Exploration of Synthesis Methods from Simulink Models to FPGA for Aerospace Applications

Invited Paper

Serena Curzel
Politecnico di Milano
Milano, Italy
serena.curzel@polimi.it

Michele Fiorito
Politecnico di Milano
Milano, Italy
michele.fiorito@polimi.it

Patricia Lopez Cueva
Thales Alenia Space
Cannes, France
patricia.lopezcueva@thalesaleniaspace.com

Tiago Jorge
GMV
Madrid, Spain
tiago.jorge@gmv.com

Thanassis Tsiodras
European Space Agency (ESA/ESTEC)
Noordwijk, The Netherlands
thanassis.tsiodras@esa.int

Fabrizio Ferrandi
Politecnico di Milano
Milano, Italy
fabrizio.ferrandi@polimi.it

ABSTRACT

Model-based development techniques in Matlab/Simulink simplify the design and implementation of software for aerospace applications, providing the required level of abstraction for scientists that work on complex navigation and control algorithms. As Field Programmable Gate Arrays (FPGAs) have become more and more relevant in space hardware platforms, developers could benefit from automated acceleration flows that do not require extensive manual rewriting of their code to port it on FPGA. We analyze existing methods that synthesize Simulink models, showing how a combination of automated C code generation and High-Level Synthesis can enable rapid prototyping, fast design space exploration, and a good trade-off between accelerator efficiency and design flexibility. We test the proposed acceleration flow on real-world guidance and navigation control systems for CubeSats.

CCS CONCEPTS

• **Hardware → High-level and register-transfer level synthesis**; **Reconfigurable logic and FPGAs**.

KEYWORDS

FPGA, aerospace, High-Level Synthesis

ACM Reference Format:
Serena Curzel, Michele Fiorito, Patricia Lopez Cueva, Tiago Jorge, Thanassis Tsiodras, and Fabrizio Ferrandi. 2023. Exploration of Synthesis Methods from Simulink Models to FPGA for Aerospace Applications: Invited Paper. In *20th ACM International Conference on Computing Frontiers (CF'23), May 9–11, 2023, Bologna, Italy.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3587135.3592766

1 INTRODUCTION

Space missions today face a growing need for on-board computing performance: low bandwidth communication between spacecraft and Earth requires sensor data to be pre-processed and compressed before transmission, and increasingly complex navigation algorithms cannot be operated remotely due to latency constraints. The computational requirements for such tasks are rapidly approaching the limits of what space-grade processors and microcontrollers can offer, also because radiation-hardened components are inherently slower than general-purpose processors. Instead, hybrid systems that include Field Programmable Gate Arrays (FPGAs) have drawn more and more attention [5], as they offer improved performances with acceptable overhead in size, power consumption, and cost, and they introduce the possibility of in-flight reconfiguration.

A possible obstacle to the adoption of FPGAs is the additional design effort of translating an existing algorithm, often written in a model-based programming framework such as Matlab Simulink, into low-level hardware description languages (Verilog/VHDL). In this paper, we analyze existing solutions, based on High-Level Synthesis (HLS) and on tools provided within the Matlab suite, to automatically synthesize Simulink models for aerospace applications into FPGA accelerators. We propose to combine the Matlab Embedded Coder tool with an open-source HLS engine to obtain efficient designs without any modification to the input Simulink models, so that existing algorithms can be seamlessly deployed on FPGA. The proposed approach allows to evaluate different HW/SW partitioning schemes by working solely within Simulink, without needing to write low-level code for each new solution, enabling fast prototyping and rapid exploration of architectural trade-offs.

2 BACKGROUND

FPGAs in the space domain - The benefits of FPGAs for critical space applications have been shown in several studies. For example, [5] outlines the computing requirements for small satellites and predicts that future missions will have to rely on FPGA-based hardware accelerators to cope with their increasingly complex functionalities. Another overview of FPGA-based systems for space applications can be found in [6], with particular emphasis placed on reconfigurability. In-flight reconfiguration can ensure that on-board computers for long-lasting satellite missions can be upgraded and optimized throughout their entire lifetime by uploading new bitstreams from a ground station. The other concern that can be addressed through

CF '23, May 9–11, 2023, Bologna, Italy

ACM ISBN 979-8-4007-0140-5/23/05.
https://doi.org/10.1145/3587135.3592766

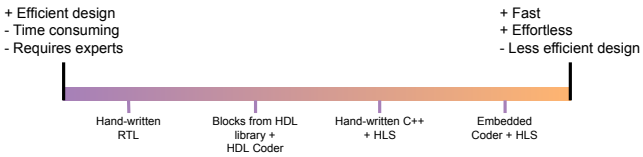

Figure 1: Synthesis tools from Simulink models to FPGA.

dynamic reconfiguration is reliability: FPGAs are notoriously susceptible to single event upsets when exposed to radiation, and errors in the configuration memory can cause incorrect behavior (especially if they accumulate in years of operation).

It is worth noting that these are not only theoretical studies: there are already several space missions that have successfully employed FPGAs as on-board computational units. To quote just a few of them, the Cibola Flight Experiment [8] performed experiments and assessed FPGA performance when exposed to radiation, and several FPGAs are used on the Mars Exploration Rovers and their landers [9]. The CHREC Space Computer [11], a design based on a multi-processor System-on-Chip platform, has been successfully deployed and validated on the International Space Station.

High-Level Synthesis - HLS tools simplify the implementation of accelerators on FPGA by automating the most complex, time-consuming step in the development flow: instead of manually writing VHDL/Verilog code, the user only needs to provide a program written in a well-known software language such as C/C++, together with constraints about timing and resource utilization that the final design has to satisfy. In this way, the increased performance offered by FPGAs is made available also to software developers that do not have hardware design expertise. Generally, the High-Level Synthesis flow begins with a compilation step to analyze data dependencies and loops in the input C/C++ program, perform typical code optimizations, and generate a Control and Data Flow Graph (CDFG). Then three core steps are performed on the CDFG (*resource allocation*, *scheduling*, *binding*) to define the structure of the output hardware by assembling functional, storage, and communication units taken from a library of RTL components. In the end, the obtained HDL code is ready to be used in a commercial FPGA design tool for further analysis, logic synthesis, and deployment. In the past, the shorter development time offered by HLS used to be at odds with the efficiency of the generated designs. Instead, several commercial and open-source tools exist today [2] that are able to generate efficient designs, competitive in speed and resource utilization with hand-optimized RTL code.

3 SYNTHESIS TOOLS AND METHODOLOGIES

Several methods exist to translate a Simulink model into an FPGA design, each having different degrees of automation, abstraction, and efficiency. We ranked them in Figure 1 according to the amount of effort required; the least automated option is manually writing RTL code that corresponds to the Simulink model functionality. Hardware accelerators designed in Verilog/VHDL are traditionally considered to be more efficient than others that have been specified at higher levels of abstraction; however, this is the most expensive method in terms of both time and effort. Moreover, domain experts who write complex Simulink models for aerospace applications likely do not have any hardware design expertise, so there is a risk of introducing inefficiency and errors during the translation. Adding FPGA experts to the team is an additional cost, and it also does not entirely eliminate the risk of translation errors when moving between such different programming models.

Matlab provides a slightly more automated method that automatically translates selected blocks from Simulink models into Verilog/VHDL: HDL Coder. Some important limitations have to be taken into consideration here, the most relevant one being that HDL Coder does not support every block available in the Matlab/Simulink libraries, so parts of the input model might need to be eliminated or substantially changed. For supported blocks, usually, there are both a generic version and one that has been optimized for RTL code generation, so the majority of blocks in an existing model might have to be manually substituted with their counterparts from the HDL library to fully exploit HDL Coder's capabilities. Writing or re-writing a Simulink model for HDL Coder requires a low-level perspective similar to that of manual RTL design, and later in Section 4.1 we have provided a concrete example of how this impacts the quality of the generated accelerators.

As HLS is an established method to raise the level of abstraction in the design of FPGA accelerators, it would be beneficial to use it when synthesizing Simulink models; to do that, the input model has to be translated into a programming language that HLS tools support (usually C or C++). The Xilinx Model Composer tool integrates Vitis HLS in the Matlab suite by providing library components (in hardware description languages or HLS-friendly C code) that can substitute Simulink blocks; to implement functions that are not available in the library it allows to manually write C/C++ code and integrate it in the model as a custom block [1]. Mentor Graphics similarly proposes to combine the use of Matlab/Simulink and Catapult HLS through a manual translation of blocks that are selected for hardware acceleration into C++ code [10]. Both of these solutions focus on the integration of HLS within the Matlab suite, so that the newly introduced blocks can be verified in the context of the whole Simulink model using familiar tools, and they correctly assume that writing C/C++ code is easier and less error-prone than writing Verilog/VHDL. However, asking aerospace engineers to manually translate parts of large and complex Simulink models would require a considerable amount of time, and domain experts might find even C++ challenging, as it represents a completely different level of abstraction.

We propose to further automate the design process by applying High-Level Synthesis to C code generated by Matlab's Embedded Coder tool. This option is not often considered because the output code of Embedded Coder is not always compatible with standard HLS tools such as Catapult or Vitis HLS. Moreover, C code generated and optimized by Embedded Coder for an embedded microprocessor might produce less efficient designs than hand-written C. Nevertheless, the combination of Embedded Coder and High-Level Synthesis would enable a fast and effortless translation of Simulink models into good quality hardware designs. We found that the open-source HLS tool Bambu [3] can successfully process C code produced by Embedded Coder, with the only additional requirement of adding a wrapper around the generated step function to map inputs and outputs of the accelerator to corresponding variables in the code.

With the proposed solution, users with no experience in hardware design or HLS can select blocks from their input Simulink

Table 1: Configuration options in Embedded Coder/Bambu.

Property	Options
Embedded Coder	
Hardware board	ARM 11
	Intel x86-64 (Windows)
	Intel x86-64 (Linux)
	Atmel AVR32
Support long long	Yes
	No
Code generation objectives	Unspecified
	Execution efficiency
	RAM efficiency
Bambu	
Experimental setup	BAMBU-AREA
	BAMBU-BALANCED
	BAMBU-PERFORMANCE
Memory allocation policy	ALL-BRAM
	NO-BRAM
Loop unrolling	Yes
	No

models and seamlessly translate them into FPGA accelerators. During the process they can test different HW/SW partitioning decisions by changing the subset of blocks to be translated, and explore different configuration options in Embedded Coder and Bambu to meet specific application requirements. An example of the available configuration options is given in Table 1. Embedded Coder provides a set of configuration options to tune the code generation process according to user-defined constraints, which are meant to adapt the output C code to a specific processor and application. Embedded Coder supports several hardware targets from different vendors, for example it can produce C code tailored to ARM, Intel, and Atmel processors; for each of them it can be specified whether support for 64 bits long long data types is available. Moreover, it is possible to specify high-level "Code generation objectives" that automatically configure Embedded Coder to prioritize fast execution time, reduced memory usage, debugging, safety, and other factors that might be critical for a specific application. Bambu exposes similar high-level options called "Experimental setups" that automatically configure other options to prioritize the generation of Verilog/VHDL code with minimized resource consumption (BAMBU-AREA), few clock cycles (BAMBU-PERFORMANCE) or to find a good compromise between the two (BAMBU-BALANCED). Other available command-line arguments provide different memory allocation and loop unrolling strategies: objects that are stored in memory can be allocated to on-chip block memories (BRAMs) or moved to external memory, loops can be unrolled to increase instruction-level parallelism if a reduction in latency is desirable and the additional area overhead is acceptable. In general, it is hard to predict in advance which configurations will result in the best quality of results after synthesis, and therefore it is crucial that the design space exploration process in the proposed approach does not require any manual modifications on the input Simulink model.

Table 2: Selected benchmarks.

Model	Simulink blocks	Lines of C code	Constants	Static variables
Filters	21	432	584 bytes	120 bytes
SAGE-GNC	445	2391	592 bytes	248 bytes
HERA	62	54 391	644 bytes	38.5 Mbytes

4 EXPERIMENTAL EVALUATION

We generated FPGA designs from three Matlab/Simulink models, briefly described in Table 2 in terms of size and complexity (neglecting the variations in the length of the C code introduced by different Embedded Coder configurations). The *Filters* model contains two filter blocks, the first one taken from the Simulink library and the second built from gain, delay, and arithmetic blocks. *SAGE-GNC* and *HERA* are two real-world aerospace applications: the full Guidance and Navigation Control system developed within the CoRA-SAGE project [4], and a real-time image-based navigation algorithm from the HERA mission [7].

4.1 Comparison of different approaches

As stated in Section 3, the Matlab suite contains a tool called HDL coder that can translate selected blocks into low-level Verilog/VHDL code. Even if the input model only contains blocks supported by HDL Coder, however, without hardware design expertise the results obtained with HDL Coder can be worse than what can be achieved with the combination of Embedded Coder and High-Level Synthesis. To show this, we built the *Filters* model so that it was possible to process it with both HDL Coder and Embedded Coder, we passed the generated C code to Bambu, and synthesized the two resulting Verilog modules with Vivado 2020.2 for a Xilinx Zinq-7000 FPGA.

One of the configuration options that can be selected in HDL Coder is the desired frequency: in this experiment the target frequency was set to 100MHz, but the post-implementation results in Table 3 show that the real operating frequency is considerably lower. We found at least two identifiable reasons that explain why HDL Coder produces a design that runs one order of magnitude slower than what we required. The first one is that part of the model performs calculations on floating-point data types, and floating-point computation tends to be inefficient on FPGA. Aerospace engineers may not be used to taking into account data representation issues when they write Simulink models, so it is realistic to expect that their applications contain blocks that can be rewritten as fixed-point computation, which would result in more efficient FPGA designs. In our scenario, however, we would like to explore different hardware/software partitions starting from a single model, so adapting data types for different subsets of blocks each time would quickly nullify the automation benefits provided by HDL Coder.

The second reason behind such a wide gap between the required frequency and the obtained one is that one of the two filters in the model contains a feedback loop, and HDL Coder does not add registers to break loops unless the user explicitly includes them as blocks from the HDL library. Overcoming this problem would again require a significant amount of manual rewriting and hardware design expertise, and it becomes unfeasible as the input model

Table 3: Implementation results for Verilog code generated through HDL Coder, or Embedded Coder and HLS (Filters model).

Tool (Required clock period)	Clock Cycles	Frequency	Slack	Registers	Slices	DSPs	BRAMs
HDL Coder (10 ns)	5	8.83 MHz	−103.250 ns	175	2062	36	0
Embedded Coder + Bambu (10 ns)	55	96.91 MHz	−0.319 ns	3412	1713	10	0
Embedded Coder + Bambu (9.5 ns)	62	107.02 MHz	0.156 ns	4451	1879	10	0

Table 4: Comparison between accelerators produced by Bambu and Vitis HLS.

Model	Configuration	HLS Tool	Clock Cycles	Frequency	Registers	Slices	DSPs	BRAMs
Filters	ARM	Vitis HLS	62	106.93 MHz	3554	1235	28	2
Filters	ARM	Bambu	55	97.70 MHz	3418	1721	10	0
Filters	ARM-LL	Vitis HLS	50	106.97 MHz	3092	1185	28	2
Filters	ARM-LL	Bambu	55	96.91 MHz	3412	1713	10	0
SAGE-GNC	ARM	Vitis HLS	4182	120.29 MHz	29504	14 682	247	6
SAGE-GNC	ARM	Bambu	3301	78.53 MHz	36483	25 285	505	100
SAGE-GNC	ARM-LL	Vitis HLS	-	-	-	-	-	-
SAGE-GNC	ARM-LL	Bambu	1735	100.03 MHz	12424	9139	549	4

becomes larger and more complex. This second issue disappears when the model is transformed into Verilog code through Embedded Coder and HLS. In fact, HLS engines schedule operations so that they fit inside a given clock period, inserting registers where they are needed rather than expecting the user to specify where they should be placed. If the resulting design is still not able to run at the desired frequency (negative slack reported after logic synthesis and implementation), specifying a stricter constraint may help guide the tool towards different scheduling and allocation decisions. Such a case is visible in Table 3, where Verilog code synthesized by Bambu 0.9.7 is not able to run at 100MHz when the initial clock period requirement is 10ns, but reaches the desired target when generated with a tighter constraint. From these considerations we can conclude that combining Embedded Coder with HLS represents a more appropriate way to translate existing Simulink models into Verilog/VHDL than using HDL Coder, especially when the input is a complex model written by aerospace engineers and fast design space exploration is a priority.

4.2 Comparison of different HLS tools

After deciding to exploit Embedded Coder to automatically translate Simulink models into C, the second important choice to build a stable design flow is to select the most appropriate HLS tool. In this section we compare the performance of Vitis HLS 2020.2 and Bambu 0.9.7, which are state-of-the-art tools supporting most of the C/C++ language. An exhaustive comparison across all experiments has not been possible because Vitis HLS was not always able to synthesize the C code produced by Embedded Coder; specifically, Vitis HLS failed to compile the C code with long long support of the *SAGE-GNC* model due to an unsupported array access (the size of the array could not be determined at compile time), and it could not finish pre-synthesis checks on the *HERA* code (the process was killed after running for more than 24 hours).

Table 4 reports post-implementation results for accelerators generated starting from C code tailored to an ARM target, with and without 64 bit data types (-LL configurations). On the *Filters* model, Bambu tends to consume fewer DSPs and BRAMs with similar or slightly worse performance; the *SAGE-GNC* model that Vitis HLS successfully synthesized resulted in performances that are comparable to what Bambu was able to achieve starting from the same C code. However, the best performance overall for *SAGE-GNC* can be achieved by changing the Embedded Coder configuration (ARM-LL instead of ARM), which is not compatible with Vitis HLS.

A significant drawback of Vitis HLS is that to introduce further optimizations (e.g., parallelism, memory interfaces), the user has to insert compiler directives (pragmas) at the right place within the input C code; Bambu, on the other hand, exposes most optimization opportunities as command-line options. In the case of existing aerospace applications, the meaning of each function and variable within the automatically generated code may be hard to trace, especially if the user is not the same person that designed the Simulink model. The effort of understanding where to insert optimization directives significantly slows down the design space exploration process, and it would be lost each time the input Simulink model is modified and the C code is regenerated.

4.3 Design space exploration

We finally tested the proposed design flow on two aerospace applications to explore the design space provided by different Embedded Coder and Bambu configurations. In all experiments, logic synthesis was performed by Vivado 2020.2 for a Xilinx Virtex-7 FPGA with a target frequency of 100 MHz.

Figure 2 reports the results of the experiments that we performed on the *SAGE-GNC* model. Each color corresponds to a different hardware target in Embedded Coder or to the same hardware target with different long long support options; groups of bars on the x axis represent different Bambu configurations. Separate charts are provided for six different hardware metrics: number of clock cycles (after simulation), clock frequency, registers, slices, DSPs, and BRAMs (after synthesis and implementation). The configuration options that we selected are the ones reported in Table 1; however, C Code generated for AVR32 is identical to the one for ARM11, so results for the AVR configurations are not shown. The figure does not contain information about the code generation objectives

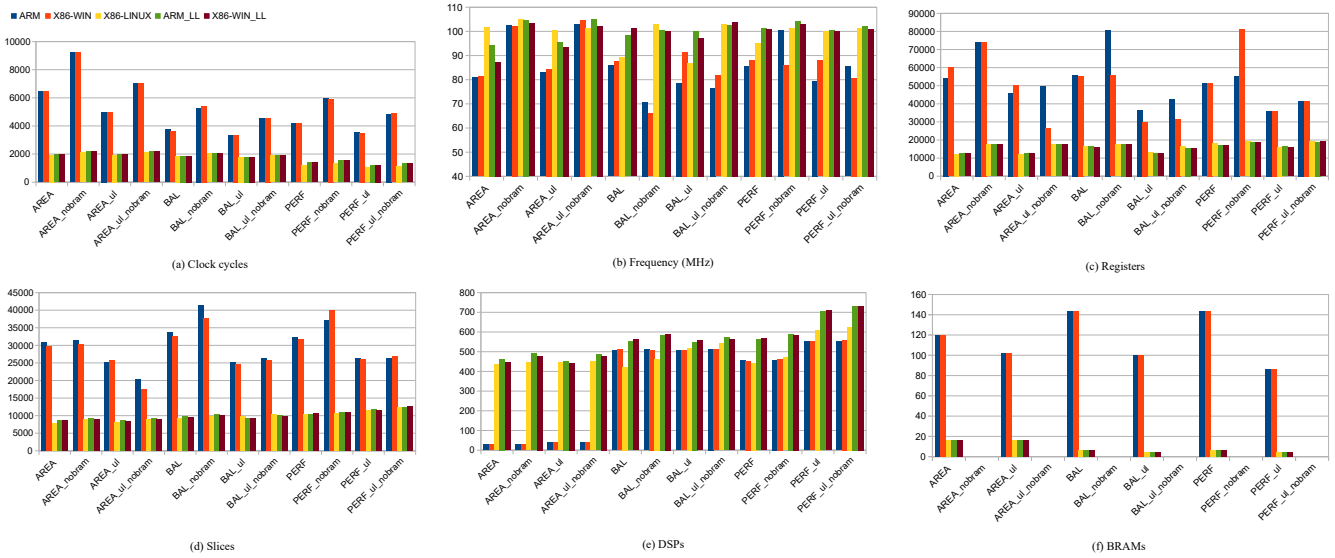

Figure 2: Timing and area utilization results across different Embedded Coder and Bambu options (SAGE-GNC model).

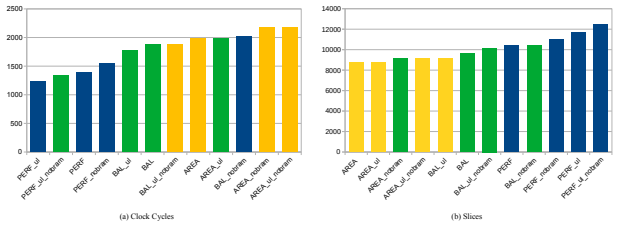

Figure 3: Effect of Bambu experimental setups.

either because they never produced any change in the output C code, as long as hardware target and long long support were the same. By default, we allocate all objects that are stored in memory to BRAMs and do not perform loop unrolling; results labeled as _nobram are obtained moving all storage to external memory, and the ones labeled as _ul contain unrolled loops.

The first consideration that can be made looking at the results is that generating C for a hardware target supporting long long data types will perform better almost in every case. There is little to no difference between an ARM or x86 target, but enabling long long support and leaving all other options unchanged produces designs that occupy on average 62% fewer registers, 62% fewer slices, 89% fewer BRAMs, and take 66% fewer clock cycles to execute. Figure 3 isolates the effect of changing Bambu configuration while maintaining the same input C code (ARM-LL): it shows the number of clock cycles and slices in ascending order, assigning different colors to the three experimental setups. It is evident that BAMBU-AREA effectively minimizes resource consumption, BAMBU-PERFORMANCE reduces the number of clock cycles, and BAMBU-BALANCED provides a good compromise between the two. When loops are unrolled, memory access patterns become less complicated: this is reflected in a lower number of clock cycles (Figures 2a, 3a) and BRAMs (Figure 2f).

The models that were translated to C without support for long long data types (blue and red bars in Figure 2) contain calculations that have been split into multiple operations on lower bit-widths: in these conditions, it is easier for Bambu to perform resource sharing, i.e., allocating different operations to the same hardware resources in a time-multiplexed way. This behavior is visible in Figure 2e, where the BAMBU-AREA experimental setup is able to share DSPs and dramatically reduce their number, while on the other hand requiring more clock cycles (Figure 2a). There are many other configuration options available in Bambu that were not considered in the first set of experiments to limit the size of the design space. For example, it is possible to introduce a "DSP allocation coefficient" to correct the estimates of the timing model within Bambu that predicts the cost of interconnections, and set a "Skip pipeline parameter" to increase the number of pipeline stages within multipliers. Taking the BAL configuration on the ARM-LL input model as the baseline, setting a DSP allocation coefficient of 1.75 and a skip pipeline parameter of 1 allows to increase clock frequency to 104.59 MHz and reduce DSPs by 48%, only increasing the number of clock cycles by 15%.

Table 5 reports results obtained on the *HERA* image-based navigation algorithm, which is considerably larger than the *SAGE-GNC* model. The accelerator has to process several MBytes of images and coefficient tables, which are too large to fit in on-chip BRAMs, so we only evaluated configurations that allocate all data to external memories. Access to the Simulink model for this algorithm is restricted, so we could only synthesize one version of the C code generated by Embedded Coder: it is the version targeting ARM processors, with support for long long data types.

Exploring the simplest configuration options in Bambu did not produce significant variations in the outputs: in fact, in the first six lines of Table 5 there is almost no difference in the number of clock cycles, and approximately 10% difference between the worst and best resource utilization results. To reduce latency, which requires

Table 5: Effect of standard and advanced Bambu optimizations (HERA model, ARM-LL).

	Clock Cycles	Frequency	Latency	Registers	Slices	DSPs	BRAMs
AREA_nobram	318 593 228	100.70 MHz	3.164 s	107 816	46 117	152	0
AREA_ul_nobram	318 593 504	101.16 MHz	3.149 s	108 380	47 217	152	0
BAL_nobram	318 592 687	101.44 MHz	3.141 s	110 035	46 004	156	0
BAL_ul_nobram	318 588 091	101.97 MHz	3.124 s	110 505	47 469	156	0
PERF_nobram	251 243 590	101.95 MHz	2.464 s	134 695	57 636	389	0
PERF_ul_nobram	251 240 824	101.31 MHz	2.480 s	136 161	59 837	389	0
*BAL_nobram	318 827 836	102.33 MHz	3.116 s	114 830	46 678	68	0
*BAL_MP_nobram	281 482 647	100.87 MHz	2.791 s	116 788	47 343	68	0
*BAL_nobram_dfp	251 489 182	102.54 MHz	2.453 s	130 838	54 587	258	0
*BAL_MP_nobram_dfp	187 929 590	103.09 MHz	1.823 s	130 264	58 409	258	0
*BAL_ul_nobram	318 823 233	101.91 MHz	3.129 s	115 262	47 559	68	0
*BAL_MP_ul_nobram	281 478 039	102.90 MHz	2.735 s	113 969	48 751	68	0
*BAL_ul_nobram_dfp	251 484 624	102.29 MHz	2.459 s	131 826	56 890	258	0
*BAL_MP_ul_nobram_dfp	187 925 011	100.92 MHz	1.862 s	131 085	59 776	258	0

either a higher frequency or a lower number of clock cycles, we applied additional optimizations (marked with an asterisk in Table 5). All the advanced configurations in the table have registered inputs, and adjustments to the DSP allocation coefficient and skip pipeline parameter. _dfp stands for disable function proxy, a Bambu option that inhibits sharing hardware modules and increases parallelism; _MP indicates that external memories can be accessed with two channels in parallel (only supported when function proxies are disabled). The best combination of options yields a 1.823s latency, which is 26% less than the best result obtained with a standard configuration. These results suggest that, as the input Simulink model increases in size and complexity, optimizations applied at the HLS level become less effective. The 26% reduction in latency that we were able to achieve with the advanced set of optimizations required up to 15 hours of processing for each new experiment (especially due to logic synthesis, place, and route). Better results could probably have been achieved with modifications to the algorithm itself and to the Embedded Coder configuration. The proposed design flow allows to exploit all these levels of abstraction to guide and speed up the refinement process: if the predicted accelerator performance fails to meet system requirements, the input Simulink model can be modified and synthesized again effortlessly instead of requiring a new and expensive manual translation.

5 CONCLUSION

Recent technology improvements have sparked a widespread interest in FPGA-based systems for aerospace applications, thanks to their performance and the possibility of in-flight reconfiguration. In this paper, we have presented a development process and a set of tools that simplify the adoption of such highly configurable systems through model-based design and HLS tools. Automatic code generation speeds up the design process and ensures the correctness of the final results, and the level of abstraction provided by HLS enables rapid prototyping and the exploration of different partitioning options. Experimental evaluation showed that producing C code with Embedded Coder for HLS is more efficient than relying on HDL coder to translate Simulink models into Verilog/VHDL, and that Bambu is always able to process the output of Embedded Coder. The configuration options for both tools offer a rich set of optimization opportunities that do not require any manual modification on the code to explore different architectural trade-offs.

Some open points remain that could lead the way for further improvement of the proposed design flow: for example, a specific process to select and evaluate possible HW/SW partitioning schemes has not been defined. At this stage, deciding which functional blocks are best suited to acceleration is entirely left to the user. In the future, analysis tools that support such decisions could be integrated into the process to provide a starting point for design space exploration and help evaluate system-level trade-offs. Another aspect that deserves more consideration is the quality of automatically generated C code: Embedded Coder offers several options to tune its output to a specific processor, but selecting the ones that will produce C code most suited to HLS is not straightforward. Moreover, the floating-point data types usually employed in Simulink algorithms do not produce efficient results on FPGA, and replacing them with fixed-point causes significant precision loss. A first solution could be to impose a constraint on the use of fixed-point data types during the development of the Simulink program; however, in order to be able to reuse existing algorithms without long and expensive redesigns, a better approach would be to optimize floating-point representations within the HLS tool.

Finally, it is worth mentioning that the modularity of our approach lends itself to research on different application domains: for example, it is reasonable to assume that aerospace systems will also rely on computer vision tasks and machine learning algorithms in the near future. The proposed design flow can be easily adapted to the development of such systems, extending the benefits of automated code generation and HLS to application domains that are not yet considered in this study.

ACKNOWLEDGMENTS

This research was partially supported by the ESA/ESTEC Contract No. 4000121154/17/NL/LF - CORA-MBAD, and by the HERMES project funded by the EU Horizon 2020 Program under grant agreement No 101004203.

REFERENCES

[1] AMD-Xilinx. 2022. Importing C/C++ Code as Custom Blocks. In *Vitis Model Composer User Guide*.

[2] J Cong, J Lau, G Liu, S Neuendorffer, P Pan, K Vissers, and Z Zhang. 2022. FPGA HLS Today: Successes, Challenges, and Opportunities. *ACM Transactions on Reconfigurable Technology and Systems* 15, 4 (2022), 1–42.

[3] F. Ferrandi, V. G. Castellana, S. Curzel, P. Fezzardi, M. Fiorito, et al. 2021. Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications. In *Proceedings of the 58th ACM/IEEE Design Automation Conference (DAC)*. 1327–1330.

[4] A Figueroa, J Cura, S Lozano, G Valle, G Rodríguez, J Corchero, et al. 2020. CoRA-SAGE: The lessons learnt from AOCS/GNC algorithms deployment in TASTE. In *Model Based Space Systems and Software Engineering Workshop (MBSE 2020)*. 1–2.

[5] Alan D. George and Christopher M. Wilson. 2018. Onboard Processing With Hybrid and Reconfigurable Computing on Small Satellites. *Proc. IEEE* 106, 3 (2018), 458–470.

[6] N Montealegre, D Merodio, A Fernández, and P Armbruster. 2015. In-flight reconfigurable FPGA-based space systems. In *2015 NASA/ESA Conference on Adaptive Hardware and Systems (AHS)*. 1–8.

[7] A Pellacani, M Graziano, M Fittock, J Gil, and I Carnelli. 2019. HERA vision based GNC and autonomy. In *Proceedings of the 8th European Conference for Aeronautics and Space Sciences (EUCASS 2019)*.

[8] H Quinn, D Roussel-Dupre, M Caffrey, P Graham, M Wirthlin, K Morgan, et al. 2015. The Cibola Flight Experiment. *ACM Trans. Reconfigurable Technol. Syst.* 8, 1, Article 3 (mar 2015).

[9] D Ratter. 2004. FPGAs on mars. *Xcell J* 50, 8 (2004), 11.

[10] P. Solanti and R. Klein. 2020. Seamless MATLAB® to Register-Transfer Level Design Methodology Using High-Level Synthesis. *International Journal of Aerospace and Mechanical Engineering* 14, 9 (2020), 406 – 413. https://publications.waset.org/vol/165

[11] C Wilson and A George. 2018. CSP hybrid space computing. *Journal of Aerospace Information Systems* 15, 4 (2018), 215–227.