Des Autom Embed Syst (2006) 10:27–47
DOI 10.1007/s10617-006-8732-6

A method for partitioning applications in hybrid reconfigurable architectures

Michalis D. Galanis · Athanasios Milidonis · George Theodoridis · Dimitrios Soudris · Costas E. Goutis

Abstract In this paper, we propose a methodology for accelerating application segments by partitioning them between reconfigurable hardware blocks of different granularity. Critical parts are speeded-up on the coarse-grain reconfigurable hardware for meeting the timing requirements of application code mapped on the reconfigurable logic. The reconfigurable processing units are embedded in a generic hybrid system architecture which can model a large number of existing heterogeneous reconfigurable platforms. The fine-grain reconfigurable logic is realized by an FPGA unit, while the coarse-grain reconfigurable hardware by our developed high-performance data-path. The methodology mainly consists of three stages; the analysis, the mapping of the application parts onto fine and coarse-grain reconfigurable hardware, and the partitioning engine. A prototype software framework realizes the partitioning flow. In this work, the methodology is validated using five real-life applications. Analytical partitioning experiments show that the speedup relative to the all-FPGA mapping solution ranges from 1.5 to 4.0, while the specified timing constraints are satisfied for all the applications.

Keywords Hybrid reconfigurable systems · Partitioning · Coarse-grain reconfigurable hardware · FPGA · scheduling

M. D. Galanis (✉) A. Milidonis · C. E. Goutis
VLSI Design Lab., Elect. & Comp. Eng. Dept., University of Patras, Greece
e-mail: mgalanis@ee.upatras.gr

G. Theodoridis
Section of Elect. & Computers, Physics Dept., Aristotle University, Thessalonica, Greece

D. Soudris
VLSI Design Center, Elect. & Comp. Eng Dept., Democritus University, Xanthi, Greece

1. Introduction

Reconfigurable architectures have been a topic of intensive research activities in the past few years. Reconfigurable fabrics can unify the performance of ASICs and the flexibility offered by the microprocessors [1]. In particular, hybrid (mixed) granularity reconfigurable systems [1–4] offer extra advantages in terms of performance and great flexibility to efficiently implement computational intensive applications, like Digital Signal Processing (DSP) and multimedia. These applications are characterized by mixed functionality, data and control. Hybrid reconfigurable architectures usually consist of: fine-grain reconfigurable units typically implemented in Field Programmable Gate Array (FPGA) technology, coarse-grain reconfigurable units implemented in ASIC technology, data and program memories, reconfigurable interconnection network and microprocessor(s). Due to the special characteristics of the heterogeneous (coarse and fine-grain) reconfigurable units included in a hybrid system architecture, certain parts of the application are better suited to be executed on the coarse-grain units and other parts on the fine-grain reconfigurable units.

The fine-grain reconfigurable hardware's granularity is typically four or five bits. Small bit-width operations can be efficiently executed on fine-grain hardware, as the granularity of the Configurable Logic Blocks (CLBs) of contemporary FPGA devices is four or five bits. Tasks of Finite State Machine type of functionality (e.g. control structures) are also good candidates to be implemented by the fine-grain reconfigurable hardware. The coarse-grain reconfigurable blocks efficiently execute word-level or sub word-level operations [1, 4, 5]. These blocks can slightly modify their functionality according to the application requirements. The execution of computational intensive parts of applications, like loops, on coarse-grain reconfigurable hardware offers great advantages in terms of performance, area and power consumption relative to the execution of these operations on the fine-grain reconfigurable units [1–3, 5, 6]. So, for exploiting the advantages offered by the coarse-grain reconfigurable hardware, the development of a methodology for partitioning an application onto the coarse and fine-grain reconfigurable hardware of a hybrid system is considered essential.

In this paper, an automated partitioning methodology between the fine and coarse-grain reconfigurable hardware of an embedded hybrid platform is introduced. The goal of the methodology is to improve system's performance for satisfying the timing constraints of application parts executed on the reconfigurable logic of the platform. The method is parameterized in respect to the fine and the coarse-grain reconfigurable parts of the target architecture. Both types of reconfigurable hardware are characterized in terms of timing and area features. The main parts of the method are the analysis procedure for detecting critical parts, called *kernels*, of the application segments mapped on the mixed granularity reconfigurable hardware, the mapping procedures for the fine and coarse-grain reconfigurable hardware, and the partitioning engine. In this work, for validating the effectiveness of the proposed method, we assume specific mapping algorithms for the fine and the coarse-grain reconfigurable logic. However, existing mapping procedures [7, 8] can be used for calculating the execution cycles on the FPGA or other types of coarse-grain reconfigurable hardware (and their corresponding mapping algorithms) can be considered [5, 9].

System's performance in embedded systems is typically improved when application's segments that contribute to the majority of the execution time, are speeded-up [10, 11]. Thus, we focus on these critical parts for accelerating an application through partitioning and mapping them on the coarse-grain reconfigurable hardware. A prototype software framework has been developed for implementing the proposed methodology. The methodology is evaluated in this paper using five real-life applications: (a) an IEEE 802.11a Orthogonal Frequency Division Multiplexing (OFDM) transmitter [12], (b) a cavity detector [13], (c) a video compression technique [14], (d) a wavelet-based image compressor [15], and (e) a JPEG still image encoder [16]. The performed experiments showed that the timing specifications of the applications are satisfied and the performance improvements, relative to an all FPGA solution, by applying the proposed part itioning methodology range from 1.5 to 4.0.

The rest of the paper is organized as follows: The related work is outlined in section 2, while section 3 describes the proposed partitioning methodology. Experimental results are given in section 4. Finally, section 5 concludes the paper and presents future activities.

2. Related work

This section presents existing research activities in developing hybrid reconfigurable systems and in hardware/software partitioning methodologies for reconfigurable architectures.

2.1. Hybrid reconfigurable systems

There has been considerable research for developing reconfigurable architectures in the past [1]. We are focusing on hybrid granularity systems containing fine and coarse-grain reconfigurable logic.

The Pleiades [3] System-on-Chip (SoC) architecture is an approach that combines an on-chip microprocessor with a number of heterogeneous reconfigurable units of different granularities connected via a reconfigurable interconnection network. The Strategically Programmable System (SPS) [2] is a hybrid reconfigurable system architecture that is composed by fine-grain reconfigurable units and coarse-grain modules which are pre-placed within a fully reconfigurable fabric. Chameleon heterogeneous SoC platform [4] contains a general purpose processor, an FPGA unit and a coarse-grain reconfigurable part. The latter part is composed by reconfigurable processor tiles, called MONTIUM. The hybrid granularity approach has been recently adopted in current FPGA devices, like the Xilinx Virtex-II/4 [17] and Altera Stratix [18]. These devices contain hardwired units, configured for example as multiply-accumulate operators, which operate on word-level operands and they can be considered as the coarse-grain hardware.

In the aforementioned hybrid reconfigurable systems, there is no automated partitioning methodology between the fine and coarse-grain reconfigurable units for improving application's performance by mapping kernels on the coarse-grain reconfigurable hardware.

2.2. Partitioning for reconfigurable architectures

Hardware/software partitioning can improve performance [19] and in some cases even reduce power consumption [20]. More recently, hardware/software partitioning techniques for single-chip systems composed by a microprocessor and FPGA [7, 11, 21–23], were developed. The FPGA unit was treated as an extension of the microprocessor. Critical parts of the application (typically loop structures) are moved for execution on the FPGA for improved performance and usually reduced energy consumption. This is due to the fact that it was observed that most embedded applications (usually DSP and multimedia ones) spend the majority of their execution time in loops or small code segments [10, 11, 23] which are characterized as kernels. This means that an extensive solution space search, as in past hardware/partitioning works [19, 20] is not necessary. However, the aforementioned partitioning methods do not consider coarse-grain reconfigurable blocks. Thus, they can not further accelerate an application since they do not benefit from the ability of the coarse-grain hardware for speeding-up applications [5, 6, 24]. Hardware/software partitioning approaches for systems composed by a processor and a coarse-grain reconfigurable array were presented in [25, 26]. Fine-grain reconfigurable hardware was not considered in those systems. So, custom or exceptional operations, like bit manipulations or divisions, cannot be efficiently executed on those systems. For example, the Processing Elements (PEs) of existing coarse-grain reconfigurable architectures [4, 5, 24, 26] do not support division operations.

3. Partitioning method

3.1. Hybrid reconfigurable architecture

The generic hybrid reconfigurable SoC considered by the methodology, which mainly targets DSP and multimedia applications, is shown in Fig. 1. This embedded system includes coarse and fine-grain reconfigurable hardware units for data processing, shared data RAM, and a reconfigurable interconnection network. In this work, the coarse-grain hardware is our-developed high-performance coarse-grain reconfigurable data-path proposed in [27], while the fine-grain reconfigurable hardware is realized by an FPGA. Both the coarse and the fine-grain reconfigurable hardware units compose the Reconfigurable Functional Unit (RFU) of the hybrid system. All the above components are integrated in a SoC and they are configured by an instruction-set processor. The microprocessor executes control-intensive parts of an application. For example, the Medium Access (MAC) layer of a wireless LAN protocol [12] is executed on the microprocessor, while the baseband processing, which is more computational intensive than the MAC layer, is implemented on the RFU of the hybrid SoC. This generic architecture can model a variety of existing hybrid reconfigurable architectures, like the ones considered in [2–4].

The execution (programming) model of the hybrid reconfigurable system architecture considers that the data communication among the computational components (microprocessor and RFU) uses shared-memory mechanism. The shared memory is comprised by the system's shared data RAM and memory-mapped registers within both types of reconfigurable logic. Scalar variables, either live-in or live-out ones, are exchanged via the shared registers. Global variables and data arrays are allocated in the system's shared data RAM. The microprocessor, the fine and the coarse-grain reconfigurable hardware have access to the shared

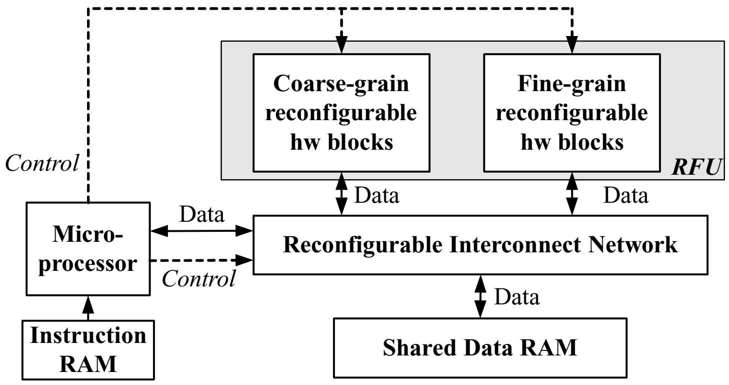

Control
Coarse-grain reconfigurable hw blocks
Fine-grain reconfigurable hw blocks
RFU
Data
Data
Micro-processor
Data
Control
Reconfigurable Interconnect Network
Data
Instruction RAM
Shared Data RAM

Fig. 1 Generic hybrid reconfigurable platform

memory. The communication mechanism used by the computational nodes preserves data coherency by requiring their execution to be mutually exclusive. The mutual exclusive execution makes the programming of the system architecture easier by eliminating complicated analysis and synchronization procedures. The system architecture is also simpler with the exclusive execution because the microprocessor and the RFU will never access the same memory address space simultaneously.

3.2. Method overview

Fig. 2 shows a generic design flow for mapping an application to a hybrid reconfigurable system. First, a hardware/software partitioning stage, like the ones in [11, 19, 20, 23], defines the software parts to be executed on the processor and on the RFU. The software parts decided to be executed on the reconfigurable hardware contribute the most to the execution time of an application. Our methodology *focuses* on partitioning the code mapped on the RFU to the fine and coarse-grain reconfigurable hardware for satisfying timing specifications.

In Fig. 3, the flow of the proposed RFU partitioning methodology is shown. The input is the code mapped on the RFU which is described in a high-level language like C/C++. As previously mentioned, research activities [10, 11, 21, 23] have shown that basic blocks (BBs) inside loop structures represent a significant portion of the execution time in DSP and multimedia applications. The term basic block expresses a sequence of operations (instructions) with no branches into or out of the middle. At the end of each basic block there is a branch instruction that controls which basic block executes next. The basic block is a compiler construct that represents the instructions composing a Data Flow Graph (DFG) representation. The proposed RFU partitioning methodology focuses on finding kernels (critical basic blocks) of the application segments selected for execution on the RFU. These kernels are executed on the coarse-grain reconfigurable hardware so that the execution time of the application segments meets the timing constraints.

In *step 1*, the Control Data Flow Graph (CDFG) representation is created from the source code. This representation is extensively used in mapping applications on reconfigurable hardware. The partitioning method utilizes a hierarchical CDFG [28] 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 DFGs. For constructing the CDFG, the SUIF2 [34] and MachineSUIF [35] compiler infrastructures are utilized. Proper compiler passes have been developed for the automation of the CDFG creation. The CDFG is input to the steps of mapping for fine and coarse-grain reconfigurable hardware and to the partitioning engine.

In *step 2*, the CDFG is mapped on the fine-grain hardware and the execution time is calculated. If the execution time of the application code mapped on the RFU meets the

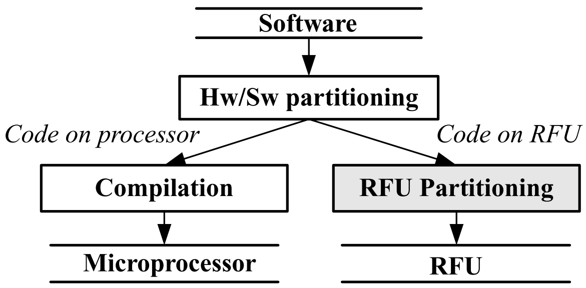


Fig. 2 General design flow for hybrid reconfigurable systems

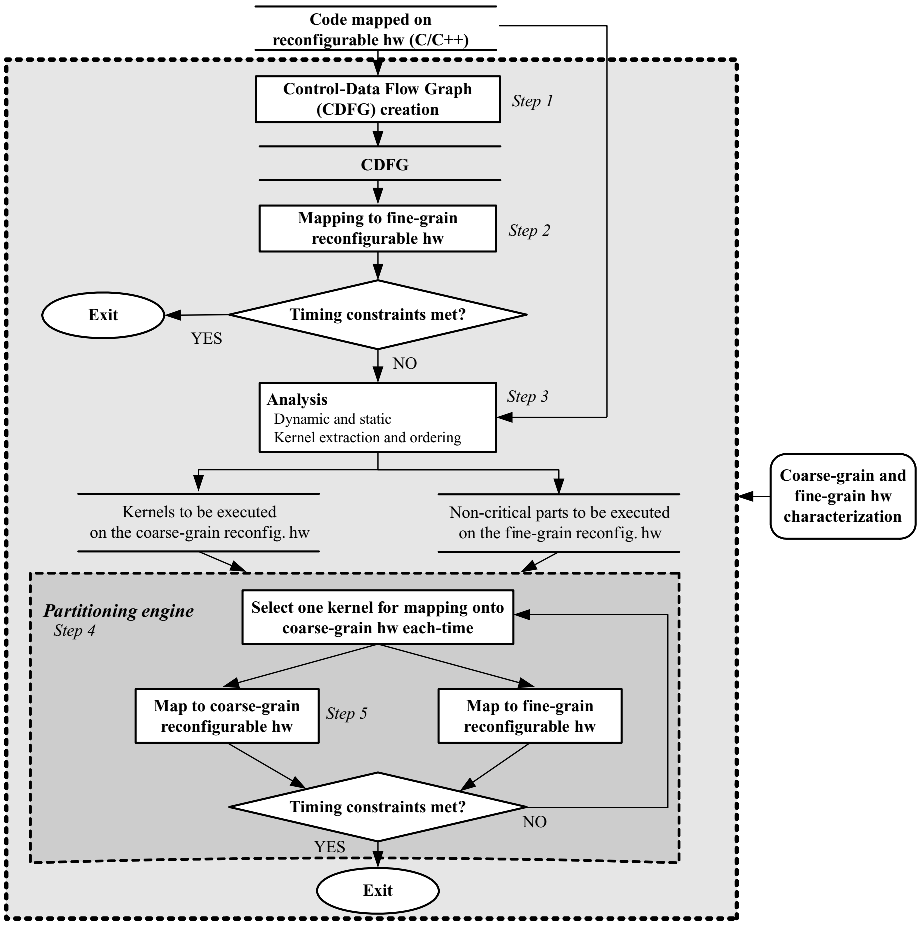

Code mapped on reconfigurable hw (C/C++)
Control-Data Flow Graph (CDFG) creation
Step 1
CDFG
Mapping to fine-grain reconfigurable hw
Step 2
Exit
Timing constraints met?
YES
NO
Analysis
Dynamic and static
Kernel extraction and ordering
Step 3
Kernels to be executed on the coarse-grain reconfig. hw
Non-critical parts to be executed on the fine-grain reconfig. hw
Coarse-grain and fine-grain hw characterization
Partitioning engine
Step 4
Select one kernel for mapping onto coarse-grain hw each-time
Map to coarse-grain reconfigurable hw
Step 5
Map to fine-grain reconfigurable hw
Timing constraints met?
NO
YES
Exit

Fig. 3 Flow of the RFU partitioning methodology

timing constraints, then the methodology exits, since there is no need to continue with the next steps, i.e. to partition the application onto fine and coarse-grain hardware. If the timing constraints are not satisfied, then we proceed to *step 3*, which is the analysis procedure.

In the analysis procedure, the application's source code is processed, for identifying the kernels, which are the candidates to be mapped onto the coarse-grain hardware. The rest of the application code is mapped onto the fine-grain logic of the architecture. The identification of kernels is a combination of dynamic and static analysis. The kernels are ordered in decreasing order of computational complexity. The analysis procedure is performed at a smaller level (the basic block level) and not at the loop level as in profiling tools of previous works [10, 11].

In the partitioning engine (*step 4*), kernels are moved one by one for execution (and thus acceleration) on the coarse-grain reconfigurable hardware. After this movement, the execution time of the application is calculated to check if the application's timing constraints are met. For computing the execution time, the mapping to the fine and coarse-grain hardware takes place.

The mapping to the coarse-grain reconfigurable logic of the architecture is the *step 5* of the proposed methodology. If there is still a violation in the execution time of the application code mapped on the RFU, the procedures of moving kernels to the coarse-grain hardware and mapping the parts of the application onto the fine and coarse-grain reconfigurable harware, are repeated until the timing constraints are satisfied.

Since the mapping procedures for both types of reconfigurable hardware determine the execution time, we propose appropriate algorithms to perform these procedures. However, the proposed method is parametric in respect to the mapping algorithms for both types of reconfigurable hardware [5, 7, 8, 9], since it is interested in knowing the execution times of application's parts on the reconfigurable blocks. Furthermore, the partitioning method is parametric to the type of fine and coarse-grain reconfigurable hardware, as the mapping algorithms abstract the hardware by considering resource constraints, timing and area characteristics. In the following sections, we describe the analysis process, the mapping procedures and the partitioning engine.

3.3. Analysis procedure

The analysis step identifies the kernels of the input application code mapped on the reconfigurable hardware and provides the input to the partitioning engine block, as it is shown in Fig. 3. The kernels are candidates to be mapped onto the coarse-grain reconfigurable hardware, while the non-critical parts of the application are executed on the fine-grain hardware. The inherent computational complexity (counts of basic operations and memory accesses) is a meaningful measure to identify critical basic blocks. This information can be obtained through a combination of: (a) dynamic analysis, and (b) static analysis, within basic blocks of the input specification. For the analysis procedure we have developed scripts in Lex [29], which is a lexical analyzer used for parsing the input code.

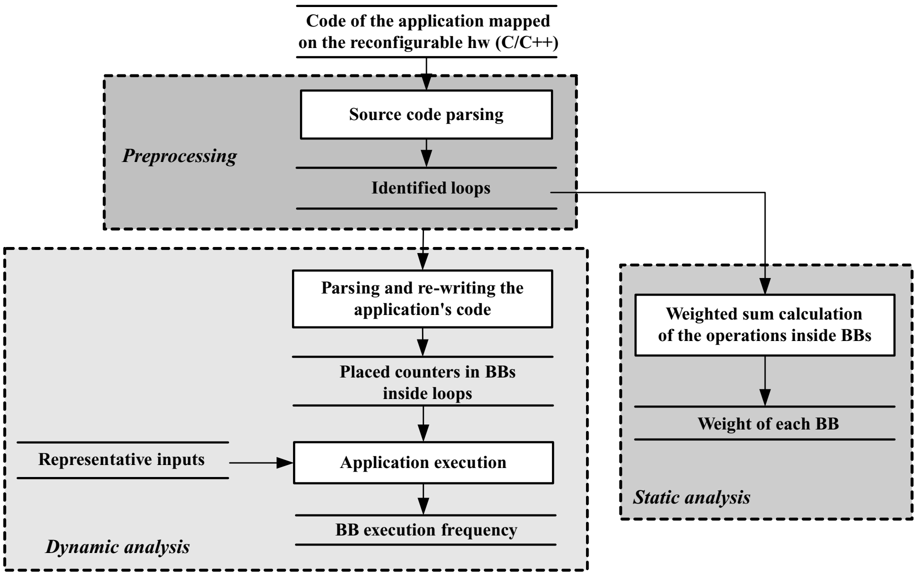

Code of the application mapped on the reconfigurable hw (C/C++)
Source code parsing
Preprocessing
Identified loops
Parsing and re-writing the application's code
Placed counters in BBs inside loops
Representative inputs
Application execution
BB execution frequency
Dynamic analysis
Weighted sum calculation of the operations inside BBs
Weight of each BB
Static analysis

Fig. 4 Diagram of the dynamic and static analysis

The diagram of the static and dynamic analysis procedures is illustrated in Fig. 4. At the *preprocessing* sub-step, loop structures (*for*, *while* and *do-while*) are identified in the source code. The identified loops are inputs to both dynamic and static analysis procedures. For the *dynamic* analysis, Lex scripts were developed for re-writing the application's source code by placing a counter at each basic block inside every loop. Then, the modified source code (after counter placement) is compiled and executed with input vectors that represent the typical operation of the application. The placed counters give the execution frequencies for all basic blocks inside loops of the input source code. Lex is also used for the *static* analysis. A Lex script has been developed for identifying the type of operations inside each basic block. Operations in a basic block do not have a uniform cost, thus a weighted sum of the operations composing a specific basic block is calculated using the same Lex script. The weight of an operation, e.g. a multiplication one, is related to the delay typically required for the execution of this operation. For example, a multiplication operation is assigned with a larger weight than an ALU one.

The total weight, representing the computational complexity of a basic block, is computed as the product of the basic block's execution frequency *(exec_freq)* times the weighted sum of the operations of this basic block (*bb_weight*), i.e.:

$$total\_weight = exec\_freq \cdot bb\_weight \quad (1)$$

The *exec_freq* is reported from the dynamic analysis, while the *bb_weight* parameter from the static analysis. After all critical basic blocks in the input source code have been identified, an ordering of these critical basic blocks takes place. These kernels are sorted in descending order of computational complexity. Thus, the first kernel which is going to be mapped onto the coarse-grain hardware, if the execution requirement is not met, is the most computational demanding one.

The developed analysis procedure makes sure that the kernels consist of word-level operations that better match the granularity of the coarse-grain reconfigurable hardware and not by bit-levels ones which are more efficiently executed on the FPGA. Additionally, computationally intensive basic blocks are not considered as kernels if they consist of operations like divisions or square roots which are characterized as exceptional since they are not supported by the functional units of our coarse-grain reconfigurable hardware and of existing architectures [4, 5, 9]. Those basic blocks are going to be mapped on the FPGA. Thus, the developed flow ensures that the detected kernels will be composed by word-level operations that they can be implemented on the functional units of the coarse-grain reconfigurable hardware (e.g. like 16-bit multiplications).

3.4. Mapping onto fine-grain reconfigurable hardware

The considered mapping procedure for the fine-grain reconfigurable hardware is based on a temporal partitioning algorithm. Temporal partitioning resolves the hardware implementation of an application that does not fit into the FPGA hardware by time-sharing the device in a way that each partition fits in the available hardware resources, i.e. the CLBs of an FPGA. Then, the partitioned application is executed by time-sharing the device such that the initial functionality is maintained. Time-sharing is achieved through the dynamic reconfiguration of the device which is a mechanism supported by modern FPGAs, either commercial [17, 18], or academic ones [21, 30]. A temporal partitioning procedure results in the concept of *virtual hardware* [31]. The importance of the temporal partitioning (and respectively of the virtual hardware notion) has been demonstrated with large and computationally complex applications [32, 33] although the reconfiguration time of the considered FPGAs was relatively high. The mapping procedure in this paper aims in efficiently exploiting the virtual hardware concept. We notice that there are mapping methodologies for FPGAs which consider static reconfiguration of the FPGA [7, 8]. However, those methodologies approach FPGAs the same way that they approach ASICs; thus the dynamic reconfiguration capabilities of the FPGAs are not exploited. For example, when an application's part does not fit on the FPGA it is discarded by those methods from execution on the FPGA, something that does not occur in the temporal partitioning method. Additionally, it has been shown in [33] that a temporal partitioning based mapping can achieve better execution times compared to a mapping that considers static reconfiguration of the FPGA device.

3.4.1. Description of the mapping procedure

The diagram of the mapping procedure for fine-grain reconfigurable hardware is given in Fig. 5. The input is the DFG of a non-critical basic block which will be executed on the FPGA. We mention that the mapping method handles CDFGs by iteratively mapping the DFGs composing the CDFG. If, for example, a CDFG contains more than one basic block (DFG), then its execution cycles are the sum of the execution cycles of the DFGs comprising it. Furthermore, if the CDFG represents a loop that has no conditional statements inside (i.e. it is composed by one basic block), then its execution cycles is the loop body's execution cycles times the number of iterations of the loop.

Afterwards, optimizations are automatically applied to the DFG, like dead code elimination, common sub-expression elimination and constant propagation. Furthermore, to accommodate the implementation of operations like divisions or square root computations on the FPGA, these operations are transformed into series of primitive operations, as multiplications and ALU-type ones. For example, the divisions are transformed to shifts, while a square root computation can be decomposed to simpler operations using a method, like the Friden algorithm [36].

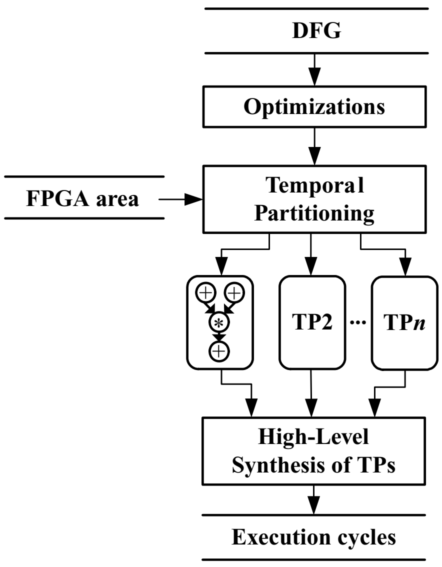


Fig. 5 Diagram of the mapping procedure for FPGAs

The temporal partitioning algorithm partitions the DFG into segments under the constraint of the FPGA area. The temporal partitioning procedure has to exploit the operation parallelism of the DFG for reducing the execution time and thus improving performance. For each temporal partition (TP), *high-level synthesis* is performed for estimating its execution cycles. For minimizing the execution time of each partition of the input DFG, we have implemented the As Soon As Possible (ASAP) scheduling algorithm [28]. This type of scheduling can be performed since the temporal partitioning algorithm does not consider resource sharing and all the operations of the partition fit on the FPGA.

The resource sharing usually does not lead to better solution in terms of performance for designs targeting FPGAs. Multiplexers are used to provide with the proper inputs the functional units when resource sharing takes place (e.g. as in list and Force Directed based synthesis systems [28]). However, multiplexers can occupy larger FPGA area than the area of the functional resources. Most importantly, the propagation delay of a multiplexer is added to the delay of the functional units (like a multiplier one) which results in the reduction of the clock frequency in each control-step [28] of the schedule. The clock frequency of the synthesized circuit is smaller than in the case of a circuit with no multiplexers; thus performance is decreased. Additionally, a large number of multiplexers is usually instantiated in order to maximize the sharing of functional units in existing behavioural synthesis systems [28]. This complicates the placement and routing (P&R) on the FPGA which has an effect in not achieving the best as possible clock frequency due to poor P&R. In high-level synthesized designs targeting FPGAs, it is better to use an extra functional unit than perform resource sharing due to the aforementioned reasons. Thus, we have considered ASAP-based temporal partitioning and ASAP scheduling in each temporal partition for avoiding resource sharing which usually degrades the performance of behavioural synthesized designs on FPGAs.

According to the application's data and control-flow, the appropriate partition is loaded to the FPGA. Data memories are used for storing the input and output values among the temporal partitions. For example, local data memories embedded in the FPGA, as in devices of [17, 18], can be used. We note that for each temporal partition, full reconfiguration of the FPGA is performed. The reconfiguration time has the same value for each partition of a basic block mapped onto FPGA and it is added to the execution time of each basic block's partition. Finally, the mapping procedure reports the execution cycles of the partitioned DFG and the configuration of the fine-grain reconfigurable logic.

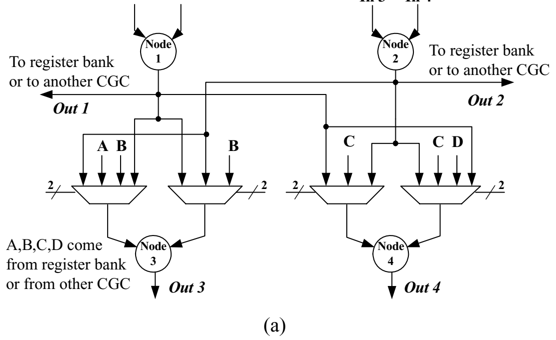
The considered temporal partitioning algorithm classifies the nodes (operations) of the input DFG according to its ASAP levels [28]. The ASAP levels expose the parallelism hidden in the DFG, i.e. all the DFG nodes with the same level can be considered for parallel execution. The approach followed is that the nodes are executed in increasing order relative to their ASAP levels. Such an approach also exploits the maximum operation parallelism from the input DFG which leads in faster as possible execution on the FPGA. For the temporal partitioning of the DFG, the ASAP level of the DFG node u_i, the FPGA area - $area(u_i)$ - occupied by each node and the area A_{FPGA} available for mapping the DFG operations on the FPGA are considered. The algorithm traverses each node of the DFG, level by level, and assigns them to a partition. The DFG nodes are assigned to partitions numbered 1 and beyond. All the nodes from level 1 to the maximum level of any node in the DFG are traversed. Nodes of the same ASAP level are placed in a single partition. If the available area in the FPGA hardware is exhausted, then the nodes are assigned to the next partition. If the nodes in the current ASAP level are all assigned to a partition, then the next level nodes are considered for that partition. Initially, a partition has no nodes. As the $area(u_i)$ and A_{FPGA} are parameters in our algorithm, the specific temporal partitioning algorithm is retargetable to the type of FPGA.

3.5. Mapping onto coarse-grain reconfigurable hardware

3.5.1. Coarse-grain reconfigurable data-path

For the coarse-grain hardware, the high-performance coarse-grain reconfigurable data-path and the mapping methodology presented in [27] are considered in this work. This data-path consists of a set of hardwired Coarse-Grain Components (CGCs), a reconfigurable interconnection network, and a register bank. A CGC unit is an $n \times m$ array of nodes, where n is the number of rows and m the number of columns. In Fig. 6a, such a CGC (called hereafter as 2×2 CGC) with 2 nodes per row and 2 nodes per column is illustrated. The 2×2 CGC consists of four nodes whose interconnection is shown in Fig. 6a, four inputs (*in1*, *in2*, *in3*, *in4*) connected to the register bank, four additional inputs (A, B, C, D) connected to the register bank or to another CGC, two outputs (*out1*, *out2*) also connected to the register bank and/or to another CGC, and two outputs (*out3*, *out4*) whose values are stored in the register bank. An $n \times m$ CGC has an analogous structure. Particularly, the first-row nodes obtain their inputs from the register bank. All the other CGC nodes obtain their inputs from the register bank and/or a row with a smaller index from the same and/or another CGC. For the case of the CGC outputs, the last-row nodes store the results of their operations to the register bank. All the other nodes can give their results to the register bank, to the same CGC or to another CGC in the data-path.

Each CGC node contains two 16-bit functional units that are a multiplier and an ALU as shown in Fig. 6b. The ALU performs shifting, arithmetic (add/subtract), and logical operations. Each time either the multiplier or the ALU is activated according to the control signals *Sel1*, *Sel2* and *Sel3*, as shown in Fig. 6b. The flexible interconnection among the nodes inside a CGC allows in easily realizing any desired operation combination (like a multiply-accumulate operation) by properly configuring the existing steering logic (i.e. the multiplexers and the tri-state buffers). Additionally, due to the CGC data-path's features, the stages of the mapping methodology are accommodated by simple, yet efficient algorithms. An average performance improvement of 44%, relative to existing high-performance data-path, was achieved with the usage of the CGC data-path [27].


In 1
In 2
In 3
In 4
Node 1
Node 2
Node 3
Node 4
To register bank or to another CGC
Out 1
To register bank or to another CGC
Out 2
A B
B
C
C D
2
2
2
2
A,B,C,D come from register bank or from other CGC
Out 3
Out 4
(a)

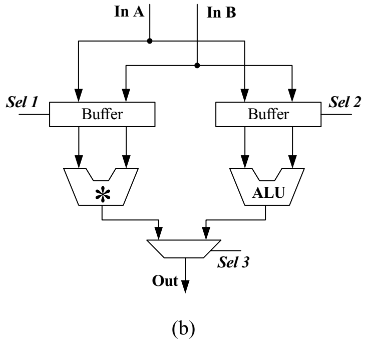


Fig. 6 (a) Architecture of the 2×2 CGC, (b) CGC node architecture

3.5.2. Description of the mapping algorithm

The diagram of the mapping procedure for the CCG data-path is shown in Fig. 7. The input to the mapping algorithm is the DFG of a candidate basic block (kernel) to be mapped on the CGC data-path. The input DFG is scheduled using the developed scheduler for the CGC data-path. In our case, scheduling is a resource-constrained problem with the goal of execution cycles minimization, since the number and type of CGCs (e.g. three 2×2 CGCs) in the data-path is input to the mapping algorithm. A proper list-based scheduler has been developed. The priority function [28] of the list scheduler is derived by properly labeling the DFG nodes. Particularly, the nodes are labeled with weights of their longest path to the sink node of the DFG, and they are ranked in decreasing order. The most urgent operations are scheduled first. The resource constraints for the list-based scheduler are determined by the total number of CGC nodes at the first rows of all the CGCs in the data-path. If there are p $n \times m$ CGCs in the data-path, there are $p \cdot m$ nodes in the first rows, since each CGC row consists of m nodes. Thus, $p \cdot m$ operations can be executed in parallel at each clock cycle of the schedule. For example, if there are three 2×2 CGCs in the data-path, six operations can be executed in parallel at every cycle of the schedule.

Due to the features of the introduced CGC-based data-path, a simple but effective algorithm is used to perform binding. The pseudo-code of the binding algorithm is illustrated in Fig. 8. The binding algorithm maps the DFG operations to the CGC nodes in a row-wise manner, using the *map_to_CGC*() function. We define a term called *CGC_index* that represents the current level of CGC nodes that bind the DFG operations. The *CGC_index* takes the values from 0 and n-1, as the CGC consists of n levels (rows) of operations. After CGC binding, the overall latency of the DFG is measured in clock cycles having period T_{CGC} that is set for having unit execution delay for the CGCs.

The mapping flow outputs the execution cycles and the configuration of the CGC data-path. The mapping procedure handles CDFGs by iteratively mapping the DFGs composing the CDFG. For more detailed description about the CGC data-path and the respective mapping algorithm, the reader is referred to [27].

3.6. Partitioning engine

The partitioning engine moves kernels one by one for execution onto the coarse-grain reconfigurable hardware until the timing constraints of the application code mapped on the RFU are satisfied. After the movement of each kernel to the coarse-grain hardware, the execution time of the application is calculated to check if these constraints are met. The mapping procedures to the fine and coarse-grain hardware are required for computing the execution times.

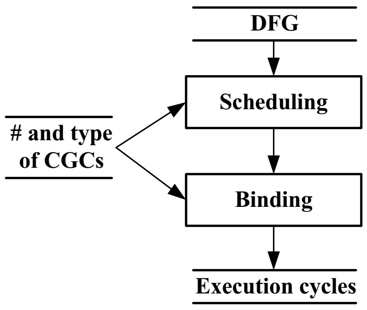


Fig. 7 Mapping procedure for the coarse-grain reconfigurable hardware

```
while (the DFG is not fully bound)
  for the number of CGCs
   for (CGC_index=0; CGC_index <n; CGC_index ++)
    while (col_idx < number of ops in a row && col_idx < number of DFG nodes not covered)
        map_to_CGC(DFG_node, CGC_index, col_idx)
    end while;
   end for;
  end for;
end while;
```

Fig. 8 CGC binding algorithm

Due to the mutual exclusive execution of the fine and coarse-grain reconfigurable hardware, the total time for executing an application onto the RFU of the targeted platform is:

$$t_{total} = t_{FPGA} + t_{coarse} \quad (2)$$

where t_{FPGA} is the execution time of non-critical basic blocks on the FPGA and t_{coarse} is the execution time of the kernels on the coarse-grain reconfigurable hardware. The communication time between the fine and coarse-grain reconfigurable hardware is included in the t_{FPGA} and in the t_{coarse}.

The t_{coarse} equals:

$$t_{coarse} = \sum_i t_{to\_coarse}(BB_i) \cdot Exec\_freq(BB_i) \quad (3)$$

where t_{to_coarse} is the time required for executing the basic block BB_i on the coarse-grain reconfigurable hardware, and $Exec\_freq(BB_i)$ is the execution frequency of the BB_i. Similarly, t_{FPGA} equals:

$$t_{FPGA} = \sum_i t_{to\_FPGA}(BB_i) \cdot Exec\_freq(BB_i) \quad (4)$$

where t_{to_FPGA} includes the reconfiguration time of the FPGA for all the generated temporal partitions after the mapping of the basic block BB_i.

4. Experiments

We have developed a prototype framework in C++, also utilizing academic tools [29, 34, 35], to implement the flow of the proposed partitioning method (Fig. 3). The execution times of the application parts mapped on the RFU were estimated using this prototype framework.

4.1. Experimental set-up

In this paper, we apply the proposed partitioning method to five real-world applications written in C language. These applications are: (a) the baseband processing of an IEEE

802.11a OFDM transmitter [12], (b) a cavity detector which is a medical image processing application [13], (c) a wavelet-based image compressor, that its source code is derived from the Honeywell benchmarks [15], (d) a video compression technique, called Quadtree Structured Difference Pulse Code Modulation (QSDPCM) [14], and (e) a JPEG-compliant still image encoder [16]. The OFDM transmitter and the JPEG encoder were developed by the AMDREL project partners [37], while the cavity detector and the QSDPCM are in-house benchmarks.

The clock cycle period of the reconfigurable part of the hybrid architecture is set to the clock period of the FPGA. We have considered that the clock cycle period T_{FPGA} of the FPGA hardware is two times larger than the CGC data-path's clock period T_{CGC}, i.e.

$$T_{FPGA} = 2 \cdot T_{CGC} \quad (5)$$

These two clock periods remain constant for the five benchmark applications partitioned on the reconfigurable logic of the platform. The coarse-grain reconfigurable hardware is clocked at a constant frequency as in existing coarse-grain architectures [4, 5, 24]. On the contrary, when different designs are synthesized from an RTL description in FPGA logic with a tool like Synplify Pro [38], different delays are reported and consequently the FPGA can be clocked at different frequencies. However, with our temporal partitioning based methodology, we can set the clock frequency of the FPGA to a constant value. As it is mentioned in section 3.4.1, ASAP scheduling is performed in each generated temporal partition. In high-level synthesis of applications, the clock period of each control-step remains constant and the execution time equals the number of control-steps (cycles) times the clock period [28]. The clock period is typically set to accommodate the delay of a functional unit. In our experiments, the clock period of the FPGA is defined by the delay of an ALU unit. In this case, we have found by experimentation that a multiplication operation endures two clock cycles, while the ALU operations have unit execution delay. We mention that the basic blocks of the five applications consist of arithmetic operations of type ALU and multiplication (e.g. no divisions are present). Also, the total number of ALU operations was considerably larger than the multiplications in these applications. We did not set the clock period of the FPGA to the delay of a multiplier since this would have resulted in large slack times wasted in control-steps executing ALU operations; thus for reducing the slack times we set the clock period to accommodate the ALU's delay.

We have synthesized an RTL VHDL description of a CGC data-path composed by three 2×2 CGCs to a 0.13 μm standard cell CMOS technology and we found that the clock frequency can be set to 200 MHz. We have also synthesized a 16-bit ALU, supporting 12 operations, on a Xilinx Virtex FPGA and the reported maximum clock frequency from the Synplify Pro was approximately 100 MHz. Due to these two experimental observations, we set $T_{FPGA} = 2 \cdot T_{CGC}$. However, there are coarse-grain reconfigurable architectures that can be clocked at a higher frequency than the 200 MHz as in the case of the reconfigurable DSP core of [24] which can be clocked up to 500 MHz. Thus, the ratio T_{FPGA}/T_{CGC} can be larger than two.

For the experiments, we assume two different FPGA devices. For the first FPGA device (*FPGA1*), the area is equal to 1500 units, i.e. $A_{FPGA} = 1500$. This area value is approximately equal to the number of CLB slices in the Xilinx Virtex-II XC2V250 device [17]. For the second FPGA (*FPGA2*), the area equals 5000 units ($A_{FPGA} = 5000$), and it approximately corresponds to the number of CLB slices in the XC2V1000 device. We also consider two cases of CGC-based data-paths composed by: (a) two 2×2 CGCs (*CGC1* case) and (b) three 2×2 CGCs (*CGC2* case). Thus, four different combinations of the FPGA and CGC data-path are assumed in the experimentation. In [27], we have found that CGCs with a value of $2 \leq n \leq 3$ and $2 \leq m \leq 3$ are adequate to be used for improving performance. This is the reason why we have considered 2×2 CGC units in this experimentation. The microprocessor of the hybrid platform is considered to be a MIPS32 4KP [39] clocked at 100 MHz.

We have assumed that the full reconfiguration of the FPGA endures 5 clock cycles. This reconfiguration time was also reported for the FPGA part of the academic Garp architecture [21]. For the contemporary commercial FPGAs [17, 18], the full reconfiguration time is usually a few milliseconds. However, with reconfiguration caches or other proper developed mechanisms, as in [21, 30], the full reconfiguration can last few FPGA cycles. On the other hand, the CGC data-path is reconfigured in a cycle-by-cycle basis.

The applications' execution clock cycles are derived by using the following inputs: (a) 6 payload symbols for the OFDM transmitter at a 54 Mbps rate, (b) an image of size 640×400 bytes for the cavity detector, (c) two video frames of size 176×144 bytes each for the QSDPCM, (d) an image of size 512×512 bytes for the wavelet-based image compressor, and (e) an image of size 256×256 bytes for the JPEG encoder. The timing constraints to be met, for the application parts mapped on the RFU, are: (a) 50,000 clock cycles for the OFDM transmitter, (b) 8,000,000 cycles for the cavity detector, (c) 5,000,000 cycles for the image compressor, (d) 16,000,000 cycles for the QSDPCM, and (e) 6,000,000 cycles for the JPEG encoder.

4.2. Experimentation

4.2.1. Execution times of the overall applications on the hybrid platform

Results are provided in this section from executing the complete applications on the hybrid platform of Fig. 1. A straightforward hardware/software partitioning approach is utilized where basic blocks contributing more than 5% to the total execution time of the application were selected for execution on the reconfigurable logic. Such kind of hardware/software partitioning decision has been previously used in works considering the partitioning in microprocessor/FPGA systems [11, 23] and it has been shown to achieve very good speedups. The rest of the code of the five applications is executed on the MIPS32 4KP processor. The number of basic blocks (BBs) of each application decided to be mapped on the RFU is shown in Table 1.

The execution times and overall application speedups for the five applications are presented in Table 2. The execution cycles of software on the MIPS are estimated using the instruction-set simulator MIPSsim^{TM} of the MIPS Software Toolkit [39]. $Cycles_{MIPS}$ represents the execution cycles of the software of the whole application on the MIPS processor. $Cycles_{BB\_MIPS}$ corresponds to the execution cycles of the basic blocks, selected for mapping on the RFU, on the MIPS. $Cycles_{BB\_FPGA}$ is the number of cycles, for executing the selected basic blocks on the *FPGA1* as reported by the developed FPGA mapping procedure.

Table 1 Number of BBs selected for execution on the reconfigurable hardware

Application	# of BBs on RFU
OFDM trans.	18
Cavity det.	11
Compressor	19
QSDPCM	25
JPEG enc.	15

Table 2 Execution cycles and overall application speedups by executing the applications on the hybrid platform. The RFU is composed by the FPGA1

Application	$\text{Cycles}_{\text{MIPS}}$	$\text{Cycles}_{\text{BB\_MIPS}}$	$\text{Cycles}_{\text{BB\_FPGA}}$	App. Sp.
OFDM trans.	2,452,875	1,706,275	101,564	2.9
Cavity det.	931,663,757	456,697,920	27,184,400	1.9
Compressor	257,513,945	144,941,251	11,149,327	2.1
QSDPCM	1,515,988,093	527,309,461	51,697,006	1.5
JPEG enc.	149,057,881	100,954,390	8,343,338	2.6
Average				**2.2**

App. Sp. represents the estimated overall application speedup after executing the basic blocks on the FPGA. The application speedup is calculated as:

$$App.Sp. = Cycles_{MIPS}/(Cycles_{MIPS} - Cycles_{BB\_MIPS} + Cycles_{BB\_FPGA}) \quad (6)$$

Such values of speedups as the ones presented in Table 2 are considered as significant in existing works [11, 23]. Furthermore, it is deduced that the performance of the basic blocks is largely improved when they are executed on the *FPGA1* instead on the MIPS32 processor. Although, the achieved application speedups are important, the timing constraints of the applications code on the RFU are not met as it is inferred by comparing the timing constraints reported in section 4.1 with the $Cycles_{BB\_FPGA}$ values. This is also the situation when the *FPGA2* is used in the RFU, although it employs more CLB slices than the *FPGA1*. So, the proposed partitioning methodology for the mixed granularity reconfigurable logic has to be utilized for meeting the timing requirements by accelerating kernels on the CGC reconfigurable data-paths.

4.2.2. Results for partitioning applications on the RFU

This section presents the results by applying the proposed partitioning methodology to the basic blocks selected to be executed on the reconfigurable logic of the platform, for satisfying the timing specifications.

We have executed the partitioning engine, using the developed framework, and we have selected the necessary basic blocks to be mapped on the CGC data-path for satisfying the specified timing constraints. The results of the analysis procedure are given in Table 3. For the OFDM transmitter the BB3, BB8 and BB10 are required for meeting the specification requirements. For the cavity detector the BB2, BB4, BB5 and BB9 are selected for mapping onto coarse-grain hardware, while for the wavelet-based image compressor the BB9, BB10, BB11 and BB12 are selected. For the QSDPCM the BB5, BB11, BB13 and BB14 are characterized as kernels. Finally, the BB1, BB2 and BB6 of the JPEG encoder are selected for satisfying the specified timing constraint. The aforementioned BBs are the kernels of each application. The kernels of the five applications are located in loops and they consist of word-level operations that match the granularity of the CGCs.

Table 4 reports the execution clock cycles (*Initial Cycles*) of the applications' parts on the FPGAs, without accelerating kernels on the CGC data-path, for the two considered FPGA devices. These cycles include the reconfiguration time of the FPGA. For meeting the timing constraints, the kernels of the five applications were mapped on the *CGC1* and *CGC2* data-paths. For each application, the number of clock cycles resulting after the partitioning

Table 3 Ordered total weights of basic blocks

Basic Block number	Execution frequency	Weighted sum	Total weight
	OFDM transmitter		
10	336	115	38,640
8	1,200	25	30,000
3	864	24	20,736
5	370	12	4,440
2	400	10	4,000
	Cavity detector		
2	95,222	40	3,808,880
4	95,222	40	3,808,880
5	63,481	32	2,031,392
9	63,481	32	2,031,392
7	64,000	12	768,000
	Image compressor		
11	21,280	96	2,042,880
9	21,280	80	1,702,400
12	21,504	48	1,032,192
10	21,504	48	1,032,192
5	27,533	20	550,660
	QSDPCM		
13	456,192	30	13,685,760
11	316,800	34	10,771,200
5	256,608	29	7,441,632
14	228,096	32	7,299,072
15	114,048	25	2,851,200
	JPEG encoder		
6	22,189	48	1,065,072
2	8,192	85	696,320
1	8,192	83	679,936
12	8,192	40	327,680
8	7,732	32	247,424

and the acceleration on the *CGC1* and *CGC2* architectures is given (*Final Cycles* column). The number after each application's name refers to the CGC data-path; the (1) corresponds to the *CGC1* case, while the (2) to the *CGC2* data-path. Also, in Table 4 the speedup of the partitioned solutions relative to the all-FPGA case is given. The speedup equals to the ratio *Initial_Cycles* / *Final_Cycles*.

It is clear from the results of Table 4, that by choosing costly BBs to be mapped on the coarse-grain reconfigurable hardware, application's performance is largely improved and the timing constraint of the application code mapped on the RFU is satisfied in every case. These results prove the effectiveness of both the proposed partitioning methodology and of the developed software framework.

The speedups (when refer to the same FPGA device) are slightly greater when the *CGC2* data-path (three 2×2 CGCs) is used for executing kernels relative to the *CGC1* case. This is due to the larger number of arithmetic units in the *CCG2* that better exploit the available operation parallelism in the applications than the *CGC1* data-path. The results of Table 4 also illustrate that the speedups due to the partitioning are greater when a smaller FPGA device is used in the hybrid platform. The average speedup after partitioning, when the *FPGA1* is

Table 4 Execution cycles and speedups by using the partitioning methodology

	$A_{FPGA} = 1500$			$A_{FPGA} = 5000$		
Application	*Initial cycles*	*Final cycles*	*Speedup*	*Initial cycles*	*Final cycles*	*Speedup*
OFDM (1)	101,564	37,404	2.7	97,796	39,472	2.5
OFDM (2)		37,164	2.7		39,232	2.5
Cavity (1)	27,184,400	7,251,366	3.7	26,930,476	7,480,688	3.6
Cavity (2)		6,870,480	4.0		7,049,863	3.8
Compressor (1)	11,149,327	4,139,023	2.7	10,743,199	4,073,375	2.6
Compressor (2)		4,053,007	2.8		3,987,359	2.7
QSDPCM (1)	51,697,006	13,671,502	3.8	35,033,326	12,746,446	2.7
QSDPCM (2)		13,443,406	3.8		12,518,350	2.8
JPEG (1)	8,343,338	5,379,562	1.6	7,610,154	5,012,970	1.5
JPEG (2)		5,362,410	1.6		4,897,206	1.5
Average			**2.9**			**2.6**

used in the hybrid platform, is 2.9. For the larger *FPGA2* device the average speedup for the five applications is 2.6. We can state that this observation is analogous to the case of existing hardware/software partitioning works for microprocessor/FPGA systems [11, 23] where the speedups are greater when the FPGA logic is coupled with a low-performance instruction-set processor. In our partitioning scenario, the hardware used for improving performance is the CGC data-path, while the FPGA is used for the "initial" execution (before partitioning) of the application which is like the case of the instruction-set processor in processor/FPGA platforms.

4.2.2.1. Effect of the FPGA reconfiguration time on the performance improvements We have performed an experiment in respect to the reconfiguration time of the FPGA. We have assumed two different times for the full reconfiguration of the FPGA: (a) 5 cycles which is the case of the results in Table 4, and (b) 1,000,000 cycles which is in the order of milliseconds, as in commercial FPGAs [17, 18]. In this experiment, the RFU of the hybrid platform includes the *FPGA2* ($A_{FPGA} = 5000$) and the *CGC1* (two 2×2 CGCs). Fig. 9 shows the speedups over the all-FPGA execution for the two reconfiguration time scenarios. The average speedup for the *FPGA2*/*CGC1* case is equal to 3.0 for the reconfiguration time of 10^6 cycles, where for the case of 5 cycles required for reconfiguration the average speedup is 2.6. We mention that for the reconfiguration time of 10^6 cycles, the average speedup for all the cases of FPGAs and CGC data-paths is 3.2. On the other hand, it is inferred from Table 4 that the average speedup for all cases is 2.8 ($= (2.9 + 2.6)/2$) when the reconfiguration time is equal to 5 cycles.

By comparing the speedups for these two FPGA reconfiguration times, we infer that even for small reconfiguration times (which can occur in academic FPGAs [21, 30]) we achieve speedups which are not significantly smaller than the ones reported with a more realistic FPGA reconfiguration time. We have selected as the main experimentation scenario in this paper, the FPGA reconfiguration time to be equal to 5 cycles for illustrating the fact that the proposed partitioning flow can achieve important speedups (as shown in Table 4) even in this reconfiguration time scenario which is not considered as a realistic situation for the majority of contemporary FPGAs [17, 18].

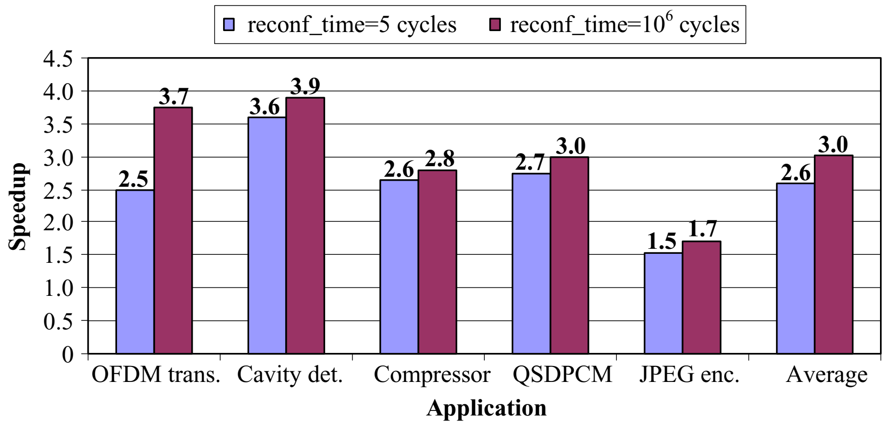


Fig. 9 Comparing the speedups over the all-FPGA execution for two different reconfiguration times ($A_{FPGA} = 5000$, two 2×2 CGCs)

5. Conclusions-Future work

A methodology for partitioning application code between fine and coarse-grain reconfigurable blocks of a hybrid granularity architecture was presented. The methodology was validated using five real-life applications. Although the methodology is parametric in respect to the mapping procedures used, specific mapping algorithms for the fine and coarse-grain reconfigurable blocks were also assumed. The experiments showed that the timing constraints of application code mapped on the reconfigurable hardware can be satisfied by proper partitioning. The speedup improvement relative to the all fine-grain solution ranges from 1.5 to 4.0 in the considered experimentation. Future work focuses on partitioning application code on the RFU for satisfying energy consumption constraints.

Acknowledgments This research has been partly funded by the Alexander S. Onassis Public Benefit Foundation. It was also conducted in part for the purposes of the AMDREL European IST project.

References

1. Hartenstein, R. A Decade of Reconfigurable Computing: A Visionary Retrospective. In *Proc. of IEEE/ACM Design Automation and Test in Europe (DATE)*, pp. 642–649, 2001.
2. Kastner, R., A. Kaplan, S.O. Memik, and E. Bozorgadeh. Instruction Generation for Hybrid Reconfigurable Systems. In *ACM Trans. on Design Automation of Electronic Systems (TODAES)*, vol. 7, no. 4, pp. 605–627, October 2002.
3. Wan, M., H. Zhang, V. George, M. Benes, A. Abnous, V. Prabhu, and J. Rabaey, Design methodology of a low-energy reconfigurable single-chip DSP system. In *Journal of VLSI Signal Processing*, vol.28, no. 1–2, pp.47–61, May–June 2001.
4. Rauwerda, G.K., P.M. Heysters, and G.J.M. Smit. Mapping Wireless Communication Algorithms onto a Reconfigurable Architecture. In the *Journal of Supercomputing*, Springer-Verlag, vol. 30, no. 3, pp. 263–282, December 2004.
5. Singh, H., L. Ming-Hau, L. Guangming, F.J. Kurdahi, N. Bagherzadeh, and 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.

6. Brisk, P., A. Kaplan, R. Kastner, and M. Sarrafzadeh. Instruction Generation and Regularity Extraction for Reconfigurable Processors. In *Proc. of Compilers, Architectures and Synthesis for Embedded Systems*(*CASES*), October 8–11, France, pp. 262–269, 2002.
7. Bazargan, K., S. Ogrenci, and M. Sarrafzadeh. Integrating Scheduling and Physical Design into a Coherent Compilation Cycle for Reconfigurable Computing Architectures. In *Proc. of ACM DAC '01*, pp. 635–640, 2001.
8. Guo, Z., B. Buyukkurt, W. Najjar, and K. Vissers. Optimized Generation of Data-path from C Codes for FPGAs. In *Proc. of ACM/IEEE DATE '05*, Munich, Germany, pp. 112–117, 2005.
9. Goldstein, S.C., H. Schmit, M. Budiu, S. Cadambi, M. Moe, and R.R. Taylor. PipeRench: a reconfigurable architecture and compiler. In *IEEE Computer*, vol. 33, no. 4, pp. 70–77, April 2000.
10. Suresh, D.C., W.A. Najjar, F. Vahid, J.R. Villareal, and G. Stitt. Profiling tools for Hardware/Software Partitioning of Embedded Applications. In *Proc. of Languages Compilers and Tools for Embedded Systems* (LCTES), pp.189–198, 2003.
11. Stitt, G., F. Vahid, and S. Nematbakhsh. Energy Savings and Speedups from Partitioning Critical Software Loops to Hardware in Embedded Systems. In *ACM Trans. on Embedded Computing Systems* (TECS), vol. 3, no. 1, pp. 218–232, Feb. 2004.
12. IEEE 802.11a Wireless LAN standard, http://grouper.ieee.org/groups/802/11/, 2006.
13. Bister, M., Y. Taeymans, and J. Cornelis. Automatic Segmentation of Cardiac MR Images. *Computers in Cardiology*, IEEE Computer Society Press, pp.215–218, 1989.
14. Strobach, P.. QSDPCM – A New Technique in Scene Adaptive Coding. In *Proc. of 4th European Signal Processing Conf.*, (EUSIPCO-88), France, pp. 1141–1144, Sep. 1988.
15. Honeywell Tech., *Adaptive Computing Systems Benchmarks*, http://www.htc.honeywell.com/projects/acsbench, 2006.
16. JPEG image compression, www.jpeg.org, 2006.
17. Virtex FPGAs, Xilinx Inc., www.xilinx.com, 2006.
18. Stratix FPGAs, Altera Inc., www.altera.com, 2006.
19. Gajski, D.D., F. Vahid, S. Narayan, and J. Gong. SpecSyn: An environment supporting the specify-explore-refine paradigm for hardware/software system design. In *IEEE Trans. on VLSI Syst.*, vol. 6, no. 1, pp. 84–100, 1998.
20. Henkel, J.. A low power hardware/software partitioning approach for core-based embedded systems. In *Proc. of the 36th ACM/IEEE Design Automation Conference* (*DAC*), pp. 122–127, 1999.
21. Callahan, T.J., J.R. Hauser, and J. Wawrzynek. The Garp Architecture and C Compiler. In *IEEE Computer*, vol. 33, no. 4, pp. 62–69, April 2000.
22. Ye, A., N. Shenoy, and P. Baneijee. A C Compiler for a Processor with a Reconfigurable Functional Unit. In *Proc. of FPGA*, pp. 95–100, 2000.
23. Stitt G. and F. Vahid. Energy Advantages of Microprocessors Platforms with On-Chip Configurable Logic. In *IEEE Design & Test of Computers*, vol. 19, no. 6, pp. 36–43, Nov.-Dec. 2002.
24. Morpho Technologies, Morpho Reconfigurable DSP (*r*DSP) core, www.morphotech.com, 2006.
25. Becker, J., R. Hartenstein, M. Herz, and U. Nadeldinger. Parallelization in Co-Compilation for Configurable Accelerators: A Host/Accelerator Partitioning Compilation Method. In *Proc. of ASPDAC '98*, Yokohama, Japan, Feb. 10–13, 1998.
26. Becker, J., A. Thomas, and M. Scheer. Datapath and Compiler Integration of Coarse-grain Reconfigurable XPP-Arrays into Pipelined RISC Processor. In *Proc. of IFIP VLSI SoC Conf.*, pp. 288–293, 2003.
27. Galanis, M.D., G. Theodoridis, S. Tragoudas, and C. E. Goutis. A Reconfigurable Coarse-Grain Data-Path for Accelerating Computational Intensive Kernels. In *Journal of Circuits, Systems and Computers* (JCSC), World Scientific Publishers, vol. 14, no. 9, pp. 877–893, August 2005.
28. De Micheli, G. *Synthesis, and Optimization of Digital Circuits*, McGraw-Hill, International Editions, 1994.
29. Levine, J.R., T. Mason, and D. Brown, *Lex & Yacc*, O' Reilly Publishers, 1995.
30. Motomura, M., Y. Aimoto, A. Shibayama, Y. Yabe, and M. Yamashina. An Embedded DRAM-FPGA Chip with Instantaneous Logic Reconfiguration. In *Proc. of IEEE FCCM '98*, pp. 264–266, 1998.
31. Long P. and H. Amano. WASMII: a Data Driven Computer on a Virtual Hardware. In *Proc. of the 1st IEEE FCCM Workshop*, California, USA, April 5–7, pp. 33–42, 1993.
32. Hudson, R.D., D.I. Lehn, and P.M. Athanas. A Run-Time Reconfigurable Engine for Image Interpolation. In *Proc. of 6th IEEE FCCM '98*, California, USA, April 15–17, pp. 88–95, 1998.
33. Kaul, M., R. Vemuri, S. Govindarajan, and I. Ouassis. An Automated Temporal Partitioning Tool for a class of DSP applications. In *Proc of Parallel Architectures and Compilation Techniques Conf.*, (PACT '98), pp. 22–27, 1998.

34. SUIF2 compiler, http://suif.stanford.edu/suif/suif2/index.html, 2006.
35. Smith, M.D. and G. Holloway. An introduction to Machine SUIF and its portable libraries for analysis and optimization. *Technical Report*, Harvard University, July 2002.
36. Crenshaw, J.W. MATH Toolkit for Real-Time Programming, CMP Books, 2000.
37. Architectures and Methodologies for Dynamic REconfigurable Logic (AMDREL) project, IST-2001-34379, http://vlsi.ee.duth.gr/amdrel/, 2006.
38. Synplify Pro, Synplicity Inc., www.synplicity.com, 2006.
39. MIPS Inc., www.mips.com, 2006.