

# Simopt - Simulation pass for Speculative Optimisation of FPGA-CAD flow

Eashan Wadhwa & Shanker Shreejith

Department of Electronic and Electrical Engineering, Trinity College Dublin

Dublin, Ireland

{wadhwa, shankers}@tcd.ie

**Abstract**—Behavioural simulation is deployed in CAD flow to verify the functional correctness of a Register Transfer Level (RTL) design. Metadata extracted from behavioural simulation could be used to optimise and/or speed up subsequent steps in the hardware design flow. In this paper, we propose Simopt, a tool flow that extracts simulation metadata to improve the timing performance of the design by introducing latency awareness during the placement phase and subsequently improving the routing time of the post-placed netlist using vendor tools. For our experiments, we adapt the open-source Yosys flow to perform Simopt-aware placement. Our results show that using the Simopt-pass in the design implementation flow results in up to 38.2% reduction in timing performance (latency) of the design.

**Index Terms**—FPGA, Simulators, Low latency, Optimisation, Electronic Design Automation

## I. INTRODUCTION

Behavioural simulation tools play an integral role in the Electronic Design Automation (EDA) design flow, allowing the design engineer to verify the functional and logical correctness of their design under the same constraints of final implementation at the higher abstraction level. A complete simulation flow generates a tremendous amount of data and can provide insights into the low-level operation of the design to primarily aid in verification and validation; however, these insights are rarely (if ever) used by subsequent steps and tools in the EDA flow. Subsequent steps in the EDA flow typically focus on optimising the performance (or similar user-driven constraint) of the implemented design. In a standard Field Programmable Gate Array (FPGA) design flow, user-generated RTL designs are mapped to gate-level netlists through logic synthesis followed by iterative place and route steps to generate the FPGA bitstream for deployment. A front-end converts a Hardware Description Language (HDL) design into a gate-level Register Transfer Level (RTL) netlist called as the logical synthesis step. This is subsequently passed through various back-ends of the EDA design flow to generate a bitstream which is then flashed onto hardware. The general aim is to always optimise the flow keeping a subset of Key Performance Indicators (KPIs) as their end target. However as effective as this flow is, a lot of optimisation pointers can be inferred from functional simulation to guide the back-end (place and route) tools. This is a common practice in compiler technologies for CPUs and GPUs which typically rely on designated entry points within the codebase to

initiate translation from high-level programming languages to executable machine code. These entry points serve as pivotal anchors for the compiler’s analysis and optimization processes, allowing for targeted optimizations tailored to specific runtime environments. The integration of runtime optimizations further enhances the efficiency and performance of compiled code, dynamically adjusting execution behavior based on runtime conditions and system characteristics. FPGA tools however are unable to leverage runtime optimisations used in CPUs and GPUs due to the following reasons:

- **Hardware execution:** Runtime optimizations in traditional compilers for CPUs and GPUs focus on modifying software instructions or data layouts at runtime to improve performance. However, FPGAs execute hardware descriptions, which are configured before runtime and cannot be modified dynamically during execution like software instructions.
- **Configuration and constraints:** FPGAs are configured with a specific hardware design during configuration time, through a bitstream generated by a synthesis tool and remain fixed (unless there is a reconfiguration). Furthermore, FPGAs require careful resource allocation and optimization during synthesis and place-and-route stages to meet timing constraints and fit within the available hardware resources, unlike CPUs and GPUs with more flexible resource management.

There is also a notable distinction in the approach taken by FPGA tools toward the Intermediate Representation (IR) of HDL code. Any tool-flow pass is done on the IR with the inputs and outputs of an HDL module always being used to the maximum extent. Any dead-code elimination cannot be done on these ports making such a pass not effective on the code. These factors, combined with the fact that behavioural simulation is only limited to functional testing, lead to missed opportunities to perform runtime optimisation. We can visualise this in the code snippet 1. Assume the snippet to be run first to be run for 31’hFOOBA clk cycles and secondly for 31’hFOOB9 clk cycles. Also assume the input ports `in` receive values from the external interface (random conditions), `reset` is always set to high and `clk` keeps toggling edge every cycle. In the first scenario (of 31’hFOOBA’ clk cycles), entity `pointless` is useful as the output port vector (`out`) is written to with a value (at line 16). In the second scenario (of 31’hFOOB9’ clk cycles),

however, even though `pointless` would have no use as the scenario only runs for 31'hFOOB9 cycles, (1 cycle short from the line 15 condition being met). If in the real-world use case, the second scenario is the most frequently evaluated condition and if we want our design to have the highest performance (timing), then the toolflow pass should be able to "speculate and optimise" such pathways in the final design to improve the latency of the datapath.

```

1 // ports:
2 // input[31:0] in, reset, clk;
3 // output[31:0] out;
4 //
5 reg [31:0] count_c; // assume = 0
6 reg [31:0] pointless; // assume = 0
7 always_ff @ (posedge clk) begin
8     count_c <= count_c + 1;
9     if (~reset) begin
10         assign out = in;
11     end
12     else begin
13         assign out = count_c;
14         if (pointless == 31'hFOOBA) begin
15             assign out = 31'hFOOBA;
16         end
17     end
18 end
19 always_ff @ (posedge clk) begin
20     pointless <= pointless + 1;
21 end

```

Listing 1: A simple conditional Verilog model which increments every clock cycle

In this paper, we present Simopt, a framework which uses the metadata generated from a behavioural simulation to speculatively optimise the HDL code to perform latency-driven placement and packing optimisation of user logic. Our specific contributions are as below:

- Develop a framework for capturing important metadata from behaviour simulation, using an open-source simulator, Verilator.
- Integration of the metadata in an open-source placement tool, Yosys, to perform guided placement optimisation
- Benchmark circuits to quantify latency gains that can be achieved using the proposed framework.

We benchmarked the design using compact Verilog circuit models that are openly available through Verilator and EPFL benchmarks to validate the tool and quantify the performance gains. Our results show that using the Simopt-pass in the design implementation flow results in up to 26.1% reduction in timing performance (latency) of the design, and 5-10% speed-up in the routing across multiple test designs.

## II. BACKGROUND

To the best of our knowledge, using behavioural simulation to accelerate a toolflow has not been widely explored in both CAD toolflow and compiler literature. The most analogous representation of this concept found in previous studies is from the FPGA design optimisation using roofline models [?]. The authors used roofline models to highlight the efficient memory access patterns in particle methods [?], wavefront algorithms [?], and sparse arithmetic computation [?] achieving significant performance enhancements. This however was



Fig. 1: Traditional Verilator flow from a Verilog HDL to a C++ behavioural simulation. Verilator coins this compilation flow as *verilate*.

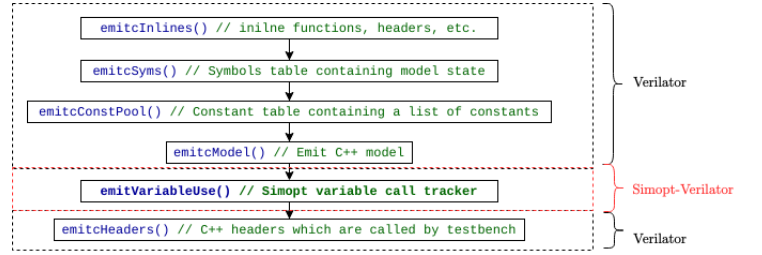


Fig. 2: Emission of the Verilator AST to C++ behavioural simulation steps

very specialised to neural networks and used an LLVM IR to generate the best directives for their roofline models and does not adapt them into or use them as proxies to guide existing optimisation passes available in the EDA toolflow chain to generate a correspondingly optimised hardware. Similar works have been proposed in [?] [?], where an inference algorithm generates an equivalent to an FPGA/ASIC design; however, these are domain-specific and rarely make any changes to the CAD toolchain. Whereas on the other side in the spectrum of academic literature, authors of [?] devised a speculative compiler optimisation for data analysis which encodes multiple data structure optimisations (such as prioritise highly selective predicates, vectorisation). However this and other works [?] focus mainly on compilers for CPUs and allow these concepts to be integrated into GPU compilers too. In this work we focus on FPGA/ASIC CAD tool flow which takes the metadata of a behavioural simulation as input, to generate a low-latency implementation.

## III. SIMOPT-VERILATOR

The notion of using simulation metadata to guide low-level optimisations is driven by extensive functional simulations that are performed on complex designs. For the development of Simopt, we chose the popular open-source simulator, Verilator [?]. Verilator generates a C++/SystemC behavioural model by compiling a Verilog (a HDL) design into a host-machine compatible executable. To run the simulation - a user needs to provide a C++ written wrapper (containing a standardised `main()`) which is executed by the "simulation runtime". The main flow of the Verilator is shown in Fig. 1.

Verilator uses third-party tools (for lexical analysis and parsing) to generate the Abstract-Syntax Tree (AST). This is then used by the subsequent step of the Verilator flow to link references (such as functions, and variables) to their definitions. After running a few local compiler passes, the AST is pseudo-flattened with each module scoped out. This makes it easier to run the final two steps on the optimised AST which is globally scoped compiler passes and emission of C++ modules. It is in the Verilator step of emission where

the Simopt comes into play. This entire compilation (from Verilog to C++) is termed as the *verilate* process in Verilator.

Emission of the C++ modules is sequenced as shown in Figure 2 illustrating Verilator’s flow. Simopt-Verilator does an addition to the usual flow through the `emitVariableUse()` C++ class. The AST is processed by the Verilator internal emitter which first inlines any scoped-out modules, functions and headers. A symbol table is then generated containing various parameters of the model state (such as model precision, number of statements generated, model initialisation state and others). A constant table is also generated with a data structure of constants that have not been replaced by the constant propagation step (in the earlier Verilator compiler steps). The C++ model is subsequently generated using the emitted tables as a reference.

To integrate Simopt into the Verilator simulator (through `emitVariableUse()`), a set of counters (called *simopt-trackers*) should be integrated which tracks each signal in the Verilog description (single-bit and vectored wire, reg, port types). Tracking each signal is also designed to be optional allowing the designer to choose specific nets/entities or modules/sub-modules for optimisation. This also allows designers to trade-off simulation execution time in Verilator by tracking only the signals/modules of interest. The simopt-counters corresponding to these signals are incremented each time they are activated by the verilated runtime, except for initialisation statements. For example, referring to the code in Listing 1, simopt-counters for all signals within the if-else construct (lines 9, 12) are generated by the tool, if tracking is enabled for them. In a specific case, if the `reset` is asserted as logic HIGH from the start of the simulation, then only the simopt-counters for signals within the `else` condition (lines 12 – 17) are executed. This conditional registered logic creates additional overheads using the above tracking logic, both to keep track of each condition and also to understand the change in state of the registered signal. To tackle this problem, *simopt-state* is introduced which checks for mismatch before and after an operation is applied to a single-bit / vectored register reference to keep track of the signal across multiple branches. Tracking using state flow instead of multiple counters reduces the overhead on the Verilator runtime for registered logic. In the case of vectored register logic, the constraints imposed by the HDL description are accordingly unrolled by verilator’s optimiser. Activations on vectored entities are treated as operations on packed datatypes and are expanded into activity at the bit-level for incrementing the simopt-counters. A pseudocode representation of this *mask-and-increment* logic for tracking *simopt-state* and selective incrementing of *simopt-counters* is shown in algorithm 1.

When the parser in Verilator encounters a signal definition, it statically generates the bit width and type definition of the internal representation of the signal (referred to as *var* in the algorithm). Each *var* generated during the Verilator compilation flow follows the same steps in the *mask-and-increment* algorithm (algorithm 1):

- the value of *var* is checked for dissimilarity with the simopt-state using the XOR ( $\oplus$ ) operator
- the simopt-counter is then incremented if the values do not match
- if the simopt-counter is updated, update the simopt-state with the new value of *var*

In algorithm 1, the function `*.simoptCounter` is a reference to the simopt-counter, and `*.simoptState()` is a reference to the simopt-state of data-type *var*. The call `__builtin_ffsll(mask)` is a GCC compiler intrinsic, which returns the position of the least significant bit that is set to high in the *mask* plus 1. *Packed* vector is a Verilog construct where multiple signals or variables are compactly organized into a contiguous bit sequence. *Unpacked* vectors denote a collection of signals or variables with separate bit representations. The *mask-and-increment* algorithm determines *var* is packable through the `*.isPackable()` boolean which is provided by the Verilator framework. For tracking unpacked entities, the “unpacked value” of *var* is treated as a mask and used for comparing against the *var*’s simopt-state (shown in line 15). Lines 16 to 20 update the simopt-counters and simopt-states of each entity if there has been any change in the *var* value. This greatly reduces the dependencies when using iterations in the design and handles each generated data type (from Verilator corresponding to an HDL signal) as a special case.

Once the Simopt components are generated by the emission stage (second last step in Fig. 2, the rest of the flow remains unchanged. Verilator creates public functions of all the steps, including those for functions that define simopt-counters and simopt-states initialisation. Verilator also creates a public header for the Simopt-enabled flow, which can be included by a testbench for instantiating the C++ model to perform functional validation. When performing functional validation, Simopt-enabled flow invokes the mask-and-increment algorithm for each internal variable in the verilated design (corresponding to the signals in the HDL) referenced by the runtime. Using the same example from listing 1, Simopt-enabled Verilator flow will emit an executable as shown in listing 2.

```

1 reg [31:0] count_c, pointless;
2 always_ff @ (posedge clk) begin
3     count_c <= count_c + 1;
4     maskAndIncrement(count_c);
5     if (~reset) begin
6         assign out = in;
7         maskAndIncrement(in, out);
8     end
9     else begin
10        assign out = count_c;
11        if (pointless == 31'hFOOBA) begin
12            assign out = 31'hFOOBA;
13        end
14        maskAndIncrement(count_c, out, pointless
15    );
16 end
17 maskAndIncrement(reset);

```

Listing 2: Conditional Verilog model with emitted Simopt’s mask-and-increments (function *maskAndIncrement()* covered by algorithm 1 )

**Algorithm 1** Simopt’s mask-and-increment algorithm, wrapped in function *maskAndIncrement()*: Used to track changes in a data type *var*

```

1: function MASKANDINCREMENT(var)
2:   if var.isSingleBit() then ▷ single bit data types
3:     if (var.value()  $\oplus$  var.simoptState())  $\neq$  0 then
4:       unpackedSingleIncrement(var, 0, 0)
5:     end if
6:   else if var.isPackable() then ▷ Packed vectored entities - can be unrolled
7:     for index  $\in$  0 : var.size() do
8:       if (var[index].value()  $\oplus$  var[index].simoptState())  $\neq$  0 then
9:         unpackedSingleIncrement(var, size, index)
10:      end if
11:    end for
12:  end if
13: end function
14: function UNPACKEDSINGLEINCREMENT(var, size, index)
15:   if size == 0 then
16:     var_entity = var
17:   else
18:     var_entity = var[index]
19:   end if
20:   if !(var_entity.isPackable()) then ▷ Unpacked entity, create a mask and increment bitflips only
21:     mask = (var_entity.value()  $\oplus$  var_entity.simoptState())
22:     mask_index = __builtin_ffsll(mask)
23:     while (mask_index  $\neq$  0) do
24:       if mask_index  $\neq$  0 then
25:         var_entity.simoptCounter[mask_index] ++
26:         var_entity.simoptState  $\oplus$  = (1  $\ll$  (mask_index - 1))
27:       end if
28:       mask_index = __builtin_ffsll(mask)
29:     end while
30:   else ▷ Single-bit entity, simple increment will do
31:     var_entity.simoptCounter ++
32:     var_entity.simoptState = var_entity.value()
33:   end if
34: end function

```

Clock signals (clk) are not tracked through Simopt and simopt-counters of such signals are assigned a maximum value (UINT128\_MAX) to assert their importance in the subsequent Simopt-driven placement optimisation stages. Synchronous reset, on the other hand, is tracked by the Simopt logic since its activation has a bearing on the outputs of the system (lines 7 and 12 in listing 2). Once the functional simulation is completed, a dump of all simopt-counters with their corresponding (Verilog) variable names is generated before invoking the destructor function. To enable seamless integration of Simopt framework to any Verilator project, we use protocol buffers, specifically Protobuf due to their lower (de)compression times and memory footprint [?], to handle serialisation of the dump. This enables the encoded dump to be used by other FPGA tools in the CAD flow as shown in figure 3. We discuss specific integration with Yosys in the next section (sec. IV)

#### IV. SIMOPT-BACKENDS

For each design under test (DUT), we generate a Protobuf dump containing simopt-counters for each data type. Vectored entities (both packed and unpacked) are flattened with indices appended to the signal name. We use open-source Yosys tools to show the integration of simopt dump and how it can aid in improved mapping and placement of logic.

Yosys provides a versatile environment for hardware description language (HDL) processing and synthesis tasks while allowing the functionality of the toolchain to be extended

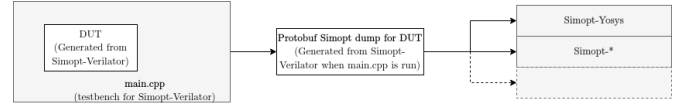


Fig. 3: The Simopt framework: Plug-and-use feature. Here *Simopt-\** can be any other framework which has integrated the Simopt encoded dump.

by user-defined optimisations [?]. Technology mapping in this flow is enabled by Berkeley’s ABC tool [?]. The technology mapping step optimises the synthesized netlist of the design and fine-tunes it for the target architecture, using advanced algorithms to perform logic minimisation and mapping of resulting logic to resources on the FPGA. These optimisations and mapping decisions have a significant impact on total resource consumption and the performance that can be achieved for a user design.

We first explain the standard Yosys-ABC flow to show how simopt\_counters can be integrated to guide placement within this flow. First, complex hierarchical structures that are generated when converting Verilog to And-Inverter Gate (AIG) logic by the Yosys synthesis flow is flattened. Subsequently, technology mapping uses internal Verilog reference modules to substitute equivalent logic in the user design, followed by another flattening step before applying mapping optimisations. At this phase, the Protobuf files from the Simopt

framework are deserialised to generate a map of netlist names with corresponding simopt counter values. To do this, a new backend script is generated that is invoked from within the Yosys flow to generate the simopt\_counter embedded flattened netlist file. This netlist is used by the modified ABC flow where the initial steps (structural hashing, logic minimization, structural correction) are applied as-is to optimise the design and to perform technology-aware synthesis. Simopt-counters are integrated into the flow during the technology mapping stage (`if` command), which maps and optimises the design by identifying and factoring out common subexpressions across multiple levels of logic. At this stage, the tool determines the cost function for mapping the logic to Lookup tables (LUTs) using “priority cuts” where user-driven priority can be imposed. To integrate simopt-driven speculation, we optimise the priority cuts using the area cost function. The standard area cost function LUT mapping in ABC is formulated in equation 1

$$A = \sum_i^N \left( \frac{L_{inputs}}{L_{max\_inputs}} \times L_{outputs} \right) \quad (1)$$

where  $A$  is the LUT area,  $L_{inputs}$  is the number of inputs to a logic design,  $L_{max\_inputs}$  is the maximum number of inputs a LUT can have and  $L_{outputs}$  is the number of outputs to a logic design. The function effectively aggregates individual area contributions ( $i$  to  $N$ ) to estimate the synthesized logics’ area. The goal of this optimisation is to determine the mapping configuration(s) that minimises  $A$ .

To enable weighting using simopt\_counters, we introduce a logarithmic scaling factor for each design as shown in the equation 2. Fig 4 shows the dampening effect we aim to achieve using the logarithmic scaling, allowing nets with lower count values to be assigned a lower weighting in the area minimisation logic, whereas nets with high count values (higher activations during simulation run) will be assigned a higher preference. The logarithmic scaling attaches a small bias to the area optimisation algorithm allowing logic elements driving and/or receiving the signals with high simopt\_counter activations to be preferentially packed and placed in this phase.

$$A_{simopt} = \sum_i^N \left[ \frac{L_{inputs}}{L_{max\_inputs}} \times L_{outputs} \times \left( \log \left( \frac{simopt\_score}{1 + simopt\_score} \right) + 1 \right) \right] \quad (2)$$

Once the packing and placement are determined, a mapped netlist is generated in the subsequent phase and the Yosys tool is configured to generate a placed netlist in the Berkely Logic Interchange Format (BLIF) format. For our evaluation, we import this as a pre-placed IP package into AMD’s Vivado tool with an AXI wrapper that integrates this design as an AXI slave IP on a Zynq platform for testing the performance. The implementation runs on Vivado are modified to treat the pre-placed IP as a black-box to ensure that the placement and routing generated through the simopt-enabled Yosys flow are preserved. The generated design is subsequently deployed on a Pynq-Z2 block and its input-output latency is quantified using

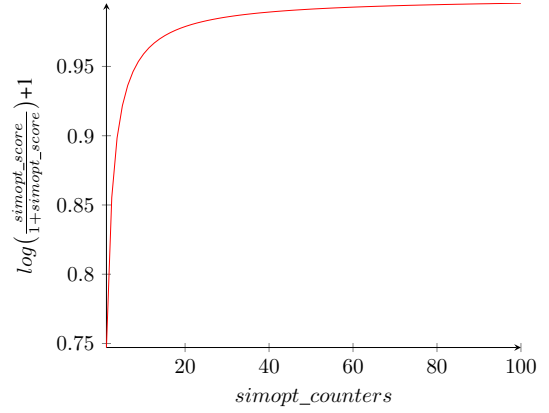


Fig. 4: A graph of logarithmic scale factor vs the simopt\_counters. Higher the simopt\_counter, lower the variance in the scalar factor, allowing all high activations to be treated with importance

a FreeRTOS C driver on the ARM cores using standard register read/write operations.

## V. RESULTS

For testing this framework, our target user designs have to be compatible with Verilator and are hence restricted to synthesisable Verilog 2001 designs that use constructs supported by Verilator. We used the internal Verilator benchmarks and open-source EPFL circuit benchmarks in this work to quantify the performance gains. The latency of the design was chosen as the parameter since the benchmark circuits are small designs and would not offer significant gains in resource consumption. We compare two versions of the design: one generated by the proposed simopt-enabled Yosys-ABC-Vivado flow and the second through a unmodified Yosys-ABC-Vivado flow. All Verilator simulations were performed on a machine with AMD Ryzen-2700X CPU and 64 GB of RAM. The FreeRTOS driver that interacts with the design sends input combinations to the deployed design and times the latency of the logic using the internal timer library that uses the timer blocks on the ARM core (running at 650 MHz). An interrupt handler registered to the interrupt line of the wrapped AXI block is used to detect the completion of the operation by the block under test. Timestamps at the start and end are used to evaluate the elapsed time and hence the latency incurred by the block. The measurements are averaged over 1000 runs to ensure that any OS-level variability is captured and eliminated. We summarise our results across the verilator\_ext\_tests test suite [?] in table I and EPFL benchmarks (arithmetic section) [?] in table II.

Circuit	Simopt Time (ms)	Vanilla Time (ms)	% savings (latency)
t_math_wallace	152.825	162.229	5.8
t_synmul_mul	120.064	126.572	5.1
t_math_cond_huge	220.358	230.452	4.4
t_wbuart32_linetest	149.93	157.98	5.1

Table I: Result of using Simopt-pass on in-built Verilator circuits



Circuit	Simopt-Time (ms)	Vanilla-Time (ms)	% savings (latency)
adder	0.21	0.34	38.2
div	4.76	5.08	6.3
hyp	5.98	6.53	7.9
max	0.31	0.43	27.9
sin	1.89	2.28	17.1
multiplier	1.36	1.42	4.2
sqrt	2.87	3.12	8.0
square	2.05	2.21	7.2

Table II: Latency results of Simopt framework of the arithmetic circuits in EPFL benchmark

In tables I and II, Simopt-Time corresponds to the latency measured for the design in which our simopt-driven optimisation was applied using our modified Simopt-Yosys-ABC and the resultant BLIF IP is used in the generated bitstream, with Vanilla-Time capturing the latency of the standard Yosys-ABC generated design. The benchmark design packaged within Verilator includes low-level primitives such as Wallace tree multiplier having inouts with bit-widths ranging from 16-bit to 128-bit connecting to sequential circuits. The results in table I show that simopt-driven optimisation can improve the circuit latency by at least 4.4% across the larger benchmark circuits that are part of the Verilator tools.

For the EPFL benchmark circuits shown in table II, a fixed input value was used across the tests by varying the precision according to the input width of the circuit. The benchmark contains circuits with a minimum bit-width of 32 bits and a maximum of 128 bits. In the case of higher bit-widths, the inputs are padded to the required precision through sign extension or by zero-padding. Among the EPFL benchmarks, latency measurements for some cases such as *bar* fell within the margin of measurement error for our setup (20  $\mu$ s tick-rate of FreeRTOS) and are hence not shown in the table.

From the results, it can be observed that Simopt-driven speculative optimisation can provide substantial improvements in the operating speed and latency of circuits, particularly in the case of combinational logic. Moreover, there was no observed change in FPGA utilisation area running the benchmarks through the Simopt framework. It should, however, be noted that additional logging required for Simopt metadata generation does incur longer simulation runtimes on Verilator. With large combinational designs, we observed that logging all signals can cause up to  $5\times$  increase in simulation time in Verilator, and further user-driven optimisations could be explored in this case. The implementation and runtime performance in the Simopt-Yosys-ABC flow was insignificant, with a worst-case of 5% increased runtime that was observed in our tests.

## VI. CONCLUSION

In this work, we demonstrated the case for using metadata generated during the circuit design and validation phase (through behavioural simulations) to guide the optimisations during the placement and routing phases of the CAD flow. The Simopt framework introduced in this work is a plug-and-

play model that can be adapted to any simulation framework to generate the *simopt-dump*, to be fed into the later place & route tools to generate lower area or lower latency models. In the use case discussed in this paper, we demonstrated the flow using an open-source Verilator tool for simulation and the open-source Yosys-ABC tool for synthesis, place & route, adapting the CAD flow to utilise simulation metadata for altering the priority of cuts during the placement flow. The integrated flow was benchmarked using openly available circuit descriptions with the results showing that substantial improvements to the circuit's latency can be achieved using this flow. In the future, we propose to investigate methods to further increase area and/or latency savings by applying optimisations across multiple hierarchies and to the routing flow, while also investigating methods to reduce overheads incurred during the metadata generation phase.

## REFERENCES

- [1] M. Siracusa, E. Del Sozzo, M. Rabozzi, L. Di Tucci, S. Williams, D. Sciuto, and M. D. Santambrogio, "A comprehensive methodology to optimize fpga designs via the roofline model," *IEEE Transactions on Computers*, vol. 71, no. 8, pp. 1903–1915, 2021.
- [2] J. M. De Haro, R. Cano, C. Alvarez, D. Jiménez-González, X. Martorell, E. Ayguadé, J. Labarta, F. Abel, B. Ringlein, and B. Weiss, "Ompss@cloudfpga: An fpga task-based programming model with message passing," in *2022 IEEE International Parallel and Distributed Processing Symposium (IPDPS)*. IEEE, 2022, pp. 828–838.
- [3] A. Haghi, S. Marco-Sola, L. Alvarez, D. Diamantopoulos, C. Hagleitner, and M. Moreto, "Wfa-fpga: An efficient accelerator of the wavefront algorithm for short and long read genomics alignment," *Future Generation Computer Systems*, vol. 149, pp. 39–58, 2023.
- [4] A. K. Jain, H. Omidian, H. Fraisse, M. Benipal, L. Liu, and D. Gaitonde, "A domain-specific architecture for accelerating sparse matrix vector multiplication on fpgas," in *2020 30th International conference on field-programmable logic and applications (FPL)*. IEEE, 2020, pp. 127–132.
- [5] K. Zeng, Q. Ma, J. W. Wu, Z. Chen, T. Shen, and C. Yan, "Fpga-based accelerator for object detection: a comprehensive survey," *The Journal of Supercomputing*, vol. 78, no. 12, pp. 14 096–14 136, 2022.
- [6] T. Mohaidat and K. Khalil, "A survey on neural network hardware accelerators," *IEEE Transactions on Artificial Intelligence*, 2024.
- [7] S. Kloibhofer, "Run-time data analysis to drive compiler optimizations," in *Companion Proceedings of the 2021 ACM SIGPLAN International Conference on Systems, Programming, Languages, and Applications: Software for Humanity*, 2021, pp. 9–12.
- [8] T. Sterling, M. Anderson, and M. Brodowicz, "A survey: runtime software systems for high performance computing," *Supercomputing Frontiers and Innovations*, vol. 4, no. 1, pp. 48–68, 2017.
- [9] W. Snyder, D. Galbi, and P. Wasson, "Verilator: Verilog simulator," <https://www.veripool.org/verilator/>, 2018.
- [10] S. Popić, D. Pezer, B. Mrzovac, and N. Teslić, "Performance evaluation of using protocol buffers in the internet of things communication," in *2016 International Conference on Smart Systems and Technologies (SST)*. IEEE, 2016, pp. 261–265.
- [11] B. L. Barzen, A. Reais-Parsi, E. Hung, M. Kang, A. Mishchenko, J. W. Greene, and J. Wawrzynek, "Narrowing the synthesis gap: Academic fpga synthesis is catching up with the industry," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*. IEEE, 2023, pp. 1–6.
- [12] A. Mishchenko *et al.*, "Abc: A system for sequential synthesis and verification," *URL <http://www.eecs.berkeley.edu/alanmi/abc>*, vol. 17, 2007.
- [13] W. Snyder, D. Galbi, and P. Wasson, "Verilator Test suite: External and extended tests for verilator testing," [https://github.com/verilator/verilator/\\_ext/\\_tests](https://github.com/verilator/verilator/_ext/_tests), 2018.
- [14] L. Amarú, P.-E. Gaillardon, and G. De Micheli, "The epfl combinational benchmark suite," in *Proceedings of the 24th International Workshop on Logic & Synthesis (IWLS)*, 2015.