are necessary (cf. Sect. 4).

Once RPU execution ends, control of the LMBs is handed back to the GPP, which executes the remainder of the CR, recovering results to its register file and resuming software execution from the memory address where it was migrated.

The RPU can perform up to two simultaneous write/read accesses. Memory accesses can be random with addresses being calculated in the RPU. As multiple independent memory accesses occur in a wide range of Megablocks, access to both ports of the memory by the RPU allows for the exploration of this latent parallelism, with considerable potential speedups.

3 RPU Architecture

Fig. 2 shows the Reconfigurable Processing Unit (RPU) architecture (omitting the PLB interface). The RPU contains an array of Functional Units (FUs) tai-

lored for a specific set of Megablocks [8]. Each row contains a number of FUs able to execute in parallel. The FU layout shown in Fig. 2 represents a synthetic example with 3 rows (details are omitted for clarity). Data are received from the preceding row and propagated to the next. Passthrough components are inserted to enable connections between FUs on non-adjacent rows. An iteration completes when all rows have computed their results, and data is fed back to the first row. FUs are reused between configurations, and the depth (no. of rows) of the RPU equals the longest Critical Path Length (CPL) of the dataflow graphs representing all the implemented Megablocks.

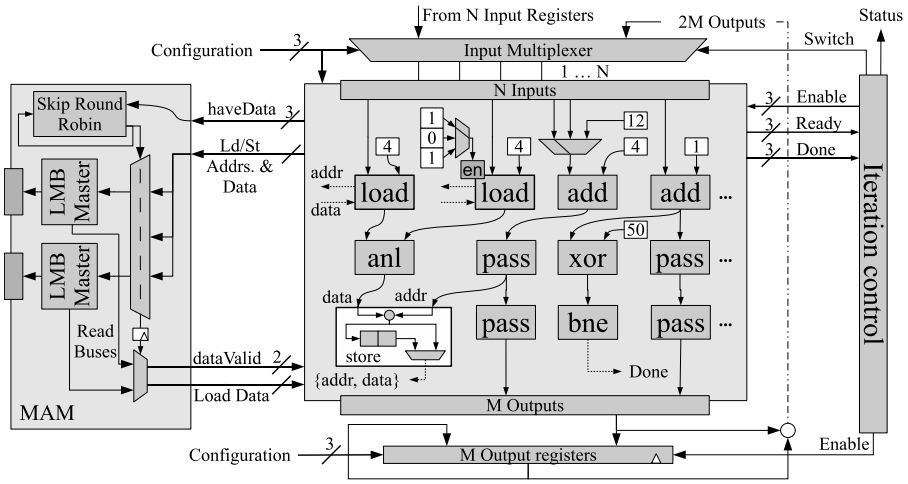

Fig. 2. Simplified diagram of the RPU's internal architecture, including the memory access handling mechanisms.

The RPU executes the equivalent of single path instructions traces that cross control-flow boundaries, and so it has one entry point and several exit points. This is exemplified in Fig. 2 by the *bne* operation which triggers a *Done* signal. When any of these exit operations is triggered, the current iteration is discarded and software execution is resumed.

Memory operations are implemented by special FUs. These special FUs are not single-cycle and may have a variable latency. To support multi-cycle operations, each row of the array generates a ready signal which is asserted when all multi-cycle FUs on that row assert their individual ready signals. The *Iteration Control* module issues an *Enable* signal per row, and checks the status of that row's *Ready* signal immediately after issuing the enable. If the row is not ready, the control logic waits until it can issue the enable, thereby stalling the array while memory requests are handled Memory accesses are managed by the Memory Access Manager (MAM) shown on the left in Fig. 2. This module and details about the load/store FUs are presented in Section 3.2. The MAM receives data

and addresses from all load/store FUs in the array. To determine the width of some ports, the number of memory operations the MAM can handle is specified at synthesis time. Actual memory accesses are performed via the RPU's two LMB ports, which interface with the LMBs through the LMB Multiplexers. As we currently use a dual-port BRAM, the number of ports in this implementation is limited to two.

3.1 Reconfiguration

Reconfiguration of the RPU is done by re-routing operands and by enabling or disabling FUs. To select a configuration before activation of the RPU, a configuration register is written to during CRs execution by the MicroBlaze.

The toolchain produces per-row HDL specifications of the connections between FUs of adjacent rows, thus minimizing the required resources to support all configurations. Each FU input is driven by a selector which is tailored at synthesis time to output one of a number of sources equal to the number of configurations. The Megablock extraction performs constant propagation, leading to some FU inputs remaining constant for all iterations during an execution. Instead of feeding the RPU with constant values at runtime, they are specified in the configuration. The inter-row selectors can either fetch values from any one of the outputs of the previous row or feed the input they drive with synthesis-time specified constants. These row interconnections are omitted for clarity from the example of Fig. 2, except for the one associated with the *add* FU in the first row, which exemplifies three configurations . In this case, for two configurations, the first input of the *add* is fed with a value from the N inputs available to the array (either values from the input registers or feedback values), and for the third configuration a constant value is supplied to the FU. There is one such selector per FU input. If only one configuration is present, the inter-row connections are optimized into wires.

Not all FUs in the RPU are actually used by each specific configuration. Although data may be fed to operations such as additions and other arithmetic even when their outputs are not used during execution, unused memory and exit FUs must be disabled. Thus, each FU is also driven by an enable signal. One of the *load* operations in Fig. 2 shows the enable signal being driven high by the selected configuration. The following excerpts from the RPU specification generated for the *chgBrghtB* benchmark illustrate how the inter-row selectors and enable signals are specified by tool-generated parameters at synthesis time.

```
parameter [0:32*(N_ROWS*N_COLS)-1]
FU_ENABLES = {
 {32'b11,  (...)  },
 {32'b11,  (...)  },
 {32'b01,  (...)  },
 (...)
 {32'b01,  (...)  }
}; // bit encoded enable signals
```

```
parameter [0:(33*9*N_CONFIGS)-1]
ROW3_CONFIGS = {
 // Config. 1      //Config. 2
 {32'h0,    1'b0}, {32'h0, 1'b0},
 {32'hffff, 1'b1}, {32'h0, 1'b0},
 (...)
 {32'h0,    1'b0}, {32'hff,1'b1}
}; // configurations per FU input
```

The left excerpt shows the enable bits for the FUs of the first column. The first two FUs are used in both configurations, while the third only in the first. This scheme of configuration specification limits the number of possible configurations to 32. The right excerpt shows the specification for 3 selectors of one of the rows of the RPU. Each row of defines one selector and each column represents a configuration. A 32-bit parameter specifies either a constant value or output index of the previous row. A 33rd bit distinguishes between each case. In this example, the second selector feeds a constant (32'hFFFF) to its associated input in configuration 1 and the first output of the previous row in configuration 2.

3.2 Memory Access Management

The RPU supports up to two simultaneous memory accesses by using the LMB Mux to interface with the BRAM ports. The RPU supports read/write operations of bytes, half-words or 32-bit words by using the byte enable signals of the LMB. The byte size of a memory operation is specified at runtime by one of the inputs to the load/store FUs. Since the architecture is not restricted by different address spaces for RPU and GPP, and because the RPU may receive memory addresses as operands from the GPP at runtime, access to heap allocated data is also possible.

Execution of a memory operation on the FU array is decoupled from the memory access itself. That is, store and load FUs only issue memory access requests to the MAM. Thus, more than two memory accesses can be issued in the same clock cycle. The RPU may or may not stall execution until they are processed, depending on the combination of operations and the current state.

If the concurrent memory operations are stores, the values to be written out are either immediately sent to memory (if both ports are free), or are instead. Since stores produce no data for use in the FU array, they introduce no latency, and only stall the RPU if execution reaches their row before any buffered data has not been written. For instance, two store operations occurring every 3 cycles can be written to memory using only one memory port without stalling. Once execution on the RPU ends, no more store operations are issued and buffered data are flushed out to memory before the RPU releases the LMB Mux switches, and allows the MicroBlaze to continue execution. Since the last iteration must be discarded and executed in software, stores must not be scheduled before the first enabled exit operation to maintain coherency. So, we adopt an As Late as Possible (ALAP) scheduling scheme for stores. If dependent loads occur a sufficient number of clock cycles after the issuing of the corresponding stores, no problems occur. The number of cycles required varies with the number of stores to be performed, and with the availability of the LMB ports on the MAM when they are issued. The placement mechanism does not yet analyze RAW dependencies.

Load operations behave differently, because they have no slack (due to the way Megablocks are generated): the loaded value is always required by the following row. When a load operation is issued, the RPU stalls until it is handled.

Two load operations in the same row can be handled simultaneously, introducing a latency of 1 clock cycle. If more are present, each one must be handled in order for execution to advance. There is currently no restriction on the number of allowed loads per row. The evaluated benchmarks have a maximum of 3 concurrent loads (*perimeter* benchmark).

The order in which the memory operations are treated is dictated by a selection logic performed by the MAM. Memory operations are treated in the same clock cycle they are issued, if a port is available. Each port is assigned a memory operation to handle in a round-robin fashion, which skips operations that do not have their request signal asserted in order to reduce latency. Both ports cannot choose the same operation simultaneously. For load operations, the selection logic directs the read data bus of a given port to the respective load FU.

Although this architecture allows for the minimum possible latency for both loads and stores, the selection logic can introduce critical path delays, if the FU array contains many memory operations. For the tested benchmarks *fft* and *perimeter*, which have 8 and 6 memory operations, respectively, the maximum operating frequency of each RPU is comparable to the delay introduced by using one hardware multiplier on our target FPGA.

4 The LMB Multiplexer

The LMB Multiplexer, shown in Fig. 3, is a peripheral with three LMB ports. Two ports connect to bus masters and a third connects to the actual Local Memory Bus. This allows for two masters to access a single LMB and its slave devices. The multiplexer is completely 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.

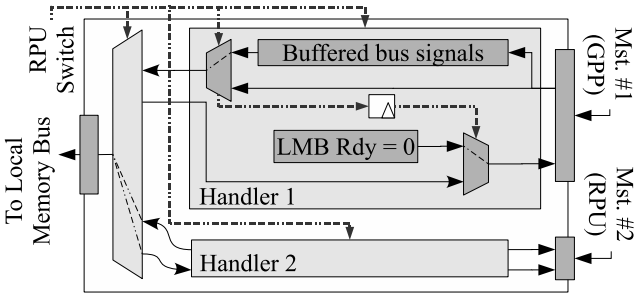

Fig. 3. Two port LMB Multiplexer. Each master device is treated by a handler.

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 only if there is no unfinished bus transaction or if a response to the last transaction is already present on the bus lines (so as not to issue another request untimely). When switching, the outputs from the newly selected master are immediately connected to the bus; the bus response signals are only sent to the newly-selected master in the next clock cycle, so that the response to the previous transaction is sent to the previously selected master.

When a master is not selected, its requests are sent to a handler module which buffers up to one access request. The handler buffers all master downstream signals when an address strobe is asserted. This is sufficient, since the LMB interface is blocking. If a request is buffered, the handler module holds the LMB Ready signal low, which halts the master. When a handler has a buffered request and the master of that handler is reselected a one-cycle delay is added. The buffered request must be strobed to the bus first. The master reads back the bus response and is then ready to perform more transactions. Since the MicroBlaze does not timeout when attempting to access the LMB it can wait indefinitely. Switching between bus masters requires no additional handshaking. Each LMB Mux 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.

5 Experimental Evaluation

The toolchain that supports the presented approach is an extension of the tools described in detail in [8]. The Megablock Extractor processes the executable files, simulates them, and uses the trace information to identify candidate Megablocks. These are passed to the RPU synthesis tool, which generates a parameterized HDL description of the RPU along with the configuration information. A second tool produces additional HDL specifications for the LMB Injector and CR code.

The CRs are added to the executable by being packed into arrays and compiled along with the benchmark, and then linked to predefined memory positions. The toolchain can produce CRs and HDL for a system in which the GPP/RPU communication is done through the PLB, or for a variant where modules are connected by low-overhead point-to-point Fast Simplex Links (FSLs). The results presented here were obtained with a PLB-based implementation.

5.1 Benchmark Results

The RPU's memory access mechanism was tested with 18 benchmarks selected from Texas Instrument's IMGLIB, from the SNU-RT Benchmark Suite and other assorted sources [10–12]. For most of these benchmarks, only one Megablock was implemented. Three additional synthetic benchmarks were written to produce RPUs implementing several Megablocks, for the sake of validation. The resulting RPUs have at least one memory operation.

The test bed was a Digilent Atlys board with a Xilinx Spartan 6 LX45 FPGA. Xilinx EDK 12.3 was used for synthesis and bitstream generation, the

system clock was set to 66 MHz, and the MicroBlaze processor was synthesized for minimum instruction latency. Benchmarks were compiled by *mb-gcc 4.1.2* with the -O2 flag.

The chosen kernels operate on data arrays of various sizes. For the *SNU-crc* benchmark there are two load operations, one which performs a load of a half-word and another, a byte addressed load. The RPU often receives operands which are memory positions of data arrays, and the addresses for data accesses are computed during execution. The number of concurrent memory accesses allows for a good use of the RPU's memory ports, as most generated RPU configurations have at most two simultaneous loads/stores. Synthetic benchmarks *synt1*, *synt2* and *synt3* are sequences of calls to routines belonging to other benchmarks. Benchmarks *synt1* and *synt2* have 6 configurations, and *synt3* has 3. To evaluate the worst case scenario, the benchmarks were written so that each set of kernels is called 500 times with an RPU reconfiguration on every call.

Table 1 summarizes the characteristics of the generated RPUs, the Instructions per Clock (IPC) achieved by hardware and the software IPC, for comparison. The *#Lds/Sts* column contains the number of load/store FUs in the RPU (not necessarily concurrent). The *#Ops* and *#Passes* columns specify how many FUs are actual operations (loads/stores included) and how many are passthroughs. The *#Rows* column refers to the number of rows of the RPU.

The *Hw. IPC* is the ratio between the number of enabled FUs (excluding pass-through components) and the number of cycles required to execute all of the rows of the RPU (i.e. one loop iteration). The number of cycles required does not equal the number of rows due to the multi-cycle memory operations (whose latency depends on their concurrency). For the tested benchmarks (excluding synthetic examples), memory operations introduce an average latency of 2.33 clock cycles. For cases with more than one configuration, the *Sw. IPC* was computed as the average of the IPCs for all corresponding Megablocks. The *Hw. IPC* was computed from the average number of enabled FUs per configuration and the average number of clock cycles required to complete an iteration.

The number of execution clock cycles was measured using a custom timer peripheral. The following measurements were made: 1) number of cycles during which the RPU is stalled; 2) number of cycles spent executing operations on the RPU (stall cycles included); 3) number of cycles required to execute the mapped Megablocks (in hardware or software) including communication and other overheads introduced by the migration mechanism; and 4) number of the cycles spent executing the entire benchmark, again including all overheads. Measurements for both hardware and software execution can be taken from the same implementation as the migration step can be easily disabled.

The *Spd.* column refers to the overall benchmark speedup, computed from 4). Overhead introduced by the execution of the CRs (i.e. GPP-RPU communication over the PLB) was derived from 2) and 3). The last column contains the potential overall speedup were this overhead completely removed. For the chosen benchmarks, the overhead accounts for an average of 32.9 % of the time required for migration and RPU execution. Each call of the RPU takes an average of 193

Table 1. RPU Characteristics and achieved Speedups

Benchmark	#Lds/Sts	#Ops	#Passes	#Rows	Hw.IPC	Sw.IPC	Spd.	Spd.(no OH)
blit1	1/1	10	17	3	2.50	0.92	1.45	1.47
chgBrght1	1/1	11	31	7	1.38	0.92	0.97	1.20
chgBrght2	1/1	11	20	5	1.83	0.91	0.54	1.80
quantize	1/1	11	35	6	1.57	0.92	1.90	2.14
SNU_crc	2/0	16	29	9	1.45	0.92	1.01	1.02
blit[1*]	1/2	14	27	4	2.10	0.92	2.38	2.48
boundary[1]	1/2	12	18	3	3.00	0.93	1.17	3.73
dotprod[1]	2/0	9	11	4	1.80	0.88	1.77	1.80
fir2[1]	2/1	12	17	4	2.00	0.91	1.40	1.78
perimeter[1]	5/1	19	12	3	3.17	0.94	1.82	2.51
bob_hash[2]	1/0	11	24	8	1.22	0.91	1.53	1.55
chgBrghtB[2*]	1/2	16	31	7	1.38	0.92	0.47	2.22
fft[2]	4/4	30	78	7	3.00	0.96	0.52	2.27
motEstim[2]	2/1	13	48	7	1.63	0.93	0.54	1.75
sad_8x8 [2]	2/0	14	39	8	1.56	0.92	0.52	1.73
checkbits[3]	1/1	64	169	16	3.65	0.98	3.56	3.94
compositing[3]	2/1	18	78	10	1.64	0.95	2.09	2.27
pop_array1[3]	1/0	22	94	15	1.38	0.96	1.86	2.11
synt1	6/3	36	27	4	2.36	0.91	2.09	2.52
synt2	6/5	53	88	8	1.79	0.92	0.68	1.64
synt3	4/3	93	180	16	2.00	0.97	1.85	1.97

[1]Included in *synt1* [2]Included in *synt2*; [3]Included in *synt3*.
[*]Benchmark has two Megablocks.

clock cycles. Even so, the average speedup achieved for the Megablocks alone is 1.52×. The overall speedup (with overhead included) is 1.41×. In some cases the overhead imposed an overall slowdown, although the mapped Megablocks were accelerated. By eliminating this overhead (for instance, using a FSL between the GPP and the RPU) the average potential speedup is 2.10×, and a speedup occurs for all cases. These averages do not include synthetic benchmarks.

5.2 Discussion

The benchmarks for which the *Hw. IPC* is largest are the ones with the greatest potential speedup. Memory operations reduce the IPC because of the latency they introduce, which accounts for stall cycles. Stalls only occur when more than 2 simultaneous memory operations are issued. Even then, this depends on the type of memory operation, since stores can be buffered and handled in later cycles. Therefore, latency above the minimum possible memory access time is only introduced when a load operation cannot be handled in the cycle it is issued. This occurs for the *perimeter* benchmark, but its execution still achieves the third best speedup (*w/o* overhead) of all the tested benchmarks.

The slowdowns that occur are due to the small number of iterations performed in hardware, which do not make up for the communication overhead.

Stall cycles account for an average 19.5 % of the total number of cycles spent on the RPU, excluding synthetic benchmarks. The largest stall time (57.1 % of the total execution time) occurs for the *perimeter* benchmark due to two consecutive rows with 3 concurrent accesses each. Execution of the *checkbits* benchmark stalls only 5.91 % of the time, because it has only one load operation. Although the *compositing* benchmark contains two loads and one store, only the two loads occur in the same row, introducing only the minimum latency of one clock cycle and resulting in the third lowest stall time of 9.12 %.

Regarding resources, the generated RPUs use on average 6.3 % of the available 27288 Lookup Tables (LUTs) and 4.3 % of 54576 Flip Flops (FFs). Maximum utilization for these resources is, respectively, 15.1 % (*pop_array1* benchmark) and 8.5 % (*fft* benchmark), while the minimum values are 2.1 % and 1.8 % (both for the *dotprod* benchmark). The average synthesis frequency of the individual RPU was 114 MHz. The lowest frequency, 62 MHz, occurs for the *fft* RPU. The associated critical path delay is due to the MAM selection logic. The highest frequency is 170 MHz for, *chgBrght1*. Although the frequency of all RPUs, save for *fft*, exceeds the base 66 MHz system clock used for most benchmarks, the clock frequency had to be lowered to 50 MHz (for the *perimeter*, *fft* benchmarks) and *synt3* benchmarks) or 33 MHz (for benchmarks *synt1* and *synt2*).

Some of the FUs are reused between configurations. Specifically, a total of 36, 37 and 10 FUs were reused for *synt1*, *synt2* and *synt3*, respectively. This does not account for pass-through components, of which 96, 170 and 216 were reused between configurations for the 3 synthetic benchmarks. With respect to the total FPGA resources, the LUT usage for the three synthetic benchmark RPUs is 25 %, 40 % and 49 %; FF usage is 6 %, 12 % and 17 %s.

Benchmarks *synt1* and *synt3* achieve considerable speedups, incurring communication overheads of 17.1 % and 5.7 %. Benchmark *synt2* exhibits a slowdown as expected, since the same occurs for the individual implementations. The overhead for this case is 59.4 %. In the overhead-free scenario, the three synthetic cases show good speedups. These benchmarks show that a good average speedup can be achieved when Megablocks have similar CPLs and significant parallelism.

6 Conclusion

This paper presented a general-purpose computing architecture based on a General Purpose Processor (GPP) and a Reconfigurable Processing Unit (RPU) automatically generated offline from instruction traces. In this architecture, a multiplexer module transparently interfaces with standard memory buses, allowing the RPUs to access the GPP's data memory. We use a two-port data memory to allow parallel memory accesses and to achieve the acceleration of instruction traces with load/store operations. Data memory accesses are easily handled by our RPU through the transparent bus multiplexer, allowing shared access to the entire address space. The RPU can thus operate on any number

of data arrays at any address regardless of their size. This allows an efficient memory access scheme that does not introduce costly data transfers between the data memory and the RPU. Future work will focus on extensions to the RPU execution model to enable both pipelining and multipath Megablock execution, and on developing efficient scheduling memory operations in the RPU, including the handling of RAW and WAR dependencies.

Acknowledgments. 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, and by the European Community's Framework Programme 7 under contract No. 248976.

References

1. Wolf, W.: A decade of hardware/software codesign. Computer **36** (April 2003) 38–43
2. Clark, N., Blome, J., Chu, M., Mahlke, S., Biles, S., Flautner, K.: An architecture framework for transparent instruction set customization in embedded processors. In: Proc. of the 32nd Annual Intl. Symposium on Computer Arch. (ISCA '05), Washington, DC, USA, IEEE Computer Society (May 2005) 272–283
3. Paek, J.K., Choi, K., Lee, J.: Binary acceleration using coarse-grained reconfigurable architecture. SIGARCH Comput. Archit. News **38**(4) (January 2011) 33–39
4. Lysecky, R.L., Vahid, F.: Design and implementation of a microblaze-based warp processor. ACM Trans. Embedded Comput. Syst. **8**(3) (April 2009) 22:1–22:22
5. Noori, H., Mehdipour, F., Murakami, K., Inoue, K., Saheb Zamani, M.: An architecture framework for an adaptive extensible processor. J. Supercomput. **45**(3) (September 2008) 313–340
6. Kim, Y., Lee, J., Shrivastava, A., Paek, Y.: Memory access optimization in compilation for coarse-grained reconfigurable architectures. ACM Trans. Des. Autom. Electron. Syst. **16**(4) (October 2011) 42:1–42:27
7. Beck, A.C.S., Rutzig, M.B., Gaydadjiev, G., Carro, L.: Transparent reconfigurable acceleration for heterogeneous embedded applications. In: Proc. of the Conf. on Design, Automation and Test in Europe (DATE '08), ACM (2008) 1208–1213
8. Bispo, J., Paulino, N., Cardoso, J.M., Ferreira, J.C.: Transparent runtime migration of loop-based traces of processor instructions to reconfigurable processing units. International Journal of Reconfigurable Computing (2012) (in press).
9. Bispo, J., Cardoso, J.M.P.: On identifying and optimizing instruction sequences for dynamic compilation. In: Proc. Intl. Conf. Field-Programmable Technology (FPT'10). (2010) 437–440
10. Seoul National University: SNU Real-Time Benchmarks. http://www.cprover.org/goto-cc/examples/snu.html Accessed 23 Dec 2012.
11. Texas Instruments: TMS320C6000 Image Library (IMGLIB) - SPRC264. http://www.ti.com/tool/sprc264 Accessed 23 Dec 2012.
12. Warren, H.S.: Hacker's Delight. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)