Dovado: RTL design space exploration with Vivado

Daniele Paletti

October 18, 2020

Abstract

The RTL design space is the set of parametrized design points of a given RTL module. Hardware description languages such as VHDL and (System)Verilog are used to encode RTL designs, parametrization is supported through generics in VHDL and parameters in (System)Verilog. Different parameter values yield different utilization-frequency trade-offs, hand tuning is not feasible with a non trivial amount of parameters.
We devised Dovado as a CAD tool for RTL design space exploration. The tool takes as input a VHDL/(System)Verilog module and returns either the value of a specific design point or the non-dominated set of design points in the design space and the corresponding design values. Each design point is evaluated by programmatically interacting with the Vivado Design Suites and parsing synthesis/implementation reports. Optimization is carried out with NSGA-II and an approximated fitness function. Fitness approximation leverages a varying threshold through which we select wether to call Vivado or a Nadaraya-Watson Model which estimates the design value of points sufficiently similar to already evaluated points.

1 Introduction

The Register Transfer Level (RTL) defines input–output relationships in terms of dataflow operations on signals and register values [1]. Hardware Description Languages (HDLs) such as VHDL and Verilog/SystemVerilog (V/SV) are used to encode a RTL design. Such languages give the designer the possibility of specifying parameters for each RTL module. The volume of the parameter space is exponential in the number of parameters so hand tuning those parameters may be unfeasible even on a low-dimensional parameter space. A designer is often interested in optimizing the area-frequency

trade off by tuning module parameters. Area utilization and frequency are computed by synthesis and implementation tools on a specific design point. Logic synthesis is the translation of the RTL design into a representation in terms of logic gates provided by a target board. From the logic representation implementation comprises placement and routing on a target board .
Dovado [2] is a command line interface (CLI) tool which programmatically interacts with a commercial hardware design suite to provide single design point evaluation and automatic design space exploration. The tool takes as input a target board and parametrized module specified either in VHDL or in V/SV. In single point evaluation mode the tool returns what we call the design value which is the set of utilisation metrics for the design point and the maximum frequency achievable, utilization metrics are percentage occupation indices for the resources provided by the selected board, e.g. the user may select the percentage utilisation of lookup tables (LUTs) as one of the utilisation indices. In design space exploration mode we iterate single point evaluation with several optimization strategies to compute the non-dominated set of design points that is the set of points such that no point exist among the solutions which is better off on all objectives.
Dovado has been tested on a simple RISC-V implementation, a field programmable gate array (FPGA) based network interface controller (NIC) and a simple vector adder.

2 Related Work

Kao et al. [3] used a commercial synthesis and implementation tool to tackle RTL design space exploration formulating it as a module selection problem by determining all possible design implementation with different Area-Time curves. Modules contain either combinatorial or sequential logic but they are not considered singularly, the area-time curve is assessed on the whole system.
Hammerquist et al. [4] devised the concept of application specific FPGAs (AFPGAs) to bridge the gap between FPGAs and Application Specific Integrated Circuits (ASICs) which are much more expensive in terms of time and money to develop and implement. The exploration problem is defined as a search among architectural features to be customised, for example we might consider 3-input 4-input and 5-input LUTs making parametrization happen at the architectural level.
Karakaya et al. [5] tackle the problem of finding an efficient algorithm for design space exploration. The problem is treated as a integer single-target

optimization problem having as a target the power-delay-area product. Problem is solved through an heuristic which shows very good performance when parallelised.
So et al. [6] estimate the synthesis process through behavioral synthesis highlighting the prohibitive cost of running a large amount of hardware synthesis. the problem of design space exploration is further extended by using a compiler approach and taking into account not only parametrisation but also code transformation techniques usually employed by designers when exploring the broader design space.
Pilato et al. [7] examine the relevance of evolutionary algorithms, especially NSGA-II. Moreover the computational cost of a full synthesis or implementation is addressed by using an inheritance model which lowers the number of actual fitness evaluations by inheriting the fitness from the parents in a portion of the population.
In order to identify bottlenecks in RTL designs several other methods are taken into consideration both in practice and in literature. Visual performance models such as LogCA[8] and Roofline[9] for example offer a completely different approach that may be a valuable tool for designers together with design space exploration tools.

3 Dovado

Dovado is a python CLI tool built on top of the Vivado Design Suite [10] (Vivado from now on).
Tool inputs:

- work directory for the file containing the module to be analysed
- module identifier
- target board
- stop step: the tool may either calculate metrics after synthesis or after implementation
- parameter identifiers
- clock port identifier
- the flow to use for synthesis/implementation: it could be either default or incremental

- directives for synthesis/implementation
- target clock for the design
- Dovado mode: either point evaluation or design space exploration
- parameter range or value depending on the mode, space exploration in the former case and point evaluation in the latter case
- utilisation metrics to take into consideration, e.g. LUT occupation, RAM block occupation ...

Tool outputs:

- single point evaluation: returns a design value which is a list of mappings between utilisation metrics and actual values, the maximum frequency is also reported; the actual values are extracted either from synthesis or from implementation depending on the stop step
- state space exploration: returns the non-dominated set of design points and design values

3.1 Single point evaluation

In single point evaluation we take a module, set some parameters and retrieve the values for the utilisation metrics and the maximum frequency. Maximum frequency is computed using the worst negative slack (WNS) through the following formula:

$$\text{max frequency} = \frac{1000}{(\frac{1}{1000} * clk) - wns} \tag{1}$$

where:

- clk: user-defined target clock (in mega-hertz), usually set so that the design cannot meet it
- wns: time interval calculated on the path which takes the longest for the signal to traverse (in nano-seconds), the slack is considered negative when the timing constraint is violated

3.1.1 Parsing

Parsing is a major part of the job when automating the design flow. In our case we are interested in parsing only the declaration section and in particular the module name, the parameter declaration section and the port/signal declaration section. VHDL and V/SV are non contextual languages thus regular expressions are not sufficient for correct parsing. Strictly speaking a non contextual language may have regular sub-languages, in our specific case the declaration section is regular. The issue arises with standards, VHDL and V/SV both supports different standards, in particular V/SV support very different declaration styles for ports and parameters thus parsing with regular expressions would need handling all these cases on their own yielding an explosion in parsing complexity. Antlr [11] is a well known parser generator which has Java, C++ and Python runtimes, moreover Antlr already has VHDL2008 and V/SV grammars fully specified following Antlr convention. A wrapper over Anltr which leverages the C++ runtime is HdlConvertor [12] which is a Python library specifically meant for parsing and reverse parsing VHDL and S/SV. Parsing is needed for identifying modules, ports and parameters while reverse parsing is used for setting parameter values by manipulating the Abstract Syntax Tree (AST).

3.1.2 Boxing: dealing with pin overflow and parsing shortcomings

Generally speaking we might be interested in studying implementation performance of non-top modules, this means in practice that we may overflow the physical pins of the target board at the implementation step. To address such an issue we use a technique which we will call boxing from now on. We create a fictional top module which wraps the real module in a box with a single input in order to avoid overflowing pins. In listing 1 we see the box frame for VHDL modules, the underscores are replaced at runtime with relevant information:

- libraries to be imported
- module name
- parameters
- module clock port
- remaining input ports, attached to constant signals.

```
-- libraries read from the module to be boxed
----
entity box is
  port (
    clk: in std_logic
    );
end entity box;

architecture box_arch of box is
  attribute DONT_TOUCH : string;
  attribute DONT_TOUCH of BOXED : label is "TRUE";
begin
  BOXED:  entity ____
    ----
    port map(
    ____ => clk,
    -- remaining input ports are attached to internal clocks
    ----
    );

end architecture box_arch;
```

Listing 1: Box for VHDL Modules

Boxing is also used in case there are constructs which the reverse parser does not support, in fact the reverse parser may output invalid VHDL or V/SV code upon finding unsupported AST sub-structures. Boxing is generally meant to solve an implementation issue but might be used at the synthesis step in case of parsing issues.

3.1.3 Synthesis and Implementation

Vivado is a comprehensive hardware design suite that we use for hardware synthesis and implementation. Vivado is spawned as a subprocess by Dovado and communication goes through the TCL interface exposed by Vivado in TCL mode [13]. As per the boxing section also in this case we built a frame for the TCL scripts which is then customised at runtime both for the specific module and for the user-selected directives. In listing 2 we see a major section of the implementation frame which is a tcl script to be evaluated by Vivado where the underscores are filled at runtime:

- top module name
- target board
- directive for the synthesis step
- eventual -incremental flag for incremental synthesis
- directive for the place step
- directive for the route step
- name of the timing report
- name of the utilization report.

Directives are flags that can be passed to synthesis, place and route to tune at a high level of abstraction the underlying heuristics by taking more into consideration runtime performance, area or frequency.
The incremental design flow, available both for synthesis and implementation, allows to reuse information from previous runs stored inside checkpoints to speed up the subsequent computations. This approach turns to be particularly useful in cases in which parameters being tuned influence only a small subsection of a larger design.

```
# Run synthesis and write checkpoint
#! read_checkpoint -incremental $outputDir/post_synth.dcp
synth_design -top ____  -part ____ -directive ____
write_checkpoint ____ -force $outputDir/post_synth.dcp

# Run implementation
opt_design
#! read_checkpoint -incremental ____ $outputDir/post_place.dcp
place_design ____

# Optimizations in case of timing violations
if {[get_property slack [get_timing_paths -max_paths 1 -nworst 1 -setup]] < 0} {
puts "found setup timing violations => running physical optimization"
phys_opt_design
}

write_checkpoint -force $outputDir/post_place.dcp

#! read_checkpoint -incremental ____ $outputDir/post_route.dcp
route_design ____
write_checkpoint -force $outputDir/post_route.dcp

report_timing -no_header -file $outputDir/____
report_utilization -no_primitives -omit_locs -format xml -file $outputDir/____
```

Listing 2: Section of the TCL Implementation Frame

3.2 Design Space exploration

In design space exploration we take a module, set ranges in which some parameters vary and retrieve the non dominated set of design points given their design values.

3.2.1 Multi-Objective Optimization

We formulate the design space exploration problem as a multi-objective integer optimization problem. We say it is multi-objective because we optimize both area and frequency at the same time considering we may have more than one utilization metric for the area. It is an integer optimization problem because we take into account only integer-valued parameters which are synthesizable both in VHDL and V/SV. Real-valued parameters are not synthesizable thus are not taken into account. Vector valued parameters do not have a unique ordering over the space of possible vectors, one might choose as an order the value encoded by the binary vector in some base, usually base 2, but this would be the same as considering an integer-valued parameter. Boolean parameters are integer valued parameters with a range between 0 and 1.
The optimization problem is solved through NSGA-II [14] which is a genetic algorithm capable of solving multi-objective constrained optimization problems. Hyper and Meta parameters for the genetic algorithm are the following:

- sampling: integer random sampling
- crossover: integer simulated binary crossover [15]
- mutation: mutation occurs with an approximately gaussian distribution with mean at 0.5 and variance controlled by a hand-tuned parameter
- duplicates are eliminated.

The actual implementation is delegated to a novel python library for multi-objective optimization called pymoo [16]

3.2.2 Incremental Synthesis and Implementation

Vivado other than the default design flow for synthesis and implementation allows the incremental design flow. In listing 2 we see an example of such flow in that we write some archives called checkpoints, similar to zip archives,

and on the second run we remove the `#!` before `read_checkpoint` in order to use the information gained through the previous run to speed up the computation in case enough information is reusable in the current design. Incremental flow may be independently switched on/off for both synthesis and implementation, which is comprised of the place part and the route part. This technique may be particularly useful on large design where small subset of the design is parametrised enabling Vivado to avoid repeating useless computation on non-parametrised sections.

3.2.3 Fitness Function Approximation

A critcal aspect of genetic algorithms is the design of efficient yet representative fitness functions. Naively we would just need to create a vector of functions where each function returns the value of a given metric for a given design point, e.g. `[utilisation(point_1) frequency(point_1)]`. This would mean calling Vivado at each iteration and complete the synthesis or the implementation flow making the exploration infeasible for non-trivial modules.
In order to reduce computational complexity we chose to reduce the calls to Vivado by using a non parametric statistical model which would estimate the actual values for design points sufficiently close to design points that Vivado already evaluated. The method is inspired by the one developed by Shokri et al. [17] for reducing calls to a computationally expensive physics simulator. Algorithm steps:

1. generate a dataset of size n by making n distinct calls to Vivado with design points randomly sampled from the user defined parameter intervals

2. train/validate the statistical model on the synthetic dataset

3. for each new design point explored by the genetic algorithm:

    - if the design point is already in the dataset: call Vivado directly (Vivado answers are cached so the answer needs only to be retrieved from memory, it is not recomputed)
    - else if the design point is sufficiently similar to a point already in the dataset: call the statistical model and estimate the design value of the design point
    - else: call Vivado directly, add the new pair (design point - design value) to the dataset and re-train/re-validate the statistical model

The similarity measure for a new design point is exactly the one used in [17]

$$D_n = \sqrt{\frac{\sum_{j=l}^{m}(x_j - z_j^n)^2}{m}}$$

- D_n: the similarity measure
- j: decision variables counter
- m: number of decision variables (dimensionality of the design point)
- x_j: j-th decision variable of the new design point
- z_j^n: j-th decision variable of the n-th nearest solution from the training archive.

Given this similarity measure we set a threshold and when the threshold is exceeded Vivado is called while in the other case the estimator is called.
Setting the threshold is a hard problem per se infact it depends on parameter ranges, that is run-time information, thus we cannot set an hard threshold but we need an adaptive one. The threshold is set by averaging the distance between the points of the dataset and updating such threshold every time a new example is added to the dataset.
The non parametric regression model chosen is the Nadaraya-Watson Model (NWM) [17]. The choice of a non parametric method is due to the fact that the dataset changes frequently and re-training everytime would be very expensive. The validation step is simplified as well because Shapiai et al. [18] showed how the NWM model performs best with the Gaussian Kernel when an appropriate choice of the parameter is carried out, this means that only the bandwidth needs to be selected as a kernel parameter. Validation is done through Leave One Out cross-validation because the size of the dataset is low and the NWM is computationally very cheap. On top of this a low variance model was needed because of the limited size of the dataset, which would have lead to overfitting with more complex models such as the Neural Network which was employed by Shokri et al. [17]. The model is loosely speaking a weighted average of the points in the dataset where the weights are defined by a gaussian Kernel function

$$\hat{y} = f(x, D, h) = \frac{\sum_{i=1}^{n} y_i K_h(x, x_i)}{\sum_{j=1}^{n} K_h(x, x_j)}$$

where:

- D denotes the dataset
- $\hat{y}$ is the estimated design value
- x is a design point
- $K_h(x, x_i)$ is the kernel function with bandwidth h:

$$K_h(x, x') = \frac{1}{\sqrt{2\pi}} exp(-(x - x')^2 2h^2)$$

4 Experimental Results

All the results below can be replicated by using the corresponding input files found in the `examples/input_files` folder in the git repository [2].

4.1 Vector Adder

(input file: `input_rtl_vadd.txt`) Point evaluation is tested through a vector adder taken from a set of Vitis example designs [19]. This example shows how Dovado may be used as a layer of abstraction over Vivado for implementation in fact we are taking the vector adder and we are able to tune in an interactive way which parameters we want to modify, what directives we want to give to the various steps and in particular what kind of utilisation metrics we want to use among the ones allowed for the board we have chosen. In this experiment we chose among the utilisation metrics available in listing 3 to measure the percentage occupation of: Configurable Logic Blocks (CLB) LUTs, Ultra Random Access Memory (URAM) and Digital Signal Processors (DSPs). On top of utilization metrics we always have the (negated) max frequency computed as in equation 1. The design point in this case is trivial, in that we chose as a free parameter to set `C_S_AXI_CONTROL_DATA_WIDTH` and we set it to the value which is specified as a default (32). While this might seem a totally trivial operation it is not in that we actually need boxing in order to handle more general cases in which the parameter does not coincide with the default value, so the example is totally general infact the series of operations carried out to solve this example would have been the same even if the value did not coincide with the default value. Results are shown in listing 5 for the design point described above. In the utilisation section we see a pair where the second item is the entry we chose above while the first is the name of the paragraph which contains it in the Vivado utilisation report, such detail is needed because for some boards there are entries with the same name under

```
CLB Logic
(1) CLB LUTs
(2) LUT as Logic
(3) LUT as Memory
(4) CLB Registers
(5) Register as Flip Flop
(6) Register as Latch
(7) CARRY8
(8) F7 Muxes
(9) F8 Muxes
(10) F9 Muxes
CLB Logic Distribution
(11) CLB
(12) LUT as Logic
(13) LUT as Memory
(14) CLB Registers
(15) Unique Control Sets
BLOCKRAM
(16) Block RAM Tile
(17) RAMB36/FIFO*
(18) RAMB18
(19) URAM
ARITHMETIC
(20) DSPs
```

Listing 3: Partial Set of Utilisation Metrics for a Zynq UltraScale+ Board

```
module krnl_vadd_rtl #(
  parameter integer  C_S_AXI_CONTROL_DATA_WIDTH = 32,
  parameter integer  C_S_AXI_CONTROL_ADDR_WIDTH = 6,
  parameter integer  C_M_AXI_GMEM_ID_WIDTH = 1,
  parameter integer  C_M_AXI_GMEM_ADDR_WIDTH = 64,
  parameter integer  C_M_AXI_GMEM_DATA_WIDTH = 32
)
```

Listing 4: Parameter Declaration of the Vector Adder Module

```
DesignValue(utilisation={
    ('CLB Logic', 'CLB LUTs'): 0.46,
    ('ARITHMETIC', 'DSPs'): 0.0,
    ('BLOCKRAM', 'URAM'): 0.0},
    negative_max_frequency=-332.22591362126246)
```

Listing 5: Adder Single Point Implementation

different paragraphs. The pair paragraph-entry is mapped to a percentage utilisation value, in our case the design is very small thus it only occupate 0.46% of the available LUTs and does not use neither URAM blocks nor DSPs blocks.

4.2 Potato: a RISC-V Implementation

(input file: `input_potato_design.txt`) Potato [20] is a simple RISC-V processor for use in FPGA design. We want to explore the design space of the top module `pp_potato` in listing 6 by setting as free parameters `ICACHE_LINE_SIZE` (number of words per instruction cache line), `ICACHE_NUM_LINES` (number of cache line in the instruction cache) and using several utilisation indices:

- Slice LUTs
- Slice Registers
- F7 Multiplexers
- F8 Multiplexers
- Block RAM Tile
- DSPs.

Visualisation is an important step of multi-objective optimization, in this case the plots are trivial because the set of non dominated solutions all have the same objective value. In picture 1 we see that `ICACHE_LINE_SIZE` can only have value 4 while different values are admissible for `ICACHE_NUM_LINES`.

Looking at the graph in 2 we clearly see that all the solutions are equal in target value meaning that any of the found design points on the metrics we chose gives the same result

```
entity pp_potato is
        generic(
                PROCESSOR_ID            : std_logic_vector(31 downto 0) := x"00000000";
                RESET_ADDRESS           : std_logic_vector(31 downto 0) := x"00000000";
                MTIME_DIVIDER           : positive                      := 5;
                ICACHE_ENABLE           : boolean                       := true;
                ICACHE_LINE_SIZE        : natural                       := 4;
                ICACHE_NUM_LINES        : natural                       := 128
        );
```

Listing 6: Parameter Declaration of the Vector Adder Module

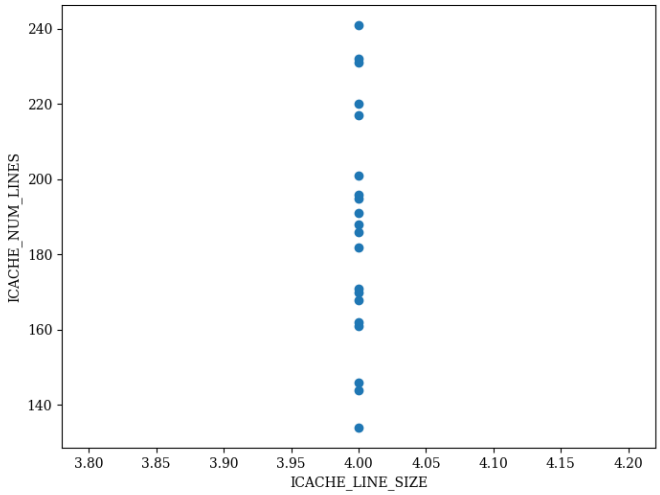

Figure 1: Non-dominated design points for the Potato design

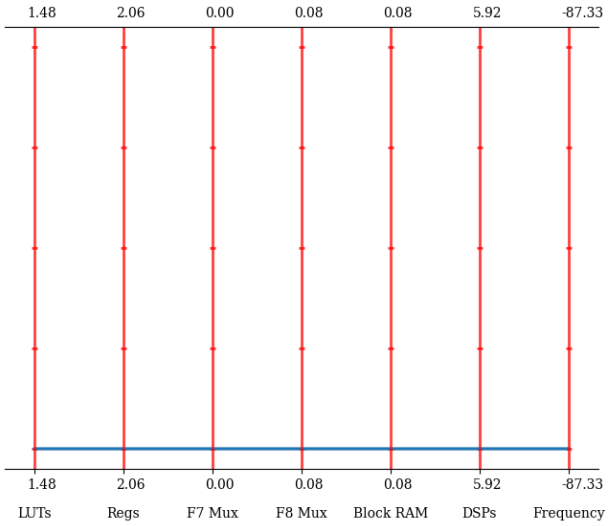

Figure 2: Non-dominated design values for the Potato design

4.3 Corundum: a High Performance NIC

Corundum is a high performance fpga-based network interface controller. In this case we are not studying the top module but the queue manager alone. This example has the peculiarity of having very few design points which would make synthesis end without an error. In this scenario, the NWM has been put to great pressure because the dataset would be at times comprised of data points all with the same design value, due to the fact that Vivado returned an error. In listing 7 we see the three parameters set as free:

- `OP_TABLE_SIZE`: number of outstanding operations
- `QUEUE_INDEX_WIDTH`: log2 of number of queues
- `PIPELINE`: pipeline stages

The utilization metrics considered in this case are:

- Slice LUTs
- Slice Registers
- Block RAM Tile

```
module queue_manager #
(
    parameter ADDR_WIDTH = 64,
    parameter REQ_TAG_WIDTH = 8,
    parameter OP_TABLE_SIZE = 16,
    parameter OP_TAG_WIDTH = 8,
    parameter QUEUE_INDEX_WIDTH = 8,
    parameter CPL_INDEX_WIDTH = 8,
    parameter QUEUE_PTR_WIDTH = 16,
    parameter LOG_QUEUE_SIZE_WIDTH = $clog2(QUEUE_PTR_WIDTH),
    parameter DESC_SIZE = 16,
    parameter LOG_BLOCK_SIZE_WIDTH = 2,
    parameter PIPELINE = 2,
    parameter AXIL_DATA_WIDTH = 32,
    parameter AXIL_ADDR_WIDTH = 16,
    parameter AXIL_STRB_WIDTH = (AXIL_DATA_WIDTH/8)
)
```

Listing 7: Parameter Declaration of the Corundum Queue Manager

- Max Frequency

As we can see in picture 3 solutions are grouped in small far-apart sets, there is a large volume in the middle of invalid design points.

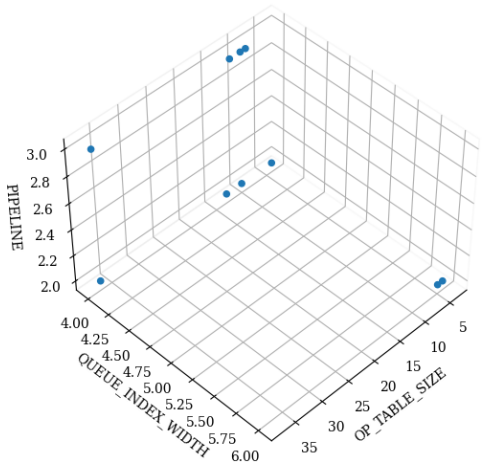

Figure 3: Non-Dominated Design Points for the Potato Design

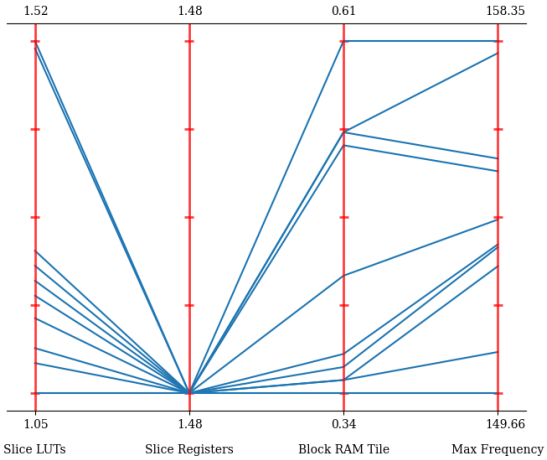

Figure 4: Parallel Coordinate Plot of Non-Dominated Design Values for Corundum Queue Manager

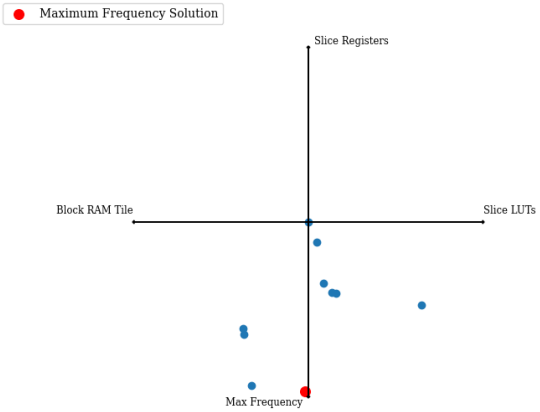

Figure 5: Star Plot of Non-Dominated Design Values for Corundum Queue Manager

Design values in this case are not as regular as in the previous example as we can see in figure 4 and 5 where the same values are plotted but with different plot types. In figure 4 each design value is a line which crosses the vertical axes at the value it takes for the corresponding function, such representation highlights the balance of solutions in all the targets. In figure 5 we have a representation which serves the same purpose but that fits much better this case because the dimensionality being not so high is mapped to a cross in the star plot that is easy to interpret, if we had more target functions we would be having a much more complex plot and thus we would have been better off with a Parallel Coordinates Plot like the one in figure 4.

5 Conclusions

Dovado is still a prototype and needs further testing before being marked as a stable tool but at the current stage the experimental work shows how the tool can give reliable insights about the set of non-dominated design points and their corresponding design values. The tool is conveniently flexible, allows both synthesis and implementation flows, has built-in help commands which retrieve documentation fragments from Vivado and allows tweaking directives and design flows (default/incremental).
The experimental results showed how VHDL and V/SV are handled seam-

lessly and how board choice does not affect the design flow but it only puts a constraint on the available resources for synthesis/implementation. Percentage occupation of resources can be tweaked by the user by choosing only specific resources for which calculating the percentage utilisation, in particular the available resources depend on the board choice and are seamlessly presented to the user.
Much more tests need to be carried out in order to assess the stability of Dovado.

6 Future Work

Many problems are still there to be tackled. First of all a more reliable reverse parsing approach for VHDL and V/SV would sensibly reduce the use of boxing (see paragraph 3.1.2) in that parameters may be replaced in-place directly in the rtl file when stopping at the synthesis step where no issue of pin overflow arises. On top of this, a better reverse parser would also enable to define the design space more broadly in terms of code transformations [6] and parameter optimization so as to optimize the solution even further.
Approaches like Chisel [21] offer a convenient way for designers to use a high-level language for hardware design by transpiling to synthesizable V/SV or VHDL. Such high-level synthesis efforts may be worth investigation [3] so as to be able to support the designer throughout the whole design process and look for optimization chances at every step.
For Dovado the computational cost of a single run is really prohibitive. The tradeoff between accuracy and computational cost is to be studied further in particular refining the ways in which the synthesis/implementation outcome can be approximated without the need of actually running synthesis/implementation. Exploring more statistical models, both parametric and non-parametric, is surely a good way to start but the focus needs also to be placed on the way we build the datasets for such models and how expensive generating a synthetic dataset is. The optimization algorithm is another spot in which there is much room for improvement, many optimization algorithms exist for multi-objective optimization with different performance profiles. A way of selecting at runtime the best performing optimization algorithm given some information collected during the generation of the synthetic dataset may be an approach worth investigating. Sticking to genetic algorithms the actual tradeoff induced by the choice of different operators and encoding strategies is still to be thoroughly evaluated.

References

[1] P. A. Simpson, "RTL Design," in *FPGA Design: Best Practices for Team-Based Reuse*, P. A. Simpson, Ed. Cham: Springer International Publishing, 2015, pp. 91–139.

[2] D. Paletti, "DPaletti/dovado," Oct. 2020.

[3] P.-C. Kao, C.-K. Hsieh, C.-F. Su, and A.-H. Wu, "An RTL design-space exploration method for high-level applications," *IEICE Transactions on Fundamentals of Electronics, Communications and Computer Sciences*, vol. E84-A, no. 11, pp. 2648–2654, 2001.

[4] M. Hammerquist and R. Lysecky, "Design space exploration for application specific FPGAS in system-on-a-chip designs," in *2008 IEEE International SOC Conference*, Sep. 2008, pp. 279–282.

[5] F. Karakaya, "Automated Exploration of the ASIC Design Space for Minimum Power-Delay-Area Product at the Register Transfer Level," *Doctoral Dissertations*, May 2004.

[6] B. So, M. W. Hall, and P. C. Diniz, "A compiler approach to fast hardware design space exploration in FPGA-based systems," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation*, ser. PLDI '02. New York, NY, USA: Association for Computing Machinery, May 2002, pp. 165–176.

[7] C. Pilato, A. Tumeo, G. Palermo, F. Ferrandi, P. L. Lanzi, and D. Sciuto, "Improving evolutionary exploration to area-time optimization of FPGA designs," *Journal of Systems Architecture*, vol. 54, no. 11, pp. 1046–1057, Nov. 2008.

[8] M. S. B. Altaf and D. A. Wood, "LogCA: A high-level performance model for hardware accelerators," in *2017 ACM/IEEE 44th Annual International Symposium on Computer Architecture (ISCA)*, Jun. 2017, pp. 375–388.

[9] S. Williams, A. Waterman, and D. Patterson, "Roofline: An insightful visual performance model for multicore architectures," *Communications of the ACM*, vol. 52, no. 4, pp. 65–76, Apr. 2009.

[10] "Vivado Design Suite User Guide: Getting Started (UG910)," p. 22, 2018.

[11] "ANTLR," https://www.antlr.org/.

[12] M. Orsak, "hdlConvertor: VHDL and System Verilog parser written in c++."

[13] "Vivado Design Suite User Guide: Using Tcl Scripting (UG894)," p. 114, 2018.

[14] K. Deb, A. Pratap, S. Agarwal, and T. Meyarivan, "A fast and elitist multiobjective genetic algorithm: NSGA-II," *IEEE Transactions on Evolutionary Computation*, vol. 6, no. 2, pp. 182–197, Apr. 2002.

[15] K. Deb and R. B. Agrawal, "Simulated Binary Crossover for Continuous Search Space," *Complex Syst.*, 1995.

[16] J. Blank and K. Deb, "Pymoo: Multi-Objective Optimization in Python," *IEEE Access*, vol. 8, pp. 89 497–89 509, 2020.

[17] A. Shokri, O. Bozorg Haddad, and M. A. Mariño, "Algorithm for Increasing the Speed of Evolutionary Optimization and its Accuracy in Multi-objective Problems," *Water Resources Management*, vol. 27, no. 7, pp. 2231–2249, May 2013.

[18] M. I. Shapiai, S. Sudin, Z. Ibrahim, and M. Khalid, "Investigation on Different Kernel Functions for Weighted Kernel Regression in Solving Small Sample Problems," in *2011 UKSim 5th European Symposium on Computer Modeling and Simulation*, Nov. 2011, pp. 64–69.

[19] "Xilinx/Vitis_Accel_Examples," Xilinx, Oct. 2020.

[20] K. K. Skordal, "Skordal/potato," Oct. 2020.

[21] t. C. Developers, "Chisel/FIRRTL: Home," https://www.chisel-lang.org/.