International Journal of Electrical, Electronic and Communication Sciences
ISSN: 2517-9438
Vol:1, No:6, 2007

Performance Improvements of DSP Applications on a Generic Reconfigurable Platform

Michalis D. Galanis, Gregory Dimitroulakos, and Costas E. Goutis

***Abstract*—**Speedups from mapping four real-life DSP applications on an embedded system-on-chip that couples coarse-grained reconfigurable logic with an instruction-set processor are presented. The reconfigurable logic is realized by a 2-Dimensional Array of Processing Elements. A design flow for improving application's performance is proposed. Critical software parts, called kernels, are accelerated on the Coarse-Grained Reconfigurable Array. The kernels are detected by profiling the source code. For mapping the detected kernels on the reconfigurable logic a priority-based mapping algorithm has been developed. Two 4x4 array architectures, which differ in their interconnection structure among the Processing Elements, are considered. The experiments for eight different instances of a generic system show that important overall application speedups have been reported for the four applications. The performance improvements range from 1.86 to 3.67, with an average value of 2.53, compared with an all-software execution. These speedups are quite close to the maximum theoretical speedups imposed by Amdahl's law.

***Keywords*—**Reconfigurable computing, Coarse-grained reconfigurable array, Embedded systems, DSP, Performance

I. 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 a microprocessor. Reconfigurable hardware is used to realize computational intensive parts of the applications, called *kernels,* and allows for the adaptation to the specific requirements of an application. The microprocessor executes non-critical sequential code parts and provides software programmability. The spatial parallelism present in kernels is exploited by the abundant Processing Elements (PEs) of the reconfigurable hardware, resulting in performance improvements.

Reconfigurable logic has 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-grained reconfigurable architectures have been introduced and successfully built [1]-[9]. Coarse-grained 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. The coarse-grained PEs exploit better than the FPGAs the word-level parallelism of many DSP applications. The FPGAs are more effective in realizing bit-level operations. The more regular structures within the PEs with their wider data bit-widths and the regularity of the interconnect network between the PEs, greatly reduces the execution time, area, power consumption and reconfiguration time relative to an FPGA device at the expense of flexibility [1].

In this work, we consider the most widespread subclass of coarse-grained architectures where the PEs are organized in a 2-Dimensional (2D) array and they are connected with mesh-like reconfigurable networks [1], [2], [3], [4], [5]. 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-Grained Reconfigurable Arrays (CGRAs).

Performance results from mapping four real-world DSP applications, coded in C language, on eight instances of a microprocessor/CGRA system template are presented. A 4x4 array of PEs is used for accelerating time critical kernel code, while either an ARM or a MIPS processor executes the non-critical code. Two different 4x4 CGRA architectures are considered that employ different connectivity among the PEs. An automated design flow for the microprocessor/CGRA is proposed that mainly consists of the following steps: (a) a procedure 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 the mapping for CGRA architectures which is the core of the design flow. The proposed mapping procedure for CGRAs is a priority (list)-based algorithm targeting a flexible CGRA template architecture that allows exploration in respect to its characteristics. The study in respect to the system architecture parameters show that the overall application speedup for all the applications and all the platform instances ranges from 1.86 to 3.67 relative to an all-microprocessor solution. Additionally, the experimental results show that the

Manuscript received April 11, 2006. This work was supported in part by the Alexander S. Onassis Public Benefit Foundation.

Michalis D. Galanis, Gregory Dimitroulakos and Costas E. Goutis are with the VLSI Design Lab., ECE Dept., University of Patras, Rio, Greece. (phone: +30 2610 997324, fax: +30 2610 994798, e-mail: mgalanis@ee.upatras.gr)

performance of kernels slightly increases when a richer interconnection among the PEs is used in the CGRA.

The rest of the paper is organized as follows: section II outlines the previous research activities. The reconfigurable system architecture is presented in section III focusing on the description of the flexible CGRA template. The proposed design flow and the mapping algorithm for CGRA are described in section IV. Section V presents the experimental results, while section VI concludes this paper.

II. Related work

Due to the lack of flexible architecture templates and automated mapping flows, there have been few works on thorough study on coarse-grained reconfigurable architectures. In [1] it was indicated that many coarse-grained reconfigurable architectures (RAW [2], Remarc [3], MorphoSys [4]) have fairly predefined architectures that do not allow a straightforward exploration of different architectural features. The Reconfigurable Pipelined Datapath (RaPiD) [9] architecture provides a flexible linear array of PEs that it is programmed using a C-like language called RaPiD-C. The programmer is responsible for scheduling the computation on a cycle-by-cycle basis by describing when and where each operation is performed in the data-path. Thus, a considerable knowledge about the underlying architecture is required fact that complicates the systematic exploration. The PipeRench [8] contains a set of physical pipeline stages, called stripes. Each stripe has an interconnection network and a set of PEs. Although the PipeRench compiler is parameterized in respect to the architecture, its source language is a dataflow intermediate language (DIL) that it is not easy to be integrated in a high-level design environment (like a C-based one) for complete application exploration.

Bansal et al. [10] surveys the performance of a list-based algorithm for CGRA architectures by introducing three different cost functions. The mapping algorithm assumes that there is enough memory bandwidth to fetch data without any delay and there are enough registers to store all the intermediate and final results. However, in CGRAs [3], [4], [5], [6] there is a limited number of storage elements inside the PEs and the memory bandwidth, although high as in [4], is limited. Additionally, only kernels were mapped on the CGRA and not complete applications since the CGRA is not coupled with a software programmable processor that would have executed non-critical application parts. In [11] a generic CGRA template, called Dynamically Reconfigurable ALU Array (DRAA), is considered. The DRAA consists of identical PEs in a 2D array with a regular interconnection network between them. Vertical and horizontal lines provide the interconnections between the PEs of the same line, as well as the access to the main memory. Results from mapping only loops on an 8x8 DRAA architecture were presented. Nevertheless, there was not any exploration performed to illustrate how the performance is influenced by the architecture's characteristics.

In the following, works that consider the mapping of complete applications on CGRAs coupled with a microprocessor are overviewed. The compilation framework of [12] achieved the acceleration of a wavelet compression and Prewitt detection on the MorphoSys architecture over the execution on a Pentium III machine. In [13], it is shown that a hybrid architecture composed by an ARM926EJ-S and an 8x8 Reconfigurable Array similar to MorphoSys [4], executes 2.2 times faster a H.263 encoder than a single ARM926EJ-S processor. The mapping flow for the ADRES architecture was applied to an MPEG-2 decoder in [14]. The the overall application speedup over an eight-issue VLIW processor was 3.05. In [15], an H.264/AVC decoder was mapped on an 8x8 array achieving application speedup of 1.88.

III. System architecture

Fig. 1 shows an overview of the reconfigurable system-on-chip (SoC) architecture considered in this work. The platform is composed by a Coarse-Grained Reconfigurable Array (CGRA), global system RAM, and an embedded microprocessor. The reconfigurable logic includes a 2D array of PEs, a scratch-pad memory (SPM) serving as a local data RAM for quickly loading data in the PEs, a control unit and a set of memory-mapped registers for exchanging data with the microprocessor. The control unit manages the execution of the CGRA by generating a central configuration pointer every cycle for controlling the operation of the PEs, a process similar to the one in [3]. The microprocessor sets the control unit at the beginning of the kernel execution on the CGRA. Thus, the microprocessor does not control the execution of the CGRA every cycle which would have caused high overhead in the microprocessor and degradation of the system's performance. The system RAM stores data, instructions for the microprocessor execution and configurations for the CGRA. The CGRA functions as a coprocessor to the microprocessor and accelerates computational intensive software parts of an application by exploiting the Instruction Level Parallelism (ILP) of these parts. The microprocessor, typically a RISC one like an ARM or MIPS, executes control-dominant sequential parts.

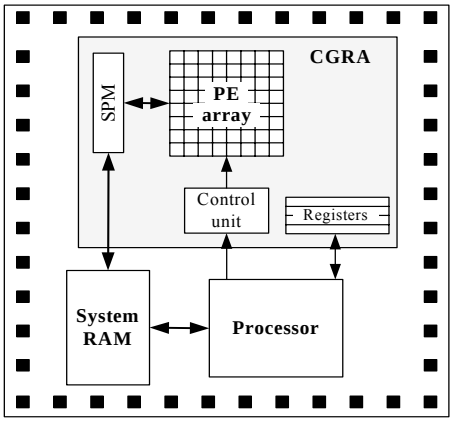

Fig. 1 Overview of the reconfigurable SoC platform

Data communication between the CGRA and the microprocessor uses shared-memory mechanism. The shared memory is comprised by the system RAM and the memory-mapped registers within the CGRA. Local variables, identified by data-flow analysis, are transferred through the shared

registers, while global variables and data arrays are allocated in the system RAM. Both the microprocessor and the CGRA have access to the shared memory. The communication process used by the processor and the CGRA preserves data coherency by requiring the execution of the processor and the CGRA to be mutually exclusive. When a call to CGRA is reached in the software, the microprocessor enables the control unit for setting the execution of the kernel on the CGRA, the proper configuration is loaded on the CGRA and the local variables are transferred to the memory-mapped registers. After the completion of the kernel execution, the CGRA informs the processor and writes the data, required for the execution of the remaining software on the microprocessor, to the shared memory. Then, the execution of the applications is continued on the microprocessor. The mutual exclusive execution simplifies the programming of the reconfigurable system since complicated analysis and synchronization procedures are not required.

A. CGRA template

The generic CGRA template can represent a variety of existing array architectures [1], [4], [5] by being parametric in respect to the number and type of PEs, the interconnections among them and their interface to the data memory. An overview of the proposed CGRA template is shown in Fig. 2a. Each PE is connected to its nearest neighbours (Fig. 2a), while there are cases [4], [5] where there are also direct connections among all the PEs across a column and a row (Fig. 2b).

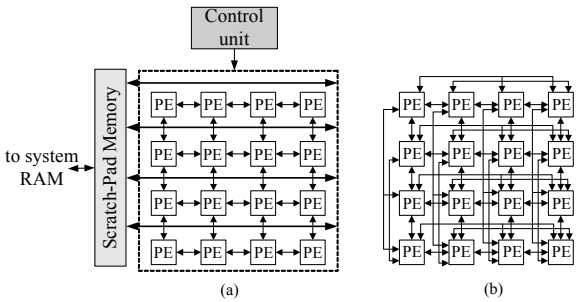

Fig. 2 (a) Outline of a CGRA architecture, (b) Point-to-point connectivity

A PE contains one Reconfigurable Functional Unit (RFU), which it is configured to perform a specific word-level operation each time. Typical operations supported by the RFU are ALU, multiplication, and shifts. Fig. 3 shows an example of a PE architecture. For storing intermediate values between computations and data fetched from the scratch-pad memory, a small local data RAM exists inside a PE. Multiplexers are used to select each input operand that can come from different sources: (a) from the same PE's data RAM, (b) from the memory buses and (c) from another PE. The output of each RFU can be routed to other PEs, using a demultiplexer, or to its local data RAM. Dissimilar to FPGAs, no switch matrices are used for the interconnections. The input multiplexers and the output demultiplexers of the PEs define the interconnections among them.

There is a local configuration (context) RAM inside each PE that stores a few configurations locally. A configuration word controls the type of operation realized by the RFU, the (de)multiplexers, the local data RAM and the output register of the PE acting like an instruction in microprocessors. During the execution of a kernel, the local configuration RAM is indexed by the central configuration pointer, set by the control unit of the CGRA, and a proper context word is loaded allowing dynamic reconfiguration of a PE within a cycle. Each PE can operate differently in a cycle according to the configurations stored in each local configuration RAM.

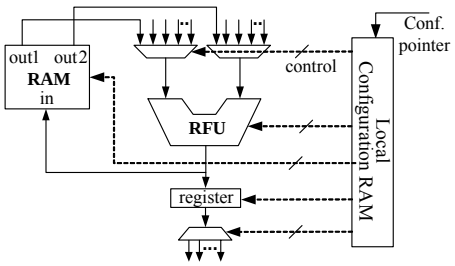

Fig. 3 Example of PE architecture

The PEs residing in a row or column share a common bus connection to the scratch-pad memory, as in [4], [5]. We note that the organization of the PEs and their interface to the scratch-pad memory largely resembles the MorphoSys reconfigurable array [4]. 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].

IV. Design Flow

The proposed design flow for the reconfigurable system architecture improves application's performance by mapping critical software parts on the CGRA. Fig. 4 illustrates the diagram of the design flow. The input is an application described in ANSI C. Initially, a kernel detection procedure, based on profiling, outputs the kernels and the non-critical parts of the source code. For performing profiling, 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) [16] can be used. Kernels are considered loops that contribute more than a certain amount to the total application's execution time on the processor. For example, loop code that accounts 10% or more to the application's time can be characterized as kernel code.

The kernels are moved for execution on the CGRA. The Intermediate Representation (IR) of the kernel source code is created using a compiler front-end. A representation widely used in reconfigurable systems is the Control Data Flow Graph (CDFG). In this work, a hierarchical CDFG [17] is used for modeling data and control-flow dependencies. The control-flow structures, like branches and loops, are modeled through the hierarchy, while the data dependencies are modeled by Data Flow Graphs (DFGs). For generating the CDFG IR from C source code, we have utilized the

SUIF2/MachineSUIF compiler infrastructures [18], [19]. Existing and custom-made compiler passes are used for the CDFG creation.

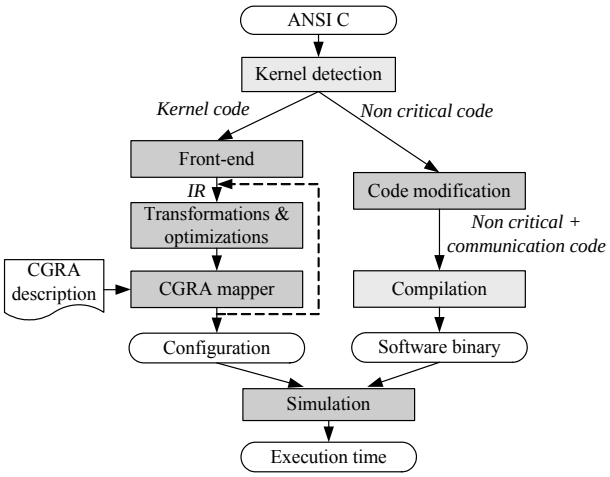

Fig. 4 Design flow for microprocessor/CGRA systems

Then, optimizations and transformations are applied to the kernels for efficient mapping after taking into account the CGRA characteristics, like the number of PEs in the CGRA. Examples of optimizations are dead code elimination, common sub-expression elimination and constant propagation. Transformations typically applied are loop unrolling and loop normalization [20]. Operations inside the kernels that cannot be directly executed on the CGRA PEs are transformed into series of supported operations. The divisions are transformed to shifts, while a square root computation can be performed by the RFUs of the CGRA using a method, like the Friden algorithm [21] that has been implemented in the proposed flow. MachineSUIF compiler passes have been developed for the automatic application of the optimizations and transformations on the kernel's CDFG. The transformed kernels are mapped on the CGRA for improving performance utilizing our algorithm presented in section IV-A, which is the core of the design flow. A prototype tool in C++ has been developed for implementing the mapping algorithm. The second input to the mapping algorithm is a description of the CGRA. The mapping tool outputs the execution cycles and the CGRA configuration. A feedback script is included in the flow for optimizing the performance of the kernels executed on the CGRA.

The non-critical source code is modified to include calls to the reconfigurable logic and to handle the transfer of local variables to and from the CGRA. Then, the source code is compiled using a compiler for the specific processor. The performance is estimated by cycle-accurate simulation having as inputs the inputs the software binary of the processor and the configuration of the CGRA. The dark grey boxes in Fig. 4 represent the procedures modified or created by the authors for the specific flow, while the light grey ones the external tools used.

The time required for executing an application on the reconfigurable system is:

$$Time_{system} = Time_{proc} + Time_{CGRA} \quad (1)$$

where $Time_{proc}$ represents the execution time of the non-critical software parts on the processor, and $Time_{CGRA}$ corresponds to time required for executing the software kernels on the CGRA. The communication time between the processor and the CGRA is included in the $Time_{proc}$ and in the $Time_{CGRA}$.

The proposed flow requires the execution times of kernels on the coarse-grained reconfigurable logic. Since, those times can be also given by other mapping algorithm than the one considered in this work, the design flow can be applied in conjunction with other mapping algorithms for CGRAs [10], [11], [14]. Additionally, it is parametric to the type of coarse-grained reconfigurable hardware, as the mapping procedures abstract the hardware by typically considering resource constraints, timing and area characteristics. Due to the abovementioned factors, the design flow can be considered retargetable to the type of coarse-grained reconfigurable hardware. Thus, the proposed design method can also take into account other types of coarse-grained reconfigurable hardware, like 1-D arrays [9], and not only CGRAs.

A. Mapping algorithm for CGRAs

The task of mapping applications to CGRAs is a combination of scheduling operations for execution, mapping these operations to particular PEs, and routing data through specific interconnects in the CGRA. The mapper considers the resource constraints by monitoring the usage of each of the resources (buses, PEs, interconnections, PE's local data RAMs) in time. The proposed mapping procedure traverses the kernel's CDFG and maps one DFG at a time. The first input to the mapping algorithm is a DFG $G(V, E)$ of the input kernel which is to be mapped to 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 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's 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 pseudocode of the proposed priority (list)-based mapping algorithm is shown in Fig. 5.

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 be executed 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;
}
```

Fig. 5 CGRA mapping algorithm

The *Predecessors()* function returns (if exist) the PEs where the *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*.

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

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 *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 [17]. Finally, the *Schedule()* records the scheduling of operation *Op*. After all operations are scheduled, the execution cycles of the input kernel and the CGRA configuration are reported.

1) Description of the costs

The *Choices* variable includes the delay and the interconnection costs. The delay cost for placing operation *Op* in a specific PE_x refers to the operation's earliest possible execution time there. As shown in eq. (3), the delay is the sum of the *RTime* (which is the earliest possible schedule time) plus the maximum of the times *tf* required to fetch the *Op*'s input operands to PE_x.

$$Choices.\text{dly}(PE_x, Op) = RTime(Op) + \max_{i=1,..,|P|}\left(tf_{P[i]},0\right) \quad (3)$$

When an operand comes from the scratch-pad memory, then *tf* equals the scratch-pad's latency, while when it originates from a CGRA's PE, equals the time for routing the operand to PE_x.

The interconnection overhead refers to the interconnections that must be reserved in order to transfer the operands to the destination PE. As shown in eq. (4), it is the sum of the CGRA interconnections which were used to transfer the predecessor operands. Higher interconnection overhead causes future scheduled operations to have larger execution start time due to conflicts.

$$Choices.\text{intercon}(PE_x, Op) = \sum_{i \in P(Op)} PathLength\left(PE_{Op_i} \rightarrow PE_x\right) \quad (4)$$

A greedy approach was adopted for calculating the time for routing the operands and the number of interconnections (eq.(4)) required for routing an operand. For each operand the shortest paths, which connect the source and destination PE, are identified. From this set of paths, the one with the minimum routing delay is selected. The delay and the length of the selected path gives the delay and interconnection costs through eq.(3) and eq.(4), respectively.

V. Experimental Results

A. Set-up

The four DSP applications, described in C language, used in the experiments are given in Table I. A brief description of each application is shown in the second column, while in the third one the input used is presented.

TABLE I
APPLICATIONS' CHARACTERISTICS

Application	Description	Input
JPEG enc.	Still-image JPEG encoder	256x256 byte image
OFDM trans.	IEEE 802.11a OFDM transmitter	4 payload symbols
Compressor	Wavelet-based image compressor [22]	512x512 byte image
Cavity det.	Medical imaging technique [23]	640x400 byte image

The performance of the applications is estimated via cycle-accurate simulation, using the proposed design flow, in eight

instances of the reconfigurable system. Two CGRA architectures are used each time for accelerating kernels. Both of them consist of 16 PEs organized in a 4x4 array. We list the common features of both CGRAs. Each CGRA's data-width is 16-bits. The RFU in each PE can execute an operation in one CGRA's clock cycle. Each PE has a local data RAM of size of eight 16-bit words, while the local configuration RAM's size is 32 contexts. The size of configuration RAMs is large enough for the considered applications to store the whole configuration for executing a kernel; thus there is no time overhead for loading the configuration RAMs during the kernel execution. Two buses per CGRA row are dedicated for transferring data to the PEs from the scratch-pad memory. The delay of fetching one word from the scratch-pad memory is one cycle. We have described in VHDL both CGRA architectures and we have synthesized them with the Synopsys Design Compiler using a 130nm process. The timing reports showed that the clock frequency for both CGRA architectures can be set to 100MHz.

The two CGRA architectures differ in respect to the interconnection network among the PEs. In the first architecture (*CGRA1*), each PE is connected only to its nearest neighbour PEs in the same row and column as shown in Fig. 2a. In the second one (*CGRA2*), the PEs are directly connected to all other PEs in the same row and same column through vertical and horizontal interconnections as illustrated in Fig. 2b. In the CGRA2, more internal bandwidth than the CGRA1 is available to transfer data among the PEs due to the richer interconnection topology. These two alternatives architectures were chosen to show the effect of the interconnection topology on the performance of the kernels.

We have used four different architectures of 32-bit embedded RISC processors coupled each one of them with the 4x4 CGRA: an ARM7, an ARM9 [16], and two SimpleScalar processors [24]. The SimpleScalar processor is an extension of the MIPS IV core. 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 RealView Developer Suite (RVDS) (version 2.2) was utilized, while the performance for the MIPS-extended processors is estimated using the SimpleScalar simulator tool [24]. Typical clock frequencies, for 130nm CMOS process, are considered for the four processors: the ARM7 runs at 133 MHz, the ARM9 at 250 MHz, and the MIPS processors at 200 MHz. The four applications were compiled to generate binary files for the processors using the highest level of software optimizations.

B. Experimentation

The results from profiling the four applications on the ARM7 processor, using the ARM RVDS tool, are presented in Table II. The applications were compiled to generate binary files for the ARM processors using the highest level of software optimizations. The threshold for the kernel detection was set to 10% of the total execution time of the application. It was observed that a threshold smaller than 10% leads in marginal additional improvements when the identified kernels are mapped on the CGRA. The *Total size* corresponds to the application's static size in terms of instructions bytes, while the *% size* to the percentage of the kernels' contribution to the total static size. The *% time* is the percentage of the execution time spent in the kernels. The *Ideal speedup* is the theoretical maximum speedup, according to Amdahl's law, if the application's kernels were ideally executed on the CGRA in zero time. The ideal speedup equals 100 / (100 - *%time*). The number of the kernels detected in each application is also given. The kernels of the four applications are innermost loops and they consist of word-level operations (ALU, multiplications, shifts) that match the granularity (data bit-width) of the PEs in the CGRA.

From the results of Table II, it is inferred that an average of 7.9% of the code size, representing the kernels' size, contributes 66.2% on average to the total execution time. Furthermore, the average ideal speedup for the ARM7 systems equals 3.10. The geometrical means of the *% size*, *% time* and of the *Ideal speedup* are also given. Thus, it is deduced that important overall application speedups will come from accelerating few small kernels. We mention that the detected loops are also kernels for the rest three microprocessors. For the MIPS-extended processor, the SimpleScalar tool was used for profiling.

TABLE II
KERNEL IDENTIFICATION RESULTS FOR THE ARM7

Application	**Total size (bytes)**	**% size**	**% time**	**Ideal speedup**	**# of kernels**
JPEG enc.	36,592	10.4	74.7	3.96	4
OFDM trans.	15,579	9.0	71.8	3.54	4
Compressor	12,835	4.8	60.2	2.51	4
Cavity det.	12,039	7.4	58.0	2.38	4
Average		**7.9**	**66.2**	**3.10**	
Geo. mean		***7.6***	***65.8***	***3.03***	

We have unrolled the bodies of the detected loops 16 times for mapping them on the two CGRAs. 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 CGRAs, slightly decrease. Thus, we have selected the unroll factor equal to 16 since it gives significant cycles reductions over the execution of the original kernel body on both 4x4 CGRAs.

The performance results from applying the design flow in the four applications are presented in Table III. For every application, each one of the four considered processor architectures (*Proc.*) is used for estimating the execution time ($Time_{sw}$) required from executing the whole application on the processor. The *Ideal speedup* is the maximum theoretical speedup if the kernels are executed on the CGRA in zero time. The estimated speedup (*Speedup*) over the execution of the whole application on the microprocessor is calculated as:

$$Speedup = Time_{sw} / Time_{system} \quad (5)$$

where $Time_{system}$ represents the execution time after accelerating kernels on the CGRA. All execution times are normalized to the software execution times on the ARM7.

From the results given in Table III, it is evident that significant overall performance improvements are achieved when critical software parts are mapped on a 4x4 CGRA. For the systems composed by the CGRA1, the application speedups range from 1.86 to 3.65, with an average speedup for all the processor systems and the four applications, of 2.53. Furthermore, the average application speedups for each processor system are: 2.77 for the ARM7 system, 2.40 for the ARM9, 2.56 for the MIPSa, and 2.41 for the MIPSb.

TABLE III
EXECUTION TIMES AND SPEEDUPS FOR THE PROCESSOR/CGRA1 SoCs

Application	Proc.	$Time_{sw}$	Ideal Sp.	Proc./CGRA1	
				$Time_{system}$	Speedup
JPEG enc.	ARM7	1.000	3.96	0.274	3.65
	ARM9	0.461	3.24	0.162	2.85
	MIPSa	0.996	3.32	0.333	2.99
	MIPSb	0.568	3.24	0.199	2.86
OFDM trans.	ARM7	1.000	3.54	0.312	3.21
	ARM9	0.485	3.43	0.164	2.95
	MIPSa	0.768	3.43	0.249	3.08
	MIPSb	0.590	3.29	0.206	2.87
Compressor	ARM7	1.000	2.51	0.457	2.19
	ARM9	0.424	2.32	0.221	1.92
	MIPSa	1.608	2.49	0.728	2.21
	MIPSb	1.044	2.34	0.514	2.03
Cavity det.	ARM7	1.000	2.38	0.493	2.03
	ARM9	0.480	2.29	0.258	1.86
	MIPSa	1.749	2.34	0.902	1.94
	MIPSb	1.154	2.23	0.617	1.87
Average			**2.90**		**2.53**
Geo. mean			***2.84***		***2.47***

Fig. 6 illustrates the application speedups for the processor/CGRA2 systems. The CGRA2 architecture enables direct connectivity among all the PEs of the same row and column. Also, the average speedup for each system is also given. By comparing the respective values of Fig. 6 and of Table III, it is inferred that for the microprocessor/CGRA2 systems the overall application speedups are marginally larger than the CGRA1 architecture. This is due to the fact that for the CGRA2 systems, the acceleration of kernels is slightly larger than the CGRA1 case owing to the richer interconnect among the PEs in the former case. The average speedup for all the processor systems and the four applications, equals 2.55 for the CGRA2 case.

From the performance improvements shown in Table III and Fig. 6, it is noticed that better performance gains are accomplished for the ARM7 system than the ARM9-based one. This occurs since the speedup of kernels on the CGRA has greater effect when the CGRA is coupled with a lower-performance processor, as it is the ARM7 relative to the ARM9. Additionally, the speedup is greater for the MIPSa system than the MIPSb case, since the latter processor employs one more integer ALU unit.

Such amounts of speedups as the ones reported in Table III and Fig. 6 were also considered as important in previous works considering processor/CGRA systems [13], [14], [15]. Furthermore, it is easily inferred that the reported speedups for each application and for each system are quite close to the ideal speedups, especially for the ARM7 systems. Thus, the proposed design flow efficiently utilized the processing capabilities of the 4x4 CGRAs for improving the overall performance of the DSP applications near to the theoretical bounds.

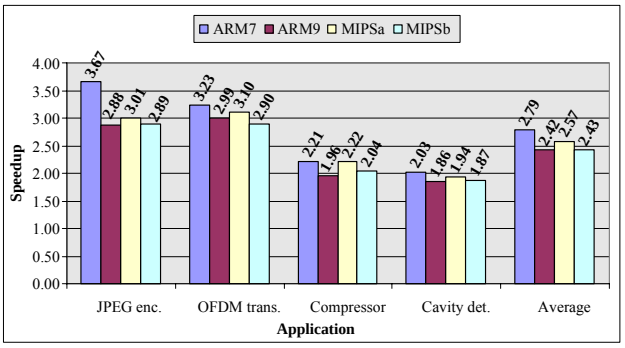

Fig. 6 Application speedups for the processor/CGRA2 SoCs

VI. Conclusions

Performance figures by mapping four DSP applications on eight instances of a processor/CGRA platform were given. A design flow for improving system performance by executing critical kernel code on the CGRA was introduced. A mapping algorithm that considers a flexible CGRA template is the core of the design flow. The results showed that the CGRAs are efficient in accelerating kernel code even when the available internal bandwidth among the PEs for exchanging data values is restricted. Important application speedups, over the software only execution on the processor, were reported.

References

[1] R. Hartenstein, "A Decade of Reconfigurable Computing: A Visionary Retrospective", in *Proc. of ACM/IEEE DATE '01*, pp. 642-649, 2001.
[2] E. Waingold, M. Taylor, D. Srikrishna, V. Sarkar, W. Lee, V. Lee, J. Kim, M. Frank, P. Finch, R. Barua, J. Babb, S. Amarasinghe, A. Agarwal. "Baring it all to software: RAW machines", in *IEEE Computer*, vol. 30, no. 9, pp. 86-93, Sept. 1997.
[3] T. Miyamori and K. Olukutun, "REMARC: Reconfigurable Multimedia Array Coprocessor", in *IEICE Trans. on Information and Systems*, vol. E82-D, no. 2, pp. 389-397, Feb. 1999.
[4] H. Singh, M.-H. Lee, G. Lu, 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.
[5] Morpho Reconfigurable DSP (*r*DSP) IP core, Morpho Technologies, www.morphotech.com, 2005.
[6] 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.
[7] J. Becker, M. Vorbach, "Architecture, Memory and Interface Technology Integration of an Industrial/Academic Configurable System-on-Chip (CSoC)", in *Proc. of ISVLSI*, IEEE Computer Society Press, pp. 107-112, 2003.

[8] S. C. Goldstein, H. Schmit, M. Budiu, S. Cadambi, M. Moe, R. R. Taylor, "PipeRench: A Reconfigurable Architecture and Compiler", in *IEEE Computer*, vol. 33, no. 4, pp. 70-77, April 2000.
[9] D. C. Cronquist, P. Franklin, S. G. Berg, C. Ebeling, "Specifying and Compiling Applications for RaPiD," in *Proc. of FCCM '98*, pp. 116-125, 1998.
[10] N. Bansal, S. Gupta, N. Dutt, A. Nikolau, R. Gupta, "Interconnect Aware Mapping of Applications to Coarse-Grain Reconfigurable Architectures", in *Proc. of FPL '04*, pp. 891-899, 2004.
[11] 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.
[12] G. Venkataramani, W. Najjar, F. Kurdahi, N. Bagherzadeh, W. Bohm and J. Hammes, "Automatic Compilation to a Coarse-Grained Reconfigurable System-on-Chip", in *ACM Transactions on Embedded Computing Systems*, vol. 2, no. 4, pp 560-589, Nov. 2003.
[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 ISOCC '04*, pp. 227-230, 2004.
[14] B. Mei, S. Vernalde, D. Verkest, R. Lauwereins, "Mapping methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture, A Case Study", in *Proc. of ACM/IEEE DATE '04*, pp. 1224-1229, 2004.
[15] F.-J. Veredas, M. Scheppler, W. Moffat, B. Mei, "Custom implementation of the Coarse-Grained Reconfigurable ADRES architecture for Multimedia purposes", in *Proc. of FPL '05*, pp. 106-111, 2005.
[16] ARM Corp., www.arm.com, 2005.
[17] G. De Micheli, *Synthesis and Optimization of Digital Circuits*, McGraw-Hill, 1994.
[18] SUIF2 compiler infrastructure, http://suif.stanford.edu/suif/suif2/index.html, 2005.
[19] M. D. Smith and G. Holloway, "An Introduction to Machine SUIF and its Portable Libraries for Analysis and Optimization", *Technical Report*, Harvard University, 2002. http://www.eecs.harvard.edu/hube/research/machsuif.html.
[20] K. Kennedy and R. Allen, "Optimizing compilers for modern architectures", *Morgan Kauffman Publishers*, 2002.
[21] J.W. Crenshaw, "MATH Toolkit for Real-Time Programming", *CMP Books*, 2000.
[22] S. Kumar, L. Pires, S. Ponnuswamy, C. Nanavati, J. Golusky, M. Vojta, S. Wadi, D. Pandalai, H. Spaanenberg, "A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future directions", in *Proc. of FPGA*, pp. 126-134, 2000.
[23] M. Bister, Y. Taeymans, J. Cornelis, "Automatic Segmentation of Cardiac MR Images", in *Proc. of Computers in Cardiology*, IEEE Computer Society Press, pp.215-218, 1989.
[24] SimpleScalar LLC, http://www.simplescalar.com, 2005.