Are HLS Tools Healthy?

The C-Cubed Project

Michael Dossis
Dept. of Informatics Engineering
TEI of Western Macedonia
Kastoria, Greece
mdossis@yahoo.gr

Georgios Dimitriou
Dept. of Electrical and Computer Engineering
University of Thessaly
Volos, Greece
dimitriu@uth.gr

Abstract— **The increasing complexity of Application Specific Integrated Circuits (ASICs) and Systems-on-Chip (SoCs) that incorporate custom and standard embedded core IP blocks dictates the need for a new generation of automated and formal system EDA tools and methodologies. High-Level Synthesis (HLS) plays a critical role in the required Electronic System Level (ESL) methodologies. However, most of the available academic and commercial High-Level Synthesis (HLS) tools still do not play an established role in the system and hardware engineering teams. This is true for a number of practical reasons, analyzed and discussed in this work. The present article is a practical perspective of the required fully automated and formal tools, which are needed to constitute integral parts in Electronic Design Automation (EDA) flows. In addition, this article is a useful guide to the system engineer who wants to familiarize with HLS tools and to select the appropriate tool for the everyday engineering practice. The advanced HLS toolset that is analyzed in this paper is developed by the first author, its C-frontend by the second author, and they are both based on formal methods and fully automated techniques, thus they guarantee the correctness of the synthesized hardware implementations. This paper completes with a number of experiments that were executed using the author's methodology and they are used to evaluate the specific HLS tools. Consequently, a number of conclusions are drawn as well as suggestions for the future directions of HLS technology. In this way, what is practically needed by the hardware systems engineering community is outlined at the end of the paper.**

Keywords-High-Level Synthesis; Formal Methods; EDA; ESL; Hardware Compilers; Digital Hardware Design

I. INTRODUCTION

Nowadays, digital integrated microelectronics feature extremely complex components, control/design hierarchy and interconnection schemes. Such components constitute dominant parts in embedded, high-performance and portable/consumer electronic computing systems. It is important to note that as the number of registers in a design increases linearly, the development and verification effort and engineering time increase exponentially. Such complexity cannot be dealt anymore with traditional design methods such as RTL coding/synthesis/simulation, since they suffer from highly iterative design flows and prolonged product development. Often, due to these problems, the products miss the market windows and engineering investment is lost. During the last couple of decades, commercial and academic organisations have invested in High-Level Synthesis (HLS) in order to achieve automation, quality of implementations and shortened specification-to-product times [1-5, 7]. However, there are a number of practical and engineering issues related to the existing HLS tools. First, they produce much lower quality of hardware implementations compared to manual techniques. Second, their hardware models that are difficult to handle, and with a lot of platform assumptions and transformation heuristics that assume an ideally-matched target environments, failing to produce real-life configurations.

Existing and well-defined formal methodologies, such as logic programming [6], compiler generators, artificial intelligence and software compilers etc, can benefit HLS tools. The most understood and explored HLS tasks are high-level optimizations, scheduling, allocation and binding [1-5, 7]. High-level transformations resemble software compiler optimizations. Allocation is the selection of functional units and storing resources for the data and operations objects found in high-level program code. Binding is the actual mapping of the above units to real hardware elements such as flip-flops, latches and combinatorial blocks such as functional operator hardware units. Scheduling is the arrangement of elementary operations to Finite State Machine (FSM) states or in other words real system's clock cycles. However, the optimization of real-world, complex applications and their mapping onto custom hardware fail to produce competitive (with the manually designed) implementations due to the tools' inability to handle arbitrary, complex, nested control flow and large data objects, as well as sophisticated interfaces through complex hierarchy and module configuration.

II. HLS TOOLS, AND ENGINEERING PRACTICE

High-Level Synthesis research commenced in the 80s, with the first academic and industrial linear processing HLS tools appearing in the early 90s. Usual problems that HLS researchers were called to handle, were allocation, scheduling and binding, as mentioned above. The most cumbersome of these problems to deal with is the building of a reliable scheduler [3]. It is well known that when the system complexity increases linearly (e.g. in terms of number of FSM states), the complexity of the scheduler increases exponentially. For extremely complex applications, scheduling is NP-complete [3, 7]. The difficulty to handle complex code

becomes serious, and even prohibitive especially when complex control flow hierarchy (e.g. nested while, if/then and for loops) are encountered in the source code model [4, 7].

Although the need for automation is pressing, HLS tools are still not widely accepted in the industrial practice because of their low quality of results, particularly for real applications with complex module/control-flow hierarchy as stated before. Usually, the specification coding style has a severe impact on the type, template and quality of the delivered implementation. For real-life applications, the execution time of the synthesis transformations (front-end compilation, algorithmic transformations, optimizing scheduling, allocation and binding), increases exponentially with a linear increase of the design size [3-5]. This demanded the use of heuristics to cut down processing time, leading to suboptimal solutions.

Most of the available HLS tools impose severe extensions or restrictions (e.g. exclusion of while loops) on the programming semantic model of the subset that they accept as specification. Heuristics are applied on the HLS transformations (e.g. guards, speculation, loop shifting, trailblazing) [2]. These are suitable for only linear/dataflow dominated (e.g. stream-based) applications, such as DSP, image processing and video/sound streaming. Once again they cannot handle any of the excluded programming constructs, such as subprograms, records, while loops, and loop breaks.

The most popular commercial HLS tools include the Catapult-C from Calypto (previously developed by Mentor Graphics), and Cynthesizer from Forte Design Systems. They all accept as input a small subset of the System-C and C++ languages. These tools have very complicated for the average user interfaces, and they are the most expensive of their class since they are licensed for something less than 300K dollars per year. So, these E-CAD systems are inaccessible for most of the small and medium sized ASIC/FPGA design SMEs.

Other commercial or industrial HLS tools are the Symfony C compiler from Synopsys, the Impulse-C from Impulse Accelerated Technologies, the CyberWorkBench from NEC, the C-to-silicon from Cadence, and the free web-based tool C-to-Verilog from an Israel-based group. These tools are mostly used only internally by the producing organization, and they are otherwise not well-known amongst the engineering community for reasons that were explained above.

The most well-known academic or research-based HLS efforts are the SPARK tool [2] which accepts as input a small subset of the ANSI-C language (e.g. while loops are not accepted), and a conditional guard based optimization method [7]. The latter set the basis for optimizing conditional source code at the beginning of the previous decade.

Recent research efforts include a multi-speculative approach to synthesize complex adders during datapath synthesis, which again contributes only towards linear flow oriented designs [8], a fixed-point accuracy analysis and optimization of polynomial data-flow graphs with respect to a reference model that is found in many DSP applications [9], a technique to improve nested loop pipelining for HLS, called Polyhedral Bubble Insertion [10], an equivalence checking method of FSMs with datapaths based on value propagation over model paths, for validation code motion, usually applied during the HLS scheduling phase [11], a formal method for accurate high-level casting of optimal adders and subtractors [12], and an exploration approach, called Spectral-aware Pareto Iterative Refinement, that uses response surface models (RSMs) and spectral analysis to predict the design quality without costly architectural synthesis procedures [13].

III. NEED FOR FORMAL TECHNIQUES

The issues discussed above dictate the need for the incorporation of intelligent and formal HLS techniques on the source-to-implementation optimizing transformations. In this way, the produced hardware implementations become correct-by-construction. Only top behavioral level verification (e.g. with rapid compile and execute of the specs) will be needed, against spending weeks and months, on lengthy RTL or annotated gate simulations that is required with traditional methods. In our approach, constraints and designer options can be applied by the user on the automatic HLS transformation, such as the number of available resources, the length of the desired schedule, the type of the micro-architecture, the generated HDL code as well as the inclusion of custom (e.g. arithmetic) logic functions throughout the HLS compilation, avoiding predefined target platforms or synthesis heuristics.

IV. THE C-CUBED EDA HLS FRAMEWORK

The first author has designed and developed an intelligent HLS toolset [4] that optimizes operations into control steps, achieving the maximum functional parallelism in the synthesized implementation [5]. The C-Cubed compiler employs an advanced scheduler called PARCS, with formal techniques such as logic programming [6] and RDF subject-predicate-object relations [7]. Thus, the delivered implementations are correct-by-construction.

A detailed description of the prototype optimizing C-Cubed synthesizer can be found in [4]. The C-Cubed tool employs formal techniques such as predicate logic [6], RDF relations and XML schema validation to improve the synthesis results. The usability and correctness of the C-Cubed HLS toolset were evaluated with a large number of benchmarks, a few of which are discussed in the following sections of this paper.

The C-Cubed ADA/C HLS design and verification flow, includes the front-end and back-end HLS tools, and the GNU C/ADA integrated compiler, development and verification environment, as shown in Figure 1. The standard programming set of the ADA and ANSI-C language are accepted by the C-Cubed synthesizer. The front-end compiler is a compiler-generator parsing and syntax processing system with all the standard software compiler optimizations. The back-end compiler is based on logic programming inference engine rules and it includes the formal PARCS scheduler and optimizer. PARCS attempts always to parallelize as many as possible operations in the same control step, obeying to data/control dependencies. Nevertheless, the tool can be guided by external module and operator specific resource constraints.

The C-Cubed design and verification flows are outlined in Figure 1. The designer, who should be familiar with either

ANSI-C or ADA programming, uses standard programming language constructs such as routines, if/then, while/for loops, arrays and records (structures) complex control constructs and as nested as needed control flow types to build the executable specification model in any of these languages. Before execution of the C-Cubed compiler, the user verifies the correctness of the given algorithm by simply compiling and executing the source code model along with any benchmark-specific code (e.g. file-I/O and conversion routines). After debugging his specification code the user passes it to the C-Cubed framework for hardware synthesis. C-Cubed tools are fully automatic, without the slightest code modification, delivering provably-correct hardware implementations in VHDL or Verilog RTL. This usually takes from seconds to a few minutes depending on the complexity of the specification model. The applications are optimized with the PARCS scheduler and high-quality RTL implementations are generated. These RTL code modules are FSM-controlled datapaths of optimized operators and functional unit sets. They are fully-synthesizable to hardware with any of the available academic or commercial RTL synthesizers without the slightest intervention or modification of the synthesized RTL model

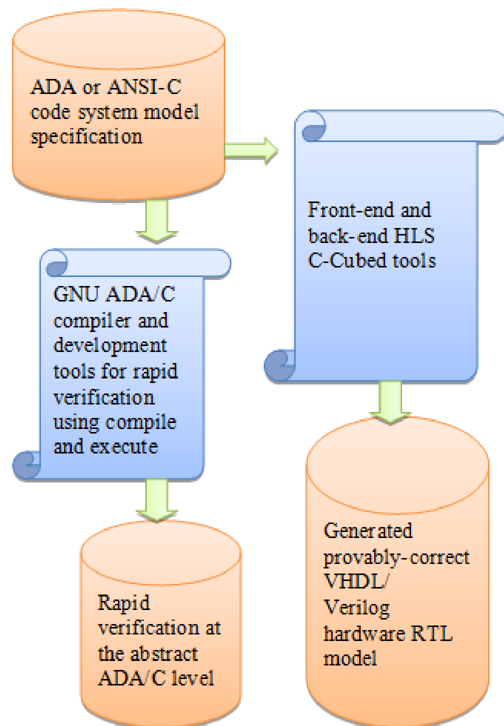

Fig. 1. C-Cubed HLS design and verification flow

. The produced RTL is highly readable and including comments and object names reflecting the C or ADA source code names and user identifiers. Thus the designer can easily trace the code names into the delivered RTL statements and if he/she desires to execute RTL simulations for further confidence on the produced implementations, although this is not needed, due to the formality of the employed synthesis transformations by the C-Cubed framework.

V. EXPERIMENTAL RESULTS

Arbitrary input ADA or ANSI-C code using any of the standard programming language constructs is rapidly and automatically synthesized into functionally-equivalent RTL VHDL/Verilog hardware implementation, using the C-Cubed framework. A great number of applications were synthesized with the C-Cubed toolset [4], some of which are discussed here. In every case, the functionality of the produced hardware implementations matched that of the input subprograms, which was expected due to the formal nature of C-Cubed [4].

After building and verifying the benchmarks in ADA or C code, they were synthesized into VHDL/Verilog RTL. Since the C-Cubed transformations utilize formal techniques there is no need to simulate the generated RTL. Nevertheless for proving this argument in experimental practice we have simulated all the generated RTL tests to ensure that they are functionally equivalent to that of the source code. A RTL simulation snapshot of a computer graphics benchmark generated HDL code is shown in Figure 2. This algorithm is based on the DDA method to draw a straight line on a digitized screen with pixels. Figure 2 demonstrates the completion of the generated hardware's function with the synchronized done/results_read signal event, as well as all the result data storing into the external memory.

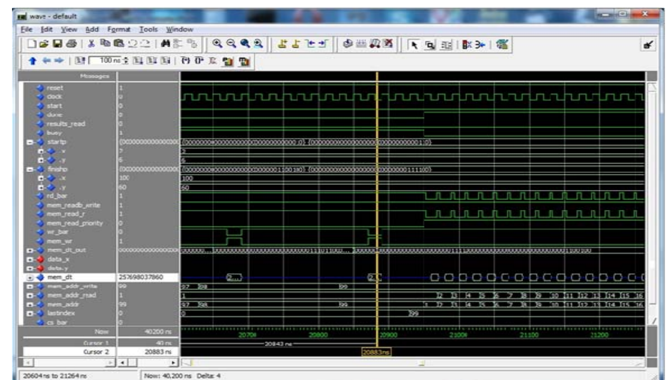

Fig. 2. The graphics drawing line benchmark simulation

All synthesized applications were implemented on Xilinx FPGAs using the Mentor Graphics Precision RTL Synthesis 2013b.15_64-bit (Production Release) tool, and the Xilinx ISE place&route utilities, targeting a Xilinx -family VIRTEX-4 -part 4VSX55FF1148 -speed -12 technology FPGA device. Table I shows the state reduction, using the PARCS optimizer for six benchmarks, the line drawing algorithm, a MPEG engine, a FIR filter, a diff. eq. solver a RSA crypto-algorithm and a nested loops benchmark. In some cases with complex control flow, the state reduction optimization rate reaches up to 41 per cent, or the initial FSM. The state reduction for these benchmarks is shown graphically in Figure 3. All the tests were compiled with C-Cubed in less than 1 minute.

Amongst the above applications, the MPEG engine comprises of a FSM with more than 1500 states. It has been designed in standard ANSI-C code and the C frontend facility of C-Cubed tools (developed by the second author of this paper) was used to convert it into ADA for synthesis. Such a

complex design is practically impossible to design and verify directly in RTL. Therefore the contribution of the C-Cubed technology in this direction is invaluable. Tables II and III provide resource use statistics of the FPGA hardware of the MPEG video compression application. Timing constraints that were used for the synthesis and implementation with a clock frequency target of 100 MHz, which was achieved with sizeable slack. The RTL synthesis with Precision took less than 3 seconds real time and the C-Cubed frontend optimization was completed in 45 seconds for the most complex MPEG test. The produced MPEG RTL for the initial schedule included 9145 VHDL lines, and the optimized by PARCS schedule was a list of 8162 VHDL lines. Figure 4 shows the Xilinx resource statistics for the initial and the optimized MPEG design schedules in a graphical way.

The C-Cubed HLS compiler can be guided by a number of options, such as local and global resource constraints, the target HDL, the massively parallel or FSM+datapath architecture, and the location of complex data objects (such as arrays) on external or embedded memories. This is the reason that Table I has multiple experiment lines with "embedded" or "external" memory options. Of course fruitful trade-offs can be extracted easily, with experimenting with the high performance embedded memory vs the more economic and realistic for small FPGA devices external memory location of the complex data structures. In any case, the location to external memory is configured for any of the data objects of the source code, using a set of elegant memory options file, and without altering the source code as it happens to other, antagonistic HLS tools.

TABLE I. STATE REDUCTION OPTIMIZATION USING PARCS

Module name	Hardware Implementation Statistics		
	Initial FSM states	**PARCS parallel FSM states**	**State reduction**
line-drawing design	17	10	41%
MPEG top routine (with external memory)	1697	1380	19%
FIR filter main routine	17	10	41%
Differential equation solver	20	13	35%
RSA main routine	16	11	31%
nested loops 1st routine	28	20	29%
nested loops 2nd routine (with embedded memory)	36	26	28%
nested loops 2nd routine (with external memory)	96	79	18%
nested loops 3rd routine	15	10	33%
nested loops 4th routine	18	12	33%
nested loops 5th routine	17	13	24%

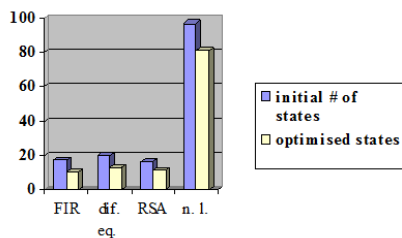

Fig. 3. State reduction rates of some benchmarks in graphical view

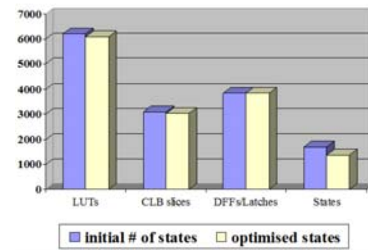

Fig. 4. Xilinx Implementation Statistics for the intial and the optimized (PARCS) FSM state schedule

TABLE II. XILINX VIRTEX-4 FPGA IMPLEMENTATION OF THE INITIAL, UNOPTIMIZED MPEG SCHEDULE

Resource	Initial (Un-optimized) Schedule Xilinx Utilization Statistics		
	Used Parts	**Available Parts**	**Utilization rate**
IOs	3093	640	483.28%
Global Buffers	1	32	3.13%
LUTs	6193	49152	12.60%
CLB Slices	3097	24576	12.60%
Dffs or Latches	3836	49152	7.80%

TABLE III. XILINX VIRTEX-4 FPGA IMPLEMENTATION OF THE OPTIMIZED MPEG SCHEDULE

Resource	PARCS (optimized) Schedule Xilinx Utilization Statistics		
	Used Parts	**Available Parts**	**Utilization rate**
IOs	3093	640	483.28%
Global Buffers	1	32	3.13%
LUTs	6088	49152	12.39%
CLB Slices	3044	24576	12.39%
Dffs or Latches	3836	49152	7.80%

VI. PROSPECTS OF HLS

What about the future? What should be the future directions in order to achieve industrial strength HLS? More input programming languages (e.g. C++, System-C, UML, Fortran, Delphi-Pascal, Java, SystemC, Python, etc.) and a more globalized and integrated use of formal techniques throughout the flow of the HLS toolset are needed in order to bring practical results with acceptable quality of HLS results. The automatic and formal nature of future HLS research is critical in order to be accepted by the engineering community. Combining formal synthesis with formal verification is the only way to deal with the extrapolated complexities of nowadays integrated digital systems. Moreover, HLS methodologies need to be more general and adaptable to the configuration of different engineering environments, to raw programming code of high-level popular languages such as C, C++ and ADA and to various targeted computing architectures and of course to all of the established industrial backend flows. Furthermore, new methods need to be investigated for merging multi-cycle operation models with pipelined designs and more advanced programming constructs such as parallelism in the source code, pointers, and dynamic data structures.

HLS can play a critical part in the engineering practice via the re-use and acceptance as input of existing hardware and software IP. To achieve this, a wide compatibility of HLS input/output with languages and formats is required, as well as

rapid prototyping capability to the future electronics product development. Furthermore, arbitrary and complex module and control flow program code in the designer's set of system models need to be transformed with ease, speed and comparable quality with manual methods, into the required software and hardware implementations. In this way, highly-complex applications will motivate the design engineers to use HLS tools and hardware-software co-design, in their everyday practice. The most important of all: the HLS transformations need to be based on formal techniques so that lengthy RTL and gate-level simulations will be avoided and rapid development cycles will be realized, with free-of-bugs, and first-time-right products. In this way, complete system hardware/software co-design flows will be realized into every day engineering practice. We believe that our C-Cubed approach is an important step towards achieving the above goals.

VII. CONCLUSIONS AND FUTURE WORK

Very often, the assumptions that many existing HLS tools make (about the targeted technology attributes, the architectural template, the HDL code style, the timing and power consumption, the available operator and data resources as well as the type of communication with the external environment components such as shared memories), produce disappointing synthesis results, since there is still no established methodology for feeding target technology characteristics back into the core of the HLS transformation process (although some academic attempts to model this problem have been made). In most cases these target implementation characteristics need to be fed into the synthesis flow and guide the complex synthesis transformations of the HLS tool, which makes the synthesis process manual, heavily interactive, cumbersome, prone to errors and very slow.

This paper contributes towards the understanding of practical issues of contemporary HLS tools and underlines the future directions and achievements which should be envisaged. The most important contribution of this work is a HLS ESL tool, that automatically, rapidly, formally and optimally transforms input algorithmic, raw, general, arbitrary, widely accepted and re-used (from software engineering) high-level program code into optimal RTL hardware implementations. The C-Cubed synthesizer is making an important step towards the requirements for the above achievements and a number of related projects are under-way to deliver better synthesis results with readable RTL code and better visibility of the design's attributes and algorithmic features. Of course, this is not the only attempt to deal with the complexities of the HLS transformations and there have been a number of research projects that target a better engineering environment to alleviate the frustrations of industries about dealing with development results that are just too late to hit the market window for many electronics products. Moreover, we can learn from other research efforts and also guide them with our own experience through the complexities and issues of HLS research with the aim to deliver better EDA tools to the design engineers and alleviate their highly-complex tasks to deliver quickly and correctly high-quality hardware and computing products.

Future work for the C-cubed tools include the inclusion of a number of input language formats such as ANSI-C, C++, SystemC and OpenCL, and a number of output formats like SystemC and cycle-accurate C test benches for fast verification. Also, a number of source code optimizations such as dynamic loop-unrolling and code motion are under development. The use of RDF and XML formalisms in the automatic validation of the internal state and structures of the C-Cubed compiler is under investigation and soon will be integrated into the standard C-Cubed framework. Moreover, there is ongoing work on the use of parameterized components, libraries, and optimized arithmetic units as well as multi-cycle operators to help in the direction of advanced pipelined, multi-operational chained designs with high performance and optimized nested loops.

REFERENCES

[1] B. Le Gal, E. Casseau, S. Huet, "Dynamic memory access management for High-Performance DSP applications using high-level synthesis", IEEE Transactions on Very Large Scale Integration (VLSI) Systems, Vol. 16, No. 11, pp.1454-1464 , 2008

[2] S. Gupta, R. K. Gupta, N. D. Dutt, A. Nikolau, "Coordinated parallelizing compiler optimizations and high-level synthesis", ACM Transactions on Design Automation of Electronic Systems, Vol. 9, No. 4, pp. 441–470 , 2004

[3] R. A. Walker, S. Chaudhuri, "Introduction to the scheduling problem", IEEE Design & Test of Computers, Vol. 12, No. 2, pp. 60–69, 1995

[4] M. F. Dossis, "A formal design framework to generate coprocessors with implementation options", International Journal of Research and Reviews in Computer Science, Vol. 2, No. 4, pp. 929-936, 2011

[5] P. G. Paulin, J. P. Knight, "Force-directed scheduling for the behavioral synthesis of ASICs", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 8, No. 6, pp. 661–679, 1989

[6] U. Nilsson, J. Maluszynski, Logic Programming and Prolog, John Wiley & Sons Ltd., 2nd Edition, 1995

[7] A. A. Kountouris, C. Wolinski, "Efficient scheduling of conditional behaviors for high-level synthesis", ACM Transactions on Design Automation of Electronic Systems, Vol. 7, No. 3, pp. 380–412, 2002

[8] A. A. Del Barrio, R. Hermida, S. O. Memik, J. M. Mendias, M. C. Molina, "Multispeculative addition applied to datapath synthesis", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 31, No. 12, pp. 1817-1830, 2012

[9] O. Sarbishei, K. Radecka, "On the fixed-point accuracy analysis and optimization of polynomial specifications", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 6, pp. 831-844, 2013

[10] A. Morvan, S. Derrien, P. Quinton, "Polyhedral bubble insertion: a method to improve nested loop pipelining for high-level synthesis", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 32, No. 3, pp. 339-352, 2013

[11] K. Banerjee, C. Karfa, D. Sarkar, C. Mandal, "Verification of code motion techniques using value propagation", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 33, No. 8, pp. 1180-1193, 2014

[12] R. Sierra, C. Carreras, G. Caffarena, C. A. López Barrio, "A formal method for optimal high-level casting of heterogeneous fixed-point adders and subtractors", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 34, No. 1, pp. 52-62, 2015

[13] S. Xydis, G. Palermo, V. Zaccaria, C. Silvano, "SPIRIT: spectral-aware pareto iterative refinement optimization for supervised high-level synthesis", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 34, No. 1, pp. 155-159, 2015