jet INTL JOURNAL OF ELECTRONICS AND TELECOMMUNICATIONS, 2024, VOL. 70, NO. 4, PP. 1049–1056
Manuscript received August 6, 2024; revised October 2024. doi: 10.24425/ijet.2024.152093

# Advanced High-Level Synthesis techniques based on metamodel

Radoslaw Cieszewski, Ryszard Romaniuk, Krzysztof Poźniak, and Maciej Linczuk

***Abstract*—This paper explores advanced techniques in high-level synthesis (HLS) utilizing metamodel structures. Metamodels act as models of hardware models, generating internal hardware models based on parameter inputs and exploring the solution space to find optimal configurations. The focus is on enhancing HLS processes through metamodeling, enabling more efficient hardware design and optimization. Key contributions include a novel metamodel framework and a case study demonstrating its application in complex system designs. The proposed methods show significant improvements in synthesis efficiency and scalability, making them highly relevant for modern FPGA and ASIC design workflows.**

***Keywords*—High-Level Synthesis; Metamodel; FPGA; Optimization; Hardware Design; Digital Signal Processing; System Design**

## I. Introduction

HIGH-LEVEL synthesis (HLS) transforms algorithmic descriptions into hardware implementations, which is a critical step in the design of FPGAs and ASICs [3]. Traditional HLS approaches often face challenges related to scalability, resource utilization, and synthesis time. This paper introduces an advanced methodology leveraging metamodels to enhance HLS by providing a structured framework that addresses these challenges [2]. Metamodels serve as abstract representations that encapsulate various hardware design models, allowing the synthesis process to dynamically generate and evaluate hardware configurations based on specified parameters.

### A. Motivation and Objectives

The motivation for this work arises from the need to improve the efficiency and adaptability of HLS processes. By integrating metamodels, we aim to unify various optimization strategies, such as pipelining, parallelism, and loop unrolling, within a cohesive framework. The objectives of this study are:

- To develop a metamodel-based framework that integrates multiple HLS optimization techniques.
- To validate the framework through a detailed case study involving a complex digital signal processing system [1].
- To analyze the performance improvements in terms of synthesis time, resource utilization, and scalability.

Authors are with Warsaw University of Technology, Poland (e-mail: {radoslaw.cieszewski, ryszard.romaniuk, krzysztof.pozniak, maciej.linczuk}@pw.edu.pl).

## II. Theory

### A. Mathematical Formulation of the HLS Problem

High-Level Synthesis (HLS) transforms high-level algorithmic descriptions into hardware implementations, represented as Register-Transfer Level (RTL) descriptions. The HLS problem can be defined as an optimization challenge involving the mapping of operations from an algorithm onto available hardware resources while scheduling these operations under data dependencies to minimize specific cost metrics such as execution time, energy consumption, or logic resources.

*a) Data Dependency Graph:* Given a data dependency graph $G = (V, E)$, where $V$ is the set of operations (nodes), and $E$ is the set of edges representing data dependencies between operations, the HLS process involves the following transformations:

1. Node Assignment: Each operation $v_i \in V$ is assigned to a hardware resource $r \in R$, where the execution time of $v_i$ on resource $r$ is denoted by $\tau(v_i, r)$.

2. Graph Partitioning: The original graph $G$ is partitioned into $k$ subgraphs $G_1, G_2, \ldots, G_k$, where $G_i = (V_i, E_i)$, $V_i \subseteq V$, and $E_i \subseteq E$. Each subgraph $G_i$ is scheduled to be executed within a single clock cycle, ensuring the critical path $\text{CP}(G_i)$ of the subgraph does not exceed the clock period $T_{\text{clk}}$:

$$\text{CP}(G_i) \leq T_{\text{clk}}, \quad \forall i \in \{1, \ldots, k\}$$

*b) Transformation and Summation of Subgraphs:* The transformation of the data dependency graph involves partitioning and modification according to HLS techniques, such as loop unrolling, pipelining, and memory banking. The transformed graph is expressed as:

$$G \rightarrow \sum_{i=1}^{k} G_i$$

The total execution time $T_{\text{total}}$ of the algorithm is the sum of the processing times of all subgraphs:

$$T_{\text{total}} = \sum_{i=1}^{k} T(G_i)$$

*c) Cost Function Optimization:* The cost function $C(f, S)$ is defined as a combination of different metrics, including resource utilization, execution time, and delays. The objective is to minimize this function:

$$\text{Minimize } C(f,S) = \sum_{a \in A} C(f(a)) + \text{Latency}(S)$$

The optimization constraints include scheduling operations to satisfy data dependencies without exceeding available hardware resources.

### B. Complexity Analysis of the HLS Problem

The HLS problem is classified as NP-hard due to several factors:

1. Resource-Constrained Scheduling: Scheduling operations under limited hardware resources is NP-hard because it requires finding an optimal assignment of operations to resources while minimizing execution time and satisfying all dependencies. The exponential number of possible assignments makes exhaustive solutions impractical for large data dependency graphs.

2. Graph Partitioning: Partitioning the data dependency graph into subgraphs that can be executed in one clock cycle is complex due to the need to optimize critical paths and resources. This requires analyzing the graph topology and potential transformations, such as loop unrolling or pipelining, which alter the graph structure.

3. HLS Transformations: Implementing HLS techniques, such as pipelining, loop unrolling, and memory banking, adds complexity to the problem. Each transformation introduces new constraints and alters the graph topology, impacting the final schedule and resource assignment.

4. Latency and Resource Management: Key aspects of HLS optimization involve minimizing delays in critical paths while managing available hardware resources. Balancing these aspects optimally is highly complex and requires considering multiple trade-offs, such as execution speed versus resource usage.

### C. Graph Transformations in HLS

The process of HLS involves several graph transformations aimed at optimizing execution and resource utilization. Below are detailed transformations and their corresponding mathematical formulations:

*a) Loop Unrolling:* Given a loop represented in the data dependency graph by a subgraph $G_L$, loop unrolling involves replicating the loop body $n$ times, effectively transforming $G_L$ into $G'_L$:

$$G'_L = \bigcup_{i=1}^{n} G_{L_i}$$

where each $G_{L_i}$ is an instance of the original loop body. The critical path of the unrolled loop $\text{CP}(G'_L)$ can be reduced if parallel resources are sufficient, leading to:

$$\text{CP}(G'_L) \leq \frac{\text{CP}(G_L)}{n} + \text{overhead}$$

*b) Pipelining:* Pipelining introduces concurrency in the execution of operations by overlapping the execution stages of different iterations. For a graph $G_P$, the pipelining transformation can be represented as:

$$G_P \rightarrow \text{Pipeline}(G_P, S)$$

where $S$ denotes the pipeline stage. The effective clock period $T_{\text{clk}}$ in a pipelined execution is given by:

$$T_{\text{clk}} = \max_i(\tau(v_i, r) + \text{setup time})$$

Pipelining can significantly reduce overall execution time by maximizing the utilization of available resources through stage concurrency.

*c) Memory Banking:* Memory banking involves dividing the memory into multiple independent banks to enable concurrent access, minimizing access conflicts. For a graph $G_M$ with memory accesses, the transformation is:

$$G_M \rightarrow \sum_{i=1}^{b} G_{M_i}$$

where $b$ is the number of memory banks, and each subgraph $G_{M_i}$ represents accesses confined to a specific bank. The effective memory access time is:

$$T_{\text{mem}} = \frac{T_{\text{single\_access}}}{b}$$

assuming ideal conditions where all accesses are uniformly distributed across the banks.

### D. Advanced Heuristics for HLS Optimization

To address the NP-hard nature of HLS, heuristic methods are employed to explore the solution space efficiently. A commonly used heuristic in HLS is based on priority-driven scheduling and resource allocation. The heuristic algorithm iteratively adjusts operation priorities and allocates resources to minimize the overall cost function:

*a) Heuristic Algorithm for HLS Optimization:* Given a data dependency graph $G = (V, E)$ and resource set $R = \{r_1, r_2, \ldots, r_m\}$, the heuristic algorithm proceeds as follows:

Initialize Schedule $S = \emptyset$ each Basic Block $B_i$ in $G$ Calculate Critical Path in $G_{B_i}$ Assign Priorities to Operations based on Data Dependencies Operations remain unscheduled Select Operation with highest Priority Resources available Assign Resources from $R$ Delay Operation until Resources are available Update Schedule $S$ and Resource Allocation Schedule $S$

This heuristic optimizes the schedule by dynamically assigning resources to high-priority operations while ensuring data dependencies are respected and available resources are not exceeded in each clock cycle.

### E. Complexity Analysis of the Heuristic Algorithm

The complexity of the heuristic algorithm is driven by the number of operations $n$ and resources $m$. The algorithm iteratively schedules operations and updates resource allocations, leading to a time complexity of:

$$O(n \cdot m \cdot \log(n))$$

The spatial complexity depends on the size of the data dependency graph and control flow graph, approximated by:

$$O(n + |E|)$$

where $|E|$ denotes the number of edges representing dependencies. Practical implementation benefits from heuristic refinements that reduce the number of iterations and improve convergence speed.

## III. Metamodel Framework

The proposed metamodel framework integrates various synthesis techniques, allowing for a more holistic optimization strategy [11]. It provides a high-level abstraction that captures the relationships between different synthesis parameters and design constraints. This framework plays a critical role in the compilation and optimization of high-level algorithms into hardware implementations on FPGA platforms. The metamodel framework consists of the following components:

### A. Component Overview

- **Parameter Space Definition:** Defines the synthesis parameters relevant to the HLS process, such as loop bounds, pipeline depths, resource types, and data widths. The parameter space is mathematically represented as a multidimensional space $\mathcal{P} = \{p_1, p_2, \ldots, p_n\}$, where each $p_i$ corresponds to a specific parameter that influences the synthesis outcome.
- **Constraint Management:** Manages design constraints, including timing, area, and power requirements. Constraints are formalized using a set of inequalities $C = \{c_1, c_2, \ldots, c_m\}$, where each constraint $c_j$ limits the feasible regions of the parameter space. The constraints ensure that the synthesis adheres to practical hardware limitations.
- **Optimization Engine:** Utilizes the metamodel to explore the parameter space and identify optimal synthesis configurations. The optimization engine employs heuristic or metaheuristic algorithms, such as Genetic Algorithms (GA) or Simulated Annealing (SA), to navigate the complex landscape of the parameter space, searching for solutions that minimize or maximize a cost function $C(f, S)$. The cost function often integrates multiple objectives such as latency, resource utilization, and power consumption.

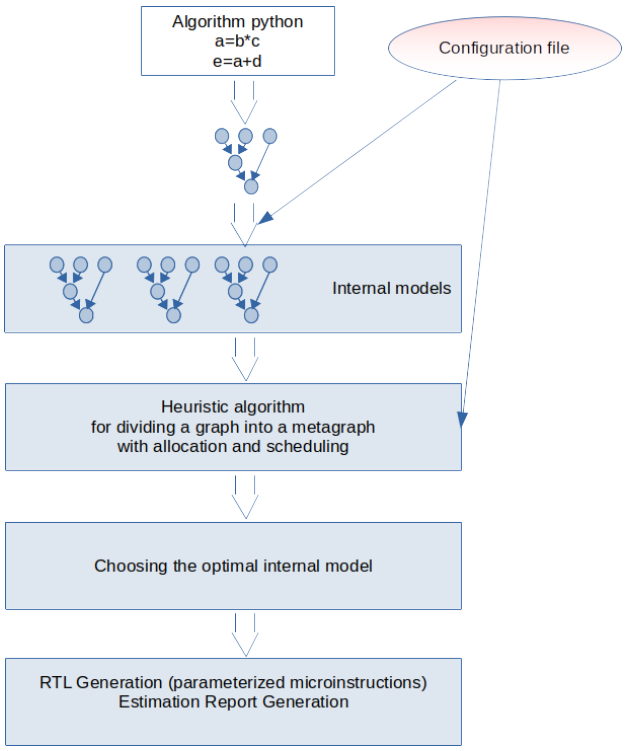


Fig. 1. Metamodel architecture for HLS integration

### B. Model of the Compilation Process

Contemporary projects in real-time data processing, such as plasma diagnostics in tokamaks, require advanced tools for designing embedded systems. A key challenge is to develop tools and methodologies that enable the efficient implementation of complex algorithms on FPGA platforms, ensuring required performance, low latency, and minimal hardware resource usage.

*a) Description of the Compilation Model:* The compilation and optimization model of an algorithm, presented in Figure 2, starts with three key input objects:

1) **Main Algorithm Code:** The algorithm code is written in a high-level language like Python, defining the main operations and data flow.
2) **Python Libraries:** A set of libraries containing predefined operations, such as matrix operations, which can be used within the algorithm. These libraries, created as part of this work, offer optimized functions for mathematical and signal processing tasks.
3) **Configuration File:** This file plays a critical role in the compilation process, allowing the user to precisely define parameters and constraints of the process. The configuration file consists of three main sections:
   - **Optimization Options:** Allows specifying optimization techniques such as loop unrolling and pipelining without modifying the original algorithm code.
   - **Constraints:** Users define boundary conditions related to latency, logical area, and power consumption. These constraints guide the optimization pro-

cess and allow the generation of different implementation models.

- **System Parameters:** This section includes information about hardware properties such as execution delays in functional units and area occupied by various FPGA blocks, enabling precise simulation and performance estimation of the system.

*b) Compilation Process:* Within the compilation block, a metamodel is used to generate various internal implementation models based on the specified constraints. The metamodel, supported by a micro-instruction library [16] and hardware data, allows analysis and selection of optimal configurations, which will be described in further sections.

The compilation process output is a description in the form of parameterized micro-instructions, which are then translated into Register-Transfer Level (RTL) code, representing the logical implementation of the algorithm in the FPGA. From this RTL code, the FPGA vendor's synthesis and compilation tools generate a bitstream that is loaded into the FPGA, enabling its physical realization.

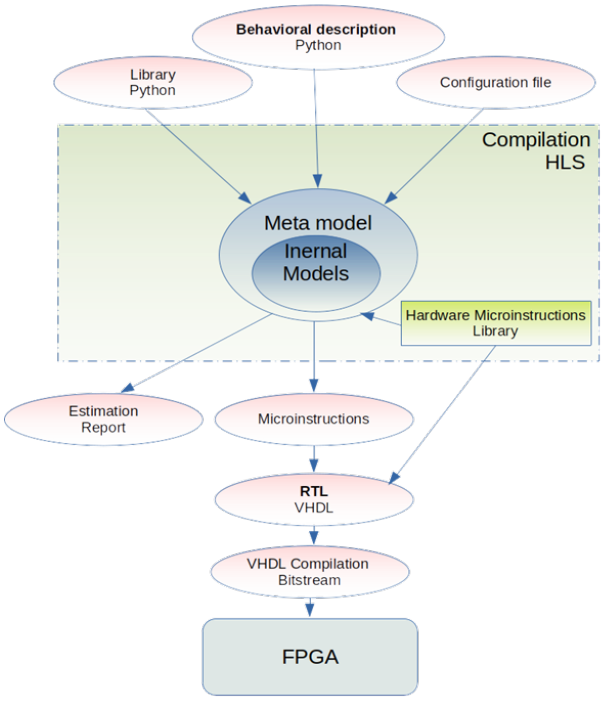


Fig. 2. Compilation and optimization process model for algorithm implementation on FPGA with bitstream generation

*c) Vendor Independence:* It is important to note that using a custom HLS tool allows for full independence from specific tools provided by FPGA vendors. By generating RTL code in standard hardware description languages, such as VHDL, this tool facilitates the portability of projects across various hardware platforms, which is particularly significant in long-term research projects, such as tokamak experiments.

### C. Metamodel as a Model of Internal Hardware Models

The metamodel is a key element of the compilation and optimization process, enabling the efficient transformation of algorithms into optimized implementations for FPGA platforms. After input data is provided—comprising the main algorithm in Python (optionally using matrix operation libraries) and the configuration file—the process begins by generating an Abstract Syntax Tree (AST), which is subsequently transformed according to specified configuration parameters.

*a) Abstract Syntax Tree (AST) Analysis and Transformation:* The AST serves as an intermediary representation of the algorithm's structure, encapsulating operations, data dependencies, and control flow.

*b) Metamodel Execution and Evaluation:* The metamodel generates different internal models by exploring combinations of the above optimization techniques. Each model is then evaluated based on a defined set of criteria, such as execution time, resource utilization, and power consumption, using a cost function $C(M)$:

$$C(M) = \alpha \cdot \text{Latency}(M) + \beta \cdot \text{Resource}(M) + \gamma \cdot \text{Power}(M)$$

where $\alpha, \beta, \gamma$ are weighting factors reflecting the importance of each metric. The optimization process aims to minimize this cost function across the space of generated models.

*c) Heuristic Algorithm and Meta-Graph Generation:* Each generated model undergoes a heuristic algorithm, which transforms the AST into a more complex graph known as a meta-graph. The meta-graph represents a structure containing all possible configurations resulting from applying different optimization parameters. The heuristic algorithm analyzes each subgraph based on data provided in the configuration file, including constraints on latency, resource usage, and other critical hardware parameters.

*d) Output and Model Selection:* The metamodel outputs the most optimal implementation model that meets the specified design criteria. Additionally, an estimation file is generated, providing detailed information on the predicted system performance, resource utilization, and latency.

*e) Detailed Metamodel Analysis:* The metamodel's operation is guided by an internal evaluation loop that continuously refines the generated models based on feedback from the heuristic evaluation phase. This loop iterates until convergence is achieved, resulting in a robust model that satisfies all design constraints while optimizing performance metrics.

## IV. Implementation of the HLS Compiler

### A. High-Level Language for Algorithm Description

Modern research and engineering projects conducted at leading scientific institutions, such as JET (Joint European Torus), ITER (International Thermonuclear Experimental Reactor), CERN (European Organization for Nuclear Research), Fermilab (Fermi National Accelerator Laboratory), SLAC (Stanford Linear Accelerator Center), and many other prominent centers, heavily rely on advanced computational techniques [2]–[15]. A critical aspect of these projects is the design and implementation of complex algorithms, which are subsequently translated into hardware architecture, often in the form of FPGA (Field-Programmable Gate Array) implementations. In this context, Python has emerged as the de facto standard

for describing and implementing algorithms at a high level of abstraction (HLL - High-Level Language).

*a) Python as a High-Level Language in Research Environments:* Python, with its simplicity, readability, and a vast array of scientific and engineering libraries, has gained widespread popularity in research environments, including JET, ITER, CERN, Fermilab, SLAC, and numerous universities and laboratories worldwide. In these settings, Python is commonly used for modeling, simulation, and data analysis, making it particularly suitable for describing algorithms that require FPGA implementation.

One of Python's greatest strengths is its ability to express complex algorithms in an intuitive and understandable manner, enabling scientists and engineers to rapidly prototype and test solutions. Python also facilitates seamless integration with other computational tools and programming environments, further accelerating the software development process. With the wealth of available libraries such as NumPy, SciPy, TensorFlow, and PyTorch, Python allows for the execution of a broad range of tasks, from numerical computations and signal processing to advanced machine learning algorithms, which can then be transformed into hardware implementations.

*b) Direct Translation of Algorithms to FPGA Implementations:* In environments such as JET, ITER, CERN, and Fermilab, where algorithms often need to be implemented in hardware, Python offers a unique capability to directly translate code into hardware implementations, particularly on FPGA devices. Tools like PYNQ (Python Productivity for Zynq) and High-Level Synthesis (HLS) enable the conversion of Python-written algorithms into HDL (Hardware Description Language) code, which is subsequently synthesized into a form suitable for FPGAs.

High-Level Synthesis (HLS) is a process that automates the conversion of high-level code written in languages like C, C++, or Python into hardware descriptions in languages such as VHDL or Verilog. HLS allows for transitioning from an algorithmic description to a hardware implementation, significantly shortening the design cycle for digital systems and facilitating easier modifications and optimizations.

By leveraging Python's extensive ecosystem of tools supporting HLS, complex computations can be effectively moved from software to hardware levels. This approach is particularly valuable in applications that demand high performance and reliability, such as in the control systems of physical experiments, where FPGAs provide unmatched computational performance and low latency.

*c) Advantages of Python in Algorithm Transformation:* One of the most advanced tools offered by Python is the built-in 'ast' (Abstract Syntax Tree) module. An Abstract Syntax Tree (AST) is a data structure that represents the abstract syntactic structure of source code, allowing for the analysis and transformation of code in a way that is independent of its original form.

The AST, or Abstract Syntax Tree, is a hierarchical structure that represents the code in the form of a tree. Each node in the tree corresponds to a syntactic element, such as a mathematical operation, loop, condition, or function call. The AST enables easier understanding of the program's structure, allowing for analysis, transformation, and optimization of the code before further processing.

The AST is particularly useful in code transformation processes because operations on the syntax tree are more natural and safer than direct manipulation of the source code text. This facilitates the introduction of advanced optimizations and transformations that might be difficult to achieve using standard methods.

*d) Implementing HLS Techniques Using AST:* Operations on the AST open up extensive possibilities for implementing advanced High-Level Synthesis methods. By manipulating the AST, it is possible to implement key HLS techniques such as:

- **Pipelining:** The AST enables automatic identification and transformation of code sections that can be processed in a pipelined fashion. Pipelining is crucial in FPGA architectures as it allows for a significant increase in performance by concurrently processing multiple computation stages.
- **Loop Unrolling:** The AST allows for loop unrolling, which involves replacing iterative constructs with their unrolled versions. This increases the parallelism of operations and reduces the number of clock cycles required to execute loops.
- **Partial Loop Unrolling:** In cases where full loop unrolling is not feasible or efficient, the AST allows for partial loop unrolling, balancing between parallelism and hardware resource usage.
- **Memory Banking:** Through AST analysis, effective memory banking can be achieved by splitting memory access into multiple independent banks to increase throughput and minimize access conflicts.
- **Data Flow Optimization:** AST operations facilitate the optimization of data flow between different algorithm elements, which is critical for minimizing delays and maximizing computational efficiency.
- **Memory Operations:** The AST enables advanced control over memory access, including operations such as reading, writing, and more complex techniques like memory banking and parallel access optimization, tailored to the specific requirements of the project. By manipulating the AST, it is possible to optimize memory access to maximize bandwidth and minimize conflicts during concurrent operations.
- **Latency Management:** The AST allows for optimization of data paths to minimize critical delays that could impact the performance of the entire system. This optimization is crucial in systems where response time is critical, such as real-time control systems.
- **Computation Parallelization:** AST operations enable the identification of code segments that can be executed in parallel, which is essential in FPGA optimization projects where parallelization is a key factor in performance enhancement.

*e) Early Error Detection Through Emulation:* Another significant advantage of Python in the context of designing algorithms for FPGAs is the ability to emulate and test transformations at the AST level before synthesis. Emulation allows for the verification of the correctness of the transformed algorithm, minimizing the risk of errors at later design stages.

Emulating code generated based on the AST enables quick and effective testing of various algorithm variants, which is extremely useful in the context of design space exploration. This allows designers to experiment with different optimizations and transformations, helping to find the most efficient solution for a given FPGA architecture.

*f) Compiler Implementation in Python:* The HLS compiler implemented in Python leverages the AST for parsing and transforming algorithmic code. The implementation is designed with modularity and extensibility in mind, allowing for the integration of various optimization techniques. Key components of the compiler include:

1) **Parser:** Converts high-level Python code into an AST, serving as the foundation for further analysis and transformation.
2) **Transformer:** Applies a series of transformations to the AST based on specified optimization parameters, such as loop unrolling, pipelining, and memory management strategies.
3) **Optimizer:** Implements heuristic algorithms to explore the optimization space, adjusting the transformations to meet design constraints like timing, area, and power.
4) **Code Generator:** Translates the optimized AST into HDL code (VHDL microinstructions [16]), which can be synthesized into an FPGA bitstream using vendor-specific tools.
5) **Verification and Emulation Module:** Provides facilities for simulating the transformed code to validate functionality and performance metrics before hardware synthesis.

## V. Implementation and Verification of FIR Filter

Digital filters are essential components in signal processing, enabling noise reduction, signal smoothing, and various other operations. Among the different types of digital filters, finite impulse response (FIR) filters are popular due to their stability and ease of implementation. This section presents the implementation details of an FIR filter using a high-level synthesis (HLS) compiler based on a metamodel framework, which allows for efficient exploration and optimization of design parameters. The implementation includes the mathematical definition of the FIR filter and an analysis of resource utilization, demonstrating how the metamodel-driven approach optimizes the filter's performance and resource efficiency on FPGA platforms.

### A. Mathematical Definition of FIR Filter

An FIR filter of order $n$ is a digital filter whose impulse response $h[n]$ is limited to a finite number of samples. For an input signal $x[n]$ and filter coefficients $h[k]$, the filter output $y[n]$ can be expressed using the following difference equation:

$$y[n] = \sum_{k=0}^{n} h[k] \cdot x[n-k]$$

Where:

- $x[n]$ is the input signal,
- $h[k]$ are the filter coefficients (impulse response),
- $y[n]$ is the output signal,
- $n$ is the filter order.

### B. Resource Utilization and Performance

The implementation of the FIR filter, which had 5 coefficients corresponding to a 4th-order filter, was synthesized using a metamodel-based approach on an FPGA platform. Table I shows the resource utilization and performance metrics of the synthesized filter.

TABLE I
Resource Utilization and Performance of FIR Filter on FPGA

| Parameter | Value |
|---|---|
| Core Frequency [MHz] | 89.73 |
| Algorithm Clock Cycles | 6 |
| Execution Time [µs] | 0.0669 |
| Adaptive Logic Modules (ALMs) | 55 |
| Logic Array Blocks (LABs) | 10 |
| ALUTs | 74 |
| Registers | 88 |
| DSP Blocks | 8 |
| M10K Memory Blocks | 0 |

The metamodel approach allows the filter to achieve optimized performance and resource efficiency, showing improvements over traditional synthesis techniques by enabling the exploration of different configuration parameters to find the optimal hardware model.

## VI. Future Directions and Enhancements

The flexibility and extensibility of the metamodel framework offer several avenues for future research and development:

- **Incorporation of Additional Heuristic Methods:** Future work will explore the integration of various heuristic optimization methods, including evolutionary algorithms. These algorithms can provide a robust mechanism for exploring large and complex design spaces, potentially leading to more efficient and innovative hardware configurations.
- **Parameter-Dependent Metamodel Testing:** A key area of future research will involve extensive testing of the metamodel under various parameter configurations. This includes studying the effects of different optimization settings, hardware constraints, and algorithmic variations on the performance and resource utilization of synthesized designs, enabling a more refined and adaptable metamodel framework.
- **Enhanced Verification and Validation Processes:** To improve the reliability and robustness of the synthesized hardware, future enhancements will focus on incorporating advanced verification and validation techniques. This includes formal verification methods and the use of hardware-in-the-loop (HIL) testing to ensure that the generated designs meet stringent performance and correctness criteria under a wide range of operational scenarios.

These advancements will further enhance the metamodel's capability to provide efficient, scalable, and adaptable solutions for modern embedded system design challenges.

## VII. Conclusion

This paper demonstrates the successful implementation of a metamodel-based high-level synthesis (HLS) framework, focusing on the design and optimization of digital signal processing components such as the FIR filter. The metamodel approach allows for a structured exploration of the design space, enabling the automatic generation of optimized hardware configurations tailored to specific requirements [2]. The implementation of an FIR filter using this framework highlights the benefits of this approach, including improved resource utilization, reduced execution time, and enhanced adaptability to varying performance constraints [3].

The FIR filter implementation showcases the practical benefits of the metamodel framework in real-world applications. By leveraging a high-level language like Python and its capabilities, such as the Abstract Syntax Tree (AST), complex algorithms can be described in a clear and concise manner, facilitating rapid prototyping and seamless integration with FPGA synthesis tools [9]. The results obtained from the FIR filter implementation confirm that the metamodel approach not only meets performance targets but also optimizes the use of FPGA resources, such as adaptive logic modules (ALMs), DSP blocks, and registers, thereby demonstrating the framework's potential for wider applications in embedded systems [4].

Resource utilization metrics from the FIR filter, including a core frequency of 89.73 MHz and minimal execution time of 0.0669 microseconds, reflect the efficacy of the metamodel approach in achieving high performance with efficient use of hardware resources. The ability to configure parameters such as loop unrolling and pipelining directly from the metamodel provides a significant advantage over traditional synthesis methods, allowing designers to achieve specific performance goals without extensive manual optimization [5].

Looking forward, the integration of machine learning techniques into the metamodel framework presents an exciting opportunity to further enhance design automation. Machine learning models can predict optimal configurations based on historical data, thus accelerating the design space exploration and reducing the time needed to achieve optimal solutions. Additionally, the potential for dynamic reconfiguration introduces a level of flexibility that can adapt to changing operational conditions in real-time, making the framework suitable for applications with fluctuating requirements.

Expanding the metamodel framework to support more complex systems, including multi-FPGA configurations, offers another promising direction. This would enable the framework to handle larger and more intricate designs, where synchronization across multiple hardware components is crucial. The scalability of the framework is key to its applicability in advanced embedded systems and could position it as a pivotal tool in the design of next-generation digital and analog mixed-signal systems.

The broader implications of this work highlight the advantages of using Python as the primary language for high-level synthesis in cutting-edge research environments such as JET, ITER, CERN, and many others. Python's readability, extensive library ecosystem, and ability to interface with synthesis tools make it an ideal choice for describing algorithms that need to be implemented on FPGA platforms [10]. The use of Python allows for the direct translation of high-level code into hardware implementations, thereby streamlining the development cycle and enabling early-stage optimization and verification through emulation.

Moreover, the implementation of advanced algorithm transformation techniques using AST manipulation positions Python as a powerful tool in the domain of digital system design. The early detection of errors and the ability to test and optimize algorithms before hardware synthesis significantly enhance the reliability and performance of the final designs. This capability is particularly valuable in research settings where rapid prototyping and iterative development are crucial.

In conclusion, the metamodel-based HLS framework presented in this paper provides a comprehensive solution for the efficient design of FPGA-based systems. By combining the strengths of high-level algorithm description in Python with the advanced capabilities of metamodel-driven optimization, the framework achieves a high degree of flexibility, scalability, and performance. Future work will continue to refine this approach, exploring new avenues for integration with emerging technologies and expanding its applicability to a wider range of complex digital systems. The continuous evolution of the metamodel framework promises to drive further advancements in the field of high-level synthesis, offering a robust platform for the next generation of embedded system design.

## References

[1] A. E. Shumack, A. Byszuk, R. Cieszewski, G. H. Kasprowicz, K. Poźniak, A. Wojeński, and W. Zabołotny, "X-ray Crystal Spectrometer Upgrade for ITER-like Wall Experiments at JET," *Review of Scientific Instruments*, vol. 85, no. 11, pp. 11E425-1–11E425-3, 2014. https://doi.org/10.1063/1.4891182

[2] R. Cieszewski, K. Poźniak, and R. Romaniuk, "Synteza wysokiego poziomu dla układów FPGA z wykorzystaniem metody partycjonowania grafów," *Przeglad Telekomunikacyjny - Wiadomości Telekomunikacyjne*, vol. LXXXVII, no. 4, pp. 80–85, 2018. https://doi.org/10.15199/59.2018.4.1

[3] R. Cieszewski and K. Poźniak, "Synteza Wysokiego Poziomu z wykorzystaniem jezyka Python," *Elektronika - konstrukcje, technologie, zastosowania*, vol. 58, no. 8, pp. 31–35, 2017. https://doi.org/10.15199/13.2017.8.7

[4] R. Cieszewski, R. Romaniuk, and K. Poźniak, "Multi-level compiler concept for high-level synthesis," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2022*, eds. R. Romaniuk, A. Smolarz, and W. Wójcik, vol. 12476, pp. 1–7, 2022. https://doi.org/10.1117/12.2659459

[5] R. Cieszewski, K. Poźniak, and M. G. Linczuk, "Widely parameterizable high-level synthesis," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2018*, eds. R. Romaniuk and M. G. Linczuk, vol. 10808, pp. 108084D-1–108084D-7, 2018. https://doi.org/10.1117/12.2502153

[6] A. Byszuk, K. Poźniak, W. Zabołotny, G. H. Kasprowicz, A. Wojeński, R. Cieszewski, B. Juszczyk, P. Kolasiński, and P. Zienkiewicz, "Fast Data Transmission in Dynamic Data Acquisition System for Plasma Diagnostics," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, ed. R. Romaniuk, vol. 9290, pp. 92902O-1–92902O-6, 2014. https://doi.org/10.1117/12.2076089

[7] R. Cieszewski, "Accelerating Artificial Intelligence with Reconfigurable Computing," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2012*, ed. R. Romaniuk, vol. 8454, pp. 84541L-1–84541L-8, 2012. https://doi.org/10.1117/12.2000098

[8] R. Cieszewski, R. Romaniuk, K. Poźniak, and M. G. Linczuk, "Algorithmic synthesis using Python compiler," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, ed. R. Romaniuk, vol. 9662, pp. 96623J-1–96623J-8, 2015. https://doi.org/10.1117/12.2205609

[9] R. Cieszewski, K. Poźniak, and R. Romaniuk, "Python Cased High-Level Synthesis Compiler," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, ed. R. Romaniuk, vol. 9290, pp. 92903A-1–92903A-8, 2014. https://doi.org/10.1117/12.2075988

[10] R. Cieszewski, M. G. Linczuk, K. Poźniak, and R. Romaniuk, "Review of Parallel Computing Methods and Tools for FPGA Technology," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2013*, ed. R. Romaniuk, vol. 8903, pp. 890321-1–890321-13, 2013. https://doi.org/10.1117/12.2035385

[11] R. Cieszewski and M. G. Linczuk, "RPython high-level synthesis," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2016*, ed. R. Romaniuk, vol. 10031, pp. 100314O-1–100314O-6, 2016. https://doi.org/10.1117/12.2249143

[12] R. Cieszewski and M. G. Linczuk, "Universal DSP module interface," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2010*, eds. R. Romaniuk and K. Kulpa, vol. 7745, pp. 7745-1T1–7745-1T1-6, 2010. https://doi.org/10.1117/12.869580

[13] T. Janicki, R. Cieszewski, G. H. Kasprowicz, and K. Poźniak, "FPGA Mezzanine Card DSP Module," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2011*, ed. R. Romaniuk, vol. 8008, pp. 80080K-1–80080K-7, 2011. https://doi.org/10.1117/12.905660

[14] M. G. Linczuk, S. Korolczuk, and R. Cieszewski, "Using Singular Value Decomposition for Neutron-Gamma Discrimination," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments*, ed. R. Romaniuk, vol. 9662, pp. 96622G-1–96622G-9, 2015. https://doi.org/10.1117/12.2204901

[15] K. Poźniak, A. Byszuk, R. Cieszewski, G. H. Kasprowicz, and W. Zabołotny, "FPGA Based Charge Fast Histogramming for GEM Detector," in *Proc. of SPIE: Photonics Applications in Astronomy, Communications, Industry, and High-Energy Physics Experiments 2013*, ed. R. Romaniuk, vol. 8903, pp. 89032F-1–89032F-6, 2013. https://doi.org/10.1117/12.2037047

[16] K. Poźniak, "VHDL-based universal programmable process for FPGA," in *Proceedings Volume 12476, Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2022*, vol. 12476, 124760X, 2022. https://doi.org/10.1117/12.2659617 Event: Photonics Applications in Astronomy, Communications, Industry, and High Energy Physics Experiments 2022, 2022, Lublin, Poland.