From Instruction Traces to Specialized Reconfigurable Arrays

João Bispo
Dep. de Eng. Inf.
IST/UTL
Lisboa, Portugal
joaobispo@gmail.com

Nuno Paulino
Dep. de Eng. Elec. e de Comp.
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
ee06211@fe.up.pt

João M. P. Cardoso
Dep. de Eng. Inf.
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
jmpc@acm.org

João Canas Ferreira
INESC Porto
Dep. de Eng. Elec. e de Comp.
Faculdade de Engenharia
Universidade do Porto
Rua Dr. Roberto Frias, s/n
4200-465 Porto, Portugal
jcf@fe.up.pt

Abstract— **This paper presents an offline tool-chain which automatically extracts loops (Megablocks) from MicroBlaze instruction traces and creates a tailored Reconfigurable Processing Unit (RPU) for those loops. The system moves loops from the CPU to the RPU transparently, at runtime, and without changing the executable binaries. The system was implemented in an FPGA and for the tested kernels measured speedups ranged between 3.9× and 18.2× for a MicroBlaze CPU without cache. We estimate speedups from 1.03× to 2.01×, when comparing to the bestestimated performance achieved with a single MicroBlaze.**

Keywords- Reconfigurable Computing; Graph Mapping; Binary Translation; Megablock; Hardware Accelerator; Instruction Trace

I. INTRODUCTION

The performance of an application running on a Central Processing Unit (CPU) can be enhanced by moving the computationally intensive parts to specialized hardware. This is a common practice in embedded systems. However, doing so, either manually or automatically, usually implies a hardware/software partitioning step from the input source code. Support for generating dedicated hardware from execution information obtained from running applications is a step towards transparent software acceleration.

In this paper we present a system which can automatically map loops of a MicroBlaze executable binary to a Reconfigurable Processing Unit (RPU). We extract a special kind of trace-based loop, which we call Megablock [1], and transform Megablocks into graphs which are used to create a tailored RPU. We detect the Megablocks by identifying a repeating pattern of elementary units of the trace (e.g., Basic Blocks) in the instruction stream of the program. The RPU is runtime reconfigurable and can use several configurations during a single program execution. In our current implementation we do Megablock extraction and loop identification offline, through cycle-accurate simulation of running applications. The RPU description and generation is done offline, while reconfiguration is done online, without changes in the executable binary.

This paper is organized as follows. Section II presents previous related work. Section III describes the proposed architecture and Section IV introduces the type of loop we are mapping, the Megablock. Section V explains some of the design decisions made for the current implementation. Section VI presents experimental results obtained for a prototype implementation of the RPU, and Section VII concludes de paper.

II. RELATED WORK

There have been a number of research efforts to map computations to hardware during runtime. More related to our work are the approaches proposed by Warp [2, 3], AMBER [4, 5], CCA [6, 7], and DIM [8, 9].

The Warp Processor [2, 3] is a runtime reconfigurable system which uses an FPGA as a hardware accelerator for a General Purpose Processor. The system performs all steps at runtime, from binary decompilation to FPGA placement and routing. The running binary 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 running program.

AMBER [4, 5] uses a profiler alongside a sequencer. The sequencer compares the current PC with previously stored PC values. If there is a match, it configures the proposed accelerator to perform the Data Flow Graph (DFG) starting at that PC. The accelerator consists of a Reconfigurable Functional Unit (RFU), composed by several levels of homogeneous 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 produced a heterogeneous RFU [4], and introduced a coarser grained architecture to reduce the configuration overhead. This approach is intrusive as it is coupled to the processor's pipeline.

The CCA [6, 7] is composed of a reconfigurable array of FUs in an inverted pyramid shape, coupled to an ARM processor. This work addresses Control-Data Flow Graph (CDFG) detection and mapping. Initially, the detection was performed during runtime, by using the rePLay framework [10] which identifies large clusters of sequential instructions as atomic frames. Graph detection was later moved to an offline phase during compilation [6]. Graphs are discovered by trace analysis and the original binary is modified with custom instructions and rearranged to facilitate the use of the CCA at runtime.

978-0-7695-4551-6/11 $26.00 © 2011 IEEE
DOI 10.1109/ReConFig.2011.43

The DIM Reconfigurable System [8, 9] proposes a reconfigurable array of FUs in a mesh-like topology used by a dynamic binary translation mechanism. The 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. Optimization is transparent to the software tool chain. The DIM array is composed of uniform columns, each with FUs of the same type.

This paper presents the first automated tool-chain capable of transparently moving repetitive instruction traces (named Megablocks) from a CPU to a reconfigurable coprocessor at runtime without changing the executable binary. Furthermore, a prototype of the architecture was implemented and tested on current commercial FPGAs.

III. HARDWARE ARCHITECTURE

Figure 1 shows the general architecture of our current embedded system prototype, which consists of a CPU and a loosely coupled RPU connected to the CPU bus. To avoid modifications to the CPU, we use an Injector module which monitors the current PC and triggers the use of the RPU. Program code is in external memory. The Reconfiguration Module (RM) is responsible for the RPU configuration. The prototype was designed for an FPGA environment: instead of proposing a single all-purpose RPU, we developed a tool chain which generates the HDL description of an RPU tailored for the application to be run on the system. This step is done offline and automatically, as detailed in Section V.

A. RPU Architecture

Figure 2 presents the main components of the RPU, and Figure 3 illustrates a possible array of FUs of an RPU. The RPU uses a peripheral bus interface unit to feed operands and retrieve results through memory mapped registers. The array of FUs contains all the blocks necessary to execute the previously detected Megablocks. The array of FUs is organized in rows with variable number of single-operation FUs. If an operation has a constant input, the RPU generation process tailors the FU to that input (e.g., *bra* FU in Figure 3). The current implementation supports arithmetic and logic operations with integers, including carry operations. Crossbar connections are used between adjacent rows, and are runtime reconfigurable, allowing the use of multiple Megablocks during the execution of a program. Connections spanning more than one row are established by pass-through FUs (*pass* FUs in Figure 3).

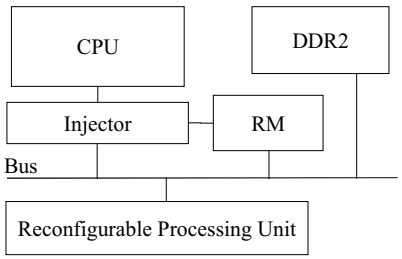

Figure 1. System Architecture

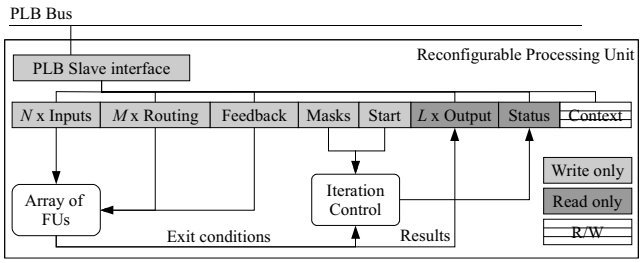

Figure 2. RPU Architecture

RPU configuration is performed by writing to configuration registers. These registers control the routing of the operands through the RPU and indicate which exit conditions should be active.

The RPU was specifically designed to run loops with one path and multiple-exits, such as Megablocks (Section IV), 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 3) of the loop and signals when an exit occurs (via a status register). When this happens the current iteration is discarded, and execution resumes in the CPU at the beginning of the iteration. 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.

B. Support Architecture

Figure 4 shows the architecture of the PLB Injector, which is responsible for interfacing the CPU with the rest of the system, as well as for starting the reconfiguration process. In this case we assume as CPU bus the PLB bus.

Each RPU configuration is associated to a single instruction address. The Injector monitors the instruction

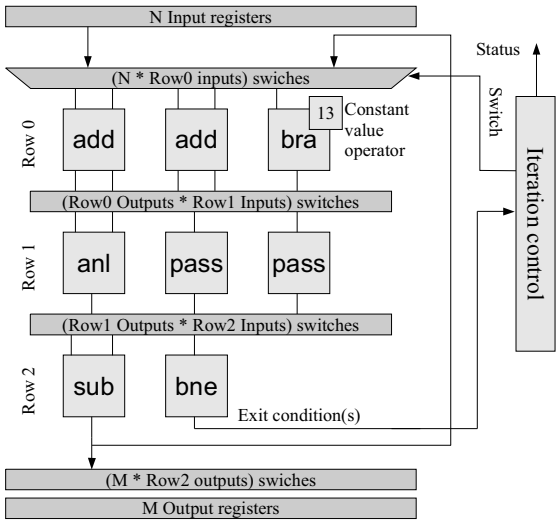

Figure 3. Array of FUs

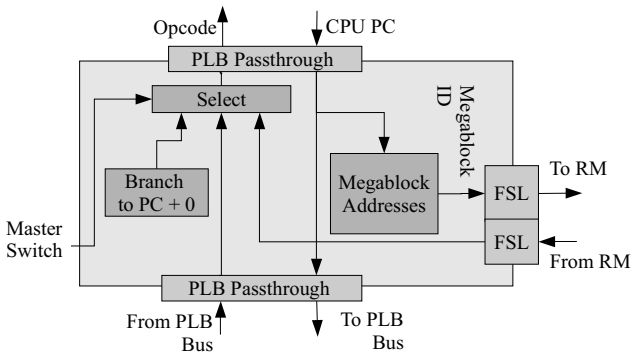

Figure 4. PLB Injector Architecture

addresses placed on the bus by the CPU, until it sees the start of a Megablock. The Injector then stalls the execution of the CPU while reconfiguration is occurring, by feeding an instruction that branches the CPU to the same address and communicates the Megablock ID to the RM. The RM first configures the RPU for that Megablock and replies to the Injector with instructions for the CPU. The instructions will cause the CPU to branch to a memory position containing a previously prepared *Communication Routine* (CR). By executing it, the CPU copies contents from its register file to the appropriate input registers of the RPU. When the RPU execution ends, the CPU completes the CR by retrieving the values from the output registers of the RPU and resumes execution of the program code. In this way the execution flow of the CPU is changed without overwriting the original instructions of the program or interfering with the original software tool chain.

IV. MEGABLOCKS

The architecture of the RPU was heavily influenced by the kind of repetitive patterns we are mapping, the Megablocks [1]. A Megablock is a pattern of instructions in the execution trace of a program and can only be extracted during program execution. Figure 5 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 was proposed in the context of dynamically moving instructions executing in a CPU to reconfigurable arrays. 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 [11]), a Megablock specifically represents a loop.

```
...
        0x00000188 addk r6, r6, r3
        0x00000174 bsra r3, r5, r4
        0x00000178 addik r4, r4, 1
        0x0000017C andi r3, r3, 1
        0x00000180 xori r18, r4, 8
        0x00000184 bneid r18, -16
        0x00000188 addk r6, r6, r3
        0x00000174 bsra r3, r5, r4
        0x00000178 addik r4, r4, 1
        0x0000017C andi r3, r3, 1
        0x00000180 xori r18, r4, 8
        0x00000184 bneid r18, -16
        0x00000188 addk r6, r6, r3
        0x00000174 bsra r3, r5, r4
...
```

Figure 5. Example of a repeating pattern of instructions in the trace of a 8-bit *count* kernel

For Megablocks to be useful, they must represent a significant part of the execution of a program. Previous work [12] shows that for many benchmarks, Megablocks can have coverage similar or greater than other runtime detection methods, such as monitoring short backward branches (used by Warp [2]).

Megablocks are found by detecting a pattern in the instruction addresses being accessed. For instance, Figure 5 shows a pattern of size 6 (0x174, 0x178, 0x17C, 0x180, 0x184, 0x188). In [12] 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 transformed into a graph. Figure 6 represents the graph obtained from the execution trace of Figure 5.

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 architecture as the identifier of the Megablock, and must define the start of the Megablock unambiguously. We are using the following heuristic to choose the start address: choose the lowest address of the Megablock which appears only once. For the example in Figure 5, the start address according to this heuristic is 0x174.

A. 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 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.

B. Megablock Graph

The graph generated from the Megablock contains all the information needed for correct implementation in hardware. Four types of nodes are considered: Operations, Constants, LiveIns and Exits. LiveIns represent values which must be available at the start of the loop. Usually, their values are internally updated on each iteration. Exits represent points where the loop can exit. Although not represented in Figure 6 for clarity, Exits also have connec-

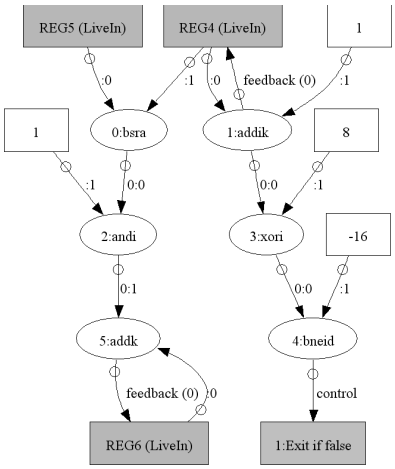

Figure 6. Graph representation of the Megablock found in the *count* kernel.

tions from the nodes that produce the output values that must be sent to the CPU on termination (LiveOuts).

Several types of connections are used to characterize the relationship between the nodes. The more important connections for this work are the data, control and feedback connections. Data connections represent regular producer-consumer relationships between operations. Control connections establish a 1-to-1 relationship between an operation and an exit node. The operation will determine if an exit is activated or not. Feedback connections establish a 1-to-1 relationship between the data output of an operation and a LiveIn. They represent values which need to be updated across iterations.

V. IMPLEMENTATION DECISIONS

A. Megablock Considerations

In a specific Megablock implementation, we need to choose if the iterations are atomic or non-atomic. In an atomic implementation, the smallest execution unit is the iteration. When an iteration triggers an exit, the execution of the iteration until that point is discarded and execution continues in software from the start of the iteration. For a non-atomic implementation, when the iteration exits the execution continues in software exactly after the exit point.

Iteration atomicity has implications in the target architecture. A non-atomic implementation needs to ensure that the state of the system is coherent at each exit point of the Megablock, and needs to keep track of more information. An atomic implementation needs support for state rollback to the beginning of the incorrectly executed iteration.

In the system implementation presented in this paper, we chose to implement Megablocks with atomic iterations. As the RPU used here currently does not support memory accesses, the state rollback is limited to the output registers to be communicated to the CPU, which are internally updated only if an iteration completes successfully.

B. Architecture Considerations

The Injector needs to be attached to the CPU's instruction bus. Since the current implementation of the Injector only supports the PLB bus and does not support the dedicated bus used by the CPU when the instruction cache is activated, the current prototype does not use instruction cache and fetches program instructions from external memory. This is an implementation issue and is not related to the general approach.

The Injector reacts to instruction addresses which correspond to the start of Megablocks mapped to the RPU. Two or more Megablocks may start at the same memory address and so ambiguity may exist. Currently, we support only one Megablock for each start address.

The CPU might fetch instructions which will never execute. This occurs if a Megablock starts after a mispredicted branch instruction. Our implementation correctly identifies this situation: if the starting Megablock address is followed by the next address in the Megablock, this means the CPU did not discard the instruction, and is attempting to execute the code region mapped to the RPU.

C. Toolflow

We developed a tool suite to extract the Megablocks, map them to the RPU, and generate the configuration bits. The tool flow is summarized in Figure 7. We feed the executable file (i.e., ELF file) to the Graph Extractor tool [13] which extracts the Megablocks. This tool uses a cycle-accurate simulator of the CPU to monitor execution traces. A set of Megablocks is processed by two tools: one generates the HDL (Verilog) descriptions for the RPU and the Injector, and the other generates the CRs for the CPU. The HDL description generation tool parses Megablock information, determines FU sharing across graphs, assigns FUs to rows, adds pass-through units, and generates files 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 inter-row switches), as well as the data required for Megablock identification. The generated RPU is tailored to a specific set of Megablocks; switching between members of this set is accomplished by configuring the inter-row switches. Livein values needed at runtime for Megablock execution are transmitted to the RPU, from the CPU's register file, by executing the CRs.

VI. RESULTS

The proposed architecture and tools were tested and eva-

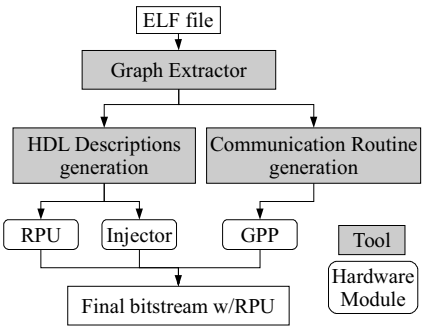

Figure 7. Toolflow

luated with six code kernels: *count*, *even_ones*, *fibonacci*, *hamming_distance*, *pop_cnt* and *reverse*. All kernels work on 32-bit values. An additional example considers all other kernels (*all-merge*), to evaluate the case where several graphs are mapped to the RPU. In this example we alternate the calls to each kernel, since this is the worst case scenario that requires RPU re-configuration between each kernel execution. The kernels are compiled with a parameter *N* which defines how many times the kernel is called. All loop kernels, except for *fibonacci*, have a fixed number of iterations. For *fibonacci*, the number of iterations of the loop is an arithmetic progression of the input value. In all benchmarks, the range from 0 to *N* is used as input value.

A. Setup

We used the Graph Extractor tool to do an offline extraction of the Megablocks from execution traces. For the detection we disabled inner loop unrolling, used Basic Blocks as the pattern unit, set the maximum pattern size to 32 and rejected any Megablock which executed less than 100 instructions. For each kernel (with exception to *all-merge*), we implemented only one Megablock. The majority of the computation was spent in the selected Megablock, the average coverage being around 96% of the executed instructions. For the evaluation, parameter *N* is set to 500, and each kernel is 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 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. The CPU used was a MicroBlaze at 66 MHz with no cache. For our configuration, the synthesis tool allowed a maximum frequency of 83 MHz for the MicroBlaze. However, since we use the same clock signal for all the modules of the system, including the RPU, the frequency is irrelevant when measuring speedups. The FUs of the RPU implement operations which are equivalent to MicroBlaze assembly instructions, and the RM is implemented as another MicroBlaze with the reconfiguration data in its local memory. This additional MicroBlaze can also be used for monitoring purposes.

B. Results

Table I summarizes the characteristics of the Megablocks used in the evaluation. All kernels present similar values for the number of instructions executed per Megablock call, with the exception of *fibonacci*. This is the only kernel where the number of iterations per call depends on the input data. Table I includes values for maximum Instruction Level Parallelism (ILP), percentage of instructions covered by the Megablocks vs. the total executed instructions, and Instructions per Cycle (IPC) assuming each instruction takes one cycle to execute. The Critical Path Length (CPL) has a value of 3 for all cases. For the *all-merge* kernel, these values are the maximum of the individual kernel values.

Table II summarizes the characteristics of the RPU for each kernel. Due to the interconnection scheme we used, most of the FUs are pass-throughs. The *OP Ratio* column represents the percentage of FUs that are not pass-throughs. However, the resulting RPUs were relatively small. All RPUs had a depth of three. Due to FU sharing, the *all-merge* benchmark uses about 32% of the total number of FUs for the individual Megablocks. Since the RPU only reconfigures connections and not FUs, the number of configuration bits for each RPU is also low (204 bits for the RPU which implements six Megablocks).

TABLE I. DETECTED MEGABLOCK CHARACTERISTICS

Kernels	Megablock Characteristics			
	Avg.Inst.Executed p/call	*Max. ILP*	*Coverage*	*IPC*
count	192	2	95%	2.00
even_ones	192	3	94%	2.00
fibonacci	1,497	3	99%	2.00
ham_dist	192	3	94%	2.00
pop_cnt	156	3	97%	2.67
reverse	224	3	96%	2.33
all-merge	425.5	3	98%	2.67

TABLE II. RPU CHARACTERISTICS

Kernels	RPU Characteristics			
	FUs	*Max. FUs per row*	*OP Ratio (%)*	*Config. Bits*
Count	12	4	50.00	72
even_ones	16	6	37.50	87
Fibonacci	15	6	40.00	87
Ham_dist	15	6	40.00	81
pop_cnt	15	6	53.33	84
Reverse	14	6	50.00	81
all-merge	28	8	64.29	204

Table III characterizes the FPGA implementation of the RPUs. The maximum clock frequencies of the RPUs ranged from 85 to 139MHz, which is above the clock frequency of the MicroBlaze. Individual RPUs do not use more than 9% (2369) of the LUTs and 2% (1170) of the FFs. The *all-merge* RPU uses about 55% of the LUTs and 27% of the FFs that would be needed if the RPU was generated with no sharing of FUs. The maximum allowed clock frequency of the combined RPU (85 MHz) is still above the used MicroBlaze clock frequency.

Figure 8 presents speedups for two cases. In the first one, referred as DDR case, results are obtained from ex-

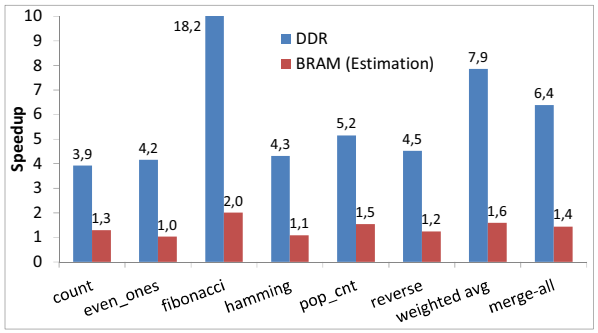

Figure 8. Speedups for DDR and BRAM scenario

ecution on the FPGA and running the kernels from DDR memory. The execution times were measured using timers.

Results include all communication overheads. In the DDR scenario, the MicroBlaze has a 23 cycle penalty for each instruction it executes. Most of the achieved speedup comes from avoiding execution of instructions and executing operations on the RPU instead. However, for each call to the RPU, the CPU executes a CR which passes the values to the RPU through the bus. Since the CRs are in DDR, they also incur that penalty (Table IV). The DDR access latency is the main contributor to the very high overhead of this scenario.

TABLE III. RPU FPGA IMPLEMENTATION

Kernels	FPGA Implementation		
	LUTs	*FFs*	*Max. Freq.(MHz)*
count	1433	926	99.30
even_ones	2331	1153	132.83
Fibonacci	2369	1170	121.56
Ham_dist	1739	1086	138.08
pop_cnt	1758	1058	137.97
reverse	1780	1072	139.06
all-merge	6325	1719	85.19

TABLE IV. COMMUNICATION OVERHEAD

Kernels	Communication Overhead		
	#Inst .of CRs	*DDR (%)*	*BRAM (%)*
count	32	92.44	51.23
even_ones	34	92.05	55.80
Fibonacci	34	63.21	13.94
Ham_dist	33	91.76	53.63
pop_cnt	33	92.05	53.63
reverse	33	92.32	53.63
all-merge	~33	86.89	22.93

The situation is aggravated by the relatively low number of instructions executed per call (Table I). 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. The same overheads were considered in the estimates for the second case. The speedups measured for the DDR scenario include all overheads and range from 3.9× to 18.2×.

The second set of results (BRAM case) was obtained by estimation, considering that the programs are stored in internal memory (BRAMs). The BRAM scenario is the best possible case for the MicroBlaze processor regarding performance. When considering the BRAM scenario, the speedups and the overheads are significantly reduced, since there is no longer a high penalty for fetching instructions from memory. We used a cycle-accurate MicroBlaze simulator [13] to calculate the execution time on the CPU. We estimated the execution time for CRs considering an average of 1.18 cycles per executed instruction, added to the PLB latency of 9 cycles to write/read operands/results to/from the RPU. RPU execution cycles were calculated by multiplying the RPU's depth and the number of iterations. We estimate speedups between 1.03× and 2.01× (including all overheads).

VII. CONCLUSION

This paper presented an automated approach to transparently moving computations from instruction traces to reconfigurable hardware, without changing the executable binary. We used Megablocks as the loop structures.

Our current system is fully functional and runtime reconfigurable. Although the prototype results were contaminated by high memory access latencies, we estimate reasonable speedups when the CPU directly accesses data stored in local memories. With the current type of coupling, the system is easily adaptable to other CPUs.

Currently, we consider extending the prototype with the dynamic identification and mapping of Megablocks. Future work will address the support of caches and/or moving the entire execution context to local memories, as well as support for memory and floating point operations.

REFERENCES

[1] J. Bispo and J. M. P. Cardoso, "On Identifying and Optimizing Instruction Sequences for Dynamic Compilation," in Intl. Conf. on Field-Programmable Tech., Beijing, China, 2010, pp. 437-440.

[2] R. Lysecky, G. Stitt, et al., "Warp Processors," ACM Trans. Des. Autom. Electron. Syst., vol. 11, pp. 659-681, 2006.

[3] R. Lysecky and F. Vahid, "Design and implementation of a MicroBlaze-based warp processor," ACM Transactions on Embedded Computing Systems, vol. 8, pp. 1-22, 2009.

[4] A. Mehdizadeh, B. Ghavami, et al., "An efficient heterogeneous reconfigurable functional unit for an adaptive dynamic extensible processor," in VLSI-SoC'07, 2007, pp. 151-156.

[5] N. Hamid, M. Farhad, et al., "An architecture framework for an adaptive extensible processor," J. Supercomput., vol. 45, pp. 313-340, 2008.

[6] N. Clark, M. Kudlur, et al., "Application-Specific Processing on a General-Purpose Core via Transparent Instruction Set Customization," in Proc. 37th Ann. IEEE/ACM Intl. Symp. Microarch., Portland, USA, 2004, pp. 30-40.

[7] N. Clark, J. Blome, et al., "An Architecture Framework for Transparent Instruction Set Customization in Embedded Processors," in Proc. 32nd Ann. Intl. Symp. Computer Architecture, 2005, pp. 272-283.

[8] A. C. S. Beck, M. B. Rutzig, et al., "Transparent reconfigurable acceleration for heterogeneous embedded applications," in Proc. Conf. Design, Automation and Test in Europe, Munich, Germany, 2008, pp. 1208-1213.

[9] A. C. Beck, M. B. Rutzig, et al., "Run-Time Adaptable Architectures for Heterogeneous Behavior Embedded Systems," in Proc. 4th Intl. Works. Reconf. Comput.: Architectures, Tools and Applications, 2008, pp. 111-124.

[10] S. J. Patel and S. S. Lumetta, "rePLay: A hardware framework for dynamic optimization," IEEE Transactions on Computers, vol. 50, pp. 590-608, June 2001.

[11] S. A. Mahlke, D. C. Lin, et al., "Effective compiler support for predicated execution using the hyperblock," in Proc. 25th Ann. Intl. Symp. on Microarch., ed: IEEE Computer Society Press, 1992, pp. 45-54.

[12] J. Bispo and J. M. P. Cardoso, "On Identifying Segments of Traces for Dynamic Compilation," in Intl. Conf. on Field Programmable Logic and Appl. (FPL'10), Milano, Italy, 2010, pp. 263-266.

[13] J. Bispo, "Megablock Tool Suite - Graph Extractor v0.19," ed. IST/UTL, Lisboa, Portugal, 2011.