Mapping DSP Applications on Processor Systems with Coarse-Grain Reconfigurable Hardware

Michalis D. Galanis[1], Gregory Dimitroulakos[2], and Costas E. Goutis[3]
VLSI Design Laboratory, Electrical and Computer Engineering Department, University of Patras, Greece
{[1]mgalanis, [2]dhmhgre, [3]goutis}@ee.upatras.gr

Abstract

In this paper, we present performance results from mapping five real-world DSP applications on an embedded system-on-chip that incorporates coarse-grain reconfigurable logic with an instruction-set processor. The reconfigurable logic is realized by a 2-Dimensional Array of Processing Elements. A mapping flow for improving application's performance by accelerating critical software parts, called kernels, on the Coarse-Grain Reconfigurable Array is proposed. Profiling is performed for detecting critical kernel code. For mapping the detected kernels on the reconfigurable logic a priority-based mapping algorithm has been developed. The experiments for three different instances of a generic system show that the speedup from executing kernels on the Reconfigurable Array ranges from 9.9 to 151.1, with an average value of 54.1, relative to the kernels' execution on the processor. Important overall application speedups, due to the kernels' acceleration, have been reported for the five applications. These overall performance improvements range from 1.3 to 3.7, with an average value of 2.3, relative to an all-software execution.

1. Introduction

Reconfigurable architectures have received growing interest in the past few years [1]. Reconfigurable systems represent an intermediate approach between Application Specific Integrated Circuits (ASICs) and general-purpose processors. Such systems usually combine reconfigurable hardware with one or more software programmable processors. Reconfigurable processors have been widely associated with Field Programmable Gate Array (FPGA)-based systems. An FPGA consists of a matrix of programmable logic cells, executing bit-level operations, with a grid of interconnect lines running among them. However FPGAs are not the only type of reconfigurable logic. Several coarse-grain reconfigurable architectures have been introduced and successfully built [1], [2], [3], [4], [5], [6], [7], [8]. Coarse-grain reconfigurable logic has been mainly proposed for speeding-up loops of multimedia and DSP applications in embedded systems. They consist of Processing Elements (PEs) with word-level data bit-widths (like 16-bit ALUs) connected with a reconfigurable interconnect network. Their coarse granularity greatly reduces the delay, area, power consumption and reconfiguration time relative to an FPGA device at the expense of flexibility [1].

In this work, we consider a subclass of coarse-grain architectures where the PEs are organized in a 2-Dimensional (2D) array and they are connected with mesh-like reconfigurable networks [1], [2], [3], [7]. This type of reconfigurable logic is increasingly gaining interest because it is simple to be constructed and it can be scaled up, since more PEs can be added in the mesh-like interconnect. In this paper, these architectures are called Coarse-Grain Reconfigurable Arrays (CGRAs). A variety of CGRA architectures has been presented in both academia [1], [2], [3] and in industry [4], [7], [8].

Recently, design flows for System-on-Chip (SoC) platforms composed by a processor and FPGA [9], [10] found that when critical parts of the application, called *kernels*, are moved for execution on the FPGA the performance is improved. This is due to the fact that most embedded DSP and multimedia applications spend the majority of their execution time in few small code segments (typically loops), the kernels. This implies that an extensive solution search space, as in past hardware/software partitioning works [11], [12] is not a necessity.

A mapping flow for improving the application performance in single-chip systems composed by an instruction-set processor and a CGRA is proposed. Speedups are achieved by accelerating critical software parts (kernels) on the CGRA. The processor executes the non-critical software parts. Mapping flows for processor-FPGA systems [9], [10] showed that such type of partitioning is feasible in embedded systems and it leads in important speedups. Processor-CGRA systems are present in both academia [2], [3], and in industry [4], [5], [7]. These SoCs is expected to further gain importance since the CGRAs lead to smaller execution times and lower power consumption of critical software parts when

This work was partially funded by the Alexander S. Onassis Public Benefit Foundation

compared with FPGAs. Thus, a mapping methodology like the one presented in this paper, is considered as a prerequisite for improving the performance of applications in such embedded systems.

The mapping flow mainly consists of the following steps: (a) profiling for detecting critical kernel code, (b) Intermediate Representation (IR) creation, (c) mapping algorithm for the CGRA architecture, and (d) compilation to the instruction-set processor. We emphasize to the mapping for CGRA architectures, since it considerably affects the performance improvements through the kernels acceleration. The proposed mapping procedure for CGRAs is a priority-based (list-based) algorithm and it targets a CGRA template architecture which can model a variety of existing architectures [2], [3], [7].

The work of [4] describes a design flow for an XPP-based system. Performance results from mapping DSP algorithmic kernels on the XPP array are given. In [6] the instruction-set extension of a RISC processor coupled with a 4x4 XPP coarse-grain reconfigurable array is described. Performance improvements relative to the stand-alone operation of the RISC processor are shown for an 8x8 IDCT. However, in [4] and in [6] the mapping of a complete DSP application is not performed. In [13], it is shown that a hybrid architecture composed by an ARM926EJ-S and a CGRA similar to MorphoSys [3], executes 2.2 times faster a H.263 encoder than a single ARM926EJ-S processor. The design flow for the ADRES architecture was applied to an MPEG-2 decoder in [14]. The kernel and the overall application speedup over an 8-issue VLIW processor were 4.84 and 3.05, respectively.

In this paper, we provide results by applying the proposed mapping flow in five real-life DSP applications, coded in C language, on three instances of a generic processor-CGRA system. A 4x4 array of PEs is used for accelerating critical kernel code, while an ARM processor executes the non-critical code. The applications are: (a) a medical image processing application [15], (b) an IEEE 802.11a OFDM transmitter [16], (c) a wavelet-based image compressor [17], (d) a still-image JPEG encoder, and (e) a video compression technique [18]. The results illustrate that the speedup from executing kernels on the CGRA ranges from 9.9 to 151.1, with an average value of 54.1, relative to the kernels' execution on the instruction-set processor. Furthermore, significant overall application speedups, ranging from 1.26 to 3.70, were achieved relative to an all-processor execution of the application.

The rest of the paper is organized as follows: section 2 presents the system architecture and the mapping flow for this system. Section 3 describes the CGRA architecture template and the mapping algorithm for it. Section 4 presents the experimental results, while section 5 concludes this paper and outlines future research activities.

2. Mapping flow

2.1. System architecture

A generic diagram of the considered hybrid SoC architecture, that targets embedded DSP applications, is shown in Figure 1. The platform includes: (a) Coarse-Grain Reconfigurable Array for executing kernels, (b) shared system data memory, and (c) an instruction-set embedded processor. The processor is typically a RISC processor, like an ARM7 [19].

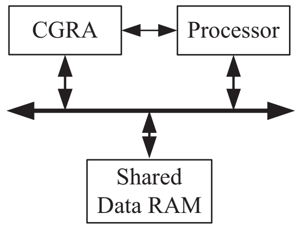

Figure 1. Generic hybrid SoC architecture.

Communication between the CGRA and the processor takes place via the shared data RAM and several direct signals. Part of the direct signals is used by the processor for controlling the CGRA by writing values to memory-mapped registers located in the CGRA. Also, direct signals are used by the CGRA for informing the processor. For example, an interrupt signal is typically present which notifies the processor that the execution of a critical software part finished on the CGRA. Local data and configuration memory exist in the CGRA, for quickly loading data and configurations, respectively. This generic system architecture can model a variety of existing processor-CGRA SoCs, like the ones considered in [2], [3], [5], [7].

2.2. Flow description

The proposed mapping flow for processor-CGRA systems interests in increasing application's performance by mapping critical software parts on the coarse-grain reconfigurable hardware. This flow takes advantage of the fact that kernels of DSP and multimedia applications contribute the most to the execution time.

The mapping flow is illustrated in Figure 2. The input is an application described in a high-level language, like ANSI C/C++. Firstly, profiling is performed in the input source code for identifying the critical code sections, the kernels. For performing profiling, the standard debugger/simulator tools of the development environment of a specific processor can be utilized. For example, for the ARM processors, the instruction-set simulator (ISS) of the ARM RealView Developer Suite (RVDS) [19] is typically used. An instruction-set simulator that targets an extension of the MIPS IV processor [20] is the SimpleScalar toolset [21]. This simulator can be used when this superset of the MIPS IV is coupled with the CGRA in the targeted SoC platform. We consider as kernels those code segments that contribute more than a certain amount to the total application's execution time on the processor. For example, parts of the code that account 10% or more of the application's time can be characterized as kernels.

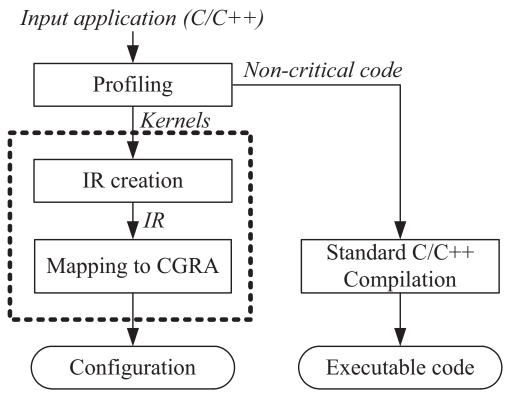

Figure 2. Mapping flow for the processor-CGRA architecture.

The profiling step outputs the kernels and the non-critical code segments. The kernels will be mapped on the CGRA for improving application's performance, while the non-critical code will be executed on the processor. The non-critical segments are compiled using a standard C/C++ compiler for the specific processor. Then, the produced executable code runs on the processor and the execution cycles are calculated using an instruction-set simulator/debugger for the specific processor.

For mapping the critical parts on the CGRA, the Intermediate Representation (IR) of each kernel code segment is created. We have chosen in this work the Control Data Flow Graph (CDFG) model of computation as the IR. The CDFG is a model of computation extensively used in mapping applications on reconfigurable hardware [22]. The CDFG of each kernel is input to our-developed mapping procedure, described in section 3.2, for CGRA architectures. The mapping procedure defines the configuration of the CGRA and reports the clock cycles of the kernels executed on the CGRA.

The communication mechanism used by the processor and the CGRA preserves data coherency by requiring the execution of the processor and the CGRA to be mutually exclusive. The kernels are replaced in the software description with calls to CGRA. When a call to CGRA is reached in the software, the processor activates the CGRA and the proper configuration is loaded on the CGRA for executing the kernel. The data required for the kernel execution are written to the shared data memory by the processor. These data are read by the CGRA. When the CGRA 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 CGRA informs the processor 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 CGRA remains idle.

The mutual exclusive execution simplifies the programming of the system architecture since complicated analysis and synchronization procedures are not required. However, the parallel execution on processor and on the CGRA is a topic of our future research activities.

The total execution cycles after partitioning the application on the processor and the CGRA are:

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

where $Cycles_{proc}$ represents the number of cycles needed for executing the non-critical software parts on the processor, and $Cycles_{CGRA}$ corresponds to the cycles that are required for executing the software kernels on the CGRA. The communication time between the processor and the CGRA is included in the $Cycles_{proc}$ and in the $Cycles_{CGRA}$ since load and store operations that refer to the shared data RAM are present in the non-critical parts and in the kernels of each application. The $Cycles_{CGRA}$ have been normalized to the clock frequency of the microprocessor. 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.

The proposed mapping flow requires the execution times of kernels on the coarse-grain reconfigurable hardware. Since, those times can be also given by other mapping algorithm than the one considered in this work, the proposed flow can be applied in conjunction with other mapping algorithms [22], [23], [24]. Additionally, the flow is parametric to the type of coarse-grain reconfigurable hardware, as the mapping procedures abstract the hardware by typically considering resource constraints, timing and area characteristics. Due to the aforementioned factors, the design flow can be considered retargetable to the type of coarse-grain reconfigurable hardware. Thus, the proposed mapping flow can also take into account other types of coarse-grain reconfigurable hardware [25], and not only CGRAs.

The steps of the IR creation and the mapping to CGRA, enclosed in the dashed line of Figure 2, have been automated for an input software description in C language. In particular, for the CDFG creation from the C code, we have used the SUIF2 [26] and MachineSUIF compiler infrastructures [27]. The mapping algorithm for the CGRA is implemented in C++. In the following section, we describe the CGRA architecture template and the developed mapping algorithm for such types of architectures.

3. Mapping algorithm for CGRAs

3.1. CGRA architecture template

The considered generic CGRA template is based on characteristics found in the majority of existing 2D coarse-grain reconfigurable architectures [1], [2], [3], [7] and it can be used as a model for mapping applications to such type of architectures. The proposed architecture template is shown in Figure 3a. Each PE is connected to its nearest neighbours, while there are cases [3], [7] where there are also direct connections among all the PEs across a column and a row. A PE typically contains one Functional Unit (FU), which it can be configured to perform a specific word-level operation each time. Characteristic operations supported by the FU are ALU, multiplication, and shifts. For storing intermediate values between computations and data fetched from memory, a small local data RAM exists inside a PE. Figure 3b shows an example of a PE architecture. The FU of this PE has two inputs and one output. The multiplexers are used to select each input operand that can come from different sources: (a) from the same PE's RAM, (b) from the memory buses and (c) from another PE. The output of each FU can be routed to other PEs or to its local RAM. The reconfiguration (context) register of a PE stores control values (context word) that determine how the FU, the local RAM and the multiplexers are configured. Also, this context word determines where the output of the FU is routed, thus defining the interconnections among the PEs.

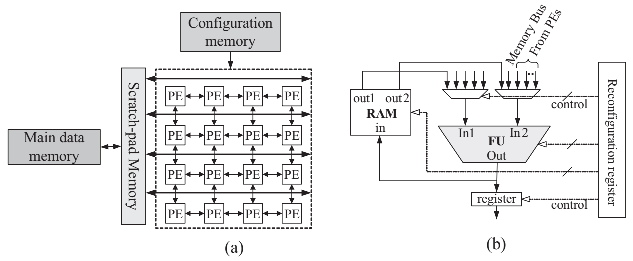

Figure 3. (a) CGRA architecture template, (b) Example of PE architecture.

The main configuration memory of the CGRA (Figure 3a) stores the whole configuration for setting up the CGRA for the execution of application's kernels. Configuration caches distributed in the CGRA and reconfiguration registers inside the PEs are used for the fast reconfiguration of the CGRA. A configuration cache stores a few contexts locally, which can be loaded on cycle-by-cycle basis. The configuration contexts can also be loaded from the configuration memory at the cost of extra delay, if the local configuration caches are not large enough to store the configuration of the kernel body.

The CGRA's data memory interface consists of: (a) the memory buses, (b) the scratch-pad memory [28] which is the first level (L1) of the CGRA's memory hierarchy, and (c) the base memory level, called L0, which is formed by the local RAMs inside the PEs. The main data memory of the CGRA is a part of the system's shared data memory (Figure 1). The PEs residing in a row or column share a common bus connection to the scratch-pad memory, as in [2], [3], [7]. The L1 serves as a local RAM memory for quickly loading data in the PEs of the CGRA. The interconnection network together with the L0 acts as a high-bandwidth foreground memory, since during each cycle several data transfers can take place through different paths in the CGRA.

We note that the organization of the PEs and their interface to the data memory largely resembles the popular MorphoSys reconfigurable array [3]. However, with little modifications it can model other CGRA architectures. For example, if we allow only the PEs of the first row of the CGRA to be connected to the scratch-pad memory through load/store units then our template can model the data memory interface of the CGRA in [14].

3.2. Algorithm description

The task of mapping applications to CGRAs is a combination of scheduling operations for execution [29], mapping these operations to particular PEs, and routing data through specific interconnects in the CGRA. The first input to the mapping algorithm is a DFG $G(V, E)$ that represents the kernel (critical basic block) which is to be mapped to the CGRA. The algorithm is applied to all the application's kernels, one at a time, for computing the execution cycles on the CGRA. The description of the CGRA architecture is the second input to the mapping process. The CGRA architecture is modelled by a undirected graph, called CGRA Graph, $G_A(V_p, E_I)$. The V_p is the set of PEs of the CGRA and E_I are the interconnections among them. The CGRA architecture description includes parameters, like the number of the PEs, the size of the local RAM inside a PE, the memory buses to which each PE is connected, the bus bandwidth and the scratch-pad memory access times.

The PE selection for scheduling an operation, and the way the input operands are fetched to the specific PE, will be referred to hereafter as a Place Decision (*PD*) for that specific operation. Each PD has a different impact on the operation's execution time and on the execution of future scheduled operations. For this reason, a cost is assigned to each PD to incorporate the factors that influence the scheduling of the operations. The goal of the mapping algorithm is to find a cost-effective PD for each operation. The proposed priority (list) based mapping algorithm is shown in Figure 4.

The algorithm is initialized by assigning to each DFG node a value that represents its priority. The priority of an operation is calculated as the difference of its As Late As

Possible (ALAP) minus its As Soon As Possible (ASAP) value. This result is called *mobility*. Also variable *p*, which indirectly points each time to the most exigent operations, is initialized by the minimum value of mobility. In this way, operations residing in the critical path are considered first in the scheduling phase. During the scheduling phase, in each iteration of the while loop, *QOP* queue takes via the *ROP()* function the ready to be executed operations which have a value of mobility less than or equal to the value of variable *p*. The first *do-while* loop schedules and routes each operation contained in the *QOP* queue one at a time, until it becomes empty. Then, the new ready to be executed operations are considered via *ROP()* function which updates the *QOP* queue.

```
// SOP        : Set with operations to be scheduled
// G(V,E)     : Kernel's DFG
// QOP        : Queue with ready to schedule operations
SOP = V;
AssignPriorities(G);
p = Minimum_Value_Of_Mobility; // Highest priority
while (SOP ≠ ø) {
  QOP = queue ROP(p);
  do {
   Op = dequeue QOP;
   (Pred_PEs, RTime) = Predecessors(Op);
   do {
      Choices = GetCosts(Pred_PEs, RTime);
      RTime++;
   } while( ResourceCongestion(Choices) );
   Decision =
            DecideWhereToScheduleTimePlace(Choices);
   ReserveResources(Decision);
   Schedule(Op);
   SOP = SOP – Op;
 } while(QOP ≠ ø);
 p = p+1;
}
```

Figure 4. CGRA mapping algorithm.

The *Predecessors()* function returns (if exist) the PEs where the operation's *Op*'s predecessors (*Pred_PEs*) were scheduled and the earliest time (*RTime*) at which the operation *Op* can be scheduled. The *RTime* (eq. (2)) equals to the maximum of the times where each of the *Op*'s predecessors finished executing t_{fin}. *P* is the set having the predecessor operations of *Op*.

$$RTime(Op) = \max_{i=1,..,|P(Op)|}\left(t_{fin}(Op_i), 0\right) \qquad (2)$$

where $Op_i \in P(Op)$. The function *GetCosts()* returns the possible PDs and the corresponding costs for the operation *Op* in the CGRA in terms of the *Choices* variable. It takes as inputs the earliest possible schedule time (*RTime*) for the operation *Op* along with the PEs where the *Pred_PEs* have been scheduled. The function *ResourceCongestion()* returns true if there are no available PDs due to resource constraints. In that case *RTime* is incremented and the *GetCosts()* function is repeated until available PDs are found. The *DecideWhereToScheduleTimePlace()* function analyzes the mapping costs from the *Choices* variable. The function firstly identifies the subset of PDs with minimum delay cost. From the resulting PD subset, it selects the one with minimum interconnection cost as the one which will be adopted. The function *ReserveResources()* reserves the resources (memory bus, PEs, local RAMs and interconnections) for executing the current operation on the selected PE. More specifically, the PEs are reserved as long as the execution takes place. For each data transfer, the amount and the duration of bus reservation is determined by the number of the words transferred and the memory latency, respectively. The local RAM in each of the PEs is reserved according to the lifetime of the variables [29]. Finally, the *Schedule()* records the scheduling of operation *Op*. After all operations are scheduled, the execution cycles of the input kernel are reported.

4. Experiments

4.1. Set-up

Five real-life DSP applications, written in C language, were mapped on three different instances of the generic processor-CGRA platform using the developed mapping flow. These applications are: (a) a cavity detector which is a medical image processing application [15], (b) the baseband processing of an IEEE 802.11a OFDM transmitter [16], (c) a wavelet-based image compressor, public available at [17], (d) a still-image JPEG encoder, and (e) a video compression technique, called Quadtree Structured Difference Pulse Code Modulation (QSDPCM) [18]. The experiments were performed using the following applications' 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.

We have used three different architectures of 32-bit ARM processors [19], which are RISC processors widely used in embedded systems. These processors are: (a) an ARM7 clocked at 100 MHz, (b) an ARM9 clocked at 250 MHz, and (c) an ARM10 having clock frequency of 325 MHz. These clock frequencies were taken from reference designs from the ARM website [19] and they are considered as typical for these processors. The five applications were compiled to generate binary files for the ARM processors using the highest level of software optimizations. The ARM RVDS (version 2.2) [19] was used for calculating the execution cycles of applications' parts for each one of the three processors. The instruction-set simulator of the RVDS was used for profiling the application's C source code for detecting kernel code segments. In this work, kernels are considered those code sections that contribute 10% or more to the application's execution time.

The CGRA architecture used in this experimentation and coupled each time with one of the three processors is a 4x4 array of PEs. The PEs are directly connected to all other PEs in the same row and same column through vertical and horizontal interconnections, as in a quadrant of MorphoSys [3]. There is one 16-bit FU in each PE that can execute any supported operation (i.e. ALU, multiplication, shift) in one CGRA's clock cycle. Each PE has a local RAM of size 8 words; thus the L0 size is 256 bytes. The direct connection delay among the PEs is zero cycles. Two buses per row are dedicated for transferring data to the PEs from the scratch-pad (L1) memory. The delay of fetching one word from the scratch-pad memory is one cycle. We assume that the CGRA configuration caches are sufficiently large to store the configuration of the applications' kernels to be mapped on the CGRA. In this case, cycle-by-cycle reconfiguration of the CGRA is supported. The CGRA's clock frequency is set to 150 MHz as in the reconfigurable array of [4].

4.2. Results

We have profiled the five DSP applications and we have detected their kernels for each one of the three ARM processors. For all applications the detected kernels were loops that they consist of word-level operations (ALU, multiplications, shifts) that match the granularity (data bit-width) of the PEs in the 4x4 CGRA. In all applications, except from the QSDPCM, the number of kernels (loops) in each application equals 4. For the QSDPCM, three loops contributed 10% or more to the total execution time. Thus, the speedup of each application will come from accelerating a small number of kernels. The small number of the detected kernels in each application means that the usage of exploration algorithms, which typically examine thousands of possible partitions and utilize complex algorithms [11], [12] is not necessary in the case of partitioning the considered applications on the processor-CGRA SoCs. We note that the detected kernels of all applications were critical code parts when executed on each one of the three ARM processors.

We have unrolled the detected critical loops 16 times for mapping them on the CGRA. We have investigated that unrolling the kernels of the considered applications more than 16 times, the execution cycles, when these kernels were mapped on the 4x4 CGRA, slightly decrease. Thus, we have selected the unroll factor equal to 16 since it gives significant reductions of the execution cycles of kernels over the execution of the original loop body on the 4x4 CGRA.

Figure 5 shows the speedups for executing all the kernels of each application on the 4x4 CGRA relative to the execution of the kernels on the processor. For every application, the speedup is relative to each one of the three ARM processor used. For example, the left most bar in each application corresponds to the performance improvement obtained when the execution cycles of the kernels are compared to the ones for the execution of the kernels on the ARM7. The kernel speedup is defined as:

$$Sp_{kernel} = Cycles_{kernels\_sw} / Cycles_{kernels\_CGRA\_norm} \quad (3)$$

where $Cycles_{kernels\_sw}$ represents the number of cycles required for executing the kernels on the processor and the $Cycles_{kernels\_CGRA\_norm}$ represents the number of cycles for executing the kernels on the CGRA. We note that the cycles reported from the CGRA mapping algorithm described in section 3.2, are normalized to the clock frequency of the processor in the system platform, using the following relation:

$$Cycles_{kernels\_CGRA\_norm} = Cycles_{kernels\_CGRA} \cdot \frac{Clock_{proc}}{Clock_{CGRA}} \quad (4)$$

where the $Cycles_{kernels\_CGRA}$ are the clock cycles reported from the developed mapping tool for CGRAs, $Clock_{proc}$ is the clock frequency of the processor and $Clock_{CGRA}$ is the clock frequency of the CGRA.

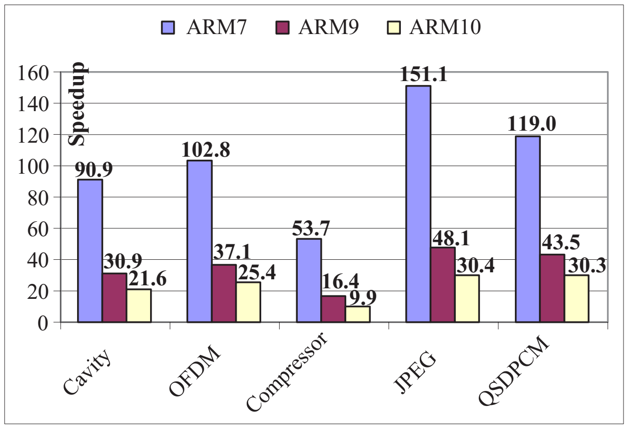

Figure 5. Kernel speedups on the 4x4 CGRA for various processor systems.

From Figure 5, it is deduced that important speedups are achieved when critical kernels are executed on the CGRA. The performance improvements range from 9.9 to 151.1, with an average value of 54.1 for all the applications and all the cases of ARM processors. Even in the case where the processors are clocked in a higher clock frequency than the CGRA (as in the ARM9 and ARM10 SoCs) the speedups are significant. The speedups are due to the fact that the inherent operation parallelism of the kernels is better exploited by the available Processing Elements of the CGRA than the functional units of the ARM processors. These results prove that the CGRA architectures are efficient in accelerating critical loops of DSP and multimedia applications which leads in improving the overall performance of an application executed on a processor-CGRA system as it will be shown in Table 1.

For the ARM7 system the average kernel speedup for the five applications is 103.5, for the ARM9 system is 35.2, and for the ARM10 system is 23.5. From Figure 5 it is noted that the kernel speedup decreases when a newer-generation and higher-clocked instruction-set processor is used in the platform. The largest speedup is obtained relative to the ARM7 solution, which is the oldest-generation of the ARM processors used in this work and the lowest-clocked one. The speedup relative to the kernel execution on the ARM9 is approximately 3 times in average smaller than the ARM7 one. Furthermore, the kernel speedup decreases more slowly when the ARM10 is used in the platform. In this case, the speedup relative to the kernel execution on the ARM10 is approximately 1.5 times in average smaller than the ARM9 one. This is justified by the facts that the clock difference of the ARM9 and ARM10 is smaller than the ARM7 and ARM9 and that the ARM9 is a more contemporary microprocessor generation to the ARM10 than the ARM7.

Table 1. Execution cycles and speedups for the processor-CGRA SoCs

Application	Proc. Type	Cycles$_{init}$	Cycles$_{hw/sw}$	Speedup
Cavity	ARM7	178,828,950	87,512,166	2.04
	ARM9	161,441,889	85,566,541	1.89
	ARM10	155,356,758	87,255,890	1.78
OFDM	ARM7	397,851	121,873	3.26
	ARM9	362,990	118,197	3.07
	ARM10	334,375	119,455	2.80
Compressor	ARM7	25,832,508	11,564,753	2.23
	ARM9	20,574,658	10,135,735	2.03
	ARM10	17,854,928	10,013,173	1.78
JPEG	ARM7	23,003,868	6,212,160	3.70
	ARM9	19,951,193	6,785,540	2.94
	ARM10	16,930,629	6,254,858	2.71
QSDPCM	ARM7	4,026,384,618	3,075,311,802	1.31
	ARM9	3,895,248,922	3,039,239,650	1.28
	ARM10	3,608,029,180	2,840,231,680	1.27
Average:				**2.27**

The execution cycles and the performance results from applying the proposed mapping flow in the five applications are presented in Table 1. For every application, each one of the three considered ARM processor types (*Proc. Type*) is used for estimating the clock cycles ($Cycles_{init}$) required from executing the whole application on the processor. The application speedup is calculated as:

$$Sp_{app} = Cycles_{init} / Cycles_{hw/sw} \quad (5)$$

where $Cycles_{hw/sw}$ represents the execution cycles after the partitioning and the mapping of the kernels on the CGRA and the non-critical code on the processor.

From the results given in Table 1, it is evident that significant performance improvements are achieved when critical software parts are mapped on the 4x4 CGRA. The application speedup for the five applications and for the processor used ranges from 1.27 to 3.70, with an average value of 2.27. Such amounts of speedups were also considered as important in previous works as in [13], where a video encoder executed 2.2 times faster on a processor-CGRA SoC than an all-software solution. It is noticed from Table 1 that the largest overall application performance gains are achieved for the ARM7 system, fact that is explained by the obtained kernel speedups illustrated in Figure 5 which were the largest ones among the three ARM-based systems. The average application speedup of the five DSP benchmarks for the ARM7 system is 2.51, for the ARM9 is 2.24, while for the ARM10 system is 2.07. Thus, even when the 4x4 CGRA is coupled with a modern embedded processor, like the ARM10, which is clocked at a higher clock frequency (approximately two times larger), the overall application speedup is significant.

Table 2. Exploration of the speedup relative to the clock frequency of the CGRA

Application	Proc. Type	Speedup (100 MHz)	Speedup (150 MHz)
Cavity	ARM7	2.03	2.04
	ARM9	1.86	1.89
	ARM10	1.75	1.78
OFDM	ARM7	3.23	3.26
	ARM9	2.99	3.07
	ARM10	2.70	2.80
Compressor	ARM7	2.21	2.23
	ARM9	1.96	2.03
	ARM10	1.71	1.78
JPEG	ARM7	3.67	3.70
	ARM9	2.88	2.94
	ARM10	2.63	2.71
QSDPCM	ARM7	1.31	1.31
	ARM9	1.28	1.28
	ARM10	1.26	1.27
Average:		**2.23**	**2.27**

We mapped the five applications in the three SoCs where the 4x4 CGRA is now clocked at 100 MHz, instead of 150 MHz as in the previous results. The clock frequency of the CGRA in the MorphoSys SoC [3] was also 100 MHz. The three considered ARM processors have similar clock frequencies as in the previous experiments. In Table 2, the application speedups for these two different clock frequencies of the CGRAs are given. From these results it is deduced that the speedup slightly decreases when the clock frequency of the CGRA becomes smaller. The average speedup for the five applications and for the three ARM-based systems is 2.23 for the clock of 100 MHz, while for the 150 MHz clock the average speedup is slightly larger since it is equal to 2.27. Thus, we can achieve somewhat similar speedups if the CGRA is clocked at a smaller frequency. In this case, the system's energy consumption is expected to be reduced.

5. Conclusions - Future work

A mapping flow for improving system performance by executing critical kernel code on the coarse-grain reconfigurable hardware of a processor-based SoC was presented. Results from mapping five DSP applications on three instances of a processor-CGRA platform show that the CGRAs are efficient in accelerating kernel code since the average kernel speedup was 54.1. This resulted in important overall performance improvements that ranged from 1.3 to 3.7. Future work focuses on exploiting the possible performance improvements of parallel execution of the processor and the CGRA.

References

[1] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective", in *Proc. of ACM/IEEE DATE '01*, pp. 642-649, 2001.

[2] T. Miyamori and K. Olukutun, "REMARC: Reconfigurable Multimedia Array Coprocessor", in *IEICE Trans. On Information and Systems*, pp. 389-397, 1999.

[3] H. Singh, L. Ming-Hau, L. Guangming, F.J. Kurdahi, N. Bagherzadeh, E.M. Chaves Filho, "MorphoSys: An Integrated Reconfigurable System for Data-Parallel and Communication-Intensive Applications", in *IEEE Trans. on Computers*, vol. 49, no. 5, pp. 465-481, May 2000.

[4] V. Baumgarte, G. Ehlers, F. May, A. Nuckel, M. Vorbach, M. Weinhardt, "PACT XPP - A Self-Reconfigurable Data Processing Architecture", in *the Journal of Supercomputing*, Springer, vol. 26, no. 2, pp. 167-184, September 2003.

[5] J. Becker, M. Vorbach, "Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC)", in *Proc. of Workshop VLSI* (WVLSI '03), IEEE Press, pp. 107-112, 2003.

[6] J. Becker, A. Thomas, "Scalable Processor Instruction Set Extension", in *IEEE Design & Test of Computers*, vol. 22, no. 2, pp. 136-148, 2005.

[7] Morpho Technologies, www.morphotech.com, 2005.

[8] D-Fabrix array, Elixent Ltd., www.elixent.com, 2005.

[9] 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.

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

[11] 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.

[12] 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, 1998.

[13] Y. Kim, C. Park, S. Kang, H. Song, J. Jung, K. Choi, "Design and Evaluation of a Coarse-Grained Reconfigurable Architecture", in *Proc. of International SoC Design Conference* (ISOCC '04), pp. 227-230, 2004.

[14] B. Mei, S. Vernalde, D. Verkest, R. Lauwereins, "Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture, A Case Study", in *Proc. of ACM/IEEE DATE '04*, pp. 1224-1229, 2004.

[15] M. Bister, Y. Taeymans, J. Cornelis, "Automatic Segmentation of Cardiac MR Images", *Computers in Cardiology*, IEEE Computer Society Press, pp.215-218, 1989.

[16] IEEE 802.11a Wireless LAN standard, http://grouper.ieee.org/groups/802/11/, 2005.

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

[18] P. Strobach, "Qsdpcm - A New Technique in Scene Adaptive Coding", in *Proc. of 4th European Signal Processing Conf.*, Grenoble, France, pp. 1141-1144, Sep. 1988.

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

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

[21] SimpleScalar LLC, www.simplescalar.com, 2005.

[22] N. Bansal, S. Gupta, N. Dutt, A. Nikolau, R. Gupta, "Network Topology Exploration of Mesh-Based Coarse-grain Reconfigurable Architectures", in *Proc. of ACM/IEEE DATE '04*, pp. 474-479, 2004.

[23] B. Mei, S. Vernalde, D. Verkest, H. De Man, R. Lauwereins, "Exploiting Loop-Level Parallelism on Coarse-grained Reconfigurable Architectures Using Modulo Scheduling", in *Proc. of ACM/IEEE DATE '03*, pp. 255-261, 2003.

[24] J. Lee, K. Choi, N. D. Dutt, "Compilation Approach for Coarse-grained Reconfigurable Architectures", in *IEEE Design & Test of Computers*, vol. 20, no. 1, pp. 26-33, Jan.-Feb., 2003.

[25] H. Schmit, D. Whelihan, A. Tsai, M. Moe, B. Levine, and R. R. Taylor, "PipeRench: A virtualized programmable datapath in 0.18 micron technology", in *Proc. of IEEE Custom Integrated Circuits Conference*, pp. 63-66, 2002.

[26] SUIF2 compiler infrastucture, http://suif.stanford.edu/suif/suif2/index.html, 2005.

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

[28] P. R. Panda, N. Dutt, and A. Nicolau, *Memory Issues in Embedded Systems-on-Chip: Optimizations and Exploration*, Springer, 1999.

[29] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.