Hindawi Publishing Corporation
International Journal of Reconfigurable Computing
Volume 2013, Article ID 340316, 20 pages
http://dx.doi.org/10.1155/2013/340316

Research Article

Transparent Runtime Migration of Loop-Based Traces of Processor Instructions to Reconfigurable Processing Units

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

[1] *Departmento de Engenharia Informática, Faculdade de Engenharia, Universidade do Porto, Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal*
[2] *INESC TEC, Faculdade de Engenharia, Universidade do Porto, Rua Dr. Roberto Frias s/n, 4200-465 Porto, Portugal*

Correspondence should be addressed to João Canas Ferreira; jcf@fe.up.pt

Received 11 May 2012; Accepted 17 September 2012

Academic Editor: René Cumplido

The ability to map instructions running in a microprocessor to a reconfigurable processing unit (RPU), acting as a coprocessor, enables the runtime acceleration of applications and ensures code and possibly performance portability. In this work, we focus on the mapping of loop-based instruction traces (called Megablocks) to RPUs. The proposed approach considers offline partitioning and mapping stages without ignoring their future runtime applicability. We present a toolchain that automatically extracts specific trace-based loops, called Megablocks, from MicroBlaze instruction traces and generates an RPU for executing those loops. Our hardware infrastructure is able to move loop execution from the microprocessor to the RPU transparently, at runtime, and without changing the executable binaries. The toolchain and the system are fully operational. Three FPGA implementations of the system, differing in the hardware interfaces used, were tested and evaluated with a set of 15 application kernels. Speedups ranging from 1.26× to 3.69× were achieved for the best alternative using a MicroBlaze processor with local memory.

1. Introduction

The performance of an embedded application running on a general-purpose processor (GPP) can be enhanced by moving the computationally intensive parts to specialized hardware units and/or to Reconfigurable Processing Units (RPUs) acting as acceleration coprocessors of the GPP [1, 2]. This is a common practice in embedded systems. However, doing so, manually or automatically, usually implies a hardware/software partitioning step over the input source code [3]. This step is static, requires the source code of the application, and does not promote code and performance portability as the hardware/software components are obtained for a specific target architecture. Dynamic partitioning and mapping of computations (hereafter simply referred as dynamic partitioning) [4–6] is a promising technique able to move computations from an GPP to the coprocessor in a transparent and flexible way, and may become an important contribution for the future reconfigurable embedded computing systems.

In this paper, we present a system which can automatically map loops, detected by running a MicroBlaze executable binary, to an RPU. We focus on a special kind of trace-based loop, named Megablock [7], and transform Megablocks into graph representations which are then used to generate Megablock-tailored RPUs. Megablocks are repeating patterns of elementary units of the trace (e.g., basic blocks) in the instruction stream of the program being executed. The RPU is runtime reconfigurable and can use several configurations during program execution.

In our current implementation, Megablocks are detected offline through cycle-accurate simulation of running applications [8, 9]. The synthesis of the RPU is also done offline, while reconfiguration of the RPU is done online. The migration of the application execution between hardware and software is done online, without changes in the binary code of the application to be executed.

This paper makes the following main contributions:

(i) with respect to our previous work [8, 9], it proposes a more efficient use of an RPU for transparent binary acceleration by using lower-overhead interface schemes between RPU and GPP.

(ii) It presents implementations of three distinct system architectures and their system components to allow transparent migration of sections of GPP execution traces to the RPU, which includes reconfiguration of the RPU and runtime insertion of communication primitives.

(iii) It analyses the runtime overhead of the partitioning and mapping stages (currently performed by offline tools) and it presents a dedicated hardware detector circuit to accelerate the runtime identification of Megablocks bearing in mind a future runtime implementation.

(iv) It includes an extensive experimental evaluation of the proposed approaches with a set of 17 benchmarks (15 kernels and 2 examples of multiple executions of kernels in the same RPU).

The rest of this paper is organized as follows. Section 2 introduces the Megablock, the type of loop considered for mapping to the RPU. Sections 3 and 4 describe the proposed hardware/software system and the RPU architectures used, respectively. Section 5 explains the toolchain of our current approach, and Section 6 presents experimental results obtained for the three prototyped hardware/software implementations using an RPU coupled to a microprocessor. Section 7 presents related work, and Section 8 concludes the paper.

2. Megablocks

The architecture of the RPU was heavily influenced by the kind of repetitive patterns we are mapping, the Megablocks [7]. A Megablock is a pattern of instructions in the execution trace of a program and is extracted from execution instruction traces. Figure 1 shows a portion of the trace of a *count* kernel. In this case, when the kernel enters a loop, the trace repeats the same sequence of six instructions until the loop is finished.

The Megablock concept [7] was proposed in the context of dynamic partitioning, that is, deciding at runtime which instruction sequences executing on an GPP should be moved to dedicated hardware. We consider four steps for dynamic partitioning: detection, translation, identification, and migration. *Detection* determines which sections of the application instruction traces are candidates for dedicated hardware execution; *translation* transforms the detected instruction traces into equivalent hardware representations (i.e., RPU resources and corresponding configurations); *identification* finds, during program execution, the sections that were previously detected; *migration* is the mechanism that shifts the execution between the GPP and the RPU.

```
. . .
0×188   addk    r6, r6, r3
0×174   bsra    r3, r5, r4
0×178   addik   r4, r4, 1
0×17C   andi    r3, r3, 1
0×180   xori    r18, r4, 8
0×184   bneid   r18, −16
0×188   addk    r6, r6, r3
0×174   bsra    r3, r5, r4
0×178   addik   r4, r4, 1
0×17C   andi    r3, r3, 1
0×180   xori    r18, r4, 8
0×184   bneid   r18, −16
0×188   addk    r6, r6, r3
0×174   bsra    r3, r5, r4
. . .
```

FIGURE 1: Example of a repeating pattern of instructions in the trace of a 8-bit *count* kernel.

In a full online approach, all the above steps would be executed online. In the current prototypes, detection and translation are done offline (Section 5), while identification and migration are done online (Section 3.4). This approach has been also used by Faes et al. [16], which manually partitions code at the method level and proposes a framework which can, at runtime, intercept arbitrary method calls and pass control to previously designed hardware modules.

A Megablock represents a single, recurring path of a loop across several basic blocks. For every instruction which can change the control flow (e.g., branches), the Megablock considers a new exit point which can end the loop if the path of the Megablock is not followed. Since we are considering only a single path, the control-flow of a Megablock is very simple and we do not need to use decompilation techniques which extract higher-level constructions such as loops and *if* structures. And unlike other instruction blocks (e.g., Superblock and Hyperblock [17]), a Megablock specifically represents a loop.

For Megablocks to be useful, they must represent a significant part of the execution of a program. Previous work [7] shows that for many benchmarks, Megablocks can have coverage similar or greater than other runtime detection methods, such as monitoring short backward branches (the approach used by Warp [10]).

Megablocks are found by detecting a pattern in the instruction addresses being accessed. For instance, Figure 1 shows a pattern of size 6 (0x174, 0x178, 0x17C, 0x180, 0x184, and 0x188). In [7], it is shown how the detection of Megablocks can be done in an efficient way.

In the mapping approach described in this paper, each Megablock is first transformed into a graph representation. Because of the repetitive nature of the Megablock, we can select any of the addresses in the Megablock to be the start address. However, the start address can influence optimizations which use only a single pass. The start address is also used in our system architecture as the identifier of the Megablock during the identification step and must define the start of the Megablock unambiguously. We use the following heuristic to choose the start address: choose the lowest address of the Megablock which appears only once.

For the example in Figure 1, the start address according to this heuristic is 0x174. Since two or more Megablocks can start at the same memory address, but the current identification procedure only supports one Megablock for each start address, we synthesize the Megablock which has the highest coverage as determined in the detection phase.

2.1. Detecting Megablocks. There are several parameters we need to take into account when detecting Megablocks. For instance, the unit of the pattern can be coarser than a single instruction (e.g., a basic block). We impose an upper limit on the size of the patterns that can be detected (e.g., patterns can have at most 32 units). We define a threshold for the minimum number of instructions executed by the Megablock (i.e., only consider Megablocks which execute at least a given number of instructions). We can detect only inner loops, or decide to unroll them, creating larger Megablocks.

The values chosen for these parameters are dependent on the size and kind of Megablocks we want to detect.

2.2. Hardware for Megablock Detection. The problem of detecting a Megablock is similar to an instance of the problem of detecting repeated substrings, for example, *xx*, with *x* being a substring containing one or more elements. This is also known as *squares*, or tandem repeats [18]. In our case, substring *x* is equivalent to the previous sequence of instructions and represents a single iteration of a loop. Although we want to find patterns with many repetitions (a square strictly represents only two repetitions), we observed that if a sequence of instructions forms a square, it is likely that more *x* elements will follow (e.g., *xxxx* ...). The detection method considers that two repetitions are enough to signal the detection of a Megablock.

Figure 2 presents a hardware solution for Megablock detection when using basic blocks as the detection unit. It has three main modules: the *Basic Block Detector* reads the instructions executed by the processor and detects which instructions correspond to the beginning of basic blocks. It outputs the instruction addresses corresponding to the beginning of basic blocks (*BB_address*), and a flag which indicates if the current instruction is the beginning of the basic block (*is_BB_address*).

The *Megablock Detector* receives pattern elements, which in this case are the first addresses of basic blocks. It outputs the size of the current pattern, or zero if no pattern is detected (*pattern_size*), and a control signal indicating the current state of the detector (*pattern_state*).

The module *Trace Buffer* is a memory that, when Megablock detection is active (i.e., the module is currently looking for Megablocks), stores the last instructions executed by the processor, their corresponding addresses, and a flag which indicates if the instruction corresponds to a pattern element of the Megablock (e.g., the start of a basic block). After a Megablock is detected, the *Trace Buffer* stops storing executed instructions and can be used to retrieve the detected Megablock.

Figure 3 presents the general diagram for the *Megablock Detector*. The *Squares Detector* finds patterns of squares. It receives pattern elements and detects squares of size one up to a maximum and outputs one flag per detected square size (*pattern_of_size_X*).

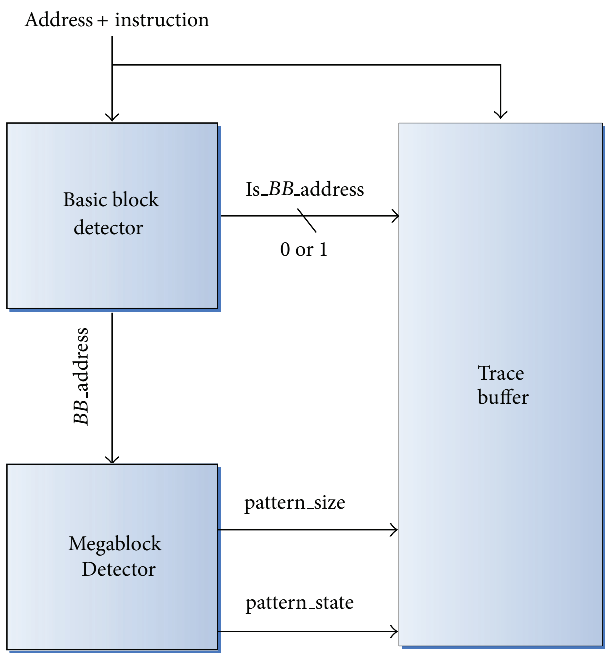


FIGURE 2: Hardware solution for Megablock detection.

A pattern element can trigger one or more square sizes. The module *Pattern Size Arbiter & Encoder* receives the individual *pattern_of_size_X* flags, chooses which pattern size should be given priority, and encodes the chosen size into a binary string. For instance, when detecting only inner loops, this module can be implemented as a priority encoder. The module *Pattern State* is a state machine which indicates the current state of the pattern, and can have one of five values: *Pattern_Started*, *Pattern_Stopped*, *Pattern_Changed_Sizes*, *Pattern_Unchanged*, and *No_Pattern*.

Figure 4 presents the block diagram for a hardware implementation of the *Squares Detector*. It shows the first three modules, which correspond to detectors for sizes 1 up to 3. The additional modules follow the same structure. The *pattern_element* signal corresponds to a basic block start address. Each detector for a specific square size (with exception of the detector for size one) uses an FIFO. When FIFO*s* have a reset signal they are usually implemented in hardware using Flip-Flops (FFs), becoming relatively expensive. However, if it is not necessary to access the intermediate values of *FIFOs*, they can be implemented with considerably less resources (e.g., if an FPGA has primitives for shift registers available). When using such *FIFOs*, the reduction factor in resources can be as high as 16× and 32× [19] (e.g., when using the primitives SRL16 and SLR32 in Xilinx FPGAs, resp.).

3. Target System Architectures

We consider three prototype implementations of the target system: DDR-PLB (Arch. 1, illustrated in Figure 5), LMB-PLB (Arch. 2, presented in Figure 6), and LMB-FSL (Arch.

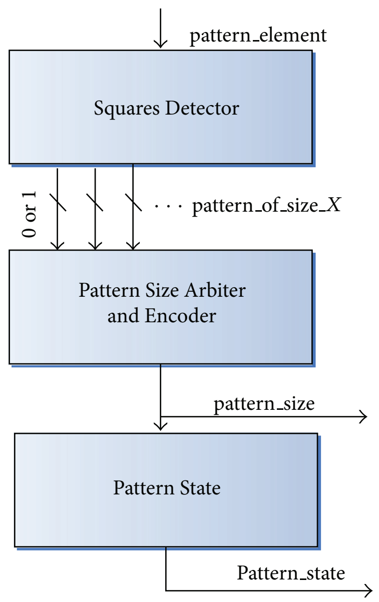


FIGURE 3: Diagram for the Megablock Detector.

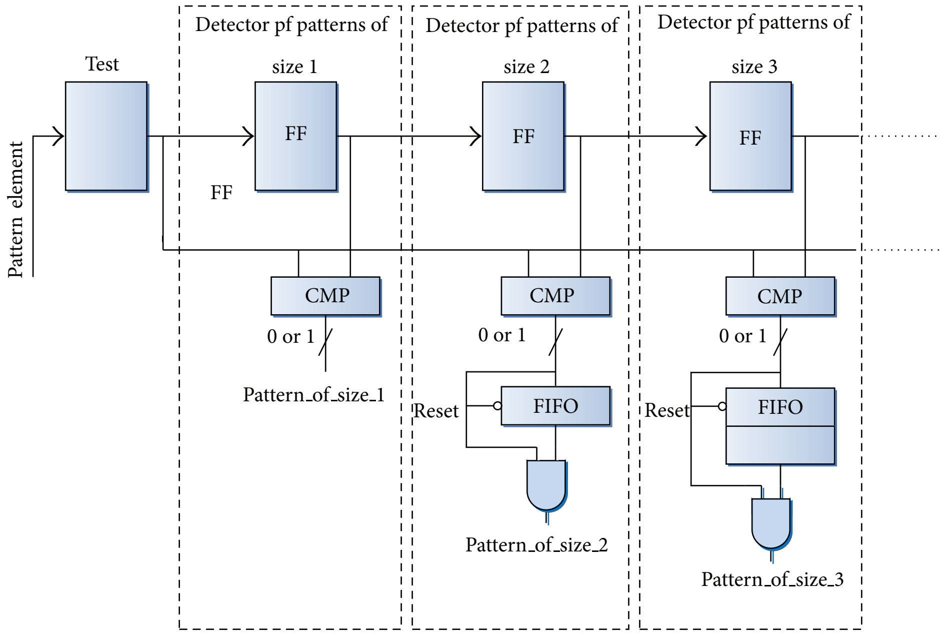


FIGURE 4: Diagram for a hardware implementation of the Squares Detector showing modules for pattern sizes from 1 to 3.

3, illustrated in Figure 7). All three implementations consist of an GPP executing a target application, an RPU used to accelerate execution of Megablocks, and additional hardware (the Injector) to support online identification and migration of Megablocks. The three system architectures share similar hardware modules, the main difference being their interfaces and arrangements. The prototypes were designed for an FPGA environment: instead of proposing a single all-purpose RPU, we developed a toolchain which generates the HDL (hardware description language) files of an RPU tailored for a set of Megablocks detected in the application to be run on the system. This step is done offline and automatically, as detailed in Section 5.

All versions use the same RPU architecture. This module is reconfigured in runtime to execute any of the supported Megablocks. To identify Megablock start addresses an auxiliary system module, named Injector (e.g., Processor Local Bus (PLB) Injector in Figure 5), is placed on the instruction bus between the GPP and its instruction memory. The Injector monitors the program execution and determines when to switch to/from execution on the RPU, by comparing the current Program Counter (PC) to a table containing the

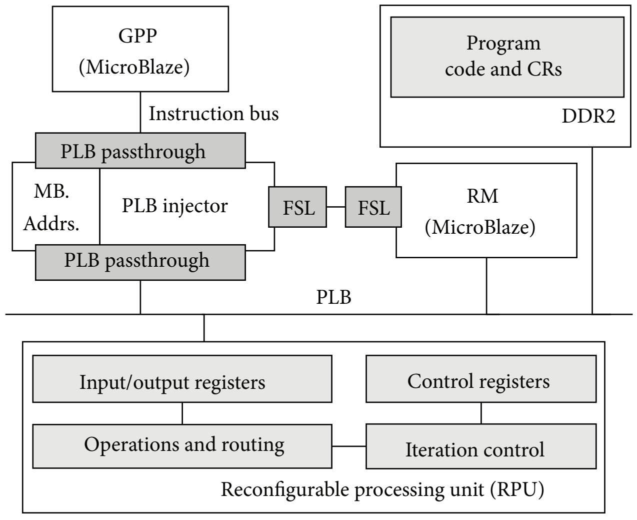


Figure 5: DDR-PLB architecture (Arch. 1).

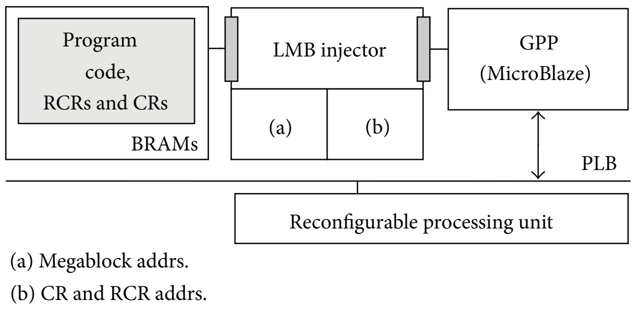


Figure 6: LMB-PLB architecture (Arch. 2).

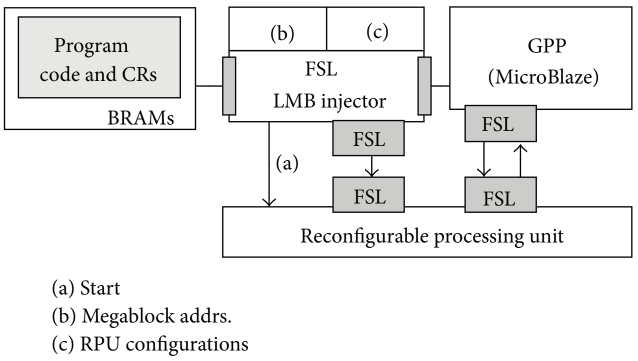


Figure 7: LMB-FSL architecture (Arch. 3).

start addresses of the Megablocks previously detected from the instruction traces. Once a Megablock is identified, the Injector executes the migration step.

The Injector is capable of controlling the execution of the GPP, by modifying the contents of the instruction bus and injecting arbitrary instructions. Thus, the Injector changes the behavior of the program running on the GPP in a transparent way, avoiding modifications to the GPP hardware and to the program binary stored in the memory.

All three architectures use a similar Injector module, adapted to the memory and RPU interfaces. Specific details on the different interfaces and behavior of the system are given in the following sections.

3.1. Architecture 1: External Memory and Processor Bus.

Figure 5 shows the first system architecture, the DDR-PLB variant (Arch.1). The program code is located in external DDR memory and the interface between the GPP and the RPU is done via the Processor Local Bus (PLB).

Arch.1 consists of the GPP, a loosely coupled RPU, the PLB version of the Injector module, and an auxiliary reconfiguration module (RM), currently implemented as a second MicroBlaze. In addition to the program to be executed on the main GPP, the external memory also contains automatically generated communication routines (CRs), which are later explained in detail in Section 5.

The system operates as follows: during boot, the GPP copies the program code from flash (not shown in Figure 5) to the DDR, while the RM copies the instructions of the CRs, which are initially within its own local memories (not shown in Figure 5), to predefined DDR positions, so they can be later accessed by the GPP. During execution, the Injector monitors the PC of the GPP and stalls the system (by placing a branch to PC + 0 on its instruction bus) if the current PC matches any entry in an internal table of Megablock start addresses. Then, the Injector indicates to the RM, via its point-to-point Fast Simplex Link (FSL) [20] connection, which Megablock was identified. The RM reconfigures the RPU for the detected Megablock (this step is skipped if the RPU is already configured for that Megablock). When done, the RM responds to the Injector with a memory address, which is the start address of an CR stored in DDR memory, specific for the detected Megablock. The Injector then inserts a branch operation to that address in the GPP's instruction bus.

From this point on, the Injector and the RM no longer interfere, and the GPP executes the CR, which contains microprocessor instructions to load operands to the RPU from its register file, to start computation, to wait for completion (by polling an RPU status register), and to retrieve results, as well as a branch back to original program code. Once the GPP returns to program code, the Megablock code executed on the RPU is skipped in software, and the execution continues normally. If the PC of the GPP reaches another address known to the Injector, the process is repeated and the RPU is reconfigured if necessary.

Currently, the RM is used to reconfigure the RPU and to return the address of the corresponding CR. In the case of a fully online approach, the RM can be used to perform Megablock detection and generation of new configurations for the RPU, that is, translation, at runtime. Section 6.4 presents execution times for the partitioning tools running on an ARM processor, considering mapping to a general-purpose RPU architecture [15].

In our current mixed offline/online approach, the RM and the loosely coupled interfaces are superfluous, and the remaining two architectures were developed with this in mind. The DDR-PLB case is still being used to analyze the viability for a later expansion for fully online operation.

3.2. Architecture 2-Local Memory and Peripheral Bus.

Using external memory introduces a large communication overhead, due to the access latency, both when accessing the original code and the CRs which have to be executed in order to use the RPU. The alternative system (Arch. 2), shown in Figure 6, reduces this latency by having the program code in local memories. In this case, the PLB Injector is replaced by the Local Memory Bus (LMB) Injector. The LMB is used by the GPP to access local, low-latency memories (Block RAMs—BRAMs). The use of these memories reduces execution time for both the CRs and the regions of program not mapped to the RPU, thus reducing global execution time. In this architecture, the RPU structure and interface are the same as the ones presented in the previous section. Also, as the GPP to RPU interface is the same, the CRs do not change.

In this approach, we removed the RM and moved its functionality to the Injector. The Injector now includes a table that maps Megablock addresses to CR start addresses (where one Megablock start address corresponds to one target CR address). The method to reconfigure the RPU was also changed: in addition to CRs there are also reconfiguration routines (RCRs) which load immediate 32-bit values to the RPUs configuration registers (explained later). RCRs are also placed in the same local memories and the LMB Injector keeps an additional table with their start addresses. Thus, when a Megablock address is identified, the Injector causes the GPP to branch to either an RCR or an CR, based on whether or not the RPU is already configured for the detected Megablock. If an RCR is executed, its last instruction is a branch to its corresponding CR. For the GPP, there is no distinction between the two situations (the only difference is the overhead introduced in the communication with the RPU).

3.3. Architecture 3: Local Memory and Point-to-Point.

Despite the reduction in overhead due to the use of local memories, the PLB access latency still introduces a significant overhead when sending/retrieving operands/results to/from the RPU. In the architecture shown in Figure 7 (Arch. 3), both the Injector and the GPP communicate with the RPU via FSL. In the previous scenarios, RCRs and CRs contain MicroBlaze load/store instructions that place configuration values or inputs on the RPU through the PLB. In this case, these instructions (whose latency was measured to be as high as 9 to 12 bus clock cycles in some cases, depending on bus arbitration) are replaced by one-cycle *put/get* instructions [21] per value sent/received.

In this case, the RPU reconfiguration is handled by the Injector itself. Configurations are held in a dedicated memory for the Injector (not shown in Figure 7), whose contents are defined at synthesis time. When a Megablock is identified, the Injector performs two tasks: it causes the GPP to branch to the corresponding CR and sends configuration data to the RPU via an FSL connection. While this last task is being done, the GPP is sending the operands to the RPU. The CRs mostly consist of *get* and *put* instructions. The GPP executes *put* instructions to load operands and then, since the *get* instructions are performed blocking, the GPP automatically stalls until data are placed on the RPU's output FSL (i.e., until the computation is finished).

The RPU only starts computing when it receives a start signal from the Injector, which indicates that all configurations have been written, and that the GPP is now stalled by the blocking *get*. The latter situation is detected by the Injector, which in this setup monitors both the instruction *opcode* and the instruction address.

3.4. Injector Architecture.

Figure 8 shows the architecture of the PLB Injector, which is responsible for interfacing the GPP with the rest of the system in Arch.1 (Figure 5), as well as for starting the reconfiguration process by identifying Megablocks. The PLB, LMB, and FSL variants of the LMB Injector vary slightly in structure due to the buses they interface with. The LMB version does not implement communication with other modules. With respect to the DDR version, both PLB-LMB and FSL-LMB versions (Figure 7) require different control logic due to different latencies at which instructions can be fetched by the GPP (1 clock cycle for the LMB bus and as many as 23 clock cycles for the external DDR memory, as measured using ChipScope Analyzer [22], a runtime bus monitor).

The Injector monitors the instruction address bus of the GPP. It reacts to instruction addresses which correspond to the start of Megablocks and acts as a passthrough for all other instructions. If the PC matches a table of addresses, the Injector stalls the GPP and execution is switched from software to hardware, as already explained. This means the system can be enabled or disabled through a single modification of the instruction bus. The current PLB Injector

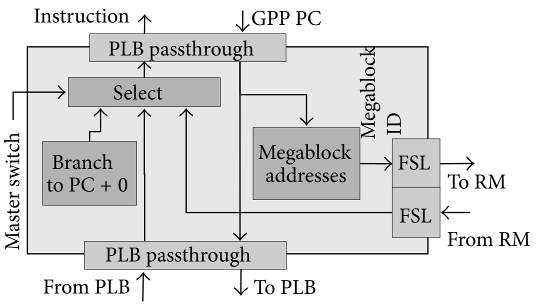


Figure 8: PLB injector architecture.

for the DDR-PLB system does not allow the use of cache. When using external memories with cache, the MicroBlaze uses a dedicated cache bus to the external memory, and an enhanced PLB Injector would be required to interface with that bus. This will be addressed in our future work.

Although the Injectors alter the instruction stream in order to modify runtime behavior, they cannot do so indiscriminately. The Injector can only interfere in order to keep the GPP from executing code that can now be executed on the RPU, but the possibility of false positives exists, due to the instruction fetch behavior of processor pipelines: the processor is performing a *fetch* during the *execute* stage of a previous instruction. If a Megablock start address comes after a branch instruction and the branch is taken, then the GPP will still fetch the instruction from the Megablock's start address, even though it will not be executed. To solve this problem, the Injector inserts a single stall (branch to PC + 0) after a Megablock address is identified, replacing the first instruction of the Megablock. If the next requested address is different from the address following the Megablock start address, it means that the initial branch was taken. In this case, the inserted branch is discarded by the GPP, and execution continues normally; otherwise the initial branch was not taken and the GPP would actually enter the corresponding region of code. The inserted branch will cause the Megablock address to repeat, and on this second occurrence, the Injector can safely cause the GPP to branch to the CR. This verification introduces a short delay equivalent to eight processor instructions. For the PLB case, this corresponds to approximately 184 clock cycles (considering an average external memory access latency of 23 clock cycles, through the PLB). For the LMB case, 8 clock cycles are required since BRAM latency is 1 clock cycle.

Another issue is caused by the fact that some instruction sequences must execute atomically in the MicroBlaze ISA. For instance, the IMM instruction loads a special register with a 16 bit immediate value. The following instruction combines these upper 16 bits with its own lower 16 bits to generate a 32 bit immediate operand. An instruction must not be injected after an IMMinstruction. As expected, the identified Megablocks do not start after IMM instructions.

4. RPU Architecture

The RPU is generated offline and is based on a parameterized HDL description. Modifying the parameterization at synthesis time produces a single RPU which can execute a particular set of Megablocks, with the required layout and number of functional units (FUs). The RPU specification is generated automatically by our toolchain, based on the Megablocks found in the detection step. Interfaces and control logic remain the same regardless of the supported set of Megablocks. According to a particular online configuration, the RPU performs calculations equivalent to one of the Megablocks it was tailored for.

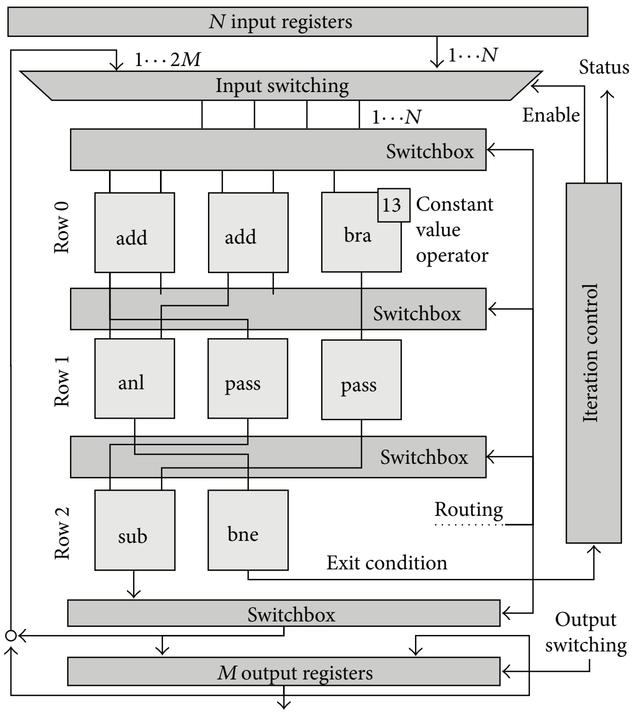


Figure 9: Array of FUs.

4.1. FU Array. Figure 9 illustrates a possible array of FUs for an RPU. The array is organized in rows with variable number of single-operation FUs. The number and width of rows are variable according to parameterization. The first row receives inputs from the input registers, which are written to by the GPP by executing a CR. The values placed in these registers originate from the GPP's register file. Likewise, values read from the output registers of the RPU are placed in the GPP's register file.

Some Megablocks produce the same result onto two or more GPP registers. Instead of having a duplicated output register, we handle this by adding equivalent value assignment instructions in CRs. Other Megablocks always produce one or more constants values onto the register file. In this case, since the Megablock detection tool optimizes the Megablock graph representation and removes such operations, we also add these constant value attributions to registers as CR instructions.

Each row contains operations that have no data dependencies and can execute in parallel, and results are then propagated to following rows. Each row of FUs is registered and data propagates at one row per clock cycle. If an operation has a constant input, the RPU generation process tailors the FU to that input (e.g., *bra* FU in Figure 9). The current implementation supports arithmetic and logic operations with integers, as well as comparison operations and operations producing or receiving carry. Carry can be retrieved by the GPP in both PLB and FSL scenarios. FUs on the array may have different number of inputs/outputs amongst themselves. The carry from additions is an example of a second output of an FU. Another is the upper 32-bit result of a multiplication. Any output of an FU can be used as an input of FUs in the following row. One or more inputs of an FU may be constant, synthesis-time-specified values.

Crossbar-type connections (switchboxes in Figure 9) are used between adjacent rows to perform this operand routing, and are runtime reconfigurable. The switchboxes automatically adapt to the width of the associated row during synthesis and can direct any of their inputs to any number of outputs. Connections spanning more than one row are established by passthrough FUs (pass FUs in Figure 9).

The RPU architecture was specifically designed to run loops with one path and multiple exits, and does not need logic for multiple paths (e.g., predicated hardware). The number of iterations of the loop does not need to be known before execution: the RPU keeps track of the possible exits (e.g., *bne* FU in Figure 9) of the loop and signals when an exit occurs (via a status register). If the number of iterations is constant in software, this is built into the array as a constant value operator. In order to terminate execution, the RPU always has at least one exit condition.

Only atomic Megablock iterations are supported. That is, either a Megablock iteration completes or is discarded. Support for nonatomic iterations would require discriminating which exit condition triggered, recovering the correct set of outputs and returning to a particular software address. Thus, when an exit occurs, the current iteration is discarded, and execution resumes in the GPP at the beginning of the Megablock. This means that the last iteration will be performed in software, allowing the GPP to follow whichever branch operation triggered an exit, maintaining the integrity of the control flow. In the current version of the RPU, all operations complete within one clock cycle and each iteration takes as many clock cycles as the number of rows (depth) of the RPU.

In the first iteration, the array is input with values from the input registers. After the first iteration is completed, control logic enables the first switchbox, and results from the previous iteration(s) are used to compute the subsequent one(s). This means that, although rows are registered, the execution is not pipelined. The RPU can keep the output values of the previous iteration and these values can be routed back to the output registers so as to change positions in order to mimic the software behavior of register assignments present in some loops. This feature is used when a Megablock includes code that places into a GPP register the value of another GPP register in a previous iteration. Along with the values produced in the current iteration, these previous values can also be routed back into the first row to be reused. Thus, the input switching *mux* takes $N + 2M$ values and produces N values. Since some input values are constant for all iterations of a call of the RPU, each of the N outputs either maintains its initial value found in the input register, or they are assigned one of the $2M$ values produced. This is followed by another switchbox that can route any of these N values to any number of FU inputs in the first row (the sequence of 2 multiplexers was kept for simplicity of implementation).

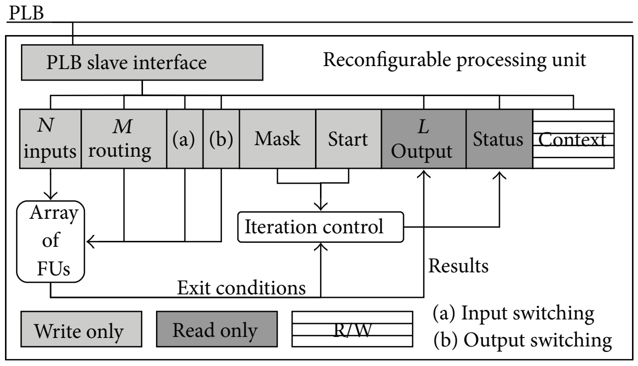


Figure 10: RPU architecture overview, with PLB interface.

4.2. RPU Interface. RPU configuration is performed by writing to configuration registers. These registers control the routing of the operands through the switchboxes and indicate which exit conditions should be active. Figure 10 presents the main components of the RPU, detailing the PLB interface. The PLB interface RPU uses the bus interface to feed operands and retrieve results through 32-bit, memory-mapped registers. The FSL interface is composed simply of three FSL ports, two being inputs (for configuration and operands) and one output (for results). Apart from these interface level differences, the array of FUs and other internal aspects of the RPU remain the same in all three architectures.

Depending on the specification of the RPU, the number of input (N), output (L), and configuration (M) registers may vary. The remaining registers are implementation independent. Values written at runtime to the *routing*, *input/output switching*, and *mask* registers are generated offline.

Input and *output* registers contain operands/results of computations, and there are as many input/output registers as the maximum number of operands/results found in the set of Megablocks implemented by the RPU. The *routing*, *input switching*, and *output switching* registers configure the switchbox connections between rows, the routing of results back to the first row (i.e., feedback) and the routing of output register values to different positions of said output register bank (i.e., managing results from the previous iteration). Both the *input* and *output switching* are handled by a single 32-bit register. This was done for simplicity of design and limits the number of input and output registers of the RPU. For instance, if there are 4 output registers, a total of 9 values (current and previous results plus one initial value from the input register) can be chosen. This requires 4 bits to represent this selection range, which limits the number of input registers to 8 in order to use a maximum of 32 bits.

The number of routing registers is a function of the width and depth of the RPU. In our implementation, the largest number of outputs between all rows determines the minimum number of bits used to perform a selection and a single register may hold routing information for more than one row. For instance, in Figure 9 (where $N = 2$ and $M = 1$), the maximum width is 5 (row 1). Three bits are required to represent this range, and since the total number of inputs is 18 (input to FUs and number of output registers), a total of 54 routing bits are required (2 registers). The synthesis time parameterization selects and wires the proper number and groups of bits from the registers into the switchboxes of the array.

The *mask* register determines which exit conditions can trigger the end of computation. If more than one Megablock is mapped to the RPU, different exit conditions may exist on the array. Since data propagates through all the FUs during execution, even though FUs that are not part of the Megablock the RPU is configured for, an exit condition may trigger incorrectly unless disabled. The *mask* register performs a bit masking of the exit conditions. This limits the number of exits allowed on the RPU to 32. However, no observed combination of Megablocks in our benchmarks exceeded this value. In the FSL case, these configuration registers have the same function, but they have to be written to in a specific sequence.

The *start* register signals the RPU to initiate computations, and is written to by the GPP as part of the CRs, after all operands have been loaded. In the FSL scenario, the start signal is directly sent by the Injector. The *status* register contains information on the current status of the RPU. A *busy* bit on this register is the value the GPP polls for completion. A *first fail* bit indicates if the execution terminated during the first iteration. This is a special case in which no results need to be recovered from the RPU. The two *context* registers are used as scratchpad memories during the execution of Reconfiguration Routines (RCRs) and Communication Routines (CRs). During execution of these routines, one or two GPP registers must be used as auxiliary registers to load/store values. In the case of a *first fail* the used GPP registers recover their original values from the *context* registers.

The iteration control enables the propagation of data to output registers after a number of clock cycles equal to the depth of the RPU, monitors the exit conditions according to the applied mask, and sets the *status* register.

5. Toolchain for Offline Megablock Detection and Translation

5.1. Tool Flow Overview. We developed a tool suite to detect Megablocks and generate an RPU and its configuration bits. The tool flow is summarized in Figure 11. We feed the executable file (i.e., ELF file) to the Megablock Extractor tool [8] which detects the Megablocks. This tool uses a cycle-accurate MicroBlaze simulator to monitor execution traces. Although this step is performed offline, it is not a typical static code analysis.

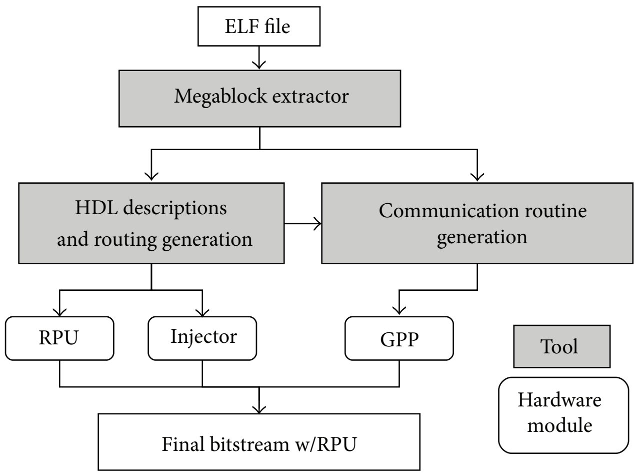
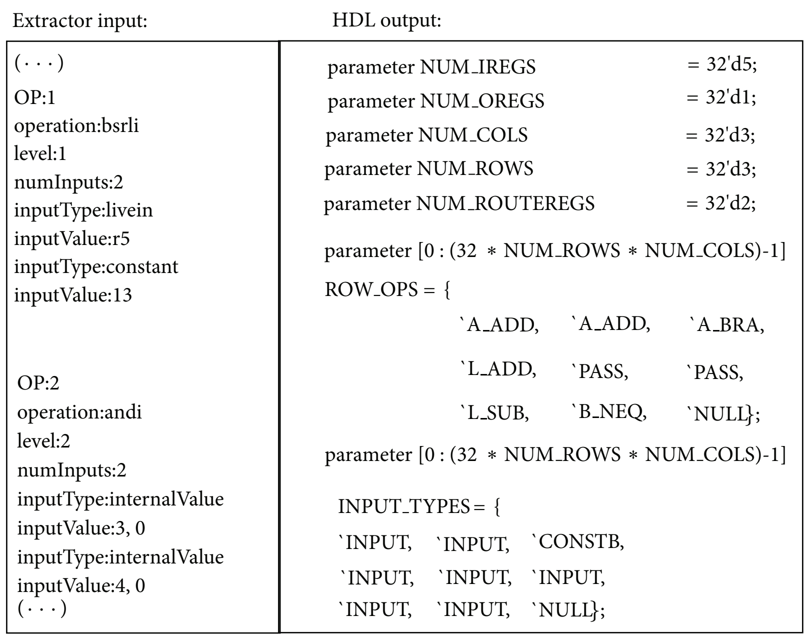


FIGURE 11: Tool flow.

For translation, Megablocks are processed by two tools: one generates the HDL (Verilog) descriptions for the RPU and the Injector, and the other generates the CRs for the GPP. The HDL description generation tool parses Megablock information, determines FU sharing across Megablock graph representations, assigns FUs to rows, adds passthrough units, and generate a file containing the placement of FUs. FUs are shared between different Megablocks, since at any given time there is only one Megablock executing in the RPU. The tool also generates routing information to be used at runtime (configuration of the interrow switches), as well as the data required for Megablock identification.

5.2. Generating the RPU Description. The RPU description generation tool produces an HDL header file that specifies the number of input/output and routing registers, the number of rows and columns of the RPU, the placement of FUs, constant value operators of the FUs, if any, and other auxiliary parameters that enable the use of the Verilog based generate constructs that instantiate the RPU. Inputs to this tool are Extractor outputs regarding the sequence of operations in the Megablock, their scheduling on the equivalent graph representation and connections between them. Figure 12 shows an excerpt of a generated HDL header that fully characterizes the RPU, along with the input Megablock information. The parameter array *ROW_OPS* specifies the layout of the RPU. The *INPUT_TYPES* array configures the *A_BRA* (arithmetic barrel shift right) to have its second input as a constant value, while all other accept 2 variable inputs.

In order to generate a combined RPU description for several Megablocks, the tool maintains information between calls. Each call treats a single Megablock. The Extractor transforms the MicroBlaze ISA into an abstract instruction set. Each operation in the Megablock is then mapped to a single FU. Different instructions can be mapped to the same FU type. For instance, a distinction between an *add* and *addi* exists only in the context of the MicroBlaze ISA. This decoupling means the toolchain and RPU could easily be expanded to any other processor ISA. Supporting new


Extractor input:

```
(· · · )
OP:1
operation:bsrli
level:1
numInputs:2
inputType:livein
inputValue:r5
inputType:constant
inputValue:13


OP:2
operation:andi
level:2
numInputs:2
inputType:internalValue
inputValue:3, 0
inputType:internalValue
inputValue:4, 0
(· · · )
```

HDL output:

```
parameter NUM_IREGS              = 32'd5;
parameter NUM_OREGS              = 32'd1;
parameter NUM_COLS               = 32'd3;
parameter NUM_ROWS               = 32'd3;
parameter NUM_ROUTEREGS          = 32'd2;

parameter [0 : (32 * NUM_ROWS * NUM_COLS)-1]
ROW_OPS = {
              `A_ADD,   `A_ADD,   `A_BRA,
              `L_ADD,   `PASS,    `PASS,
              `L_SUB,   `B_NEQ,   `NULL};
parameter [0 : (32 * NUM_ROWS * NUM_COLS)-1]

  INPUT_TYPES = {
  `INPUT,   `INPUT,   `CONSTB,
  `INPUT,   `INPUT,   `INPUT,
  `INPUT,   `INPUT,   `NULL};
```

Figure 12: RPU HDL header excerpt.

types of FUs would be equally straightforward, as each is an individual hardware module.

After this mapping, FU placement is performed. Since connections between rows are crossbar-like, horizontal placement is unrestricted, and rows are filled from left to right. During this placement phase, the arrays current status is checked to reuse already mapped FUs, if possible, to reduce resources (this also reduces the number of required routing bits). Only operations, from two distinct Megablocks, that occur on the same row and map to the same type of FU may reuse the same FU between them. After placement of operation FUs, passthroughs are placed. Rows are checked bottom to top so as to propagate passthroughs upwards. If an FU requires as an input, an output originating from an FU that spans more than one row, a passthrough is inserted. If two FUs require the same operand that originates several rows above, the same chain of passthroughs is used. This repeats until all connection spans equal 1. Due to the nature of the Megablock graphs, passes tend to be created in an inverted pyramid fashion. As this behavior repeats from Megablock to Megablock, passthroughs are heavily reused between them.

Values for routing and configuration registers are then generated and saved to files. As explained, routing information is concatenated across all routing registers, and selection values depend on the width of the RPU rows. As a consequence, already generated routing information must be regenerated if the depth and/or width of the RPU vary. If the depth increases, passthroughs need to be inserted for Megablocks of smaller depth. This implies that Megablocks of a smaller depth will suffer a delay (in clock cycles per iteration) equal to the difference between their depth and the maximum depth of the RPU.

5.3. Generating the Communication Routines. Megablock Extractor outputs also detail which GPP registers contain RPU inputs and which are destinations of RPU outputs. An additional file provided by the RPU generation tool (after translating the Megablock) includes the routing and configuration register values and associations between each GPP register in the Megablock and an RPU register. The Communication Routine generation tool either generates a PLB CR or an FSL CR, along with a HDL header containing the Megablock addresses. This file also contains the addresses of communication routines if generated for Arch. 2 (RCRs and CRs) or Arch. 3 (CRs).

For all three architectures, the routines are executed by the GPP, and they are located in the GPP's program memory, along with the program itself. The generated RCRs and CRs are placed in *C*-code structures (arrays), which are then compiled together with the application. This does not imply altering application code. These instructions are merely linked to tool predefined memory positions (defined in the linker script), which are known by the Injector.

Figure 13 shows the PLB and FSL CRs for the *reverse* benchmark. Not only is the FSL CR shorter, the instruction latencies are also smaller than those of the PLB case. The tool attempts to optimize CRs by using relative load/stores and immediate value assignments to registers (which occur in the RCRs). Relative instructions may shorten the length of the CR, depending on the number of values to send/receive. If such instructions are used, the RPUs scratchpad registers are used to store the original values of the GPP registers used by the instructions at the start of the routine.

For the PLB, CR operands are loaded (one is automatically saved into one *context register* when writing to the first input register), a start signal is written and the RPU is polled

```
PLB CR
Load live-ins:
    0×1d40: imm -15136
    0×1d44: swi r5, r0, 0
    0×1d48: imm -15136
    0×1d4c: swi r4, r0, 8
    0×1d50: imm -15136
    0×1d54: swi r6, r0, 4
Send start signal:
    0×1d58: addi r5, r0, −1
    0×1d5c: imm -15136
    0×1d60: swi r5, r0, 36
Wait for fabric:
    0×1d64: imm -15136
    0×1d68: lwi r5, r0, 64
    0×1d6c:  andi r5, r5, 4
    0×1d70: bnei r5, −12
Check for exit status:
    0×1d74: imm -15136
    0×1d78: lwi r5, r0, 64
    0×1d7c: andi r5, r5, 32
    0×1d80: beqi r5, 16
Return if First fail true:
    0×1d84: imm -15136
    0×1d88: lwi r5, r0, 68
    0×1d8c: brki r0, 440
Restore live-outs:
Set address offset:
    0×1d90: imm -15136
    0×1d94: addi  r6, r0, 0
    0×1d98: lwi  r18, r6, 40
    0×1d9c: lwi  r3, r6, 52
    0×1da0: lwi  r4, r6, 56
    0×1da4: lwi  r5, r6, 60
Recovering carry:
    0×1da8: imm -15136
    0×1dac: lwi  r6, r0, 44
    0×1db0: bnei r6, 12
    0×1db4: msrclr r6, 4
    0×1db8: bri 8
    0×1dbc: msrset  r6, 4
Recovering last live-out:
    0×1dc0: imm -15136
    0×1dc4: lwi r6, r0, 48
Return Jump:
    0×1dc8: brki r0, 440
```

```
FSL CR
Putting live-ins:
    0×1e00: nput r4, rfsl0
    0×1e04: nput r6, rfsl0
    0×1e08: nput r5, rfsl0
Getting control:
    0×1e0c: get r5, rfsl0
    0×1e10: beqi r5, 12
    0×1e14: get r5, rfsl0
    0×1e18: brki r0, 440
Getting live-outs:
    0×1e1c: get r18, rfsl0
Getting carry:
    0×1e20: msrclr r6, 4
    0×1e24: get r6, rfsl0
    0×1e28: beqi r6, 8
    0×1e2c: msrset r6, 4
Remaining live-outs:
    0×1e30: get r6, rfsl0
    0×1e34: get r3, rfsl0
    0×1e38: get r4, rfsl0
    0×1e3c: get r5, rfsl0
Return Jump:
    0×1e40: brki r0, 440
```

Figure 13: Comparison between PLB- and FSL-based CRs for the *reverse* benchmark.

for a done signal; once done, the status register is checked for the *first fail* bit. If set, values are recovered from the context registers, and execution immediately returns to software. If not, results are retrieved (in this case using relative loads), including the set/clear of the GPP carry bit according to the respective RPU result and execution returns to software. For the FSL CR, each operand is sent with a nonblocking *put* instruction. The *get* instructions are blocking until the output FSL contains data; the first output sent by the RPU is the status register. If a *first fail* occurs, the GPP reads another value. This value restores the content of the GPP register used to perform the *first fail* check. If this situation does not occur, results are recovered, including carry, and a branch back to software is taken.

For the benchmarks used in this paper, PLB-based CRs consist of 32 instructions on average. Arch. 2 (see Figure 6) also uses RCRs to reconfigure de RPU. Their average length is 28 instructions. For the case of Arch. 2, a maximum average length of 61 instructions for communication may occur if reconfiguration of the RPU is required at every call. For the FSL case, the average number of instructions is only 16, with reconfiguration occurring in parallel, if required.

5.4. Example. Figure 14 exemplifies the behavior of the system for the reverse benchmark. The outlined addresses (1b8 to 1d0, where 1b8 is the start address) constitute the Megablock, which iterates 32 times. This totals 255 clock cycles to execute this kernel. Considering the latency of each instruction (labeled on the right) and the number of instructions, the IPC (number of instructions per clock cycle) of this kernel is 0.89. When the Injector triggers at address 1b8, the execution of these instructions is replaced with the steps outlined in Figure 14. The first part of the CR (sending operands) takes place. Then execution proceeds on the RPU, and since the array depth is 3 and the number of iterations is 32, a total of 107 cycles are required. The RPU executes eight instructions in 3 clock cycles and achieves an IPC of 2.67. The last iteration must be performed in software for 7 cycles (the bneid branch is not taken, reducing its latency to 1). Results are then retrieved during 15 cycles. The resulting reduction in cycles provides an execution speedup of 1.97.

6. Experimental Results

The proposed architectures and tools were tested and evaluated with 15 code kernels. All kernels work on 32-bit values. Each individual benchmark calls the corresponding kernel N times. The reported results were obtained for $N = 500$.

Two additional tests (merge1 and merge2) group together six kernels in order to evaluate the case where an RPU is

```
Original program:                     Megablock execution in RPU + CR:
⟨reverse⟩:               cycles         ⟨reverse⟩:
1b0: addk r6, r0 , r0                   1b0: addk r6, r0, r0
1b4: addk r4, r6, r0                    1b4: addk r4, r6, r0
1b8: andi r3, r5, 1         1           1b8: Replaced by:
1bc: or r3, r3, r6          1           Injector delay:                = 8
1c0: addik r4, r4, 1        1           + 3 CR cycles                  = 11
1c4: addk r6, r3, r3        1           + 3 cycles per iteration:
1c8: xori r18, r4, 32       1           11 +3 * 32                     = 107
1cc: bneid r18, − 20        2           + 15 CR cycles                 = 122
1d0: sra r5, r5             1           + last iteration in software
1d4: rtsd r15, 8                        122 + (8 − 1)                  = 129
1d8: addk r3, r6, r0                    1d4: rtsd 15, 8
                                        1d8: addk r3, r6, r0
Total cycles per iteration: 8
Total cycles: 8 * 32 − 1 = 255          delta = 255 − 129 = 126 → Speedup
```

FIGURE 14: Cycle reduction for the *reverse* kernel, for a LMB-FSL system.

generated from several Megablocks. The RPUs generated for these cases have six possible configurations. The *merge1* benchmark contains *count, even_ones, fibonacci, hamming, popcount32,* and *reverse*. Benchmark *merge2* includes *compress1, divlu, expand, gcd2, isqrt2,* and *maxstr*. For these cases, we evaluate the scenario where the calls to each kernel are alternated (for $N = 500$, the total number of RPU configuration changes during kernel execution is equal to 500 × 6 = 3,000). This is the worst-case scenario, which requires RPU reconfiguration between each kernel execution. We also consider an additional scenario where each kernel is called N times in sequence without intermediate reconfiguration (*merge1/2 n/s*).

The loops of most kernels have a constant number of iterations (16 or 32). Five kernels iterate a variable number of times per call, according to the inputs. The number of iterations of *fibonacci*, for instance, is an arithmetic progression of the input value. In all benchmarks, the current iteration count (between 0 and $N - 1$) is used as an input.

6.1. Setup. We used the Megablock Extractor tool to do an offline detection of the Megablocks from execution traces. For the detection we disabled inner loop unrolling (except in the *popcount3* case, where we map the Megablock of an unrolled inner loop), used basic blocks as the elementary pattern unit, set the maximum pattern size to 32, and rejected any Megablock which executed less than 100 instructions. For each kernel (except for *merge1/2*), we implemented only one Megablock. The majority of the computation was spent in the selected Megablock, the average coverage being 91.59% of the executed instructions.

Each kernel was compiled with *mb-gcc* 4.1.2 using the −O2 flag and additional flags which enable specific units of the MicroBlaze processor (e.g., -mxl-barrel-shift for barrel shifter instructions). The MicroBlaze version used was v8.00a.

The prototype was implemented on a Digilent Atlys board with a Xilinx Spartan-6 LX45 FPGA and DDR2 memory. We used Xilinx EDK 12.3 for system synthesis and bitstream generation. All benchmarks run at 66 MHz except *merge1/2* and *usqrt*, which run at 33 MHz. In most cases, the RPU achieved higher operating clock frequencies than the 66 MHz used for the MicroBlaze processor. Since we use the same clock signal for all the modules of the system, including the RPU, speedups can be computed from measurements given in number of clock cycles, and are therefore, independent of the actual system frequency. To count clock cycles, we used a timer peripheral attached to the PLB.

6.2. Megablocks and RPUs. Table 1 summarizes the characteristics of the Megablocks used in the evaluation. The average number of instructions per call of the Megablock is a product of the number of instructions per iteration and the average number of iterations. Table 1 includes values for maximum instruction level parallelism (ILP), percentage of instructions covered by the Megablocks versus the total executed instructions, and instructions per cycle achieved in software (SW IPC). IPC was computed considering the number of clock cycles required to complete one iteration over the number of instructions. As all Megablocks include branch operations, which have a 2 cycle latency, the SW IPC is always below 1 instruction per clock cycle. The SW IPC values shown assume a latency of 1 clock cycle for instruction fetch, which is not the case for the DDR-PLB architecture, but is valid for LMB-based architectures. Since all implemented operations in the tested benchmarks have one clock cycle of latency, the critical path length (CPL) has a value equal to the depth of the RPU (see Table 2, RPU characteristics). For the *merge1/2* cases, the values presented are the averages of the values of the individual kernels implemented in each case.

Table 2 summarizes the characteristics of the RPU for each kernel. The *#OP. FUs* column presents the number of FUs used as operations (i.e., not passthroughs). Due to the interconnection scheme used, passthroughs often outnumber operation FUs. However, the resulting RPUs were relatively small. Due to FU sharing, the *merge1/2* cases use about 35.44% and 50.43% of the total number of FUs for the individual Megablocks. This is equivalent to 51 and 58 reused FUs between kernels (including operations and passthroughs). Since passthroughs occur frequently in all kernels, they are reused more often; *merge1* reuses passthrough FUs 41 times and *merge2* 104 times. This is equivalent to reducing a total

Table 1: Detected Megablock characteristics.

Kernels	Megablock characteristics			
	Avg. Inst. Executed p/call	Max. ILP	Coverage (%)	SW IPC
count	192	2	94.9	0.857
even_ones	192	3	94.0	0.857
fibonacci	1,497	2	99.4	0.857
ham_dist	192	3	94.0	0.857
pop_cnt32	256	3	97.2	0.889
reverse	224	3	95.6	0.875
compress	138	3	89.7	0.889
divlu	155	2	90.5	0.833
expand	138	3	89.7	0.889
gcd	330	2	98.8	0.889
isqrt	96	3	84.0	0.857
maxstr	120	2	88.1	0.800
popcount3	15500	3	85.4	0.912
mpegcrc	465	4	87.6	0.934
usqrt	288	6	84.9	0.947
merge1	444	2.7	N/A	0.865
merge2	166	2.5	N/A	0.860

Table 2: RPU characteristics.

Kernels	RPU characteristics				
	#OP. FUs	# Pass. FUs	Max. row depth	Depth	HW IPC
count	6	6	5	3	2.00
even_ones	5	4	7	3	1.67
fibonacci	4	6	4	3	1.33
ham_dist	6	11	6	3	2.00
pop_cnt32	8	7	8	3	2.67
reverse	7	9	7	3	2.33
compress	8	21	8	4	2.00
divlu	5	4	5	3	1.67
expand	8	21	8	4	2.00
gcd	8	17	8	6	1.33
isqrt	6	9	6	3	2.00
maxstr	4	6	4	3	1.33
popcount3	18	33	18	9	2.00
mpegcrc	14	32	14	7	2.00
usqrt	17	42	17	8	2.13
merge1	16	12	16	3	1.65
merge2	24	35	24	6	1.07

of 43 passthroughs to 12, for *merge1*, and reducing a total of 78 pass–throughs to 35, for *merge2*.

The maximum ILP achieved by each RPU is the same as the maximum ILP shown for each Megablock in Table 1. The average ILP is 2.93 (*merge1/2* excluded) and the highest value occurs for *usqrt* (6 instructions in parallel). The IPC achieved

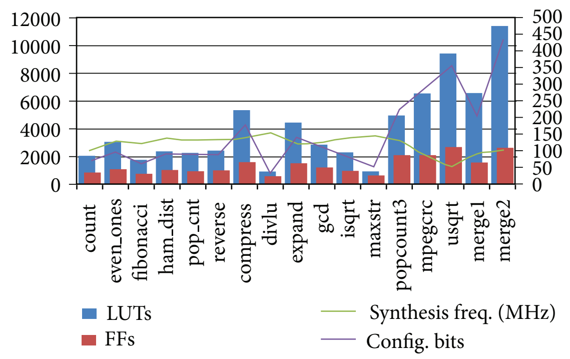


Figure 15: FPGA resources, synthesis frequency, and required configuration bits for each RPU with a PLB interface (LUTs and FFs shown on the left axis).

by an RPU depends on the total number of operations it performs per iteration and its depth. Each RPU contains, at most, as many operations as the Megablock it implements. Due to graph-level optimizations such as register assignment simplification and constant propagation, the actual number of operations can be lower (e.g., *popcount3*, requires only 18 operations to implement its original 31 assembly instructions). *IMM* instructions [23] are an example of instructions that do not need an additional FU. If ILP is high and depth of the RPU is low, this results in a higher IPC. Ignoring any overheads, speedups are obtained when the RPU IPC is larger than software IPC.

Figure 15 shows the implementation characteristics of the individual RPUs for the PLB interface case (it is very similar to the FSL case). The reported synthesis maximum clock frequencies of the RPUs ranged from 52 MHz to 154 MHz. Except for the minimum case (which occurs for *usqrt*), all RPU frequencies are higher than the clock frequency of the MicroBlaze. The largest RPU uses 34.57% (9,433) of the LUTs and 4.91% (2,680) of the FFs. The average usage for these resources is 12.62% and 2.32%, respectively. Due to the reuse of FUs performed by the tools, the RPUs for *merge1/2* require a number of LUTs and FFs that is smaller than the sum of LUTs and FFs of the RPUs for the individual kernels they implement. The *merge1* RPU uses about 47% of the LUTs and 27% of the FFs. For the *merge2* RPU, these values are 68% and 40%. In both cases, the RPU frequency is above the GPP frequency, being 94 MHz and 102.6 MHz, respectively. Since the RPU only reconfigures interconnections and not FUs, the number of configuration bits for each RPU is relatively low, with an average of 133 bits for the RPUs of individual kernels (i.e., excluding *merge1/2*).

6.3. Speedups. Figure 16 presents speedups for all architectures. In the DDR-PLB scenario, the MicroBlaze has a 23-cycle penalty for each instruction (note that this scenario does not use caches), while execution of a single row of the RPU takes 1 clock cycle. So, most of the achieved speedup comes from executing operations on the RPU instead of executing the original instructions in the GPP. However, for each call to the RPU, the GPP executes an CR and since the CRs are in

Table 3: Communication overhead.

Kernels	Communication overhead				
	#Inst. of PLB CR	#Inst. of FSL CR	DDR-PLB (%)	LMB-PLB (%)	LMB-FSL (%)
count	27	12	92.4	56.39	26.22
even_ones	34	18	92.1	62.25	30.49
fibonacci	27	14	63.2	15.58	4.22
ham_dist	35	17	91.8	61.18	29.98
pop_cnt32	35	17	92.1	61.34	30.49
reverse	35	17	92.3	61.34	30.49
compress	35	19	95.0	71.82	39.97
divlu	25	10	92.2	53.79	26.83
expand	35	19	95.0	71.76	40.75
gcd	32	15	77.3	34.51	12.81
isqrt	34	16	96.2	74.77	45.52
maxstr	25	10	92.5	55.25	26.89
popcount3	37	18	46.97	8.16	2.77
mpegcrc	36	20	85.5	47.00	21.69
usqrt	31	18	89.9	59.56	28.93
merge1	56.3	22.5	87.0	58.14	17.07
merge1 (n/s)	32.17	15.8	N/A	41.94	16.13
merge2	57	22.0	89.6	70.32	24.43
merge2 (n/s)	31	14.8	N/A	48.04	21.26

DDR, they also suffer of the DDR access latency. In fact, the DDR access latency is the main contributor to the very high overhead of this scenario (Table 3). The situation is aggravated by the relatively low number of instructions executed per call (Table 1). The overhead includes the detection of the Megablock, configuration of the RPU, and execution of the CR. Since the RM fetches instructions from local memories, a large part of the overhead comes from executing the CRs afterwards. It is noticeable that, for a greater number of iterations, the overhead becomes less significant, as is the case of *fibonacci* and *popcount3.* The speedups measured for the DDR scenario include all overheads and range from 2.25× (*isqrt*) to 43.37× (*popcount3*). Speedups in this no-cache scenario do not show the best case for sequential software execution. However, it demonstrates the architecture concept and is a starting point for future work on cache support.

For the LMB-PLB case, slowdowns still occur frequently since the number of iterations and operations found in many of the kernels is still relatively small, and the possible parallelism is not enough to compensate for the overhead. Since program code is now in local memories, GPP execution is not hindered by high memory latencies due to the lack of cache support. However, access to the RPU still suffers from PLB latency, which in this case results in an average overhead of 52.98%. For *merge1/2*, the overhead introduced by reconfiguration is noticeable in the resulting speedup. In these benchmarks, the kernels are executed alternately: every time the RPU is called, it has to be reconfigured. The speedup of *merge1/2* is lower than that of *merge1/2 n/s* due to reconfiguration overhead. Speedups for *merge1/2* are equal to 57% and 72% of the speedups for *merge1/2 n/s,* respectively. This is equivalent to reconfiguration overheads of 27.8% and 42.9%, respectively.

For the LMB-FSL case, the average overhead for individual kernels is 26.54%. Performing the RPU reconfiguration in parallel with the transfer of input operands reduces the effect of reconfiguration overhead. For *merge1*, reconfiguring the RPU at every call introduces negligible additional overhead when compared to *merge1 n/s*, since the number of operands is close, on average, to the number of reconfiguration values. This is not the case for *merge2,* which requires over twice as much configuration values. This means compact RPUs with many configurations and implementing Megablocks with many inputs can be done by switching between configurations within the operand transfer time, that is, without suffering from additional overhead. The overhead introduced by reconfiguration in this case is near zero for *merge1* and 4.03% for *merge2*.

The effect of different overheads is visible in Figure 16, where the speedup trend across benchmarks is consistent, and where the LMB-FSL case is the one closest to the maximum possible speedups. This maximum was computed assuming a software instruction fetch latency of 1 clock cycle (implying IPC = 1), which does not hold true for the DDR case. Table 3 shows the overhead for each scenario along with the number of instructions in the communication routines (CRs) for each interface type. The average number of cycles for an FSL CR, for these kernels, is 17, and the average number of instructions is 16. Since reconfiguration occurs in parallel in this architecture, if necessary, the time required to start computation depends on the maximum between the number of operands to send and the number of configuration values to send. Considering this, the average number of cycles for a complete communication with the RPU in this architecture can be as high as 23.5, for *merge1*, and 31, for *merge2*.

For PLB-based CRs, the averages are 129.2 cycles and 32.2 instructions. For reconfiguration routines (RCRs), which are used in the LMB-PLB architecture, these averages are 110.9 and 28.73, respectively. In the worst-case scenario for this architecture, in which reconfiguration has to be performed at every call, the sequence of RCR and CR takes an average of 239.47 cycles and totals an average of 60.93 instructions. Since generating an RPU for several kernels will increase the number of configuration registers, the RCRs for each kernel in a combined RPU will differ (as the structure of the RPU differs). For *merge1,* the average number of cycles in an RCR is 138.83 and the average number of instructions is 32.8. For *merge2*, the averages are 223.2 and 52.2, respectively. Since the RPU for *merge2* is larger (higher depth and width), it requires more configuration information and the RCRs increase in size.

The gain, in cycles, of using the RPU, as shown in the example of Figure 14, must exceed these communication cycles, in order for a speedup to still be possible. Speedup is a direct function of the ratio between SW IPC and HW IPC, as shown in (1), and is valid for all three architectures (the fetch latency of the DDR-PLB case is accounted for in the $\mathrm{IPC}_{\mathrm{SW}}$ factor). In (1), N_{rSW} represents the number of assembly instructions per iteration of a Megablock in

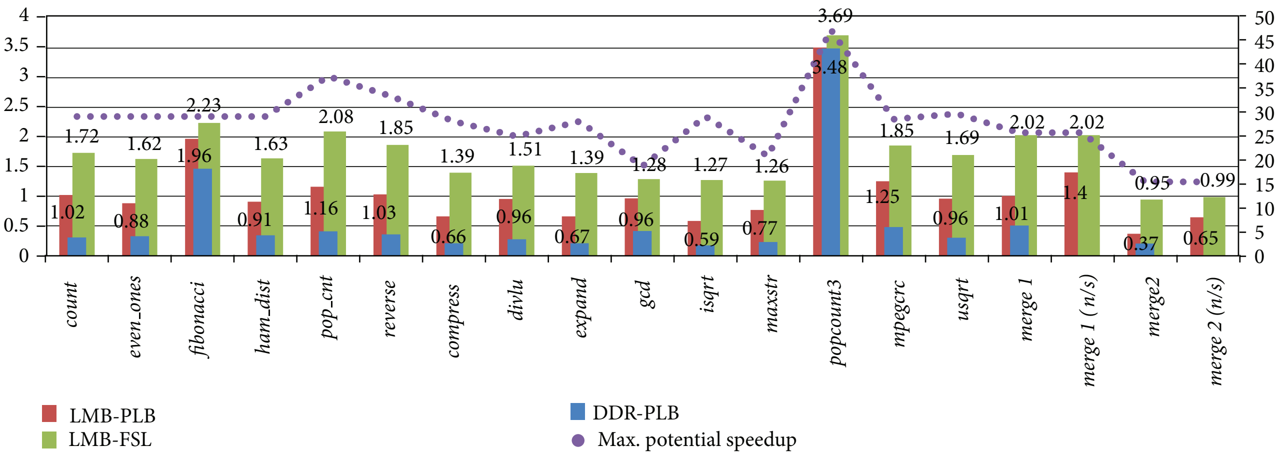


FIGURE 16: Speedups for all three architectures. Results for DDR-PLB architecture (Arch. 1) use the axis on the right. Bar labels show the results for the LMB-PLB (Arch. 2) and LMB-FSL (Arch. 3) architectures (axis on the left). The maximum possible speedups (dotted line relative to the left axis) are estimations calculated using (1), assuming an instruction fetch latency of 1 cycle. A trend can be observed for all three cases. The different overheads dictate the relative scales of the attained speedups.

software, N_{rHW} represents the number of operations per iteration in the RPU (these values are not necessarily the same since some operations can be optimized during translation), OH_{c} represents the number of clock cycles due to overhead (in which the communication routine, injector delay and last iteration cycles are accounted for), and N_{it} is the number of iterations of the Megablock:

$$\text{Speedup} \cong \frac{N_{\mathrm{rSW}}}{N_{\mathrm{rHW}}} \times \frac{\mathrm{IPC}_{\mathrm{SW}}^{-1}}{\mathrm{IPC}_{\mathrm{HW}}^{-1} + \mathrm{OH}_{\mathrm{c}} / \left(N_{\mathrm{it}} \times N_{\mathrm{rHW}}\right)}. \quad (1)$$

There is a 2.0% difference for the LMB-FSL case and 1.5% for the LMB-PLB between the values given by (1) and actual measured speedup values; deviations occur due to additional clock cycles. These correspond to instructions that are executed between the activation and deactivation of the timer and are not part of the Megablock. The speedup estimates have been corrected for these effects. The FSL case is less precise because measurement errors become more significant as the measurements become finer (i.e., smaller number of cycles).

The maximum speedup would be the direct ratio of both IPCs, if there were no overhead cycles. The overhead effect can be reduced when there are many iterations and/or instructions mapped on the RPU. See for example the following equation for the *reverse* kernel in the LMB-FSL architecture:

$$\text{Speedup} \cong \frac{7}{7} \times \frac{1.143}{0.429 + 38 / (32 \times 7)} = 1.91. \quad (2)$$

6.4. Hardware Module for Megablock Detection. We developed a proof-of-concept HDL generator which outputs VHDL for a Megablock Detection hardware module, as depicted in Figure 3 in Section 2. Figure 17 presents the resources needed to implement the module when varying some of the parameters accepted by the generator (maximum pattern size and the bit width of the pattern element).

For the explored parameter ranges, the number of LUTs and FFs increases linearly with the increase of the maximum pattern size. Higher bit widths generally represent a higher number of used resources, although the increase is more significant for FFs than for LUTs. The shape of the LUT resources used is more irregular than the shape of the FFs. We attribute this to the way the synthesis tool maps certain FPGA primitives (e.g., SRLs), used in the HDL code.

For the base case with a maximum pattern size of 24 elements, and considering an address space for instructions of 20 bits, the module needs 455 LUTs and 636 FFs, which represent around 1% of the targeted FPGA (a Xilinx Spartan-6 LX45). These values include the encoder and the state machine for determining the current state of the detector. The decrease of the maximum clock frequency with the increase of the maximum pattern size was expected, as higher values for the maximum pattern size implies more complex logic paths in some parts of the Megablock Detection module (e.g., the comparison between the current pattern element and all the positions in the FIFO).

However, the current implementation working frequencies are sufficient for the considered scenarios. For instance, considering the base case of a maximum pattern size of 24 elements, the maximum estimated clock frequency is between 134 MHz and 147 MHz (depending on the bit width of the elements), which is enough to meet the clock frequency of the MicroBlaze softcore for the targeted FPGA. Higher bit widths generally produce designs with lower clock frequencies, although the impact is relatively small. The maximum impact of the bit width on the clock frequency is on average 14% for the cases studied.

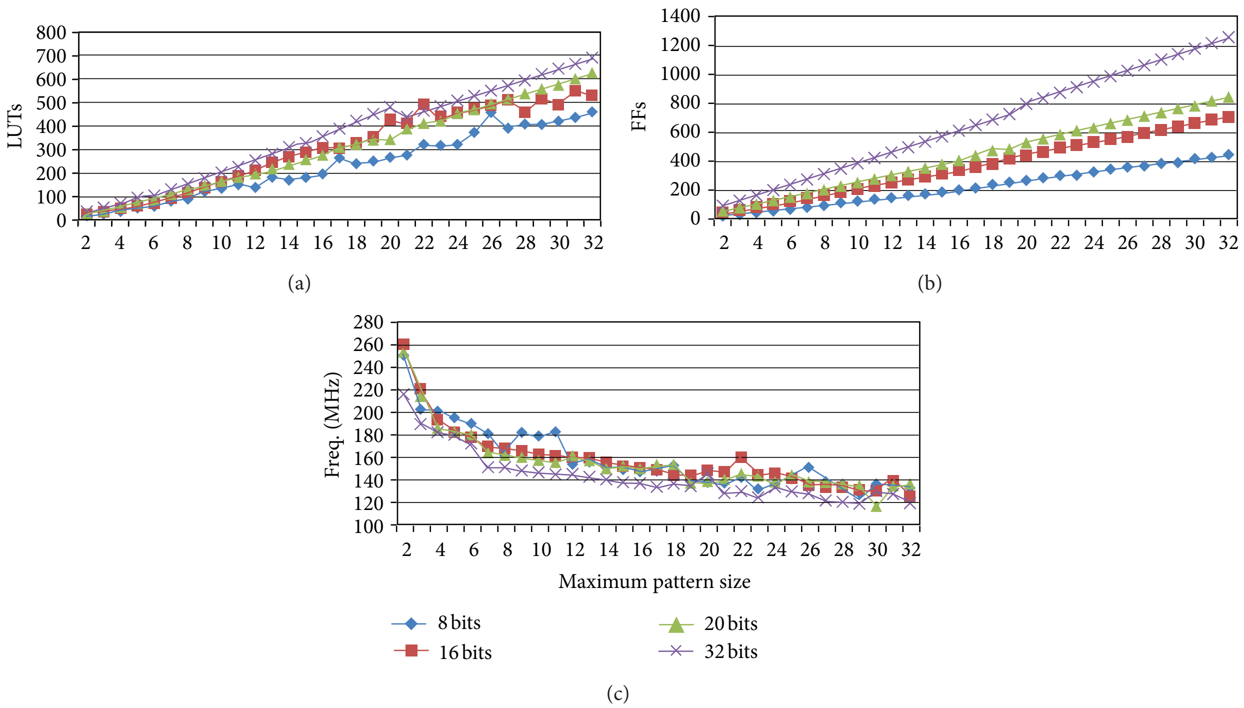


Figure 17: LUTs, FFs, and estimated maximum frequencies for Megablock Detector hardware designs.

Table 4: Execution times for several implementations of the pattern detector for megablocks.

# Addrs	Execution times (ms)			Speedup (HM versus MB/HM versus A8)
	HM@50 MHz	*MB@50 MHz*	*Cortex-A8@1 GHz*	
12	0.0002	2.7	0.6	11,251/2,500
24	0.0005	5.7	1.3	11,963/2,708
48	0.0010	14.0	2.8	14,594/2,917
96	0.0019	30.8	5.9	16,036/3,073
192	0.0038	64.3	12.5	16,757/3,255
384	0.0077	131.5	24.8	17,118/3,229
768	0.0154	265.7	78.7	17,298/5,124

Table 4 contains execution times for three implementations of the pattern detector used to detect Megablocks, executing on different targets. The execution times represent the time each implementation needed to process the given number of addresses (column *#Addresses*). The given addresses are repetitions of the 6 address sequence of the *fir* Megablock. The values in the column *HM@50 MHz* correspond to an implementation of the architecture described in Section 2.2, clocked at 50 MHz. It can process one address every clock cycle. The column *MM@50 MHz* represents a *C* implementation of the equivalent detection functionality running directly on a MicroBlaze processor clocked at 50 MHz. Column *Cortex-A8@1 GHz* corresponds to an implementation in Java, running on a Cortex-A8 clocked at 1 GHz, over the Android 2.2 platform.

Generally, the execution times grow linearly with the input (doubling the size of the input doubles the execution time). There is an exception in the Cortex case, where going from 384 addresses to 768 addresses tripled the execution time, instead of doubling. We think this is due to calls from the system to the garbage collector, during execution of the detector.

When comparing execution speeds, the hardware module for Megablock detection is much faster than the software implementations of the same functionality: around 3,000x faster than the Cortex case and around 16,000x faster than the MicroBlaze case. This difference can be explained by the highly parallel design of the hardware module, and by the software version not being fully optimized for the target platforms.

Table 5 shows average execution times for the several phases needed to perform the translation step, when running their Java implementation on the Cortex-A8, and considering a Megablock of the *fir* loop. The translation step took, on average, about 79 ms to transform the assembly code of the Megablock into a mapping configuration for a general purpose RPU architecture [15]. The most expensive operation is the conversion from assembly code to the graph intermediate representation, which needs 58% of the execution time. The following most expensive operations are Placement and Transform, which take 20% and 12% of the execution time, respectively. The most light-weight steps are Routing and Normalization, each one with 6% and 4% of the total execution time, respectively.

From the values of Table 4, we expect software execution times for the *Translation* phase below 1s (possibly around

Table 5: Average execution times in milliseconds of the translation step.

Normalize	Graph generation	Transform	Mapping		Total
			Placement	Routing	
3.03	46.00	9.71	15.45	4.89	79.09

400 ms) when executed in a MicroBlaze at 50 MHz. Our future work will consider a complete software implementation of the tools in order to achieve a fully runtime mapping system. We will then consider the need to accelerate by hardware the most computationally intensive stages of the mapping process.

7. Related Work

There have been a number of research efforts to map computations to RPUs during runtime. Typically, those efforts focused on schemes to execute in the RPU one or more iterative segments of code, that is, loops, in order to reduce execution time.

These systems can be classified based on the level of coupling between the RPU and the GPP, the granularity of the RPU, the capability to support memory operations, and on the type of approach: online or offline. Although there have been many authors focusing on partitioning and compilation of applications to systems consisting of an GPP and an RPU (see, e.g., [24]), we focus here on the approaches that consider runtime efforts. Related to our work are the approaches proposed by Warp [4, 10], AMBER [12, 13], CCA [5, 11], and DIM [6, 14].

The Warp Processor [4, 10] is a runtime reconfigurable system which uses a custom FPGA as a hardware accelerator for a GPP. The system performs all steps at runtime, from binary decompilation to FPGA placement and routing. The running binary code is decompiled into high-level structures, which are then mapped to a custom FPGA fabric with tools developed by the authors. Warp attains good speedups for benchmarks with bit-level operations and is completely transparent. It relies on backward branches to identify small loops in the program.

AMBER [12, 13] uses a profiler alongside a sequencer. The sequencer compares the current Program Counter (PC) with previously stored PC values. If there is a match, it configures the proposed accelerator to execute computations starting at that PC. The accelerator consists of a reconfigurable functional unit (RFU), composed by several levels of homogeneous functional units (FUs) placed in an inverted pyramid shape, with a rich interconnection scheme between the FUs. The RFU is configured whenever a basic block is executed more times than a certain threshold. Further work considered a heterogeneous RFU [12], and introduced a coarser-grained architecture to reduce the configuration overhead. The AMBER approach is intrusive as the RFU is coupled to the GPP's pipeline stages.

The CCA [5, 11] is composed of a reconfigurable array of FUs in an inverted pyramid shape, coupled to an ARM processor. The work addresses the detection of computations suitable to be mapped to a given CCA, as well as discovering a CCA architecture that best suits a set of detected control-data flow graphs (CDFGs). Initially, the detection was performed during runtime, by using the rePLay framework [25], which identifies large clusters of sequential instructions as atomic frames. The detection was later moved to an offline phase, during compilation [11]. Suitable CCA CDFGs are discovered by trace analysis, and the original binary is modified with custom instructions and rearranged to enable the use of the CCA at runtime.

The DIM reconfigurable system [6, 14] proposes a reconfigurable array of FUs in a multiple-row topology and uses a dynamic binary translation mechanism. The DIM array is composed of uniform columns, each with FUs of the same type. DIM transparently maps single basic blocks from a MIPS processor to the array. DIM also introduced a speculation mechanism which enables the mapping of units composed by up to 3 basic blocks. The system is tightly coupled to the processor, having direct access to the processor's register file.

Table 6 presents the main characteristics of the approaches previously described and of our approach (Megablock column). The main difference between our approach and previous ones is the use of repetitive patterns of machine instructions (Megablocks, in this case) as the partitioning unit [7, 8, 15]. To the best of our knowledge, we have presented the first automated toolchain capable of transparently moving repetitive instruction traces from an GPP to an RPU at runtime without changing the executable binary. Our system is fully operational and all evaluations presented in this paper were actually based on real measurements using an FPGA board. We have shown in greater detail how the hardware system works. Although we previously presented the main concepts, this paper extends them by presenting details for three architectures (two of them implemented for the first time) and an evaluation using a more representative set of benchmarks. Furthermore, three architecture prototypes were implemented and tested on a current commercial FPGA.

8. Conclusion

This paper presented an automated approach to transparently move computations from GPP instruction traces to reconfigurable hardware, without changing the executable binary of the application being run on the GPP. The computations of interest are represented by Megablocks which are patterns of machine instructions that repeat contiguously. Those Megablocks are then mapped to a reconfigurable processing unit implemented with an FPGA.

Using an FPGA board, we evaluated three system architectures that are fully operational. We implemented the detection and translation steps offline to generate RPU descriptions and we introduced an architecture which allows for very fast identification and replacement of Megablocks at runtime. Preparing for a full online approach, we also introduced a hardware module for Megablock detection at runtime.

Table 6: Summary of characteristics for the more relevant approaches.

Characteristics	Approaches				
	Warp [4, 10]	CCA [5, 11]	Amber [12, 13]	DIM [6, 14]	Megablock [7, 15]
Partitioning approach	Detect and decompile inner loops, dynamically translate those loops into configurations for a custom FPGA	Detect segments of instructions which are transformed into subgraphs and executed as macroinstructions on the CCA. Migration by modifying the instruction stream	Detection of hot basic blocks by trace analysis, which are translated to DFGs and mapped to mesh-type RPU configurations	Identify as many instructions as possible, inside one or more basic blocks, to be mapped to DIM	Detect repeating patterns of instructions in the execution trace and migrate those loops to an RPU
Coupling	Loose RPU/GPP coupling, shared instruction and data memory	Tight RPU coupling to the GPP pipeline	Tight RPU coupling to the GPP pipeline	Tight RPU coupling to the GPP pipeline	Loose RPU/GPP coupling through bus or dedicated connections
Granularity	Fine-grained RPU (LUTs, MAC)	Coarse-grained RPU (ALUs)	Coarse-grained RPU (ALUs)	Coarse-grained RPU (ALUs)	Coarse-grained RPU (ALUs)
Size of the segment of code to be mapped in a configuration	Inner loops with up to tens of lines of code	From a couple to a dozen of instructions across basic blocks	Up to 1 basic block	(1) A couple to a dozen of instructions inside a basic block or (2) across up to three basic blocks with speculation	Inner and outer loops with up to hundreds of lines of code
Benchmarks	NetBench, MediaBench, EEMBC, Powerstone, and in-house tool ROCM	MediaBench, SPECint, and encryption algorithms	MiBench suite	MiBench suite	Texas DSPLIB and IMGLIB
Target domain	General Embedded systems	General Embedded and General-Purpose Systems	General Embedded and General-Purpose Systems	General Embedded and General-Purpose Systems	General Embedded Systems
GPP	(1) ARM7 at 100 MHz (2) MicroBlaze at 85 MHz	(1) 4-issue superscalar ARM (2) In-order 5-stage pipelined ARM (ARM-926EJ)	4-issue in-order MIPS-based RISC	Minimips softcore based on the MIPS R3000	MicroBlaze
Size of the RPU	14.22 mm^2 with 180 nm library (~852,000 gates)	0.61 mm^2 with 130 nm library	n.a.	>1 million gates	n.a.
Average speedup	(1) 6.3x (2) 5.9x	(1) 1.2x (2) 2.3x	1.25x	(1) 2.0x (2) 2.5x	2.0x
Average energy reduction	(1) 66% (2) 24%–55%	n.a.	n.a.	.7x	n.a.

Our current system is runtime reconfigurable, both in terms of the resources of the RPU and in terms of the insertion of communication and synchronization primitives. The hardware infrastructure for migration is easily adaptable to other GPPs. For the small benchmark kernels used in the evaluation, the speedups are very dependent on communication latencies. In the most favorable scenario for GPP performance (program code in local memory), the present approach achieved speedups in the range from 1.26× to 3.69×. Furthermore, we have shown that the runtime detection and translation of Megablocks on FPGA-based embedded systems is feasible when assisted by a dedicated hardware detector. However, to consider a fully online partition, mapping, and synthesis approach, one needs to consider the migration to specific hardware of the most execution time demanding tasks. Our future work will be focused on providing full support for the dynamic identification and mapping of Megablocks.

Although the results presented in this paper are encouraging, further work is required to process larger kernels, and in particular kernels which contain memory accesses. Future work will also address the support for caches.

Acknowledgments

This work was partially funded by the European Regional Development Fund through the COMPETE Programme (Operational Programme for Competitiveness) and by National Funds (ERDF) through the Fundação para a Ciência e a Tecnologia (FCT) (Portuguese Foundation for Science and Technology) within (project FCOMP-01-0124-FEDER-022701). J. Bispo and N. Paulino acknowledge the support of FCT through Grants SFRH/BD/36735/2007 and SFRH/BD/80225/2011, respectively.

References

[1] J. Henkel, "Low power hardware/software partitioning approach for core-based embedded systems," in *Proceedings of the 36th Annual Design Automation Conference (DAC '99)*, pp. 122–127, June 1999.

[2] L. Jóźwiak, N. Nedjah, and M. Figueroa, "Modern development methods and tools for embedded reconfigurable systems: a survey," *Integration, the VLSI Journal*, vol. 43, no. 1, pp. 1–33, 2010.

[3] T. Wiangtong, P. Y. K. Cheung, and W. Luk, "Hardware/software codesign," *IEEE Signal Processing Magazine*, vol. 22, no. 3, pp. 14–22, 2005.

[4] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," *Transactions on Embedded Computing Systems*, vol. 8, no. 3, article 22, 2009.

[5] 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 *Proceedings of the 32nd Interntional Symposium on Computer Architecture (ISCA '05)*, pp. 272–283, June 2005.

[6] A. C. S. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Transparent reconfigurable acceleration for heterogeneous embedded applications," in *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*, pp. 1208–1213, Munich, Germany, March 2008.

[7] J. Bispo and J. M. P. Cardoso, "On identifying and optimizing instruction sequences for dynamic compilation," in *Proceedings of the International Conference on Field-Programmable Technology (FPT '10)*, pp. 437–440, Beijing, China, December 2010.

[8] J. Bispo, N. Paulino, J. M. P. Cardoso, and J. C. Ferreira, "From instruction traces to specialized reconfigurable arrays," in *Proceedings of the International Conference on ReConFigurable Computing and FPGAs (ReConFig '11)*, pp. 386–391, Cancun, Mexico, 2011.

[9] J. Bispo, N. Paulino, J. C. Ferreira, and J. M. P. Cardoso, "Transparent trace-based binary acceleration for reconfigurable HW/SW systems," *IEEE Transactions on Industrial Informatics*. In Press.

[10] R. Lysecky, G. Stitt, and F. Vahid, "Warp processors," *ACM Transactions on Design Automation of Electronic Systems*, vol. 11, no. 3, pp. 659–681, 2006.

[11] N. Clark, M. Kudlur, H. Park, S. Mahlke, and K. Flautner, "Application-specific processing on a general-purpose core via transparent instruction set customization," in *Proceedings of the 37th International Symposium on Microarchitecture (MICRO '04)*, pp. 30–40, Portland, Ore, USA, December 2004.

[12] A. Mehdizadeh, B. Ghavami, M. S. Zamani, H. Pedram, and F. Mehdipour, "An efficient heterogeneous reconfigurable functional unit for an adaptive dynamic extensible processor," in *Proceedings of the IFIP International Conference on Very Large Scale Integration (VLSI-SoC '07)*, pp. 151–156, October 2007.

[13] H. Noori, F. Mehdipour, K. Murakami, K. Inoue, and M. S. Zamani, "An architecture framework for an adaptive extensible processor," *Journal of Supercomputing*, vol. 45, no. 3, pp. 313–340, 2008.

[14] A. C. Beck, M. B. Rutzig, G. Gaydadjiev, and L. Carro, "Run-time adaptable architectures for heterogeneous behavior embedded systems," in *Proceedings of the 4th International Workshop Reconfigurable Computing: Architectures, Tools and Applications*, pp. 111–124, 2008.

[15] J. Bispo, *Mapping runtime-detected loops from microprocessors to reconfigurable processing units [Ph.D. thesis]*, Instituto Superior Técnico, 2012.

[16] P. Faes, P. Bertels, J. Van Campenhout, and D. Stroobandt, "Using method interception for hardware/software co-development," *Design Automation for Embedded Systems*, vol. 13, no. 4, pp. 223–243, 2009.

[17] S. A. Mahlke, D. C. Lin, W. Y. Chen, R. E. Hank, and R. A. Bringmann, "Effective compiler support for predicated execution using the hyperblock," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 45–54, IEEE Computer Society Press, December 1992.

[18] J. V. Leeuwen, *Handbook of Theoretical Computer Science: Algorithms and Complexity*, MIT Press, 1990.

[19] J. Bispo and J. M. P. Cardoso, "Synthesis of regular expressions for FPGAs," *International Journal of Electronics*, vol. 95, no. 7, pp. 685–704, 2008.

[20] H. P. Rosinger, "Connecting Customized IP to the MicroBlaze Soft Processor Using the Fast Simplex Link (FSL) Channel," XAPP529 (v1. 3), Xilinx2004.

[21] I. Xilinx, "Microblaze processor reference guide v13. 4," reference manual, 2011.

[22] I. Xilinx, "ChipScope pro 11. 1 software and cores user guide (v11. 1)," 2009.

[23] I. Xilinx, "Microblaze software reference guide v2. 2," reference manual, 2002.

[24] Y. Kim, J. Lee, A. Shrivastava, and Y. Paek, "Memory access optimization in compilation for coarse-grained reconfigurable architectures," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 16, p. 42, 2011.

[25] S. J. Patel and S. S. Lumetta, "rePLay: a hardware framework for dynamic optimization," *IEEE Transactions on Computers*, vol. 50, no. 6, pp. 590–608, 2001.