Comparing C and SystemC Based HLS Methods for Reconfigurable Systems Design

Konstantinos Georgopoulos(✉), Pavlos Malakonakis, Nikolaos Tampouratzis, Antonis Nikitakis, Grigorios Chrysos, Apostolos Dollas, Dionysios Pnevmatikatos, and Ioannis Papaefstathiou

Telecommunication Systems Institute, Campus Kounoupidiana, Technical University of Crete, Chania, Greece
kgeorgopoulos@isc.tuc.gr

Abstract. This paper provides an extensive analysis of the key characteristics, efficiency and overall user-friendliness that stem from the use of two different input methods for the design of *High-Level Synthesis* (HLS) reconfigurable systems, i.e. *C-based* and *SystemC-based*. Each input language has been used within the context of a separate HLS tool that is especially suitable for the particular input method chosen. The study has been based on the use of key fundamental computational and data processing algorithms, while the outlined observations have been drawn by traversing the full HLS flows. The algorithms address both memory- and compute-intensive modules, which are the main cores for numerous modern applications. In this way, detailed observations are made not only with respect to the performance of each approach individually but also against each other. Hence, this paper provides information on an extensive list of issues of major interest to modern reconfigurable systems design engineers, such as design cycle definition, time for HLS flow completion, implementable features, parallelisation, input model complexity and more importantly design effort/time. Moreover, this work presents detailed results on implementation issues as well as implementation guidelines concerning the presented schemes; these guidelines can certainly be used as a reference for any designer implementing such classes of algorithms.

Keywords: High-Level Synthesis · Reconfigurable hardware · C SystemC

1 Introduction

The motivation for this work has been the rapidly developing sub-field of Electronic Design Automation (EDA) [1] tools known as *High-Level Synthesis* (HLS) [3], especially since they play a key role in the design of reconfigurable systems. Their importance becomes greater by the day, a fact that is clearly demonstrated

N. Voros et al. (Eds.): ARC 2018, LNCS 10824, pp. 459–470, 2018.
https://doi.org/10.1007/978-3-319-78890-6_37

by the number of different HLS tools that has currently been proposed and/or marketed [4]. The strength of HLS is found in the ability to generate production-quality Register Transfer Level (RTL) [5] implementations from high-level specifications. In reality, this is a task that is constantly performed manually by design engineers and programmers while HLS promises to automate it, thus eliminating the source of many design errors and accelerating the currently very long development cycle. This design cycle is also proving to be highly expensive due to escalating non-recurrent engineering costs.

Additional benefits include a design source that is truly generic and, thus, more versatile and portable. Working with purely functional specifications, details, such as clock frequencies, technology and micro-architecture, are eliminated, allowing for greater and easier reuse and re-targeting of the developed models and functional Intellectual Property (IP) cores.

Two of the most prominent input methods in HLS reconfigurable systems design are based on using *(i)* C or C-based variants and *(ii)* SystemC. They are both extensively used and constitute the focus of this paper which provides evaluation insights that are beneficial to the growing number of designers that resort to their use.

Since the purpose of this study has been to investigate the use of those two types of input languages in the context of HLS design, the main focus has been placed towards selecting algorithms that are not overly complicated and have a fast turnaround in terms of results. The interest here has been to use algorithms that bring forward key observations related to the two input methods and their behaviour within the context of a HLS tool. Hence, the use cases include both compute-intensive problems and memory-intensive ones, namely (i) *Mutual Information* [6], (ii) *Transfer Entropy* [7], (iii) *List Manager* and, finally, (iv) *Memory Allocator* [8]. The first two are characterised as compute-intensive, List Manager is a memory-intensive algorithm and the Memory Allocator constitutes a combination of both memory- and compute-intensive characteristics.

Hence, the contributions of this paper are based on the use of two different HLS input modelling languages and convey information related to (i) development time, (ii) Lines of Code (LoC), (iii) latency/performance, (iv) anticipation for mathematical operations, (v) learning curve for efficient modelling, (vi) robustness of high-level models and, finally, (vii) parallelisation capabilities.

The paper is organised as follows. Section 2 presents the two different design flows along with their respective characteristics. Section 3 presents a brief description of the algorithms used in the context of this work. Section 4 has performance figures and technical information on all RTL designs and for all the memory- and compute-intensive algorithms. Section 5 presents a thorough comparative analysis based on both the technical results as well as on the qualitative authors' experience. Finally, Sect. 6 concludes the paper.

2 Tools' Flows and Characteristics

The main development stages of the two input method flows are shown in Fig. 1, from the initial algorithm description to bitstream generation.

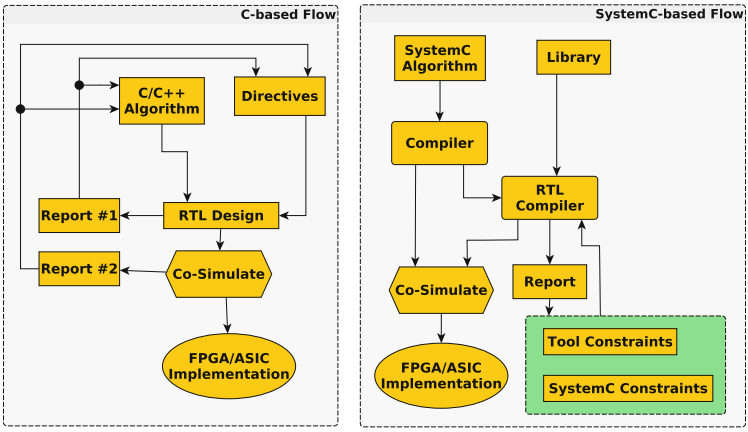

Fig. 1. C-based flow & SystemC-based flow

Note that these two flows do not represent the only way that either a C-based or a SystemC-based HLS design approach can materialise, rather they show the main stages that constitute the design cycle specific to this particular work. The reasons for which these particular flows came about are twofold, *(i)* they empirically settled to be the most efficient means for completing a design cycle, and *(ii)* they were shaped by the nature of the two HLS tools used. However, in order to avoid benchmarking complications, the authors have omitted the names of the tools but are available for clarifications.

In the C-based approach, the process begins with the high-level C code description. Here, the initial software code, most usually untimed C, is subjected to a series of modifications of two major types. The first type refers to the satisfaction of a number of requirements that must be met in order for the high-level model to become compatible with the particular tool's restrictions and characteristics.

The second type of modification is a function of the programmer's/designer's proficiency who develops code optimisations that assist in the extraction of characteristics such as parallelisation and speedup. This is realised in two ways: *(i)* through the use of directives that guide the tool to process certain sections of the code in a certain way and *(ii)* through re-structuring/modifying the initial code in certain ways that will make it easier for the tool to translate the high level model described in C into an efficient low-level equivalent. Consequently, the high-level design description is combined with the user-selected directives in order to generate an RTL model that is accompanied by a detailed report, which points to a number of characteristics of interest, such as, BRAM use, area utilisation, clock speed and others. This information is taken into account in order to re-visit the original high-level model description if desired so.

Alternatively, the process moves onto the stage of co-simulation. Here, the functional and behavioural validation of the RTL-model takes place through the

juxtaposition of the hardware simulation results against those that correspond to the original high level description, i.e. untimed C model. This stage also offers significant performance information, in the form of a second report, i.e. Report #2. It should be noted that performance figures are only good estimates and the actual design may have slightly different performance.

Finally, the designer studies the performance summary of the RTL-model and decides whether additional design iterations are required or whether it is now ready to be processed for bitstream generation. The SystemC-based Flow is similar in its principle. The high level model is subjected to a number of user modifications to assist the compiler, as well as the RTL compiler, achieve the best possible low-level RTL model. Here, the high-level model can be potentially complemented by a selection of constraints specific to the targeted hardware located in a corresponding library file. The advantage here is that the implementation constraints are kept separate from the design's functionality, and, therefore, the same high-level model can be re-applied to different end-products with different requirements and process libraries. Potential low-level model optimisations, such as parallelism and pipelining, are triggered by specialised HLS directives that are in effect embedded in the high-level code while they are also imposed by the high-level code itself. At the end of the *RTL Compiler* stage, a fully synthesizable model is produced. Here, the designer gets a report of the key performance criteria. In this way potential additional improvements can be introduced at the high-level SystemC model and the process is repeated until the designer is satisfied. Finally, a co-simulation activity follows, whereby the behavioural operation of the generated RTL model is verified against the desired one.

3 Use Cases

Mutual Information (MI) [6] measures the extent to which the uncertainty about one of the two is reduced through the information known about the other. Subsequently, Transfer Entropy (TE) [7] measures the amount of directed (time-asymmetric) transfer of information between two random processes. Both MI and TE are correlation metrics fundamental in key applications and scientific principles such as biology, astrophysics, image processing, economics and more. A List Manager (LM) is used for linked list organisation in protocols/algorithms, such as the MPI and Portals [9]. It performs three basic list operations, *search*, *insert* and *delete* and each list is implemented by a head and a tail pointer. Finally, the Memory Allocator (MA) [8] provides ways to dynamically allocate portions of memory to programs at their request, as well as free it for reuse when it is no longer needed. The Memory Allocator used here is the *Buddy Memory Allocator* [10].

4 Results

The results of this work have been categorised into three major types, i.e. (i) *Lines of Code (LoC) & Development Time*, (ii) *Latency* and (iii) *Area Utilisation*. Furthermore, the platforms used for evaluation purposes have been

(i) an *Intel Xeon E5620* operating at 2.4 GHz and with 16 GB of RAM for the *software* execution of the original C and SystemC code (single core and single thread) and (ii) the *Virtex-7 XC7VX690T* FPGA as the target reconfigurable *hardware* implementation device in both the C-based and SystemC-based flows. In addition, the Mutual Information (MI) and Transfer Entropy (TE) results are based on a data size of 20K samples and a number of MI and TE iterations of 100 and 10 respectively; these are typical values in order to get results of high accuracy. Finally, the List Manager (LM) and Memory Allocator (MA) analysis has been based on two different testbenches. The LM testbench performs 128K *insert*, 128K *search* and 128K *delete* operations with a 20 byte header size. On the other hand, the MA testbench contains 1,024 allocations and 512 free operations. Finally, all generated RTL models are in Verilog.

4.1 Development Time and Lines of Code (LoC)

This section offers a quantifiable insight on the gains that a designer ought to anticipate when using either of the C-based or SystemC-based flows for reconfigurable system design according to results related to LoC and total development time; two metrics that are indirectly linked to productivity. LoC results are presented in two figures, Fig. 2 for MI and TE and Fig. 3 for LM and MA.

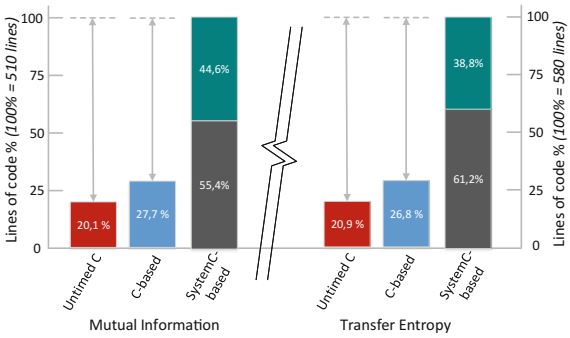

Fig. 2. LoC for Mutual Information (MI) & Transfer Entropy (TE)

The information in both figures refers to the *(i)* original C code, *(ii)* modified HLS C code and *(iii)* HLS SystemC code as percentages with 100% corresponding to the description with the highest LoC number. Hence, Fig. 2 reveals that SystemC requires the longest descriptions for the MI and TE implementations. The reason for this has been twofold, i.e. apart from the necessary SystemC adaptations so that is passes the compiler stage, we also had to account for the support of standard mathematical functions, which are quite common in compute-intensive problems. Hence, the designer has had to not only code-in the description of the target design, but also, develop the code for the mathematical functions required, thereby generating additional code overhead. In terms

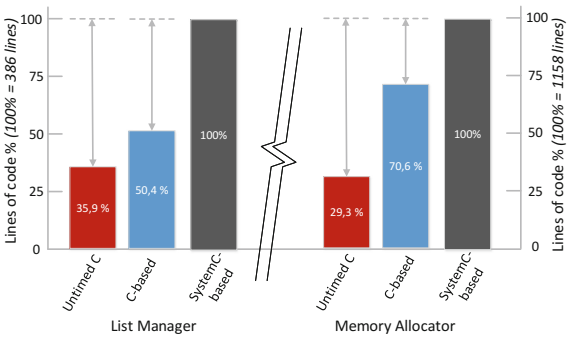

Fig. 3. LoC for List Manager (LM) & Memory Allocator (MA)

of actual numbers, MI requires a SystemC maximum of 510 lines from which 55.4% is the description of the algorithm itself and an additional 44.6% is due to the mathematical function that had to be developed. In contrast, the C-based approach requires 27.7%, i.e. 116 LoC, for a ready-to-be-transformed model. Similarly, the SystemC TE model is described in 580 lines out of which 61.2% is the model itself and 38.8% is the overhead due to the mathematical functions that are missing. Once more, the modified C-based model remains close in size to the original untimed C model thereby underlining a clear advantage of the C-based approach over the SystemC-based approach. Note that even if the additional lines of code for mathematical operations where excluded, SystemC LoC remains greater compared to the C-based description. With respect to LM and MA, Fig. 3, the picture does not change drastically although a distinguishable difference exists. This time, the SystemC approach is not burdened by mathematical function overheads, however, it remains the longest in terms of LoC. For LM, the C-based approach remains noticeably closer to the original untimed C model length, however, in the MA case, C and SystemC are very close and both drift away from the original model's LoC. Finally, although not shown since it has been an anticipated result, the LoC for all developed HLS models is radically, shorter compared to the RTL tool-generated models. In terms of overall development time, results are shown in Table 1 and it becomes apparent that the C-based flow provides the shortest in all four cases. Specifically, it offers synthesisable models for the two compute-intensive algorithms (MI & TE) in less than 1/4 of the time needed by the SystemC model description whereas the difference in the memory-intensive and memory- & compute-intensive algorithms (LM & MA) between the two flows is over 50%. Quite notably, SystemC requires the longest development time even without the need for manually developing mathematical functions for compute-intensive problems, as clearly demonstrated for the memory-intensive cases of LM and MA. Finally, all development times presented have been derived by the same engineers in each use case and the reasoning behind those differences is analytically given in Sect. 5.

Table 1. C-based & SystemC-based design time for all four algorithms

Tool	Algorithm			
	MI	TE	LM	MA
C-based	8 hrs	10 hrs	7 hrs	14 hrs
SystemC-based (math)	20 hrs	20 hrs	-	-
SystemC-based (design)	19 hrs	32 hrs	22 hrs	31 hrs

4.2 Latency

The latency results of this section have been reported in HLS co-simulation using specific reconfigurable hardware specifications that gave the optimal results. Hence, in the case of MI, the best suited Virtex-7 clock frequency for the SystemC-based flow was 115 MHz whereas for the C-based flow was 110 MHz. For TE, the SystemC-based HLS flow clock frequency was 115 MHz whereas for the C-based flow it was 100 MHz, and, finally, for the LM and MA implementations, 303 MHz and 416 MHz, respectively. Table 2 shows the latency for each of the algorithms addressed in this work and presents the corresponding speedup when compared to the software execution on the CPU listed in the beginning of Sect. 4. In addition, the latency/speedup results for LM and MA have been split between the individual operations that each algorithm entails. It is clear that all HLS implementations offer speedups when compared to the latency on a 2.4 GHz CPU, even though the resulting netlists are expected to work at 100 to 400 MHz.

Table 2. Latency results for Intel Xeon (Software), SystemC and C (Hardware)-based approaches

Algorithm		Software (μs)	SystemC (μs)	C (μs)	Speedup	
					SystemC	C
MI		2105	1225	980	1.71x	2.15x
TE		892	857	510	1.04x	1.75x
LM	*Insert*	0.054	0.022	0.035	2.45x	1.54x
	Search	6.877	0.248	0.598	27.7x	11.5x
	Delete	7.169	0.282	0.661	25.42x	10.8x
MA	*Allocation*	12.363	0.371	0.974	33.3x	12.6x
	Free	13.367	0.594	1.605	22.5x	8.32x

The speedup ranges considerably and at the lowest end comes that of SystemC development for MI and TE, which is 1.71x and 1.04x respectively. At the other end of the spectrum, however, SystemC development triggers a speedup for LM and MA which can be up to 33.3x for certain operations. Hence, it

becomes obvious that the very vast majority of the HLS implementations significantly over-perform their software counterparts since they are all well above one. Based on the results of this section, two major conclusions can be drawn and are analytically discussed in Sect. 5.

4.3 Area Utilisation

Area utilisation is another important indicator of how the input method used for model description affects the efficiency of the RTL model that an HLS tool will generate. Hence, Table 3 shows how the SystemC-based and C-based modelling methods behave in terms of RTL model area utilisation. These results has been based on the same target FPGA device specified in Sect. 4. The results are split into Look-Up Table (LUT) and DSP utilisation. Subsequently, the C-based flow generates RTL models that utilise the most hardware resources in all four cases. Overall, LUT utilisation is greater for the two compute-intensive algorithms, which also employ DSPs for their mathematical operations. It should be noted that LM and MA, which are memory and compute/memory-intensive algorithms, do not use fixed-point mathematical operations and as a result do not require any DSP processing as part of their implementation.

Table 3. SystemC-based & C-based flow area utilisation

Algorithm	SystemC-based		C-based	
	LUTs	DSPs	LUTs	DSPs
MI	22098	135	33703	134
TE	25063	160	27273	196
LM	1051	0	1324	0
MA	3995	0	4952	0

5 Comparative Analysis

This section offers explanations on why certain aspects of the results came to be the way they did and, also, a concise set of advice as to what the advantages and disadvantages of the two input methods are, from a *qualitative* point-of-view.

5.1 Results-Based

First, the **LoC and latency, i.e. productivity related,** results make a very strong statement that, both in terms of code length as well as overall development time, the **C-based approach always performs better** compared to the SystemC-based input method. This, primarily has to do with the fact that transitioning from an untimed C model to an HLS tool-friendly C model is easier

due to the inherent similarity between the two models. This is in contrast to the SystemC-based input method, a C-based language with, nonetheless, significant deviations from standard C, such as template class definitions for all processing functions.

Regarding LoC, it is notable that for **compute-intensive** applications, the **SystemC** high-level description is burdened further by the lack of mathematical operations, which the designer had to develop from scratch. For instance, in the case of MI, what would originally be a 282 (55.4%) LoC description, eventually, increased to 510 due to this disadvantage. Nevertheless, this has been a tool-induced problem and it must not be assumed that this shall always be the case. At any rate, even without the LoC overhead due to the mathematical operations, the SystemC description remained significantly longer compared to the C-based description. Naturally, the original C models are always the shortest compared to the rest of the HLS-friendly models and this is due to the fact that synthesisable C requires a lower level of micro-architecture directives as well as that it does not support certain C features such as pointers. Furthermore, it has been emphatically reaffirmed that the LoC size of both HLS models is always radically shorter than that of their RTL counterparts. This is an anticipated observation, which, however, acquires significance when taking into account published documentation [11,12], which argues that HLS-generated RTL models are comparable (or even shorter) in size to those that can be achieved through a handwritten/handcoded approach.

With regard to the zero mathematical code overhead in the LoC results for LM, this is because it does not use mathematical operations since it is mainly a memory-handling scheme. The MA model also does not have a similar overhead since its computational operations are not fixed-point and can, therefore, be handled as is by the SystemC-based flow's tool. One last remark in regard to the LoC results is that both model descriptions, C-based and SystemC-based, where performed by proficient designers and the difference in the very nature of the two languages meant that no experience from describing one algorithm in C could be brought in its SystemC description or vice versa. This means that the design process for all four algorithms can be thought of as independent since no experience in, for instance, C-based coding for TE could be taken advantage of during its SystemC description to the extent of having an impact in reducing both the LoC or the development time. Subsequently, judging by the **latency** results of Table 2, it becomes apparent that, in addition to the considerable code length reduction over a direct RTL implementation, the **HLS** methodology has the potential for **speedups over a *software* approach** whilst maintaining a comparable code size to the original C code that is executed on a conventional processor. This is something that has been proven in its entirety in this work. In addition, the **compute-intensive applications offer greater speedup** when processed by **the C-based flow** whereas the **memory- and memory and compute-intensive applications are faster** in the case that **the SystemC-based flow** is used. Second, the margin that separates the two approaches differs considerably between the compute-intensive and memory-intensive cases, i.e.

the C-based flow is *slightly* faster compared to the SystemC flow for MI and TE, however, the latter is *significantly* faster compared to the former for LM and MA.

5.2 Qualitative

The information presented in this sub-section is mainly based on the extensive use of the two flows by several experienced hardware designers. First, the HLS tool used in the **SystemC-based flow** offers a characteristic/feature of significant importance. This is the ability to **pre-specify the level and extent of functionality the designer expects to be executed within a given clock cycle**. On the contrary, such feature was not available by the tool of the C-based flow, which, therefore, lost on the ability to define explicitly the design functionality that is to be executed on specific and pre-determined cycle intervals. With this specification, the operations executed at every state of the control flow should be explicitly defined either in source code or in terms of micro-architecture specifications during the scheduling phase. Moreover, the control flow was described by inserting *wait* statements that represent the clock registers separating the combinational logic. Also, each function was triggered during every clock cycle and thus certain signals had to be added in order to mimic the software's control flow such as a 4-phase handshake. Benchmarking reasons forbid the authors to disclose the tools' names, however, they are available to discuss their work further. Subsequently, in the C-based flow, the elimination of functionality specification and 4-phase handshaking leads to an automated transformation from C to RTL code without special intervention from the designer (except from some directives).

A second discernible characteristic that differentiates the two input methods is that of the **level of experience** needed for efficient model descriptions. Here, **the C-based method** is the one with the **shortest time** required to attain a satisfactory level of proficiency in efficient high-level design modelling. On the other hand, **SystemC** is more particular and, therefore, poses a **steeper learning curve** to a designer. A direct consequence of this is that high level description with a C-based flow becomes much more economical, i.e. significantly fewer lines of code, since it is closer to the original untimed-C model of an algorithm, which is almost always the case. Hence, at the early stages of HLS usage, a C-based flow can be considerably faster in completing the process of starting with a high-level model and generating a working low-level RTL equivalent, compared to a SystemC-based flow. The downside for the former, on the other hand, is that what **SystemC** lacks in modelling speed makes it up in thoroughness, making its **models more robust and comprehensive**. Once again, this is due to the level of designer expertise required for high-level modelling when using this language. Hence, as soon as a capable level of expertise has been attained, a SystemC-based flow becomes a much more attractive candidate for HLS reconfigurable systems design. Moreover, it is a risk of some probability, as was in this case, that the **SystemC-based** flow's HLS tool may not offer a library for any kind of **mathematical operation** such as *math.h* in synthesisable code. This

Table 4. SystemC-based & C-based flows

Characteristic	SystemC-based	C-based
Define functionality per cycle	Yes	Not available
Input modelling learning curve	Steep	Smooth
Effort for input model description	Hard	Easy
Level of expected expertise	High	Moderate
Input model modifications	High	Moderate
Input model abstraction level	Medium	High
Math libraries	Weak	Strong
Parallelisation & speed	Linear	Non-Linear
Input model code length	Worse	Better

can be a serious burden in the high-level description of a system and, specifically for the needs of the systems addressed here, the designers had to develop two different mathematical operations from scratch. These have been a fixed-point division and a logarithmic operation on base 2, which is performed with the same latency as in a C++ software implementation, i.e. two cycles. Notably, this has been achieved with the same precision as that met in software. Hence, in the event of selecting this type of input method, the designer must investigate the capabilities of the HLS tool that will be selected as part of the flow. In addition, the **SystemC-based** method required that the model underwent **heavy modifications and adaptations** in order to be compatible with the HLS tool's requirements; for instance, the function of every class must be implemented as a thread and each thread is then executed concurrently which brings up the need for handshaking in order to be able to emulate the function's series of operation.

Another notable observation has to do with parallelisation issues. Specifically, the **SystemC-based** flow is much **more efficient in parallelising a design** and this is a direct consequence of the more detailed description of the SystemC models. Hence, doubling the area of a design lead to half the execution time while this did not apply to the C-based input method. In general, it is always the case that SystemC needs more lines of high-level code compared to a C-based method in order to describe the same design. The significant difference between the two is mainly due to the code inserted for modularity purposes, e.g. template class definitions for all data structure processing functions, wait statements etc. Finally, Table 4 summarises the main points outlined in this sub-section.

6 Conclusions

This paper presents a concise and thorough investigation and comparison between two of the most popular HLS model description input methods, i.e. C-based and SystemC-based, for reconfigurable systems design. This is achieved by analysing the design process for four different compute- and memory-intensive

popular schemes. The paper compares the two methods whilst going through their respective flows, starting from a high-level model description all the way to a low-level RTL equivalent. Finally, this work unfolds an in-depth account on how the two input methods along with their respective design flows compare against one another. This account is based both on the technical results as well as the accumulated experience from their extensive use, thereby creating a foundation on which a designer can base their attempts on transforming popular computing algorithms to their RTL equivalents for rapid prototyping/implementations for reconfigurable systems.

References

1. Birnbaum, M.D.: Essential Electronic Design Automation (EDA). Prentice Hall Modern Semiconductor Design, Upper Saddle River (2004)
2. Whitson, C., Michelsen, M.: The negative flash. J. Fluid Phase Equilib. **35**, 51–71 (1989)
3. Coussy, P., Morawiec, A. (eds.): High-Level Synthesis: From Algorithm to Digital Circuit, vol. 1. Springer, Netherlands (2008). https://doi.org/10.1007/978-1-4020-8588-8
4. Nane, R., Sima, V., Pilato, C., Choi, J., Fort, B., Canis, A., Chen, Y., Hsiao, H., Brown, S., Ferrandi, F., Anderson, J., Bertels, K.: A survey and evaluation of FPGA high-level synthesis tools. Trans. Comput.-Aided Des. Integr. Circuits Syst. **35**(10), 1591–1604 (2016)
5. Thomas, D.E., Lagnese, E.D., Walker, R.A., Nestor, J.A., Rajan, J.V., Blackburn, R.L.: Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench. The Kluwer International Series in Engineering and Computer Science, vol. 85, 1st edn. Springer, Boston (1990). https://doi.org/10.1007/978-1-4613-1519-3
6. Cover, T., Thomas, J.: Elements of Information Theory. Wiley-Interscience, New York (1991). ISBN 0-471-06259-6
7. Schreiber, T.: Measuring information transfer. Phys. Rev. Lett. **85**(2), 461–464 (2000)
8. Knowlton, K.: A fast storage allocator. Commun. ACM **8**(10), 623–624 (1965). ISSN 0001–0782
9. Portals 4.0. http://www.cs.sandia.gov/Portals/portals4.html
10. Wikipedia: Buddy memory allocation – Wikipedia, The Free Encyclopedia. https://en.wikipedia.org/w/index.php?title=Buddy_memory_allocation&oldid=653135858. Accessed 21 July 2015
11. Winterstein, F., Bayliss, S., Constantinides, G.: High-level synthesis of dynamic data structures: a case study using Vivado HLS. In: International Conference on Field-Programmable Technology, FPT, pp. 362–365, 9–11 December 2013
12. Karras, K., Blott, M., Kees, A.: High-Level Synthesis Case Study: Implementation of a Memcached Server. CoRR, vol. abs/1408.5387 (2014)