The Journal of Supercomputing, 35, 185–199, 2006

Performance Improvements from Partitioning Applications to FPGA Hardware in Embedded SoCs

MICHALIS D. GALANIS mgalanis@ee.upatras.gr
GREGORY DIMITROULAKOS
COSTAS E. GOUTIS
VLSI Design Laboratory, ECE Department, University of Patras, Patras 26500, Greece

Abstract. A hardware/software partitioning methodology for improving performance in single-chip systems composed by processor and Field Programmable Gate Array reconfigurable logic is presented. Speedups are achieved by executing critical software parts on the reconfigurable logic. A hybrid System-on-Chip platform, which can model the majority of existing processor-FPGA systems, is considered by the methodology. The partitioning method uses an automated kernel identification process at the basic-block level for detecting critical kernels in applications. Three different instances of the generic platform and two sets of benchmarks are used in the experimentation. The analysis on five real-life applications showed that these applications spend an average of 69% of their instruction count in 11% on average of their code. The extensive experiments illustrate that for the systems composed by 32-bit processors the improvements of five applications ranges from 1.3 to 3.7 relative to an all software solution. For a platform composed by an 8-bit processor, the performance gains of eight DSP algorithms are considerably greater, as the average speedup equals 28.

Keywords: hardware/software partitioning, performance improvement, FPGA, embedded system-on-chips, kernel identification

1. Introduction

In past few years, academic [4, 9] and commercial [6, 7, 26, 28] 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) [7], in Triscend's E5 device [26] and 32-bit processors as in the Altera's Excalibur [6], in Xilinx's Virtex-II Pro [28], Triscend's A7 and in Garp architecture [4]. 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 (WLAN) standards IEEE 802.11x [14].

It is important to efficiently utilize the reconfigurable logic in single-chip microprocessor-FPGA systems. 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 [5, 8] and in some cases even reduce power consumption [10]. More recently, hardware/software partitioning techniques for SoC platforms composed by a microprocessor and FPGA [2, 20, 21, 27, 29], 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 [5, 8, 10], is not a requisite.

In this work, we propose a hardware/software partitioning methodology 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 generic processor-FPGA architecture can model a variety of existing systems, like the ones considered in [6, 7, 26, 28]. Furthermore, the proposed method 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 [2, 20, 21, 27].

A kernel identification 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. This tool identifies kernels in the input software and targets RISC processor based SoCs, which is the mainstream case in both academia and in industry [4, 6, 7, 9, 26, 28].

For verifying the hardware/software partitioning method, we have used three 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,
(ii) an 32-bit processor with two devices from the Altera's APEX FPGAs [6], and
(iii) an 8-bit microcontroller coupled with an ATMEL's AT40K FPGA device [7].

The (ii) and (iii) platform instances correspond to the processor and the FPGA units used in the Altera's Excalibur family [6] and the ATMEL's FPSLIC [7], respectively.

We have used two set of benchmarks for the experimentation:

(a) five real-life applications coded in C language: an IEEE 802.11a Orthogonal Frequency Division Multiplexing (OFDM) transmitter [14], a video compression technique [22], a cavity detector [3], a wavelet-based image compressor [12] and a JPEG compliant image encoder [15]. This set of benchmarks is used for the partitioning experiments with the 32-bit systems.

(b) Eight DSP and multimedia algorithms, coded in C, from the Texas Instruments (TI) benchmark suite [25]. This set of benchmarks is used for the FPSLIC-simulated platform.

The extensive 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 an all-software solution. The performance improvements of the TI's algorithms on the FPSLIC-simulated platform range from 3.2 to 68.4, with an average value of 28.1.

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

2. Hardware/software partitioning methodology

2.1. Hybrid SoC architecture

A general diagram of the considered hybrid SoC architecture is shown in Figure 1. The platform includes: (a) FPGA logic for executing software kernels, (b) shared system data memory, (c) instruction and configuration memories, and (d) an embedded microprocessor. The microprocessor is typically a RISC processor, like an ARM7 [1]. 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 from 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.

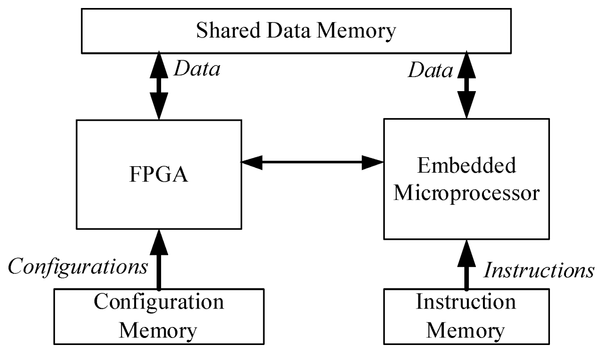

Figure 1. Considered hybrid SoC.

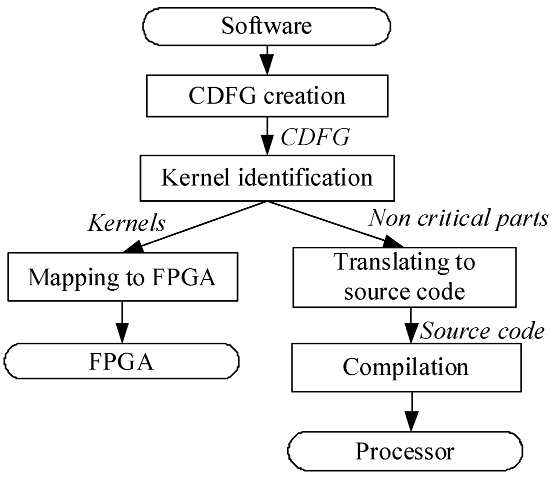

Figure 2. Hardware/software partitioning flow.

Local data memories exist in the FPGA for quickly loading data, as in modern FPGAs [6, 7, 28]. The main configuration memory of Figure 1 is used to store the whole configuration bitstream for programming the execution of the application's kernels on the FPGA. This generic system architecture can model the majority of the contemporary processor-FPGA SoCs, like the ones considered in [6, 7, 26, 28].

2.2. *Methodology description*

The proposed hardware/software partitioning method 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 illustrated in Figure 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 kernel identification step. In the kernel detection, an ordering of the basic blocks in terms of the computational complexity is performed. The computational complexity is represented by the instruction count, which is the number of instruction 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 [27]. 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.

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 run on the microprocessor. After the hardware/software partitioning, there is a separation of the application since there are parts (non-critical ones) that they are going to be executed on the processor and parts (the kernels) which are executed on the FPGA. This separation of the application to critical and non-critical parts, defines the data communication requirements between the processor and the FPGA. The proposed design method considers the data exchange time through the shared memory for calculating the application's execution time, which is not the case in previous works in processor-FPGA SoCs [2, 20, 21, 27].

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. When the FPGA executes a specific critical software part, the processor usually enters an idle state for reducing power consumption. After the completion of the kernel execution, the FPGA signals an interrupt that causes the processor to continue executing the rest of the application. So, the FPGA, by asserting the interrupt, wakes up the processor. Additionally, the FPGA 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. With the mutual exclusive operation, the processor and the FPGA never access the data memory concurrently, fact that simplifies the system architecture. Since the partitioning method interests in accelerating a sequential software program, which is often the case in implementing embedded applications in a high-level language like C, the performance gains from concurrent execution of the FPGA and the processor could be likely small. We mention that works in single-chip processor-FPGA systems [2, 20, 21, 27, 29] also assumed a mutual exclusive operation. However, the parallel execution on the processor and on the FPGA is a topic of our future research activities.

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

$$Cycles_{hw/sw} = Cycles_{sw} + Cycles_{\mathrm{FPGA}} + Cycles_{\mathrm{comm}} \tag{1}$$

where $Cycles_{sw}$ represents the number of cycles needed for executing non-critical parts on the processor, $Cycles_{\mathrm{FPGA}}$ corresponds to the cycles that are required for executing the kernels on the FPGA, and $Cycles_{\mathrm{comm}}$ is the time required for transferring data, through the shared data memory of Figure 1, between the processor and the FPGA. 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 $Cycles_{\mathrm{FPGA}}$ 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 performance 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 doable 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. [24].

Parts of the hardware/software partitioning method 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 [23] and MachineSUIF compiler infrastructures [16], as it is described in Section 2.3. The automation of the kernel identification step is described in Section 2.4. For the translation from the CDFG format to the C source code, the *m*2*c* compiler pass from the Machine-SUIF distribution is utilized.

2.3. CDFG creation

Figure 3 shows the flow for extracting the CDFG from an application coded in C. For this purpose the SUIF2 [23] and Machine-SUIF [16] toolsets are used. We have utilized existing compiler passes in the SUIF2/Machine-SUIF distribution for constructing the CDFG. We have also developed a new pass (*cfg_to_cdfg*), which is shown in the shaded box in Figure 3.

The *c2suif* compiler pass is used to transform the C source code to a SUIF2 High-Level IR (HIR) representation. Constructs like *for* loops, *while* loops, and *if-then-else* structures remain visible in the HIR. The HIR is the input to the *lower* pass, which performs various transformations, as loop and conditional statements dismantling to lower operations. An instruction in a basic block in the Machine-SUIF has an opcode that describes its functionality, a set of input operands and an output operand. The output of the *lower* pass is a Low-Level IR (LIR). The LIR is transformed to SUIF Virtual

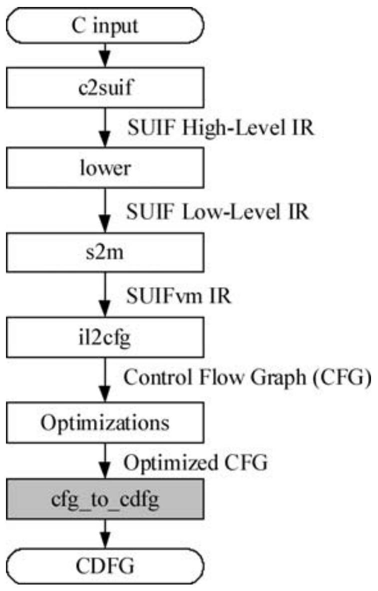

Figure 3. CDFG creation from C code.

Machine (SUIFvm) IR [11] with the *s2m* pass. The opcodes of the SUIFvm library are architecture independent. The *il2cfg* pass of the Machine-SUIF distribution transforms the SUIFvm IR to Control Flow Graph (CFG) format. Then, optimizations like Constant Sub-expression Elimination (CSE), Constant Propagation, Dead Code Elimination, are performed to produce optimized CFG. Finally, our *cfg_to_cdfg* pass constructs for each CFG node the corresponding Data Flow Graph (DFG); thus the CDFG of the C input code is produced.

2.4. *Kernel identification*

The kernel identification step of the partitioning methodology outputs the kernel and non-critical parts of the software description. The inherent computational complexity of basic blocks, represented by the dynamic instruction count, is a meaningful measure to detect dominant kernels. The number of instructions executed when an application runs on the microprocessor is obtained by a combination of profiling and static analysis within basic blocks. Figure 4 shows the diagram of the kernel identification.

The input to the kernel identification process is the CDFG IR of the input source code. As already mentioned, for the CDFG representation, we have chosen the SUIFvm representation for the instruction opcodes inside basic blocks [11]. The SUIFvm instruction set assumes a generic RISC machine, not biased to any existing architecture. Thus, the information obtained from the kernel identification, could stand for any RISC

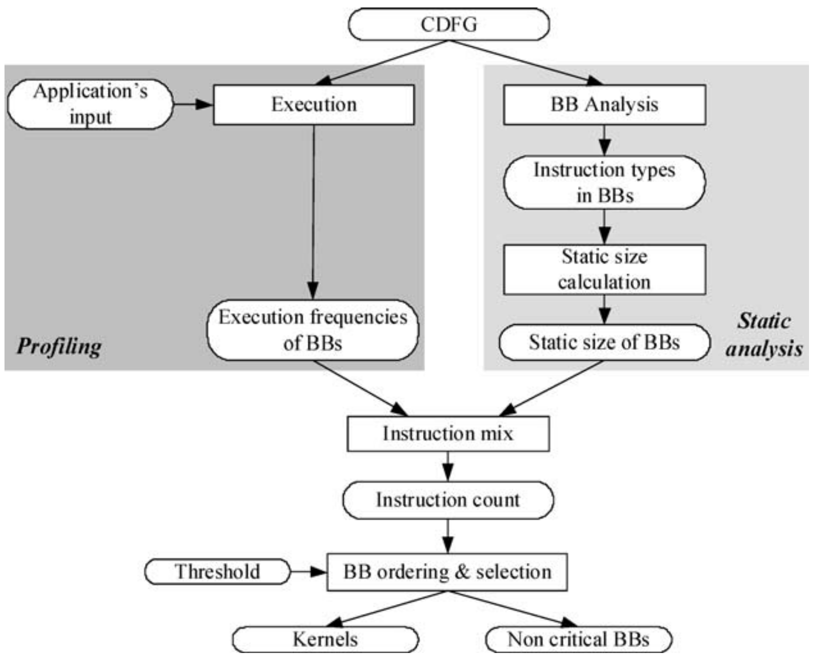

Figure 4. Kernel identification procedure.

processor architecture. This means that the detected critical software parts 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 [17] of the Machine-SUIF distribution for performing *profiling* at the basic block level. The profiling 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% in the total instruction count can be considered as kernels.

3. Experimental results

3.1. Set-up

Two set of benchmarks are used for validating the proposed hardware/software partitioning methodology. The first one consists of five applications and it is used for the experimentation with the Virtex-based [28] and the Excalibur-simulated [6] systems which are both composed by 32-bit RISC processors. The first application is an IEEE 802.11a OFDM transmitter [14]. The second one is a cavity detector which is a medical image processing application [3]. The third one is a video compression technique, called Quadtree Structured Difference Pulse Code Modulation (QSDPCM) [22], while the fourth one is a still-image JPEG encoder [15]. Finally, the fifth one is a wavelet-based image compressor [12]. The partitioning experiments are performed with the following applications' inputs: (a) 4 payload symbols for the OFDM transmitter at a 54 Mbps rate, (b) an image of size 640×400 bytes for the cavity detector, (c) an image of size 512×512 bytes for the wavelet-based image compressor, (d) two video frames of size 176×144 bytes each for the QSDPCM, and (e) an image of size 256×256 bytes for the JPEG encoder.

The second set of benchmarks is used for the 8-bit FPSLIC-simulated [7] platform. It is composed by smaller applications, actually algorithmic kernels, that can be handled by the computational capabilities of a low-cost and low-performance (compared to the 32-bit processors) 8-bit RISC processor core. These algorithms are derived from the TI's DSP and imaging benchmark suite, which is public available at [25]. Eight representative DSP and multimedia algorithms are used: an 3×3 convolution kernel, a 2-Dimensional (2D) 8×8 Forward Discrete Cosine Transform (FDCT), a 2D 8×8 Inverse DCT (IDCT), a 64-point Fast Fourier Transform (FFT), a complex FIR filter, a 16×16 point

Minimum Absolute Differences (MAD) unit, a 168 by 256 matrix multiplication, and the vertical pass of a 2D Discrete Wavelet Transform (DWT).

For calculating the $Cycles_{\mathrm{comm}}$ in Eq. (1), we assume that the shared data memory is modelled as a multi-port and single clock cycle accessible Static RAM (SRAM). This is a reasonable assumption since our methodology targets configurable SoCs. In modern configurable SoCs-like the Xilinx's Virtex-Pro [28] and the Altera's Excalibur [6]–there are fast on-chip data SRAMs that can be configured in respect to their number of ports and to their access delay. Thus, they can be configured to be multi-ported and single-cycle accessible RAMs as it is the case in the conducted experiments.

3.2. Kernel identification results

The analysis results for the first set of benchmarks using the developed kernels identification flow are shown in Table 1. The results correspond to the kernels of each application. The threshold for the kernel detection was set to the 10% of the total dynamic instructions of the application. We have found by experimentation with the considered applications that a threshold set to the 10% contributes the most to the performance improvements

Table 1. Results from the kernel identification process

Kernel	Static size (bytes)	Instruction count	% total static size	% total instructions
		Cavity detector		
K1	181	367,681,952	1.5	42.5
K2	150	114,265,800	1.2	13.2
K3	150	114,265,800	1.2	13.2
K4	429	95,038,944	3.6	11.0
		OFDM transmitter		
K1	264	304,128	1.7	23.1
K2	862	206,880	5.5	15.7
K3	154	162,624	1.0	12.3
K4	160	138,240	1.0	10.5
		Wavelet-based image compressor		
K1	186	31,664,640	1.4	24.2
K2	173	29,451,520	1.3	22.5
K3	123	21,159,936	1.0	16.2
K4	120	20,643,840	0.9	15.8
		JPEG encoder		
K1	409	25,987,860	3.7	36.6
K2	1,056	8,650,752	9.6	12.2
K3	1,048	8,585,216	9.5	12.1
K4	21	7,455,504	0.2	10.5
		QSDPCM		
K1	121	1,045,161,216	0.5	27.4
K2	1,088	496,336,896	4.4	13.0
K3	1,268	401,702,400	5.1	10.5

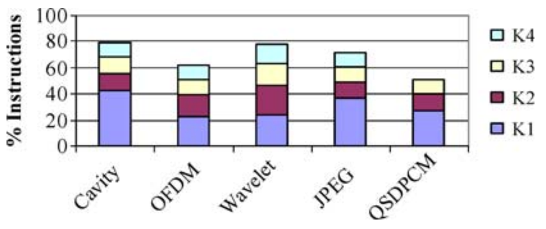

Figure 5. Contribution of the applications' kernels to the instruction count.

when hardware/software partitioning took place on the Virtex-based and the Excalibur-simulated platforms.

The kernels are given in descending order of instruction count. Table 1 reports the kernel's static size in instruction bytes for the SUIFvm opcode representation, the instruction count for each kernel, and the percentages of its contribution to the total static size and instruction count of the application. From the results of Table 1, it is inferred that a small part of the code, the kernel's code, corresponds to the majority of the instruction count, and thus to the execution time of the application. We mention that all the kernels in the considered applications are loop bodies without conditional statements inside them.

Figure 5 illustrates the % instruction count of the kernels to the total instruction count. The contribution of each kernel to the accumulated instruction count for each application is also given. The applications spend an average of 68.5% of their instruction count in 10.9% on average of their code.

3.3. *Virtex-based SoCs*

In this section, we present the results from partitioning the five applications of the first set of benchmarks on a SoC that has a Virtex FPGA device as its reconfigurable logic. 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 [1], and two SimpleScalar processors [19]. The SimpleScalar processor is an extension of the MIPS32 IV core [18]. 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) [1] 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 results from applying the partitioning methodology in the five applications are presented in Table 2. For each application, the four considered processor architectures (*Proc. Arch.*) are used for estimating the clock cycles ($\mathit{Cycles}_{\mathrm{init}}$) required

Table 2. Speedups for the Virtex-based SoCs

				XCV50		XCV400	
App.	Proc. Arch.	$Cycles_{\text{init}}$	Ideal Sp.	$Cycles_{hw/sw}$	Est. Sp.	$Cycles_{hw/sw}$	Est. Sp.
Cavity	ARM7	178,828,950	2.4	87,819,165	2.0	86,707,865	2.1
	ARM9	161,441,889	2.3	86,334,039	1.9	83,555,789	1.9
	MIPSa	470,433,835	2.3	242,474,404	1.9	240,251,804	2.0
	MIPSb	310,248,110	2.2	165,856,481	1.9	163,633,881	1.9
OFDM	ARM7	397,851	3.3	137,070	2.9	126,878	3.1
	ARM9	362,990	3.4	141,791	2.6	116,310	3.1
	MIPSa	459,594	3.4	164,186	2.8	143,801	3.2
	MIPSb	352,788	3.3	130,003	2.7	109,618	3.1
Compressor	ARM7	25,832,508	2.5	12,845,410	2.0	11,399,596	2.3
	ARM9	20,574,658	2.3	13,337,377	1.5	9,722,842	2.1
	MIPSa	62,468,206	2.5	30,410,887	2.1	27,519,259	2.3
	MIPSb	40,541,866	2.3	22,117,384	1.8	19,225,756	2.1
JPEG	ARM7	23,003,868	4.0	6,705,747	3.4	6,215,818	3.7
	ARM9	19,951,193	3.2	8,019,506	2.5	6,794,684	2.9
	MIPSa	34,451,609	3.3	12,542,045	2.7	11,562,186	3.0
	MIPSb	19,637,417	3.2	7,669,659	2.6	6,689,801	2.9
QSDPCM	ARM7	4,026,384,618	1.6	3,069,033,988	1.3	3,054,066,323	1.3
	ARM9	3,895,248,922	1.5	3,052,413,515	1.3	3,014,994,353	1.3
	MIPSa	7,006,016,541	1.7	4,606,022,856	1.5	4,576,087,526	1.5
	MIPSb	4,910,759,258	1.7	3,365,035,660	1.5	3,335,100,331	1.5
				Averages:	2.1		2.4

from executing the whole application on the processor. We have assumed two different Virtex FPGA devices [28]: (a) the smallest available Virtex device, the XCV50 FPGA, and (b) the medium size device XCV400. The ideal speedup (*Ideal Sp.*) reports the maximum performance improvement, according to Amdahl's Law, if application's kernels were ideally executed on the FPGA in zero time. The estimated speedup (*Est. Sp.*) is the measured performance improvement after utilizing the developed partitioning method. The estimated speedup is calculated as:

$$Est\_Sp = Cycles_{\text{init}}/Cycles_{hw/sw} \tag{2}$$

where $Cycles_{hw/sw}$ represents the execution cycles after the partitioning.

The clock frequencies after synthesizing, placing and routing the designs using the Synplify Pro toolset [24], range from 45 to 77 MHz for the XCV50 device and from 37 to 77 MHz for the XCV400. From the results given in Table 2, 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.

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 estimated speedup is 2.1 for the XCV50 and 2.4 for the XCV400. We also notice that the reported estimated speedups for each application and for each processor-FPGA SoC are fairly close to the ideal speedups determined by the Amdahl's Law, especially for the XCV400 case.

3.4. *Excalibur-simulated SoCs*

The results from accelerating the kernels of the five applications of the first set of benchmarks on the Excalibur-simulated SoC [6] are given in this section. In the Excalibur devices, an ARM9 processor is used that it is clocked at 200 MHz, which is also the 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.

The performance gains after the hardware/software partitioning are given in Figure 6. Greater speedups 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 these average speedups with the 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 for the APEX devices, are smaller than the ones in the Virtex FPGAs.

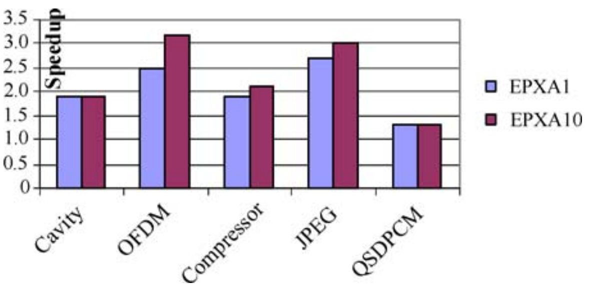

Figure 6. Speedups for the Excalibur-simulated SoCs.

Table 3. Speedups for the FPSLIC-simulated SoC

App.	$Cycles_{\text{init}}$	Ideal Sp.	$Cycles_{hw/sw}$	Est. Sp.
FFT	213,427	3.4	66,883	3.2
Matrix Mult.	1,491,462	36.9	43,445	34.3
FDCT	308,472	34.5	9,528	32.4
IDCT	329,298	35.5	9,858	33.4
Convolution	180,385	5.2	36,001	5.0
MAD	122,374,066	96.9	1,787,826	68.4
FIR complex	259,889	32.9	8,289	31.4
Wavelet	193,524	17.4	11,403	17.0
			Average	28.1

Finally, and in the Excalibur-based system the speedups are quite close to the ideal ones obtained with the Amdahl's Law.

3.5. *FPSLIC-simulated SoC*

In this section we present the speedups after partitioning the second set of benchmarks on an ATMEL's FPSLIC-based system. In FPSLIC devices, an 8-bit AVR core, capable of 30 Millions Instruction Per Second (MIPS), is coupled with a AT40K FPGA. In this experiment, both the AVR and the configurable logic are clocked to 20 MHz. The execution cycles of the non-critical parts of the algorithms on the AVR microcontroller are estimated using the profiling utilities of the Embedded Workbench suite from the IAR Systems Inc. [13]. The algorithms were optimized for best performance when compiled for the AVR. We have used the developed kernel identification flow for detecting critical basic blocks, although manual kernel detection can be performed since these eight algorithms are relatively small programs. Two basic blocks, at maximum, were characterized as kernels, which were the cases in the FDCT, in the IDCT and in the wavelet algorithms. All the kernels are inner for-loops (without conditional structures) of the main computation bodies in each algorithm.

From Table 3, it is deduced that the speedups are significantly greater than the ones obtained for the five large applications with the 32-bit platforms. This is due to two reasons: (a) the kernel(s) of each algorithm contributes to a larger amount to the total instruction count than the kernels in each of the five real-life applications; (b) the speedups are greater when the reconfigurable logic is coupled with a lower-performance instruction-set processor, as it is the AVR relative to the 32-bit processors.

For a straightforward comparison of the larger speedups when a low-performance processor is coupled with reconfigurable logic relative to a higher-performance processor system, we have performed the following experiment. We have assumed a 32-bit ARM7 processor coupled with an AT40K FPGA, both clocked at 20 MHz, as in the case of the AVR-AT40K system. The eight applications from the TI's benchmark suite were partitioned on the ARM7 system. Significantly smaller speedups are achieved for the ARM7-based SoC, as shown in Figure 7. The average value of the execution cycles improvement for the ARM7 platform is 6.0, while for the AVR is 28.1, as illustrated in Table 3.

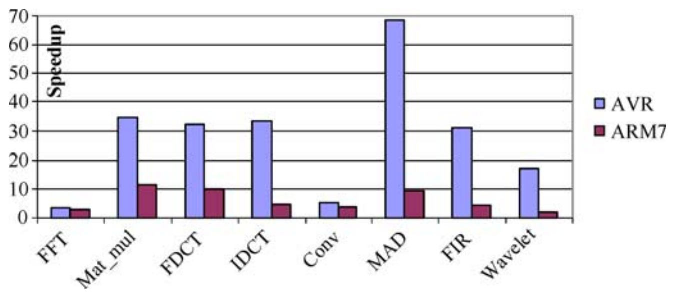

Figure 7. Speedup comparison for AVR and ARM7 AT40-based SoCs.

4. Conclusions-future work

A partitioning method for accelerating critical software parts in processor-FPGA SoCs was presented. Two set of benchmarks were used and three instances of a generic processor-FPGA platform were used. Important speedups have been achieved when the kernels were executed on the FPGA logic. For the Virtex-based system the execution cycles improvement ranges from 1.3 to 3.7, while for the Excalibur-simulated system the speedup ranges from 1.3 to 3.2. For these two 32-bit systems, it is shown that the speedup increases when a larger FPGA is used and the effect of the kernel acceleration on the FPGA is different when a different processor is utilized. For the low-cost 8-bit FPSLIC-simulated SoC, significantly greater speedups are achieved, with an average value of 28.1, when DSP algorithms are partitioned on this system. Future work focuses on the parallel execution of the processor and the FPGA for possible greater performance improvements. Also, the effect of the proposed partitioning methodology in the energy consumption of an application will be researched.

Acknowledgments

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

References

1. ARM Corp, www.arm.com, 2005.
2. K. Bazargan, R. Kastner, S. Ogrenci, and M. Sarrafzadeh. A C to hardware/software compiler. In *Proc. of FCCM'00*, pp. 331–332, 2000.
3. M. Bister, Y. Taeymans, and J. Cornelis. Automatic segmentation of cardiac MR images. *Computers in Cardiology*. IEEE Computer Society Press, pp. 215–218, 1989.
4. T. J. Callahan, J. R. Hauser, and J. Wawrzynek. The garp architecture and C compiler. *IEEE Computer*, 33(4):62–69, 2000.
5. P. Eles, Z. Peng, K. Kuchchinski, and A. Doboli. System level hardware/software partitioning based on simulated annealing and tabu search. *Design Automation for Embedded Systems,* (Springer) 2(1):5–32, 1997.
6. Excalibur devices, Altera Inc., www.altera.com, 2005.

7. FPSLIC devices, ATMEL Inc., www.atmel.com, 2005.
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. *IEEE Trans. on VLSI Syst.*, 6(1):84–100, 1998.
9. S. Hauck, T. W. Fry, M. M. Hosler, and J. P. Kao. The chimaera reconfigurable functional unit. *IEEE Trans. on VLSI Syst.*, 12(2):206–217, 2004.
10. 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.
11. G. Holloway and M. D. Smith. The machine-SUIF SUIFvm library. Technical Report, Harvard University, July 2002.
12. Honeywell Inc., http://www.htc.honeywell.com/projects/acsbench, 2005.
13. IAR Embedded Workbench, IAR Systems Inc., www.iar.com, 2005.
14. IEEE 802.11a Wireless LAN standards, http://grouper.ieee.org/groups/802/11/, 2005.
15. JPEG image standard, www.jpeg.org, 2005.
16. MachineSUIF, http://www.eecs.harvard.edu/hube/research/machsuif.html, 2005.
17. M. Mercaldi, M. D. Smith, and G. Holloway. The HALT library. *Technical Report*, Harvard University, July 2002.
18. MIPS Corp., www.mips.com, 2005.
19. SimpleScalar LLC, www.simplescalar.com, 2005.
20. G. Stitt and F. Vahid. Energy advantages of microprocessors platforms with on-chip configurable logic. *IEEE Design & Test of Computers*, 19(6):36–43, 2002.
21. G. Stitt, F. Vahid, and S. Nematbakhsh. Energy savings and speedups from partitioning critical software loops to hardware in embedded systems. *ACM Trans. on Embedded Computing Systems* (TECS), 3(1):218–232, 2004.
22. P. Strobach, Qsdpcm—A new technique in scene adaptive coding. In *Proc. of 4th European Signal Processing Conf.* (EUSIPCO-88), Grenoble, France, pp. 1141–1144, Sept. 1988.
23. SUIF2 compiler, http://suif.stanford.edu/suif/suif2/index.html, 2005.
24. Synplify Pro, Synplicity Inc., www.synplicity.com, 2005.
25. Texas Instruments Inc., www.ti.com, 2005.
26. Triscend Corp. www.triscend.com, 2004.
27. J. Villareal, D. Suresh, G. Stitt, F. Vahid, and W. Najjar. Improving Software Performance with Configurable Logic. In *Design Automation for Embedded Systems*, Springer, Vol. 7, pp. 325–339, 2002.
28. Virtex devices, Xilinx Inc., www.xilinx.com, 2005.
29. A. Ye, N. Shenoy, and P. Baneijee. A compiler for a processor with a reconfigurable functional unit. In *Proc. of FPGA*, pp. 95–100, 2000.