# Design flow for Optimizing Performance in Processor Systems with on-chip Coarse-Grain Reconfigurable Logic

Michalis D. Galanis[1], Gregory Dimitroulakos[2], and Costas E. Goutis[3]

*VLSI Design Lab., Electrical and Computer Engineering Department, University of Patras, Rio 26500, Greece*
*e-mail: {[1]mgalanis, [2]dhmhgre, [3]goutis}@ee.upatras.gr*

## Abstract

*A design flow for processor platforms with on-chip coarse-grain reconfigurable logic is presented. The reconfigurable logic is realized by a 2-Dimensional Array of Processing Elements. Performance is improved by accelerating critical software loops, called kernels, on the Reconfigurable Array. Basic steps of the design flow have been automated. A procedure for detecting critical loops in the input C code was developed, while a mapping technique for Coarse Grain Reconfigurable Arrays, based on software pipelining, was also devised. Analytical results derived from mapping five real-life DSP applications on eight different instances of a generic system architecture are presented. Large values of Instructions Per Cycle were achieved on two Reconfigurable Arrays that resulted in high-performance kernel mapping. Additionally, by mapping critical code on the reconfigurable logic, speedups ranging from 1.27 to 3.18 relative to an all-processor execution were achieved.*

## 1. Introduction

Reconfigurable architectures have received growing interest in the past few years. Reconfigurable systems represent an intermediate approach between Application Specific Integrated Circuits (ASICs) and general-purpose processors [1]. 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 [2], [3], [4], [5], [6]. These architectures have been mainly proposed for speeding-up loops of multimedia and DSP applications in embedded systems. They consist of a large number of Processing Elements (PEs) with world-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]. We consider a popular subclass of coarse-grain architectures where the PEs are organized in a 2-Dimensional (2D) array [2]-[6]. 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], [5].

Although several coarse-grain reconfigurable architectures have been introduced, few automatic mapping flows have been proposed. In this paper, we present a design flow where most of its steps have been implemented as prototype tools. This flow interests in improving application's performance in System-on-Chips (SoCs) composed by an instruction-set processor and a CGRA, like the ones in [2], [3], [4], [5], [6]. Speedups are achieved by partitioning the input C description and accelerating critical software loops, called *kernels*, on the CGRA. The processor executes the non-critical software parts. Recently, design flows for SoC platforms composed by a processor and FPGA [7], [8] illustrated that such type of partitioning is feasible in embedded systems and it leads in important speedups. Such a 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 (typically loops), the kernels. This means that an extensive solution space search, as in past hardware/software partitioning works [9], [10] is not a requisite.

The proposed design flow mainly consists of the following steps: (a) an analysis procedure for detecting kernels at the input source code, (b) Intermediate Representation (IR) creation, (c) mapping algorithm for the CGRA architecture, and (d) compilation to the microprocessor. We emphasize to the mapping for CGRAs, since it considerably affects the performance

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

1-4244-0054-6/06/$20.00 

improvements through the kernels' acceleration. The proposed mapping procedure for CGRAs is based on a new modulo scheduling algorithm. Modulo scheduling is a software pipelining technique mainly used in Instruction Level Parallel (ILP) processors, like VLIWs, for improving operation parallelism by executing different loop iterations in parallel. Our modulo scheduler targets a generic CGRA template architecture which can model a variety of existing architectures [1], [2], [3], [4].

In this work, we present analytical results by mapping five real-world DSP applications on eight instances of a generic system architecture using the proposed design flow. The results from mapping applications' kernels on two CGRAs, a 4x4 and 6x6 array of PEs, using our-developed modulo scheduling algorithm show that high-performance mapping was achieved. The average Instructions Per Cycles (*IPC*), when the kernels of the five applications are mapped on a 4x4 array, equals 13.0 and it is considerably larger than the achieved IPC of previous modulo schedulers for CGRAs. The average kernels speedup over the execution on four 32-bit RISC processors is 60 for the 4x4 array, while for the 6x6 CGRA is 105. Additionally, the overall application speedup for all the applications and all the platform instances ranges from 1.27 to 3.18 relative to an all-microprocessor solution.

The rest of the paper is organized as follows: section 2 presents the related work, while section 3 describes the proposed design flow. Section 4 presents the analysis procedure. The CGRA architecture template and the developed modulo scheduling algorithm are given in section 5. Section 6 presents the experimental results and section 7 concludes this paper.

## 2. Related work

In recent years, some design flows for CGRAs coupled with a processor have been presented. The work of [5] describes a design flow for an XPP-based system. Performance results from mapping DSP algorithmic kernels on the XPP array are given. In [11] 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 [5] and in [11] the mapping of a complete DSP application is not performed. In [12], 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 [13]. The kernel and the overall application speedup over an 8-issue VLIW processor were 4.84 and 3.05, respectively. In our work, we apply the design flow in five realistic DSP applications and in eight different instances of a generic microprocessor-CGRA architecture, where useful conclusions can be drawn from this exploration.

Modulo scheduling is a loop pipelining technique that exploits instruction (operation) level parallelism out of loops by overlapping successive iterations of the loop and executing them in parallel. The main idea is to construct the schedule of one loop iteration such that this same schedule is repeated at regular intervals while satisfying data dependencies and resource constraints. The number of cycles between the initiations of successive iterations in a software pipelined loop is defined as the Initiation Interval (*II*). Various modulo schedulers have been proposed for VLIW architectures [14], [15], [16]. The modulo scheduling algorithms for VLIWs cannot be directly applied to CGRAs, since the algorithm must combine the scheduling, placement and routing of data values. The routing problem does not exist or it is rather easy to be solved even for clustered VLIW architectures. Thus, the realization of modulo scheduling algorithm for CGRAs is a challenging issue as it was also stated in [17].

## 3. Design flow

### 3.1. Generic system architecture

A generic embedded SoC architecture, shown in Figure 1, is considered by the design flow. The system includes: (a) Coarse-Grain Reconfigurable Array for executing kernels, (b) shared system data memory, (c) instruction and context (configuration) memories, and (d) an instruction-set processor. The processor is typically a RISC one, like an ARM9. Communication between the CGRA and the microprocessor takes place via the shared data RAM and several direct signals. Part of the direct signals is used by the microprocessor for controlling and communicating with 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, a *done* signal is typically present which notifies the microprocessor that the execution of a critical software part finished on the CGRA.

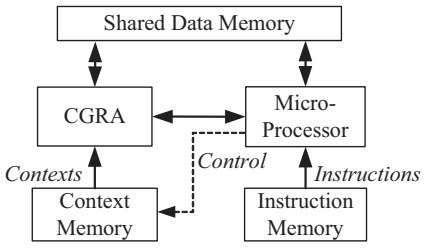


**Figure 1. Target system architecture.**

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. 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. 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 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 parallel execution on processor and on the CGRA is a topic of our future research activities.

### 3.2. Flow description

The proposed design flow for processor-CGRA SoCs interests in improving application's performance by mapping critical software parts on the coarse-grain reconfigurable hardware. This flow takes advantage of the fact that few kernels of DSP and multimedia applications contribute the most to the execution time. The design flow is illustrated in Figure 2. The input is C source code implementing an application. Firstly, we identify the critical loops of the application using an our-developed analysis tool. The computational complexity of a loop is represented by the instruction count, which is the number of instructions executed in running the application on the microprocessor. The dynamic instruction count has been also used as a measure for identifying critical loop structures in previous works [7]. A threshold, set by the designer, is used to characterize specific loops as kernels. The non-critical source code is modified to include calls to CGRA and to handle the communication with the CGRA. Then, the source code is compiled using a compiler for the specific processor.

The Intermediate Representation (IR) of the kernel loops is created. We have selected the Data Dependence Graph (DDG) representation. An IR creation tool based on SUIF2 [18] and MachineSUIF [19] compilers has been developed. Optimizations are then applied to the kernel's DDG 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, constant propagation and loop transformations. 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 PEs of the CGRA using a method, like the Friden algorithm [21] that has been implemented in the proposed flow. MachineSUIF [19] compiler passes have been developed for the automatic application of the optimizations and transformations on the kernel's DDG.

The optimized DDG of each kernel is input to an our-developed mapper tool for CGRAs based on a new modulo scheduling algorithm, which is the core of the design flow. The proposed modulo scheduler is explained in section 5.2. The second input to the mapper is the description of the CGRA architecture. The feedback arrow refers to the exploration performed for achieving the best performance for an input kernel. The configuration of the CGRA and the execution cycles of kernels are reported by the CGRA mapper. The dark grey boxes in Figure 2 represent the procedures modified or created by the authors for the specific flow, while the light grey one the external tool used.

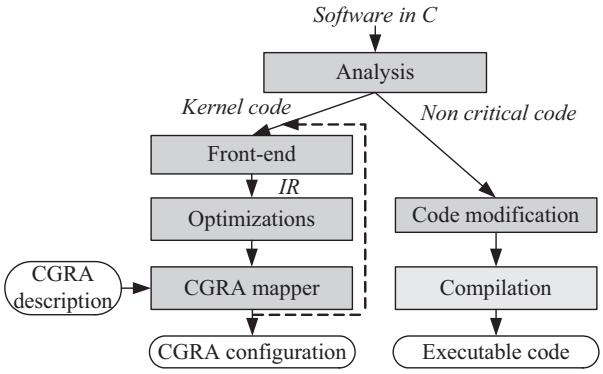


**Figure 2. Design flow for processor-CGRA systems.**

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

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

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 proposed flow requires the execution times of kernels on the coarse-grain reconfigurable logic. Since, those times can be also given by other mapping algorithm than the one proposed in this work, the design method can be applied in conjunction with other mapping algorithms [17], [22], [23]. Additionally, it 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 design flow can also consider other types of coarse-grain reconfigurable hardware like linear arrays [24].

## 4. Analysis procedure

The analysis step of the design method outputs the kernels and non-critical parts of the input source code. We have developed a new analysis method since tools, like the *gprof*, *Vtune*, *Quantify*, provide profiling results when the application runs on a host computer (PC) that its characteristics can be fairly different from the embedded processor used in the platform. The inherent computational complexity of loops, represented by the dynamic instruction count, is a rational measure to detect kernels. The instruction count when an application runs on the microprocessor is obtained by a combination of dynamic and static analysis within loops. Figure 3 shows the analysis flow. The analysis has been automated using the SUIF2 [18] and MachineSUIF [19] compiler infrastructures.

The DDG of the application's source code is constructed using our IR creation tool. For the DDG description, we have chosen the SUIF Virtual Machine (SUIFvm) representation for the instruction opcodes [19]. The SUIFvm instruction set assumes a generic RISC machine, not biased to any existing architecture. Thus, the information obtained by the analysis flow, could stand for any RISC processor architecture. This means that the detected critical loops are kernels for various types of RISC processors. This was justified by 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 loops is retained in the RISC processors used in our experiments.

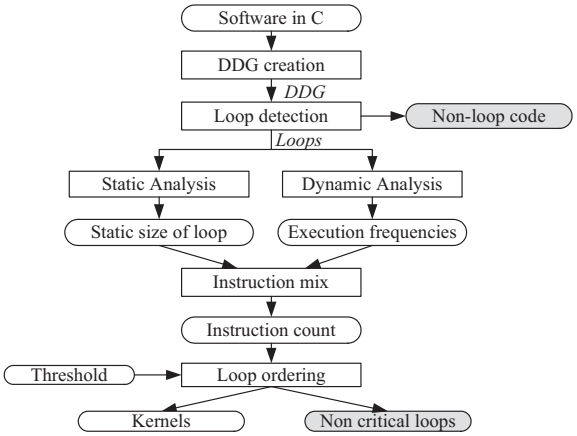


**Figure 3. Analysis flow.**

A MachineSUIF compiler pass was developed for detecting loops in the input DDG. Then, static and dynamic analysis is performed in the detected loops. We have used the HALT library of the MachineSUIF distribution [19] for performing dynamic analysis at the loop level. The dynamic analysis step reports the execution frequency of the loops. For the static analysis, a MachineSUIF compiler pass has been developed that identifies the type of instructions (operations) inside each loop and calculates the static size of the loop using the SUIFvm opcodes. The static size and the execution frequency of the loops are inputs to a developed instruction mix pass that outputs the dynamic instruction count. After the instruction count calculation for each loop, an ordering of the loops is performed. We consider kernels, those loops which have an instruction count over a user-defined threshold. This threshold represents the percentage of the contribution of the loop's instruction count in the application's overall dynamic instructions. For example, a loop contributing 10% or more to the total instruction count can be considered as kernel. The non-loop code and the non-critical loops will be executed on the instruction-set processor.

# 5. Mapping algorithm for CGRAs

## 5.1. CGRA template

The considered generic CGRA template is based on characteristics found in existing 2D coarse-grain reconfigurable architectures [1], [2], [3], [4] and it can be used as a model for mapping applications to CGRAs. A generic diagram of the proposed architecture template is shown in Figure 4a. Each PE is connected to its nearest neighbours, while there are cases [3], [4] 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. Figure 4b shows an example of PE architecture. The specific FU supports predication; thus through "if-conversion" [20], loops containing conditional statements are supported by the CGRA. Hence, the FU has three inputs and three outputs, where there is an (1-bit wide) input for the predicate guard [20] and two (1-bit) outputs for the predicate definitions. For storing intermediate values between computations and data fetched from memory, a small local data RAM exists inside a PE. The 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 FU can be routed to other PEs or to its local data RAM.

There is local context (configuration) RAM inside the PE that stores a few contexts locally which can be loaded on cycle-by-cycle basis. A context word controls the type of operation implemented by the FU, the multiplexers and the local data RAM behaving like an instruction in microprocessors. The context RAMs of the PEs form a distributed context memory which allows for the fast reconfiguration of the CGRA. The configurations can also be loaded from the main context memory at the cost of extra delay, if the local PE context RAM is not large enough. The main context memory of the CGRA (Figure 4a) stores the whole configuration for setting up the CGRA for the execution of application's kernels.

The main data RAM 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 memory bus connection to the scratch-pad memory, as in [2], [3], [4]. The scratch-pad serves as a local memory for quickly loading data in the PEs of the CGRA.

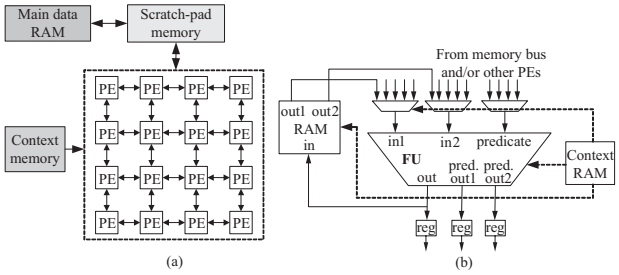


**Figure 4. (a) CGRA template, (b) Example of PE architecture.**

We note that the organization of the PEs and their interface to the data memory largely resembles the 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 [13].

## 5.2. Modulo scheduler description

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. Figure 5a shows the flow of the proposed modulo scheduling algorithm for CGRAs that has been implemented in C++. The first input to the scheduler is the kernel's DDG, while the second one is a description of the CGRA. 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 description includes parameters, like the size of the PE's local data RAM, the memory buses to which each PE is connected, the memory bus bandwidth and the scratch-pad memory access times.

The scheduler is based on the two-stage hierarchical reduction technique described in [14] where firstly the operations inside each Strongly Connected Component (SCC) of the DDG are scheduled. Secondly, the operations that belong to each SCC are condensed to a single operation. Then, the algorithm schedules the resulting condensed DDG of the input kernel which is acyclic. In [14], in both stages, the closure of dependence constraints is used to assure the satisfaction of dependence constraints where a fixed execution delay for each operation is assumed. However, for CGRAs, due to the non-deterministic delay of the data routing, the execution delay depends on the place where the operations are scheduled. For this reason, the proposed scheduling algorithm was adapted to take into account the non-deterministic nature of the operation scheduling in CGRAs. Firstly, the scheduler considers both the SCCs and the regular operations when they are ready to be executed. Secondly, instead of using the closure of dependence constraints which is calculated prior scheduling in [14], it checks during each operation's scheduling if the operation's execution time is compatible with execution times imposed by the data routing delays.

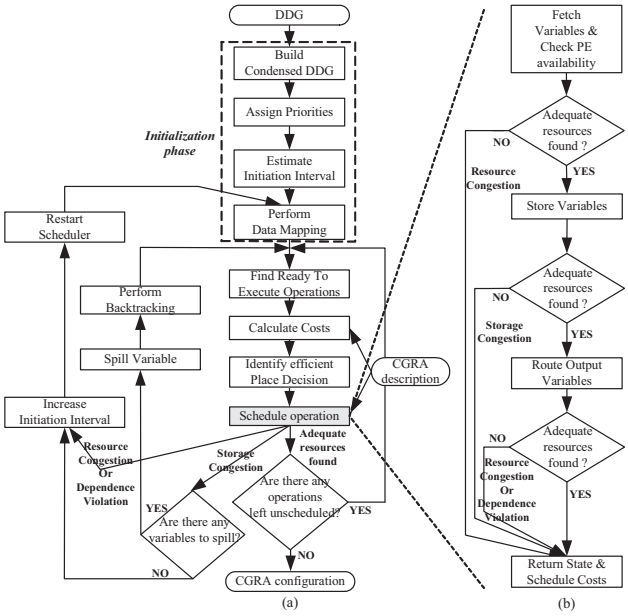


**Figure 5. (a) Modulo scheduling flow, (b) Scheduling phases.**

After the creation of the condensed DDG, the scheduler calculates the priorities of the operations. The priority of an operation is set to its *height* which is defined as the maximum distance to an operation without successors. Afterwards, the initiation interval is calculated as: $II = \max(RecII, ResII)$ (2) where *RecII* is the initiation interval imposed by cycles created by loop-carried dependences and *ResII* is the initiation interval defined by the resource constraints of the architecture [20]. Then, the data mapping is performed which means that the place, where the variables are stored after consumption and production, is defined. Initially, we assume that all variables are stored in the PEs' local data RAMs after production or consumption.

After the *initialization phase*, the mapping algorithm iterates for scheduling all operations one by one, scheduling each time, from the ready to be executed operations the one which has the highest priority (height). An operation is considered ready to be executed when all its predecessors with zero dependence distance are

scheduled. The SCCs have higher priority than the regular operations since they require more resources to be scheduled and the II is largely affected by them [14]. Two costs, which are described in section 5.2.1, are used for identifying an efficient place decision (*PD*) for executing an operation. In this step, the availability of resources together with the two costs are estimated for a possible execution of the operation in every PE in the CGRA.

Next, the actual scheduling of the operation takes place. The scheduling of an operation finishes normally if the required resources exist. Depending on the availability of resources, different actions are performed by the scheduler. In case where the PE's local data RAM size is not adequate for finding a possible PD for an operation in the CGRA, the algorithm *spills* the appropriate variables for scheduling the operation. Among the set of candidate variables for spilling, the modulo scheduling algorithm chooses the one that belongs to the operation with the minimum priority. Then, the algorithm *backtracks* to the operation to which the variable belongs and continues the scheduling process. If there are no variables left to be spilled, the algorithm fails for the current II and the scheduling phase restarts with an increased value of the II by one. Additionally, in case where some other resource (PE, interconnection, memory bus) is not adequate for finding a feasible PD or in case where dependences are violated, the mapping algorithm increases again the II by one and restarts the scheduler.

As shown in Figure 5b, each operation's scheduling is performed through a series of phases. In the *first phase*, the input variables are fetched to the PE where the execution takes place. In case where the input variables come from memory, the memory bus resources are reserved for transferring the data values. On the contrary, when the variables exist in a PE, the appropriate number of interconnections and storage locations in the local data RAMs are reserved for routing the data values to the target PE. Additionally, in the first phase, the availability of the PE where the execution takes place is checked. If there are not sufficient resources for completing the first phase, the scheduling stops and a *resource congestion* status is returned. In the *second phase*, storage of the variables occurs at the PE, where the execution takes place, when the variables do not arrive simultaneously at the target PE. Furthermore, local RAM is reserved at the source PEs for storing the data values until they are routed to the target PE. If the algorithm fails to find the storage resources, then the scheduling phase fails and a *storage congestion* status is returned. Finally, in the *third phase*, for loop dependent dependencies, routing of the output variables is required. The output variables should reach the target PE before the operation that consumes them takes place. If this time bound is violated, then dependencies are also violated and a *dependence violation* status is returned. Additionally, the availability of appropriate resources for routing the output variables is checked and a *resource congestion* status is returned if there are no adequate resources.

#### 5.2.1. Mapping Costs

For finding an effective place decision for an operation, two costs are utilized. These costs are calculated for a possible execution of the operation in each PE of the CGRA. The first one, called *delay cost*, refers to the operation's *Op* earliest possible schedule time if it is placed for execution in a specific $PE_x$. As shown in eq. (4), the delay is the sum of the *RTime* plus the maximum of the times *tf* required to fetch the *Op*'s input operands to $PE_x$. The *RTime* (eq. (3)) equals to the maximum of the times where each of the *Op*'s predecessors with zero dependence distance *P(Op)* finished executing $t_{fin}$. *P* is the set having the predecessor operations of *Op*.

$$RTime(Op) = \max_{i=1,\ldots,|P(Op)|}\left(t_{fin}(Op_i),0\right) \text{ where } Op_i \in P(Op) \quad (3)$$

$$Delay\_cost(PE_x, Op) = RTime(Op) + \max_{i=1,\ldots,|P|}\left(tf_{P[i]},0\right) \quad (4)$$

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

The second cost is the *interconnection cost* that refers to the interconnections that need to be reserved for scheduling an operation in a specific PE. As shown in eq. (5), 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.

$$Interconnection\_cost(PE_x, Op) = \sum_{i \in P(Op)} PathLength\left(PE_{Op_i} \rightarrow PE_x\right) \quad (5)$$

A greedy approach was adopted for calculating the time for routing the operands and the number of interconnections (eq.(5)) 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.(4) and eq.(5), respectively. The adopted PD for each operation is the one with the minimum delay cost. If there are multiple PDs with the same delay cost, the one that minimizes the interconnection cost is adopted.

## 6. Results

### 6.1. Experimental set-up

Five real-life DSP applications, written in C language, were mapped on eight different instances of the generic processor-CGRA platform using the developed design

flow. These applications are: (a) a still-image JPEG encoder, (b) an IEEE 802.11a OFDM transmitter, (c) a wavelet-based image compressor [26], (d) a cavity detector which is a medical image processing application [26], and (e) a video compression technique, called Quadtree Structured Difference Pulse Code Modulation (QSDPCM) [28]. The experiments were performed using the following applications' inputs: (a) an image of size 256x256 bytes for the JPEG encoder, (b) 4 payload symbols for the OFDM transmitter at the 54 Mbps rate, (c) an image of size 512x512 bytes for the wavelet-based image compressor, (d) an image of size 640x400 bytes for the cavity detector, and (e) two video frames of size 176x144 bytes each for the QSDPCM.

We have used four different types of 32-bit embedded RISC processors coupled each time with the CGRA: an ARM9, an ARM10, and two SimpleScalar processors [24]. The SimpleScalar processor is an extension of the MIPS32 IV core. The first type of the MIPS processor (*MIPSa*) uses one integer ALU unit, while the second one (*MIPSb*) has two integer ALUs. We have used instruction-set simulators for the considered embedded processors for estimating the number of execution cycles of the non-critical parts. More specifically, for the ARM processors, the ARM RealView Developer Suite (version 2.2) was utilized, while the performance on the MIPS-based processors was estimated using the SimpleScalar simulator tool [24]. Typical clock frequencies are considered for the four processors: the ARM9 runs at 250 MHz, the ARM10 at 325 MHz, and the MIPS processors at 200 MHz. The five applications were compiled to generate binary files for the processors using the highest level of software optimizations (-O3).

Two different CGRA architectures were used each time for accelerating critical kernels. The first architecture is a 4x4 array of PEs, while the second one consists of 36 PEs connected in a 6x6 array. In the following, we list the characteristics for both CGRAs. In both architectures, the PEs are directly connected to all other PEs in the same row and same column through vertical and horizontal interconnections. There is one 16-bit FU in each PE that can execute any operation (i.e. multiplication, ALU, shift) in one CGRA's clock cycle. Each PE has a local data RAM of size 8 words, while the local configuration RAM's size is 32 contexts as in [2]. 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 memory. The delay of fetching one word from the scratch-pad memory is one cycle. The CGRA clock frequencies are 100 MHz, as in the case of an implementation of the MorphoSys SoC [3].

## 6.2. Experimentation

The results using the developed analysis flow are shown in Table 1. The static size of the kernels (in instruction bytes) and their contributions of the kernels to the total static size and to the total instructions are reported. The threshold for the kernel detection was set to the 10% of the total dynamic instructions of the application. The theoretical speedup, according to Amdahl's Law, if the application's kernels were ideally executed on the CGRA in zero time is given by the Speedup bound. The number of kernels detected in each application is also given. For all applications the detected loops consist of word-level operations (ALU, multiplications, shifts) that match the granularity (data bit-width) of the PEs in the CGRA. From the analysis results, it is inferred that an average of 10.9% of the code size, representing the kernels' size, contributes 63.9% on average to the total executed instructions. The speedup of each application will come from accelerating few 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 [9], [10] is not necessary in the case of partitioning the considered applications on the processor-CGRA systems.

**Table 1. Results from the analysis procedure**

| App. | Kernels' size | % size | % total instructions | Sp. bound | # of kernels |
|---|---|---|---|---|---|
| JPEG | 2,534 | 23.0 | 75.6 | 4.10 | 4 |
| OFDM | 1,440 | 9.2 | 72.4 | 3.62 | 4 |
| Compress | 602 | 4.7 | 61.2 | 2.58 | 4 |
| Cavity | 910 | 7.6 | 59.5 | 2.47 | 4 |
| QSDPCM | 2,477 | 10.0 | 51.0 | 2.04 | 3 |
| **Average:** | | **10.9** | **63.9** | | |

The results from mapping the critical loops of the applications on the 4x4 CGRA are given in Table 2. The first column refers to the kernel *kn* of an application, while the second one to the number of operations composing each loop after the unrolling performed for achieving better CGRA utilization and consequently better performance. The MII is the computed minimal initiation interval, while II is the interval actually achieved during modulo scheduling. The obtained Instructions Per Cycle (*IPC*) is given in the fifth column. The IPC indicates the average number of operations executed per clock cycle in the scheduled loop. The IPC is a measure of the operation parallelism exploited and dictates the performance in modulo schedulers. Finally, the sixth column gives the execution cycles of each kernel on the 4x4 CGRA, while the speedup relative to the execution of each kernel on the ARM9 is shown in the seventh column.

From Table 2, it is inferred that the achieved II is equal to the MII in 17 out the 19 kernels something that reflects the quality of the CGRA mapping. Additionally, the 4x4 CGRA is efficiently utilized since the average IPC is 13.0. The average IPC in Table 2 is considerably larger

than the corresponding IPC values achieved when various DSP kernels were mapped on 4x4 CGRAs in [21], [29]. Actually, in [29] the average IPC resulted from mapping 20 DSP kernels on a 4x4 CGRA (Table 15 of [29]) using the modulo scheduler introduced in [17], is 1.36 times smaller than the one achieved in our work. The large values of IPC in Table 2 reveal the high-performance mapping of the kernels on the 4x4 array. Furthermore, large values of speedup relative to the execution on the ARM9 are achieved, with an average speedup equal to 94. The maximum combined II (sum of the II values for an application's loops) among the applications equals 16 and refers to the JPEG encoder. Since, the II defines the number of configuration contexts needed to execute the loop and a PE's local context RAM stores 32 contexts, there will be no time overhead for loading the context RAMs during the execution of each application.

**Table 2. Results from mapping kernels on the 4x4 CGRA and speedup relative to ARM9**

| Kernel | # ops | MII | II | IPC | Cycles | Sp. |
|---|---|---|---|---|---|---|
| Jpeg_enc.k1 | 100 | 7 | 7 | 14.3 | 28,674 | 71 |
| Jpeg_enc.k2 | 104 | 7 | 7 | 14.1 | 28,692 | 73 |
| Jpeg_enc.k3 | 12 | 1 | 1 | 12.0 | 15,887 | 541 |
| Jpeg_enc.k4 | 16 | 1 | 1 | 16.0 | 44,378 | 16 |
| Ofdm_trans.k1 | 16 | 1 | 2 | 8.0 | 1,058 | 30 |
| Ofdm_trans.k2 | 30 | 2 | 2 | 15.0 | 1,161 | 138 |
| Ofdm_trans.k3 | 43 | 3 | 4 | 10.8 | 965 | 53 |
| Ofdm_trans.k4 | 4 | 1 | 1 | 4.0 | 433 | 20 |
| Compressor.k1 | 32 | 2 | 2 | 16.0 | 85,135 | 36 |
| Compressor.k2 | 16 | 1 | 1 | 16.0 | 21,505 | 112 |
| Compressor.k3 | 32 | 2 | 2 | 16.0 | 85,135 | 38 |
| Compressor.k4 | 16 | 1 | 1 | 16.0 | 43,010 | 56 |
| Cavity_det.k1 | 24 | 2 | 2 | 12.0 | 507,850 | 88 |
| Cavity_det.k2 | 8 | 1 | 1 | 8.0 | 55,385 | 100 |
| Cavity_det.k3 | 24 | 2 | 2 | 12.0 | 190,443 | 80 |
| Cavity_det.k4 | 24 | 2 | 2 | 12.0 | 190,443 | 68 |
| Qsdpcm.k1 | 27 | 2 | 2 | 13.5 | 633,609 | 33 |
| Qsdpcm.k2 | 29 | 2 | 2 | 14.5 | 912,398 | 38 |
| Qsdpcm.k3 | 16 | 1 | 1 | 16.0 | 4,318,851 | 190 |
| **Average:** | | | | **13.0** | | **94** |

When the kernels were mapped on the 6x6 CGRA, the average speedup relative to the ARM9 execution was 181. The achieved average IPC for the 6x6 array was 25.2. Additionally, the maximum combined II equals 7 and corresponds to the loops of the OFDM transmitter. Thus, for the 6x6 array there is also no overhead for loading the PEs' context RAMs.

Figure 6a 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. Figure 6b shows the respective speedups when the 6x6 array is used in the processor platforms. For every application, the speedup is relative to each one of the four RISC processors used. For example, the left most bar in each application corresponds to the speedup obtained when the execution cycles of the kernels are compared to the ones for the execution of the kernels on the ARM9. The kernel speedup is defined as:

$$Sp_{kernel} = Cycles_{kernels\_sw} / Cycles_{kernels\_CGRA\_norm} \quad (6)$$

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 5.2, are normalized to the clock frequency of the processor in the system platform, using the following relation:

$$Cycles_{kernels\_CGRA\_norm} = Cycles_{kernels\_CGRA} \cdot \frac{Clock_{proc}}{Clock_{CGRA}} \quad (7)$$

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

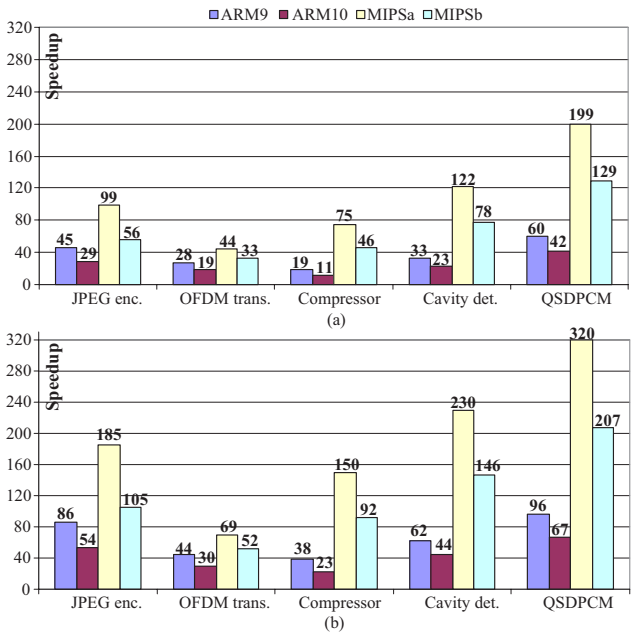


**Figure 6. Kernel speedups (a) on the 4x4 CGRA and (b) on the 6x6 CGRA, for various processor systems.**

From Figure 6, it is deduced that important speedups are achieved when critical kernels are executed on the CGRA. For the systems composed by 4x4 CGRA, the speedups range from 11 to 199, with an average value of 60 for all the applications and all the cases of microprocessors. When the 6x6 array is used, the kernel speedup ranges from 23 to 320, with an average value of 105. Thus, the average kernel speedup for the 6x6 array is 1.76 times larger than the one obtained with the 4x4 CGRA. The ratio of the number of PEs in the 6x6 array to the PEs in the 4x4 is 36/16=2.25. We can infer that the

additional PEs of the 6x6 array were efficiently exploited by the modulo scheduling algorithm since the ratio of the average kernel speedups is fairly close to the 2.25.

The achieved 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 RISC processors. Even in the case where the processors are clocked in a significantly higher clock frequency than the CGRAs (as in the ARM10 systems) the speedups are important. 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 microprocessor-CGRA system as it will be shown in Table 3. The kernel speedup is smaller in the ARM10-based systems than the ARM9-based ones since the ARM10 is a more contemporary microprocessor generation and it is clocked at a higher frequency. Furthermore, the effect of accelerating kernels on the CGRA is smaller when the MIPSb is employed in the platform since it has one more ALU unit than the MIPSa.

**Table 3. Execution cycles and speedups**

| Application | Proc. Arch. | $Cycles_{sw}$ | Ideal Sp. | 4x4 CGRA | | 6x6 CGRA | |
|---|---|---|---|---|---|---|---|
| | | | | $Cycles_{system}$ | Est. Sp. | $Cycles_{system}$ | Est. Sp. |
| JPEG enc. | ARM9 | 19,951,193 | 3.24 | 6,799,939 | 2.93 | 6,663,142 | 2.99 |
| | ARM10 | 16,930,629 | 3.16 | 6,273,578 | 2.70 | 6,095,741 | 2.78 |
| | MIPSa | 34,451,609 | 3.32 | 11,353,377 | 3.03 | 11,243,939 | 3.06 |
| | MIPSb | 19,637,417 | 3.24 | 6,694,006 | 2.93 | 6,584,568 | 2.98 |
| OFDM trans. | ARM9 | 362,990 | 3.43 | 120,465 | 3.01 | 117,182 | 3.10 |
| | ARM10 | 334,375 | 3.23 | 122,402 | 2.73 | 118,135 | 2.83 |
| | MIPSa | 459,594 | 3.43 | 147,125 | 3.12 | 144,499 | 3.18 |
| | MIPSb | 352,788 | 3.29 | 120,942 | 2.92 | 118,316 | 2.98 |
| Compressor | ARM9 | 20,574,658 | 2.32 | 10,045,845 | 2.05 | 9,752,357 | 2.11 |
| | ARM10 | 17,854,928 | 2.21 | 9,896,315 | 1.80 | 9,514,782 | 1.88 |
| | MIPSa | 62,468,206 | 2.49 | 27,777,661 | 2.25 | 27,542,871 | 2.27 |
| | MIPSb | 40,541,866 | 2.34 | 19,484,158 | 2.08 | 19,249,368 | 2.11 |
| Cavity det. | ARM9 | 161,441,889 | 2.29 | 85,387,604 | 1.89 | 84,283,439 | 1.92 |
| | ARM10 | 155,356,758 | 2.17 | 87,023,271 | 1.79 | 85,587,857 | 1.82 |
| | MIPSa | 470,433,835 | 2.34 | 241,717,256 | 1.95 | 240,833,924 | 1.95 |
| | MIPSb | 310,248,110 | 2.23 | 165,099,333 | 1.88 | 164,216,001 | 1.89 |
| QSDPCM | ARM9 | 3,895,248,922 | 1.54 | 3,033,750,555 | 1.28 | 3,028,219,028 | 1.29 |
| | ARM10 | 3,608,029,180 | 1.48 | 2,833,095,857 | 1.27 | 2,825,904,871 | 1.28 |
| | MIPSa | 7,006,016,541 | 1.73 | 4,682,812,043 | 1.50 | 4,678,386,821 | 1.50 |
| | MIPSb | 4,910,759,258 | 1.68 | 3,407,830,222 | 1.44 | 3,403,405,000 | 1.44 |
| | | | | **Average:** | **2.23** | | **2.27** |

The execution cycles and overall application speedups from applying the proposed design flow in the five applications are presented in Table 3. The results are given for accelerating the kernels on the 4x4 and the 6x6 CGRAs. For each application, the four considered processor architectures (*Proc. Arch.*) are used for estimating the clock cycles ($Cycles_{sw}$) required for executing the whole application on the processor. The ideal speedup (*Ideal Sp.*) reports the maximum performance improvement, according to Amdahl's Law, if application's kernels were ideally executed on the CGRA in zero time. The estimated speedup (*Est. Sp.*) is the measured performance improvement after utilizing the developed design flow. The estimated speedup is calculated as:

$$Est\_Sp = Cycles_{sw} / Cycles_{system} \quad (8)$$

where $Cycles_{system}$ represents the execution cycles after the partitioning to the processor and to the CGRA takes place.

From the results given in Table 3, it is evident that significant overall performance improvements are achieved when critical software parts are mapped on a CGRA. It is noticed that better performance gains are accomplished for the ARM9 system than the ARM10-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 ARM9 relative to the ARM10. This is clearly shown in the kernel speedup results in Figure 6. Furthermore, the speedup is almost always greater for the MIPSa system than the MIPSb case, since the latter processor employs one more integer ALU unit.

For the case of mapping the kernels on the 6x6 CGRA, the speedups are somewhat larger than the 4x4 CGRA case. The larger application speedups are due to the better kernel speedups (as shown in Figure 6) that obtained with the 6x6 array relative to the 4x4 CGRA. However, even though the kernel speedup is significantly improved for the 6x6 CGRA, the overall application speedup slightly increases due to the fact that the non-critical code segments are executed on the microprocessor. The average estimated application speedup is 2.23 for the 4x4 architecture, while for the 6x6 CGRA is equal to 2.27. We also notice that the reported estimated speedups for each application and for each processor type are somewhat close to the ideal speedups determined by the Amdahl's Law, especially for the case of the 6x6 CGRA.

## 7. Conclusions - Future work

A flow for improving the performance in processor-CGRA single-chip systems was presented. An efficient modulo scheduling algorithm is used to accelerate kernel code where high IPC values are achieved using this scheduler. Extensive experiments show that the application speedup ranges from 1.27 to 3.12 when a 4x4 CGRA is employed for kernels' acceleration. The kernel speedup is 1.76 times larger when a 6x6 CGRA is used in the platform, while the overall application speedup slightly increases with an average value of 2.27. Future work focuses on the parallel execution of the processor and the CGRA for achieving even greater performance improvements.

## 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. Olukotun, "REMARC: Reconfigurable Multimedia Array Coprocessor", in *IEICE Trans. on Information and Systems*, pp. 389-397, 1999.

[3] H. Singh et al., "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] Morpho Technologies, www.morphotech.com, 2005.

[5] V. Baumgarte et al., "PACT XPP - A Self-Reconfigurable Data Processing Architecture", in the *Journal of Supercomputing*, Springer, vol. 26, no. 2, pp. 167-184, September 2003.

[6] J. Becker et al., "Datapath and Compiler Integration of Coarse-grain Reconfigurable XPP-Arrays into Pipelined RISC Processor", in *Proc. of IFIP VLSI SoC*, pp. 288-293, 2003.

[7] J. Villareal et al., "Improving Software Performance with Configurable Logic", in *Design Automation for Embedded Systems*, Springer, vol. 7, pp. 325-339, 2002.

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

[9] D. D. Gajski et al., "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.

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

[12] Y. Kim et al., "Design and Evaluation of a Coarse-Grained Reconfigurable Architecture", in *Proc. of ISOCC '04*, pp. 227-230, 2004.

[13] B. Mei et al., "Design Methodology for a Tightly Coupled VLIW/Reconfigurable Matrix Architecture, A Case Study", in *Proc. of ACM/IEEE DATE '04*, pp. 1224-1229, 2004.

[14] M. S. Lam, "Software pipelining: An effective scheduling technique for VLIW machines", in *Proc. of SIGPLAN '88*, pp. 318-328, 1988.

[15] S. Pillai and M. F. Jacome, "Compiler-Directed ILP Extraction for Clustered VLIW/EPIC machines: Predication, Speculation and Modulo Scheduling", in *Proc. of ACM/IEEE DATE '03,* pp. 422-427, 2003.

[16] J. Zalamea et al., "Register Constrained Modulo Scheduling", in *IEEE Trans. on Parallel and Distributed Systems*, vol. 15, no. 5, pp. 417-430, May 2004.

[17] B. Mei et al., "Exploiting Loop-Level Parallelism on Coarse-grained Reconfigurable Architectures Using Modulo Scheduling", in *Proc. of ACM/IEEE DATE '03*, pp. 255-261, 2003.

[18] SUIF2, 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.

[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] N. Bansal et al., "Network Topology Exploration of Mesh-Based Coarse-Grain Reconfigurable Architectures", in *Proc. of ACM/IEEE DATE '04*, pp. 474-479, 2004.

[23] J. Lee et al., "Compilation Approach for Coarse-grained Reconfigurable Architectures", in *IEEE Design & Test of Computers*, vol. 20, no. 1, pp. 26-33, Jan.-Feb., 2003.

[24] D. C. Cronquist et al., "Specifying and Compiling Applications for RaPiD," in *Proc. of FCCM*, pp. 116-125, 1998.

[25] SimpleScalar LLC, http://www.simplescalar.com, 2005.

[26] S. Kumar et al., "A Benchmark Suite for Evaluating Configurable Computing Systems - Status, Reflections, and Future directions", in *Proc. of FPGA*, pp. 126-134, 2000.

[27] M. Bister et al., "Automatic Segmentation of Cardiac MR Images", in *Proc. of Computers in Cardiology*, IEEE Computer Society Press, pp.215-218, 1989.

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

[29] Z. Kwok and S. J. E. Wilton, "Register File Architecture Optimization in a Coarse-Grained Reconfigurable Architecture", in *Proc. of IEEE FCCM '05,* pp. 35-44, 2005.