High-Level Synthesis: A Practical Perspective

Michael Dossis*

Department of Informatics Engineering, TEI of Western Macedonia, Kastoria Campus, Fourka Area, Kastoria, GR 52 100, Greece

Abstract

The current complexity of custom and embedded core or IP integrated electronics demand for a new generation of automated system design and development methods. High-Level Synthesis plays a critical part of such automated methods. However, existing HLS tools are not widely accepted by the engineering community for a number of practical reasons. This article is a practical perspective of such issues, and it analyses the reasons for this. Morever, the article is a useful introduction to the system engineer that wants to consider HLS as part of his everyday system design practice. An alternative HLS toolset is presented that the author has developed and which is based on formal methods, thus it guarandees the correctness of the synthesized hardware and system. The paper completes with conclusions and a number of suggestions about the future directions of HLS technology and what is actually needed by the engineering community.

Keywords: High level synthesis; Automation

Introduction

Digital microelectronics found in embedded, high-performance and portable computing systems have highly complex components, design hierarchy and interconnections. This design complexity cannot be dealt anymore with conventional methods such as RTL coding, which suffer from prolonged development times, so often products miss the market windows. During the last couple of decades, commercial and academic organisations have invested in High-Level Synthesis (HLS) and optimisation techniques, so as to achieve design automation, quality of implementations and short specification-to-product times [1,2].

HLS Tools and Practical Problems

Research in High-Level Synthesis started in the 80s and the first robust linear processing HLS tools appeared in the academic and industrial labs, in the early 90s. Important problems that researchers of HLS were called to handle included the allocation, scheduling and binding problems. The most difficult of these three tasks is the building of a reliable scheduler [3]. It is well known that when the system complexity increases linearly, the complexity of the scheduler algorithm increases exponentialy and for some applications, scheduling is NP-complete. This problem became even more critical and difficult in practice when input code with complex module and control flow hierarchy (e.g. nested while and for loops) is to be processed by the HLS tool [4-7].

Existing HLS tools are still not widely accepted by the engineering community because of their poor results, especially for large applications with complex module and control-flow hierarchy. Very often, the programming style of the source code has a severe impact on the quality of the synthesized implementation. For large-scale applications, the complexity of the synthesis transformations (front-end compilation, algorithmic transformations, optimizing scheduling, allocation and binding), increases exponentially, when the design size increases linearly [3], [4,5], leading to suboptimal solutions when synthesis heuristics are employed to cut down the long processing times.

Many existing HLS tools impose proprietary extensions or restrictions (e.g. exclusion of while loops) on the programming model of the specifications that they accept as input, and various heuristics

on the HLS transformations that they utilize (e.g. guards, speculation, loop shifting, trailblazing) [2]. Most of them are suitable for only linear, and dataflow dominated (e.g. stream-based) designs, such as pipelined DSP, image processing and video/sound streaming.

The most important commercial existing HLS tools include the Catapult-C from Calypto (previously developed by Mentor Graphics), and Cynthesizer from Forte Design Systems. They both accept as input a small subset of System-C and C++. Both of these tools are too complicated for the average system developer and they are the most expensive of their class since they are licensed for something less than 300K dollars per year. Therefore, these E-CAD products are very difficult to access for many small 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 Cyber Work Bench from NEC, the C-to-silicon from Cadence, and the free web-based tool C-to-verilog from an Israel-based group. Most of these tools are either used internally by the owner company, or they are not well-established amongst the engineering community for reasons that were explained above.

Amongst the academic or research-based HLS tools 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] which set the basis for processing conditional code in the beginning of the previous decade.

Requirement for Formal Techniques

It concludes that what is needed from a HLS toolset is the incorporation of intelligent and formal techniques in order to apply the source-to-implementation optimizing transformations, and thus turn

*Corresponding author: Michael Dossis, Department of Informatics Engineering, TEI of Western Macedonia, Fourka Area, Kastoria, GR 52 100, Greece, Tel: 44-777-838-92; E-mail: mdossis@yahoo.gr

the produced hardware implementations to correct-by-construction. In this way, only top behavioral level verification (e.g. with rapid compile and execute of the specs) is required, without spending weeks and months, on lengthy RTL or annotated gate simulations. Constraints and other 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.

The C-Cubed EDA HLS Framework

The author has designed and developed an intelligent HLS compiler [4] that includes a scheduler of operations into control steps, achieving the maximum functional parallelism in the synthesized implementation [5]. It employs an advanced HLS scheduler called PARCS, which utilizes formal techniques such as logic programming [6] and RDF subject-predicate-object relations [7], to formally achieve the maximum possible parallelism of operations. In this way, the functionality of the delivered implementations is correct-by-construction.

A detailed description of the above intelligent approach of the prototype optimising CCC synthesizer can be found in [4]. The CCC tool employs advanced techniques such as formal predicate logic [6], RDF relations and XML schema validation to improve the quality of the synthesis results. The usability and correctness of the C-Cubed HLS toolset were evaluated with a large number of benchmarks. The CCC design flow is shown in Figure 1.

The C-cubed ADA HLS design and verification flow, includes the front-end and back-end HLS tools, and the GNU ADA integrated compiler, development and verification environment. The full standard programming construct set of the ADA and ANSI-C language sets are accepted by the CCC synthesizer. The front-end compiler is a compiler-generator parsing and syntax processing system with all the standard
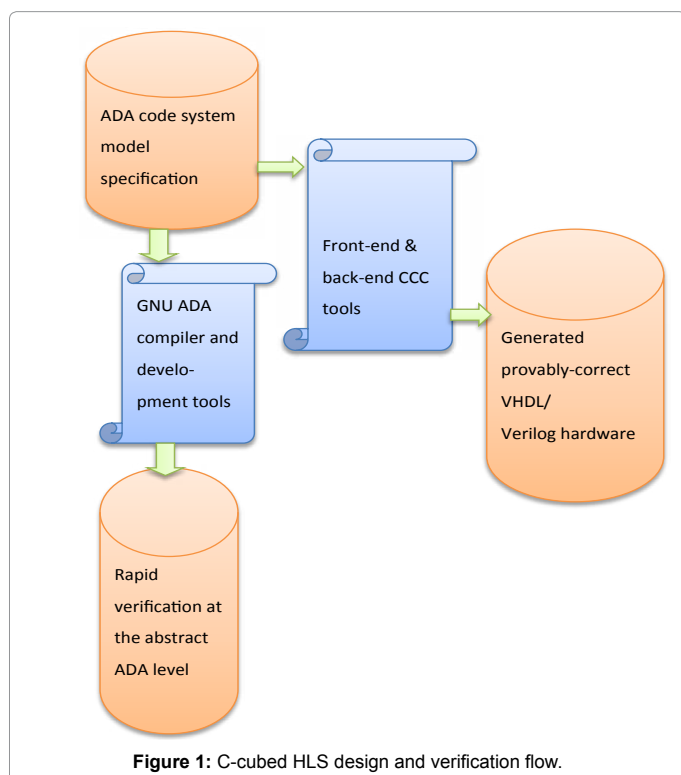
Module name	Initial schedule states	PARCS parallel states	State reduction
line-drawing design	17	10	41%
MPEG 1st routine	88	56	36%
MPEG 2nd routine	88	56	36%
MPEG 3rd routine	37	25	32%
MPEG top routine (with embedded memory)	326	223	32%
MPEG top routine (with external memory)	462	343	26%

Table 1: State reduction optimization using PARCS.

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 parallelise as many as possible operations in the same control step, as far as there are no dependency violations. However, the tool can be driven by external module and operator specific resource constraints (Table 1).

Experimental Results

Arbitrary and general input ADA or ANSI-C code is synthesized into functionally-equivalent RTL VHDL/Verilog hardware implementation. Many applications were synthesized with the C-Cubed toolset [4]. In any case, the functionality of the produced hardware accelerators (coprocessors) matched that of the input subprograms.

After the tests were coded and verified in ADA they were synthesized into VHDL/Verilog RTL. Since the C-cubed tools are based on formal techniques there is no need to simulate the generated RTL. Nevertheless for proving this argument in practice we have simulated all the generated RTL tests to ensure that they feature an equivalent to that of the source code behaviour. A RTL simulation of a computer graphics benchmark generated HDL code is shown in Figure 2. It is shown clearly in this figure that the generated hardware FSM completes its function with the synchronized done/results_read signal event, as well as all the external memory transactions after the completion point, which writes the result into the external memory.

Table 1 shows the state reduction, using the PARCS optimizer for two benchmarks, the line drawing algorithm and the MPEG engine. It is important to mention that in some cases of complex control flow, the state reduction rate reaches up to 41 per cent.

Many benchmarks and tests were synthesized with CCC tools so far. They include a DSP FIR filter, an MPEG engine, and a cryptographic RSA processor. The state reduction for these benchmarks is shown graphically in Figure 3. All the tests were compiled with CCC in less than 10 minutes. The MPEG engine comprises of a FSM with more than 400 states! Such designs are practically impossible to design and verify directly in RTL. Therefore the contribution of the C-cubed technology is invaluable.

Prospects of HLS and the Future

What about the future? What are current and future directions of industrial interest in HLS? More input programming languages (e.g. C++, System-C, UML, Fortran, Delphi-Pascal, Java) and a more globalized use of formal techniques throughout the flow of the HLS toolset are needed in order to bring practical results with acceptable HLS outcomes. Also, HLS methodologies need to be more adaptable to the needs of different engineering environments and many established industrial backend flows.
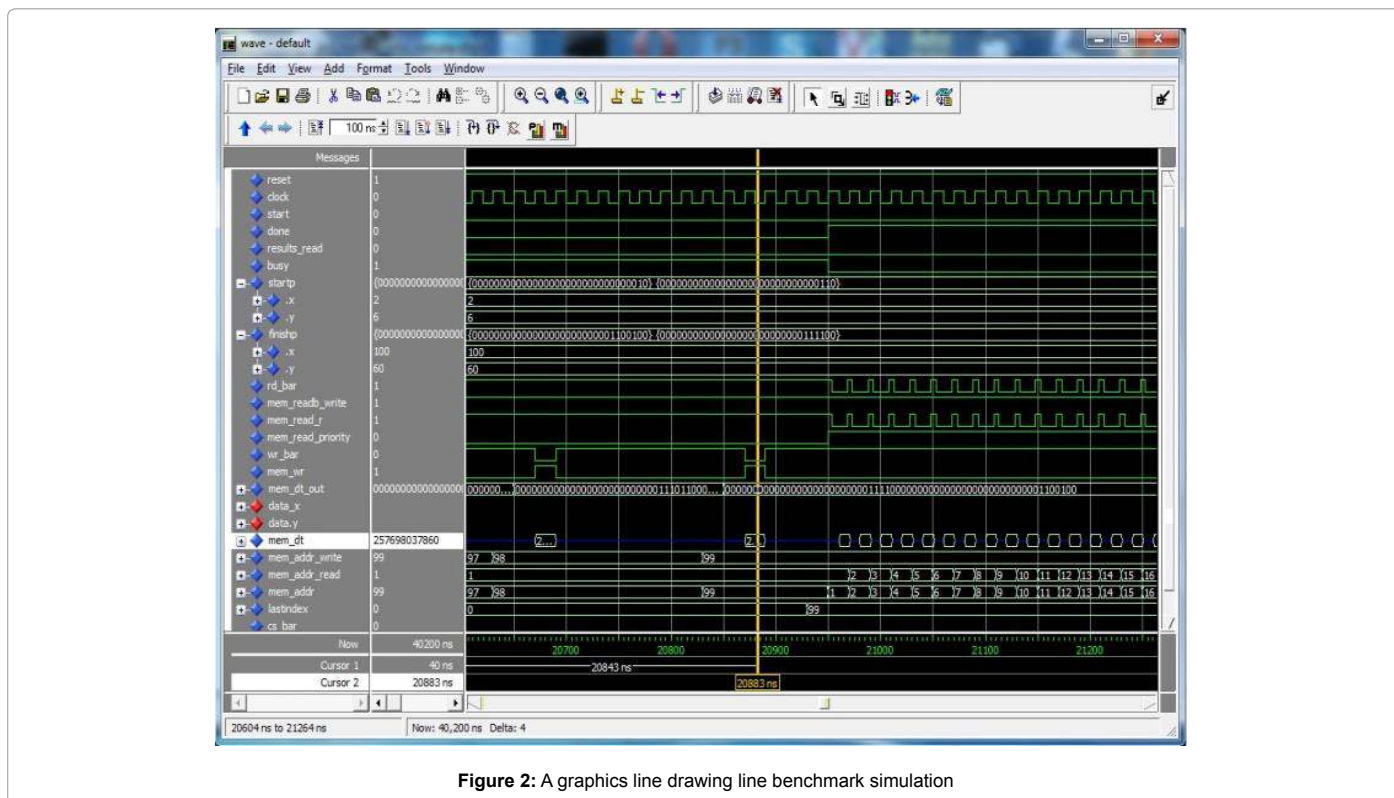

Figure 1: C-cubed HLS design and verification flow.

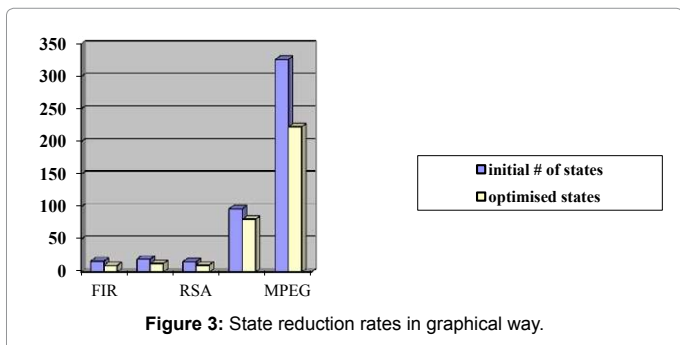**Figure 2:** A graphics line drawing line benchmark simulation

Figure 3: State reduction rates in graphical way.

Another important role that HLS can bring in the engineering practice is the re-use of existing hardware and software IP. To achieve this a wide compatibility of HLS input/output with languages and formats is required to use HLS in practical every-day system engineering, as well as rapid prototyping capability to the future electronics product development. Moreover, arbitrary and complex module and control flow in the designer's set of system models need to be transformed with ease, speed and quality into the required software and hardware implementations.

Conclusions and Future Work

Sometimes the assumptions that many existing HLS tools make about targeted technology attributes such as timing and power consumption, 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 many cases these target implementation characteristics need to be fed into the synthesis flow and guide the complex synthesis transformations of the HLS tool.

The C-Cubed synthesizer is making an important step towards the above requirements 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 it is not the only attempt to deal with the complexities of the HLS transformations and there 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. 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 for quick verification like SystemC and cycle-accurate C. Also, a number of source code optimizations such as dynamic loop-unrolling and code motion are under development.

References

1. Gal BL, Casseau E, Huet S (2008) Dynamic Memory Access Management for High-Performance DSP Applications Using High-Level Synthesis. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 16: 1454-1464.

2. Gupta S, Rajesh KG, Dutt ND, Nikolau A (2004) Coordinated Parallelizing Compiler Optimizations and High-Level Synthesis. ACM Transactions on Design Automation of Electronic Systems 9: 441-470.

3. Walker RA, Chaudhuri S (1995) Introduction to the scheduling problem. IEEE Design & Test of Computers 12: 60-69.

4. Dossis MF (2011) A Formal Design Framework to Generate Coprocessors with Implementation Options. International Journal of Research and Reviews in Computer Science (IJRRCS) 2: 929-936.

5. Paulin PG, Knight JP (1989) Force-directed scheduling for the behavioral synthesis of ASICs. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems 8: 661-679.

6. Nilsson U, Maluszynski J (1995) Logic Programming and Prolog.

7. Kountouris AA, Wolinski C (2002) Efficient Scheduling of Conditional Behaviors for High-Level Synthesis. ACM Transactions on Design Automation of Electronic Systems 7: 380-412.