A Partitioning Flow for Accelerating Applications in Processor-FPGA Systems

MICHALIS D. GALANIS[1], GREGORY DIMITROULAKOS[2], COSTAS E. GOUTIS[3]
VLSI Design Laboratory, Electrical & Computer Engineering Department
University of Patras, Rio Campus, GREECE

Abstract: - This paper presents a hardware/software partitioning flow for improving performance in systems-on-chip comprised by processor and Field Programmable Gate Array. Speedups are achieved by executing critical software parts on the reconfigurable FPGA logic. A generic hybrid system architecture is considered by the methodology. The partitioning flow uses an automated analysis process at the basic-block level for detecting critical application parts. Two different instances of the generic platform and five real-world applications are used in the experiments. The analytical experimentation illustrates that the speedup of the applications ranges from 1.3 to 3.7 relative to an all software solution.

Key-Words: - Performance improvement, Hardware/software partitioning, FPGA, Embedded systems, Analysis.

1 Introduction

In past few years, academic [1], [2] and commercial [3], [4], [5], [6] single-chip platforms emerged that employ processor(s) with Field Programmable Gate Array (FPGA) logic. These System-on-Chip (SoC) platforms are mainly composed by 8-bit microcontrollers, as in the ATMEL's Field Programmable System-Level Integrated Circuit (FPSLIC) [5], in Triscend's E5 device [6] and 32-bit processors as in the Altera's Excalibur [4], in Xilinx's Virtex-II Pro [3], Triscend's A7 and in Garp architecture [1]. A significant advantage of using FPGA logic is that the functionality of custom made coprocessors or peripherals implemented in this logic, can be changed due to the reconfiguration capabilities of such devices. This is not the case in the implementation in Application Specific Integrated Circuits (ASIC), where a small change in an application or in a standard requires the re-design of the ASIC component. Additionally, significantly less time is spent in implementing a design in FPGA technology than in ASIC one. The microprocessor-FPGA SoCs are expected to become more widespread in the future due to emergence of standards, like telecom ones, that their specification changes over time to meet the contemporary demands. For example, this is already the case in the Wireless LAN standards IEEE 802.11x.

It is important to efficiently utilize the reconfigurable logic in microprocessor-FPGA SoCs. A hardware/software partitioning methodology that divides the application into software running on the microprocessor and on the FPGA logic is essential for such systems. Partitioning can improve performance [7], [8] and in some cases even reduce power consumption [9]. More recently, hardware/software partitioning techniques for SoCs composed by a microprocessor and FPGA [10], [11], [12], [13], were developed. The FPGA unit is treated as an extension of the microprocessor. Critical parts of the application, called *kernels*, are moved for execution on the FPGA for improved performance and usually reduced energy consumption. This design choice stems from the observation that most embedded DSP and multimedia applications spend the majority of their execution time in few small code segments (usually loops), the kernels. This means that an extensive solution space search, as in past hardware/partitioning works [7], [8], [9], is not a requisite.

In this work, we propose a hardware/software partitioning flow for accelerating software kernels of an embedded application on the reconfigurable logic of a generic processor-FPGA SoC. The processor executes the non-critical part of the application's software. This type of partitioning is possible in embedded systems, where the application is usually invariant during the lifetime of the system or of the specification. The considered processor-FPGA architecture can model a variety of existing systems, like the ones considered in [3], [4], [5], [6]. Furthermore, the proposed flow considers the communication time for exchanging data values

between the FPGA and the processor, which was not the case in past works for partitioning in processor-FPGA systems [11], [12], [13].

An analysis tool at the basic block (BB) level has been developed. The term basic block expresses a sequence of instructions (operations) with no branches into or out of the middle. At the end of each basic block there is a branch instruction that controls which basic block executes next. The basic block is actually a Data Flow Graph (DFG). This tool identifies kernels in the input software and targets RISC processor based SoCs, which is the typical case in both academia and in industry [1]-[6].

For evaluating the hardware/software partitioning methodology, we have used two different instances of the considered processor-FPGA platform: (i) four embedded 32-bit processors coupled with two devices from the Xilinx's Virtex FPGA family, and (ii) an 32-bit processor with two devices from the Altera's APEX FPGAs [4]. The (ii) platform instance corresponds to the processor and the FPGA units used in the Altera's Excalibur family [4].

We have used five real-life applications, coded in C language, in the experiments: an IEEE 802.11a Orthogonal Frequency Division Multiplexing (OFDM) transmitter, a video compression technique, a medical imaging application, a wavelet-based image compressor [14] and a JPEG compliant image encoder. The analytical performed experiments show that the kernels in the five real-world applications contribute an average of 69% of the total dynamic instruction count, while their size is 11% on average of the total code size. For the Virtex-based platform the speedups of the five applications range from 1.3 to 3.7, while for the Excalibur-simulated SoC the speedups are from 1.3 to 3.2 relative to the all-software solution.

The rest of the paper is organized as follows: section 2 describes the hardware/software partitioning methodology. Section 3 presents the analytical experiments for the two different platforms. Finally, section 4 concludes this paper and describes future activities.

2 Hardware/software partitioning flow

2.1 Hybrid system architecture

A generic view of the considered hybrid SoC architecture is shown in Fig. 1. The platform includes: (a) an FPGA for executing kernels, (b) shared system data memory, and (c) an embedded microprocessor. The microprocessor is typically a RISC processor, like an ARM7 [15]. Communication between the FPGA and the microprocessor takes place via the system's shared data memory. Direct communication is also present between the FPGA and the processor. Part of the direct signals is used by the processor for controlling the FPGA by writing values to configuration registers located in the FPGA. The rest direct signals are used by the FPGA for informing the processor. For example, an interrupt signal is typically present which notifies the processor that the execution of a critical software part on the FPGA has finished. Local data memories exist in the FPGA for quickly loading data, as in modern FPGAs [3], [4], [5]. This generic system architecture can model the majority of the contemporary processor-FPGA systems, like the ones considered in [3], [4], [5], [6].

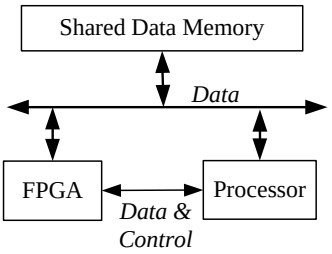

Fig. 1. Target hybrid SoC.

2.2 Flow description

The proposed hardware/software partitioning flow for processor-FPGA systems interests in increasing application's performance by mapping critical software parts on the reconfigurable hardware. The flow of the methodology is shown in Fig. 2. The input is a software description of the application in a high-level language, like C/C++. Firstly, the Control Data Flow Graph (CDFG) Intermediate Representation (IR) is created from the input source code. The CDFG is the input to the analysis step. In the kernel detection, an ordering of the basic blocks in terms of the computational complexity is performed. The basic block's complexity is represented by the instruction count, which is the number of instructions executed in running the application on the microprocessor. The dynamic instruction count has been used as a measure of identifying critical loop structures in previous work [12]. However, in this work the computational complexity is defined at a smaller granularity, at the basic block level. The instruction count is found by a combination of dynamic (profiling) and static analysis. A threshold, set by the designer, is used to

characterize specific basic blocks as kernels. The rest of the basic blocks are going to be executed on the processor.

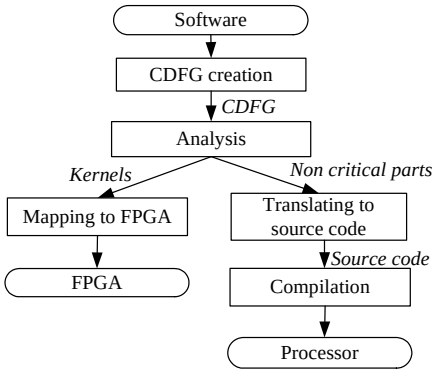

Fig. 2. Hardware/software partitioning flow.

The kernels are synthesized on the FPGA architecture for acceleration. The non-critical application's parts are converted from the CDFG IR back to the source code representation. Then, the source code is compiled using a compiler for the specific processor and it is executed on the microprocessor. The separation of the application's part to the critical and non-critical parts, defines the data communication requirements between the processor and the FPGA. The proposed design flow considers the data exchange time through the shared memory for calculating the application's execution time, which is not the case in previous works for single-chip processor-FPGA systems [11]-[13].

Currently, we consider the case where the processor and the FPGA execute in mutual exclusion. The kernels are replaced in the software description with calls to FPGA. When a call to FPGA is reached in the software, the processor activates the FPGA and the proper state of the Finite State Machine (FSM) is enabled on the FPGA for executing the kernel. The data required for the kernel execution are written to the shared data memory by the processor. Then, these data are read by the FPGA. After the completion of the kernel execution, the FPGA informs the processor by typically using a direct interrupt signal and writes the data required for executing the remaining software. Then, the execution of the software is continued on the processor and the FPGA remains idle. Since the partitioning flow interests in accelerating a sequential software program, which is often the case in implementing embedded applications in a high-level language like C, the speedups from the parallel execution of the FPGA and the processor could be likely small. We mention that works in single-chip processor-FPGA systems [10], [11], [12], [13] also assumed a mutual exclusive operation.

With the mutual exclusive operation of the processor and the FPGA, the total number of execution cycles after hardware/software partitioning is:

$$Cycles_{hw/sw} = Cycles_{sw} + Cycles_{FPGA} \quad (1)$$

where $Cycles_{sw}$ represents the number of cycles needed for executing non-critical parts on the processor, $Cycles_{FPGA}$ corresponds to the cycles that are required for executing the kernels on the FPGA. The communication time between the processor and the FPGA is included in the $Cycles_{sw}$ and in the $Cycles_{FPGA}$ since load and store operations that refer to the shared memory are present in the non-critical parts and in the kernels of each application. The $Cycles_{hw/sw}$ are multiplied with the clock period of the processor for calculating the total execution time $t_{hw/sw}$ after the partitioning.

For estimating the $Cycles_{FPGA}$ of the application's kernels on the FPGA, we consider the following procedure. We describe each kernel in a synthesizable Register-Transfer Level (RTL) description using VHDL language. Loop unrolling and pipelining transformations are used for achieving better speedup when each kernel is synthesized on the FPGA. Each kernel is a state of an FSM (controller), so that when the kernels are synthesized they could share the same hardware. This sharing is achievable because the kernels are not executed concurrently since they are belonging to a sequential software description. For executing a specific kernel on the FPGA, the proper state of the controller is selected. The reconfigurable logic runs at the maximum possible clock frequency after synthesizing all the kernels of an application. For synthesis, placing and routing of the RTL descriptions of the kernels, standard commercial tools can be used. In this work, we have utilized the Synplify Pro (ver. 7.3.1) of the Synplicity Inc. [16].

Parts of the hardware/software partitioning methodology have been automated for a software description in C language. In particular, for the CDFG creation from the C code, we have used the SUIF2 [17] and MachineSUIF compiler infrastructures [18]. The automation of the analysis step is described in sub-section 2.3. For the translation from the CDFG format to the C source code, the *m2c* compiler pass from the Machine-SUIF distribution is used.

2.3 Analysis

The analysis step of the partitioning methodology outputs the kernel and non-critical parts of the input software description. The inherent computational complexity of basic blocks, represented by the

dynamic instruction count, is a rational measure to detect dominant kernels. The number of instructions executed when an application runs on the microprocessor is obtained by a combination of dynamic and static analysis within basic blocks. Fig. 3 shows the diagram of the analysis.

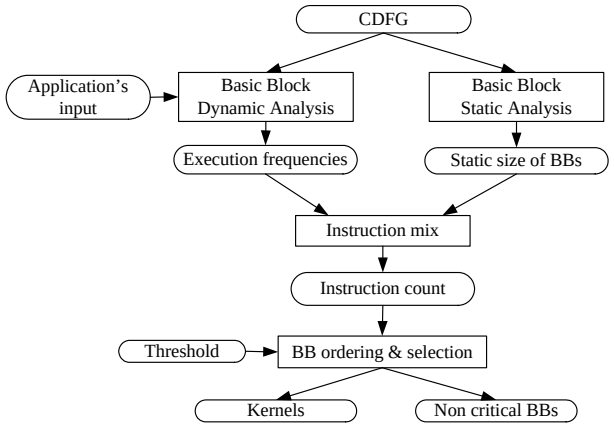

Fig. 3. Analysis procedure.

The input to the analysis process is the CDFG IR of the source code. For the CDFG representation, we have chosen the SUIFvm representation for the instruction opcodes inside basic blocks [18]. The SUIFvm instruction set assumes a generic RISC machine, not biased to any existing architecture. Thus, the information obtained from the analysis, could stand for any RISC processor architecture. This means that the detected critical basic blocks are kernels for various types of RISC processors. The aforementioned statement was justified by experimentation, using the profiling utilities of the compilation tools of the processors considered in the experiments. In fact, the order of the instruction counts of the basic blocks is retained in the RISC processors used in our experiments.

We have used the HALT library of the Machine-SUIF distribution [18] for performing dynamic analysis at the basic block level. The dynamic analysis step reports the execution frequency of the basic blocks. For the static analysis, a MachineSUIF pass has been developed that identifies the type of instructions inside each basic block. Afterwards, a custom developed compiler pass calculates the static size of the basic block using the SUIFvm opcodes. The static size and the execution frequency of the basic blocks are inputs to a developed instruction mix pass that outputs the dynamic instruction count. After the instruction count calculation for each basic block, an ordering of the basic blocks is performed. We consider kernels, the basic blocks which have an instruction count over a user-defined threshold. This threshold represents the percentage of the contribution of the basic block's instruction count in the application's overall instruction count. For example, basic blocks contributing more than 10% to the total instruction count can be considered as kernels.

3 Results

3.1 Experimental set-up

Five DSP applications were used for the experimentation with the two systems composed by 32-bit RISC processors. The applications are: (a) a medical image processing application called cavity detector, (b) an IEEE 802.11a OFDM transmitter, (c) a wavelet-based image compressor [14], (d) a still-image JPEG encoder, and (e) a video compression technique, called Quadtree Structured Difference Pulse Code Modulation (QSDPCM). The experiments are performed with the following inputs: (a) an image of size 640x400 bytes for the cavity detector, (b) 4 payload symbols for the OFDM transmitter at a 54 Mbps rate, (c) an image of size 512x512 bytes for the wavelet-based image compressor, (d) an image of size 256x256 bytes for the JPEG encoder, and (e) two video frames of size 176x144 bytes each for the QSDPCM.

3.2 Analysis results

The results using the developed analysis flow are shown in Table 1. The contributions of the kernels to the total static size (in instruction bytes) and to the total instructions are reported. The threshold for the kernel detection was set to the 10% of the total dynamic instructions of the application. The number of kernels detected in each application is also given. The kernels of the five applications are loop bodies without conditional statements inside them.

Table 1. Results from the analysis procedure.

App.	Total Size	Kernels size	% size	% total instructions	# of kernels
Cavity	12,039	910	7.6	79.8	4
OFDM	15,579	1,440	9.2	61.5	4
Compressor	12,835	602	4.7	78.8	4
JPEG	10,995	2,534	23.0	71.3	4
QSDPCM	24,767	2,477	10.0	51.0	3
Average			**10.9**	**68.5**	

From the analysis results, it is inferred that an average of 10.9% of the code size, representing the kernels' size, contributes 68.5% on average to the total executed instructions. Thus, the speedup of an application will come from accelerating a small number of kernels. The results of Table 1 imply that

the usage of exploration algorithms, which typically examine thousands of possible partitions and utilize complex algorithms [7], [8], [9], is not necessary in the case of partitioning the considered applications on the processor-FPGA system.

3.3 Virtex-based systems

The results from partitioning the five applications in a SoC that has a Virtex FPGA device [3] as its reconfigurable logic are given in this section. These results correspond to the speedups after executing the kernels on the FPGA.

We have used four different types of 32-bit embedded RISC processors: an ARM7, an ARM9 [15], and two SimpleScalar processors [19]. The SimpleScalar processor is an extension of the MIPS32 IV core [20]. These processors are widely used in embedded SoCs. The first type of the MIPS processor (MIPSa) uses one integer ALU unit, while the second one (MIPSb) has two integer ALU units. We have used instruction-set simulators for the considered embedded processors for estimating the number of execution cycles. More specifically, for the ARM processors, the ARM Developer Suite (version 1.2) [15] was utilized, while the performance for the MIPS-based processors is estimated using the SimpleScalar simulator tool [19]. Typical clock frequencies are considered for the four processors: the ARM7 runs at 100 MHz, the ARM9 at 250 MHz, and the MIPS processors at 200 MHz. These clock frequencies were taken from reference designs from the ARM and MIPS websites. The five applications were optimized for best performance when compiled for the considered processors.

The performance gains from applying the partitioning flow in the five applications are presented in Fig. 4. For each application, the four aforementioned processor architectures co-exist with the FPGA in the hybrid SoC. We have assumed two different Virtex FPGA devices: (a) the smallest available Virtex device, the XCV50 FPGA, and (b) the medium size device XCV400. The clock frequencies after synthesizing, placing and routing the designs using the Synplify Pro toolset [16], range from 45 to 77 MHz for the XCV50 device and from 37 to 77 MHz for the XCV400. From the speedups shown in Fig. 4, it is evident that significant performance improvements are achieved when critical software parts are mapped on the FPGA. It is noticed that better performance gains are achieved for the ARM7 case than the ARM9-FPGA SoC. This occurs since the speedup of kernels in the FPGA has greater effect when the FPGA co-exists with a lower-performance processor, as it is the ARM7 relative to the ARM9. Furthermore, the speedup is almost always greater for the MIPSa than the MIPSb processor, since the latter one employs one more integer ALU unit.

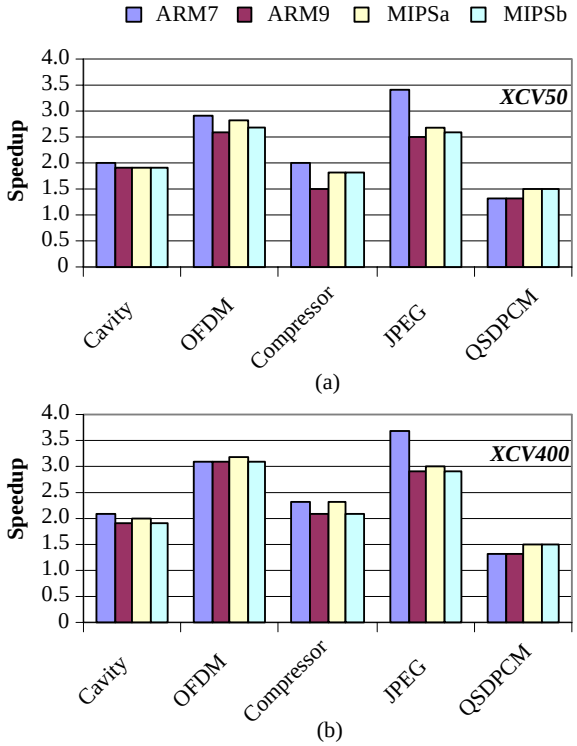

Fig. 4. Speedups from accelerating kernels on (a) XCV50 and (b) XCV400 devices.

For the case of the different Virtex devices, the performance improvements are greater for the XCV400 due to the larger number of Control Logic Blocks (CLBs) which permit the implementation of more operations on the FPGA hardware. This leads to better kernels' acceleration through the larger amount of spatial computation due to the increased number of instantiated operations in the reconfigurable logic relative to the smaller FPGA device, the XCV50. The average speedup for the five applications is 2.1 for the XCV50 and 2.4 for the XCV400.

3.4 Excalibur-simulated systems

The results from accelerating the kernels of the five applications on the Excalibur-simulated system [4] are given in this section. In the Excalibur devices, an ARM9 processor is used that it is clocked at 200MHz, which is the also case in these experiments. The applications were again optimized for best performance when compiled for the ARM9. The ARM Developer Suite was used for estimating the cycles required for the software execution. Two

cases of APEX FPGAs are utilized for simulating the EPXA1 and the EPXA10 Excalibur devices, where the EPXA10 stands for a larger amount of reconfigurable logic. After the kernels' synthesis with the Synplify Pro, the reported clock frequencies range from 20 to 38 MHz for the EPXA1, and from 22 to 30 MHz for the EPXA10.

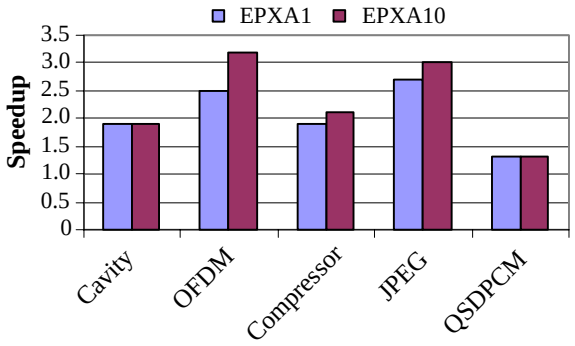

Fig. 5. Speedups for the Excalibur-simulated systems.

The speedups after partitioning are given in Fig. 5. Greater improvements are achieved for the EPXA10-simulated system, as in the case of the Virtex-based SoCs, where greater performance was achieved for the larger Virtex device. The average speedup is 2.1 for the EPXA1 and 2.3 for the EPXA10. Comparing the speedups of Fig. 5 with the respective ones for the ARM9-Virtex system, they are approximately the same although the ARM9 is clocked at a lower speed and the clock frequencies after the kernel synthesis on the APEX devices, are smaller than the ones on the Virtex FPGAs.

4 Conclusions

A partitioning flow for speeding-up critical software parts in processor-FPGA systems was presented. Five DSP applications were executed on two instances of a generic processor-FPGA platform. Important performance improvements, which range from 1.3 to 3.7, have been achieved.

Acknowledgements

This research has been partly funded by the Alexander S. Onassis Public Benefit Foundation.

References:

[1] T.J. Callahan, J. R. Hauser, J. Wawrzynek, "The Garp Architecture and C Compiler", in *IEEE Computer*, vol. 33, no. 4, pp 62-69, April 2000.

[2] S. Hauck, T.W. Fry, M.M Hosler, J.P Kao, "The Chimaera Reconfigurable Functional Unit", in *IEEE Trans. on VLSI Syst.*, vol.12, no.2, pp. 206-217, Feb. 2004.

[3] Virtex FPGAs, Xilinx Inc., www.xilinx.com, 2005.

[4] Excalibur FPGAs, Altera Inc., www.altera.com, 2005.

[5] FSPLIC, ATMEL Inc., www.atmel.com, 2005.

[6] Triscend Corp. www.triscend.com, 2004.

[7] P. Eles, Z. Peng, K. Kuchchinski and A. Doboli, "System level hardware/software partitioning based on simulated annealing and tabu search", in *Design Automation for Embedded Systems*, Springer, vol. 2, no. 1, pp. 5-32, Jan. 1997.

[8] D.D. Gajski, F. Vahid, S. Narayan, and J. Gong, "SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design", in *IEEE Trans. on VLSI Syst.*, vol. 6, no. 1, pp. 84–100, March 1998.

[9] J. Henkel, "A low power hardware/software partitioning approach for core-based embedded systems", in *Proc. of the 36th ACM/IEEE DAC*, pp. 122–127, 1999.

[10] A. Ye, N. Shenoy, P. Baneijee, "A C Compiler for a Processor with a Reconfigurable Functional Unit", in *Proc. of FPGA*, pp. 95-100, 2000.

[11] K. Bazargan, R. Kastner, S. Ogrenci, M. Sarrafzadeh, "A C to Hardware/Software Compiler", in *Proc. of FCCM '00*, pp. 331-332, 2000.

[12] J. Villareal, D. Suresh, G. Stitt, F. Vahid, W. Najjar, "Improving Software Performance with Configurable Logic", in *Design Automation for Embedded Systems*, Springer, vol. 7, pp. 325-339, 2002.

[13] G. Stitt, F. Vahid, S. Nematbakhsh, "Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems", in *ACM TECS*, vol.3, no.1, pp. 218-232, Feb. 2004.

[14] Honeywell,http://www.htc.honeywell.com/projects/acsbench, 2005.

[15] ARM Corp, www.arm.com, 2005.

[16] Synplify Pro, www.synplicity.com, 2005.

[17] SUIF2,http://suif.stanford.edu/suif/suif2/index.html, 2005.

[18] MachineSUIF, http://www.eecs.harvard.edu/hube/research/machsuif.html, 2005.

[19] SimpleScalar, www.simplescalar.com, 2005.

[20] MIPS Corp., www.mips.com, 2005.