Data-Flow Programming Paradigm for High Level Synthesis Improvement

Aimad Eddine Debbi*
Department of Electronics, Farhat Abbass University in Setif, Setif, Algeria.
Department of Computer Science, Mohamed Boudhiaf university, M'sila, Algeria.

* Corresponding author. Email: aimad_ne@yahoo.fr

Manuscript submitted June 19, 2017; accepted August 10, 2017.
doi: 10.17706/jcp.13.6.622-637

Abstract: High Level synthesis (HLS) tools are now attracting much the attention of hardware developers. Their high rates of productivity compared to hand hardware description language (HDL) coding are quite proved. In this paper much attention is accorded to the quality feature of HLS tools and we propose a total novel approach for building high quality HLS tools and an associate framework. We illustrate how the proposed policy is certifying upper quality where both speedup and throughput are ultimately improved. Those qualities are very interesting when target application is subjected to rigid time constraints and they were been ensured within the proposed tool through full exploration of intrinsic parallelism. We were guided to consider a data-flow programming paradigm where programs are transposed towards a novel representation form that has a parallel-spatial consistency. This paradigm is enabling an efficient and rigorous investigation of some important issues like Design Space Exploration, load balancing, task management and co-design challenges. We have been able to generate a soft core for the Advanced Encryption Standard (AES) cipher treated as a case study using the proposed tool. We reached 29.44 Gbps of throughput and 360 ns of latency at 230 MHz of functioning frequency which proves the validity of our approach for conducting best qualities solutions.

Key words: Hardware description languages, high level synthesis, high performance computing, parallel architectures, parallel processing, reconfigurable architectures.

1. Introduction

Improving the performance of general and specific purpose systems by the adjunction of accelerators is seen as a viable approach and it gains this last years a good admission in a markets industry as well in research fields. FPGAs are widely adopted as accelerators [1]. Those devices are used in hybrid and mixed systems for improving the execution time of the major critical programs parts. Those parts are referred mostly kernels in literature. However the developments flow for such hybrid systems lacks until now major facilities as it was stated in [2]. High Level Synthesis (HLS) tools begin to be dispensed those last years as a viable alternative for making easy and simpler the design tasks. HLS addresses the challenges of generating Register Transfer Level (RTL) specifications from High level languages (HLL), and is now attracting much the attention of hardware developers.

Several studies analyzed and discussed the challenges and opportunities of use of HLS tools. In [3], authors have covered the main causes and factors that have triggered the growing demand for innovative and high-quality HLS solutions. Despite the productivity and efficiency features that HLS tools afford for hardware developers who are the classical accelerators developers, HLS tools make the design issues accessible too for software developers who are not much familiar with the hardware primitives. So, hardware synthesis can now benefits from contributions of broader range of computer engineers. Authors of [3] and [4] exposed the main characteristics and essential operating concepts of the most emergent HLS tools. A large number of works like [5], [6] have showed that HLS achieves usually a higher productivity order than manual HDL coding. They conveyed also comparative analyses of resources and area consumption. The qualification of the HLS tools quality needs to be made with further caution and careful examinations. So, in this contribution we present HSCoT a HLS tool we have recently developed, we reveal our novel approach based on data-flow programming paradigm for building this framework and we prove its validity. We try to demonstrate how the proposed policy is allowing to get within the tool a best hardened solutions where speedups and the throughput are ultimately improved. Most of the existing tools encounter several limitations when optimizations are applied, whereas our proposed strategy is enabling to reach the theoretical speedups.

Our original motivation in constructing such tool is to provide a framework that enables the automation development of applications targeting hybrid and reconfigurable systems including a General Purpose Processor (GPP) and FPGAs assuming the role of accelerators. In this first release, our tool is able to assist developers to get the fastest needed IP cores from an algorithmic specifications expressed in C language. It is based on the intrinsic parallelism exploration approach, and it permits the generation of the most accelerated hardened solutions for intensive computation kernels and critical parts of applications written in a subset of C ANSI and targeting FPGAs. We wish in this paper to expose the essentials of the strategy adopted for building that tool HSCoT, and demonstrate how our employed policies are reliable.

2. Embedded Techniques and Optimizations in HLS Compilers

We bring here a brief examination of the several strategies, approaches and optimizations adopted for some HLS tools construction. We wish to highlight and inspect for each tool the major optimizations that are eventually affecting the quality feature of resultant solution. We have chosen four HLS tools among which two are academic projects and two are commercial products. The academic projects are LegUp [7]-[9], ROCCC [10], [11]. The commercial tools are Vivado [12] and Altera OpenCL [13], [14] originally initiated in [15]. Our criticisms are established on suspecting two things. First we examine the technique and the manner in which the optimization is done and how it can impact the speed-up of the final solution. Second we look at the performance ranks obtained for some examples of generated solutions. We want to point out that although if sometimes HLS compilers are embedding some similar optimizations or even same engines, the quality of the final generated solutions could be different. For example the three tools LegUp [8], ROCCC [11] and Vivado [12], [16] are all built using the low level virtual machine (LLVM) compiler [17], but we can see that each tool ensures a particular quality. LegUp covers almost the C language specifications, while ROCCC 2.0 focuses on streaming and the pipelining aspect and at the same time is unable to compile multiple loops at same nesting level as it is stated by ROCCC authors themselves [11]. That means that the affirmation of the use of a similar of one or more particular optimization in a number of tools does not formally lead to expect that all tools will features the same quality.

2.1. Academic Projects

LegUp is an open source HLS tool [8], [9]. It targets a hybrid FPGA-based software/hardware system, where some program segments are executing on an FPGA-based 32-bit MIPS soft processor and other program segments are automatically synthesized into FPGA circuits. Its strength is that it supports a large subset of C language specifications and is able to compile applications in entirety. LegUp uses the low level virtual machine (LLVM) compiler framework [17] to obtain from a C program an executable for the TigerMIPS soft processor. Then, it profiles the code to segment the sections that are intensive computation regions to be targeted to costume hardware accelerators, and keeps the rest in its assembly code to target the FPGA-based 32-bit TigerMIPS soft processor. LegUp operates at function level and replaces the bodies of functions to be hardened and targeted to accelerators by what the authors nominate wrappers function. The soft processor each time it encounters those wrappers, triggers the accelerators via the Avalon interface. For the qualification of the quality of the generated solution we simply inspect two results [7] and [18] outlined respectively by the LegUp authors themselves. First for some kernels like loop, Fir and FFT that are holding a substantial intrinsic parallel potential, LegUp comes to undertake those processing respectively in more than 292, 223 and 7377 clock cycles, while if the potential parallel is explored those processing may be achieved just at a few clock cycles (less than 30 clock cycle in worst cases) as we can do it with our proposed tool "HSCoT". Second the authors of LegUp afford three possible kinds of compilations: Pure software solution, pure hardware solution and hybrid solution. The execution time of those three kinds of solution are somewhat close each to other and the hardened solution was not achieved with a significant speedup respect to the pure software solution as it may be observed in [7]. We remember that for the quality feature we are interesting a long this paper particularly to the speedups. We don't attach further considerations to the resources consumption for the moment.

ROCCC 2.0 [10], [11] looks for the generation of customized hardware accelerators for frequently executed segments code. It does not address the compilation of entire application; rather it focuses on the critical parts. ROCCC is built using the Stanford University Intermediate Format (SUIF) framework [19] in the release ROCCC 1.0, and then it was revised and rewritten to work with LLVM framework [17] in the release ROCCC 2.0. The concepts long maintained in the ROCCC 2.0 compiler are the modularity and reusability. The design folds into two abstractions: Systems and modules. Modules are hardware blocs that should be reused in larger design that have frequently pipelined structures. This is referred Bottom-up and Reuse-design approaches [11]. Aside criticism that may be made to the effects brought by this approach to the development process with ROCCC, the full design usually tends to have a pipelined structure. The throughput will be likely improved; however latency is not necessarily enhanced. The authors of [11] afford as example a “bitonic-sort”, which is a module that makes the sorting of two data inputs. This module is then used to form a large design that sorts nine inputs for median filtering. We can easily expect that the final solution will fit in a pipelined structure scheme.

2.2. Market Tools

Vivado HLS tool [13], [16] is typically oriented towards the synthesis of C functions into IP blocs that can be integrated in a larger project design flow under Vivado design suite environment. Vivado like LegUp is also built using LLVM compiler. The optimizations of the design are allowed to be done by the end user during the development flow by means of particular directives that developer could insert in the program. Mainly, the optimizations allowed within Vivado HLS are loops unrolling, functions and loops pipelining, loops flattening and other optimizations relating to arrays. However those optimizations are not allowed in all conditions, they are several limitations for their application [13], [16]. Vivado HLS does not schedule several loops to execute in parallel; it encounters problems with variable loops bounds and some optimizations like loop unrolling should not be invoked if there is variables dependency. Those difficulties impact the quality of generated solutions within Vivado HLS and prevent the achievement of high rates of speedups.

Altera OpenCL compiler [13], [14], initiated originally within [15] allows the use of C based programming language for targeting variety of platforms including CPUs, GPUs, DSP, FPGAs and also heterogeneous boards. It provides a parallel oriented C specifications extension that extract eventual parallelism and API libraries that abstracts communications between host and accelerators. OpenCL suggests a global programming model where is specified a set of recommendations for platforms, execution flow, memory, and programming style [20]. The recommended platform consists of a host connected to what authors nominate OpenCL devices. Each device is a collection of compute units (CUs) and each CU in turn is also a collection of processing elements (PEs). The OpenCL run time executing on the host breaks the data-parallel tasks scheduled previously by the end user into a OpenCL kernels and dispatch them among the OpenCL devices. OpenCL kernels are written with OpenCL C extension programming language and compiled with OpenCL compiler. The end user rather to be able to synthesize the desired kernels directly in C ANSI language, should soon before learn and assimilate these specific OpenCL C extensions. OpenCL bears certain similarity with ROCCC in the sense that is oriented towards data streaming architecture and pipeline structures as it can be depicted within [13] and [10]. Even though, that OpenCL supports the unrolling loops optimizations, like is the case with Vivado, there are several limitations for their application.

3. "HSCoT" a Proposed HLS Tool and an Embedded Novel Programming Paradigm

Our initial motivation when we started the development of the proposed framework we have sought to find the appropriate way for making Total Distribution Arithmetic (TDA) of a sequential program written for the Von Neumann machine in such manner that all independent arithmetic can be executed in parallel. This will leads normally to the full exploration of the intrinsic parallel potential present in the program and hence getting the maximum speedups. The theoretical maximum achievable speedup for a given program or a segment code of program is determined by Amdahl's law and evaluated as the inverse of the inherent sequential fraction in that algorithm, commonly referred fs. The first result driven from Amdahl's law is that there is always a certain threshold for acceleration. This threshold cannot be exceeded even we use a number of resources much larger than necessary. The graph depicted in “Fig. 1.” and which in essence is taken from [21] illustrates also this inference. So we should not use a number of resources that exceeds the limits (Amax in the graph) that meet these thresholds and obviously we need to know in advance the limits of achievable accelerations. In our knowledge, almost all currently present tools have not raised the issue of the specification of the inherent sequential fraction or theoretical speedups. Within our proposed tool "HSCoT" we are able to specify the sequential fraction inherent of algorithms with great accuracy. Unlike other emergent tools [9], [10], [18] which are built using the particular frameworks [17], [22], and hence will be subjected to several licenses and dues, our tool is produced from scratch, and freed from any licenses and any dues. The tool parses programs given in C language specifications and after a series of lexical and syntactic processing rebuild the grammar of the program, profiles the existing dependencies for constructing after that step by step a total dependency data flow graph (DDFG) corresponding to the source program. That graph is then investigated to get in final steps the TDA image for the considered kernel. Even the concept of DFG is addressed thoroughly in a lot of works, in our knowledge right now we are not able to show a concrete compilers that are adopting an approach that is similar to the one proposed here. We are not as well able to find frameworks that profiles accurately parallelism in algorithms and to promise an ease way for its full exploration. In the present contribution we can do that. So, in the proposed framework we are making a transformation that brings application from the original classical representation featuring a sequential consistency to a parallel-spatial representation. The first form fits on the sequential-stored programing model (SSPM) targeting usually Von Neumann machines, while the second representation fits on parallel-spatial programing model (PSPM) that can targets circuits having high architectural parallel potential as FPGAs. The total work flow of the tool is illustrated in “Fig. 2.” The engine numbered 1 forms the preprocessing part. It encompasses lexical and syntactic analyses. The succeeding engine numbered 2 contains the hard tasks in the tool and leads to get the total DDFG. That DDFG is considered an Intermediate Representation IR and is nominated here “the map” . So, this IR may serve other future investigations (like Design Space Exploration DSE, co-design challenge…). The map is a two dimensional representation that may reflect in an early step the intrinsic parallel potential. Obviously, the most essential optimizations (constant propagation, inlinning, loop unrolling, and blocs flattening …) are undertaken in this step. For making further clarification we afford the following example:

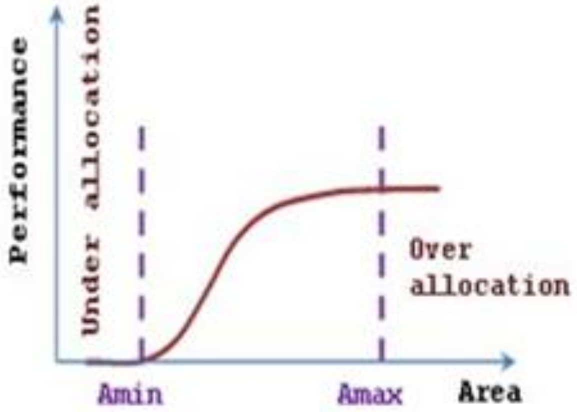


Fig. 1. Practical model of accelerators: Performance Vs area.

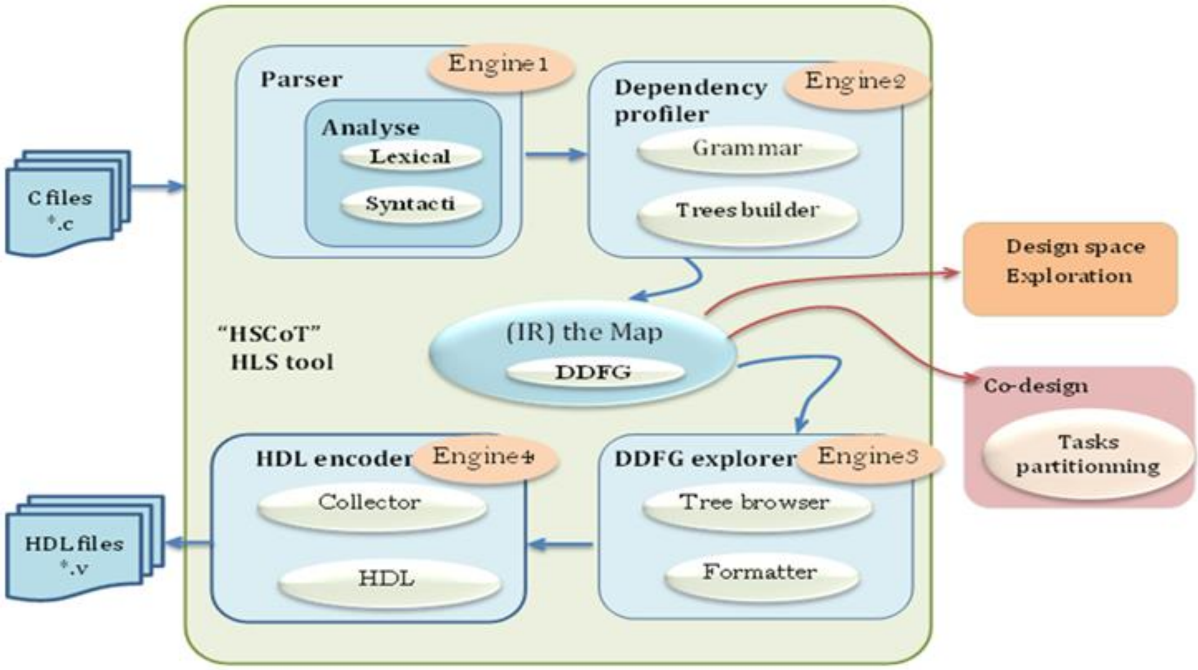


Fig. 2. Main engines in “HSCoT” tool and its workflow.

The (Table 1) provides an alternative pseudo algorithm for adaptive filtering core. It was chosen as simple example comprising nested loops with data dependency. In this algorithm we are about incrementally adjusting a weight vector, based on the error calculated as the difference between the desired signal and the estimated signal. This algorithm contains no much, and just a limited intrinsic parallel potential. It would be difficult to intuitively define data dependencies in that algorithm. The C fragment code corresponding to this algorithm contains three nested loops in an outer loop with crossed dependencies. Real world applications algorithms are often more complex structures, and until now, there are not reliable methods ensuring the accurate qualification of their parallel potential. The proposed tool affords an appropriate way allowing the accurate specification of the intrinsic parallel potential and its efficient exploration. Partial DDFG generated within the tool for this algorithm is shown in “Fig. 3.”. Evidently, all loops are unrolled despite there are interdependencies. Those kinds of displays for generated DDFGs are allowed within the tool just for debugging purposes. We have chosen a limited filter order M=4, and also a limited number of recurrence N=5, for getting no huge illustration example and somewhat intelligible graph. The generated graphs are explored as follow: all arithmetic that belongs to the same depth level in the trees must be executed concurrently, and each depth level on the trees is assumed to form one stage in the full pipeline. The subsets of arithmetic that cannot be hold concurrently will be pipelined. So, two kinds of parallelization are made: The parallelization in the sense of concurrency and parallelization by pipelining. Concurrency improves latency, while pipelining improves throughput.

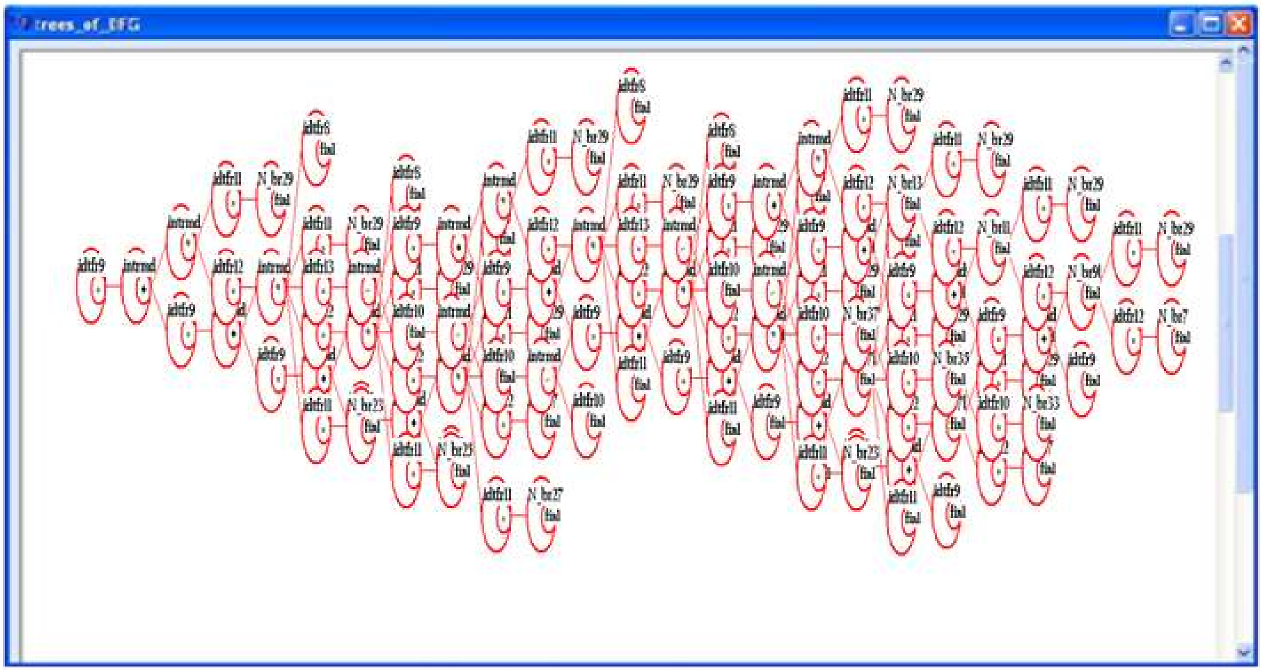


Fig. 3. Partial DDFG generated within “HSCoT” for one output in adaptive filtering kernel (screen capture showing loops unrolling effect).

Table 1. A Generic Pseudo Algorithm for Adaptive Filtering Kernel

```
$for\ n = 0 : N - 1$
        $Y_{estimated} = W^T * X_{input}$
        $err = Y_{estimated} - Y_{wanted}$
        $W = W + m * err * X_{input}$
$end$
```

Hence, with "HSCoT", the final solutions generated are consequently improved both in latency and throughput. The “map” will be explored in two orthogonal directions: x-axis direction and y-axis direction. Additional comments are reported in “Fig. 4” for further clarification. All arithmetic that belongs to the same depth level in the trees is independent and must be executed concurrently, and each depth level on the trees is assumed to form one stage in the full pipeline. The subsequent of depth levels in the x-axis direction match the pipeline stages and determine the latency in number of execution cycles, while the nodes per depth level in the y-axis direction define the speedup occasioned by existing concurrency.

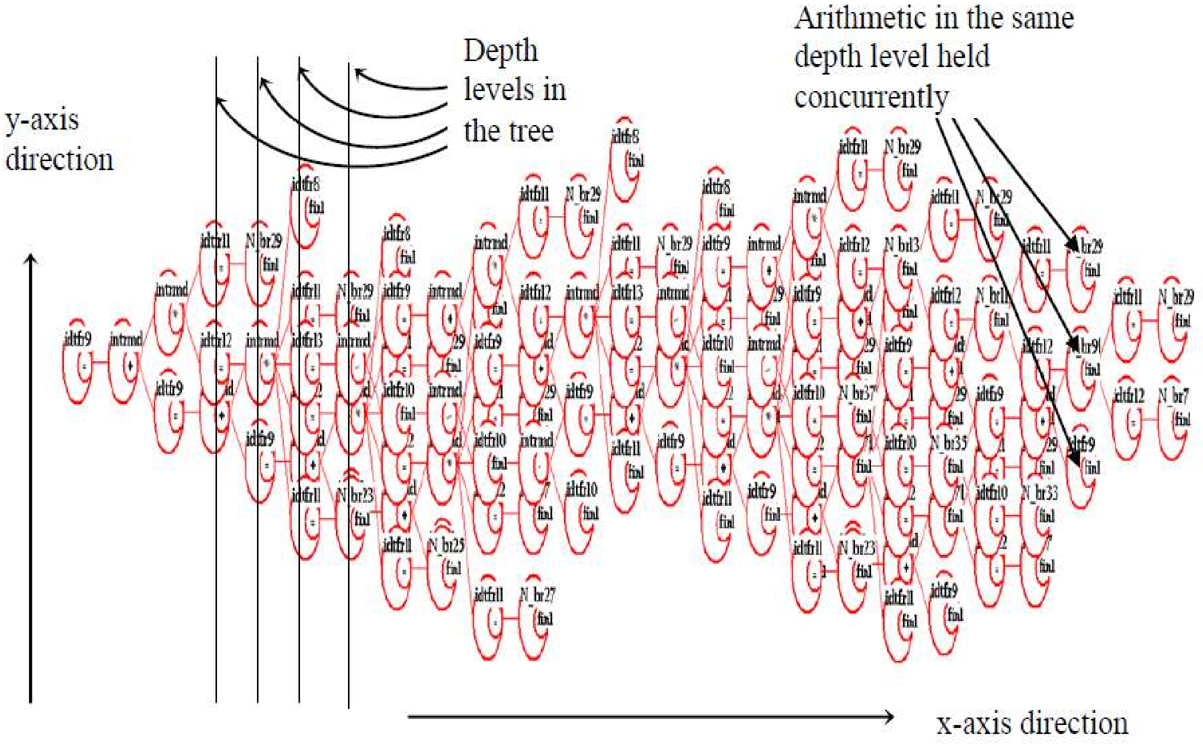


Fig. 4. Comments on partial DDFG generated within the tool (One tree corresponding to one output in adaptive filtering kernel).

The DDFG extends to fit on several levels depth due to the sequential nature of that algorithm. The number of depth levels determines the length of the pipeline or a number of pipeline stages, while the number of operator's nodes per level determines the speedup per level. Thus, the total speedup is the mean of the different speedups per depth levels. As there are about 22 depth levels, this processing will be held in 22 execution cycles which is the latency. The outer loop has 5 rounds in the correspondent C algorithm and the number of arithmetic in each round is 16 which raise the total number of arithmetic to be roughly 90. So, the speedup achievable is more than x4. The same value of speedup may be also directly deduced from the DDFG, where there are at least in average 4 operator nodes per depth level. Furthermore, while each depth level is expected to be held in one execution cycle, this filtering in whole is allowed to be done at the throughput of one sample per clock cycle.

3.1. Map Construction Process

Lexical and syntactic analyses are undertaken primarily in the tool and the grammar of the source was been rebuilt. The engine numbered 2 encompasses necessary tasks for dependency profiling, loops and nested loops unrolling, conditional structures handling and some other optimizations making. The result produced by this process is a group of DDFGs (the map). The essential of map construction process is coarsely summarized within the adjoined algorithms (Algorithm. 1 and Algorithm. 2). At this step we are getting a two dimensional representation where it would be easy to appreciate the intrinsic parallel potential. The map is then investigated subsequently within the engine 3 and then engine 4.

Algorithm1: Elementary trees building.

```
Procedure build__elementary__graphs(bloc__beginning, bloc__end)
    Input : list__of__struct grammar__previously__generated
      Ouput : list__of__trees
      Begin
          for index : bloc__beginning to bloc__end
              Begin
                  if (entry_sttmt is pertinent__entry)
                      build__elementary__graphs()
                  else if (entry_sttmt is conditional__struct__entry)
                      begin
                          Delimite__bloc(&beginning,&end__bloc)
                          Call build_elementary_graph__for__condt_blc(
                          beginning, end__bloc)
                      end
                  else if (entry_sttmt is loop__struct__entry)
                      begin
                          Delimite__bloc(&beginning,end__bloc)
                          Call build__elementary__graph__for__loop_blc(
                          beginning, end__bloc)
                      end
                  else if (entry_sttmt is other__struct__entry)
                      begin
                          Delimite__bloc(&beginning,end__bloc)
                          Call build__elementary__graph__for__other_blc(
                          beginning, end__bloc)
                      end
              end
      end
```

3.2. Map Investigation

Within the engine 3, the map is parsed again and each depth level in each tree is annotated. Nodes that are annotated to belong to the same depth level are collected to be held afterward concurrently. Accordingly, each group of concurrent arithmetic in the collection already formed will be assigned to be held in one execution cycle. Finally within the last engine the outcome is restructured in an appropriate Verilog HDL structures.

Algorithm2: Map construction

```
Procedure build__Map__for__bloc(frst__item__index, last__item__index)
    Input : list__of__ trees
    Ouput : the__map
    Begin
        index ← last__item__index
        while index # frst__item__index
            Begin
                if (item is pertinent__item)
                    Begin
                        for each item__component do
                            Begin
                                While(missmatch (item__component , current_item))
                                    current_item← previous_item
                                    Mount__item(current_item)
                              end
                        end
                  else if(item    matches particular__item__struct)
                        Begin
                              delimite__struct(&top, & bottom)
                              call build__Map_of_particulr_bloc(top, bottom)
                              update(index)
                        end
            decrement(index)
            end
      end
```

Algorithm3: Map construction (complement)

```
Procedure   build__Map_of_particulr_bloc (bloc__beginning, bloc_end)
      Input :bloct__of__ trees
      Ouput : partial__map

      Begin
            deal-with-particul-struct()
            Delimite-bloc(&beginning,&end-bloc)
            build-Map-for-bloc(beginning, end-bloc)
      end
```

4. Parallelization in Kernels Featuring High Intrinsic Parallel Potential

Table 2. Performance Comparative Outcomes in Different HLS Tools

Kernel		*Tool*		
		HSCoT	LegUp	DWARV 2.0 [12]
fir	Cycles	3	223	127
	Speedup	30	0.23	4.41
loop	Cycles	3	292	380
	Speedup	10	1.30	0.77
FFT	Cycles	10	7377	8053
	Speedup	200	0.71	1.4
DCT	Cycles	10	24004	41338
	Speedup	300	N/A	N/A

Intensive computing applications that contain habitually a significant parallel potential are a trusted field where our tool is mostly devoted. Depending on the intrinsic parallel potential we can exceed hundreds in speedup rate. The speedups achieved by mean of the tool are close to theoretical limits. We investigate here some kernels samples that have been already implemented with LegUp and the HLS tool DWARV 2.0 [18]. The set of chosen algorithms are: *loop, fir, FFT* and *DCT*. This set of kernels is given in a strict C specifications within the documentation fold included in LegUp distribution [23]. In this earliest release, our tool supports restricted subset of the ANSI C specifications including particularly: all control flow statements, all arithmetic, logic, bitwise operations and integer types and it will be improved soon to cover more large subsets. "HSCoT" in this first release does not support floats arithmetic and the kernels *FFT* and *DCT* are calculated in fixed point arithmetic with the precision of 10 bits. This precision is much wide sufficient to allow an appropriate achievement of most common applications. We can certainly agree that each algorithm in this set encompasses a reasonable parallel potential. In this set of kernels, algorithms contain loops and nested loops without data dependency or more exactly without crossed data dependencies. For this kind of kernels the generated DDFGs extend towards the y-axis direction and the trees have only few depth levels in x-axis direction with great number of nodes per depth level. We get trees with just 3 to 4 depth levels for *loop* and *fir* kernels and we get less than 10 depth level trees for *FFT* kernel. So, *loop*, *fir* and *FFT* are achieved respectively in 3-4, 10 clock cycles. And the speedups reached are at least respectively 10, 30 and 200. Results are summarized in “Table 2”.

5. Implementation of a Cryptographic Kernel Sample (The AES Cipher): Case Study

The advanced encryption standard (AES) cipher is treated here as a case study. We have chosen this algorithm, not only since it has a large adoption in several benchmarks suites, but also because it was given with an explicit and strict specifications in [24]. The intrinsic parallel potential within this algorithm as we will see in the coming sections is not really very substantial, whereas the efficiency of our tool may be further proven if application is encompassing a great parallel potential. So we wish in this last section to show in some details how it is always allowed by means of the tool at once to explore the parallel potential even if is slight in order to improve latency and to perform the pipelining for improving the throughput. We will evaluate the resources occupancy rates in the generated solutions and really we are expecting that the amount of resources utilization will grows somewhat because the tool makes a total flattening and TDA. The area occupancy minimization is not our objective for the moment, we did the resource utilization evaluation just for inspecting if it is tending to be excessive or it stills in a reasonable margins.

5.1. The AES Kernel

Table 3. The Pseudo Algorithm for AES Cipher [24]

```
$Cipher\,(byte\ in[4*Nb], byte\ out[4Nb], word\ w[Nb*(Nr+1)])$
$begin$
$byte\ state\,[4, Nb]$
$state\ =\ in$
$AddRoundkey\,(state, w[0, Nb-1])$
$for\ round\ = 1\ step\ 1\ to\ Nr-1$
    $SubBytes(state)$
    $ShiftRows(state)$
    $Mixcolumns(state)$
    $AddRoundkey\,(state, w[roun*Nb, \quad (round+1)*Nb-1])$
$end\ for$
$SubBytes(state)$
$ShiftRows(state)$
$AddRoundkey\,(state, w[Nr*Nb, \quad (Nr+1)*Nb-1])$
$out\ =\ state$
$end$
```

AES is a symmetric block cipher based on Rijndael algorithm. The specification set for that standard is given in the (FIPS Pub 197) [24]. The AES algorithm is capable to use cryptographic keys of 128, 192, and 256 bits to encrypt and decrypt data in blocks of 128 bits. Each block is copied into an array of 4xNb bytes referred *State* (in the FIPS specification). The details of the standard specifications can be consulted in [24]. The plaintext is encrypted on carrying a subsequent of transformations applied to the State array. The pseudo algorithm for AES cipher is given in (Table 3). These subsequent transformations are applied a number of times (Nr=10, 12 or 14) depending on the cipher key length. AES with cipher key of length 128 bit is referred to as AES-128. AES using a cipher key of 192 bit of length is referred AES-192 and so on AES-256 is using a cipher key of 256 bit of length. From the initial key is generated with the *KeyExpansion()* routine a set of (Nr+1)*Nb key schedule. The subsequent of the keys schedule are used in the routine *AddRoundkey()* respectively in the subsequent rounds of AES *Cipher()* routine. The pseudo algorithm for *KeyExpansion()* routine is showed in (Table 4) and further details could be found in (FIPS PUB 197) [24].

Table 4. The Pseudo Algorithm for Key Expansion Routine in AES [24]

```
$KeyExpansion(byte\ key[4 * Nk],$
                        $wordw[Nb * (Nr + 1)], Nk)$
begin
Word temp
 $i = 0$
 $while(i < Nk)$
       $w[i] = word(key[4 * i], key[4 * (i + 1)],$      $key[4 * (i + 2)], key[4 * (i + 3)])$
       $i = i + 1$
 end while
  $i = Nk$
  $while(i < Nb * (Nr + 1))$
      $temp = w[i - 1]$
      $if(i \bmod Nk = 0)$
        $temp = SubWord(RotWord(temp)) \oplus Rcon[i/Nk]$
      $else\ if\ (Nk > 6\ and\ i \bmod Nk = 4)$
                $temp = SubWord(temp)$
      end if
      $w[i] = w[i - Nk]\ \_\ temp$
      $i = i + 1$
 end while
 end
```

5.2. The AES Implementation

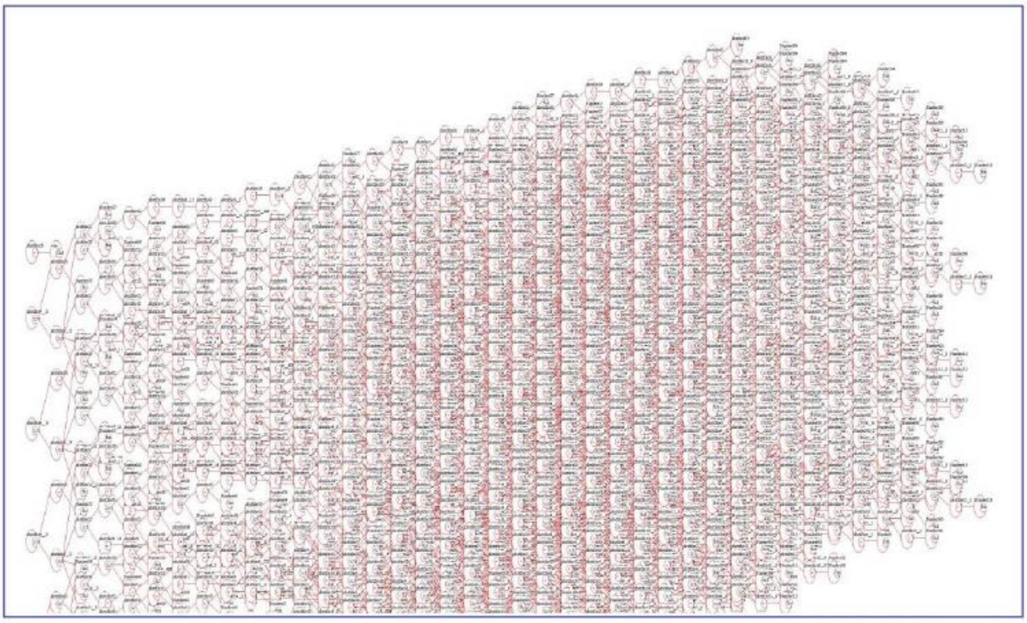


Fig. 5. Partial DDFG (partial map) generated within our tool "HSCoT" and flattening two rounds in the AES cipher routine. Obtaining a map that features a form in clouds reflecting the parallel potential.

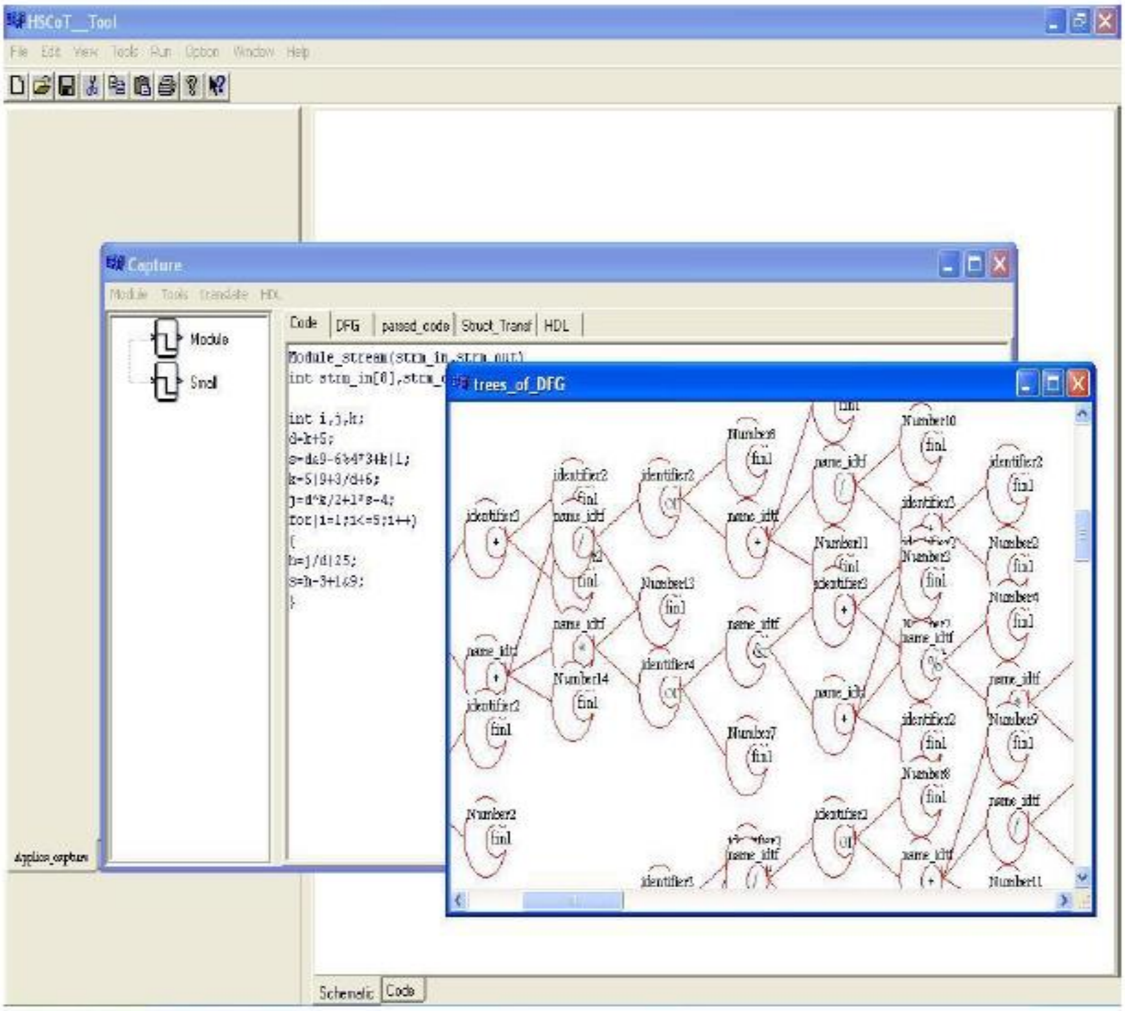
Once AES algorithm was be rewritten in C specifications and conceded to the tool, a hardened solution is automatically generated. The solution is given in HDL specifications. Mainly the tool transposes the program into parallel-spatial representation given in form of DDFG the ”map”. This map (possessing form in clouds) allows making earlier appreciations about the parallel potential, the pipeline length, the latency and relatively the achievable speedups. Knowing that our objective is specifically to depict how improvements both in speedup and throughput are made, in a first step, we illustrate the maps generated within the tool for the flattening of the first two rounds in *Cipher()* routines and *KeyExpansion()* routine. The maps (DDFGs) are given respectively in “Fig. 5” and “Fig. 6”.

Fig. 6. Partial DDFG (partial map) generated within our tool "HSCoT" and flattening two rounds in the keyexpansion routine. Obtaining a map that features a form in clouds reflecting the parallel potential.

Indeed, we are about showing just partial graphs from the maps. We remember again, the display of the maps is allowed within the tool just for early debugging purposes. We don’t look to show the details of nodes composing the trees, rather we aim to give a total appearance of the map in form of “clouds” reflecting the program nature (how much is parallel and how much is sequential). The map generated for the full rounds is some much huge and so, we have privileged to restrict the illustrations only to the two first rounds. We are observing that we have nearly 38 depth levels in the x-axis direction from left to right in the most extended trees given in “Fig. 5”. The tool processes the map and come to provide an HDL solution that can be achieved in 16 execution cycle. The number of execution cycles is always roughly less than the number of depth levels because the tool performs some optimizations. So the latency of the solution generated for the full AES ciphering algorithm where we have 11 rounds is 83 execution cycle. We may observe a condensation of nodes in the middle of the map. This result is logical because in this region we have partial recovery between the two rounds. All the 16 bytes of State (the plaintext) are treated at once, which leads to get a throughput of 128 bit per clock cycle. In our case where we have a 230 MHz of functioning frequency in virtex5 XC5VLX30 board, the achievable throughput is 29,4Gb/s.

5.3. Results

The generated solution for AES ciphering is implemented as a single core in the Xilinx ISE design suite 12.1 environment targeting the virtex5 XC5VLX30 at 230 MHz of clock frequency. It has been achieved in 83 clock cycles and at the throughput of one plaintext block (128 bit) per clock cycle. Those results are very significant, whereas other implementation in other works being summarized in (Table 5.) have also reached as well an important throughput rates, however, we should highlight above all that most of these solutions are a hand written synthesis and not an automated generated solutions. The tool permits to get the ultimate improvements in throughput and latency. We are being able to process one plaintext block each one clock cycle which matches a throughput rate of 29,4 Gb/s at 230 MHz of functioning frequency. Evidently, the throughput may double if we make replication of the generated module. Our solution permits to reach a latency of about 83 clock cycles for ciphering one plaintext block which correspond to 360 ns. We still retain that our tool permits to obtain latencies that are close to the theoretical limits and we estimate that theoretical latency is slightly less than 83 clock cycles at the difference of only few clock cycles. From beginning we have sought to make only latency and throughput improvements without according a major concern for area occupancy. Really, we expected to get a solution with somewhat much area consumption because our tool “Fig. 7” makes TDA. So, we gain latency and throughput improvement to the detriment of area occupancy. Although we have not until now embedded any optimization that concerns the area minimization, we think that those results of resources consumption remain reasonable and are not excessive.



```
Module_stream(strm_in,strm_out)
int strm_in[8],strm_o
int i,j,k;
d=k+5;
s=d&9-6%4*3+k|1;
k=5|9+3/d+6;
j=d^k/2+1*s-4;
for(i=1;i<=5;i++)
{
h=j/d|25;
s=h-3+1&9;
}
```

Fig. 7. “HSCoT” tool: Main graphic user interfaces.

Table 5. Comparative Performance Outcomes for AES-128 Implementation

Author	Synthesis method	Throughput (Gbps)	Latency (ns)	Area (slices)
this work	**automatic**	**29.44**	**360**	**10549**
[25]	hand coding	23.57	162.9	4901
[26]	hand coding	18.56	496	5016
[27]	hand coding	21.56	422	11022
[28]	hand coding	24.92	210.5	3576
[29]	hand coding	17.80	318	10750

6. Conclusion

The strength of FPGAs is mainly due to their parallel architectural potential. To efficiently exploit these circuits, it is first necessary to identify the parallel potential present in the algorithms to be implemented and then find the way for making their adequate matching to the set of architectural resources. The tools currently provided, although they provide quite satisfactory productivity factors, it seems that their solutions qualities are not yet quite approvable. In this paper, we have inspected a set of academic tools and market products and we have tried to describe the factors that have prevented obtaining the desirable qualities. We have proposed a completely different approach which essentially performs the full transposition of a representation model with sequential dominant character to a new model of representation given as DDFG that has a parallel-spatial consistency, allowing not only to construct the adequate HLS tools but it allows to address the challenge of the co-design and DSE more rigorously. This DDFG is exploited in two orthogonal directions (x-axis and y-axis directions) to make at once parallelization and pipelining. The processing within our tool of the AES-128 algorithm has confirmed its quality and has proved the validity of our approach. As future work, we pretend to exploit again that novel parallel representation model generated within our tool to build necessary runtime engines that make automatic dynamic partitioning and eventual load balancing.

References

[1] Herbordt, M. C., Vancourt T., & Yongfeng, G. (2007). Achieving high performance with FPGA-Based computing. *IEEE, Computer, 40(3),* 50-57.

[2] Jozewiak, L., Nedjah, N., & Figueroa, M. (2010). Development methods and tools for embedded reconfigurable systems: A survey. *Integration, the VLSI Journal, 43(1),* 1-33.

[3] Jason, C., Bin L., Stefen, N., & Juanjo, N. (2011). High-level synthesis for FPGAs: From prototyping to deployment. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 30(4),* 473-491.

[4] Skyler, W., Xiaoyin, M., Robert, J. H., & Prerna, B. (2015). High-level language tools for reconfigurable computing. *Proceedings of the IEEE, 103(3),* 390-408.

[5] Gregor, S., Christian, Z., Holger, B., Stefan, W., & Jan, E. (2014). HLS-based algorithm — A field report. *Proceedings of IEEE Conference on Design and Architectures for Signal and Image Processing (DASIP)* (pp. 1-8).

[6] Kenneth, H., Stefan, C., Alan, G., & Herman, L. (2015). Comparative analysis of opencl vs HDL with image-processing kernels on stratix-V FPGA. *Proceedings of IEEE 26th International Conference on Application-Specific Systems, Architectures and Processors* (pp. 189-193).

[7] Andrew, C., Jongsok, C., Mark, A., Victor, Z., Ahmed, K., & Jason, A. (2011). LegUp: High-Level synthesis

for FPGA-based processor/accelerator systems. *Proceedings of the 19th ACM/SIGDA International Symposium on Field programmable Gate Arrays* (pp. 33-36).

[8] Andrew, C., Jongsok, C., Blair, F., & Ruolong, L. (2013). From software to accelerators with LegUp high-level synthesis. *International Conference on Compilers, Architecture and Synthesis for Embedded Systems* (pp. 1-9).

[9] Andrew, C., Jongsok, C., Mark, A., Victor, Z., Ahmed, K., Tomasz, C., Stephen, D. B., & Jason, H. A. (2013). LegUp: An open-source high-level synthesis tool for FPGA-based processor/accelerator systems. *ACM Transactions on Embedded Computing Systems*, *13(2),* 1-25.

[10] Jason, V., Adrian, P., Walid, N., & Robert, H. (2010). Designing modular hardware accelerators in C with ROCCC 2.0. *Proceedings of the 18th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines (FCCM)* (pp. 127-134).

[11] Walid, N., & Jason, V. (2013). FPGA code accelerators — The compiler perspective. *Proceedings of the 50th ACM/EDAC/IEEE Conference on Design Automation (DAC)* (pp. 1-6).

[12] Xilinx. (2015). *Vivado Design Suite User Guide: High Level Synthesis.* Retrieved from http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug902-vivado-high-level-synthesis.pdf/

[13] Altera. (2015). *Altera SDK for OpenCL: Programming Guide.* Retrieved from https://documentation.altera.com/link/mwh1391807965224/mwh1391807939093/en-us/

[14] Altera. (2015). *Altera SDK for OpenCL: Getting Started Guide.* Retrieved from https://documentation.altera.com/link/mwh1391807309901/mwh1391807297091/en-us/

[15] Khronos Group. (2007). *OpenCL Reference Pages.* Retrieved from http://www.khronos.org/registry/cl/sdk/1.0/docs/man/xhtml/

[16] Xilinx. (2015). *Vivado Design Suite Tutorial: High Level Synthesis*. Retrieved from http://www.xilinx.com/support/documentation/sw_manuals/xilinx2014_1/ug871-vivado-high-level-synthesis-tutorial.pdf

[17] (2010). *The LLVM Compiler Infrastructure Project.* Retrieved from http://www.llvm.org/

[18] Razfan, N., Vlad-Mihai, S., Bryan, O., Roel, M., Yana, Y., & Koen, B. (2012). DWARV 2.0: A CoSy-based C-to-VHDL hardware compiler. *Proceedings of the 22nd International Conference on Field Programmable Logic and Applications (FPL 2012)* (pp. 29-31).

[19] Stanford University Intermediat Format (SUIF) Compiler System. (2004). Retrieved from http://suif.stanford.edu/

[20] Khronos Group. (2009). *The OpenCL Specification.* Retrieved from http://www.khronos.org/registry/cl/specs/opencl-1.0.48.pdf/

[21] Amir, M., Tomer, Y. M., Leonid, Y., Ran, G., & Uri, W. (2014). Generalized multiamdahl: Optimization of heterogeneous multi-accelerator Soc. *IEEE Computer Architecture Letters*, *13(1)*, 37-40.

[22] Associated-compiler-experts. (2011). CoSy compiler platform. Retrieved from http://www.ace.nl/

[23] LegUp. (2013). LegUp distribution. Retrieved from http://legup.eecg.utoronto.ca/download.php/

[24] FIPS. (2001). Federal information processing standards publication: Advanced encryption standard. Retrieved from http://csrc.nist.gov/publications/fips/fips197/fips-197.pdf/

[25] Zambreno, J., Nguyen, D., & Choudhary, A. (2004). Exploring area/delay tradeoffs in an AES FPGA implementation. *Field Programmable Logic and Applications (FPL)*, 575-585.

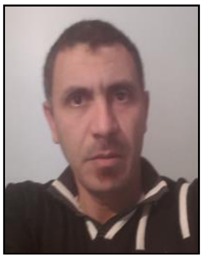
[26] Standaert, F. X., Rouvroy, G., Quisquater, J. J., & Legat, J. D. (2003). Efficient implementation of Rijndael encryption in reconfigurable hardware: Improvements and design tradeoffs. *Proceedings of the 5th Cryptographic Hardware and Embedded Systems (CHES)* (pp. 334-350).

[27] Zhang, X., & Parhi, K. K. (2004). High-speed VLSI architectures for the AES algorithm. *IEEE Transactions*

on Very Large Scale Integration (VLSI) Systems, *12(9)*, 957-967.

[28] Granado-Criado, J. M., Vega-Rodriguez, M. A., Sanchez-Perez, J. M., & Gomez-Pelido, J. A. (2010). A new methodologie to implement the AES algorithm using partial and dynamic reconfiguration, integration. *The VLSI Journal*, *43,* 72-80.

[29] Jarvinen, K. U., Tommiska, M. T., & Skytto, J. O. (2003). A fully pipelined memoryless 17.8 Gbps AES-128 encryptor. *Proceedings of the 11th International Symposium on Field Programmable Gate Arrays* (pp. 207-215).



Aimad Eddine Debbi received an engineer degree in electronics from Farhat Abbass University in Setif, Algeria and magister degree from University of Guelma, Algeria in 2002. Since that he is holding an assistant professor position. He resumed again the research activities since 2010. He participated to a number of international conferences. His research interests actually include computer aided design, design automation, systems architecture, runtime and operating systems for embedded systems.