RECURSION IN RECONFIGURABLE COMPUTING: A SURVEY OF IMPLEMENTATION APPROACHES

Iouliia Skliarova, Valery Sklyarov

Department of Electronics, Telecommunications and Informatics / IEETA
University of Aveiro
3810-193 Aveiro, Portugal
email: iouliia@ua.pt, skl@ua.pt

ABSTRACT

Reconfigurable systems are widely used nowadays to increase performance of computationally intensive applications. There exist a lot of synthesis tools that automatically generate customized hardware circuits from specifications in both high-level and hardware description languages. However, such tools have a limited applicability because they are unable to handle recursive functions whereas it is known that recursion is a powerful problem-solving method widely used in computer science. Therefore a great deal of research effort is aimed at efficient implementation of recursion in reconfigurable hardware. This paper presents the state of the art in this area. The existing proposals are described, analyzed, and compared according to such criteria as level of parallelism supported, approach to concurrency, ease of use, availability of automated high-level synthesis tools, etc.

1. INTRODUCTION

Recursion is a powerful problem-solving method widely used in computer science. The basic idea is to divide a given initial problem into a finite (and usually very small) set of simpler sub-problems in such a way that every sub-problem is of exactly the same type as the original problem [1]. Subsequently, the same decomposition can be applied *recursively* to each of the sub-problems until newer sub-problems turn out to be so simple that their solution is known (this situation is identified as a *base case*). Once the base case is reached, a solution to the previous sub-problem can easily be composed. By moving gradually from smaller sub-problems to bigger sub-problems, a solution to the original problem is finally obtained. Sometimes it is necessary to solve all the sub-problems in order to get the solution to the original problem (such as in inefficient recursive calculation of Fibonacci sequence). Other times, only part of the sub-problems needs to be solved (*e.g.* binary search).

Recursive specifications often produce elegant and easier to understand solutions than the respective iterative specifications. Therefore, practically all modern programming languages provide support for recursion. Recursive functions (as well as functions in general) are implemented in general-purpose computers with the aid of stack memory which keeps all necessary information permitting to return from a recursive call and to restore the state of data as it was before the recursive call. Managing stack (pushing there all the required data before a function call and popping these data as soon as a recursive return is to be done) incurs an overhead for each function call, and recursive functions magnify this overhead because eventually a large number of recursive calls can be generated [1]. But since the use of recursion frequently clarifies complex programs and, in some cases, can be very efficient (*e.g.* binary and N-ary search), additional overhead may be ignored.

Taking into account the main advantages of recursive specifications it would be worthwhile to use recursion in reconfigurable hardware. Besides, recursive functions are the most time consuming parts in many algorithms and accelerating their execution with reconfigurable hardware would be very beneficial. However, modern hardware description languages (HDL) as well as system-level specification languages (SLSL) do not provide direct support for recursion. This can be explained by two reasons. First of all, HDL and SLSL descriptions can be synthesized and further implemented over different hardware platforms. These platforms, as a rule, do not possess a dedicated stack memory which could be used for supporting recursive calls. So, the first reason is the lack of hardware support. Obviously, stack could be synthesized specifically for each problem but it is not easy to calculate recursion depth (and, consequently, the required stack size). One alternative solution would be to substitute recursion automatically by the respective iterative specification. However, commercial synthesis tools do not follow this approach because of its inherent complexity. Therefore, the second reason of not synthesizing recursion is the associated complexity. Moreover, for some algorithms, iteration requires more data movement, is less clear, and is often slower [2].

Nevertheless a number of techniques have been suggested aimed at implementing recursion in

978-1-4244-3892-1/09/$25.00 ©2009 IEEE

reconfigurable hardware. The main objective of this paper is to survey published proposals, to identify their relative advantages and drawbacks, and to give recommendations for future work.

The remainder of the paper is organized in three sections. Section 2 provides an overview of methodologies for implementing recursive functions in reconfigurable hardware. Section 3 makes a comparison of approaches discussed in section 2. Finally, concluding remarks are given in section 4.

2. METHODOLOGIES FOR IMPLEMENTING RECURSION IN RECONFIGURABLE HARDWARE

Different attempts have been made to implement recursion in reconfigurable hardware. Basically, all proposals fall into one of two broad categories: unroll recursive calls into a pipelined circuit or implement it with a stack. In the following sub-sections all the suggested methodologies will be reviewed.

2.1. Maruyama *et al.*

One of the first works on implementing recursion in reconfigurable hardware was done by Mariyama *et al.* [3, 4]. In particular, two techniques have been proposed: multi-thread execution and speculative execution.

The first of these techniques was applied to combinatorial optimization problems that require the whole search space to be traversed (the knapsack problem was selected as a case study). First of all, all tail recursions (recursive calls that are the last statements in a given recursive function) were optimized into loops for the sake of efficiency. Then a pipeline was constructed whose number of stages is equal to the number of blocks into which a given algorithm can be decomposed. Operations within each block are executed in parallel. As all pipeline stages become active at the same time, the maximum attainable speedup is limited by the depth of the pipeline. When a recursive call is to be done, the arguments are either forwarded directly to the respective stage in the pipeline or pushed into a stack if the stage is occupied. This technique was applied for solving the knapsack problem in an Altera FPGA [3]. The resulting circuits supported the maximum clock frequency of 35 MHz and occupied 17% of logic resources of EPF10K100 device. The achieved speedup calculated as an average of solving 100 small problem instances (with 32 objects to be put into a knapsack) was 6.7x over the implementation of the same algorithm in software executing in Ultra-Sparc 200 MHz. The application of multi-thread execution method, however, seems to be very limited since, in a general case, all pipeline stages can potentially need reading and writing data from/to memory and supporting parallel accesses to different locations of the data memory in parallel is not feasible. Moreover, it is not clear how to proceed if there is a data dependency between the results of a recursive call and subsequent function's statements.

The second technique proposed by Mariyama *et al.* [3] is aimed at solving combinatorial search problems (for which only one, not necessarily optimal, solution has to be found) and is more suitable for loops which include recursive calls. Knight's tour problem was considered as an example. The main idea is to speculatively execute consecutive loop iterations in parallel assuming that no one of these iterations will make a recursive call. As soon as this assumption fails for iteration *i*, the recent computations for iterations *i+1*, *i+2*,…, *i+n* (*n* is the number of pipeline stages) are cancelled, the current data are pushed into stack and the loop is restarted from the beginning (simulating a recursive call). On a recursive return, the data are popped from the stack and execution is resumed at the interrupted stage. Mariyama *et al.* were able to achieve 4 times speedup by speculative execution (clocked at 31 MHz) over software implementation running on Ultra-Sparc 200 MHz. It is not clear however whether this speedup accounts for FPGA configuration time as well as for data transmission time.

In order to automate the process of generating circuits for speculative execution from high-level programming languages, a compiler was developed and reported in [4]. The compiler accepts C code augmented with special notations (such as for specifying the size of data in bits and for identifying statements to be executed in parallel) and generates synthesizable HDL code (based on speculative execution). All recursive calls are previously transformed into iterative loops by a pre-processor. When memory holding data is accessed more than once by different pipeline stages, the pipeline has to be stalled and memory access operations are executed sequentially. The generated circuit speculatively executes next loop iterations and resets and restarts them when data feedback dependencies are detected [4]. The resulting circuits generated by the compiler run in a range of 39-47 MHz (on Altera EPF10K100 FPGA) and achieve a speedup of 2 times (measured in the number of clock cycles) when compared to non-speculative execution.

2.2. Sklyarov *et al.*

Sklyarov *et al.* proposed a technique for implementing recursive functions in reconfigurable hardware with the aid of hierarchical finite state machines (HFSM) [5-7]. The main idea is to implement recursive calls in hardware in the same manner as it is done in software and to parallelize just those operations that occur in between recursive calls. Each function (which can be recursive or not) is supposed to be executed by a specific hardware module which attempts to execute as many operations as possible in parallel. Recursive calls invoke operations over stacks in such a way that the state of the module (where recursive invocation has happened) is saved onto a stack and the stack pointer is incremented to address the storage for a recursively called module. When the recursively called module ends, the stack pointer is decremented in order to restore the state of the interrupted module [7].

The suggested hardware model of HFSM is depicted in Fig. 1. The HFSM consists of a combinational circuit and two stacks (that keep track of hierarchical functions invocations), one for states (*FSM_stack*) and the other for modules (*M_stack*).

The stacks are managed by a combinational circuit that is responsible for new module invocations and state transitions in any active module that is designated by outputs of the *M_stack*. Any non-recursive transition is performed through a change of a code only on the top register of the *FSM_stack*. Any recursive call alters the states of both stacks in such a way that the *M_stack* will store the code for the new module and two values will be pushed into the *FSM_stack*: first, the code of the next state in the calling module and then the code of the first state in the called module. Any hierarchical return just activates a pop operation without any change in the stacks. As a result, a transition to the state following the state where the terminated module was called will be performed. The stack pointer is common to both stacks. If the final state in a module is reached when the stack pointer is equal to zero, the algorithm terminates execution.

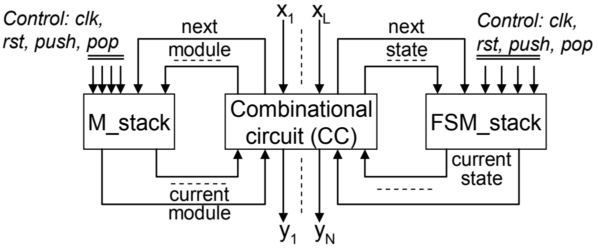
Control: clk, rst, push, pop
next module
x1
xL
next state
Control: clk, rst, push, pop
M_stack
Combinational circuit (CC)
FSM_stack
current state
current module
y1
yN

Fig. 1. The hardware model of HFSM.

Besides of two control stacks depicted in Fig. 1, an additional data stack is usually required to preserve all necessary data in between recursive calls (that allows a previous function's data to be easily restored). Since all the stacks are constructed on the basis of embedded in FPGA memory blocks the additional FPGA resources required for stack management are negligible.

The technique suggested in [5-7] has three main advantages. First of all, this method can be applied for implementing any recursive function with an arbitrary number of recursive calls done within that function. The only condition that has to be assured is that a function does not contain infinite recursion (the same has to be guaranteed in software). Obviously, the maximum depth of recursion has to be known before runtime, but since memory resources are abundant in recent FPGA families, stacks can be created so as to be able to accommodate the biggest number of modules that could eventually be called by a given algorithm.

The second advantage is that parameterizable VHDL and Handel-C templates have been developed for the stacks and the combinational circuit composing an HFSM [5-7]. Consequently the design process is very easy: it is only necessary to customize the templates for a particular algorithm.

And, finally, the same synthesized and implemented circuit can be used for solving various problem instances. This is not possible with the majority of other approaches that require the circuit to be resynthesized (for example, when initial problem data influence the number of times that a function has to be unrolled).

The main disadvantage of the method is that parallelism is limited to executing in parallel operations that occur in between recursive calls. If the amount of work in these operations is high, then the respective implementation can outperform significantly the corresponding software implementation. Otherwise, if there is a limited number of operations whose execution can be parallelized, then the resulting circuit will require roughly the same number of clock cycles as software running on a single-core microprocessor (actually, the number of clock cycles in hardware will be smaller because invocation of new modules can be overlapped with execution of other algorithm's operations [7]). Since the clock frequency of microprocessors is generally higher than that of FPGA-based systems, the resulting speedup would be negative.

Several experiments have been realized for problems of binary tree sorting, matrix covering, knapsack problem and computation of the greatest common divisor [5]. The respective recursive algorithms were described in VHDL and Handel-C and implemented on FPGAs of Spartan-IIE, Spartan-3 and Virtex-II families. The main conclusion was that using modular algorithms, in general, and recursive algorithms, in particular, was more advantageous for problems of binary (N-ary) search compared to iterative implementations [6].

2.3. Ferizis *et al.*

Ferizis *et al.* proposed a method for mapping recursive functions to reconfigurable hardware without the use of stack [8-11]. The main idea was to parallelize function execution as much as possible and for this it was suggested to unroll the function as many times as necessary in an FPGA at run time and to execute recursive calls in parallel.

In [8-9] all the statements in a recursive function are split into two disjoint sets: pre-recursive (containing statements that occur before the recursive calls) and post-recursive (containing statements that occur after the recursive calls). The resulting hardware circuit includes one pre-recursion and one post-recursion logic unit for each level in the function call graph. The logic units are arranged in a pipeline as indicated in Fig. 1 for a case of recursion of depth 2 (a function calls itself once and when it is activated for the next time the base case is reached). If a function includes several recursive calls (more than 1) then every level of the function call graph contains several nodes, however, just one of them can be active at a time.

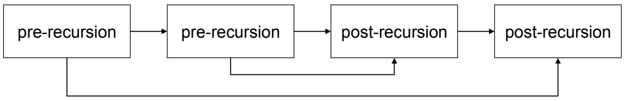

Fig. 2. Arrangement of pre- and post-recursion units for a recursive function of depth 2.

If the recursion depth is known before the execution then the pipeline is configured so as to match exactly a problem instance. Otherwise, a minimum estimated number of logic units are configured and if a recursion level is reached which exceeds the number of levels supported, then the required additional recursive levels are added to the pipeline by partial run-time reconfiguration [8-9]. To simplify placement and routing all logic units are placed in columns (that span the whole height of an FPGA of Xilinx Virtex-II family) and interlinked with a network-on-chip. The need for additional logic levels is predicted by a special circuit whose task is to monitor input into the system and detect the need for more logic before it is actually needed, allowing to minimize the overhead of reconfiguration [9].

In [9] different case studies are presented where the suggested method was applied to merge sort and Strassen's matrix multiplication. However, the method's applicability was only studied at the theoretical level and no real hardware implementation was done.

In [10-11] it was proposed that each stage of pipeline, instead of computing either pre- or post-recursive statements of a single function, would control all the functions that occupy the current recursive level, and, as a result, would execute multiple instances of the recursive function in parallel.

In order to deal with limited hardware resources (that could not be sufficient to fully unroll a recursive function) it was proposed to force each stage of the pipeline to compute multiple recursive levels [10]. It is not clear however, how data dependencies between recursive levels are resolved. Actually, it seems that the whole model does not account for memory access component (simultaneous memory access by different logic units, available memory bandwidth, etc.).

Run-time reconfiguration was not implemented and therefore a simulator was developed to estimate the possibility and effectiveness of unrolling recursion on the fly. Several case studies are presented in [11] which were either implemented on a Celoxica RC200 board or ran on a simulator (that was employed to test run-time reconfiguration). It was shown that merge sort, quick sort and quad tree partitioning algorithms, when fully unrolled, run in linear time and outperform stack-based implementation [10-11]. However, the required hardware resources are unknown and the level of details presented in [8-11] does not allow to reproduce experiments and to comment the results fairly.

2.4. Ninos *et al.*

The work of Ninos *et al.* extends the results of previous proposals and suggests a data-oriented approach [12]. This approach relies on a *recursion simplification operation* which essentially transforms recursion to iteration with a stack. Basically, if the condition for recursion is met then local data are pushed into the data stack and the function execution is restarted. Otherwise, if the condition for recursion is not met, the function simply continues its execution. When the last statement within the function is reached, if the data stack is empty the execution is ended. Otherwise, a recursive return is performed by popping the previous data from the stack and returning to a state from which recursive call was done. Since on a recursive return control always goes back to the state from which a recursive call has been done, it means that this state has to be repeated in order to evaluate the next state to follow, which will result in performance degradation.

The authors claim that *recursion simplification operation* can be expanded to as many recursive calls within a function as necessary [12]. It seems however that some additional information should be stored on the stack that would allow to identify unequivocally the point to return to from recursion in case of multiple recursive calls. Actually, this assumption is proved in an example presented in [12] for knight's tour problem.

The results of experiments with knight's tour problem have shown that implementations in Spartan-3 FPGA run 2-3 times faster than software executed on Pentium-4 clocked at 3.4 GHz [12]. Once again communication and FPGA configuration times have not been taken into account.

2.5. Stitt *et al.*

Stitt *et al.* proposed a new synthesis optimization technique, called *recursion flattening*, which eliminates recursive function invocations by unrolling and inlining recursive calls [13]. Recursion flattening is realized in two steps. First, the maximum depth of recursion is calculated as a function of a given set of inputs. Then recursive calls are inlined until the required depth of the recursion is reached.

The suggested technique is not capable of eliminating all recursion but succeeds for some recursive algorithms. Actually, the main difficulty is detecting recursion depth that can only be done if a certain set of conditions is satisfied (such as that every recursive call must modify at least one variable that is used in checking if a base case has been reached). An example of algorithm whose recursion depth cannot be determined statically (before runtime) is quick sort [13].

A high-level synthesis tool was developed that performs recursion flattening and outputs register-transfer level VHDL code [13]. The generated VHDL code was further synthesized and implemented on an FPGA of Xilinx Virtex-4 family. The results of experiments with several recursive benchmarks have shown that flattened recursive hardware (running at a frequency ranging from 100MHz to 400 MHz) was on average 75x faster than software executing iterative versions of the same algorithms over ARM926EJ running at 400 MHz. Recursive-flattened hardware was also on average 6.5x faster than the iterative hardware and occupied a bit more resources.

The main problem with this method is that, according to [13], various instances of a recursive function (flattened recursion) are synthesized to parallelized circuits. If so, it is not clear how to deal with different instances of inlined functions that require simultaneous access to memory.

3. COMPARISON OF THE PROPOSED METHODS

As it was shown in the previous section neither of the suggested methods for implementing recursion in reconfigurable hardware can be classified as a clear winner. Efficiency of multi-thread execution and speculative execution of Mariyama *et al.* greatly depends on particular problem characteristics and is very limited by memory bandwidth. Those methods that rely on stack (Sklyarov *et al.* and Ninos *et al.*) have a restricted parallelizability. However, their main advantage is that they can easily be applied for implementing any algorithm and therefore concept-to-implementation time is short. The recursion unrolling method of Ferizis *et al.* potentially achieves high level of parallelizability but tends to consume much more hardware resources than other methods. Moreover it is very platform-specific (because of the need to support run-time reconfiguration), lacks clearly defined memory access component and is difficult to conceptualize and apply on practice. The recursion flattening method of Stitt *et al.* is not suitable to all recursive algorithms because not always the maximum recursion depth can be determined. Simultaneous access to memory by different instances of inlined functions is also a problem.

As it has been shown all the methods differ in the level of parallelism supported [14]. Sklyarov *et al.* and Ninos *et al.* explore statement-level parallelism (SLP), where sets of nearby statements are processed simultaneously. The amount of SLP is limited by characteristics of a particular algorithm. Mariyama *et al.* [3] and Stitt *et al.* explore pipelining with statements being executed in an overlapped sequence. As in the previous case, the maximum amount of achievable parallelism is limited by inter-statement dependencies. Finally, Mariyama *et al.* [4] and Ferizis *et al.* explore process-level parallelism (PLP) augmented with pipelining, where multiple instances of recursive function are dispatched simultaneously. The efficiency of this approach also depends on the algorithm and is very restricted by inter-process dependencies and memory bandwidth. Moreover, PLP is difficult to identify automatically [14].

The methods also differ according to approaches to exploring concurrency. Proposals of Mariyama *et al.* [3], Sklyarov *et al.*, Ferizis *et al.* and Ninos *et al.* force the designer to identify explicitly which statements/functions will be executed in parallel, while Mariyama *et al.* [4] and Stitt *et al.* provide automated high-level synthesis compilers that generate synthesizable HDL code.

The most important characteristics of the reviewed methods, such as the supported level of parallelism, approach to concurrency, whether a stack is required, amount of occupied hardware resources, ease of use, and limitations to applicability, are summarized in Table 1.

Table 1. Principal characteristics of different methods.

	Mariyama *et al.*	Sklyarov *et al.*	Ferizis *et al.*	Ninos *et al.*	Stitt *et al.*
Level of parallelism	pipelining process-level	statement-level	process-level	statement-level	pipelining
Approach to concurrency	designer [3] compiler [4]	designer	designer	designer	compiler
Stack required	yes [3]	yes	no	yes	no
Occupied hardware resources	medium-large	medium	large	medium	medium-large
Ease of use	not easy [3] easy [4]	easy	difficult	easy	easy
Applicability	limited by data dependencies between the results of a recursive call and subsequent function's statements	can be applied to any recursive function	requires run-time reconfiguration	fully supports only direct recursion	recursion depth must be determinable before run-time

It is known that recursion can always be substituted by iteration, but we believe that recursion should be used when it permits a clearer specification to be provided. Therefore, further work has to be developed to guarantee a simple and efficient support for recursive functions in reconfigurable hardware. One of the potential ways is to explore techniques for improving the performance of recursive computations that have been proposed for optimizing software compilers. For example, Rugina *et al*. noticed that a typical recursive function spends too much time in divide and combine phases instead of performing useful computations [15]. Therefore, Rugina *et al*. proposed a temporary recursion unrolling targeted at increasing the size and applicability of base cases. As a result, large base cases can automatically be generated eliminating in this way a big number of recursive calls and improving the overall performance. We believe that a combination of such pre-processing techniques with an easy-to-implement stack-based approach can lead to very good results in reconfigurable hardware.

4. CONCLUSION

This paper is dedicated to the description and comparison of different approaches to implement recursion in reconfigurable hardware. The analysis leads to the following conclusions:

- Practically all the methods reported in this paper were tested on very small problem instances and it is impossible to draw conclusions about their scalability. It seems that only stack-based methods are fully scalable.
- It is quite difficult to compare the results that have been achieved because just a few proposals fully reveal all the details and make the projects publicly available and the results reproducible.
- The majority of methods require the designer to identify statements/functions to be executed in parallel. Just two proposals include a high-level synthesis tool that automates this task.
- The speedups achieved by reconfigurable hardware compared to software are significant just for certain classes of algorithms (and it is not clear whether they account the hardware configuration time). Consequently, although many interesting and worthwhile methods have already been proposed, innovative approaches still need to be explored in the reconfigurable hardware domain.

5. REFERENCES

[1] F.M. Carrano, *Data abstraction and problem solving with C++: walls and mirrors*, 5th ed., Addison-Wesley, 2007.

[2] J.V. Nobble, "Recurses!", *Computing in Science & Engineering*, May/June 2003, vol. 5, issue 3, pp. 76-81.

[3] T. Maruyama, M. Takagi, T. Hoshino, "Hardware implementation techniques for recursive calls and loops", in *Proc. of the 9th International Workshop on Field-Programmable Logic and Applications* - FPL'99, Glasgow, UK, August/September 1999, pp. 450-455.

[4] T. Maruyama, T. Hoshino, "A C to HDL compiler for pipeline processing on FPGAs", in *Proc. of IEEE Symposium on Field-Programmable Custom Computing Machines* FCCM'2000, CA, USA, 2000, pp. 101-110.

[5] V. Sklyarov, "FPGA-based implementation of recursive algorithms," *Microprocessors and Microsystems. Special Issue on FPGAs: Applications and Designs*, vol. 28/5-6, pp. 197–211, 2004.

[6] V. Sklyarov, I. Skliarova, B. Pimentel, "FPGA-based Implementation and Comparison of Recursive and Iterative Algorithms", in *Proc. of the 15th International Conference on Field-Programmable Logic and Applications* - FPL'2005, Finland, August 2005, pp. 235-240.

[7] I. Skliarova, V. Sklyarov, "Recursive versus Iterative Algorithms for Solving Combinatorial Search Problems in Hardware", in *Proc. of the 21st International Conference on VLSI Design* – VLSI Design'2008, Hyderabad, India, January 2008, pp. 255-260.

[8] H. ElGindy, G. Ferizis, "Mapping basic recursive structures to runtime reconfigurable hardware", in *Proc. of the 14th International Conference on Field-Programmable Logic and Applications* - FPL'04, 2004, pp. 906-910.

[9] H. ElGindy, G. Ferizis, "Mapping basic recursive structures to runtime reconfigurable hardware", technical report, July 2004, available online at: ftp://ftp.cse.unsw.edu.au/pub/doc/papers/UNSW/0419.pdf.

[10] G. Ferizis, H. ElGindy, "Mapping recursive functions to reconfigurable hardware", in *Proc. of the 16th International Conference on Field Programmable Logic and Applications* - FPL 06, Madrid, Spain, August 2006, pp. 283-288.

[11] G. Ferizis, "Mapping recursive functions to reconfigurable hardware", Ph.D. thesis, University of New South Wales, Australia, 2005.

[12] S. Ninos, A. Dollas, "Modeling recursion data structures for FPGA-based implementation", in *Proc. of the 18th International Conference on Field Programmable Logic and Applications* – FPL'08, Heidelberg, Germany, September 2008, pp. 11-16.

[13] G. Stitt, J. Villarreal, "Recursion flattening", in *Proc. of the 18th ACM Great Lakes symposium on VLSI* – GLSVLSI'08, FL, USA, May 2008, pp. 131-134.

[14] S.A. Edwards, "The Challenges of Synthesizing Hardware from C-Like Languages", *IEEE Design & Test of Computers*, vol. 23, issue 5, September-October 2006, pp. 375-386.

[15] R. Rugina, M. Rinard, "Recursion unrolling for divide and conquer programs", in *Proc. of 13th International Workshop on Languages and Compilers for Parallel Computing* - LCPC'2000, NY, USA, August 2000, pp. 34-48.