Large-scale Gatelevel Optimization Leveraging Property Checking

Lucas Klemmer Dominik Bonora Daniel Große
Institute for Complex Systems, Johannes Kepler University Linz, Austria
lucas.klemmer@jku.at, dominik.bonora@jku.at, daniel.grosse@jku.at

***Abstract*—Often the full range of all possible input combinations of circuits is not needed for a specific use case. For example, an embedded processor might only use a small subset of all available instructions, or the operands to a multiplier are guaranteed to be within certain bounds. These external don't cares result in the interesting case of gates in the netlist that are completely inactive (or redundant), since they are never activated by the inputs to the design. Those gates can be safely eliminated, reducing the size of the netlist without loss of functionality.**

In this paper, we present PSYN, an approach to detect and eliminate inactive gates using *Property Checking* (PC). Our approach can be viewed as a technology-independent logic optimization to be used before technology mapping. PSYN uses only free and open-source tools for each step from synthesis to formal. We split the underlying large problem into many small sub-problems, which can be effectively distributed over multiple machines. This enables PC-based gate optimization on a large scale, eventually producing an optimized netlist. In the experiments, we show that our approach can lead to significant logic reductions in moderate runtimes, even for large netlists.

I. Introduction

In *Very Large Scale Integration* (VLSI) design, synthesis refers to the process of transforming a high-level hardware description, typically given as a *Register Transfer Level* (RTL) design, into a gate-level representation.

During synthesis, various optimizations are applied to improve the design's performance, timing characteristics, power consumption and area utilization. These optimizations include logic minimization, technology mapping, resource sharing, and clock tree synthesis. In this work, our focus is on technology-independent logic minimization of Verilog netlists.

Logic optimization has been fundamental for digital circuit design. In the early days of digital design, logic optimization primarily targeted the reduction of a Boolean function to a two-level sum-of-products form. In the late 1980s, multi-level logic optimization techniques gained prominence. Methods like algebraic manipulation, factoring, common sub-expression elimination, and functional decomposition were employed to minimize logic complexity [1]. The techniques further evolved and *Don't Cares* (DCs) started to play a central role in logic optimization [2]. DCs provide significant flexibilities for optimization algorithms. Typically, *Internal Don't Cares* and *External Don't Cares* are distinguished. Internal DCs result from the netlist structure in the presence of reconvergent paths. They are divided into *Satisfiability Don't Cares* (SDCs) and *Observability Don't Cares* (ODCs). In essence, SDCs occur when certain input combinations are not generated for a node, while ODCs arise when the output value of a node is not important under specific conditions. The computation of both has been formulated using incompletely specified functions also called permissible functions [3], [4]. Strong improvements became possible by using simulation and SAT, see e.g., [5].

The above-mentioned external DCs are given by the environment or explicitly by the user. While there have been attempts to unify the characterization of the different DC types [6], the challenge when targeting external DCs is that the optimization problem changes from considering a (completely specified) Boolean function to optimizing a Boolean relation [7]. Moreover, as also reported in [7] there is currently no open-source synthesis tool available which accepts external DCs. From a practical perspective, it is crucial that the **specification of external DCs is very user-friendly** such that the full optimization potential can be leveraged.

To the best of our knowledge, [8] is the only existing work in the direction of user-friendliness: formal properties are used to specify external DCs. However, the paper targets the very specific application of eliminating the logic in a processor for unneeded instructions from an *Instruction Set Architecture* (ISA). More precisely, it presents the *Property-Driven Automatic Transformation* (PDAT) framework. The user specifies properties to capture the unneeded instructions, essentially assumptions for the instruction signal. Then, PDAT operates on the processor netlist and uses PC to detect gates that are guaranteed not to toggle for the reduced ISA. This is done by checking several properties per gate. Finally, PDAT eliminates these non-toggling gates to generate a new design. While the general PDAT approach is very interesting, the paper has several deficiencies: First, PDAT only uses commercial tools. Second, the properties which are annotated to each gate are not fully provided as well as other important details on the (verification) directives to the commercial tools are missing. Finally, no run-times are reported in the experiments. All these deficiencies motivated us to revisit the principle of the property-based optimization idea.

Contribution: In this paper, we present PSYN, an approach to detect and eliminate inactive gates using PC. Our approach leverages open-source tools for each step from synthesis to formal, in particular Yosys [9] which integrates ABC [10]. The user can specify assumptions about the input data[1] (describing external DCs) in form of Verilog assertions. Then, for each gate in a Verilog netlist, a set of properties is generated. A distinct PC problem must be solved for each of these prop-

[1]PSYN can also be used to assume properties on internal signals.

erties, all while considering the user-specified assumptions to be true. Undoubtedly, there is a substantial potential for parallelization. We split the underlying large problem into many small sub-problems, which can be effectively distributed over multiple machines, thus enabling PC on a large scale, eventually producing an optimized netlist.

In our experiments, we analyze the performance and gate reduction capabilities of PSYN. First, since the number of gates that are handled in each PC problem simultaneously is configurable, we experimentally find advantageous task sizes depending on the size of the netlist in Section VI-B. Next, we evaluate how much impact the assumptions have on the runtime of PSYN. For this, we apply PSYN to several designs from the EPFL logic synthesis benchmarks using randomized assumptions. Then, in Section VI-D, we recreate several optimized versions of the Ibex processor based on subsets of the RISC-V ISA as presented in [8]. Finally, in Section VI-E we compare the open-source Yosys backend against the backend using a well-known commercial formal verification tool.

The rest of this paper is structured as follows. First, we give an overview about PSYN and the distributed computation in Section II. Next, with a particular focus on open-source software, Section III introduces which tools are used in PSYN. Then, we present the architecture of PSYN in Section IV followed by a description of the main algorithm in Section V. Finally, after an experimental section in Section VI we conclude this paper in Section VII.

II. Overview and Methodology Extensions

This section provides a brief overview of our PSYN flow. We based PSYN on the idea of finding redundant gates using PC as presented in [8] as PDAT. We try to answer the question which gates can be safely removed from a Verilog netlist without changing the function of the circuit. By adding (temporal) properties to each gate, which can detect such behavior, and by considering user-provided assumptions on the input data describing the external DCs, we can solve this problem using a property checker. Therefore, we first need to attribute every gate with the corresponding properties as partially described in [8].

At this point, our PSYN flow extends the PDAT methodology. We noticed that the PC problems for all annotated gates are independent of each other. This means that we can solve them separately, which we also found to be a very effective way to make the open-source PC tools handle large netlists. Therefore, we split the problem according to our resources into smaller tasks of a few hundred gates each. These tasks can then be distributed to as many machines as available, making use of the high parallelization potential. The result of all checked PC problems is a list of gates together with rules how they can be replaced without compromising the functionality of the design.

In the next step, the original netlist is rewritten according to this list of possible optimizations. In this step, we go over the complete netlist and, if we come across a gate that can be optimized, we remove the gate adhering to the mentioned rules. In general, there are two kinds of rules for most gates: we either assign one of the inputs to the output of the gate, or we assign a constant value to the output of the gate. Since more than one property can be satisfied for a gate at the same time, the properties are prioritized. Within this priority system, a constant value is given priority over a wire.

Every such property can now be used to change a single gate in the circuit. Once this replacement is done, we resynthesize the circuit with Yosys. This allows us to simplify the circuit further, by incorporating any cascading effects the gate removal had into the design. This resulting circuit is then used for a comparison with the initial circuit, to generate our gate count metrics.

III. Utilized Software

One of the main goals of PSYN is that it is implemented using open-source tools exclusively, and that PSYN itself is available open-source[2]. This is especially important considering the highly specialized tools required for the PSYN flow, namely logic synthesis and formal verification. Until recently, these tools were only available from commercial vendors, which posed a very high entry barrier for smaller companies, researchers, and hobbyists. Only recently it became feasible to perform all steps of the hardware design flow including synthesis, place and route, and formal verification using only free and open-source tools. This is in large part thanks to Yosys [11], which acted as a crystallization point for the open-source EDA community.

Yosys is also at the heart of our flow, leverages ABC [10] and provides both, the synthesis and formal verification to PSYN. Additionally, we utilize *sby*, a front-end for the formal verification capabilities of Yosys, for solving our properties. We currently use *Z3* [12] as the solver engine, however since sby supports multiple engines (e.g., Boolector [13], Yices [14]) supporting them is one of our next goals.

As we will present in Section IV-2, the optimization of a design is split into many smaller tasks. Each task is a subset of gates for which properties will be checked in a single call to sby. Ideally, we would like to add the properties of a task to the gates in the design using SystemVerilog's *bind* construct. However, this is not supported in the currently available open-source version of Yosys, and thus we had to implement this feature in another way. We circumvented this problem by injecting the properties of a task into the netlist just before sby is started. This is done using the *AWK* text processing language. Finally, PSYN itself is implemented as a Python application with a simple command-line interface.

IV. PSYN Architecture and Configuration

This section presents the proposed PSYN approach in detail. First, we describe the two available backends in PSYN in Section IV-1. These include a backend utilizing only free and open-source tools and a backend utilizing a well-known commercial formal verification tool. Then, in Section IV-2,

[2]PSYN is available at: https://github.com/ics-jku/psyn

we present how multiple machines can be connected to form a compute cluster onto which PSYN can effectively distribute the computation. Finally, Section IV-3 briefly introduces how PSYN and the compute cluster can be configured.

1) PSYN Backends: The PSYN flow can utilize different backends to perform the property-driven synthesis. We implemented two backends, one using only the previously mentioned open-source tools and one backend using a commercial PC tool. Adding new backends is straightforward, as each backend is a class that has to provide only three functions: (1) *inject*, which adds the assumptions to the design; (2) *bind*, which binds the properties to the gates; (3) *work*, which performs the PC. In the remainder of this section, we will focus only on the open-source backend, as this backend allows the large-scale distributed flow and is therefore our main contribution.

2) Open-source Backend Compute-Cluster: Our *open-source backend* leverages sby and allows distributing PC onto multiple machines, if the machines are connected in a specific setup. An overview of this setup is shown in Fig. 1. First, the number of machines in the setup is not limited, however one of the machines must act as the main coordinating process (light red box around Worker 1). This process is the entry point to PSYN and performs the initial synthesis, rewrites and re-synthesizes the netlist, and performs the equivalence checking. It also provides the task API, to which the workers connect during PC to get new tasks. To this end, a port on the main machine must be open such that they can make requests to the task API (blue lines).

Further, all machines must have access to a shared directory over a network file system (orange lines and cylinder). This shared directory is required to exchange the netlist, the assumptions, and the redundant gates. Finally, ssh connections between the machines must be possible so that the coordinating process can start the clients on all workers (red lines).

3) Configuration and Usage: Each hardware design is unique and distinct from one another. Therefore PSYN provides configuration features: First, the *config.ini* file is the main place for declaring designs that should be made available to PSYN. To add a new design, a new section has to be added to the config file. Listing 1 shows an exemplary configuration for the Ibex processor. A new section is declared by writing the name inside brackets. The filename and the name of the top module are specified using the keys *design* and *top* respectively. Next, the file holding the assumptions is specified using the *assumptions* key. If the design is sequential, the reset signal and its polarity can be set via the *reset* and *active-low* keys. Additionally, the number of times a sequential design is unrolled and the number of cycles that are skipped by the PC tool are declared using the *unroll* and *skip* keys.

The next configuration file defines which machines are used by PSYN. This is done using a *workers.json* file, which must contain a JSON dictionary with the hostname of the workers being the keys and the number of parallel threads on a worker being the values. An example of this is shown in Listing 2. Note, that the main machine that runs the coordinating process does not have to be, but can be, included in this list, it only has to be connected to the workers like discussed above.

Listing 1 PSYN configuration file.

```
[ibex]
design = ibex_top.v
top = ibex_top
assumptions = ibex_assumptions.v
reset = rst_ni
active-low = 1
unroll = 8
skip = 2
```

Listing 2 Worker configuration file.

```
{
  "worker1": I,
  "worker2": J,
  ...
  "workerN": K
}
```

V. PSYN Algorithm

In this section, we present the main algorithm of PSYN. The algorithm consists of three phases. The first phase, in which the design is synthesized into a suitable netlist, happens before PC starts and is presented in Section V-B. After the first phase, PC is distributed to the defined workers. This second phase is explained in Section V-C. Finally, in Section V-D the third phase is described. In this phase, the netlist is rewritten based on the PC results, the netlist is cleaned-up, and an equivalence check with the original netlist is performed.

A. Preparing the Assumptions

Before PSYN can be applied to a design, users have to provide a file containing the assumptions they would like to make about the input data. An exemplary assumption file is shown in Listing 3. In this example, we want to remove some instructions from a RISC-V processor. To accomplish this, we write an assumption about the *signal* coming into the core from the instruction memory. In this assumption, we enable specific instructions that should be included in the final netlist by only allowing certain legal values for the *op*, *funct3*, and *funct7* values of RISC-V instruction words. Now, removing an instruction is as simple as commenting out a line from the assumption. For example, the assumptions below enable all RV32I instructions[3] except the *slli* instruction (Line 6–8).

B. Before PC

In this section, we describe all steps of the PSYN flow that are executed by the coordinating process using our open-source backend. This includes the first synthesis to a netlist of standard gates (Section V-B1), the distribution of tasks via an HTTP API (Section V-B2), the rewriting of the netlist (Section V-D1) with the following re-synthesis (Section V-D2), and the final equivalence check of the original and the optimized netlists (Section V-D3).

[3]We omit most assumptions for brevity.

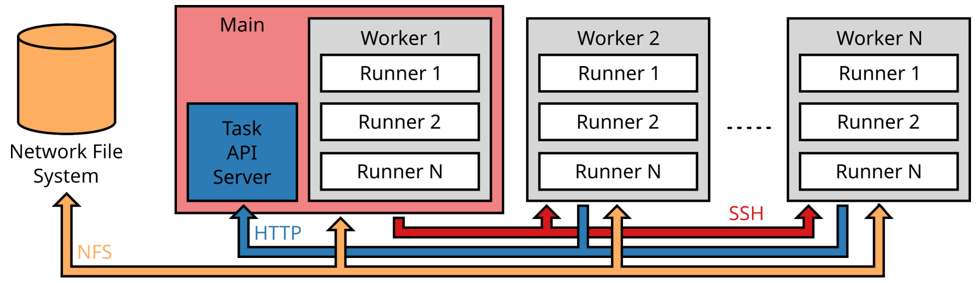

Fig. 1: The distributed architecture of PSYN.

Listing 3 An exemplary assumption file for removing instructions from a RISC-V core.

```
always @* begin
  assume ((instr[6:2] == 5'b01101) // lui
          || (instr[6:2] == 5'b00101) // auipc
          || (instr[6:2] == 5'b11011) // jal
          ...
          // || ((instr[31:25] == 7'b0000000)
          //    && (instr[14:12] == 3'b001)
          //    && (instr[6:2] == 5'b00100)) //
              slli

          || ((instr[31:25] == 7'b0000000)
               && (instr[14:12] == 3'b101)
               && (instr[6:2] == 5'b00100)) //
                   srli
          ...
          );
  end
```

1) *First Synthesis:* In this first step of the PSYN flow, the design specified in the configuration file is synthesized to a standard gate-level library. PSYN can be applied to *Register Transfer Level* (RTL) level designs and to netlists. However, since the format of the netlist is important for the following phases of PSYN, synthesis has to be performed even if the design is already a netlist to get the expected output format. The TCL script controlling the Yosys synthesis is shown in Listing 4. The top-level module is set via the environment variable *PS_TOP* which is set by PSYN before the script is started. Most of the synthesis commands follow a basic Yosys synthesis, however the call to *ABC* [10] is very important. This maps the logic elements to a small set of gates which is build into Yosys: In fact, this guarantees that all gates are mapped to standard Verilog unary and binary logic operators (i.e., &, |, ^, and ~). For each of these basic operators, PSYN contains a set of properties that are used to check if a gate is redundant. These properties will be presented in detail in Section V-C1.

2) *Task Server and Worker assignment:* After the first synthesis is done and the problem was split into tasks, the tasks have to be distributed to the workers. For this, PSYN starts an HTTP server that provides thread-safe access to a queue of all remaining tasks. The server provides an API with four endpoints that is used to synchronize and exchange tasks between the workers and the PSYN coordinating process. Each endpoint either returns a result in the form of JSON data, or it can be used to POST JSON data to update the state of a task. The four endpoints of the API are:

Listing 4 TCL script for the first synthesis stage of PSYN.

```
yosys read_verilog -sv -formal -DSYNTHESIS=1 work
    /step1.v
yosys hierarchy -check -top $::env(PS_TOP)
yosys flatten
yosys proc
yosys opt
yosys fsm
yosys opt
yosys memory
yosys opt
yosys techmap
yosys opt
yosys abc -g gates
yosys opt -full -share_all
yosys clean -purge
yosys write_verilog work/synthesized.v
```

/task/get	Retrieve an unsolved task from the queue.
/task/done	Mark a task as solved.
/done	Check if all tasks have been solved.
/remaining	Return the number of remaining tasks.

This API is started by the open-source backend after the design was synthesized and split into tasks. On startup, the API server reads the list of all tasks and inserts them into a thread-safe queue data structure. It also creates a hash map in which the state of all tasks (i.e., if the task was already solved) is kept. This hash map is initialized with all tasks set to the state of "not-solved".

After the task server is started, the number of tasks is compared to the number of available threads on all workers. PSYN now assigns each available worker a number of threads on which runners should be started. This assignment tries to fit as many tasks as possible to the workers in the *worker.json* file, starting with the first specified worker. Therefore, if the number of tasks is smaller than the number of all available threads, not every worker is assigned to start runners. All assigned tasks are stored in the *.assigned_worker.json* file. Following the task assignment, the coordinating process connects to every worker with assigned tasks to start the *psyn_client* application, which in turn spins-off as many runner processes as have been assigned to this worker.

Now, the main phase of the PSYN flow starts in which all tasks are solved by the workers following the principle we will present in Section V-C. The API server is bundled with the PSYN application and no additional software has to be installed. In particular, executing one *psyn run config* command on one of the workers is enough to start the full flow, everything else including synthesis, task dispatching, and PC is done automatically. While all tasks are being processed, the coordinating process waits until all runner processes on all workers are stopped. If all runner processes stopped, the final work is continued as presented in Section V-D.

C. On Worker

This section presents the parts of PSYN that are executed on the workers by the *psyn_client* application. The client application first reads the number of assigned threads from *.assigned_worker.json* file and starts as many runners as were assigned to this worker.

1) Enriching: Now, each runner process starts asking the API for an unsolved task by requesting the API endpoint */task/get*. If the API returns a task, the gates of this task must be annotated with properties. The SystemVerilog *bind* statement is not available in Yosys without commercial add-ons; therefore, we had to implement a different method to add the properties. Since every design must go through the first synthesis step, we can expect the netlist to always follow the same format. In particular, each gate of the netlist is implemented as a statement following the `assign _n_ = a OP b;` format, where *N* is an integer number corresponding to the gate numbers from the task and *OP* is one of the basic Verilog operands &, |, or ^. Additionally, gates can also come inverted in a format like `assign _n_ = ~(a OP b);`. We exploit this regular format by parsing each assign statement and, if the assign belongs to a gate from the task, keep a list of all gates and the type of this gate. After all gates are parsed, the properties are injected into the netlist by adding a module containing the properties for every required gate. The type of the module is injected depending on the type of the gate. By wiring the correct input and output signals into the module, the properties get bound to a gate. This property injection is implemented using the AWK text-processing language, and the result is a rewritten netlist containing the injected properties. We call this the *enriched* netlist which is saved into the *enriched.v* file in a task-specific local working directory.

Gate Properties: The gate properties we use are based on the ones in [8] and are listed in Listing 5. We support the gate types *and*, *or*, *not*, and *xor*. For all gate types with two inputs, we check 4 basic assertions. A gate can be simplified if either the output is constant, which we denoted by the *_Y_0_p1* and *_Y_1_p2* assertions, or if it always follows one of the inputs, denoted by the *_Y_A_p3* and *_Y_B_p4* assertions. For the *not* gate only the constant output assertions apply.

The properties follow the simple naming scheme *name_output_value_weight*. This naming scheme means, that if this property holds, the gate can be replaced by *output*. Additionally, each property is assigned a *weight*. This is required, as each gate is assigned multiple properties, of which some can hold at the same time. In this system, low weights mean a higher optimization potential. We assigned the weights using a simple method: replacing a gate by a constant value (e.g., Line 4), is better than replacing it by a wire (e.g., Line 6).

Listing 5 Verilog properties for the supported gate types.

```
1   `ifdef FORMAL
2   module and2_properties(input A, B, Y);
3      always @(*) begin
4         and_Y_0_p1: assert (Y == 1'b0);
5         and_Y_1_p2: assert (Y == 1'b1);
6         and_Y_A_p3: assert (!A || B);
7         and_Y_B_p4: assert (!B || A);
8      end
9   endmodule
10
11  module or2_properties(input A, B, Y);
12     always @(*) begin
13        or_Y_0_p1: assert (Y == 1'b0);
14        or_Y_1_p2: assert (Y == 1'b1);
15        or_Y_A_p3: assert (!B || A);
16        or_Y_B_p4: assert (!A || B);
17     end
18  endmodule
19
20  module not1_properties(input A, input Y);
21     always @(*) begin
22        not_Y_0_p1: assert (Y == 1'b0);
23        not_Y_1_p2: assert (Y == 1'b1);
24     end
25  endmodule
26
27  module xor2_properties(input A, B, Y);
28     always @(*) begin
29        xor_Y_0_p1: assert (Y == 1'b0);
30        xor_Y_1_p2: assert (Y == 1'b1);
31        xor_Y_A_p3: assert (!B);
32        xor_Y_B_p4: assert (!A);
33     end
34  endmodule
35  `endif
```

2) PC and Redundant Gate detection: After the enriched netlist is written, the runner calls SymbiYosys to perform PC. When sby is done with PC, it creates a log file with the PC results. This log file contains lines indicating if a given property failed during solving or if it holds. This means that we have to identify all properties for which no failing line exists in the log file. Like the property injection step before, we implemented this behavior with AWK by keeping track of a mapping of all properties and if they failed. Initially, all properties are set to holding. Then, all lines of the log file are analyzed if they contain information about a failing property. If all lines are processed, we eliminate all failing properties from the mapping and write out a list of all holding properties together with their gate into a file in the shared directory. However, as every gate has multiple properties that can hold at the same time, we also have to filter out the best holding property for each gate. This is done using the property weights described in Section V-C1.

At this point, the runner is finished with the current task. It then posts the task to the */task/done* endpoint of the task API to mark this task as completed. Now, the runner checks if any tasks are left to do by requesting the *task/get* endpoint of the task API. If a task is returned by this request, the runner starts executing this task, else the runner stops itself.

D. After Property Synthesis

After all tasks are processed and all runners are stopped, the control is returned to the coordinating process. At this stage, the only thing left to do is to rewrite the netlist with the optimization results, to re-synthesize the rewritten netlist to perform additional optimizations and clean-up, and to perform the equivalence checking on the rewritten netlist.

1) Netlist Rewriting: In this stage, the netlist is rewritten based on the files containing the holding properties for each gate which were produced by the runners. Similar as in previous steps, the netlist rewriting is implemented using an AWK program that performs the rewriting directly in the Verilog netlist. First, this program parses the file containing all gates and their rules according to which the gate can be optimized. Next, it checks each line of the netlist for a gate that can be optimized. If such a gate is found, it is rewritten according to the rules presented in Section V-C1.

2) Resynthesis: Rewriting the netlist results in two additional optimization cases. First, the rewritten netlist might contain new optimization potential for the traditional logic synthesis in Yosys. Second, the rewritten netlist might contain noise like chains of wire assigns (e.g. $a = b = c$) that can be reduced (e.g. to $a = c$). For this, we perform a second synthesis on the rewritten netlist using a very similar set of commands as in the first synthesis. The result of this second synthesis is also the final result of the PSYN flow.

3) Equivalence Checking: Cutting out big parts of a design is an aggressive process and therefore requires certain checks to ensure the correctness. To this end our pipeline includes equivalence checking of the resulting circuit with the base one. Because the parts we cut out could only be removed under certain assumptions, we also have to annotate both circuits with the same assumptions because we only expect them to be equivalent under these circumstances. Following the annotation, we can use the Yosys equivalence checking flow with a miter circuit to check for equivalence.

VI. Experimental Results

In this section, we present experimental results demonstrating the optimization potential and performance characteristics of PSYN on various experiments ranging from arithmetic circuits all the way to RISC-V processors. First, the experimental setup including the compute cluster on which the experiments were conducted is presented in Section VI-A. Next, in Section VI-B we find an advantageous number of gates that make up a task to decrease the overall runtime of PSYN. Then, in Section VI-C we analyze the runtime behavior of PSYN by running selected designs from the EPFL synthesis benchmarks, each 20 times using randomized assumptions. Section VI-D presents experiments reducing logic for a RISC-V processor. Finally, we compare the open-source against the commercial backend in Section VI-E.

A. Experimental Setup

All experiments are conducted on a cluster of five compute servers (workers) running Ubuntu 20.04 LTS, connected over a local network and a shared NFS file system. Four of these workers have Intel Core i7-10700 8-Core processors with 16 threads, of which two have 64 GB RAM and two have 128 GB RAM. The last worker has an AMD EPYC 7713 64-Core Processor with 128 threads and 256 GB RAM.

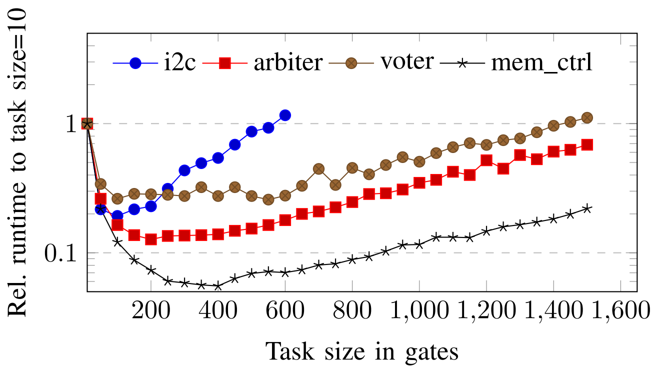

Fig. 2: Runtime for increasing task sizes.

B. Determining Favorable Task Sizes

As described before, after synthesizing the netlist it is divided into multiple tasks, each representing a slice of all gates that have to be checked. Slicing the design into many smaller tasks has not only the advantage of enabling large-scale parallelism, but it also significantly speeds up the time required to solve each task. For example, solving all properties on all gates at once quickly overwhelms the solver, on the other hand, solving each gate individually is fast, but it also incurs some additional cost as the solver has to be started more often. Finding the sweat spot between solver speed and reducing the overhead of additional startups is therefore mandatory.

In this section, we experimentally find a favorable task size by applying PSYN on selected designs from the EPFL benchmarks with different task sizes. The EPFL benchmarks contain arithmetic and non-arithmetic designs including an adder, memory controller, arbiter, and router. Fig. 2 shows a chart of the runtimes for the four largest designs from the *random_control* directory of the EPFL benchmarks, ordered from smallest (i2c) to largest (mem_ctrl). The runtimes are normalized, with the runtime of a task size of 10 serving as the baseline, and are shown on the y-axis. For each design, the task size is increased in steps of 50 starting with a task size of 10 all the way up to a task size of 1500. This process is stopped prematurely if either the task size is larger than the number of gates, or the runtime surpasses the baseline value. The graph indicates that a small task size significantly increases the runtime, and that runtimes sharply decrease with growing task sizes. However, this effect peaks at some point and starts to reverse for growing task sizes. A good task size depends on the size of the design, with larger designs requiring a larger task size. For example, the advantageous task size for the *i2c* design is at around 100 gates, while the advantageous task size for *mem_ctrl* is at around 400 gates.

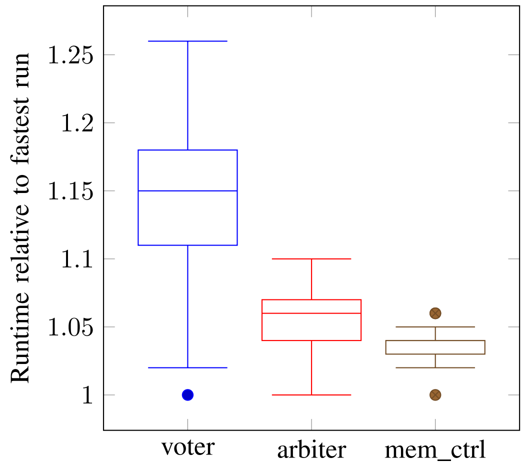

Fig. 3: Distribution of property checking runtimes for the 20 randomized runs.

C. Gate Reductions and Runtime

In this section, we apply PSYN on the EPFL benchmarks using randomized assumptions to analyze the reduction and runtime behavior for varying assumptions. For each selected design of the EPFL benchmarks, we generate 20 test cases with random assumptions on the top-level signals and observe the number of reduced gates and the runtime. By generating randomized assertions, we try to verify that the assumptions have little impact on the runtime of our PSYN flow. The results for the randomized runs are shown in Table I. The first row lists the names of the selected benchmarks, and the second row lists the number of gates in this benchmark after the first synthesis step. The next rows list the selected task size, the summed runtime of all 20 runs, followed by the mean gate reductions and mean runtimes. Since it takes some time to start up the API server, a 3-second wait time is included in the original measurements. This waiting time distorts the following results for smaller designs, and therefore we subtracted these 3 seconds from all runtimes. The last row shows the average runtime per gate. The results in Table I show that PSYN can handle even large netlists with more than $40,000$ gates in a moderate amount of time. In particular, PSYN scales for these larger netlists with no drop in the time each gate takes to check.

In addition to the time each gate takes to check, we investigated how much the runtime varies for different assumptions. This is a very interesting metric, since knowing the runtime scales approximately with the size of the design and independently of the assumptions makes running PSYN more predictable. The runtimes of the three largest EPFL benchmarks are plotted in a box plot in Fig. 3. Note, that the runtimes are plotted relative to the shortest measured runtime. In our experiments we observed, that the runtimes vary by at most 26% and that larger netlists result in less varying runtimes.

D. Reducing Netlists of RISC-V Processors

We used our PSYN flow to synthesize different configurations of the Ibex processor and compare the resulting gate counts. These configurations are based on the ones used in [8]. The first configuration, *Full*, puts the Ibex core without any additional assumptions into our tool and is used as a comparison value. *RV32im* limits the processor to instructions in the RV32im standard. *RV32i* removes the multiplication instructions. *Reduced Addressing* further removes any R-type instructions from the core. *Safety Critical* also removes JALR, AUIPC, FENCE, ECALL and EBREAK. Lastly *RISC16* limits the core to the ADD, ADDI, AND, XOR, LUI, LW, SW, BEQZ and JALR instructions. The results of applying PSYN to the Ibex core follow those in [8]. In particular, we see a similar large decrease in gate count when limiting the core to a set of instructions and once again having a noticeable decrease when further limiting the core to the RISC-16 ISA. All 6 configurations can be processed by PSYN in roughly the same time, taking around 8.5 minutes (approximately 500 seconds) for the PC using a task size of 80 and all machines described in Section VI-A.

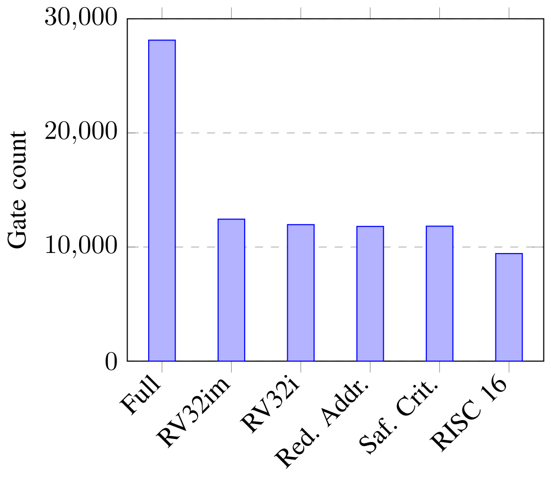

Fig. 4: Results for different RV32I subsets of the Ibex processor.

E. Comparison to Commercial Tools

In addition we implemented a PSYN backend using a well-known commercial formal verification tool. The general flow stays the same, but instead of injecting assertions in the netlist file, we can use SystemVerilog's full capabilities to bind the assertions to the gates. Similar to the open-source backend, the commercial backend can also solve multiple tasks in parallel. However, the number of concurrent processes is limited by the number of available licenses, which is 10 in our case.

The runtimes using the Yosys and the commercial backend for selected designs of the EPFL benchmarks are shown in Table II. We applied PSYN to 5 designs of the EPFL benchmarks selected to cover a range of netlist sizes. Each design was optimized with PSYN, using a task size of 80. First, we ran PSYN using the open-source backend on the largest machine once using all the available machines from Section VI-A (Row "Open-Source Full") and once with only 10 parallel tasks (Row "Open-Source 10"). Then, we ran PSYN twice using the commercial backend. In the first commercial run, we employed

TABLE I: Results of 20 runs for each selected EPFL benchmark with randomized assumptions.

Benchmark	ctrl	int2float	router	dec	cavlc	priority	i2c	voter	arbiter	mem_ctrl
Gates	99	201	207	304	633	641	1,086	6,164	11,843	40,876
Tasksize	80	80	80	80	80	80	80	200	200	400
Summed Runtimes [s]	39	42	65	51	70	140	141	2716	2058	7765
Mean Gate Reduction [%]	83.44	65.97	93.57	81.65	76.12	18.73	7.46	1.44	1.69	4.44
Mean Runtimes [s]	1.95	2.10	3.25	2.55	3.50	7.00	7.05	135.80	102.90	388.25
Time per Gate [s]	0.02	0.01	0.02	0.01	0.01	0.01	0.01	0.02	0.01	0.01

the tool's parallelization feature, utilizing all 10 licenses in parallel for solving (Row "Commercial Multi"). In this run, we split the assertions into sub-tasks using PSYN with a task size of 200. These sub-tasks are passed to the commercial tool which then distributes the assertions to the parallel tasks. In the second run (Row "Commercial PSYN") we applied the same settings of PSYN to split the assertions into smaller tasks which are then distributed to 10 workers without using the parallelization option of the commercial tool.

Both backends result in the same removed gates. However, the open-source backend is significantly faster even when restricted to only 10 workers. We attribute the better performance of the open-source backend primarily to the behavior of the assertion checking process: The open-source backend can find multiple failing assertions at the same time, while the commercial tool checks all assertions individually. In a design considering realistic assumptions, most gates cannot be optimized which means that most of the assertions are expected to fail. This particular setup, coupled with the requirement to verify a vast number of assertions which are most likely to fail, appears to result in a significantly reduced overhead in our case. However, from the data, we see that the commercial backend starts catching up with the open-source backend with increasing design sizes. Therefore, we believe that the commercial tool is geared towards solving "few" very complex assertions on large designs and not towards numerous very simple assertions. Considering this, we assume that the commercial backend starts outperforming the open-source backend for very complicated designs. However, the open-source backend will always have the advantage of practically unlimited scaling since it is not limited by the number of available licenses. Moreover, exploring new task-finding heuristics and innovative formal problem formulations that make use of specific proof engines can only be achieved when the source code can be accessed.

Finally, we also compared the multithreaded mode of the commercial tool (Row "Commercial Multi") to solving multiple tasks of the split up problem using PSYN at the same time (Row "Commercial PSYN"). We found, that the commercial tools multi-threading offers less of a speed-up than solving more of the small sub-tasks if confronted with larger circuits. In this case, it pays off to let PSYN handle all parallelism.

VII. Conclusions

In this paper we presented PSYN, an approach to detect and eliminate inactive gates leveraging PC and distributed computing. By only performing PC on a small part of the netlist's gates, we can effectively distribute the problem and thus achieve significant runtime reductions. Further, our flow is implemented using only freely available open-source tools. We have shown in multiple experiments that PSYN can lead to significant reductions of the number of gates in a netlist in moderate runtimes.

TABLE II: Runtime comparison between the open-source backend and the commercial backend.

Benchmark	ctrl	int2float	cavlc	adder	arbiter
Gates	99	201	633	772	11,843
Open-Source Full [s]	4	4	5	8	64
Open-Source 10 [s]	4	4	5	8	234
Commercial Multi 10 [s]	12	25	63	82	2,677
Commercial PSYN 10 [s]	70	109	101	103	818

Acknowledgments

This work has partially been supported by the German Research Foundation (DFG) within the project VerA (GR 3104/6-1) and by the LIT Secure and Correct Systems Lab funded by the State of Upper Austria.

References

[1] J.-H. R. Jiang and S. Devadas, "Chapter 6 - logic synthesis in a nutshell," in *Electronic Design Automation*, L.-T. Wang, Y.-W. Chang, and K.-T. T. Cheng, Eds. Boston: Morgan Kaufmann, 2009, pp. 299–404.

[2] H. Savoj and R. Brayton, "The use of observability and external don't cares for the simplification of multi-level networks," in *DAC*, 1990, pp. 297–301.

[3] K. Bartlett, R. Brayton, G. Hachtel, R. Jacoby, C. Morrison, R. Rudell, A. Sangiovanni-Vincentelli, and A. Wang, "Multi-level logic minimization using implicit don't cares," *TCAD*, vol. 7, no. 6, pp. 723–740, 1988.

[4] S. Muroga, Y. Kambayashi, H. Lai, and J. Culliney, "The transduction method-design of logic networks based on permissible functions," *TC*, vol. 38, no. 10, pp. 1404–1424, 1989.

[5] A. Mishchenko and R. Brayton, "SAT-based complete don't-care computation for network optimization," in *DATE*, 2005, pp. 412–417.

[6] ——, "Simplification of non-deterministic multi-valued networks," in *ICCAD*, 2002, p. 557–562.

[7] S. Lee, H. Riener, and G. D. Micheli, "External don't cares in logic synthesis," in *Int'l Workshop on Boolean Problems*, 2022.

[8] N. Bleier, J. Sartori, and R. Kumar, "Property-driven automatic generation of reduced-isa hardware," in *DAC*, 2021, pp. 349–354.

[9] C. Wolf, "Yosys open synthesis suite," https://yosyshq.net/yosys/.

[10] R. K. Brayton and A. Mishchenko, "ABC: an academic industrial-strength verification tool," in *CAV*, 2010, pp. 24–40.

[11] D. Shah, E. Hung, C. Wolf, S. Bazanski, D. Gisselquist, and M. Milanovic, "Yosys+nextpnr: An open source framework from Verilog to bitstream for commercial FPGAs," 2019, pp. 1–4.

[12] L. De Moura and N. Bjørner, "Z3: An efficient SMT solver," in *TACAS*, 2008, pp. 337–340, available at https://github.com/Z3Prover/z3.

[13] A. Niemetz, M. Preiner, and A. Biere, "Boolector 2.0," *J. Satisf. Boolean Model. Comput.*, vol. 9, no. 1, pp. 53–58, 2014. [Online]. Available: https://doi.org/10.3233/sat190101

[14] B. Dutertre, "Yices 2.2," in *Computer-Aided Verification (CAV'2014)*, ser. Lecture Notes in Computer Science, A. Biere and R. Bloem, Eds., vol. 8559. Springer, July 2014, pp. 737–744.