POLITECNICO DI TORINO

Master degree course in Electronic Engineering

Master Degree Thesis

ASIC Design-Space Exploration using High-Level Synthesis and Multi-Objective Bayesian Optimization

Advisors:
prof. Mario Roberto Casu
dott. Luca Urbinati

Candidate:
Francesco Ilacqua

Academic Year 2024-2025

To all who have supported me,
with special gratitude to those whose quiet kindness
and unassuming help made a meaningful difference
along the way.

Summary

Artificial intelligence and machine learning are transforming industries by enabling systems to learn from data, make predictions, and automate complex tasks. Their importance lies in their ability to enhance efficiency, improve decision-making, and drive innovation across various sectors, from healthcare to finance and beyond. The demand for hardware and software to perform this algorithm is rapidly increasing, driving the need for more efficient and optimized solutions. Depending on the complexity and the specific requirements for each particular application, there are many possible hardware solutions to execute this kind of algorithms, i.e. CPUs or GPUs, FPGAs and ASICs.

Nowadays this kind of operation is typically executed with GPUs, which have several downsides, like high power consumption and limited efficiency for certain specialized tasks. While GPUs excel at parallel processing, they are not always optimized for the specific computational patterns of advanced machine learning algorithms, leading to suboptimal performance and energy inefficiencies. Additionally, the general-purpose architecture of GPUs can be less efficient compared to customized hardware like ASICs, which are designed specifically for the requirements of machine learning workloads. This leads to a more optimized HW solution, that could have smaller size, less power consumption, and high computational power.

The goal of this thesis is to automate a tool-chain that produces an ASIC integrated circuit(**IC**), starting from an algorithm written in C-like language, whose behavior is different based on several variables, that influence the behavior of the algorithm and the cad tool behavior. The set of values is generated by a tool called Spearmint, which performs a multi-object optimization, to find Pareto solutions.

With the setup, a tool has been developed that is capable of performing

a complete integrated circuit design, starting from C-style code, and generating performance reports. This tool has been further extended into an optimization framework, which together enables automated design-space exploration.

With all the experimental part it is possible to design more efficient in specific properties accellerators. With the experimental results it is also possible to define set of knobs, which are more influent and set of parameters which defines that eccells in specific fields.

There are several directions in which the project could be further improved. One possible extension would be the development of more sophisticated simulation scenarios, making use of richer and more realistic input data in order to better approximate real-world conditions. Another important aspect concerns the refinement of the tool itself: improving its usability through a more intuitive interface, as well as strengthening its error detection and management mechanisms, would make the framework more robust and accessible to a wider range of users.

Acknowledgements

I would like to express my heartfelt gratitude to my family, friends, and professors for their unwavering support and guidance throughout the completion of my Master's thesis.

To my family, your belief in me, encouragement, and constant support have been invaluable, and I am deeply grateful for your presence throughout this journey.

To my professors and advisors, your expertise, insights, and constructive feedback have been instrumental in shaping both the direction and the quality of this thesis.

I am also grateful to all those who offered words of encouragement or small acts of kindness along the way; your support, often quiet yet meaningful, has not gone unnoticed.

I particularly enjoyed the opportunity to develop a solution with considerable freedom—starting from a well-defined problem and iteratively refining an algorithm to solve a seemingly simple task in a more sophisticated way, which manually would have required considerable time.

Completing this thesis has been a challenging yet immensely rewarding journey, and I am sincerely thankful to everyone who contributed to this achievement. Thank you for being part of this milestone in my academic career.

"Destiny is not a matter of chance;
it is a matter of choice.
It is not something to be waited for;
it is something to be achieved."
— William Jennings Bryan

Contents

Chapter 1

INTRODUCTION

The exponential growth of Artificial Intelligence (AI) has brought with it a corresponding rise in the computational requirements of modern algorithms. As models grow in depth, complexity, and deployment frequency particularly in edge devices and real-time environments, the limitations of general-purpose hardware such as CPUs and GPUs become increasingly apparent. These platforms, though instrumental in the early stages of AI development, struggle to meet modern demands for low latency, high throughput, and energy-efficient execution.

To address these challenges, the industry is turning toward **specialized hardware accelerators** that are purpose-built to execute AI workloads more efficiently. Among these, **Application-Specific Integrated Circuits (ASICs)** stand out for their ability to deliver exceptional performance tailored to specific computational patterns. By optimizing logic, memory hierarchy, and data flow at the silicon level, ASICs can outperform general-purpose devices in terms of speed, power consumption, and physical area—key metrics in large-scale or resource-constrained applications.

Alongside ASICs, **Field-Programmable Gate Arrays (FPGA)** represent another valuable class of accelerators. These devices provide a flexible, reconfigurable platform that can be adapted to execute a wide range of AI models. FPGA and ASICs both offer significant benefits over traditional architectures, but they serve different design and deployment needs. FPGA prioritize versatility and rapid prototyping, whereas ASICs focus on maximum efficiency and scalability.

Furthermore, with the increasing use of **TinyML**—a subfield of machine learning focused on deploying models on ultra-low-power and resource-constrained devices such as microcontrollers and edge sensors—ASICs are

becoming the default solution for on-device intelligence. TinyML enables real-time AI inference directly at the edge, eliminating the need for constant cloud connectivity and drastically reducing latency and energy consumption. In such scenarios, the extreme efficiency and compact footprint of ASICs make them ideal candidates for always-on, battery-operated applications.

Given the long design and fabrication times of ASIC chips, there is a pressing need to improve productivity. One promising solution is the adoption of frameworks such as **Spearmint** [4], which leverages Bayesian optimization for hardware design. These frameworks represent a significant step forward in bridging the gap between algorithm development and physical deployment.

In particular, Bayesian optimization enables the efficient identification of high-performing solutions without the need for exhaustive grid search, thereby significantly reducing exploration time and computational overhead.

ASICs and FPGAs: Complementary Trade-Offs

Alongside ASICs, FPGAs represent another class of devices, offering a flexible and reconfigurable architecture that supports rapid prototyping and hardware-level customization which is ideal for AI accelerators. FPGAs are especially well-suited for applications where adaptability and moderate production volumes are prioritized, such as embedded systems or industrial automation.

While FPGAs excel in terms of development agility, ASICs become increasingly advantageous as deployment scales up. By development agility, we refer to the ability of FPGAs to support rapid design iterations, testing, and reconfiguration throughout the development lifecycle. Unlike ASICs, which require lengthy and costly fabrication processes for each design revision, FPGAs provide a flexible and programmable hardware platform that can be updated or modified in a matter of hours or days. This capability enables engineers to quickly prototype, validate, and optimize their designs without the need for new silicon fabrication, significantly shortening the time-to-market for new AI models and algorithms.

Despite their non-reconfigurable nature and significantly longer fabrication cycles—often spanning several months—ASICs offer considerably lower per-unit costs at high volumes. Moreover, ASICs can be finely optimized at the gate level, allowing designers to eliminate redundancies, streamline data paths, and tailor memory hierarchies specifically for the target workload. This level of hardware specialization results in superior performance, reduced power consumption, and a smaller physical footprint compared to FPGAs.

Application Domains for Optimized ASICs

Custom AI ASICs are particularly compelling in application domains with strict constraints on power, latency, or throughput. Key areas include:

- **Latency-sensitive systems:** autonomous vehicles, robotics, and drones
- **Power-constrained environments:** mobile devices, wearables, and embedded IoT systems
- **High-throughput workloads:** large-scale data centers performing billions of inferences per second
- **Real-time edge AI:** augmented reality, video analytics, and smart sensing

In these scenarios, ASICs often outperform both GPUs and FPGAs, as they typically ensure lower energy consumption per inference while occupying only a minimal silicon area. At the same time, they provide high performance-per-watt and benefit from dedicated datapaths.

TinyML and the Rise of Edge Intelligence

The growing field of *TinyML*—machine learning on ultra-low-power, resource-constrained devices—further highlights the importance of ASIC-based solutions. By enabling inference directly on microcontrollers and edge sensors, TinyML eliminates the dependency on cloud connectivity, thereby reducing latency and power draw. Given their compact footprint and energy-efficient operation, ASICs are becoming the de facto hardware solution for always-on, battery-powered intelligence at the edge.

Design Optimization Tools

Designing ASICs is a complex task that involves balancing trade-offs among area, power, performance. Given the vastness of the design space, exhaustive evaluation of all possible solutions is infeasible. To address this challenge, Bayesian optimization techniques have emerged as a powerful approach, enabling efficient convergence toward optimal solutions without the need for exhaustive simulations. Several optimization frameworks have been developed such as **Spearmint** [4], useful when trying to minimmize the cost

function of a black-box function. In this application is not a black-box function, but a sperimental function expensive to evaluate. These frameworks employ Bayesian optimization (**BO**) and multi-object-bayesian-optimization (**MOBO**). in the single or multi-objective version to systematically and efficiently navigate the design space, identifying high-performing solutions with minimal manual intervention. Such methodologies represent a significant advancement in bridging the gap between algorithm development and physical silicon deployment, particularly enabling rapid prototyping and energy-efficient hardware design. BO engine leverages the Gaussian Process (GP) regression to build surrogate models of the objective functions, which are expensive to evaluate directly due to the time-consuming nature. By learning from prior evaluations, the BO engine predicts the performance of unseen design candidates and selects the next configurations to evaluate using an acquisition function, such as Expected Improvement (EI). This strategy allows the engine to efficiently balance exploration of new regions in the design space and exploitation of known promising areas.

1.1 Related Works

The following works are presented as examples of how Bayesian Optimization (BO) has been employed to explore large design spaces in the context of hardware accelerators. While many of these studies also focus on joint optimization of both neural network architectures and their corresponding hardware implementations, this thesis primarily addresses the application of BO as a tool for efficient design space exploration on a fixed structure. The aspect of joint software-hardware co-design is not directly tackled here, but it represents a promising direction for future research.

Multi-objective Framework for Training and Hardware Co-optimization in FPGAs

A related study by **Mansoori** [2] presents a framework for the **simultaneous optimization of CNN hyperparameters and FPGA hardware configurations** using a **multi-objective Bayesian optimization** strategy. Unlike earlier methods that optimized network or hardware separately, or collapsed multiple objectives into a single one, this work maintains separate goals—**prediction accuracy**, **hardware latency**, and **throughput**—while satisfying constraints on FPGA resources. The framework is

built on **High-Level Synthesis (HLS)** and uses the **Spearmint** library for Bayesian optimization.

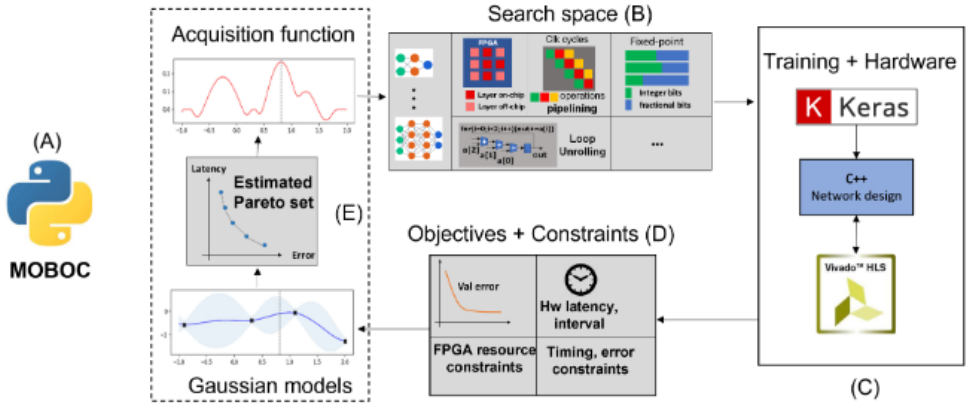

Figure 1.1: Framework proposed by Mansoori for CNN and FPGA co-optimization.

The study falls within the broader field of **Hardware-aware Neural Architecture Search (Hw-NAS)**, which focuses on optimizing neural networks for deployment on constrained hardware. Hw-NAS approaches are typically classified into:

- **Fixed-Hardware NAS**, which tunes the neural architecture for a predefined hardware target.
- **Hardware-aware NAS** (**Multi-Hardware NAS** in [2]), which jointly explores both the neural network and hardware design space.

In terms of optimization techniques, the literature includes:

- **Reinforcement Learning (RL)** and **Evolutionary Algorithms (EAs)** effective but computationally heavy.
- **Gradient-based methods**, mostly limited to fixed hardware and requiring pre-trained super-networks.
- **Bayesian Optimization (BO)**, less explored for FPGAs, yet well-suited for black-box problems with multiple conflicting objectives.

While previous BO-based work primarily focused on Fixed-Hardware NAS, the innovation of Mansoori's work is in Hardware-aware NAS, i.e., **jointly optimizing both the network and hardware configurations**, making it more applicable to real-world edge AI deployments.

The authors of [2] use a method called Multi-Objective Bayesian Optimization with Constraints (**MOBOC**), which uses Gaussian surrogate models and an acquisition function—**PESMOC (Predictive Entropy Search for Multi-objective Optimization with Constraints)**—to iteratively improve candidate solutions and explore the **Pareto front**. The search space includes both **neural parameters** (e.g., number of filters, neurons, kernel sizes) and **hardware synthesis parameters** (e.g., loop unrolling, memory types, pipelining, quantization, clock frequency).

At each iteration, performance is evaluated based on:

- **Network accuracy**, assessed post-synthesis to account for quantization.
- **Hardware metrics**, such as latency and throughput from simulation.

The objectives are jointly optimized under resource constraints (BRAM, DSP, LUT, FF) and a maximum allowable prediction error of 10%.

The experimental validation was conducted on a Xilinx Zynq-7000 SoC (XC7Z020), using a modified LeNet-5 CNN. The search space contains approximately 10^{16} combinations, making brute-force search infeasible. The authors compared three strategies: random search, separate optimization of network then hardware, and the joint MOBOC method.

After 100 optimization steps, the joint method outperformed the others, identifying configurations with better trade-offs between accuracy and performance. Specifically, it achieved up to **1.7× speedup over random search** and **1.4× over the separate optimization** approach. The integration of an exponential error term in the acquisition function accelerated convergence to low-error regions, producing a Pareto front concentrated below the 10% error threshold and satisfying all hardware constraints.

This work demonstrates the effectiveness of **co-optimization** strategies in achieving **hardware-aware neural network deployment**, especially for **resource-constrained environments** like FPGAs in embedded and edge AI.

The aim of this thesis is to implement a solution similar to the one described, but with a reconfigurable ASIC accelerator instead of an FPGA with an embedded neural network. Both approaches share a common challenge: identifying the best-performing parameters to provide as input to the configurable solution and subsequently extracting the results. To address this, Spearmint, a promising optimization software, has been adopted and integrated into the workflow.

Tree-Structured Design Space Exploration for HLS

In the work by Kuang and Wang [7], the authors address the complexity of high-level synthesis (HLS) by proposing an efficient design-space exploration (DSE) methodology tailored for multi-objective optimization. Traditional DSE methods for HLS suffer from scalability issues due to the exponentially growing number of parameter combinations, many of which result in invalid or redundant configurations.

To overcome this, the authors introduce a *tree-structured design space model*, which captures hierarchical dependencies between parameters. In this tree, child parameters are only activated if their parent nodes are set to specific values. This approach effectively prunes the search space by automatically excluding invalid configurations, thus improving the efficiency of the exploration process.

To search this reduced space, they adopt a surrogate-based method called the **Multi-Objective Tree-structured Parzen Estimator (MOTPE)**, an extension of the TPE algorithm that supports both tree-structured search spaces and multi-objective criteria. MOTPE efficiently models the nonlinear relationships between design parameters (e.g., instruction-level choices) and objectives such as performance, area, or power.

Moreover, the authors propose a *float encoding strategy* to uniformly represent discrete and categorical parameters in a continuous domain. These float-encoded parameters are later scaled and rounded to generate valid HLS instruction configurations, allowing for easier integration with the optimization algorithm and reducing implementation overhead.

Experimental results demonstrate that their method significantly outperforms baseline grid and random search techniques. Specifically, the proposed approach achieves a 66.30% improvement in Latency-Performance Design Area (LPDA) over Simulated Annealing (SA) and a 41.25% improvement over NSGA-II. Additionally, the learned Pareto front is closer to the reference Pareto front, with an average improvement in Average Distance to Reference Set (ADRS) of 94.72% compared to SA and 69.58% compared to NSGA-II. These findings underscore the potential of the proposed methodology in enhancing HLS compiler toolchains and automating energy-efficient hardware design.

This paper suggests that applying a combination of Bayesian Optimization algorithms to High-Level Synthesis is effective in identifying the best-performing solutions with fewer iterations and in less time compared to traditional methods such as Simulated Annealing (SA) and Non-dominated Sorting Genetic Algorithm II (NSGA-II). This further supports the validity of the methodology applied in this thesis.

Co-exploration of Neural Architectures and ASIC Designs

The work by Yang et al. [6] introduces a novel framework for the *co-exploration* of deep neural network (DNN) architectures and heterogeneous ASIC accelerator designs, tailored for multi-task deployment scenarios. Traditionally, neural architecture search (NAS) and hardware design optimization have been approached separately, often leading to suboptimal solutions when the model and the hardware are not jointly optimized. This work addresses that gap by enabling a unified exploration of both the algorithmic and hardware design spaces.

The proposed framework operates by jointly optimizing two coupled spaces: (i) the architectural parameters of DNNs (e.g., number of layers, filter sizes, and kernel types) and (ii) the design parameters of heterogeneous ASIC accelerators (e.g., compute unit types, memory hierarchies, and interconnect bandwidths). The goal is to identify optimal model–hardware pairs that strike the best trade-off among multiple objectives such as accuracy, latency, and energy consumption across a diverse set of AI tasks.

A key feature of the framework is the support for heterogeneous accelerators. Rather than limiting the search to a single homogeneous ASIC configuration, the framework allows for combinations of specialized compute units optimized for different neural operators. This heterogeneity enables finer-grained hardware-model matching, which is particularly beneficial in multi-task scenarios where diverse models may require different computational patterns (e.g., convolution vs. attention).

Experimental results presented in the paper demonstrate that the co-exploration approach yields significant improvements in both model accuracy and hardware efficiency compared to baseline strategies that optimize DNNs and accelerators in isolation. For instance, across various image classification tasks, the co-explored solutions achieve better accuracy–latency trade-offs than fixed-architecture baselines when deployed on custom accelerators.

Overall, the framework represents an important step toward holistic AI

system design, where algorithm and hardware are co-optimized under realistic constraints. By reducing the need for manual tuning and enabling efficient design space navigation through Bayesian techniques, it addresses the productivity bottlenecks in deploying specialized AI accelerators at scale.

Co-design and Co-search of Algorithms and Accelerators for Edge AI

The work by Zhang et al. [8] introduces a unified co-design and co-search framework aimed at jointly optimizing both neural network architectures and hardware accelerators for edge AI applications. Given the stringent constraints of edge devices—such as limited power, area, and compute capacity—traditional design pipelines that sequentially optimize models and hardware fall short in delivering optimal performance.

To address this, the authors propose a methodology where the neural network architecture and its corresponding accelerator configuration are co-optimized within a single search loop. This approach enables synergistic tuning of both the algorithmic structure (e.g., layer depth, convolutional parameters) and hardware-level features (e.g., compute array size, memory hierarchy), aligning performance, latency, and power targets simultaneously.

A central component of the framework is the use of **SkyNet**, a population-based optimization engine built upon the *Particle Swarm Optimization (PSO)* algorithm. PSO is inspired by the collective behavior of birds or fish swarms and is employed to explore a high-dimensional design space. Each particle in the swarm represents a candidate solution—comprising both network and hardware configurations—and is evaluated based on a multi-objective fitness function that includes metrics such as accuracy, latency, and energy efficiency. Specifically, SkyNet searches over architectural decisions like convolutional channel expansion, pooling locations, and layer ordering, while simultaneously tuning hardware parameters. The integration of these dimensions within a single optimization process ensures that selected designs are well-balanced and feasible for real-world edge deployment.

The proposed framework demonstrates superior performance compared to decoupled design approaches, delivering co-optimized models that meet strict resource budgets without compromising accuracy. This work highlights the effectiveness of population-based heuristics like PSO in tackling complex co-design problems in hardware-aware neural architecture search (HW-NAS).

Multi-Objective Hardware-Aware NAS with Cost Diversity

In their recent work, Sinha et al. [9] propose **MO-HDNAS** (Multi-Objective Hardware-Diverse Neural Architecture Search), a framework that performs neural architecture search (NAS) while explicitly incorporating hardware-awareness through multi-objective optimization. The framework is designed to address the growing need for deployable AI models that are not only accurate but also efficient across diverse hardware platforms.

MO-HDNAS leverages a population-based evolutionary algorithm specifically the well-established **NSGA-II** (Non-dominated Sorting Genetic Algorithm II)—to simultaneously optimize three objectives: (i) *maximizing representation similarity*, (ii) *minimizing hardware cost*, and (iii) *maximizing hardware cost diversity*. The first objective ensures that searched architectures maintain strong representational capacity, preserving the learning capabilities of deep networks. The second objective promotes energy-efficient and latency-aware model designs. The third objective, which is the novel contribution of this work, introduces a diversity-aware constraint that encourages the population of architectures to span a wide range of hardware cost characteristics. This additional objective serves to enhance the exploration of the design space, preventing premature convergence and increasing the likelihood of discovering high-performing architectures that would otherwise be overlooked in more narrowly focused search strategies.

The optimization proceeds by evolving a population of candidate architectures using genetic operations such as crossover and mutation. Hardware cost estimations are integrated directly into the fitness function, allowing the search process to remain sensitive to deployment constraints such as inference latency, energy consumption, and memory footprint.

Through extensive experimentation, MO-HDNAS is shown to outperform traditional NAS methods in discovering architectures that achieve superior trade-offs between accuracy and hardware efficiency. This makes it particularly relevant for edge and embedded AI applications where hardware constraints are paramount.

1.2 Thesis Contribution

This thesis merges two lines of research carried out in the group of Prof. Casu at Politecnico di Torino. In particular, it continues the work of **Urbinati**

and Casu [1], who developed novel hardware accelerators for mixed-precision quantized deep neural networks.

Mixed-Precision Quantization (MPQ) optimizes inference efficiency by varying the bitwidths across DNN layers, requiring hardware innovations like **Precision-Scalable (PS)** multipliers.

Their key contributions include novel multiplier designs called **Sum-Together (ST)**. These ST multipliers enable dynamic reconfiguration between high- and low-precision modes by combining multiple low-precision multiplications and summations in a single unit, allowing highly parallel Multiply-and-Accumulate (MAC) operations.

The work also presents enhanced ST-based accelerators tailored for:

- 2D convolution,
- Depth-wise convolution,
- Fully connected layers,

with added support for **Uniform Integer Quantization (UIQ)**. These accelerators support quantized inference using integer-only arithmetic and include fused operations (e.g., quantized ReLU).

A comprehensive **High-Level Synthesis (HLS)**-based **Design Space Exploration (DSE)** was conducted, varying:

- ST multiplier type,
- Parallelism and loop unrolling,
- Quantization configuration (bitwidths per layer/channel).

Evaluation on **MLPerf Tiny** benchmarks shows that SoCs embedding these ST-based accelerators achieve:

- $\sim$0.9% area overhead with 1.46$\times$ speedup (low-area target),
- $\sim$2.5% power overhead with 1.33$\times$ speedup (low-power target),
- $\sim$8.0% latency overhead with 1.29$\times$ speedup (low-latency target).

ST-based accelerators consistently provide energy savings compared to standard 16-bit fixed-precision multipliers.

While the work of Urbinati and Casu provides an effective foundation, its design space exploration methodology is low-size. The original DSE is based

on a grid search over a relatively narrow configuration range and a limited number of tunable parameters (knobs). As a result, the generated Pareto fronts (e.g., area vs. latency or power vs. latency) revealed gaps, indicating unexplored but potentially promising regions in the design space. For a faster and wider design-space exploration **MOBO** is implemented via spearmint. This approach lead to a more defined pareto fronts, without evaluating the whole desing-space.

This thesis addresses these limitations by introducing an **automated DSE framework** based on **BO**, implemented via the **Spearmint** library. This approach replaces static, brute-force exploration with an intelligent, feedback-driven optimizer that adapts based on prior synthesis results. The improved pipeline enables:

- It supports a significantly broader search space than [1], including extended parameter ranges and a larger number of tunable knobs (e.g., pipelining, loop unrolling factors, memory banking),
- It replaces exhaustive grid search with a feedback-driven optimizer that iteratively queries synthesis results (area, latency, power) to guide exploration,
- It avoids infeasible or suboptimal solutions by leveraging prior results to inform future evaluations,
- It achieves faster convergence towards Pareto-optimal configurations, requiring fewer synthesis runs despite the full design space.

This automation makes the exploration process scalable, computationally efficient, and adaptive, which is crucial for navigating complex and heterogeneous design spaces in modern hardware design.

Furthermore, the complete tool-chain, including the automated DSE scripts and synthesis infrastructure, is made publicly available in a GitHub repository to encourage reproducibility and further research contributions [12].

1.3 Bayesian Optimization theory

In this section, the effectiveness and the advantages of the BO method are discussed. The information of this chapter have been taken mainly from Frazier's paper [3].

Bayesian Optimization (BO) is a technique for optimizing expensive, black-box functions that lack closed-form expressions and gradients. Such functions

frequently occur in machine learning hyperparameter tuning, engineering simulations, and physical experiments. BO is designed for global optimization problems where each function evaluation may take minutes or hours, and thus must be used efficiently.

This tutorial introduces the core principles of BO, particularly the use of Gaussian Process (GP) regression to model the objective function, and acquisition functions to select where to evaluate next. The framework also extends to complex settings involving noise, constraints, multiple fidelities, and parallel evaluations.

Problem Formulation

The central objective is to solve:

$$\max_{x \in A} f(x),$$

where $f : \mathbb{R}^d \to \mathbb{R}$ is continuous but unknown and expensive to evaluate. x[1] is the input vector. The domain A is typically a compact, simple set like a hyperrectangle. The function f is assumed to be:

- Expensive, limiting the total number of evaluations.
- Black-box and derivative-free.
- Possibly noisy.
- Non-convex, with multiple local optima.

Basic Concepts of Gaussian Processes

A **Gaussian Process** is a collection of random variables, any finite number of which have a joint Gaussian distribution. It generalizes the notion of a multivariate Gaussian distribution to infinite-dimensional function spaces.

[1] In the context of Gaussian Process Regression, $x \in \mathbb{R}^d$ represents an input vector — for example, a configuration of parameters or hyperparameters. It is one point in the input space over which the unknown function $f(x)$ is defined. Another input x' refers to a different point in the same space, used to compare or model the relationship between $f(x)$ and $f(x')$. The similarity between such points, often measured through a kernel function $k(x, x')$, reflects assumptions about the smoothness and correlation structure of f.

Formally, a function $f(x)$ is said to follow a Gaussian Process if:

$$f(x) \sim \mathcal{GP}(\mu(x), k(x, x')),$$

where:

- $\mu(x) = \mathbb{E}[f(x)]$ is the **mean function**, usually assumed to be zero or constant.
- $k(x, x') = \text{Cov}(f(x), f(x'))$ is the **kernel** or **covariance function**, which encodes assumptions about the function's smoothness and structure.

This means that for any finite set of input points $x_1, \ldots, x_n$, the vector $[f(x_1), \ldots, f(x_n)]$ is multivariate normally distributed:

$$[f(x_1), \ldots, f(x_n)] \sim \mathcal{N}(\mu, K),$$

where $\mu = [\mu(x_1), \ldots, \mu(x_n)]$ and $K_{ij} = k(x_i, x_j)$.

Gaussian Process Regression

Given a set of noisy observations $\mathcal{D}_n = \{(x_i, y_i)\}_{i=1}^n$, where $y_i = f(x_i) + \epsilon_i$ and $\epsilon_i \sim \mathcal{N}(0, \sigma^2)$, GP regression computes the posterior distribution over f at a new input x.

The posterior predictive mean and variance are:

$$\mu_n(x) = k(x, X)[K + \sigma^2 I]^{-1} y,$$
$$\sigma_n^2(x) = k(x, x) - k(x, X)[K + \sigma^2 I]^{-1} k(X, x),$$

where:

- $k(x, X)$ is the covariance vector between x and training points.
- K is the kernel matrix between all training points.
- y is the vector of observed outputs.

The GP provides both an estimate $\mu_n(x)$ and an uncertainty measure $\sigma_n(x)$, essential for balancing exploration and exploitation.

The Role of the Kernel Function

The **kernel** defines how similar two inputs x and x' are. A common form is:

$$k(x, x') = \alpha_0 \exp\left(-\sum_{i=1}^{d} \alpha_i (x_i - x_i')^2\right),$$

where:

- $x, x' \in \mathbb{R}^d$ are input vectors.
- α_i are inverse length-scale parameters.

If x and x' are close, the kernel value is high, implying strong correlation between $f(x)$ and $f(x')$. This spatial correlation enables smooth function modeling.

Acquisition Functions: Expected Improvement

An acquisition function guides where to evaluate next. The **Expected Improvement** (EI) is:

$$\mathrm{EI}_n(x) = \mathbb{E}_n\left[[f(x) - f_n^*]^+\right],$$

where $f_n^* = \max_{i=1,\dots,n} f(x_i)$. Using the GP posterior:

$$\mathrm{EI}_n(x) = (\mu_n(x) - f_n^*)\Phi(z) + \sigma_n(x)\phi(z),$$

with $z = \frac{\mu_n(x) - f_n^*}{\sigma_n(x)}$, and Φ, ϕ the CDF and PDF of the standard normal distribution.

Alternative Acquisition Functions

The **Knowledge Gradient (KG)** selects the point that maximizes the expected increase in the mean of the best value:

$$\mathrm{KG}_n(x) = \mathbb{E}_n\left[\max_{x'} \mu_{n+1}(x') - \max_{x'} \mu_n(x')\right].$$

Entropy Search (ES) and **Predictive Entropy Search (PES)** aim to reduce uncertainty about the location of the global maximum:

$$\mathrm{PES}_n(x) = H(f(x)) - \mathbb{E}_{x^*}[H(f(x) \mid x^*)],$$

where $H(\cdot)$ denotes entropy.

Bayesian Optimization Algorithm

The BO algorithm proceeds as follows:

1. Initialize with a small number of evaluations.
2. Fit a GP to the data.
3. Maximize the acquisition function to select the next input.
4. Evaluate $f(x)$ and update the dataset.
5. Repeat until evaluation budget is exhausted.

Extensions and Applications

BO can handle:

- **Noise**: Incorporated into the GP model.
- **Parallelism**: Batched evaluations via generalized acquisition functions.
- **Constraints**: Modeled using additional GPs.
- **Multi-fidelity**: Combine approximate and accurate evaluations.
- **Random environments**: Optimize expected performance over stochastic conditions.

Common applications include hyperparameter tuning, engineering optimization, and experimental design.

In conclusion Bayesian Optimization is a sample-efficient strategy for optimizing expensive black-box functions. It combines probabilistic modeling with decision-theoretic selection of evaluation points, achieving excellent performance with minimal evaluations. The Gaussian Process prior, coupled with acquisition functions like EI and KG, makes BO a powerful and flexible tool in modern optimization.

1.4 Multi objective BO extension

In this thesis, the optimization task involves more than a single objective. Traditional BO methods have addressed single-objective and constraint-free

optimization problems. However, real-world applications often require optimizing multiple conflicting objectives under a set of constraints. The paper *"Predictive Entropy Search for Multi-objective Bayesian Optimization with Constraints"* [4] introduces PESMOC—a novel BO framework capable of addressing such complex problems.

PESMOC extends information-theoretic approaches in BO to the multi-objective, constrained domain. Specifically, it uses a predictive entropy search to identify evaluation points that maximize the information gain about the Pareto set, thereby efficiently discovering optimal trade-offs under feasibility conditions.

Problem Setting

The paper considers the following optimization problem:

$$\min_{x \in \mathcal{X}} \quad (f_1(x), f_2(x), \ldots, f_K(x)) \text{ s.t.} \qquad c_j(x) \geq 0, \quad j = 1, \ldots, C,$$

where $f_k(x)$ are objective functions and $c_j(x)$ are constraints, all modeled as black-boxes. Evaluations are noisy and costly, and no gradients are available.

A feasible solution is one that satisfies all constraints. Due to the conflicting nature of the objectives, the goal is to approximate the Pareto set $\mathcal{X}^*$ in the feasible space $\mathcal{F}$, which contains all non-dominated feasible solutions. These solutions represent trade-offs such that improving one objective would deteriorate at least one other.

PESMOC Framework

PESMOC is an acquisition function based on **predictive entropy search**, designed to maximize the expected information gain about the Pareto set. Given a Gaussian process (GP) prior for each objective and constraint, PESMOC selects the next evaluation point x to maximize the mutual information:

$$\alpha(x) = \mathbb{H}(\mathcal{X}^* | \mathcal{D}) - \mathbb{E}_y[\mathbb{H}(\mathcal{X}^* | \mathcal{D} \cup (x, y))],$$

where $\mathbb{H}(\cdot)$ denotes entropy and $\mathcal{D}$ is the set of observations. This criterion favors points whose evaluation would reduce uncertainty about $\mathcal{X}^*$ the most.

To simplify the intractable direct computation of the entropy over sets, the acquisition function is reformulated as:

$$\alpha(x) = \mathbb{H}(y | \mathcal{D}, x) - \mathbb{E}_{\mathcal{X}^*}[\mathbb{H}(y | \mathcal{D}, x, \mathcal{X}^*)].$$

This transformation simplifies implementation by swapping the complexity of entropy over sets with the tractable entropy of GP predictions.

Modeling with Gaussian Processes

Each black-box function is modeled with a GP, which provides a predictive distribution for new input points. This probabilistic modeling enables calculation of uncertainty and information gain. Hyperparameters of GPs are inferred using slice sampling, which integrates over their posterior to improve robustness.

GPs offer not only mean predictions but also a measure of uncertainty, which is essential for entropy-based acquisition. The covariance structure is typically defined using the Mat'ern kernel or similar smooth kernels, encoding assumptions about function regularity.

Entropy Approximation via EP

Expectation Propagation (EP) is used to approximate the conditional predictive distribution $p(y|\mathcal{D}, x, \mathcal{X}^*)$. EP approximates non-Gaussian factors in the integrals with Gaussian ones, allowing efficient analytical computation of the posterior entropy.

The EP updates are critical to accurately reflecting the effect of conditioning on the Pareto set and involve moment-matching steps. The EP approximation permits real-time evaluation of the acquisition function in practice.

Decoupled Evaluations

One of PESMOC's distinctive strengths is its ability to support **decoupled evaluations**. This means that the individual objective and constraint functions can be queried separately, rather than requiring all evaluations at the same input location. Such flexibility is highly beneficial in scenarios where the cost or feasibility of evaluating different functions varies.

In mathematical terms, PESMOC's acquisition function is additively decomposable as follows:

$$\alpha(x) = \sum_{k=1}^{K} \alpha^{\text{obj}} k(x) + \sum j = 1^{C} \alpha_j^{\text{const}}(x),$$

where $\alpha_k^{\text{obj}}(x)$ corresponds to the expected entropy reduction attributed to the k-th objective, and $\alpha_j^{\text{const}}(x)$ to that of the j-th constraint.

This decomposition allows PESMOC to not only choose the most informative input location x but also to determine which specific function (objective or constraint) should be evaluated at that location. In practice, this leads to

more efficient exploration strategies, particularly in domains where evaluations are bottlenecked by experimental constraints, such as physical testing or human assessments.

Experimental Evaluation

Experiments conducted on synthetic, benchmark, and real-world tasks (e.g., tuning neural networks and optimizing control parameters for robots) [4] demonstrate the effectiveness of PESMOC in handling constrained multi-objective optimization problems. The implementation of PESMOC used in the experiments is publicly available at `https://github.com/EduardoGarrido90/spearmint_ppesmoc`, allowing for reproducibility and further experimentation.

The results highlight several advantages of PESMOC:

- It outperforms state-of-the-art methods such as BMOO, particularly under tight feasibility constraints.
- Its ability to perform decoupled evaluations leads to better performance when evaluation costs differ across objectives and constraints.
- It is robust in the presence of noise and scales well with increasing problem complexity (number of objectives or constraints).
- In high-dimensional scenarios, it shows more stable convergence toward feasible and optimal Pareto solutions.

Performance was evaluated using standard metrics such as the hypervolume indicator, convergence to the true Pareto front, and computational overhead.

To better contextualize PESMOC, Table 1.1 compares it to other representative Bayesian optimization methods. **PESC** focuses on single-objective constrained problems, while **PESMO** extends to multiple objectives but lacks constraint handling. **BMOO** supports both multiple objectives and constraints but relies on scalarization and extended domination rules, which limit its scalability in high-dimensional or tightly constrained settings. In contrast, **PESMOC** uniquely supports all three capabilities—multi-objective optimization, constraint handling, and decoupled evaluations—making it a flexible and efficient solution for real-world applications.

Method	MO	Con	Dec-Eval	Notes
PESC	✗	✓	✗	Designed for single-objective problems with constraints.
PESMO	✓	✗	✗	Supports multiple objectives but no constraint handling.
BMOO	✓	✓	✗	Uses extended domination rules; limited in highly constrained or high-dimensional settings.
PESMOC	✓	✓	✓	Handles multiple objectives and constraints, supports decoupled evaluations, and flexible acquisition strategies.

Table 1.1: Comparison of Bayesian Optimization methods related to PESMOC

Chapter 2

CAD WORKFLOW

2.1 Introduction

One of the core objectives of this thesis project was the development of an end-to-end automation framework capable of executing the complete sequence of steps required for digital microchip generation. This process spans from High-Level Synthesis (HLS), converting C++ to RTL, through logic synthesis (RTL to gate-level netlist), and finally to physical implementation via Cadence Innovus (place-and-route: gate-level to physical layout).

Figure 2.1 provides a pyramidal overview of the framework, illustrating the hierarchical levels of abstraction and their dependencies. Each layer of the pyramid corresponds to a stage in the automation flow, which is discussed in detail in the respective dedicated chapters.

Figure 2.4 shows the complete diagram, highlighting the tools used (rounded parallelograms), input and output files (circles), and the results obtained (diamonds).

As the theoretical background and operating principles of each tool have already been discussed in previous chapters, the present section focuses exclusively on the implementation and structure of the automation framework.

Given the requirements for flexibility and modularity in such scripting tasks, Python was selected as the primary development language. Python offers a balanced combination of simplicity, flexibility, and readability—qualities particularly suited for orchestrating heterogeneous design flows. Compared to C-style languages, Python facilitates faster development cycles and dynamic data handling; and relative to Bash scripting, it offers significantly improved structure and maintainability. While Python does suffer from limitations such as lower execution speed and limited hardware-level control,

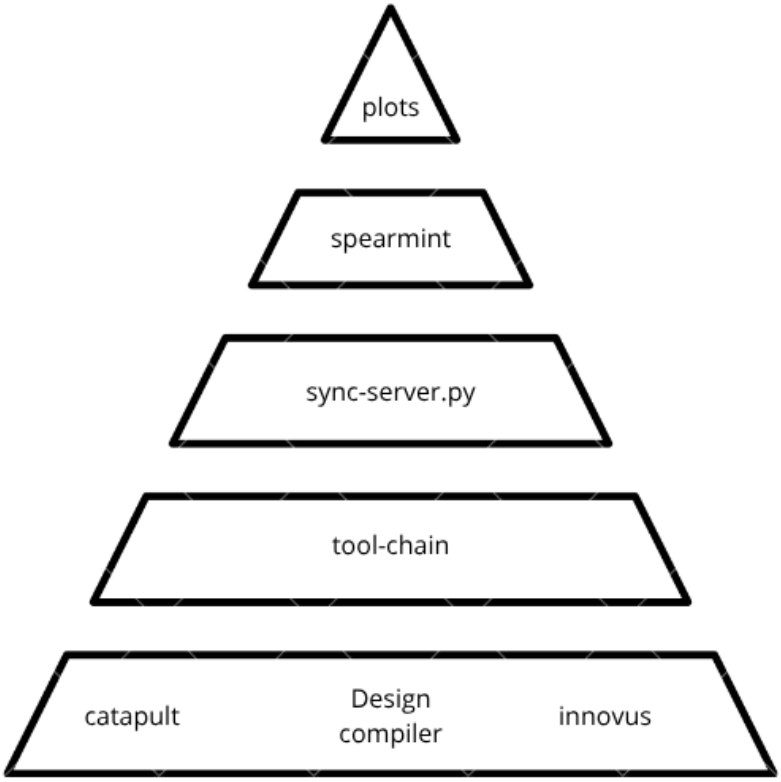

Figure 2.1: Summary of the thesis structure: each layer represents a level of abstraction dependent on the layers below.

these disadvantages are negligible in this context, where the script's primary function is to trigger shell commands and parse log files upon event-driven triggers.

2.2 High-level-synthesis with Catapult

Catapult is a High-Level Synthesis (HLS) tool developed by Synopsys, designed to translate behavioral hardware descriptions—typically written in high-level programming languages such as C or C++—into synthesizable Register Transfer Level (RTL) models. This approach significantly accelerates the hardware development process by reducing both design and functional verification time, when compared to traditional RTL-centric methodologies. Catapult is particularly well-suited for the design of hardware accelerators, as it enables automatic exploration of architectural configurations,

including pipeline depth, degree of parallelism, and loop unrolling strategies. The generated RTL output is compatible with standard downstream tools used in digital design flows, such as logic synthesis, and simulation, making Catapult a critical component in modern ASIC and FPGA development pipelines.

Core Steps in High-Level Synthesis

High-Level Synthesis (HLS) translates a high-level behavioral specification into a Register Transfer Level (RTL) implementation through a structured sequence of transformations. Each stage of the flow can be invoked using a corresponding `go <stage>` command, which controls the progression of synthesis tasks. In fig. 2.2 are showed all steps of a complete sinthesis.

1. **Front-End Parsing and Compilation (`go compile`)**
   The source-level design specification is parsed and semantically analyzed. The top-level function is identified, and a high-level behavioral model is constructed. Initial dataflow and control-flow representations are elaborated.

2. **Library Binding (`go libraries`)**
   Technology-specific components, such as arithmetic operators and memory macros, are linked to the design. This enables resource-aware synthesis and accurate estimation of area, delay, and power.

3. **Infrastructure Setup (`go setup`)**
   Hardware-level constraints and environment settings are defined. These include clock specifications, reset behavior, and I/O interface protocols (e.g., AXI, handshake). Design constraints such as latency, initiation interval (II), and throughput are also specified.

4. **Micro-Architecture Synthesis (`go extract`)**
   The untimed behavioral model is refined into a cycle-accurate microarchitecture. High-level synthesis directives (e.g., loop pipelining, unrolling) are applied. Key transformations include operation scheduling and finite state machine (FSM) generation.

5. **Resource Allocation (`go allocate`)**
   Hardware resources (e.g., ALUs, multipliers, memory banks) are allocated and mapped to the actual library component. Resource sharing

and duplication strategies are applied to balance area, timing, and performance.

6. **RTL Generation (`go generate`)**
   A synthesizable RTL design is generated in Verilog or VHDL. The output includes datapaths, control logic, module hierarchies, and interface wrappers. Design reports summarize key metrics such as resource utilization, latency, and throughput.

7. **Optional Power Estimation (`go power`)**
   If enabled, a power estimation step leverages switching activity annotations and technology parameters to estimate both dynamic and static power consumption.

Post-Synthesis

- **Functional Verification**
  Although not part of the `go` stages, functional verification is essential. The cycle-accurate RTL output is validated against the original C/C++/SystemC behavioral model using co-simulation tools such as SCVerify. This ensures functional correctness and behavioral equivalence.

- **Logic Synthesis**
  The RTL output generated by Catapult is passed to a downstream logic synthesis tool (e.g., Synopsys Design Compiler, Cadence Genus, or Xilinx Vivado). This stage includes technology mapping, optimization for area, power, and timing, and results in a gate-level netlist suitable for physical implementation.

HLS Recipe

To enable the high-level synthesis (HLS) process, several key components must be prepared:

- **C++ Source Files:** The hardware accelerator must be described using a combination of `.cpp` and `.h` files that define its untimed functional behavior.

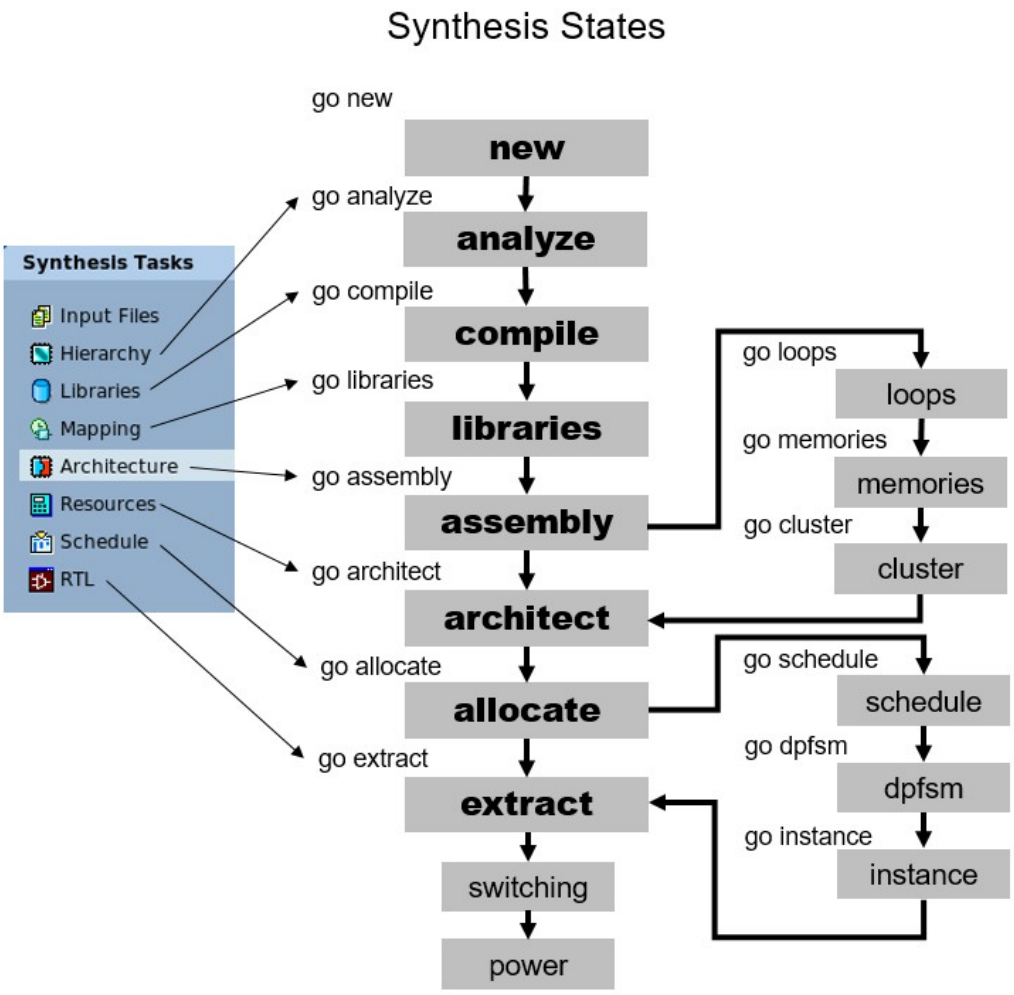

Figure 2.2: Catapult synthesis steps

- **Technology Library:** A library file describing the target technology characteristics (e.g., 28nm SOI) is required. This file must be compiled using Catapult's integrated Library Builder to enable accurate resource mapping, area, and timing estimations.
- **Tcl Script:** A `.tcl` script is used to automate the synthesis flow. Since Catapult requires multiple invocation stages (`go compile`, `go extract`, etc.), scripting ensures reproducibility and enables a fully autonomous execution pipeline.

Catapult Tcl Key Features

The Catapult Tcl scripting interface provides powerful capabilities for automating and customizing the HLS process. Key features include:

- **Parameterization via Global Variables:**
  By defining key variables at the beginning of the script, it becomes easier to create a fully parametric and reusable flow. This approach improves readability and simplifies design-space exploration.

- **Timeout and Error Management:**
  The script can be extended to handle execution timeouts and unexpected failures, ensuring robustness during batch runs or long simulations.

- **Source Code Tuning:**
  While the `.cpp` and `.h` files are not inherently parametric, the script leverages `awk` within Bash to programmatically patch source files prior to synthesis. In future implementations, they can be entirely rewritten via the `echo` command.

- **Pipeline and Parallelism Configuration:**
  Through symbolic variables (e.g., loop initiation intervals and unrolling factors), the script allows dynamic configuration of parallelism at loop level. Listing 5.4 shows an extract of the .tcl script, which configures, for each loop defined in a variable, its level of pipelining or unrolling.

Listing 2.1: Pipeline directives in Tcl

```
dict for {key value} $loop_dict {
  set z $value
  if { $z == 4 } {
    directive set /$CXX_NAME/core/$key -PIPELINE_INIT_INTERVAL 0
  } elseif { $z == 3 } {
    directive set /$CXX_NAME/core/$key -PIPELINE_INIT_INTERVAL 2
  } elseif { $z == 2 } {
    directive set /$CXX_NAME/core/$key -PIPELINE_INIT_INTERVAL 1
  } elseif { $z == 1 } {
    set zz [ expr $MAX_OUTPUT_CHANNELS / 2 ]
    directive set /$CXX_NAME/core/$key -UNROLL $zz
  } elseif { $z == 0 } {
    directive set /$CXX_NAME/core/$key -UNROLL yes
  }
}
```

- **Memory Interleaving Strategy:**
  Interleaving factors for memory access are set dynamically based on

runtime values such as loop parameters or buffer sizes. This tuning enables bandwidth-aware memory mapping.

- **PowerPro Integration:**
  When enabled, power optimization features are configured and executed conditionally. Power estimates are collected using switching activity annotations. Listing 2.2 shows the main power optimization commands.

Listing 2.2: Example of Power Pro commands implementation

```
if { $power_pro == 1 } {
  directive set -/USE_MODES/base/PWR_OPT_WEIGHT 1.0
  directive set -/USE_MODES/base/PWR_ACTIVITY_TIME {{} {} {} {}}
  directive set -/USE_MODES/base/PWR_CLOCK_MODE {clk default}
  go power
  flow run /PowerAnalysis/report_post_pwropt_VHDL
}
```

- **Configurable RTL Synthesis with Design Compiler:**
  Conditional logic allows for synthesis under multiple configurations (with or without power optimization). This flow integrates downstream RTL tools for gate-level netlist generation. In listing 2.3 and example of implementation is shown.

Listing 2.3: Design compiler integration command implementation example

```
if { ($dc_int_rtl == 1) || ($dc_int_rtl == 3) } {
  flow run /DesignCompiler/dc_shell ./concat_rtl.vhdl.dc vhdl
  file rename ... base_gate.dc.vhdl.v
}
if { (($dc_int_rtl == 2) || ($dc_int_rtl == 3)) && ($power_pro == 1) } {
  flow run /DesignCompiler/dc_shell ./concat_power.vhdl.dc vhdl
  file rename ... power_gate.dc.vhdl.v
}
```

- **Cycle-Accurate Verification with SCVerify:**
  Verification is performed by toggling a testbench flag in the source header file and running co-simulation for both configurations. This setup is useful for estimating execution time differences between untimed and RTL implementations.

Listing 2.4: Testbench quantization configuration and simulation

```
set EN_QUANTIZATION_TB 0
exec awk -v var=$EN_QUANTIZATION_TB ... > defs_synthesis_mod1.h
exec cp defs_synthesis_mod1.h defs_synthesis.h
flow run /SCVerify/launch_make ... SIMTOOL=msim sim

set EN_QUANTIZATION_TB 1
exec awk -v var=$EN_QUANTIZATION_TB ... > defs_synthesis_mod1.h
```

```
8 exec cp defs_synthesis_mod1.h defs_synthesis.h
9 flow run /SCVerify/launch_make ... SIMTOOL=msim sim
```

HLS - Result

After the high-level synthesis (HLS) process is completed, several valuable outputs are generated:

- **RTL Netlist:**
  Catapult produces a synthesizable RTL netlist in Verilog or VHDL, representing the hardware implementation of the original high-level model. This can include both power-optimized and non-optimized versions, depending on whether PowerPro was enabled during synthesis.
- **Timing Constraints File (`.sdc`):**
  A Synopsys Design Constraints (SDC) file is generated, containing all relevant timing constraints, including clock definitions, reset conditions, and interface timing requirements. This file is crucial for downstream logic synthesis and physical implementation.
- **Synthesis Reports:**
  Detailed reports are available summarizing:
  - Resource utilization (e.g., number of registers, multipliers, memory blocks)
  - Latency (measured in clock cycles)
  - Initiation Interval (II)
  - Estimated dynamic and static power (if PowerPro is enabled)
- **Limitations of the HLS Model:**
  Although the output provides accurate estimates and functional RTL, it does not reflect all the physical characteristics of a finalized silicon implementation. Key aspects that are not yet accounted for include:
  - Clock tree synthesis and its associated skew and insertion delay
  - Interconnect effects, such as wire delay and parasitics
  - Additional buffering or logic inserted during placement and routing

  These effects are typically addressed in later stages of the ASIC/FPGA flow, particularly during place-and-route and post-layout simulations.

Therefore, while HLS results provide valuable early insights into performance and resource usage, final validation and optimization must rely on full backend synthesis and implementation tools.

Pseudo-codes

Here are some pseudo-code implementations of the 2D-convolutional layer (**CONV**) and fully connected (FC) accellerators. Taken from Urbinati's work [**?**] and used in the experimental results of this thesis, with a detailed explanation of the functionality available in the original paper.

2.3 Logic Synthesis with Design Compiler

Logic synthesis proceeds through a series of steps, each of which can be controlled via specific commands. These steps include elaboration, optimization, technology mapping, and timing-driven restructuring, in order to produce more precise power, area estimations.

The logic synthesis flow is orchestrated through a highly parameterized TCL script targeting Synopsys Design Compiler. This script automates the translation of RTL-level VHDL descriptions—generated by Catapult–into optimized gate-level netlists, ready for the downstream of place-and-route and power analysis tools.

The script begins by initializing key design variables, such as the top-level module name, the target operating frequency, the selected multiplier architecture, and many other configuration parameters. It then sets paths to required RTL sources, technology libraries, while also configuring the tool for either low-leakage or high-power standard cell libraries depending on the specified logic type.

A complete flow is implemented, including RTL analysis and elaboration, clock definition and timing constraints setup, power domain definitions, and the application of design constraints via SDC. A standard practice is to apply a delay equal to 10% of the clock period on input and output ports, representing typical timing margins without any custom optimization. If switching activity files (SAIF) are enabled, they are imported to allow more accurate dynamic power estimation; otherwise, static toggle rates are assigned.

The script proceeds to perform logic synthesis using the compile_ultra command, enabling automatic advanced optimizations such as register retiming and clock gating across hierarchies. Reports on timing, area, power, and clock gating efficiency are generated and stored in dedicated folders. The

Listing 2.5: ST-based 2D-Conv.

```
#include <ac_int.h>

#pragma map_to_operator "X"
int32 st_multiplier_function(uint3 CONFIG, int16 A, B){...}

void conv2d(
        int4 IBUF_A[IS], IBUF_B[IS], IBUF_C[IS], IBUF_D[IS],
        int4 WBUF_A[WS], WBUF_B[WS], WBUF_C[WS], WBUF_D[WS],
        ac_int<28, true> OBUF[OS],
        uint3 CONFIG, uint1 RESET) {
 int ic_lim;
 if (CONFIG==(8x8 || 8x4)) { ic_lim = IC/2-1; }
 else if (CONFIG==4x4) { ic_lim = IC/4-1; }
 else { ic_lim = IC-1; }

 #pragma pipeline_init_interval 1
 for (int oh=0; oh<OH; oh++) {
  #pragma pipeline_init_interval 1
  for (int ow=0; ow<OW; ow++) {
   #pragma pipeline_init_interval 1
   for (int oc=0; oc<OC; oc++) {
    if (RESET==1) { OBUF[OC*(OH*oh+ow)+oc] = 0; }
   }
   #pragma pipeline_init_interval 1
   for (int ic=0; ic<IC; ic++) {
    #pragma pipeline_init_interval 1
    for (int kh=0; kh<KH; kh++) {
     #pragma pipeline_init_interval 1
     for (int kw=0; kw<KW; kw++) {
      // Memory addressing and concatenating logics for A
      int A_idx = IC*(IW*(oh+kh)+(ow+kw))+ic;
      int4 A_HH = IBUF_A[A_idx];
      int4 A_HL = IBUF_B[A_idx];
      int4 A_LH = IBUF_C[A_idx];
      int4 A_LL = IBUF_D[A_idx];
      int16 A = ((A_HH<<12)&0xF000) | ((A_HL<<8)&0x0F00) |
                ((A_LH<< 4)&0x00F0) | ( A_LL
  &0x000F);
      #pragma unroll OC
      for (int oc=0; oc<OC; oc++) {
       // Memory addressing and concatenating logics for B
       int B_idx = OC*(IC*(KH*kh+kw)+ic)+oc;
       int4 B_HH = WBUF_A[B_idx];
       int4 B_HL = WBUF_B[B_idx];
       int4 B_LH = WBUF_C[B_idx];
       int4 B_LL = WBUF_D[B_idx];
       int16 B = ((B_HH<<12)&0xF000) | ((B_HL<<8)&0x0F00) |
                 ((B_LH<< 4)&0x00F0) | ( B_LL
  &0x000F);
       // Reconfigurable ST-based PSMAC array
       int28 P = st_multiplier_function(CONFIG,A,B);
       OBUF[OC*(OH*oh+ow)+oc] += P;
      } //oc
     } //kw
    } //kh
    if (ic==ic_lim) { break; }
   } //ic
   #pragma pipeline_init_interval 1
   for (int oc=0; oc<OC; oc++) {
    // Quantization logic (Eq. 14) and ReLU (when needed)
    ...
   }
  } //ow
 } //oh
}
```

Listing 2.6: ST-based FC.

```
#include <ac_int.h>

#pragma map_to_operator "X"
int32 st_multiplier_function(uint3 CONFIG, int16 A, B){...}

void fc(int4 IBUF1_A[IS/2], IBUF2_A[IS/2],
        int4 IBUF1_B[IS/2], IBUF2_B[IS/2],
        int4 IBUF1_C[IS/2], IBUF2_C[IS/2],
        int4 IBUF1_D[IS/2], IBUF2_D[IS/2],
        int4 WBUF1_A[WS/2], WBUF2_A[WS/2],
        int4 WBUF1_B[WS/2], WBUF2_B[WS/2],
        int4 WBUF1_C[WS/2], WBUF2_C[WS/2],
        int4 WBUF1_D[WS/2], WBUF2_D[WS/2],
        ac_int<28, true> OBUF[OS],
        uint3 CONFIG, uint1 RESET) {
 int ia_lim;
 if (CONFIG==(8x8 || 8x4)) { ia_lim = IA/4-1; }
 else if (CONFIG==4x4) { ia_lim = IA/8-1; }
 else { ia_lim = IA/2-1; }

 #pragma pipeline_init_interval 1
 for (int oa=0; oa<OA; oa++) {
  if (RESET==1) { OBUF[oa] = 0; }
 }
 #pragma pipeline_init_interval 1
 for (int ia=0; ia<IA/2; ia++) {
  // Memory addressing and concatenating logics for A1, A2
  int a_idx = ia;
  int4 A1_HH = IBUF1_A[a_idx], A2_HH = IBUF2_A[a_idx];
  int4 A1_HL = IBUF1_B[a_idx], A2_HL = IBUF2_B[a_idx];
  int4 A1_LH = IBUF1_C[a_idx], A2_LH = IBUF2_C[a_idx];
  int4 A1_LL = IBUF1_D[a_idx], A2_LL = IBUF2_D[a_idx];
  int16 A1 = ((A1_HH<<12)&0xF000) | ((A1_HL<<8)&0x0F00) |
             ((A1_LH<< 4)&0x00F0) | ( A1_LL
  &0x000F);
  int16 A2 = ((A2_HH<<12)&0xF000) | ((A2_HL<<8)&0x0F00) |
             ((A2_LH<< 4)&0x00F0) | ( A2_LL
  &0x000F);
  #pragma unroll OA
  for (int oa=0; oa<OA; oa++) {
   // Memory addressing and concatenating logics for B1, B2
   int b_idx = OA*ia+oa;
   int4 B1_HH = WBUF1_A[b_idx], B2_HH = WBUF2_A[b_idx];
   int4 B1_HL = WBUF1_B[b_idx], B2_HL = WBUF2_B[b_idx];
   int4 B1_LH = WBUF1_C[b_idx], B2_LH = WBUF2_C[b_idx];
   int4 B1_LL = WBUF1_D[b_idx], B2_LL = WBUF2_D[b_idx];
   int16 B1 = ((B1_HH<<12)&0xF000) | ((B1_HL<<8)&0x0F00) |
              ((B1_LH<< 4)&0x00F0) | ( B1_LL
  &0x000F);
   int16 B2 = ((B2_HH<<12)&0xF000) | ((B2_HL<<8)&0x0F00) |
              ((B2_LH<< 4)&0x00F0) | ( B2_LL
  &0x000F);
   // Reconfigurable ST-based PSMAC array
   int28 P1 = st_multiplier_function(CONFIG,A1,B1);
   int28 P2 = st_multiplier_function(CONFIG,A2,B2);
   int28 P1_plus_P2 = P1+P2;
   OBUF[oa] += P1_plus_P2;
  } //oa
  if (ia==ia_lim) { break; }
 } //ia
 #pragma pipeline_init_interval 1
 for (int oa=0; oa<OA; oa++) {
   // Quantization logic (Eq. 6) and ReLU (when needed)
   ...
 }
}
```

Figure 2.3: 2D-Conv and FC accelerators (pseudo-code).

synthesized netlist is exported in both Verilog and DDC formats, along with the final SDC constraints for backend integration.

Finally, the design is flattened and recompiled to allow more aggressive optimizations, preparing the result for physical synthesis or signoff-level simulations.

2.4 Place and route

Starting from the synthesized output of Design Compiler, Innovus performs a series of backend operations including floorplanning, placement, clock tree synthesis (CTS), routing, and parasitic extraction. These steps are executed in a timing- and power-aware manner to ensure that the final layout meets stringent performance and manufacturability requirements.

By leveraging advanced optimization engines and parallelism, Innovus supports rapid design convergence across multiple objectives—such as timing closure, signal integrity, and power delivery network robustness. The tool integrates seamlessly into the broader ASIC design ecosystem and generates all necessary signoff data, including standard delay format (SDF), parasitic extraction files (SPEF), and power/timing reports. Its scripting flexibility and automation capabilities make it a key enabler for tapeout-quality digital IC implementation.

The physical implementation flow is driven by a comprehensive TCL script executed within Cadence Innovus, responsible for transforming a synthesized netlist into a placed-and-routed layout, complete with parasitic extraction and signoff-quality reports. The script automates floorplanning, power planning, placement, clock tree synthesis, routing, and timing optimization, following a standard digital backend methodology.

The design under implementation is specified through variables such as the module name, operating frequency. The netlist and associated constraints generated by Design Compiler are imported, along with a collection of technology and standard cell *.LEF* files required for layout generation.

The flow begins with initial floorplanning, where the core area is defined using a density-driven strategy. Power rings and stripes are added using addRing and addStripe, ensuring robust power distribution. Global nets for power and ground (vdd, gnd) are then connected across the entire design.

Placement is executed using timing-driven and congestion-aware strategies, enabling optimal cell positioning. This is followed by pre-clock-tree optimization (preCTS) to refine timing paths, and then clock tree synthesis (CTS) using ccopt_design, which inserts and balances the clock distribution network.

Post-CTS optimization and detailed routing are carried out sequentially, each step followed by timing analysis using timeDesign, ensuring that setup and hold constraints are satisfied. After final routing, the script performs parasitic extraction (extractRC) and generates output files including *.SDF* (for back-annotated simulation), *.SPEF* (for signoff parasitics), and the final

netlist.

A comprehensive set of reports can be produced, including:

- Power analysis, either
  - under default toggle rate assumptions, or
  - based on simulation data for more realistic results
- Area utilization
- Gate count

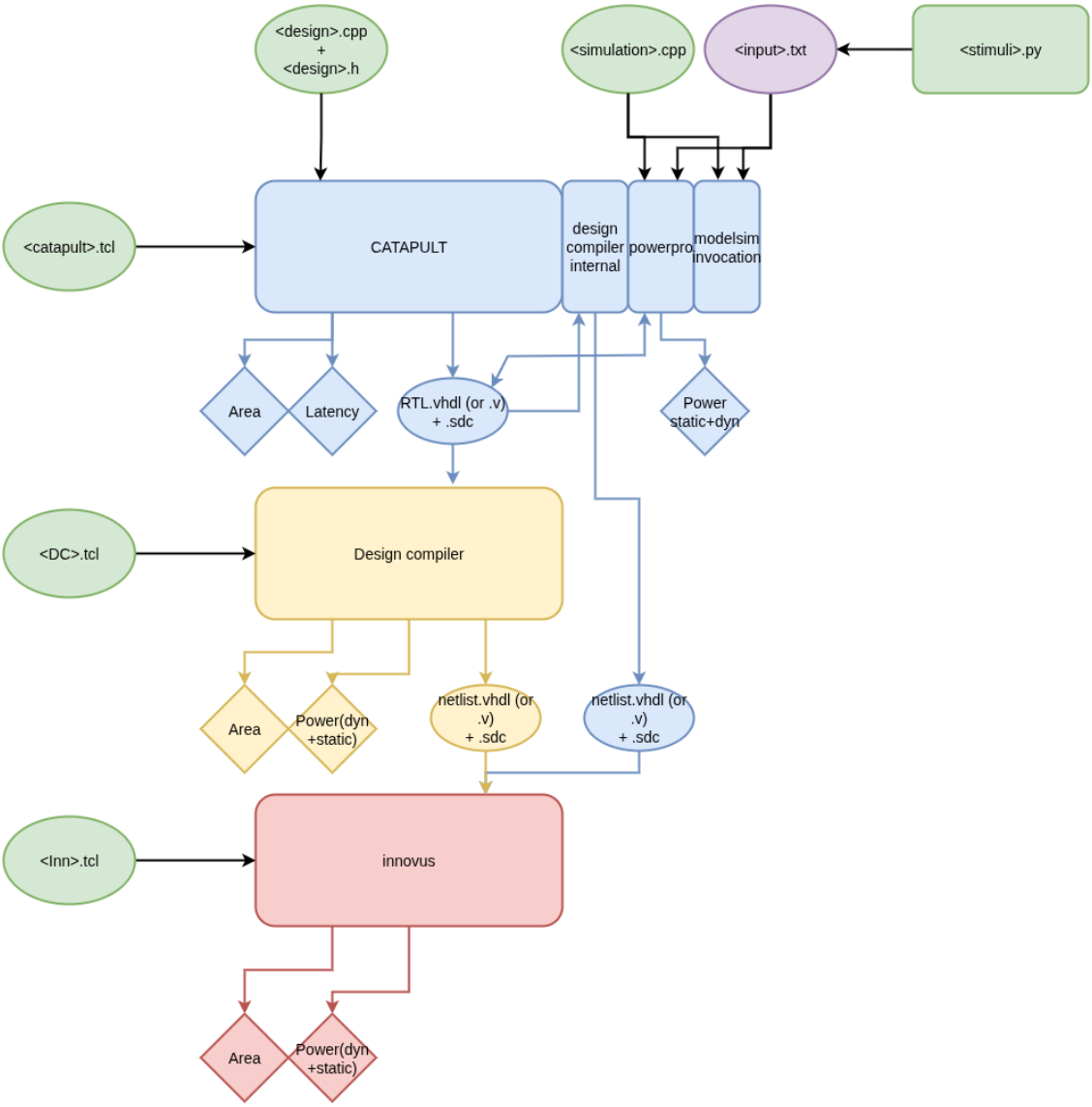

Figure 2.4: Full CAD sequence

Chapter 3

TOOL-CHAIN

3.1 FSM main tool-chain module

The main toolchain script, implemented in Python, functions as the central orchestration layer for the entire execution workflow. It coordinates all stages—from data acquisition and parsing, through tool invocation and intermediate data collection, to the generation and return of final results.

Conceptually, the script follows a *finite-state-machine* (FSM) paradigm. Each iteration in the toolchain pipeline is decomposed into a series of well-defined stages, each corresponding to the execution of a specific CAD tool. Within each stage, the process can transition through sub-states such as `ready`, `busy`, or `end`, enabling fine-grained control and precise monitoring of execution progress.

The execution can be logically divided into two primary phases:

3.1.1 Setup Stage

In this phase, all relevant database files containing parameter sets that can be hand-passed are loaded into memory and stored in structured variables. A preliminary report is also generated, providing an up-to-date view of the current progress without requiring a full sequential execution of all pipeline steps. Databases from the *Spearmint* module, which contain parameter sets requested directly by Spearmint's research, are also loaded at this stage, ensuring that the environment is fully synchronized with prior experiment configurations.

3.1.2 Loop Stage

The main processing loop manages the dynamic execution of CAD tool stages. If enabled, the loop terminates once all iterations[1] of the complete end-to-end digital design flow have been completed. During this stage, new CAD tool executions may be triggered, provided that sufficient tool licenses are available.

Before launching a new execution, the system checks the caching subsystem to determine whether valid results already exist, thus avoiding redundant runs. For example, if a Catapult synthesis has already been performed, the tool reloads the results before directly invoking Design Compiler. Periodic statistics are computed and printed, and a dedicated evaluation function is invoked to inspect each iteration for tool-specific termination conditions or error states. Once all CAD stages for a given iteration have been executed, the system generates a new cache entry and produces a consolidated report file, which can optionally be transmitted to a remote server for storage or further analysis.

Parameterization: Knobs and Values

To support repeated execution with varying configurations, a robust parameterization mechanism is essential. The adopted approach relies on storing configuration data in nested Python dictionaries, which are easily serialized using the JSON format. These nested dictionaries are then aggregated into lists, each representing a specific iteration.

This tree-like configuration model reflects the hierarchical nature of design parameters—e.g., `database` → `connection` → `timeout`. The approach offers high flexibility, extensibility, and ease of use, especially for dynamic access and environment-specific overrides. It also integrates well with industry-standard formats like JSON or YAML. Compared to global variables or flat configuration files, this structure is more scalable, maintainable, and semantically clear for complex applications.

This architecture enables the toolchain to operate in a resource-aware, fault-tolerant, and incrementally evaluable manner, ensuring that partially

[1] an iteration refers to a complete execution of the end-to-end digital design flow, from high-level synthesis through HLS, logic synthesis and place-and-route, producing a set of results for a given parameter sets.

completed iteration are leveraged efficiently and that the execution state remains transparent at all times.

3.2 Thread Module

The `thread` module provides the core mechanisms for managing the tool execution within the automated design flow. It encapsulates both initialization routines and the orchestration of tool-specific CAD launch, while also implementing timeout supervision and cleanup procedures. This modular organization allows for clear separation of concerns and facilitates future extensibility.

`generic_thread(tool, index)`

The `generic_thread()` function provides an abstraction layer for launching the tool processes based on the specified tool name and task index. Prior to invocation, it consults the `read_timeout()` utility to determine whether the tool has exceeded its allocated execution time window. If a timeout condition is detected and global timeout enforcement is enabled, the task is flagged as `timed_out` and execution is halted. Otherwise, the function delegates control to the appropriate tool-specific thread function—`catapult_thread()`, `DC_ext_thread()`, or `innovus_thread()`—according to the task configuration.

Execution CAD Tools: `[tool]_thread(index)`

This function is implemented for all three CAD tools and is responsible for preparing the working environment, generating the corresponding TCL scripts, and executing the tool through the `terminal_{tool}` function. During execution, a timestamp is recorded and the task status is updated to `busy` to facilitate monitoring, all this information are saved inside the main dictionary.

Additional steps are performed in the case of Innovus. Before execution, the thread verifies that a valid netlist has been provided; if the input is invalid, the task is marked as faulty and the status is reverted to `catapult_ready`. Furthermore, a patch is applied to the `.sdc` file.

`thread_terminal_kill(index)`

The `thread_terminal_kill()` routine terminates the `tmux` session associated with a given task and removes its index from the corresponding `running_tasks` list. This centralized cleanup procedure ensures consistency between in-memory state and process-level execution, while preventing resource leakage (also licenses).

Timeout Management and Error Recovery

Timeout supervision is implemented through the `out_of_time()` function, which iterates over all active tasks to compute elapsed execution times based on their recorded start timestamps. If the duration exceeds the predefined tool-specific limit, the task is forcibly terminated, its status is updated to `tool_timeout`, and a marker file is created on disk to document the event. This mechanism prevents stalled processes from indefinitely blocking the execution pipeline, thereby maintaining the responsiveness and reliability of the overall workflow.

3.3 Tool Terminal Module

This module provides runtime control and terminal session management for the execution of individual toolchain stages. It leverages the `tmux` terminal multiplexer to isolate and supervise each synthesis tool invocation (Catapult, Design Compiler, and Innovus), allowing asynchronous, parallel, and restartable runs in a shared server environment.

Session Initialization and Environment Setup. Upon import, the module sources the required EDA tools using `tool_folder.source_eda_tools()`, ensuring that all terminal sessions are correctly initialized with the appropriate environment variables.

Tool Execution via Tmux. All tools are launched through dedicated functions following the pattern `[tool]_terminal()`, which:

- create a new `tmux` session named after the tool and the run index,
- set up the working environment and the execution script,
- run the tool in batch/`no_gui` mode,

- capture logs (via `tee` or redirection) for later debugging and post-processing.

In addition, **Catapult** (`terminal_cata()`) performs a preparatory step: it automatically generates input stimuli using a helper Python script, parameterized by the accelerator type (e.g., `conv2d`, `fcmei`, `dwconv`), before executing the high-level synthesis flow.

The choice of `tmux` over direct subprocess execution provides several advantages: it enables session persistence after disconnection, decouples the execution from the Python process, and facilitates real-time inspection or manual intervention when needed. The module also avoids tight coupling to the execution logic of individual tools by isolating them into named tmux sessions, following a reproducible and traceable naming scheme.

This modular approach supports horizontal scaling and fault recovery, and it integrates seamlessly with the tool dispatch scheduler via the candidate index, ensuring deterministic session tracking and reproducibility of log outputs.

tool-folder: Functional Overview

This module provides essential utilities for setting up, managing, and cleaning up the directory structure and configuration environment of Electronic Design Automation (EDA) workflows. It facilitates deterministic path generation, tool-specific data management, and preprocessing operations critical to the seamless orchestration of multi-stage toolchains such as Catapult, Design compiler (DC_ext), and Innovus.

Beyond the initial role of environmental preparation, the `tool-folder` module encapsulates logic for deterministic folder generation and runtime file management tailored to each tool stage. Folder organization is guided by the conversion of configuration dictionaries into hashable string representations via the `dictionary_to_string()` and `hash_md5()` utilities. These functions ensure the creation of unique and reproducible directory names based solely on input parameters, thereby facilitating both traceability and cache reuse. Folder paths are constructed recursively through `generic_tool_dir()`, which encodes tool-specific hierarchies and supports toggling between folder- and file-level resolution.

To support downstream tool execution and integration, the module automates the generation of tool-specific export paths. For instance, `script_path()` and `script_name()` yield deterministic *.tcl* script filenames and their respective locations for execution within tool-specific flows. The `catapult_export()`

routine consolidates synthesized files such as .VHDL, .V, timing reports, and constraint files from Catapult-generated directories, packaging them into a standardized output format suitable for use by later tools in the flow.

Cleaning routines are provided to prevent filesystem clutter and guarantee a clean execution state. The `catapult_cleaner()` function removes stale intermediate directories and large waveform files.

Timing constraint patching is handled through `sdc_pathcer()`, which programmatically adjusts clock specifications in SDC files to reflect updated frequency settings, ensuring consistent timing assumptions across tools. Finally, the `innovus_preparator()` function orchestrates the export and formatting of all files required for Innovus placement and routing, including netlists, .SDC, and .Tcl scripts.

Tool Chain Cleaner

This module implements an automated sanitation utility for tool-chain project directories. Its primary objective is to enforce workspace hygiene by programmatically removing obsolete or unauthorized files and optionally pruning empty subdirectories. The cleanup process operates recursively over the standard tool output directories: `./catapult`, `./DC_ext`, and `./innovus`.

Tool Constants Module

This module acts as a centralized repository of constants shared across all components of the toolchain. Its purpose is to maintain consistency and configurability, while avoiding hard-coded parameters throughout the codebase.

Tool Database module

This module operates as the central registry and synchronization layer between tool execution states and external optimization engines, such as Spearmint. Its core responsibility revolves around the ingestion, enumeration, and export of a dynamically evolving database encoded in JSON format.

Real-time statistics regarding toolchain status are collected via `statistics_status()`, which tallies entries by execution state and identifies active Spearmint-driven evaluations. This status-tracking enables conditional scheduling and progress reporting. The resulting dictionaries can be used to visualize bottlenecks or to prioritize the dispatch of new requests.

The system supports import of entire databases through `import_new_dbs()`, which merges compatible entries from a specified folder into the active JSON database, ensuring proper deduplication and reindexing. Externally generated request objects, such as those produced by Spearmint, are integrated via `import_spearmint()`, with hash-based caching and request naming to prevent redundant reprocessing. Valid responses are exported using `export_spearmint()` and `export_spearmint_unique()`, completing the feedback loop between optimization and synthesis.

The `generate_tcl_script()` function constructs tool-ready Tcl scripts from JSON entries, appending reusable pre-written logic. This ensures parameterized reproducibility of tool flows while abstracting configuration propagation.

To avoid resource waste, the `catapult_flush_higher_freq_by_index()` function discards design candidates that match a given configuration but exceed its frequency target. This pruning mechanism prevents reevaluation of higher-frequency versions of the same design point when a lower-frequency variant has already been marked as infeasible.

tool cache

This module implements a robust backend engine for reading, validating, and processing toolchain requests based on JSON data files. It also provides fault-tolerant caching to support reproducibility, traceability.

Request and Cache Management

Incoming requests are detected and processed using `process_requests()`, which scans the designated input folder for JSON files prefixed with `Req_`. Each request is either matched against the cache or staged for evaluation, depending on system state and flags.

For each request, a hash is computed based on a reduced dictionary of base parameters. This hash is used as a unique identifier to check for cached results. If a cached response is found, it is reused to populate the output directory. Otherwise, the request is copied to the queue for deferred evaluation.

The `extract_base_dict()` function strips extraneous data (e.g., tool-specific reports) to isolate core parameters for consistent hashing. This ensures cache coherence and prevents redundant recomputation.

Data Validation and Structural Checks

Validation is performed using two complementary functions. The `is_good_element()` function provides a high-level check for the presence of required subreports, ensuring that all critical stages (e.g., Catapult, DC, Innovus) are represented.

The more detailed `is_data_ok()` function performs nested validation of critical fields. It ensures that numeric performance metrics (e.g., area, power, latency) are present and properly typed. Invalid records are flagged with detailed error logs, and faulty cache entries are pruned accordingly.

To keep the cache clean, `remove_faulty_cache()` iterates through existing cache files, validating their contents. Files that do not meet the expected structure are removed to avoid corrupt downstream behavior.

tool check

`check_by_code(tool_name, dictionary, code, delete=False)`
This function verifies the existence of a marker or flag file (`code`) within the directory structure associated with a specific tool. The directory path is dynamically constructed using metadata provided in the `dictionary` argument. If the marker is detected, the file may optionally be deleted based on the `delete` parameter. This functionality is commonly employed to confirm tool execution completion or to detect success indicators post-run.

`search_line(tool_name, dictionary, file_name, target_string)`
This function performs a search operation within a given text file (typically a log file) to locate a specified `target_string`. It returns one of the following status codes:

- `0` if the target string is found;
- `1` if the target string is not found;
- `2` if the file does not exist.

This logic underpins automated failure detection, warning identification, and conditional workflows driven by tool-generated logs.

`generic_tool_report(tool, dictionary, clean=False)`
A dispatch function abstracting the tool-specific reporting process. Based on the input tool name (e.g., `"catapult"`, `"DC_ext"`, or `"innovus"`), this

function determines the correct working directory and calls the corresponding report generator from the `tool_rpt` module. For tools such as Catapult, an optional cleaning routine can be triggered before report extraction. This abstraction offers a uniform interface across heterogeneous tools.

Tool Report

Overview and Functionality

This module provides a comprehensive framework for parsing and consolidating key results indicators extracted from a variety of CAD tool reports. The framework is designed to handle heterogeneous sources of data, enabling the systematic aggregation of performance, timing, area, and power metrics into a unified dictionary. By abstracting the details of each tool's reporting format, it facilitates downstream analysis and comparison of design outcomes across different synthesis and implementation flows.

Catapult Report Extraction

For reports generated by the Catapult high-level synthesis tool, the module offers specialized routines for extracting essential design metrics. Specifically, the `extract_slack()` function retrieves timing slack information, `extract_area()` quantifies hardware area utilization, and `extract_latency_throughput()` derives latency and throughput values from simulation outputs.

Design Compiler Report Extraction

The framework also includes dedicated functions for parsing reports produced by Synopsys Design Compiler. The `dc_extract_total_cell_area()` function determines the total synthesized cell area, `dc_extract_power()` estimates the power consumption, and `dc_extract_slack()` obtains the reported timing slack. To facilitate a holistic analysis, the `DC_int_full_report()` routine integrates these results, combining outputs from both standard and power-optimized synthesis flows.

Innovus Report Extraction

For the physical implementation stage, the module supports the parsing of reports generated by Cadence Innovus. The `innovus_report_summary()`

function extracts timing-related metrics such as Worst Negative Slack (WNS), Total Negative Slack (TNS), and placement density. The `innovus_report_area()` function reports the total physical layout area, while `innovus_report_power()` evaluates post-routing power consumption, thus enabling a post-layout assessment of the design's physical and power characteristics.

Report Aggregators

At a higher abstraction level, the module provides unified aggregator functions that compile all relevant metrics from each CAD environment into structured dictionaries. These include `catapult_full_report()`, `DC_ext_full_report()`, and `innovus_full_report()`.

Chapter 4

SPEARMINT STAGE

4.1 Introduction

This section outlines the configuration steps required to perform MOBO of an unknown objective function using Spearmint. The goal is to create an application whose behavior is compatible with the objectives of this thesis project.

The initial phase consisted in analyzing the internal workflow of Spearmint. By examining one of the provided example configurations, it is possible to understand how the search space is defined, how candidate configurations are evaluated, and how the corresponding results are stored and utilized in subsequent iterations of the optimization process.

Once the framework has been understood, a custom search space was constructed for the case study. This required identifying the most relevant design parameters (knobs) and assigning suitable value ranges to each. The selection of variables was based on their expected impact on key performance metrics.

With this understanding, we then apply it to the current case study, identifying how to provide the necessary configuration and how to customize it to address potential issues—such as offloading computations to a different server.

4.2 Spearmint Framework

Spearmint is a Python-based framework for Bayesian Optimization, available on GitHub. In this work, the fork developed by Eduardo Garrido[1] is used, as it integrates extensions for multi-objective optimization and has been adopted in previous academic projects.

The framework is structured around a modular architecture. The core optimization process is launched by executing the `main.py` script, which serves as the entry point. Before execution, a configuration file named `config.json` must be prepared and placed in the working directory. This file defines the search space, the type of optimization (e.g., single- or multi-objective), the number of iterations, and other relevant settings.

When `main.py` is launched, it parses the `config.json` file and initializes the internal optimization pipeline accordingly. One of the key elements specified in the configuration is the evaluation script, typically named `bo.py`. This script must implement an `evaluate()` function, which takes as input a set of parameter values sampled by the optimizer and returns one or more objective values (e.g., area, latency, power).

In summary:

- `config.json` defines the problem setup, search space, and objectives
- `main.py` and reads the configuration, and implements the MOBO
- `bo.py` executes the user-defined evaluation for each sampled configuration.

This structure enables a clear separation between the optimization engine and the problem-specific evaluation logic, making it easy to adapt Spearmint to different design tasks.

4.3 Constrained example (from github)

This section summarizes how the "constrained example" available on GitHub[2] is constructed and how to make use of its features. In this example, a function is analyzed in order to identify its minimum points—a task that could also be

[1] `https://github.com/EduardoGarrido90/spearmint_ppesmoc`

[2] `https://github.com/HIPS/Spearmint/tree/master/examples/constrained`

accomplished using basic mathematical tools such as derivatives. However, since the example involves Multi-Objective Bayesian Optimization (MOBO), the function has multiple input variables and, most importantly, is unknown to Spearmint. As a result, Spearmint must identify the *local minima* of the unknown objective function without relying on grid search methods. Instead, it leverages probabilistic models and gradient-based optimization techniques to efficiently explore the search space and converge to optimal solutions.

$$\text{obj} = \left(y - \left(\frac{5.1}{4\pi^2}\right)x^2 + \left(\frac{5}{\pi}\right)x - 6\right)^2 + 10\left(1 - \frac{1}{8\pi}\right)\cos(x) + 10 \qquad (4.1)$$

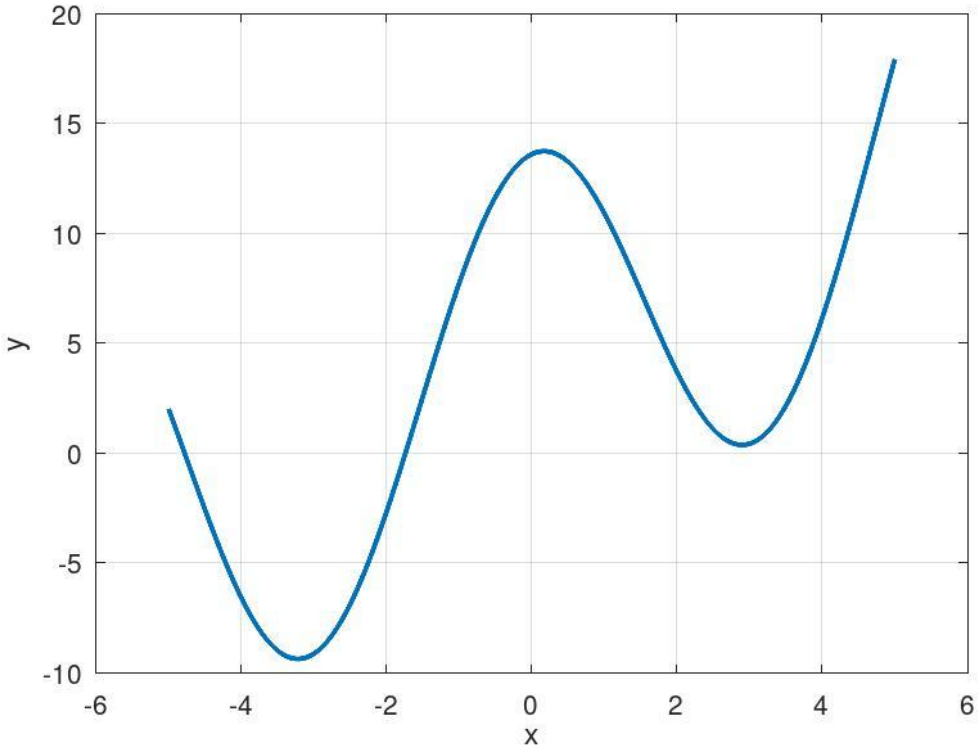

Figure 4.1: Example function with constraints

The objective of the provided example is to find the minimum of a complex function (see Eq. 4.1 and Fig. 4.1), subject to the constraints $y \geq x$ and $y \leq 10$. The use of the constraint fields is particularly useful when designing complex accelerators, as constraints can represent specific design requirements—such as enforcing minimum latency or limiting power and area

utilization. Later in the optimization process, constraints will also help manage infeasible solutions, enabling Spearmint to effectively learn how to filter them out.

4.3.1 BO.py

`BO.py` is the file invoked by Spearmint's main script `main.py` during the evaluation process. It receives the suggested parameters and returns the corresponding results. In this example 4.1, the file is written in Python, but it can also be implemented in other languages such as MATLAB, Bash, and other languages.

```
import math
import numpy as np

def evaluate(job_id, params):

    x = params['X']
    y = params['Y']

    print 'Evaluating at (%f, %f)' % (x, y)

    if x < 0 or x > 5.0 or y > 5.0:
        return np.nan
    # Feasible region: x in [0,5] and y in [0,5]

    obj = float(np.square(y - (5.1/(4*np.square(math.pi)))*np.square(x) + (5/
        math.pi)*x- 6) + 10*(1-(1./(8*math.pi)))*np.cos(x) + 10)

    con1 = float(y-x)   # y >= x

    con2 = float(10.0-y)  # y <= 10

    return {
        "branin"       : obj,
        "y_at_least_x" : con1,
        "y_at_most_10" : con2
    }

    # True minimum is at 2.945, 2.945, with a value of 0.8447

def main(job_id, params):
    try:
        return evaluate(job_id, params)
    except Exception as ex:
        print ex
        print 'An error occurred in branin_con.py'
        return np.nan
```

Listing 4.1: `BO_example.py`

The key components of the script are:

- **main** function: Called by Spearmint, receives two arguments: `job_id` and `params`.
  - `job_id`: A numeric identifier representing the current iteration.
  - `params`: A dictionary containing the parameters suggested by Spearmint based on the configuration.
- A `try-except` block: Used for graceful error handling during evaluation.
- A `return` statement: Returns a dictionary containing the evaluation results.
  - `objective(s)`: output value that has to be minimized (leading to zero).
  - `constraint(s)`: a numeric value that has to be positive defined

4.3.2 config.json

This configuration file 4.2 defines the essential elements for Spearmint optimization. It is present inside the spearmint working folder. It is read by the main spearmint script in order to identify the properties of the research. It has many elements inside it which are explained below. The information was obtained by analyzing many source codes inside the Spearmint GitHub folder[3].

```
{
    "language"        : "PYTHON",
    "main-file"       : "branin_con",
    "experiment-name" : "branin-constrained-example",
    "variables": {
        "X": {
            "type": "FLOAT",
            "size": 1,
            "min":  -5,
            "max":  10
        },
        "Y": {
            "type": "FLOAT",
            "size": 1,
```

[3]Useful links :
link to github file managing search space and tasks via config.json
`https://github.com/HIPS/Spearmint/tree/master/spearmint/tasks/base_task.py`
Link useful to look at other spearmint examples
`https://github.com/HIPS/Spearmint/tree/master/examples`

```
            "min":  0,
            "max":  15
        }
    },
    "tasks": {
      "branin" : {
          "type"         : "OBJECTIVE",
          "likelihood"   : "NOISELESS"
      },
      "y_at_least_x" : {
          "type"         : "CONSTRAINT",
          "likelihood"   : "NOISELESS"
      },
      "y_at_most_10" : {
          "type"         : "CONSTRAINT",
          "likelihood"   : "NOISELESS"
      }
    },
    "polling-time" : 1
}
```

Listing 4.2: `conf_bo_example.json`

- **language** and **main-file**: Specify the BO.py .
- **variables** `X` and `Y`: Define the search space.
  - type: specifies the type of variable, it can be *float*, *int* or *enum*.
  - min: the minimum value, that can be generated
  - max: the maximum value, that can be generated
  - size: specifies the amount of values inserted in a vector. If it is '1', it generates a single value.
  - `options`: used only in enum case, it specifies the values, that can be used.
- **tasks: OBJECTIVE**: The metric that Spearmint attempts to minimize.
- **tasks: CONSTRAINT**: Conditions that must be satisfied (i.e., be ≥ 0) for a solution to be considered valid.

These settings clarify the role of `con1` and `con2` in the BO script, enforcing $Y \geq X$ and $Y \leq 10$, respectively. From visual inspection, the global minimum is found around $X = 2.945$, $Y = 2.945$, with obj ≈ 0.8447, which matches the Spearmint output.

4.4 Spearmint implementation file: CONV

This section discusses how *BO.py* and *config.json* were written for the convolution accelerator.

4.4.1 BO.py

```
import math
import numpy as np
import json
import os
import time
import random
import correction as crr
from datetime import datetime
import dbcache as db

timeout = 60*60*24*5   #60sec for 60 min for 24h for 5 days
base_dict = "/home/ilacqua/spearmint/examples/conv/conv2d.json"
req_location = "/home/ilacqua/tool_chain/request/"
resp_location = "/home/ilacqua/tool_chain/response/"
flag_location = "/home/ilacqua/tool_chain/flags/"
flag_full_path = "/home/ilacqua/tool_chain/flags/conv_done.flag"

db_bck = "/home/ilacqua/tool_chain/results/"
debug = False
infinit = 1e10

def set_flag():
    def create_empty_file_if_not_exists(file_name):
        if not os.path.exists(file_name):
            with open(file_name, 'w') as file:
                pass
    create_empty_file_if_not_exists( flag_full_path )

def read_json(file_path):
    json_data = []
    with open(file_path, 'r') as json_file:
        json_data = json.load(json_file)
    return json_data

def save_json(json_data, file_name):
    if debug :
        print("debug, save req skipped")
        return
    with open(file_name, 'w') as json_file:
        json.dump(json_data, json_file)

def wait_and_read_json(file_path, timeout_):
    if debug:
        print("debug, read req skipped")
        time.sleep(5)
        debug_response = read_json(base_dict)
        return debug_response
    start_time = time.time()
```

```
49      while not os.path.exists(file_path):
50          if time.time() - start_time > timeout_:
51              return -1
52          time.sleep(1)  # Wait for 1 second before checking again
53      with open(file_path, 'r') as f:
54          json_data = json.load(f)
55      return json_data
56
57  def evaluate(job_id, params, iteration):
58      global Response
59
60      job_id_ = job_id
61      Response = {}
62      print "generating new query"
63      # unpacking
64      print params
65      # generating from input vector
66      Freq = int(params['FREQ']) * 200
67      Outch = int(params['OUTCH']) * 4
68      Inch = 32
69
70      # packing
71      print base_dict
72      Request = read_json(base_dict)
73      print Request
74
75      Request["main"]["FREQ"] = Freq
76      Request["catapult"]["CatapultFreq"] = Freq
77      Request["main"]["INCH"] = Inch
78      Request["main"]["OUTCH"] = Outch
79
80      def adj( value ):
81          if value >= 2:
82              return value+1
83          else:
84              return value
85
86      Request["catapult"]["lp_init"] = int(adj(params['lp_init']))
87      Request["catapult"]["lp_ci"] = int(params['lp_ci'])
88      Request["catapult"]["lp_k_handw"] = int(params['lp_k_handw'])
89      Request["catapult"]["lp_co"] = int(adj(params['lp_co']))
90      Request["catapult"]["lp_wb"] = int(adj(params['lp_wb']))
91      Request["catapult"]["lp_q"] = int(adj(params['lp_q']))
92
93      Request["tool_chain"]["status"] = "catapult_ready"
94      Request["tool_chain"]["spearmint_id"] = job_id_
95      Request["tool_chain"]["spearmint_id_number"] = iteration
96
97      Request["tool_chain"]["status"] = "catapult_ready"
98      print "final request"
99      print Request
100
101     save_json(Request, "{0}Req_{1}.json".format(req_location, job_id_))
102     print "evaluating {0}".format(params)
103     Response = wait_and_read_json("{0}response_Req_{1}.json".format(
            resp_location, job_id_), timeout)
104
105     err_code=2
```

```
106     status = Response["tool_chain"]["status"]
107
108     print Response
109
110     info = Response.get("tool_chain", {}).get("notes", "good")
111
112     if 'error design' in info:
113         print("error_tool_chain")
114         err_code=-1
115         return { "Area": 0, "Power": 0, "Latency": 0, "error": err_code }
116
117     print("reading data from response")
118
119     area_innovus = Response.get("innovus_report", {}).get("Area", 0)
120     area = crr.area_correction_sram( Outch, area_innovus * (10**-6) )
121     print( "base area : ",area_innovus, " adj area : ", area  )
122
123     power = Response.get("innovus_report", {}).get("power", {}).get("total",
            0)
124     print( "adj power : ", power )
125
126     lat0= int (Response.get("catapult_report", {}).get("main", {}).get("
            latency", {}).get("en_quant_tb_0", 0))
127     lat1= int (Response.get("catapult_report", {}).get("main", {}).get("
            latency", {}).get("en_quant_tb_1", 0))
128     latency = crr.latency_correction( Inch, Outch, lat0, lat1)
129     print("lat0, lat1 : ", lat0, lat1, "\n","adj latency [ns] : ", latency  )
130
131     results_={}
132     results_["iteration"]=iteration
133     results_["resp_dict"]=Response
134     results_["Area"] = area
135     results_["Power"] = power
136     results_["Latency"] = latency
137     results_["error"] = err_code
138
139     json_location_ = "{}/{}.json".format(db_bck, job_id_)
140     save_json( results_, json_location_ )
141     set_flag()
142
143     for el in [ area, power, latency]:
144         if el==None:
145             raise ValueError("None found watch log to find issue")
146
147     return { "Area": area, "Power": power, "Latency": latency, "error":
            err_code }
148
149 def main(job_id, params):
150     print( "\n----job_id----\n",job_id,"\n---params---\n", params, "\n")
151     if job_id == 1:
152         working_directory = os.getcwd()
153         current_time = datetime.now().strftime("%Y-%m-%d-%H-%M-%S")
154         data = {
155             "working_directory": working_directory,
156             "start_time": current_time
157             }
158         save_json( data, "data.json")
159         start_time = current_time
```

```
160     else :
161         ttt = 0.3 * int( job_id )
162         if job_id > 10 :
163             ttt = 1
164         time.sleep( ttt )
165         env_data = read_json( "data.json" )
166         start_time = env_data["start_time"]
167 
168     try:
169         job_id_name = "conv_{0}x{1}".format(start_time, job_id)
170         evaluated = evaluate(job_id_name, params, job_id)
171         print(evaluated)
172         return evaluated
173     except Exception as ex:
174         print ex
175         print 'An error occurred'
176         #return np.nan
177 
178 def test():
179     job_id_ = "giacomino_22"
180     params_ = { "FREQ" : 1.1, "INCH" : 4, "OUTCH" : 4,
181                 'lp_init': 4, 'lp_ci': 4, 'lp_k_handw': 4, 'lp_co': 4, 'lp_wb
                    ': 4, 'lp_q': 4 }
182 
183     try:
184         return evaluate(job_id_, params_)
185     except Exception as ex:
186         print ex
187         print 'An error occurred'
188 
189 if __name__ == '__main__':
190     test()
```

Listing 4.3: `conf.json`

As previously stated, the purpose of this script is to process the arguments passed through the *params* dictionary, which is generated by the main Spearmint script, and to construct a corresponding request dictionary to be saved in JSON format. BO.py waits for a response json-formatted, which is generated by another script (`sync_server.py` discusses in chapter 5).

Since the latency and area values do not account for additional hardware overhead beyond the accelerator itself, these values must be adjusted using a correction function. After that, the value of Area, power, latency can be returned.

BO.py: Key Steps

By analyzing the execution flow of the script, it is possible to identify its key features. The *main* function serves as the entry point. During the first iteration, it stores the current date (inside a JSON file) to differentiate new requests from previous ones. In subsequent iterations, this date is read

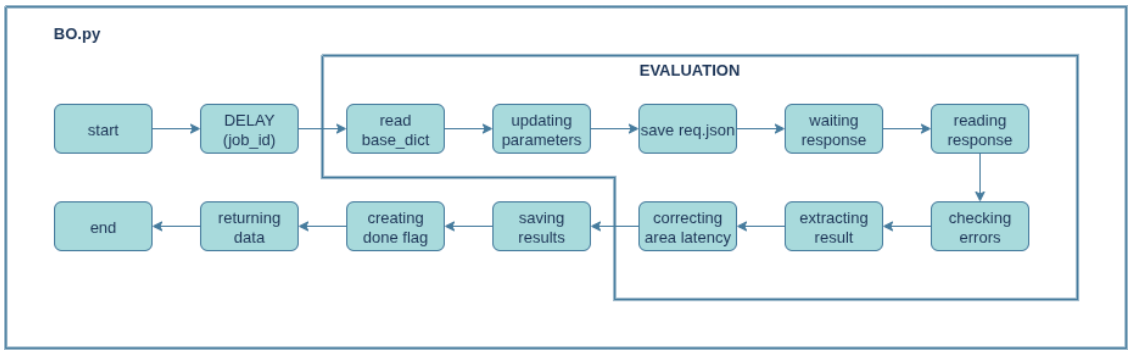

Figure 4.2: BO key steps

from the file, rather than using the system's current time. This approach ensures consistency, since only the first iteration executes the full Bayesian Optimization (BO) pipeline when multiple instances (often called *jobs* inside documentation) are launched simultaneously. To support this behavior, a slight delay is introduced to enforce sequential request file generation and prevent potential race conditions.

An important feature of the script is its internal error management, which is crucial to prevent crashes in the main Spearmint process. This is achieved by wrapping the call to the evaluation function in a try-except block within the *main* function. The evaluation function may occasionally fail, due to missing files or other unforeseen errors. Within the evaluation function, a base dictionary in JSON format is read. This dictionary contains a default set of parameters and is especially useful when the search space is altered or reduced due to further toolchain developments. These base parameters are then adjusted to match the constraints of the toolchain, which not support direct remapping. Direct mapping is infeasible because Spearmint internally works with abstract numerical representations (e.g., integers or floats), rather than with specific tool-compatible values. Although this may seem like a limitation, it can be advantageous. For instance, if frequency values are sorted with a positive trend, the tool can better leverage this ordering in the optimization process.

Once the refined request dictionary is ready, it is saved as a JSON file in the designated request directory. The script then waits for the corresponding response, using a dedicated timeout function that can terminate the iteration if the response is not received within a reasonable time window. This mechanism is particularly useful when the toolchain is required to explore complex or resource-intensive configurations.

The script includes an error-handling mechanism to manage Spearmint's limited set of expected return types: (1) success, (2) constraint violation, and (3) failure. The "constraint violation" return type is especially useful for signaling infeasible configurations (e.g., failures at the HLS level), thus helping the tool identifying possible bad configuration properties.

After receiving and reading the response, a correction function is invoked to adjust the latency and area values, which initially omit extra-hardware components outside the accelerator. If all steps are completed successfully, a combined JSON file containing the response and Spearmint-compatible return values is saved in a dedicated folder. This design choice allows for independent storage and analysis, bypassing the need to interface with Spearmint's default MongoDB backend, which adds unnecessary complexity and uses an undocumented data structure.

Finally, BO is invoked in a way that prints in a log file, rather than printing it directly to the terminal. This log file is placed in a folder named with the current date, located in the working directory, together with the `config.json`. Debugging this log can be challenging—especially when an iteration fails and enters in a loop of failing iterations. For effective troubleshooting, it is recommended to run the BO script in standalone mode. For this purpose, a test function is included and can be invoked when the script is executed independently. This is particularly useful given the latency between the main Spearmint invocation and the actual start of the BO process.

Frequent print statements have also been included throughout the code to monitor the execution flow, which can be particularly helpful during debugging. Also when an iteration is completed, a placeholder file is generated to notify other scripts that an iteration is completed and a new one will be launched soon.

4.4.2 CONFIG.JSON

The following JSON configuration is used to set up and control a multi-objective optimization experiment in Spearmint. The file defines the optimization algorithm. The variables search space will be discussed in experimental Chapter 7.1. The objectives choices are Area, Power, Latency, error, like already discussed.

```
{
  "language": "PYTHON",
  "random_seed": 1,
  "main_file": "conv",
  "grid_size": 10000,
  "max_finished_jobs": 10000,
```

```
  "max_concurrent": 7,
  "experiment-name": "branin_constraint_mulitobjective",
  "moo_use_grid_only_to_solve_problem": true,
  "moo_grid_size_to_solve_problem": 1000,
  "pesm_use_grid_only_to_solve_problem": true,
  "likelihood": "GAUSSIAN",
  "acquisition": "PESMC",
  "pesm_pareto_set_size": 5000,
  "database": {
    "address": "127.0.0.1:27012"
  },
  "pesm_not_constrain_predictions": false,
  "variables": {
    "FREQ": {
      "type": "INT",
      "size": 1,
      "min": 1,
      "max": 8
    },
    "OUTCH": {
      "type": "INT",
      "size": 1,
      "min": 1,
      "max": 8
    },
    "lp_init": {
      "type": "INT",
      "size": 1,
      "min": 0,
      "max": 3
    },
    "lp_ci": {
      "type": "INT",
      "size": 1,
      "min": 2,
      "max": 4
    },
    "lp_k_handw": {
      "type": "INT",
      "size": 1,
      "min": 2,
      "max": 4
    },
    "lp_co": {
      "type": "INT",
      "size": 1,
      "min": 0,
      "max": 3
    },
    "lp_wb": {
      "type": "INT",
      "size": 1,
      "min": 0,
      "max": 3
    },
    "lp_q": {
      "type": "INT",
      "size": 1,
```

```
      "min": 0,
      "max": 3
    }
  },
  "tasks": {
    "Area": {
      "type": "OBJECTIVE"
    },
    "Latency": {
      "type": "OBJECTIVE"
    },
    "Power": {
      "type": "OBJECTIVE"
    },
    "error": {
      "type": "CONSTRAINT"
    }
  }
}
```

Listing 4.4: `conf.json`

Used parameters

- **language**: Specifies the language of the evaluation script, here set to `PYTHON`.
- **main_file**: Entry point for evaluations, when spearmint is executed.
- **max_concurrent**: Maximum number of parallel evaluations.
- **variables**: The design space is defined through a set of integer variables, which will be described in experimental Chapter 7:
  - `FREQ`, `OUTCH`, `lp_init`, `lp_ci`, `lp_k_handw`, `lp_co`, `lp_wb`, `lp_q`. Each with defined `min` and `max` values.
- **tasks**: Defines the optimization targets:
  - `Area`, `Latency`, `Power`: These are treated as objectives to minimize.
  - `error`: A hard constraint that must be satisfied for a design to be valid.

Tricks used

- **Scaling**: Since variables such as *Freq* and *OutCh* have linearly spaced values, they are initially mapped to integer values ranging from 1 to N,

where N is the total number of distinct values in the search space. These integer representations are passed to *BO.py*, where they are subsequently scaled by a factor 200 for *Freq*, and 4 for *OutCh*. This scaling choice will be further discussed in the dedicated 7.1.

- **Pointers**: Since Spearmint performs better with numerical inputs, loop configurations are encoded by associating them with corresponding numeric values.
- **Monotonic mapping**: In order to make the design parameters compatible with Spearmint, a numerical mapping is defined between integer values and specific loop configurations (e.g., loop unrolling factors). This enables Spearmint to operate within a numerical search space while still evaluating meaningful architectural alternatives. Although Spearmint treats all input values purely as numeric and does not infer any semantic relationship, the mapping is constructed with a monotonic trend—such that increasing numerical values correspond to progressively more compact loop configurations. This structure may facilitate the optimizer's exploration by introducing a degree of smoothness in the search space, although such behavior is not formally guaranteed.
- **Adjustment functions**: Since certain loop configurations may be missing or undefined, a dedicated function is used to remap and complete these cases.
- **Parallel jobs**: Due to the high execution time of each evaluation and the fact that hardware resources are not fully utilized by a single iteration, multiple evaluations are executed in parallel. This parallelism does not imply that all evaluations are launched simultaneously—except possibly at the beginning—but rather that new iterations are triggered dynamically as previous ones complete. On average, Spearmint is able to generate a new configuration approximately every 20 minutes (as experienced during testing). If the number of concurrent evaluations has not yet reached the predefined maximum, a new iteration is launched as soon as one finishes. Otherwise, the system waits until at least one job has completed before proceeding. Each new suggestion is generated based on the results of all previously completed iterations, ensuring that the optimization process remains consistent and informed.

4.5 Spearmint implementation: FC

The same workflow has been applied to the **FC** (fully connected) case, with a few adjustments compared to the CONV configuration. These modifications do not affect the overall structure or logic of the Bayesian Optimization script, which remains unchanged in its core components.

4.5.1 config.json

```
{
  "language": "PYTHON",
  "random_seed": 1,
  "main_file": "fcmei",
  "grid_size": 1000,
  "max_finished_jobs": 9000,
  "max_concurrent": 10,
  "experiment-name": "branin_constraint_mulitobjective",
  "moo_use_grid_only_to_solve_problem": true,
  "moo_grid_size_to_solve_problem": 1000,
  "pesm_use_grid_only_to_solve_problem": true,
  "likelihood": "GAUSSIAN",
  "acquisition": "PESMC",
  "pesm_pareto_set_size": 5000,
  "database": {
    "address": "127.0.0.1:27012"
  },
  "pesm_not_constrain_predictions": false,
  "variables": {
    "FREQ": {
      "type": "INT",
      "size": 1,
      "min": 1,
      "max": 8
    },
    "OUTCH": {
      "type": "INT",
      "size": 1,
      "min": 1,
      "max": 8
    },
    "lp_init": {
      "type": "INT",
      "size": 1,
      "min": 0,
      "max": 3
    },
    "lp_c": {
      "type": "INT",
      "size": 1,
      "min": 2,
      "max": 4
    },
    "lp_k": {
      "type": "INT",
```

```
      "size": 1,
      "min": 2,
      "max": 4
    },
    "lp_wb": {
      "type": "INT",
      "size": 1,
      "min": 0,
      "max": 3
    },
    "lp_q": {
      "type": "INT",
      "size": 1,
      "min": 0,
      "max": 3
    }
  },
  "tasks": {
    "Area": {
      "type": "OBJECTIVE"
    },
    "Latency": {
      "type": "OBJECTIVE"
    },
    "Power": {
      "type": "OBJECTIVE"
    },
    "error": {
      "type": "CONSTRAINT"
    }
  }
}
```

Listing 4.5: `conf.json`

A slight modification must be applied to the *config.json* file in order to make it compatible with the **FC** case. The updated configuration, shown in Listing 4.5, reflects the structure of the search space defined for **FC** in Section 6.1. These changes are necessary to ensure proper alignment between the configuration file and the specific parameters used in the FC scenario.

Chapter 5

SYNC SERVER

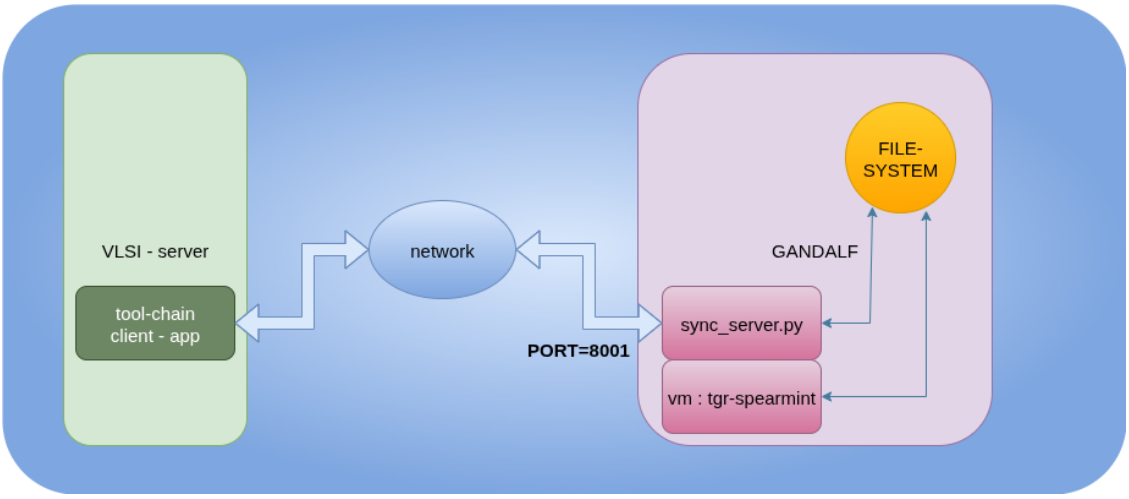

Figure 5.1: Servers configuration

At this stage of the project, with a developed framework for computer-aided design (CAD) automation and a Spearmint setup capable of generating requests and retrieving results from database files. However, another issue remains: Spearmint is running inside a virtual machine (VM) on the Gandalf server, while the CAD tools are running on the VLSI server.

Managing multiple applications distributed on several servers presents challenges in maintaining reliable communication between the different servers. In the context of this application, a central server is required to coordinate and exchange data with the other components in the system. After evaluating the available options, `gandalf` was selected as the main server,

primarily because it already exposes a public URL and open web ports. Although this may seem like a minor detail, it enables external devices to initiate a connection over the internet, which is a fundamental requirement for server reachability.

5.1 server script

To allow `gandalf` to exchange data with the `VLSI` or other servers, a dedicated application was developed using Python and Flask. This Flask-based server exposes a set of RESTful endpoints[1] (routes) designed to facilitate interaction, resolve communication issues, and manage data transmission.

Flask is a micro web framework written in Python, designed for simplicity and modularity. It provides essential features such as URL routing, HTTP request handling, and JSON serialization. Its lightweight nature makes it particularly well-suited for building REST APIs and applications where fine-grained control over behavior is required.

One particular communication challenge involves exchanging JSON-formatted requests and responses with the `VLSI_server`. In a standard client-server architecture, the client is responsible for initiating the connection. Therefore, while `gandalf` functions as a server, it cannot autonomously push data to other servers. To address this, the main script running on the `VLSI_server` is configured to periodically poll the Flask server every few seconds. This inversion of control ensures that data flows correctly despite architectural constraints.

Since a Flask server is already running on the `gandalf` machine, a custom route has been added to expose a downloadable data package containing the results of all completed Spearmint iterations. This package is used by external tools to generate plots and perform data analysis. The ability to access this data remotely is particularly useful when visualizations or reports need to be generated on machines that are not part of the same local network. In the future, this server could be extended with additional endpoints to start or stop optimization jobs remotely, further enhancing the usability and flexibility of the overall system.

[1] A RESTful API (Representational State Transfer) adheres to a stateless, client-server communication model where resources are accessed via standard HTTP methods such as GET, POST, PUT, and DELETE. RESTful design emphasizes scalability, simplicity, and the use of uniform resource identifiers (URIs) to interact with structured data.

5.2 Server-side Implementation with Flask

The Python script reported in listing 5.1, implements a simple RESTful API using the Flask framework. It facilitates communication between a client and a server through the exchange of JSON files placed in specific directories. The core functionalities exposed by the server are summarized below:

- **`/list_requests`**: Returns a list of all request files found in the `REQUEST_DIR` directory. Only files prefixed with `Req_` are considered.
- **`/list_pending_request`**: Lists all request files that do not yet have a corresponding response file (prefixed with `response_`) in the `RESPONSE_DIR`.
- **`/get_request`**: Given the name of a request file (via the `req_name` parameter), it retrieves the content and validates whether the file contains valid JSON.
- **`/push_response`**: Accepts a JSON response and writes it to disk in the `RESPONSE_DIR` using the specified name (`resp_name`).
- **`/get_all_data`**: Invokes the `main_slave` function from the `data_merger` module, processes the data, and returns the resulting file to the client. basically, given the accelerator name, it return the merge of all responses on a `JSON`.

The script also includes two utility functions:

- **`is_valid_json`**: Validates whether a file contains properly formatted JSON, retrying up to three times in case of read or parse errors.
- **`write_json`**: Writes a Python dictionary to a file in JSON format with indentation. Returns a boolean indicating success.

Logging is used extensively to track server activity and potential issues. Depending on the debug flag, the server can run either in production mode (with absolute paths) or in debug mode (with local paths). The application listens for HTTP requests on port `8001`, because default is already in use.

```
1 from flask import Flask, request, jsonify, send_file
2 import os
3 import time
4 import json
5 import logging
6
```

```
import data_merger as dm

########################
## DECLARING CONSTANTS##
########################
debug = False

if not debug:
    REQUEST_DIR = "/home/ilacqua/tool_chain/request"
    RESPONSE_DIR = "/home/ilacqua/tool_chain/response"
    SYNC_DIR = "/home/ilacqua/tool_chain/sync"
    DONE_SYNC_DIR = "/home/ilacqua/tool_chain/done_sync"
    PORT = 8001
else:
    REQUEST_DIR = "request"
    RESPONSE_DIR = "response"
    SYNC_DIR = "sync"
    DONE_SYNC_DIR = "done_sync"
    PORT = 8001

STR_FILTER_RESPONSE = "response_"
STR_FILTER_REQUEST = "Req_"

dir_list= [ REQUEST_DIR, RESPONSE_DIR]
for directory in dir_list:
    if not os.path.isdir(directory):
        os.mkdir(directory)

# Setup logging
logging.basicConfig(level=logging.DEBUG if debug else logging.INFO,
                    format='%(asctime)s - %(levelname)s - %(message)s')

# init flask
app = Flask(__name__)

@app.route('/list_requests', methods=['POST'])
def receive_data():
    try:
        list_files = os.listdir(REQUEST_DIR)
        list_req = [item for item in list_files if item.startswith(STR_FILTER_REQUEST)]
        response_data = {"request_list": list_req}
        logging.info(f"List of requests successfully retrieved. Found : {len(list_req)}")
        return jsonify(response_data), 200
    except Exception as e:
        logging.error(f"Error retrieving list of requests: {e}")
        return jsonify({"error": "Internal server error"}), 500

@app.route('/list_pending_request', methods=['POST'])
def list_pending_request_():
    try:
        list_files = os.listdir(REQUEST_DIR)
        list_pending = []
        for item in list_files:
            if item.startswith(STR_FILTER_REQUEST):
                resp_name = STR_FILTER_RESPONSE + item
                if not os.path.exists(os.path.join(RESPONSE_DIR, resp_name)):
                    list_pending.append(item)
        response_data = {"list_pending": list_pending}
        logging.info(f"List of pending requests successfully retrieved. Pending:{list_pending}")
        return jsonify(response_data), 200
    except Exception as e:
        logging.error(f"Error retrieving list of pending requests: {e}")
        return jsonify({"error": "Internal server error"}), 500

@app.route('/get_request', methods=['POST'])
def get_request():
    try:
        data = request.json
        req = data.get("req_name")
        if not req:
            logging.warning("Missing 'req_name' in request.")
            return jsonify({"error": "Missing required parameter 'req_name'"}), 400

        req_path = os.path.join(REQUEST_DIR, req)
        valid, content = is_valid_json(req_path)

        response = {"valid": valid, "data": content}
```

```
84          logging.info(f"Request '{req}' successfully retrieved with valid JSON: {valid}.")
85          return jsonify(response), 200
86      except Exception as e:
87          logging.error(f"Error retrieving request '{data.get('req_name', 'unknown')}': {e}")
88          return jsonify({"error": "Internal server error"}), 500
89
90  @app.route('/push_response', methods=['POST'])
91  def push_response():
92      try:
93          data = request.json
94          resp_name = data.get("resp_name")
95          resp_data = data.get("resp_data")
96
97          if not resp_name or resp_data is None:
98              logging.warning("Missing 'resp_name' or 'resp_data' in push_response request.")
99              return jsonify({"error": "Missing required parameters 'resp_name' or 'resp_data'"}), 400
100
101         resp_path = os.path.join(RESPONSE_DIR, resp_name)
102         result = write_json(resp_data, resp_path)
103
104         if result:
105             logging.info(f"Response '{resp_name}' successfully written.")
106             return jsonify({"result": "success"}), 200
107         else:
108             logging.error(f"Failed to write response '{resp_name}'.")
109             return jsonify({"error": "Failed to write JSON data"}), 500
110     except Exception as e:
111         logging.error(f"Error pushing response '{data.get('resp_name', 'unknown')}': {e}")
112         return jsonify({"error": "Internal server error"}), 500
113
114 @app.route('/get_all_data', methods=['POST'])
115 def get_all_data():
116     try:
117         data = request.json
118         #here
119         result= "ok"
120         logging.info(result)
121         file_path = dm.main_slave(force=data.get("force", None), acc_name=data.get("acc_name", "any")
                )
122         return send_file(file_path, as_attachment=True)
123     except Exception as e:
124         logging.error(f"Error: {e}")
125         return jsonify({"error": "Internal server error"}), 500
126
127 def is_valid_json(file_path, retries=3, delay=1):
128     """Check if the file content is a valid JSON, with up to 3 retries.
129     Returns a tuple: (is_valid, content)."""
130     attempt = 0
131     while attempt < retries:
132         try:
133             with open(file_path, 'r') as f:
134                 content = json.load(f)
135             logging.debug(f"Valid JSON found in file '{file_path}' on attempt {attempt + 1}.")
136             return True, content
137         except (ValueError, json.JSONDecodeError) as e:
138             logging.warning(f"Attempt {attempt + 1}: Invalid JSON in file '{file_path}': {e}")
139             attempt += 1
140             time.sleep(delay)
141     logging.error(f"File '{file_path}' is not a valid JSON after {retries} attempts.")
142     return False, None
143
144 def write_json(data, filename):
145     try:
146         with open(filename, 'w') as file:
147             json.dump(data, file, indent=4)
148         logging.debug(f"Data successfully written to '{filename}'.")
149         return True
150     except (IOError, TypeError) as e:
151         logging.error(f"An error occurred while writing to the file '{filename}': {e}")
152         return False
153
154 if __name__ == '__main__':
155     logging.info(f"Starting Flask server on port {PORT}.")
156     app.run(host='0.0.0.0', port=PORT)
```

Listing 5.1: sync_server.py

Future works:

- **Security** has not been a major concern in the current implementation, as the application is not publicly advertised, and neither the address nor the port used is shared beyond the internal development environment. In addition,on, the service is not publicly accessible or indexed. Although this provides basic obscurity, in a production setup, it would be necessary to introduce stronger mechanisms such as authentication tokens, HTTPS encryption, and possibly IP whitelisting.

- **Scalabilty:** Currently, the system communicates with only one client node. However, the architecture allows for easy extension: multiple client could be introduced in the future with appropriate request logic. This would allow better load distribution across available clients and improve parallelism during large-scale experiments.

- **Asynchronous model:** The current implementation relies on an asynchronous polling mechanism in which the client periodically checks for new tasks. Although simple and robust, this approach can introduce unnecessary latency and load. Future improvements could involve keeping the connection open using techniques like long polling or server-sent events. This would enable the server to notify the client when new data become available, significantly reducing polling overhead and response time. However, such a solution would require a direct, NAT-free connection between the communicating machines.[2]

- **Logging and monitoring:** Due to the simplicity of the project in its current stage, extensive logging has not been implemented. Only basic logs are collected to trace major events and errors. For future debugging and troubleshooting purposes, a more detailed and structured logging system could be introduced. This might include request tracking, event timestamps, and separate debug levels to support performance monitoring and diagnostics.

- **web interface and Automation:** A web interface could be added in

[2]A NAT-free connection implies that both client and server can communicate directly, without passing through Network Address Translation (NAT). NAT, commonly used in routers (also ISPs), prevents devices without a public IP address from easily receiving inbound connections

the future to further simplify user interaction and enable visual monitoring of the system status. In particular, such a UI could help automate and manage Spearmint-based experiments from a graphical dashboard.

5.3 Launch and automation

This section summarizes, how the whole tool-chain is invoked and correctly executed:

1 - tool-chain: in vlsi server

The sequence of commands outlined above performs a systematic reset and execution of the primary Python script within a controlled environment. Initially, the working directory is set to the project folder, ensuring that all subsequent operations are conducted within the correct context. Any pre-existing `tmux` sessions are terminated, eliminating potential conflicts or residual processes from previous runs.

Next, the procedure clears all relevant data structures by overwriting the JSON database files associated with requests and responses, effectively initializing them to empty states. The corresponding directories storing temporary request and response files are also purged, thereby preventing the contamination of new runs with residual files.

Following this cleanup, the main Python script, `tool_fsm_2.py`, is launched within a dedicated `tmux` session. This setup allows the script's output to be logged in real time while simultaneously being saved to a file for post-execution review. Finally, the user attaches to the session, enabling direct monitoring and interaction with the script's execution.

```
# inside vlsi_wall
cd /home/thesis/francesco.ilacqua/2_table_man/
tmux kill-server
echo "[]" > ./scripts/tables/database.json

rm -r scripts/tables/request/*
rm -r scripts/tables/response/*

echo "[]" > /home/thesis/francesco.ilacqua/2_table_man/
   scripts/tables/uploaded_response_list.json

tmux new-session -d -s tool_chain 'python3 scripts/tool/
   tool_fsm_2.py | tee fsm.log; read -p "don t forget to say
   hello"'
```

```
12
13 tmux attach -t tool_chain
```

Listing 5.2: tool chain launch commands

2 - sync server: in gangalf

For the `sync_server` script, the procedure is straightforward. The script itself encapsulates all necessary management and execution logic, so the user only needs to invoke an aliased command to launch the server.

```
1 # run flask server in gandalf server
2 alias run_my_sync="tmux new-session -d -s sync_server '
    sync_server.py '"
3 alias start_my_sync=" cd ~/tool_chain; run_my_sync"
4 echo '"start_my_sync" for launching flask server'
```

Listing 5.3: sync server launch commands

3 - spearmint: runtime

The workflow on the spearmint-server side is slightly more complex due to the dependency on a running MongoDB server for Spearmint. To streamline operations, several aliases have been introduced to manage the database lifecycle, including starting, removing, and terminating the MongoDB instance.

A dedicated function is provided to execute a clean Spearmint session (e.g., for the `conv` example). This function prompts the user for confirmation via `are_u_sure` to ensure that they are aware that all previous progress will be lost. Upon confirmation, the current MongoDB instance is cleaned, all temporary folders are removed, and a fresh run of the Spearmint main script is executed.

Additionally, another function is available to resume an interrupted execution, which may occur due to unexpected errors or system interruptions. This design ensures that both full and partial runs of Spearmint can be managed in a controlled and reproducible manner.

```
1
2 alias stc_mongo_remove="source scripts_new/mongod_remove_db.sh"          # remove files
3 alias stc_mongo_start="source /home/ilacqua/scripts_new/mongod_start.sh" # starts mongo
4 alias stc_mongo_terminate="source scripts_new/mongod_terminate.sh"       # terminates mongo
5
6 are_u_sure(){ read -p "${1:-Are you sure you want to proceed? (y/n): }" -r response && [[ $response
    =~ ^[Yy]$ ]]; }
7
8 stc_retry_run_conv(){
9     echo "Restarting spearmint search..."
```

```
10      are_u_sure "This will erase all progress. Are you sure? (y/n): " || { echo "Operation canceled.";
            return 1; }
11
12      cd ~/spearmint
13      python cleanup.py examples/conv/
14
15      rm -rv ~/tool_chain/{request/*conv*,response/*conv*}
16      rm -rv ~/spearmint/examples/conv/output_*
17
18      python main.py ./examples/conv
19  }
20
21  stc_start_conv(){
22      cd ~/spearmint
23      pwd
24      tmux new-session -d -s backup-tmux 'python main.py ./examples/conv'
25      python main.py ./examples/conv
26  }
```

Listing 5.4: spearmint launch commands

Chapter 6

FC: EXPERIMENTAL RESULTS

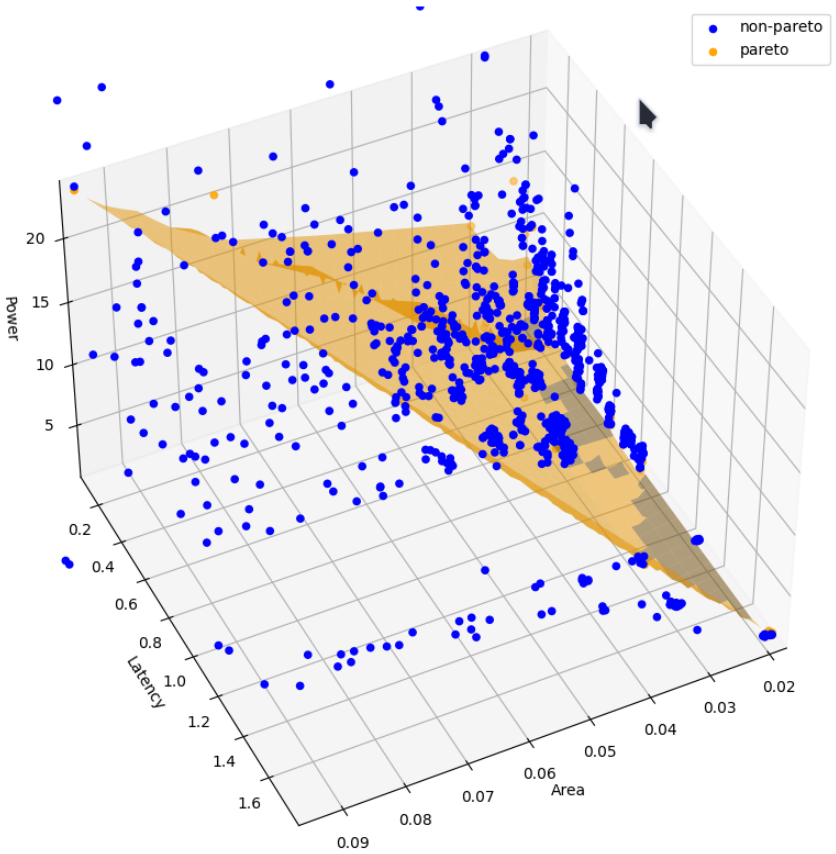

Figure 6.1: FC 3D cloud of points

6.1 Spearmint search space: FC

At this stage is necessary to define a search space in order to properly tune Spearmint. With this data is possible to define a search space of well behaving accelerators and which can surely work.

For the frequency (Freq) parameter, a maximum value of 1600 MHz is selected, since higher frequencies become unfeasible, while values below 200 MHz do not provide sufficient throughput. The step size between consecutive values is set to 200 MHz, in line with the discrete nature of the other parameters. However, because even small frequency variations can trigger significant changes at the RTL level, it remains interesting to observe how the optimization process explores this parameter and whether it identifies local minima where additional hardware is introduced to improve timing performance.

The number of output neurons **OUTNEU** is searched from 4 to 32 with steps of 4. These steps were taken to follow the same approach as `Urbinati's paper`, ensuring alignment and the generation of comparable data.

When considering possible loop configurations, the approach remains fixed: both unrolling and pipelining are applied, as previously discussed in Chapter 4.4.2. The search space of loops within *FC* is broad, comprising seven loops, some of which are nested (see Pseudocode 2.2). For external loops, full and half unrolling (FU and HU) are avoided, since unrolling inner loops is generally more beneficial, while unrolling external loops would significantly complicate the internal structure of the accelerator. Instead, these external loops are assigned a pipelining interleave of 2 (*II-2*), which improves throughput without adding excessive hardware complexity. Table 6.1 summarizes the configuration of these loops, including the labels used both in the configuration file and in the external framework for troubleshooting. For example, the *lp_init* loop appears with two entries in the table: one referring to the configuration file and another to the tool-chain framework, with the translation handled by the **BO** script.

6.2 FC: Search advancement

During the search for Pareto points, various scripts were developed to perform data analysis and plotting operations. The main processing step involved excluding duplicate points, as Spearmint unexpectedly chose the same input knobs within the same search leading to the same outputs and causing it to get stuck in local minima. These issues can distort the search space

Table 6.1: Numerical Encoding of Loop Optimization Levels for **FC**

Loop	II-0	II-2	II-1	HU	FU
Catapult Level	4	3	2	1	0
lp_init_config	3		2	1	0
lp_init_BO	4		2	1	0
c_for_config	4	3	2		
c_for_BO	4	3	2		
k_for_config	4			3	2
k_for_BO	4			1	0
wb_for_config	3		2	1	0
wb_for_BO	4		2	1	0
q_for_config	3		2	1	0
q_for_BO	4		2	1	0

and negatively affect the quality of the results. As future work, Spearmint could be modified to include a limited number of random searches during the optimization process, in order to escape local minima and reduce the amount of duplicated points.

The research in the case of the **FC** has been stopped at around 1700 iteration, due the time taken to perform the research, about one month, and due the fact, that the Pareto curve was well populated.

6.2.1 FC: pareto found

The three objectives of the search are: silicon area, power consumption, and execution latency. According to the definition of a Pareto point, the goal is to identify the best-performing solutions in at least one of the three objectives. In fact, there is no single Pareto solution that outperforms all others. Instead, multiple solutions may dominate in more than one objective.

With the progression of Bayesian Optimization iterations, the lack of sufficient output log from Spearmint made it impossible to directly track the MOBO's advancement. To overcome this, the progression of MOBO is extrapolated from the data collected during these iterations, resulting in the **Pareto Found** graph for FC in Fig. 6.2.

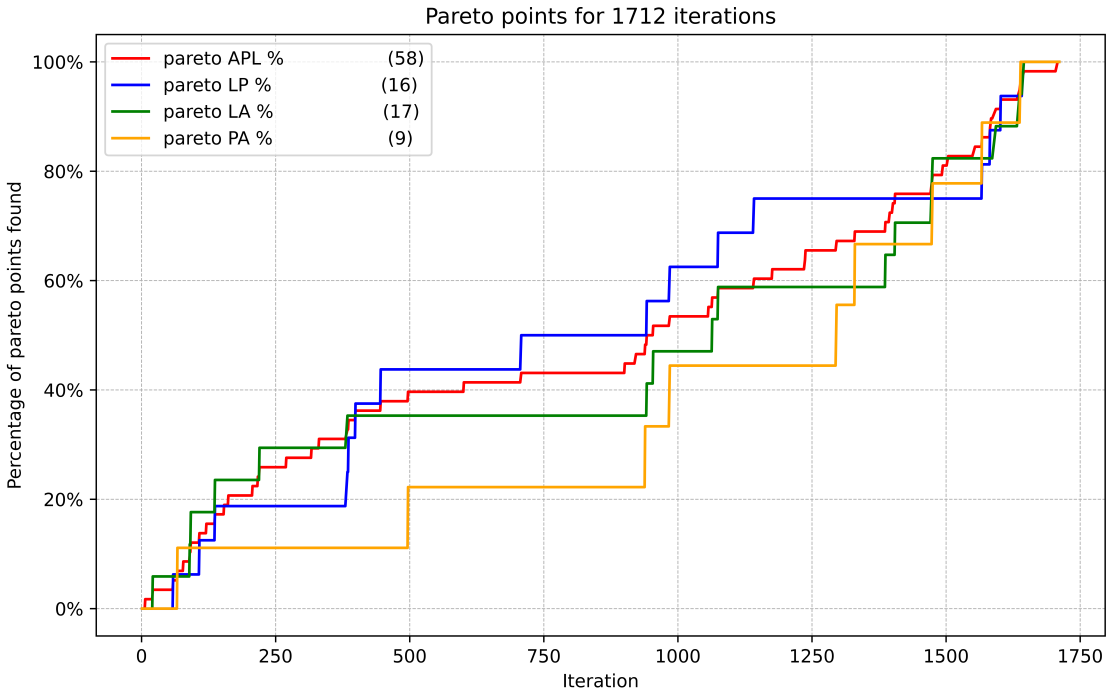

Figure 6.2: FC, Pareto search advancement over iterations

The **Pareto Found** graph 6.2 is one of the most essential visualization tools showing the percentage of Pareto points discovered at each iteration. This graph helps identify the iterations where most Pareto points are found and where a plateau occurs, indicating that no new Pareto points are discovered in that interval. If the graph shows a plateaus towards the end, the search can potentially be stopped, as most Pareto points have already been identified. However, this remains speculative unless a prior grid search has been conducted. In fact, it is uncertain whether additional Pareto-optimal points are still undiscovered among the remaining candidates, however the maximum allowed search time is another important constraint to consider for stopping the MOBO search. The graph in Fig. 6.2 displays four distinct lines *LP, LA, AP, APL*, representing the percentage of pareto point discovered in the Latency-Power (LP), Latency-Area (LA), Power-Area (AP) planes, and the union of these sets, respectively. The legend also indicates the total number of points found for each category in bracket, while the title specifies the total number of iterations completed by the MOBO algorithm.

Several observations can be made from this graph:

- By approximately the 300-th iteration, one-third of the total Pareto points have already been discovered.

- Two plateaus are visible in the ranges of 500–900 and 1100–1300 iterations.
- In the final 200 iterations, the trend continues to rise, suggesting that further iterations could yield additional Pareto points.
- The number of Pareto AP points is significantly lower than the others.

6.3 FC: 2D projections

6.3.1 FC: The LA projection

The **2D Projection** graph in Fig. 6.3 projects the 3D Pareto points on the 2D latency vs area space. This type of visualization helps analyze the Pareto points easily on a 2D plane rather than on the original 3D space. Moreover, thanks to the iterations on the color bar, it tracks with a color gradient the order in which Pareto points were discovered and how the Pareto front evolves over time, from purple (initial iterations) to red (last iterations).

Pareto APL points are marked with circles, while points on the 2D Pareto front are marked with crosses. Since each 2D Pareto point is a subset of a Pareto APL point, all crosses are included in circles. Finally, the crossed points are connected with a black line to better illustrate the Pareto front, which defines the space of LA Pareto solutions.

Fig. 6.3 has two subplots: the right one displays all data points, while the left one provides a zoomed-in view of the Pareto front. The latter curve is particularly useful for choosing an FC accelerator solution with a specific latency requirement, allowing a designer to determine the minimum area needed to achieve it, and vice versa.

For the **FC** case in Figure 6.3 is possible to take some conclusions:

- Hyperbolic trend, that identifies the inverse correlation between latency and area.
- The iteration-color distribution is not concentrated in clusters, which indicates that the research does not work by small steps, but in perceived probability. This comes from the nature of the BO (see Chapter 1.3).
- Latency has a more "discrete" distribution steps. This is due to the choice of the search space which is composed of knobs with a discrete set of values.

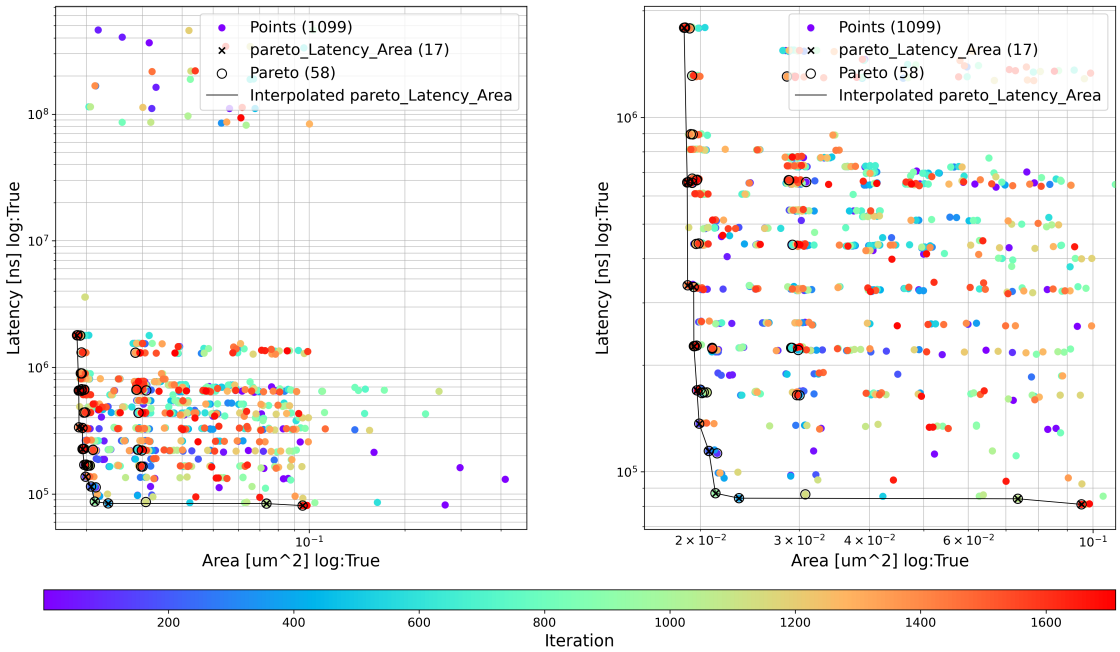

Figure 6.3: FC, Pareto 2D projection on Latency-Area (right is zoom on front)

FC: LA Pareto front points

Table 6.2: FC pareto **LA** plane sorted by latency

Iteration	Lp C	Lp K	Lp Init	Lp Q	Lp WB	OutCh	InCh	Freq	Latency[s]	Power[mW]	Area[µm²]
1645	II-2	FU	II-2	II-2	II-2	32	1024	1600	$8.094e-05$	23.950	$9.545e+04$
1075	II-2	II-0	FU	II-0	HU	24	1024	1600	$8.393e-05$	20.360	$7.353e+04$
384	II-2	HU	II-2	II-2	II-2	4	1024	1600	$8.425e-05$	14.030	$2.344e+04$
942	II-2	HU	II-2	II-0	HU	4	1024	1600	$8.700e-05$	9.430	$2.128e+04$
92	II-2	HU	HU	II-0	HU	4	1024	1200	$1.145e-04$	10.300	$2.071e+04$
220	II-2	II-0	HU	II-0	HU	4	1024	1200	$1.145e-04$	10.300	$2.071e+04$
137	II-2	HU	II-2	II-0	II-2	4	1024	1000	$1.367e-04$	6.679	$1.990e+04$
21	II-2	II-0	II-2	II-0	HU	4	1024	800	$1.698e-04$	5.892	$1.972e+04$
90	II-2	HU	II-2	II-0	HU	4	1024	800	$1.698e-04$	5.892	$1.972e+04$
1064	II-2	II-0	II-2	II-0	HU	4	1024	600	$2.262e-04$	2.751	$1.947e+04$
954	II-2	II-0	FU	II-2	HU	4	1024	400	$3.323e-04$	3.157	$1.946e+04$
106	II-2	HU	II-2	II-0	II-2	4	1024	400	$3.360e-04$	3.055	$1.912e+04$
1154	II-0	II-0	II-2	II-0	FU	4	1024	1000	$4.890e-04$	4.372	$1.907e+04$
1405	II-1	II-0	II-2	II-0	HU	4	1024	400	$6.555e-04$	2.556	$1.895e+04$
1637	II-1	FU	II-2	II-0	HU	4	1024	400	$6.555e-04$	2.556	$1.895e+04$
1475	II-0	HU	FU	II-0	HU	4	1024	200	$1.791e-03$	1.610	$1.871e+04$

In Tab. 6.2 high area solutions are at the top and low area ones are at the bottom. The situation with the Latency is opposite.

Frequency: at first glance, the *frequency* exhibits a clear downward trend. Higher *frequency* results in lower latency, but also requires more area, possibly

due to stronger gates and a higher use of buffer cells in the digital design. On the other hand, at lower *frequency*, as expected, latency increases while the area decreases. The frequency range is fully utilized, but a finer granularity could lead to more optimal solutions. For instance, 1400 MHz solutions are missing from the table, likely due to suboptimal results.

OutNeu is almost always fixed at 4, with a few exceptions at high frequency and low area. Obviously, a higher number of output neurons produced by the accelerator leads to a lower latency, but in this case, saturation to 4 occurs very quickly. From Table 6.2, it is evident that this parameter significantly influences the search process, and its entire range is utilized. Since the values *32* and *24* appear only once, while intermediate values do not appear at all, *4* emerges as the most favorable value. The brief occurrences of these high values are associated with lower latency, but comes at the cost of increased area (and power consumption, which is not considered in this analysis). The benefits of higher **OutNeu** values appear to result in only marginal improvements, suggesting its use only for extremly-low latency designs. Thus, setting a low-latency constrain might force the research to look for higher **OutNeu**.

Here is a brief commentary on the effect of high-level synthesis directives on loops for the latency-area 2D space.

- **loop init**:

  The impact of this parameter appears to be minimal, suggesting that other design choices play a more significant role in determining performance. Since the **II-2** value frequently appears in *LA* Pareto-optimal solutions, it may indicate a reasonable trade-off between performance and resource utilization.

- **loop c**:

  The **II-2** setting is dominant in Pareto-optimal points, highlighting its effectiveness in balancing latency and area. **II-2** is associated to low-latency.

  The inverse relationship between initiation interval (II) and area consumption follows a predictable monotonic trend.

- **loop K**:

This loop's influence appears limited, as no significant variation is seen across different Pareto-optimal points. The presence of **FU** and **HU** in the optimal configurations suggests that unrolling has a moderate impact on resource usage but does not strongly influence latency in this particular case.

- **loop q**:

  The values of **II-0** and **II-2** dominate, confirming that these settings provide the best area efficiency. II-0 is the most frequently observed setting, likely due to its ability to minimize area while maintaining reasonable latency. The fact that no full unrolling (FU) appears indicates that excessive unrolling is not beneficial in this loop for latency-area trade-offs.

- **loop wb**:

  Half Unrolling (**HU**) is the most frequently occurring value, indicating that provides a strong trade-off between reducing latency and limiting area growth. **FU**, while appearing occasionally, consumes significantly more area without proportional gains in latency reduction, making it less favorable in most Pareto-optimal points.

6.3.2 FC: LP projection

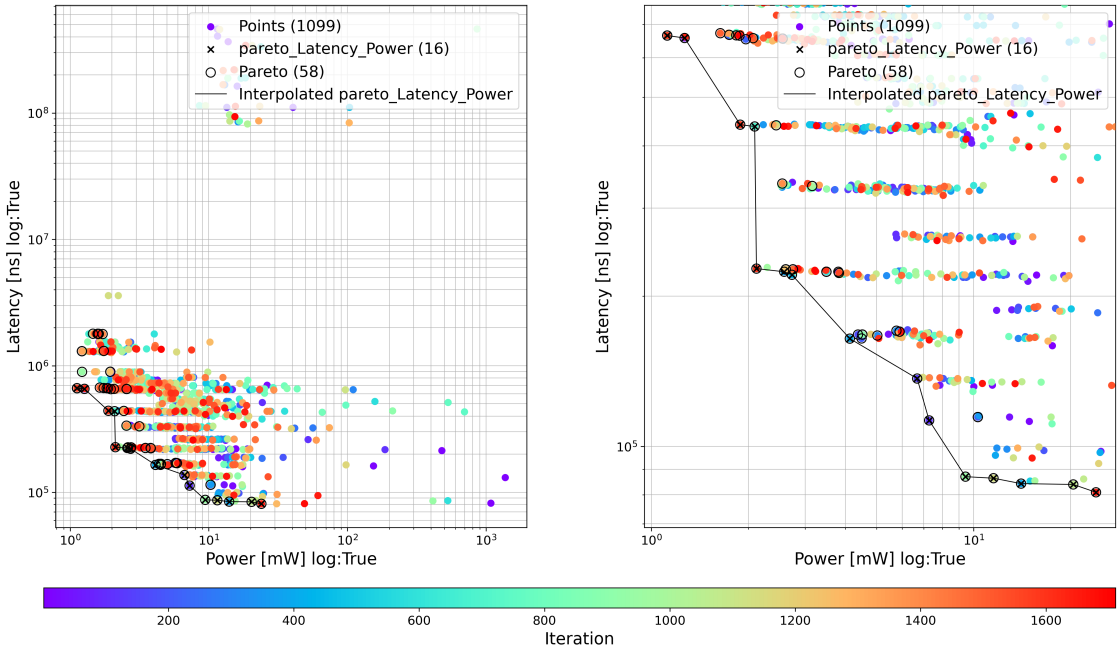

Figure 6.4: FC, Pareto 2D projection on Latency-Power (right is zoom on front)

A similar graph is made for Latency-Power subset in Fig. 6.4 This kind of solution looks at a trade-off between power consumption and latency. This graph can be used to designs high-performance or low-power FC accelerator design. Other conclusions can be derived:

- Here iteration-colors does not have a pattern, so the research is not linear.
- LP points still has an inverse correlation between latency and power.

FC : LP Pareto front points

In this table, high area and power solutions are at the top and low area are at the bottom.

Frequency As expected, solutions that require more area and are more power-hungry are found at high **frequency**. The distribution is quite uniform, and the same conclusions can be drawn as in the latency-area case.

Table 6.3: FC pareto **LP** plane sorted by latency

Iteration	Lp C	Lp K	Lp Init	Lp Q	Lp WB	OutCh	InCh	Freq	Latency[s]	Power[mW]	Area[µm²]
1645	II-2	FU	II-2	II-2	II-2	32	1024	1600	$8.094e-05$	23.950	$9.545e+04$
1075	II-2	II-0	FU	II-0	HU	24	1024	1600	$8.393e-05$	20.360	$7.353e+04$
384	II-2	HU	II-2	II-2	II-2	4	1024	1600	$8.425e-05$	14.030	$2.344e+04$
1142	II-2	II-0	FU	II-0	FU	8	1024	1600	$8.637e-05$	11.520	$3.076e+04$
942	II-2	HU	II-2	II-0	HU	4	1024	1600	$8.700e-05$	9.430	$2.128e+04$
108	II-2	HU	HU	II-2	FU	4	1024	1200	$1.128e-04$	7.275	$2.142e+04$
137	II-2	HU	II-2	II-0	II-2	4	1024	1000	$1.367e-04$	6.679	$1.990e+04$
399	II-2	HU	FU	II-2	HU	8	1024	800	$1.644e-04$	4.130	$3.004e+04$
446	II-2	HU	FU	II-2	FU	8	1024	600	$2.206e-04$	2.731	$2.987e+04$
386	II-2	HU	II-2	II-0	HU	8	1024	600	$2.237e-04$	2.584	$2.913e+04$
1602	II-2	II-0	HU	II-0	HU	4	1024	600	$2.270e-04$	2.120	$1.969e+04$
1582	II-1	II-0	II-0	II-0	II-0	4	1024	600	$4.408e-04$	1.888	$1.987e+04$
59	II-2	FU	FU	HU	II-0	8	1024	200	$6.572e-04$	1.269	$3.086e+04$
985	II-2	FU	FU	II-0	HU	8	1024	200	$6.647e-04$	1.121	$2.878e+04$

OutNeu In high-frequency versions, **OutNeu** exhibits a downward trend, decreasing from 32 to 4. However, this trend is not globally consistent across all configurations, even within fixed-frequency clusters. **OutNeu is generally set to either 4 or 8**, as these values provide the best trade-off in LP Pareto-optimal solutions. This parameter proves to be highly effective in optimization, but at lower values, the impact of other parameters becomes more significant. In future, restricting the search space by focusing on the most effective **OutNeu** values could enhance the efficiency of the optimization process.

Analysis of Latency and Power Consumption in Pareto-optimal Points Below is an analysis of how various design knobs influence latency and power consumption.

- **loop init**:

  The loop initialization parameter has a relatively subtle impact on both latency and power consumption. The value **II-2** tends to appear more frequent in Pareto-optimal points, suggesting that it strikes a balance between performance and resource utilization. The initialization value impacts the speed of execution without significantly affecting power, meaning that its main role is to optimize the trade-off between latency and area.

- **loop c**:

  II-2 is the most common choice in Pareto-optimal solutions, as it consistently yields a favorable balance between power and latency.

Similar conclusions can be taken as for **LA**.

- **loop K**:

  The impact of **loop K** on latency and power consumption appears minimal. The presence of **FU** and **HU** in the Pareto-optimal configurations indicates that the unrolling factor has a modest effect on both latency and power consumption. However, the variation in power is not drastic across different points, implying that this parameter is not a dominant factor for optimization in the current design space.

- **loop q**:

  The **II-0** and **II-2** values dominate the Pareto-optimal solutions, and this reinforces the idea that these settings offer a good balance between power efficiency and latency reduction. **II-0** is particularly effective in minimizing power consumption, making it a common choice. The absence of **FU** in the Pareto-optimal points suggests that aggressive unrolling leads to a non-optimal increase in power consumption.

- **loop wb**:

  HU stands out as the most frequent value for loop **wb**, showing that half-unrolling provides an efficient trade-off between reducing latency and keeping power consumption lower than in configurations that use full unrolling. **FU** appears infrequently and tends to consume more power without offering substantial latency benefits, which explains why it is less favored in the Pareto-optimal points.

6.3.3 FC: AP projection

FC : AP pareto

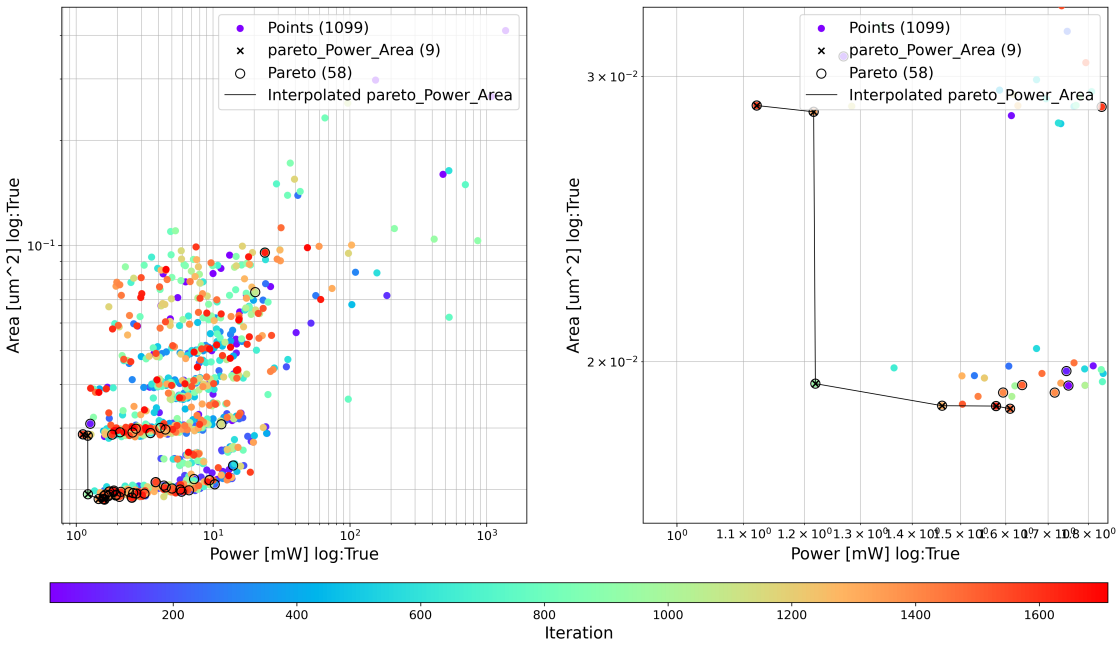

Figure 6.5: FC, Pareto 2D projection on Area-Power (right is zoom on front)

In the same way, Fig. 6.5 shows the Area-Power subset of Pareto points. This graph is quite different from the previous one, since area and power have a linear correlation. Due to this, the pareto points are the lowest amount in the projection. A design that minimize both power and area by neglecting latency is quite useless. This graph starts to make sense if only the points with some latency target are filtered. Few Pareto solutions are identified in this projection.

FC : AP Pareto front points

In this table, high-area solutions are at the top, while low-area solutions are at the bottom. Power exhibits the opposite behavior.

Frequency : In this case there is a different behavior, since latency is neglected. There is no need to increase the frequency to produce faster data. The frequencies utilized is 200 MHz in most cases, but with one overshoot

Table 6.4: FC pareto **PA** sorted Power

Iteration	Lp C	Lp K	Lp Init	Lp Q	Lp WB	OutCh	InCh	Freq	Latency[s]	Power[mW]	Area[µm²]
985	II-2	FU	FU	II-0	HU	8	1024	200	$6.647e-04$	1.121	$2.878e+04$
67	II-1	II-0	II-2	II-0	HU	8	1024	200	$1.305e-03$	1.216	$2.853e+04$
1330	II-1	HU	II-2	II-0	HU	8	1024	200	$1.305e-03$	1.216	$2.853e+04$
939	II-0	HU	II-2	II-0	HU	4	1024	400	$8.955e-04$	1.219	$1.939e+04$
1296	II-0	HU	HU	II-0	HU	4	1024	200	$1.793e-03$	1.461	$1.879e+04$
497	II-0	HU	II-2	II-0	HU	4	1024	200	$1.791e-03$	1.578	$1.878e+04$
1475	II-0	HU	FU	II-0	HU	4	1024	200	$1.791e-03$	1.610	$1.871e+04$

at 400 MHz. If the aim is to find this kind of Pareto set, the frequency knob can be fixed to reduce the search space.

OutNeu exhibits a similar behavior as before, but in this case, there is a linear trend. Low-power and high-area configurations are associated with 8 **OutNeu**, while high-power and low-area ones are linked to 4 **OutNeu**.

Based on the analysis of the data across various loop parameters, and considering the low amount of pareto points to have a strong static to analyze the following conclusions can be drawn about power and area:

- **loop init**:

  The impact of **loop init** on power and area is relatively minimal. The most frequent value, **II-2**, suggests that it is favored for power efficiency while maintaining a reasonable area size.

- **loop c**:

  The **II-0** configuration is the most dominant in Pareto-optimal solutions, offering a solid benefits in both power and area. This suggests that **II-0** is a more efficient choice for minimizing area and power, while unrolling (e.g., **FU**) is beneficial for reducing latency but increases both power and area significantly.

- **loop K**:

  For this knob unrolling techniques, in particular **HU**, represents the best performing values in AP constraints.

- **loop q**:

  The **II-0** is the only appearing, this indicates the strong effect of this value-knob, to the overall contribution of area and power.

- **loop wb**:

  The most frequent value for **loop wb** is **HU**, which appears to offer a good balance between reducing both latency and power consumption, while keeping area relatively low. **FU**, is not present in some points, tends to increase both power consumption and area significantly. This suggests that while full unrolling can reduce latency, it is not always the best choice for optimizing power and area in the Pareto-optimal solutions.

6.4 FC: search validation graphs

6.4.1 FC: duplicates

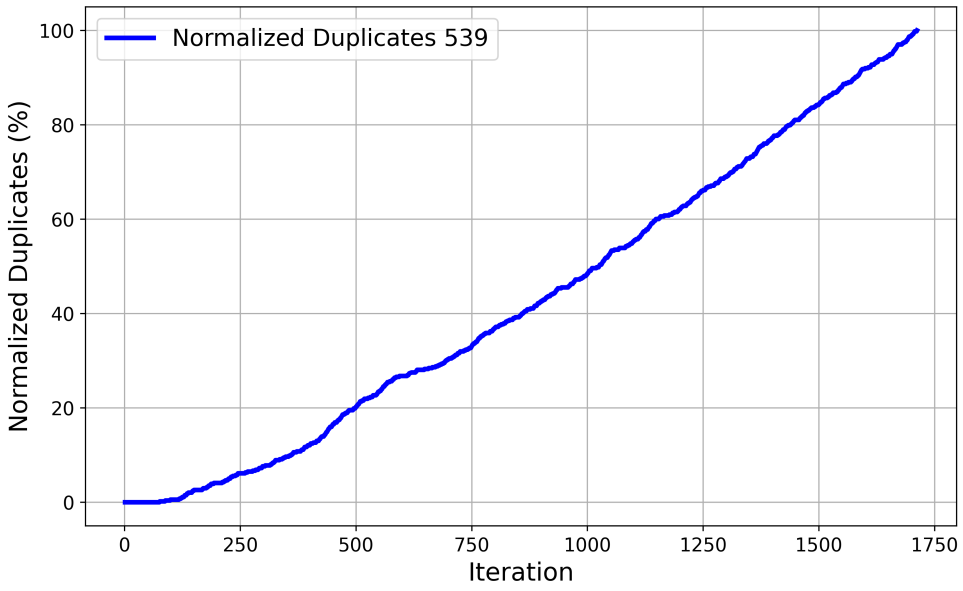

Figure 6.6: FC, Pareto duplicated designs.

A plot that illustrates the search progression is presented in Fig. 6.6. It shows the **Duplicates**, *i.e.*, the frequency of re-utilization of the same input parameters during the search. This behavior indicates that the algorithm is not exploring new solutions effectively, instead it is reconsidering previously

evaluated ones. A sharp increase toward the end of the graph may suggest that the algorithm has converged to a local minimum.

From Fig. 6.6 it can be seen that the overall duplicates for FC are 539 over 1712 iterations. Some conclusions can be extracted:

- A trend can be observed in the first 1000 iterations of the MOBO process, where approximately 50% of the total duplicates are found — this corresponds to approximately 250 duplicates. This implies an average of one duplicate every four iterations during the first half of the search.
- In the second half of the search, the remaining 50% of duplicates are distributed over the final 712 iterations, resulting in a denser trend of approximately one duplicate every three iterations.
- In the early iterations, the growth of duplicates follows a sub-linear exponential trend, meaning that the tool starts revisiting previously evaluated points as it approaches the end of the MOBO process.

In such cases, the search process can either be terminated or continued by injecting additional random points to promote further exploration. It is possible to believe that this behavior may be due to a bug in the implementation of the MOBO algorithm within Spearmint. Nevertheless, the tool could be modified to automatically introduce new exploration points whenever a certain number of previously evaluated configurations are selected again.

6.4.2 FC: max-min-distance

The **Max-Min Distance** graph is the last that we propose to study the quality of the MOBO search. During the research, it is difficult to determine from the 2D projections in Figs. 7.3–7.4 whether a new cluster of Pareto points has been discovered. This graph tries to solve this problem by evaluating the Euclidean distance between previously found Pareto points and newly discovered ones. Ideally, this distance should approach zero over time, indicating that the Pareto curve is well populated.

By analyzing the plot, one can observe three distinct tracks, each representing the distance between the Pareto-optimal points across the three different Pareto projections. The number of points in each curve varies, as each projection has a different number of Pareto-optimal solutions. It is evident that the distances remain relatively small, given that the Euclidean distance is computed using normalized values across all three dimensions.

Overall, the curves exhibit a tendency to converge towards zero, indicating that newly discovered Pareto points are increasingly located near previously identified ones. The only point against tendency occurs in **LA** set, where a new cluster/point is discovered around 1500-th iteration.

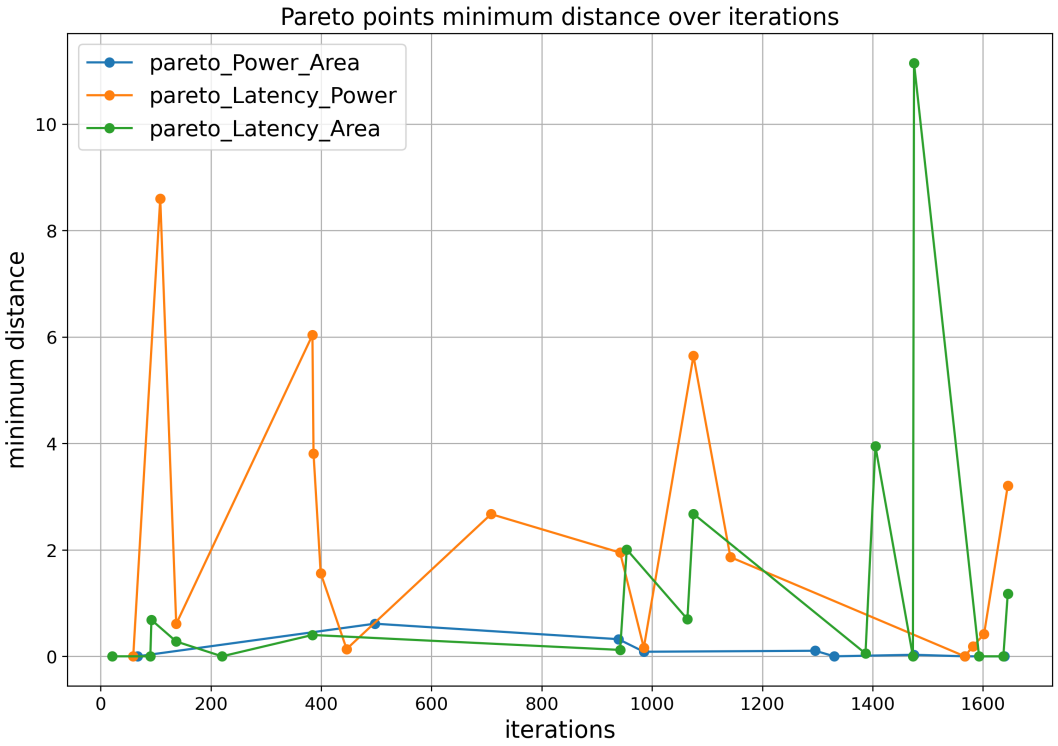

Figure 6.7: Qualitative Euclidean distance between found points.

6.4.3 FC: Reported Errors

Error graph keeps track of the failed iteration of Spearmint. Often they are linked to the attempt to design a RTL inside Catapult with unfeasible memory design. This information, which is tracked inside the Spearmint search advancement, helps to distinguish which knob combinations are not feasible in order to reduce failed attempts. The faster the error curve approaches zero, the quicker Spearmint learns to avoid invalid configurations. As stated before, errors don't appear in FC accelerators.

For the **FC** case, no faulty iterations were observed: none of the explored solutions resulted in a design failure at any level.

Chapter 7

CONV: EXPERIMENTAL RESULTS

7.1 Spearmint search space : CONV

The search space for the **CONV** design is defined analogously to that of the **FC**. For **CONV**, the number of output channels (**OUTCH**) is varied linearly from 4 to 32 in steps of 4, while the number of input channels (**INCH**) is fixed at 32, as this setting provides the highest throughput. The same strategy adopted for the **FC** design is applied to loop transformations in **CONV**: the more complex loops can either be pipelined with an initiation interval of 2 or 1, or left sequential, whereas simpler loops can be either partially or fully unrolled.

7.2 CONV: Search advancement

7.2.1 CONV: pareto found

In Fig. 7.2, the Pareto Found graph is reported. A total of 1074 iterations were executed, with the majority of Pareto points identified within the first 600 iterations. Excluding the *LP* points, more than 80% of the Pareto *APL* points were discovered by the 400-th iteration. The search was interrupted at this stage due to an error in the `spearmint` script, which caused the

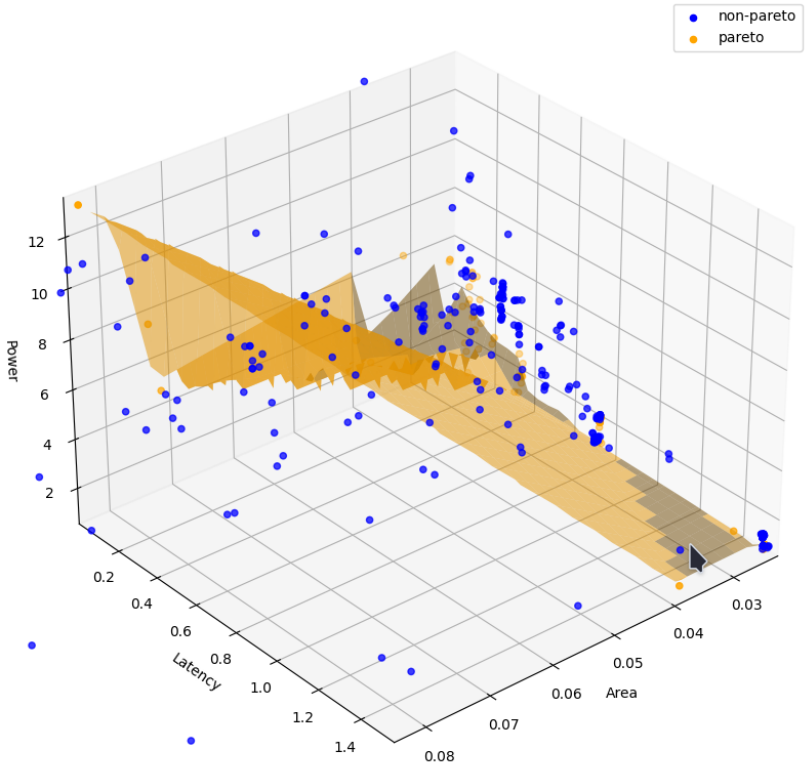

Figure 7.1: CONV 3D cloud of points

Table 7.1: Numerical Encoding of Loop Optimization Levels

Loop	II-0	II-2	II-1	HU	FU
Catapult Level	4	3	2	1	0
lp_init_config	3		2	1	0
lp_init_BO	4		2	1	0
ci_for_config	4	3	2		
ci_for_BO	4	3	2		
k_h_for_config	4	3	2		
k_h_for_BO	4	3	2		
k_w_for_config	4	3	2		
k_w_for_BO	4	3	2		
co_for_config	3		2	1	0
co_for_BO	4		2	1	0
wb_for_config	3		2	1	0
wb_for_BO	4		2	1	0
q_for_config	3		2	1	0
q_for_BO	4		2	1	0

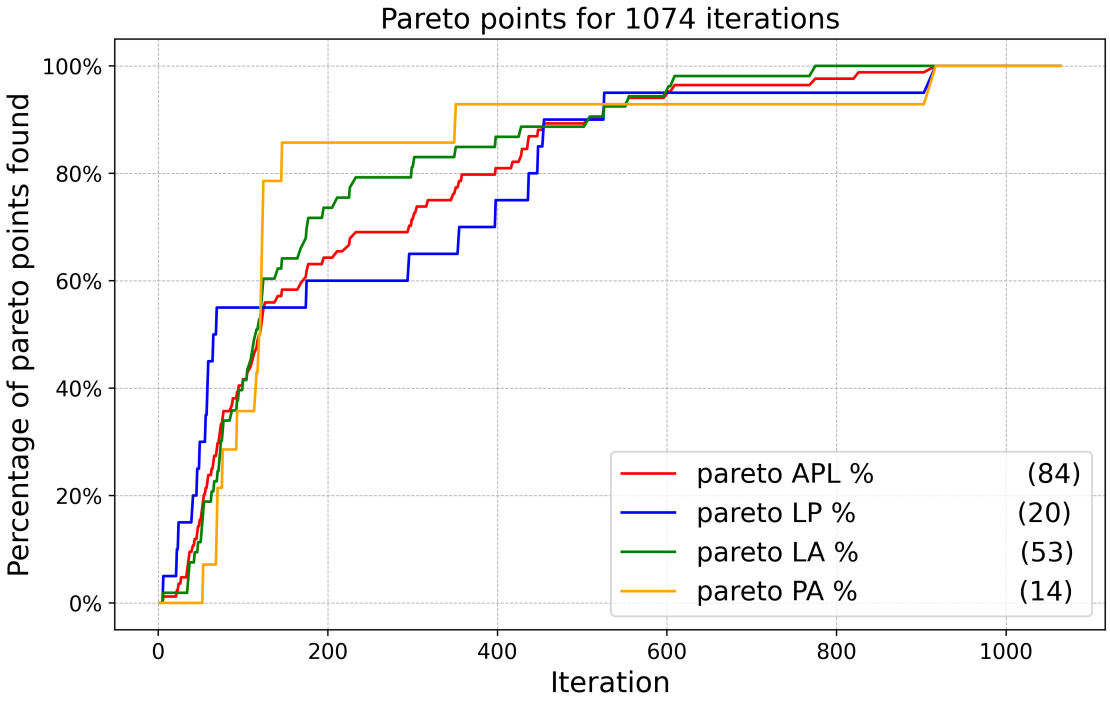

Figure 7.2: CONV Pareto APL search progression

process to stall around a local minimum. Notably, *LP* required more iterations to converge and reveal its full set of Pareto points. When aggregating the Pareto points obtained from *LA*, *LP*, and *AP*, their total exceeds that of *APL*, indicating that some Pareto points are shared across sub-groups. As previously discussed, owing to the linearity of the Pareto projection *AP* (6.3.3), only a limited number of Pareto points fall within this category. In other words, the low-area–low-power solutions occupy the lower-left region of the plot, dominating the others.

7.3 CONV: 2D projections

7.3.1 CONV: LA projection

CONV : LA pareto

As in the case of **FC**, the Pareto 2D curves for the **LA**, **LP**, and **AP** spaces are plotted in Figs. 7.3–7.5, respectively. In **LA** (Fig. 7.3), the inverse proportionality is confirmed, and the discovered points appear to be randomly distributed. Latency points, however, tend to align horizontally. The total

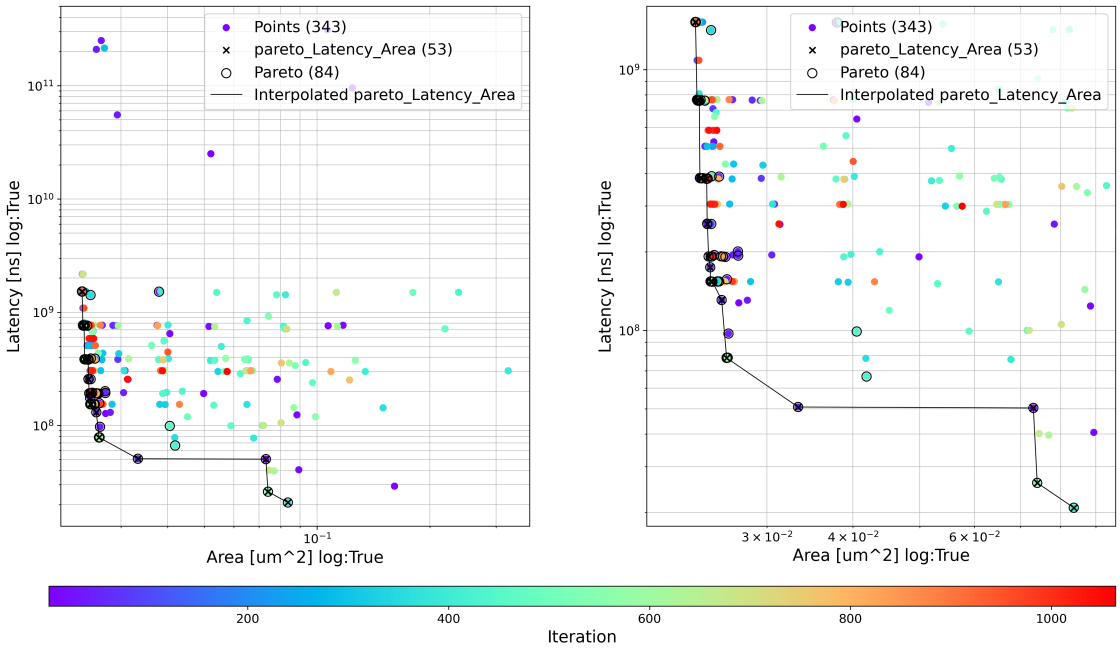

Figure 7.3: CONV Pareto projection on Latency-Area

number of identified Pareto points exceeds 50, although the plot displays slightly fewer. Several points lie in close proximity, making them difficult to distinguish in the figure; nevertheless, upon closer inspection (e.g., through zooming), their separation becomes evident.

7.3.2 CONV: LP projection

CONV : LP pareto

As in the **FC** case, the Pareto front for the **LP** scenario is visualized in 2D for **CONV** (Fig. 7.4). Since the main characteristics have already been discussed for the fully connected design, no further in-depth analysis is provided here. An additional zoomed view is included to highlight finer details. The inverse relationship between latency and area is once again evident, although the distribution of Pareto-optimal points appears scattered. Latency values, however, exhibit a more structured alignment, suggesting an underlying pattern. In total, 20 Pareto points were identified, a number that can be readily verified from the zoomed portion of the plot.

Table 7.2: CONV pareto **LA** sorted Latency

Iteration	Lp CI	Lp CO	Lp W&H	LP Init	Lp Q	Lp WB	OutCh	InCh	Freq	Latency[s]	Power[mW]	Area[µm²]
398	II-1	FU	II-0	HU	II-0	II-1	16	32	1000	0.021	13.260	$8.360e+04$
526	II-2	HU	II-2	II-1	II-0	HU	16	32	800	0.026	7.685	$7.400e+04$
6	II-2	HU	II-2	FU	II-1	HU	16	32	400	0.050	5.032	$7.305e+04$
52	II-2	FU	II-1	II-1	HU	II-1	4	32	800	0.051	6.512	$3.330e+04$
602	II-2	HU	II-0	HU	II-0	II-1	4	32	1000	0.078	5.779	$2.623e+04$
36	II-2	HU	II-2	HU	II-0	II-1	4	32	1000	0.078	5.779	$2.623e+04$
54	II-2	HU	II-1	II-1	II-0	II-1	4	32	600	0.131	4.026	$2.580e+04$
609	II-2	II-1	II-0	HU	II-0	HU	4	32	1000	0.154	5.457	$2.502e+04$
555	II-2	II-0	II-0	II-0	II-0	HU	4	32	1000	0.154	5.480	$2.495e+04$
428	II-2	II-1	II-0	II-1	II-0	II-1	4	32	1000	0.154	5.221	$2.486e+04$
299	II-2	II-0	II-0	II-1	II-0	II-1	4	32	1000	0.154	5.221	$2.486e+04$
51	II-0	II-1	II-2	II-0	II-0	II-1	4	32	1000	0.175	5.384	$2.483e+04$
302	II-2	II-1	II-0	II-0	II-0	HU	4	32	800	0.192	3.498	$2.470e+04$
37	II-2	II-1	II-0	FU	II-0	II-1	4	32	600	0.256	3.200	$2.460e+04$
47	II-2	II-1	II-1	II-1	II-0	II-1	4	32	600	0.257	3.161	$2.458e+04$
775	II-1	II-1	II-0	II-0	II-0	II-1	4	32	800	0.381	3.116	$2.454e+04$
63	II-2	II-1	II-1	HU	II-0	HU	4	32	400	0.383	2.147	$2.414e+04$
211	II-2	II-0	II-2	HU	II-0	HU	4	32	400	0.383	2.147	$2.414e+04$
509	II-2	II-1	II-0	HU	II-0	HU	4	32	400	0.383	2.147	$2.414e+04$
43	II-2	II-1	II-2	II-0	II-0	HU	4	32	400	0.383	2.367	$2.402e+04$
35	II-2	II-1	II-1	II-1	II-1	II-0	4	32	400	0.384	2.666	$2.399e+04$
168	II-1	II-0	II-2	FU	II-1	HU	4	32	400	0.759	1.858	$2.395e+04$
72	II-1	II-1	II-0	HU	II-0	HU	4	32	400	0.760	2.071	$2.390e+04$
109	II-1	II-0	II-2	HU	II-0	HU	4	32	400	0.760	2.071	$2.390e+04$
111	II-1	II-0	II-0	HU	II-0	HU	4	32	400	0.760	2.071	$2.390e+04$
177	II-1	II-1	II-2	HU	II-0	HU	4	32	400	0.760	2.071	$2.390e+04$
87	II-1	II-0	II-1	HU	II-0	HU	4	32	400	0.760	2.071	$2.390e+04$
174	II-1	II-1	II-1	HU	II-0	HU	4	32	400	0.760	2.071	$2.390e+04$
226	II-1	II-1	II-0	II-1	II-0	HU	4	32	400	0.761	2.245	$2.385e+04$
77	II-1	II-0	II-1	II-1	II-0	HU	4	32	400	0.761	2.245	$2.385e+04$
105	II-1	II-1	II-1	II-1	II-0	HU	4	32	400	0.761	2.245	$2.385e+04$
233	II-1	II-1	II-2	II-1	II-0	HU	4	32	400	0.761	2.245	$2.385e+04$
73	II-1	II-0	II-0	II-1	II-0	HU	4	32	400	0.761	2.245	$2.385e+04$
113	II-1	II-0	II-2	II-1	II-0	HU	4	32	400	0.761	2.245	$2.385e+04$
100	II-1	II-1	II-1	II-1	II-0	II-1	4	32	400	0.761	2.287	$2.382e+04$
74	II-1	II-0	II-2	II-1	II-0	II-1	4	32	400	0.761	2.287	$2.382e+04$
95	II-1	II-0	II-0	II-1	II-0	II-1	4	32	400	0.761	2.287	$2.382e+04$
195	II-1	II-1	II-2	II-1	II-0	II-1	4	32	400	0.761	2.287	$2.382e+04$
66	II-1	II-0	II-1	II-1	II-0	II-1	4	32	400	0.761	2.287	$2.382e+04$
141	II-1	II-1	II-0	II-1	II-0	II-1	4	32	400	0.761	2.287	$2.382e+04$
175	II-2	II-1	II-0	HU	II-0	HU	4	32	200	0.766	1.210	$2.376e+04$
351	II-1	II-1	II-2	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
70	II-1	II-1	II-1	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
116	II-1	II-1	II-0	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
121	II-1	II-0	II-0	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
119	II-1	II-0	II-1	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
93	II-1	II-0	II-2	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
124	II-1	II-1	II-0	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
123	II-1	II-1	II-2	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
122	II-1	II-0	II-1	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
53	II-1	II-0	II-0	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
146	II-1	II-0	II-2	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
76	II-1	II-1	II-1	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$

7.3.3 CONV: AP projection

CONV : PA pareto

In this case, the **AP** curve appears somewhat atypical, as it seems to consist of only three distinct points. A more detailed analysis is therefore required. By examining the tables generated from the same dataset (as shown later

Table 7.3: CONV pareto **LA** occurrences

Lp CI		Lp CO		Lp W&H		Lp init		Lp Q		Lp WB		OutCh		InCh		Freq	
Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%
II-1	62.26	II-1	50.94	II-0	39.62	II-1	58.49	II-0	92.45	HU	54.72	4.0	94.34	32.0	100.0	400	47.17
II-2	35.85	II-0	35.85	II-2	32.08	HU	26.42	II-1	5.66	II-1	43.4	16.0	5.66			200	24.53
II-0	1.89	HU	9.43	II-1	28.3	II-0	9.43	HU	1.89	II-0	1.89					1000	15.09
		FU	3.77			FU	5.66									800	7.55
																600	5.66

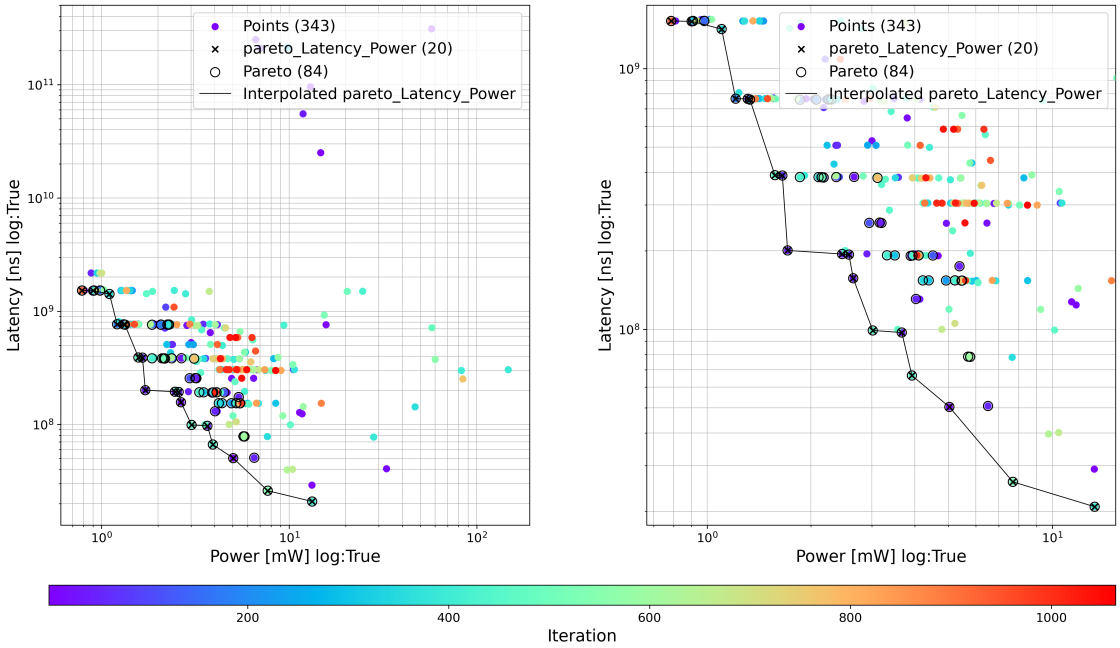

Figure 7.4: CONV Pareto projection on Latency-Power

in Table 7.6), the individual points on the Pareto front can be identified. In fact, the set includes more than ten points. However, since all of them share the same operating frequency of 200 MHz and four output channels, the only variation arises from the initiation interval applied to most loops. This suggests that, at low operating frequencies, the initiation interval has little impact on either area or power.

7.4 CONV: search validation graphs

7.4.1 CONV: errors

Fig. 7.6 reports the **CONV** errors, defined as unfeasible design generated by CADs when processing the CONV design selected at each iteration. The vertical axis is normalized with respect to the total number of errors observed

Table 7.4: CONV pareto **LP** sorted Latency

Iteration	Lp CI	Lp CO	Lp W&H	LP Init	Lp Q	Lp WB	OutCh	InCh	Freq	Latency[s]	Power[mW]	Area[µm²]
398	II-1	FU	II-0	HU	II-0	II-1	16	32	1000	0.021	13.260	$8.360e+04$
526	II-2	HU	II-2	II-1	II-0	HU	16	32	800	0.026	7.685	$7.400e+04$
6	II-2	HU	II-2	FU	II-1	HU	16	32	400	0.050	5.032	$7.305e+04$
437	II-2	HU	II-2	II-0	II-0	HU	8	32	600	0.067	3.920	$4.185e+04$
46	II-2	HU	II-2	II-1	II-1	FU	4	32	800	0.097	3.664	$2.639e+04$
448	II-2	HU	II-2	FU	II-1	II-1	8	32	400	0.099	3.019	$4.050e+04$
24	II-0	HU	II-2	II-0	II-0	FU	4	32	600	0.157	2.650	$2.626e+04$
56	II-1	FU	II-2	II-1	II-0	FU	4	32	400	0.193	2.576	$2.727e+04$
49	II-2	HU	II-1	HU	II-0	HU	4	32	400	0.194	2.461	$2.516e+04$
41	II-2	FU	II-0	II-1	II-0	HU	4	32	200	0.201	1.711	$2.724e+04$
65	II-2	HU	II-1	HU	II-1	HU	4	32	200	0.389	1.652	$2.558e+04$
455	II-2	HU	II-2	HU	II-0	II-1	4	32	200	0.390	1.572	$2.495e+04$
296	II-1	II-1	II-2	HU	II-1	HU	4	32	400	0.760	1.331	$2.439e+04$
58	II-1	II-1	II-0	HU	II-1	HU	4	32	400	0.760	1.331	$2.439e+04$
59	II-2	II-0	II-0	II-0	II-0	FU	4	32	200	0.765	1.312	$2.404e+04$
175	II-2	II-1	II-0	HU	II-0	HU	4	32	200	0.766	1.210	$2.376e+04$
355	II-0	HU	II-0	HU	II-1	II-1	4	32	200	1.417	1.102	$2.490e+04$
22	II-1	II-0	II-1	FU	II-1	HU	8	32	200	1.517	0.902	$3.791e+04$
69	II-1	II-1	II-0	II-0	II-0	HU	4	32	200	1.521	0.787	$2.367e+04$
917	II-1	II-1	II-2	II-0	II-0	HU	4	32	200	1.521	0.787	$2.367e+04$

Table 7.5: CONV pareto **LP** occurrences

Lp CI		Lp CO		Lp W&H		Lp init		Lp Q		Lp WB		OutCh		InCh		Freq	
Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%
II-2	55.0	HU	50.0	II-2	50.0	HU	40.0	II-0	60.0	HU	60.0	4.0	70.0	32.0	100.0	200	45.0
II-1	35.0	II-1	25.0	II-0	35.0	II-0	25.0	II-1	40.0	FU	20.0	16.0	15.0			400	30.0
II-0	10.0	FU	15.0	II-1	15.0	II-1	20.0			II-1	20.0	8.0	15.0			600	10.0
		II-0	10.0			FU	15.0									800	10.0
																1000	5.0

Table 7.6: CONV pareto **PA** sorted Power

Iteration	Lp CI	Lp CO	Lp W&H	LP Init	Lp Q	Lp WB	OutCh	InCh	Freq	Latency[s]	Power[mW]	Area[µm²]
69	II-1	II-1	II-0	II-0	II-0	HU	4	32	200	1.521	0.787	$2.367e+04$
917	II-1	II-1	II-2	II-0	II-0	HU	4	32	200	1.521	0.787	$2.367e+04$
70	II-1	II-1	II-1	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
93	II-1	II-0	II-2	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
116	II-1	II-1	II-0	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
119	II-1	II-0	II-1	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
121	II-1	II-0	II-0	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
351	II-1	II-1	II-2	II-1	II-0	HU	4	32	200	1.521	0.911	$2.364e+04$
53	II-1	II-0	II-0	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
76	II-1	II-1	II-1	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
122	II-1	II-0	II-1	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
123	II-1	II-1	II-2	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
124	II-1	II-1	II-0	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$
146	II-1	II-0	II-2	II-1	II-0	II-1	4	32	200	1.523	0.982	$2.364e+04$

Table 7.7: CONV pareto **PA** occurrences

Lp CI		Lp CO		Lp W&H		Lp init		Lp Q		Lp WB		OutCh		InCh		Freq	
Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%	Value	%
II-1	100.0	II-1	57.14	II-0	35.71	II-1	85.71	II-0	100.0	HU	57.14	4.0	100.0	32.0	100.0	200	100.0
		II-0	42.86	II-2	35.71	II-0	14.29			II-1	42.86						
				II-1	28.57												

over 1074 MOBO iterations. Each red star denotes a faulty design, *i.e.*, a design that failed in one of the three EDA steps: high-level synthesis, logic

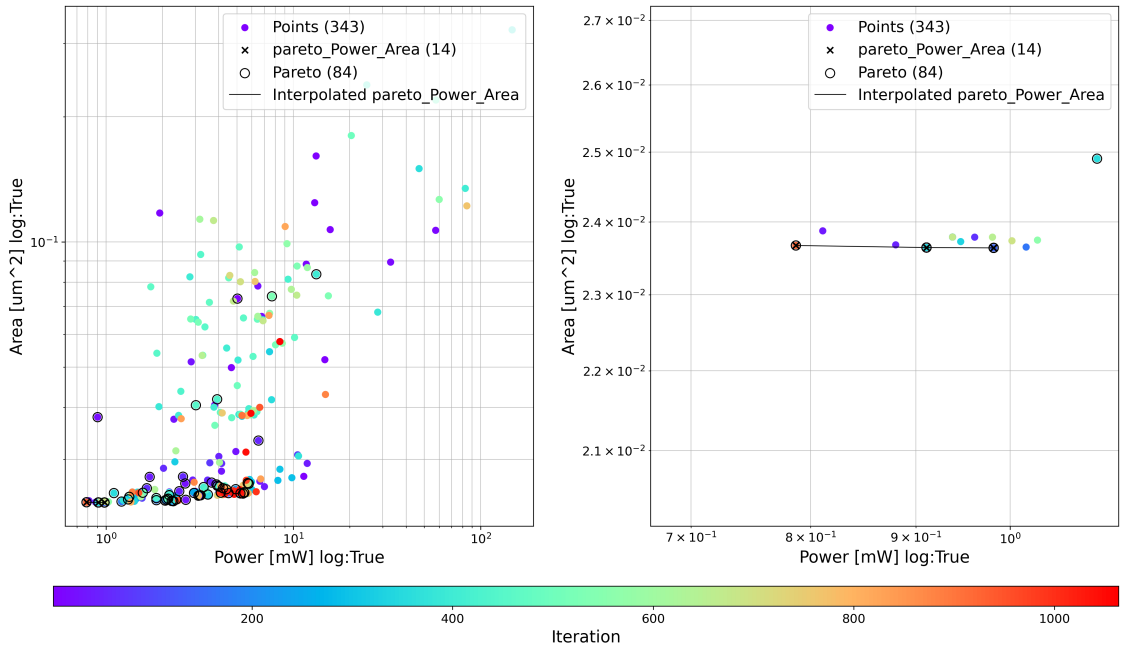

Figure 7.5: CONV Pareto projection on Area-Power

synthesis, or place and route. The most frequent cause is unfeasible memory generation, a Catapult-related issue associated with nested-loop unrolling. Most errors occur in the 0–50 and 300–450 iteration ranges, while after the 800th iteration Spearmint correctly identifies only feasible designs. In total, 25 errors were recorded, corresponding to a very small fraction of the overall iterations (2.3%).

7.4.2 CONV: duplicates

Figure 7.7 illustrates the duplicated solutions obtained for **CONV**. Approximately 50% of the duplicates occur before the 700th iteration, resulting in a total of 702 repeated iterations. This implies that more than half of the iterations correspond to duplicated solutions. Moreover, most of the iterations beyond the 700th iteration are also duplicates.

7.4.3 CONV: max-min-distance

This graph is generated using the same methodology adopted for the **FC**. It can be observed that, during the early stage (before the 100th iteration), the tool explores more distant points. After this phase, with few exceptions

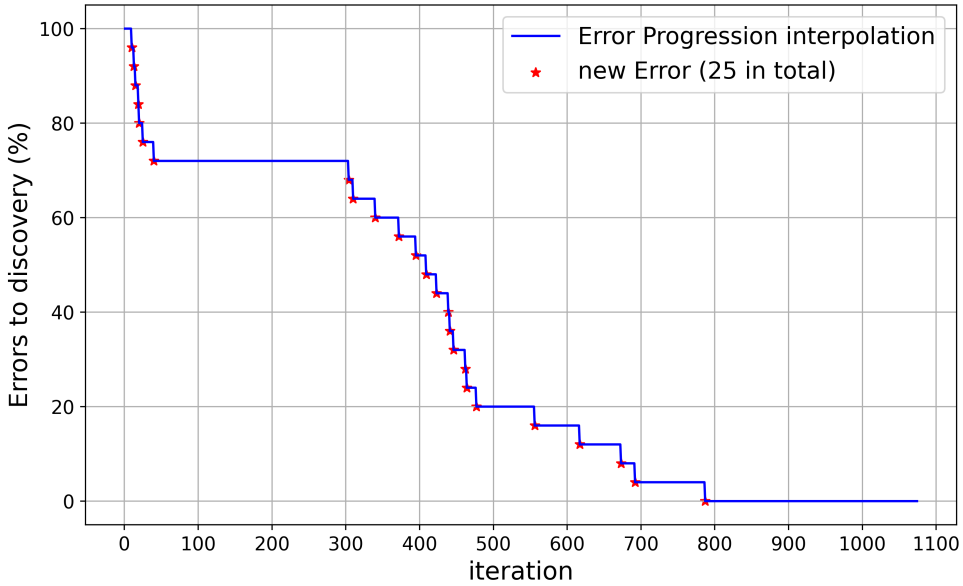

Figure 7.6: CONV Errors alongside spearmint search

in the latency–power projection, the search predominantly identifies points located close to the already discovered clusters.

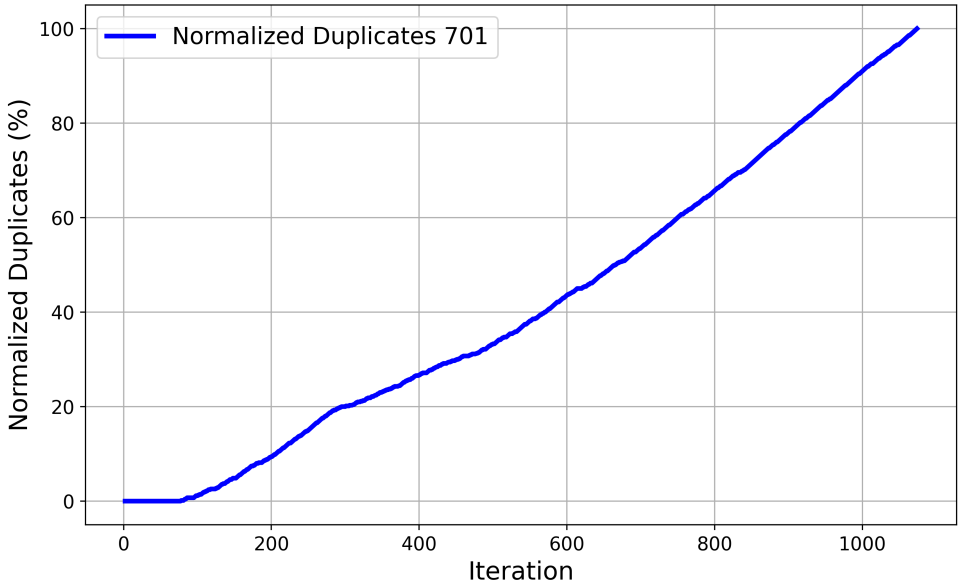

Figure 7.7: CONV Duplicate designs

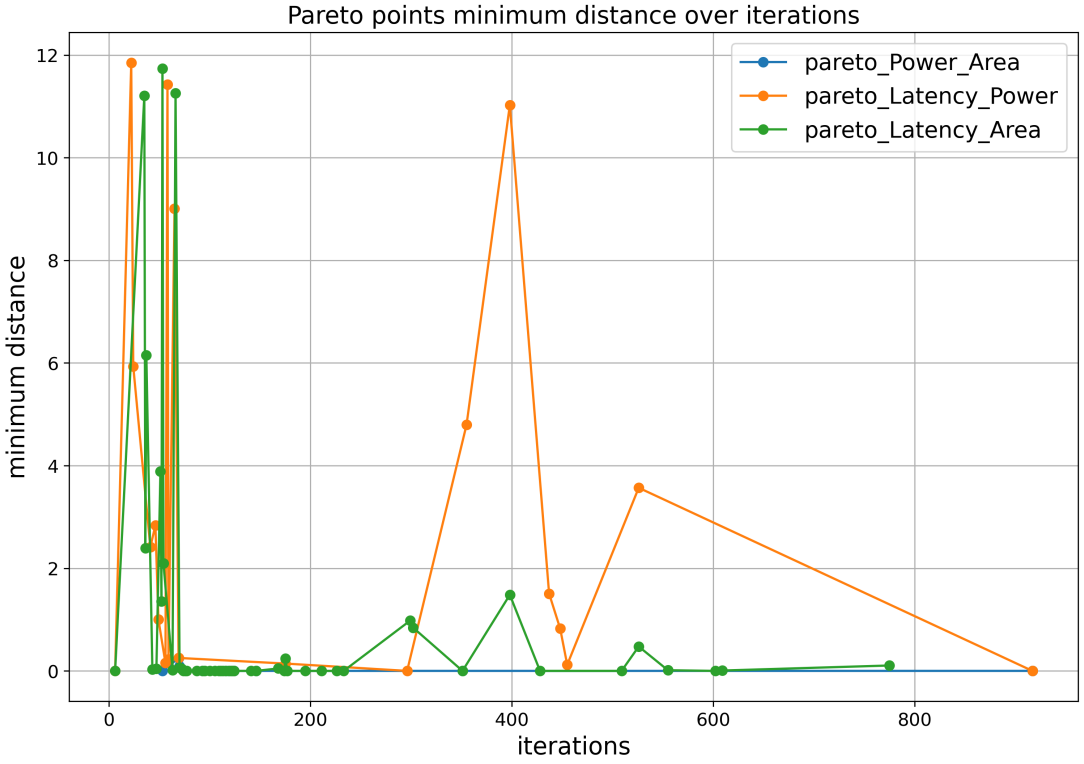

Figure 7.8: CONV, Min Distance

Chapter 8

CONCLUSION

Purpose of the Study

Given the long design and fabrication times of ASIC chips, there is a pressing need to improve productivity. One promising solution is the adoption of frameworks such as Spearmint [4], which leverage Bayesian optimization to efficiently identify high-performing hardware configurations without the need for exhaustive grid search. These frameworks represent a significant step forward in bridging the gap between algorithm development and physical deployment.

Designing ASICs is a complex task that involves balancing trade-offs among area, power, and performance. The design space is vast, highly non-linear, and characterized by complex interactions between parameters, making exhaustive evaluation of all possible solutions infeasible. Several optimization frameworks have been developed to address this challenge, among which Spearmint [4] stands out as a widely used tool for minimizing the cost of black-box functions. Bayesian optimization techniques enable efficient convergence toward optimal solutions without the need for exhaustive simulations, making them particularly suitable for complex ASIC design spaces.

The design of integrated circuits through high-level synthesis (HLS) involves a large number of possible design parameters, or *input knobs*, such as frequency, number of input/output channels, and loop configurations. Exploring this space manually is time-consuming, costly, and prone to suboptimal outcomes, as the impact of individual choices is difficult to predict without extensive trial and error. Moreover, while EDA frameworks provide detailed synthesis and implementation reports, leveraging this information effectively in an iterative optimization loop remains non-trivial.

The purpose of this work is to address these challenges by automating the EDA flow from C code to synthesized hardware. The proposed framework integrates design-space exploration with Bayesian optimization via Spearmint, systematically analyzing synthesis reports and feeding them back into the optimization process. This approach converges toward efficient hardware implementations while reducing exploration time and computational resources compared to traditional grid-search methods, enabling more effective identification of Pareto-optimal solutions and relieving designers from manual parameter tuning.

Contributions

The proposed framework provides several key contributions to the process of hardware generation from high-level descriptions.

First, it automates the complete flow from high-level synthesis (HLS) to RTL and ultimately to ASIC implementation, minimizing manual intervention and simplifying the design process.

Second, the framework significantly reduces the time required for design-space exploration compared to exhaustive or random search strategies. By leveraging guided optimization techniques, it efficiently navigates the vast configuration space and converges toward promising solutions with fewer iterations.

Third, the methodology facilitates the automatic identification of optimal trade-offs among latency, area, and power consumption. Instead of relying on heuristic or ad hoc decisions, the optimization process systematically balances competing objectives, yielding hardware implementations that are both efficient and well-adapted to application requirements.

Fourth, the approach is inherently portable and applicable to a wide range of input C codes, making it suitable for diverse accelerators, including machine learning kernels, digital signal processing modules, and other compute-intensive tasks.

Fifth, the framework enables data-driven decision making by incorporating EDA reports directly into the optimization loop. The tool collects, processes, and exploits this feedback to refine parameter selection, closing the loop between design exploration and synthesis.

Additional Context

While many related studies focus on the joint optimization of neural network architectures and their corresponding hardware implementations, this thesis primarily addresses the application of Bayesian Optimization (BO) as a tool for efficient design-space exploration on a fixed hardware structure. The aim is to implement a solution similar to these approaches, but using a **reconfigurable ASIC accelerator** instead of an FPGA with an embedded neural network. Both approaches share the common challenge of identifying the best-performing parameters for the configurable solution and subsequently extracting the results. To address this, **Spearmint**, a state-of-the-art optimization software, has been adopted and integrated into the workflow.

This thesis merges two lines of research conducted in the group of Prof. Casu at Politecnico di Torino. In particular, it continues the work of Urbinati and Casu [1], who developed novel hardware accelerators for mixed-precision quantized deep neural networks **(MPQ)**. MPQ optimizes inference efficiency by varying bitwidths across DNN layers, requiring hardware innovations such as **Precision-Scalable (PS) multipliers**.

While the work of Urbinati and Casu provides an effective foundation, its original design-space exploration (DSE) methodology was limited in scope, relying on a grid search over a narrow configuration range with a restricted set of tunable parameters. This led to gaps in the generated Pareto fronts (e.g., area vs. latency or power vs. latency), leaving unexplored but potentially promising regions in the design space.

To overcome these limitations, this thesis introduces an automated DSE framework based on BO and implemented via Spearmint. This approach replaces static, brute-force exploration with a feedback-driven optimizer that iteratively adapts based on prior synthesis results. The improved pipeline enables:

- Exploration of a broader search space, including extended parameter ranges and a larger number of tunable knobs (e.g., pipelining, loop unrolling factors, memory banking);
- Guided selection of configurations using synthesis feedback to avoid infeasible or suboptimal solutions;
- Faster convergence towards Pareto-optimal configurations, requiring fewer synthesis runs;

- Scalability, computational efficiency, and adaptability for complex, heterogeneous design spaces.

Finally, the complete toolchain, including automated DSE scripts and synthesis infrastructure, is made publicly available to support reproducibility and further research contributions.

Results

The results obtained from the proposed framework demonstrate its effectiveness along several dimensions. The validity of the data was confirmed by cross-checking synthesis reports generated by the EDA tools with the predicted outcomes of the optimizer. The consistency between these sources indicates that the feedback loop is reliable and robust, ensuring that subsequent optimization steps are grounded in accurate performance and resource measurements.

The framework also showed a clear ability to identify near-optimal configurations within the design space. By leveraging multi-objective Bayesian optimization, the system converged towards Pareto-efficient solutions, capturing trade-offs between area, latency, and power consumption. The tool consistently discovered configurations that would otherwise be difficult to obtain through intuition or ad hoc exploration.

Additionally, the approach resulted in a substantial reduction of exploration time compared to exhaustive search. The optimizer required significantly fewer synthesis iterations to reach competitive solutions, effectively cutting down the computational cost of design-space exploration.

Analysis of the optimizer's choices confirmed the relevance of specific parameters. By examining which input knobs were most frequently selected in high-performing configurations, the tool provided insights into the parameters with the strongest impact on design quality.

Finally, the overall validation of the methodology was limited by practical constraints. A complete assessment would require an exhaustive grid search to evaluate the absolute quality of discovered points and determine the number of iterations needed to reach them. Nevertheless, the results indicate that the tool is capable of identifying high-quality configurations efficiently, demonstrating its potential as a reliable optimization framework and a useful design aid for hardware engineers.

Limitations and Observations

While the proposed framework demonstrates significant potential in automating design-space exploration for HLS-generated hardware, several limitations should be considered. Complete validation would ideally require an exhaustive grid search, which is impractical due to the combinatorial explosion of possible parameter configurations. Therefore, it is not possible to definitively quantify the absolute optimality of the solutions or the exact convergence rate of the optimizer.

The methodology relies on Bayesian optimization through Spearmint, which assumes that the objective function is smooth and can be effectively modeled by a Gaussian process. In highly irregular or discontinuous design spaces, these assumptions may limit the optimizer's ability to explore efficiently, potentially leaving some high-quality configurations undiscovered. Although the framework reduces the number of synthesis iterations compared to brute-force exploration, each HLS synthesis remains computationally expensive, which can limit scalability for very large designs or high-dimensional parameter spaces.

While the framework is domain-agnostic, its effectiveness may vary depending on the characteristics of the input C code and the complexity of the synthesized hardware. Some irregular or application-specific designs may require additional tuning of optimizer settings or occasional manual intervention. Analysis of the optimizer's behavior reveals that certain input knobs consistently have a stronger influence on performance and resource trade-offs, suggesting that a more targeted exploration of these parameters could further improve efficiency.

Despite these limitations, the framework offers clear advantages over manual exploration by providing a data-driven, automated approach to identifying efficient hardware configurations. The observations gathered throughout the study provide guidance for refining future iterations of the tool and improving design-space exploration strategies.

Future Work and Prospects

Several directions can be envisaged to extend and improve the proposed framework. One natural progression is the integration of additional optimization techniques beyond Bayesian optimization, such as evolutionary algorithms or reinforcement learning, to enhance exploration of highly irregular

or high-dimensional design spaces. Additionally, testing the framework with more widely documented and parametrizable tools, such as **Optuna**[10] or **BoTorch**[11], could provide greater flexibility and potentially improved performance. These tools allow a stepwise approach, enabling early-stage evaluation by stopping the design flow at intermediate stages, such as high-level synthesis (HLS) or logic synthesis. This allows designers to obtain preliminary estimates of area, latency, or other design metrics without completing the full synthesis at every iteration, thereby reducing computational cost. Moreover, this stepwise approach supports partial optimization: individual metrics can be targeted sequentially—for instance, area and timing can be optimized first, while power, which is evaluated at the final stage, can be optimized last. Combining multiple strategies in this manner increases robustness and can reveal configurations that might be overlooked when using a single-step approach. Overall, this methodology leverages the staged execution of different steps and the characterization of individual objectives to efficiently guide the optimization process.

From a usability perspective, future work could focus on enhancing the interface and visualization of results, providing designers with interactive tools to examine parameter sensitivities, understand trade-offs, and directly compare candidate configurations. Moreover, the exploration chain could be expanded by integrating additional EDA and analysis tools to allow more comprehensive evaluation of hardware metrics.

Another important direction is the improvement of power estimation and modeling. Incorporating specialized tools for accurate power analysis would enable the generation of realistic input parameters and more reliable predictions of energy consumption, further enhancing optimization outcomes.

Finally, the methodology could be applied to a broader range of applications beyond traditional ML or DSP kernels, including cryptography, signal processing, and heterogeneous computing, validating its generality and revealing additional opportunities for optimization.

In summary, these prospective developments aim not only to improve the efficiency, accuracy, and flexibility of the optimization process but also to enhance the practical utility of the framework as a comprehensive tool for automated hardware design.

Bibliography

[1] L. Urbinati and M. R. Casu, "High-Level Design of Precision-Scalable DNN Accelerators Based on Sum-Together Multipliers," *IEEE Access*, 2024.

[2] M. A. Mansoori and M. R. Casu, "Multi-objective Framework for Training and Hardware Co-optimization in FPGAs," *Department of Electronics and Telecommunications, Politecnico di Torino*, Italy.

[3] P. I. Frazier, "A Tutorial on Bayesian Optimization," July 10, 2018. [Online]. Available: `https://arxiv.org/abs/1807.02811`

[4] E. C. Garrido-Merchán and D. Hernández-Lobato, "Predictive Entropy Search for Multi-objective Bayesian Optimization with Constraints," *Neurocomputing*, vol. 361, pp. 50–68, 2019.

[5] R. P. Adams, "A Tutorial on Bayesian Optimization for Machine Learning," School of Engineering and Applied Sciences, Harvard University, 2012. [Online]. Available:

[6] L. Yang, Z. Yan, M. Li, H. Kwon, L. Lai, T. Krishna, V. Chandra, W. Jiang, and Y. Shi, "Co-exploration of Neural Architectures and Heterogeneous ASIC Accelerator Designs Targeting Multiple Tasks," in *Proc. 57th ACM/IEEE Design Automation Conference (DAC)*, IEEE, 2020, pp. 1–6. [Online]. Available: `https://doi.org/10.1109/DAC18072.2020.9218640`

[7] H. Kuang and L. Wang, "Multi-objective Design Space Exploration for High-Level Synthesis via Bayesian Optimization," in *Proc. 2023 International Symposium of Electronics Design Automation (ISEDA)*, IEEE, pp. 150–155, 2023. [Online]. Available: `https://doi.org/10.1109/ISEDA57803.2023.10288521`

[8] X. Zhang, Y. Li, J. Pan, and D. Chen, "Algorithm/Accelerator Co-design and Co-search for Edge AI," *IEEE Transactions on Circuits and Systems II: Express Briefs*, vol. 69, no. 7, pp. 3064–3070, 2022. [Online]. Available: `https://doi.org/10.1109/TCSII.2022.3147954`

[9] N. Sinha, P. Rostami, A. E. R. Shabayek, A. Kacem, and D. Aouada, "Multi-Objective Hardware Aware Neural Architecture Search using Hardware Cost Diversity," *arXiv preprint arXiv:2404.12403*, 2024. [Online]. Available: `https://arxiv.org/abs/2404.12403`

[10] T. Akiba, S. Sano, T. Yanase, T. Ohta, and M. Koyama, "Optuna: A Next-Generation Hyperparameter Optimization Framework," in *Proceedings of the 25th ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD '19)*, pp. 2623–2631, 2019. [Online]. Available: `https://github.com/optuna/optuna`

[11] M. Balandat, B. Karrer, D. R. Jiang, S. Daulton, B. Letham, A. G. Wilson, and E. Bakshy, "BoTorch: A Framework for Efficient Monte-Carlo Bayesian Optimization," *Advances in Neural Information Processing Systems (NeurIPS 33)*, 2020. [Online]. Available: `https://github.com/pytorch/botorch`

[12] Francesco Ilacqua, "Tool-Chain for Automated Design-Space Exploration with Spearmint," [Online]. Available: `https://github.com/francesco-ila/tool-chain-spearmint`