Automatic Generation of Cycle-Accurate Timing Models from RTL for Hardware Accelerators

Yu Zeng, Aarti Gupta, Sharad Malik
Princeton University, Princeton, USA
yuzeng@princeton.edu, aartig@cs.princeton.edu, sharad@princeton.edu

***Abstract*—Simulation is widely used during different stages of hardware development. This paper focuses on one specific type of simulation – cycle-accurate timing simulation, which measures the number of cycles for a given computation. We propose a pioneering approach for automatically generating cycle-accurate timing models of hardware accelerators from their RTL designs based on dependency analysis and constraint solving, making this the first technique of its kind in this domain. We demonstrate the applicability of our approach for six non-trivial designs. We show that our method achieves a 1.5x-6.9x speedup for cycle-accurate simulation over RTL models for computation-intensive accelerators, demonstrating its effectiveness. Our approach provides a cost-effective way to quickly determine the execution time of accelerators.**

I. Introduction

Hardware simulation is used in a variety of settings, including design development, design space exploration, debugging, verification/validation, firmware development, etc. This paper focuses on one specific type of simulation – cycle-accurate timing simulation, which measures the number of cycles for a given computation. Cycle-accurate RTL simulation can be slow, especially for complex designs. Enhancing the performance of simulation would yield substantial cost-benefits and consequent design quality improvement. Due to the importance of simulation, significant effort has been made to improve its efficiency. Existing works can be broadly classified as shown in the quadrant diagram of Figure 1. The lower left quadrant includes works that speed up general cycle-accurate RTL simulation. There are several commercial tools (e.g., VCS [1]) as well as open-source tools employing different techniques including exploiting low activity factors [2–5] and emulation using FPGAs [6] and ASICs [7].The upper right quadrant represents solutions that manually create different high-level hardware models [8] using languages such as Verilog, SystemC, C++, etc. These models describe abstract hardware behavior and are less accurate than RTL but have faster simulation times. Depending on the use case, models at varying abstraction levels may be used. The effort in manually creating design-specific abstract models can be significant.

However, there are relatively fewer works in the upper left quadrant that generate abstract models automatically. Automated approaches greatly minimize the effort involved in constructing these models. A^{2}T [9] and AutoILA [10] automatically generate transaction-level models (TLMs) and functional models, respectively, from RTL designs. These can be used to validate the correctness of the hardware behavior at

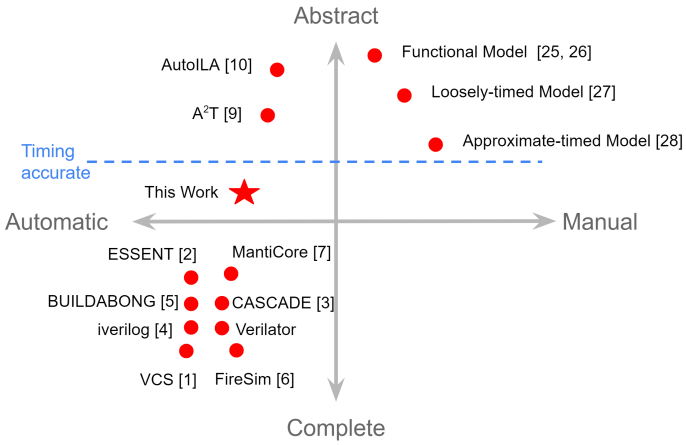

Fig. 1: Categorization of simulation speed-up work. "Abstract" refers to works that do not perform simulations as complete as RTL simulations. "Manual" denotes that the speedup is achieved through manually created models, while "Automatic" signifies that the speedup is achieved without human intervention. The dashed line labeled 'Timing Accurate' distinguishes between works that preserve exact timing information of instruction sequences (below the line), from those that lack precise timing details (above the line). Among the abstract models, only our work keeps the exact timing information.

different levels. **However, no prior work automatically generates abstract models with timing information, such as the number of cycles an instruction sequence takes.** The dashed line labeled "Timing accurate" in Fig. 1 distinguishes between works that preserve exact timing information of instruction sequences, located below the line, from those that lack precise timing details, situated above the line. *This instruction-specific timing is the focus of this paper.* This is useful for processors and accelerators, both of which have an instruction-level interface, and the timing for specific instruction sequences has several applications as discussed below.

1) Design-space exploration with templates: Hardware design templates are commonly used to enable the exploration of hardware designs with varying parameters [11, 12]. Designers can run simulations to determine the optimal parameters that lead to the highest performance (fewest cycles). Typically, these templates undergo rigorous functional verification. As a result, when exploring the design space through simulation, verifying the computed result is unnecessary, and timing information becomes the primary concern. In such cases,

simulation based on a timing model is preferable over full RTL simulation, as it helps reduce simulation costs.

2) Writing high-performance programs for accelerators: Once a hardware accelerator is designed, developers typically need to write high-performance software programs to fully leverage its computational power. Developers evaluate various program transformations and fine-tune parameters iteratively to optimize performance [13]. In cases where the hardware has not yet been fabricated, performance measurement requires simulation. Thoughtfully-crafted program transformations and templates can help ensure the correctness of the transformed program, obviating the need for results verification during simulation.

Creating cycle-accurate timing models manually demands exceptional effort: it requires identifying the logic that influences the overall timing. For complex designs, this process not only takes excessive time but is also challenging in ensuring correctness. In this work, we propose two techniques for automatically generating RTL-based timing models for hardware accelerators through dependency analysis and constraint solving. (While our methods can be applied to processors, their effectiveness is somewhat limited by processor architectures, as elaborated in Section IV.) The generated models are correct by construction, provided that the implementation of the algorithms is bug-free. Our methods identify RTL code that is not needed for timing simulation and remove that code to obtain an RTL-based timing model. The code to be retained is the computation that will be needed to determine the timing, e.g., control flow. Our methods are orthogonal to the other simulation acceleration methods mentioned previously, allowing our RTL-based timing model to benefit from the other techniques. The two techniques work independently, and users need only apply each once.

The key contributions of this work are:

- Automatic generation of instruction-specific timing models from RTL designs for hardware accelerators, which to the best of our knowledge is **the first of its kind**. Although our method might not produce a timing model of optimal performance, it does create one with decent performance at a low cost, requiring little manual intervention.
- A novel *Timing-Centric Pruning* algorithm for design simplification based on dependency analysis. Although it is a natural extension of a dead-code elimination algorithm, we are the first to propose and demonstrate that such an algorithm can be surprisingly effective in simplifying the RTL code of an accelerator for deriving a timing model.
- Development of a novel *Constraint Propagation* algorithm for removing RTL code associated with unused functionality due to specific instruction sequences, resulting in further speed up of the timing model.
- Through experimental evaluations, we demonstrate 1.5x-6.9x speed improvement in cycle-accurate timing simulation over RTL simulation for a range of accelerators.

II. Timing-Centric Pruning

In this section, we introduce the Timing-Centric Pruning (TC-Pruning) algorithm, which prunes the unnecessary RTL code of an accelerator in order to derive a timing model. Complete RTL simulation provides more information than timing alone, such as output values and memory writes. In cases where only timing information is needed, removing RTL statements that do not affect timing can lead to performance gains.

```
1  module odd_vector_accumulator (...);
2  input clk, reset, run;
3  input [7:0] vectors [0:3];
4  output done, [15:0] result;
5  reg [15:0] result, [1:0] state, [3:0] idx;
6  localparam IDLE=2'b00, CHECK=2'b01, ADD=2'b10;

8  wire last_idx = (idx == 4'b0011);
9  wire is_odd   = (vectors[idx] % 2 != 0);
10 // state machine
11 always @(posedge clk) begin
12   if (reset) begin
13     state  <= IDLE;
14     result <= 16'h0000;
15     idx    <= 4'h0;
16   end
17   else begin
18     case (state)
19       IDLE: begin
20         state <= run ? CHECK : IDLE;
21       end
22       CHECK: begin
23         if (is_odd) begin
24             state <= ADD;
25         end
26         else begin
27           idx   <= last_idx ?
28                    4'b0000 : idx + 4'b0001;
29           state <= last_idx ? IDLE: CHECK;
30         end
31       end
32       ADD: begin
33         result <= result + vectors[idx];
34         idx    <= last_idx ?
35                   4'b0000 : idx + 4'b0001;
36         state  <= last_idx ? IDLE: CHECK;
37       end
38       endcase
39     end
40 end
41 assign done = (state == IDLE);
42 endmodule
```

Listing 1: Verilog code for accumulating vectors. Lines 14 & 33 can be removed without impacting the "done" signal in a timing model.

Listing 1 shows Verilog code for a motivating example. It takes four input vectors and computes the sum of the vectors that contain odd numbers. The input vectors are stored in the array `vectors`. Each vector is checked in turn in the `CHECK` state. If the current vector is odd, its value is added to `result` in the `ADD` state, which requires one cycle. If the vector is even, the `ADD` state is skipped and the program proceeds to check the next vector. The program execution is complete when all vectors have been processed. While execution time depends on the values of the vectors, the accumulation step is not required to determine cycle count. Notably, the `done` signal that indicates program completion depends only on

`state`, not on `result`. Thus, the code can be simplified by removing the accumulation code in lines 14 and 33.

The above example highlights that not all RTL statements are necessary to determine the execution time of an accelerator. This provides an opportunity to remove unnecessary RTL code and create a timing model. However, determining which code is relevant to timing can be a challenge. Our proposed approach is to work backward from the "commit" signals (e.g., `done` in the example) in the accelerator that indicate the completion of instructions/computation. If an RTL statement does not affect these commit signals, then it is safe to remove it. The Timing-Centric Pruning (TC-Pruning) algorithm is based on this idea. It performs a cone-of-influence dependency analysis of the commit signals and prunes the signals that cannot affect them.

Algorithm 1: TC-Pruning Algorithm

Input: List of RTL statements S, where each $s \in S$ has the form: $dest = op(src)$, the set of commit signals C

Output: The simplified RTL statements S'

```
// return the dependency set for variable v, E
   is the set of variables seen before
Function Get_Dependencies(S, v, E):
    depSet ← ∅
    foreach s ∈ S do
        if s.dest ∈ E then
            continue
        else
            E ← E ∪ {s.dest}
        if s.dest == v then
            foreach u ∈ s.src do
                depSet ← depSet ∪ {u}
                ∪ Get_Dependencies(S, u, E)
    return depSet
Function Remove_Unrelated(D, S):
    S' ← ∅
    foreach s ∈ S do
        if s.dest ∈ D then
            S' ← S' ∪ {s}
    return S'
D ← ∅                  // entry point of the algorithm
E ← ∅                  // set of variables seen before
foreach c ∈ C do
    D ← D ∪ Get_Dependencies(S, c, E)
S' ← Remove_Unrelated(D, S)
return S'
```

Algorithm 1 presents the pseudo-code of TC-Pruning. It is an extension of a standard dead-code elimination algorithm used in compiler optimization. The algorithm takes two inputs: the RTL statements S of the design, and a set of commit signals C. After pre-processing, all RTL statements have the form: $dest = op(src)$, and control flow is handled through conditional assignments (e.g., the intermediate representation in FIRRTL [14] and Yosys [15] are of this form). Here, $dest$ is the destination variable being assigned, which can be a wire, register, or memory cell, op is the operator used, and src is a set of source variables. For example, in line 41 of Listing 1, $dest$ is `done`, op is ==, src includes `state` and `IDLE`. The function `Get_Dependencies` returns a set of variables that a given commit signal depends on. `Remove_Unrelated` removes the RTL statements whose $dest$ are not in the dependence set of any commit signal. This eliminates all timing-unrelated statements, resulting in a simplified RTL design for timing modeling.

III. Constraint Propagation

The TC-Pruning algorithm uses a backward dependency analysis from the "commit" signals to determine which variables and their update statements can be removed. We can also simplify the RTL code by using a forward analysis starting from the input ports. Our proposed approach is based on the observation that an accelerator often supports multiple instructions, but not all of them are needed for a particular computation (instruction sequence). For example, the deep learning accelerator VTA [16] supports multiple matrix operations, such as matrix multiplication ($matmul$), element-wise matrix Fadd/max/min, etc. Operations like $matmul$ may require extensive optimizations at both the hardware level (with design-space exploration) and the software level (by writing high-performance computing libraries). When running simulations for optimizing $matmul$, there is no need to simulate the other hardware operations, such as element-wise max/min, since they are not used in this computation. This optimization opportunity is referred to as *low activity factors* in the literature (this includes event-driven simulation) [2, 3].

```
1  module simple_alu (opcode, a, b, result);
2    input      [1:0] opcode;
3    input      [7:0] a, b;
4    output reg [7:0] result;
5
6    wire is_add   = opcode == 2'b01;
7    wire is_sub   = opcode == 2'b10;
8    wire is_shift = opcode == 2'b11;
9
10 always @(*) begin
11   if(is_add)        result = a + b; // Add
12   else if(is_sub)   result = a - b; // Subtract
13   else if(is_shift) result = a << b; //Left shift
14 end
15 endmodule
```

Listing 2: Verilog code for a simple ALU. If we know `opcode != 2'b01`, then `is_add` must be false, and line 11 can be removed.

Previous optimizations [2, 3] that exploit low activity factors skip unused operations dynamically during simulation. If the inputs of a subset of circuits do not change, then the subset can be skipped. Although the dynamic approach can effectively skip many unnecessary circuits, it introduces a run-time overhead since the simulator must monitor which

circuits need to be simulated. In this work, we introduce a *static* approach that augments the dynamic approach. We statically determine a subset of circuits that will never be simulated for specific instructions and remove them from the RTL design. For example, if a simulation only runs $matmul$, the element-wise max operation will never be executed, so it can be removed. Note that we are not replacing the dynamic approach with the static approach. Instead, the simplified RTL may further benefit from the dynamic approaches (since our static approach may not find all opportunities), and run-time overhead is reduced thanks to the simplification. Similar to TC-Pruning, Constraint Propagation is predominantly effective for accelerators rather than processors, due to substantial circuit sharing among processor instructions.

```
1 input     vld, clk, rst;
2 reg [1:0] op_reg;
3 always @(posedge clk) begin
4   if(rst)       op_reg <= 2'b00;
5   else if (vld) op_reg <= opcode;
6 end
7
8 wire is_add = op_reg == 2'b01;
9 ...
```

Listing 3: Partial Verilog code for another ALU with an internal register.

A. Example of Constraint Propagation

To identify irrelevant circuits for a specific instruction sequence, we propose a constraint propagation algorithm, which is conceptually similar to the widely-used constant propagation algorithm [17]. However, instead of propagating constant values, constraints of values are propagated, starting from the input ports.

The values on the input ports are constrained by the excluded instructions. These constraints are then propagated to the internal signals. In certain cases, constraints propagated to internal signals can simplify the design. For example, consider the simple ALU design in Listing 2, where different computations are done for the two inputs based on the input opcode value. If the input instructions of interest exclude the `Add` instruction, we can add the constraint `opcode != 2'b01` for excluding `Add`. By propagating this constraint to the assign statement for `is_add`, we can infer that:

$$opcode\ != 2'b01\ \wedge\ is\_add \equiv (opcode == 2'b01) \implies is\_add = false \tag{1}$$

Since `is_add` is inferred as false, we can safely remove the "`result = a + b`" branch since it will never be executed. This technique can be especially effective in inferring Boolean variables (e.g., `is_add`) with propagated constraints. Once their values are determined to be true or false, they can be used to further simplify the RTL code using constant propagation. We use a Satisfiability Modulo Theory (SMT) solver [18] to automate inference.

For accelerators with well-defined instructions that are fed to the input ports for just a single cycle (although they may require multiple cycles to execute), formulating input constraints is straightforward. In such scenarios, **users simply need to identify the encoding for unused instructions (e.g., by reading the manual of the design)** and then formulate constraints to effectively exclude them. For these accelerators, our algorithm works pretty well. We recognize that devising input constraints for every accelerator can be challenging. In certain cases, an accelerator's instruction might involve a sequence of inputs, complicating the constraint formulation. Developing a simpler method for writing input constraints and checking their correctness are left to future work.

B. Constraint Propagation Algorithm

Algorithm 2 presents the pseudo-code for our Constraint Propagation (CP) algorithm. In addition to the RTL statements of the design, the user must provide a set of constraints for the inputs that specify the opcodes of excluded instructions. These constraints initialize the set of constraints C, e.g., in the example above, the constraint `opcode != 2'b01` is provided by the user. We maintain a worklist W of variables with a constraint. For each constrained variable v, we examine every RTL statement where v is utilized as a source variable (line 17). Two operations are performed for those RTL statements: (1) propagate the constraints through the RTL statements (line 22-28), and (2) check the unsatisfiability of the RTL statement under constraint set C (line 18-19). If the result of an RTL statement can be inferred as always true or false, its destination variable is assigned the corresponding constant and used in a final constant propagation pass (line 29). The following subsections discuss the details of the two operations.

1) Propagate Constraints: The propagation of constraints starts from the constrained inputs (e.g., `opcode` in Listing 2). Then constraints are added for the variables that depend on those constrained inputs (lines 24 and 27). The rules for adding constraints for different types of variables are provided below. The constraints are then further propagated. A worklist is used to track which variables are constrained (lines 25 and 28). If a variable is proven to be a constant by the unsatisfiability check (discussed in the next subsection), its constraint is *not propagated further* (line 21), since the constant value is enough to simplify the code. The constraint propagation rules for wires, registers, and memories are as follows:

a) Wires: If *all* the source variables of a wire's statement are constrained (or constant), the wire is also constrained (line 23). A new constraint is added to the constraint system for the wire ($s.dest$) that has the same expression as the RTL statement s (line 24), with the assignment replaced by $\equiv$. The wire is also added to the work list (line 25). However, no constraint is added if *some* of the source variables are not constrained/constant for two reasons. First, doing otherwise would add almost all the wires to the constraint system, resulting in a long SMT-solving time. Second, constraints with free variables are less likely to be useful.

b) Registers: Consider the ALU design in Listing 3, where a register `op_reg` is used. A register can take any of the values it is assigned depending on when the "enable" signal

(vld) is true. All the assigned values have the same constraint as the assignment signal (opcode). Further, the register can also take an implicit reset value. Thus, the constraint added for the register (line 27) is as follows: it is equal to either its "update" (e.g., opcode) or the reset value ($reset(s.dest)$).
Example: The constraint for op_reg is as follows:

$$op\_reg == opcode \;||\; op\_reg == 2'b00 \tag{2}$$

Algorithm 2: Constraint Propagation (CP) Algorithm

Input: RTL statements: S, each $s \in S$ has the form: $dest = op(src)$; a set of mappings from an input v to its constraint $\phi(v)$: $C = \{v \rightarrow \phi(v)\}$. The constraint $\phi(v)$ excludes unused instruction opcodes for v.
Output: Optimized RTL statements

```
// check if the destination of s is always true
   or false, with the constraints C
1  Function Check_Bool(s, C):
2      result ← check(s.dest == true, C)
3      if result is UNSAT then
4          s.dest ← false
5      result ← check(s.dest == false, C)
6      if result is UNSAT then
7          s.dest ← true
8      return s.dest
// entry point. C.keys are the keys of map C
9  Initialize work list: W ← {v | v ∈ C.keys}
10 E ← ∅                    // set of variables seen before
11 while W ≠ ∅ do
12     v ← W.pop()
13     if v ∈ E then
14         continue        // skip variables seen before
15     else
16         E ← E ∪ v
17     foreach s : s ∈ S ∧ v ∈ s.src do
18         if (s.op is ==) or (s.op is ≠) then
19             s.dest ← Check_Bool(s, C)
20         if is_const(s.dest) then
21             continue
           // add to the constraint set & work list
22         if is_wire(s.dest) then
23             if s.src ⊂ {constant, C.keys} then
24                 C ← C ∪ {s.dest → s}
25                 W ← W ∪ {s.dest}
           // s.update is a source variable of s,
              which is used to update the register
26         else if s.update ∈ C.keys then
27             C ← C ∪ {s.dest → {s.dest ==
                 s.update || s.dest == reset(s.dest)}}
28             W ← W ∪ {s.dest}
29 return constant_propagation_pass(S)    // the pass
   can come from an existing tool like Yosys [15]
```

c) Memories: Memories are treated similarly to registers since logically they are just arrays of registers. However, instead of adding constraints separately for each memory cell, we add constraints only for the memory output. The memory output has the same constraints as the memory data input, irrespective of the memory cell being accessed. Like registers, the memory output can also be the reset value. Hence, the constraint added for the memory follows the same rule as those for registers (line 27).

2) Check Unsatisfiability: For statements with Boolean results, we use an SMT solver to determine if the result is always true or false. A constant result can further simplify the RTL in a constant propagation pass. We only consider statements where the Boolean result is from the relational operators "equal" and "not equal" (defined in lines 1-8 and used in line 19). We do not check statements with Boolean results for other relational operators (e.g., $>$, $<$). This is because the input constraints of excluded opcodes are simple "unequal" constraints, and it is unlikely that we will obtain an UNSAT result for the other relational constraints. For example, if the excluded opcode constraint is $a \neq 2$, it is unlikely to help determine whether the result of a comparison $a > 1$ in an RTL statement is true or false. Therefore, we choose to skip these checks to minimize the number of SMT solver calls.
Example: in Eq. 1, the result of the assertion is_add == true is UNSAT, so we can conclude is_add is always false.

IV. Experimental Evaluation

We implemented the TC-Pruning algorithm on top of the FIRRTL compiler [14] of the Chisel language [12]. Chisel, as a high-level hardware description language (HDL), shares essential similarities with Verilog, demonstrating that TC-Pruning could be adapted for other HDLs such as Verilog. This pass takes a Chisel design and a list of user-provided "commit" signals and optimizes the code based on Algorithm 1. It is based on the Dead-Code Elimination (DCE) pass provided by FIRRTL, which already performs most of the optimizations we need but misses some optimization opportunities. For instance, modules with no outputs should be removed, and if memory has a constant input, its output can be replaced with that constant value. We added these missing optimizations. Constraint Propagation is implemented as a pass in Yosys [15]. We use Yosys to parse the RTL code and convert it to an Abstract Syntax Tree (AST). The CP algorithm (Algorithm 2) is implemented by traversing the AST, where we use the SMT solver Z3 [18] to check assertions. All the abstract models are finally translated to Verilog and simulated with the Verilator simulator [20].

A. TC-Pruning: Evaluation

We conducted experiments on six non-trivial hardware designs to verify the effectiveness of TC-Pruning, as shown in Table I. All six designs have been developed in Chisel, enabling the application of our FIRRTL-based TC-Pruning. As there is no existing work in this field, we undertook the task of independently sourcing hardware designs for our experiments.

Designs	LoC	LoC-pruning	LoC-manual	Code Size %	Algo. Time (sec)	Verilator (sec)	TC-Pruning (sec)	TC-pruning + Algo. Time (sec)	Manual Model (sec)	Speedup
AES	87416	346	341	0.4%	6	3620	515	521	508	6.95
VTA	19833	8310	N/A	41.9%	17	3502	889	906	N/A	3.87
FPU	772	339	320	43.9%	< 1	3663	2095	2095	2073	1.75
CORDIC	260	67	67	25.8%	< 1	3611	1036	1036	1033	3.49
CMAC	8221	4899	N/A	59.6%	1.4	3588	873	874	N/A	4.11
RV-MINI	1964	1926	N/A	98%	< 1	3564	3551	3551	N/A	1.00

TABLE I: Results for TC-Pruning. The "LoC" column indicates the number of lines of Verilog code after compiling the Chisel designs to Verilog. "LoC-pruning" indicates the code size after applying the TC-Pruning. "LoC-manual" shows the code size of manually-made models ("N/A" means manually making models is too challenging due to design complexity). "Code Size %" displays the ratio of the size of the optimized code to the original code. "Algo. Time" reports the runtime of our algorithm. "Verilator" indicates the simulation time for the complete RTL code with Verilator, set to around 1 hour (3600 seconds) through the provision of ample input stimuli. **It's important to note that the simulation time can be extended by supplying additional input stimuli.** "TC-Pruning" reports the simulation time of the abstract models obtained with TC-Pruning. "TC-Pruning + Alg. Time" reports the total time of "TC-Pruning" and "Algo. Time". "Manual Models" reports the simulation time of manually-made models. "Speedup" illustrates the speedup of timing simulation with the generated models. It is calculated as: Speedup = Verilator / (TC-Pruning + Algo. Time).

	Automatic & Abstract		**Automatic & Complete**	**Manual & Abstract**
	This work	**A^2T [19], AutoILA [10]**	**Verilator [20], etc.**	**Loosely-timed models [21], etc.**
Timing accurate?	Yes	No	Yes	No
Complete computation results?	No	Yes	Yes	Yes for some
Use cases	Rapidly obtaining the timing of instruction sequences to measure performance	Rapidly computing the results of instruction sequences	Complete simulation of RTL	Simulation at the system-level

TABLE II: Comparison of different simulation models. The definition of "Automatic", "Abstract" and "Manual" can be found in Fig. 1.

These designs have been sourced from a diverse range of repositories on GitHub, each representing hardware tailored for distinct applications. AES [22] is an accelerator for the encoding and decoding of the Advanced Encryption Standard (AES) algorithm. VTA [16] is a programmable hardware accelerator designed for deep learning applications. FPU [23] is an accelerator for fused multiply-add operations of floating-point numbers. CORDIC [24] is a module that implements the efficient Cordic algorithm, capable of computing various functions such as the logarithmic function. CMAC [25] is a module for the inference of the convolution layers in neural networks. For these designs, the "commit signals" are simply the "valid" signals indicating the output results, which can be easily identified by referring to the RTL code or the design manual. RV-MINI [26] is a RISCV core with three pipeline stages. We conducted all experiments using an Intel Core i7 CPU and 32GB of memory.

The results are presented in Table I. We evaluate the simulation time of our abstract model against that of a complete RTL simulation using Verilator [20], one of the fastest open-source simulators available. The column of "Verilator" shows the simulation time of Verilator-based simulation. "TC-pruning" show the simulation time of abstract models obtained with TC-Pruning. "Manual Model" shows the time for manually-made timing models only for small designs, **as for more complex designs manually creating the models is excessively challenging.** The table shows that the code size for the first 5 designs has been reduced to less than 60% of the original and the simulation is 1.75x-6.95x times faster than complte RTL simulation with Verilator. The only exception is RV-MINI (See Section IV-A3 for analysis). For all designs except RV-MINI, the automatically generated timing models facilitate a speedup in simulation. TC-Pruning takes less than 17 seconds for all designs, demonstrating its cost-effectiveness.

Source of Simplification: We found that the majority of the code reduction occurred in circuits where the number of control steps, i.e., cycles, was not dependent on the computation results. These are circuits with pure data flow, i.e., data flow that does not determine any control flow. Next, we proceed with a detailed case analysis of three representative designs.

1) AES: AES supports two instructions: encryption and decryption. The two instructions are executed independently in two data paths. After the optimization, all the RTL code for the compute blocks for both encryption and decryption are removed. Only the control logic remains. Since the computation logic is very complex, only 0.4% of the code is retained after the optimization.

2) VTA: Unlike AES, VTA is a more flexible design with versatile instructions, including load, store, and multiple matrix computations. In the timing model, not only the arithmetic modules but also all the pure data-flow code is removed. For example, the internal memories for storing the data are all removed. However, the internal memories/buffers for storing instructions are retained.

Designs	Code Size %	Algo. (sec)	Sim (sec)	Sim-opt (sec)	Speedup
① AES: no dec	50.2%	1	603	274	2.20
② AES: no enc	52.6%	1	603	337	1.79
③ VTA: no ALU	91.7%	12	583	389	1.50
④ VTA: no GEMM	41.4%	8	583	105	5.57
⑤ RV-MINI: no shift	99.6%	3	594	592	1.00
⑥ FIXED: no div	52.4%	1	603	339	1.78

TABLE III: Results for Constraint Propagation. The column "Code Size %" indicates the percentage of code size remaining after simplification. "Algo." reports the runtime of Constraint Propagation. "Sim" reports the simulation time of the original RTL, while "Sim-opt" indicates the simulation time after Constraint Propagation. "Speedup" shows the speedup in simulation time. No manually made timing models due to complexity of these designs.

3) RV-MINI: The decision of whether to take a branch instruction is based on the result from the Arithmetic Logic Unit (ALU). Consequently, even the ALU can affect the timing and cannot be optimized away. Fortunately, branching instructions are not commonly found in hardware accelerators for computation-intensive applications, which are usually the focus of timing models due to their long simulation times.

B. Constraint Propagation: Evaluation

The objective of the Constraint Propagation algorithm is to eliminate hardware circuits associated with instructions that are not used for simulating the desired functionality. For this experiment, only the AES, VTA, and RV-MINI designs from the previous experiment were considered suitable, as they could be partitioned into distinct components that serve different instructions. They are compiled into Verilog, enabling the application of the implemented Constraint Propagation pass. Each of the other three designs uses all instructions for any meaningful simulation. We included another design, FIXED, which is a module for performing multiple fixed-point computations such as addition, multiplication, and division. In the experiment, our goal was to eliminate the division logic when it is not utilized. We could not apply TC-Pruning to FIXED because it is implemented in Verilog, and our TC-Pruning is implemented for Chisel designs.

The results are shown in Table III. As explained in Section III, our Constraint Propagation algorithm is conservative, checking only the "equal" and "not equal" expressions, leading to fast optimization runtimes. Speedup has been achieved for all designs, except for the RV-MINI processor. This is primarily due to the fact that a single arithmetic operation constitutes only a small fraction of the overall logic. We did not manually make timing models for these designs because they are too complex.

C. Comparison with other models

In Table II, we compare the features of our work with other models mentioned in Fig. 1. The primary distinction of our work from other abstract models is its unique and pioneering focus on speeding up the timing simulation of hardware instructions. Therefore, comparing the speedup with other abstract models is not meaningful because they simulate varying aspects of hardware behavior and cater to different use cases.

V. Related Work

Acceleration of RTL simulations. There are many commercial RTL simulators [1] as well as open-source simulators. One interesting line of work [2–5] focuses on leveraging low activity factors through coarse and conditional executions. Another direction is acceleration with hardware, e.g., using FPGA emulation [6], ASICs [7], and GPUs [27]. FireSim [6] deploys FPGA-based emulation on public-cloud platforms, achieving high scalability for large designs. MantiCore [7] utilizes a specific hardware architecture for parallel simulation. GPUs have also been used for accelerating RTL simulations [27, 28]. Our timing models can be translated to Verilog. Therefore, they can further benefit from all the acceleration techniques above.

Abstract hardware models. Hardware models at different levels of abstraction are utilized during the design process [8, 29]. These models include Functional models [30], Loosely-timed (LT) models [21], Approximate-timed models (AT) [31], etc. These models are manually constructed and do not provide accurate timing information. In contrast, our generated timing model provides the same cycle-accurate timing as RTL models.

Automatic generation of hardware models. A^2T [9] developed algorithms to generate transaction-level models from RTL designs. The algorithm converts RTL to Extended Finite-State Machine and then merges the hardware state for simplification. The process of merging states alters the timing, thus making the algorithm unsuitable for generating timing models. The resulting transaction-level models can be used for high-level simulation. AutoILA [10] introduced a methodology for the automatic generation of architecture-level models. This approach employs LLVM optimizations to remove implementation specifics, preserving only the essential architecture-level behaviors. Its concept parallels that of TC-Pruning: both methodologies employ code optimization algorithms to simplify RTL designs, yet they diverge in their specific target objectives. The automatically generated models produced by A^2T and AutoILA can be used for system-level evaluation, such as functional simulation and firmware development, just like the manually created versions. But neither of these works preserves the precise timing information.

VI. Conclusions

In this work, we propose an automatic approach for generating instruction-specific timing models from RTL designs of accelerators. Two algorithms, based on dependency analysis of the completion signals and instruction-based constraint solving, are proposed to remove the code irrelevant to the timing model. Experiments show that our approach can generate high-quality timing models with simulation speedups for a range of accelerators.

REFERENCES

[1] "VCS - The industry's highest performance simulation solution." https://www.synopsys.com/verification/simulation/vcs.html.

[2] S. Beamer and D. Donofrio, "Efficiently exploiting low activity factors to accelerate RTL simulation," in *DAC'2020*, pp. 1–6, 2020.

[3] J. Grossman, B. Towles, J. A. Bank, and D. E. Shaw, "The role of cascade, a cycle-based simulation infrastructure, in designing the anton special-purpose supercomputers," in *DAC'2013*.

[4] steveicarus, "iverilog." https://github.com/steveicarus/iverilog, 2023. Accessed on March 25, 2023.

[5] A. Kupriyanov, F. Hannig, and J. Teich, "High-speed event-driven RTL compiled simulation," in *Computer Systems: Architectures, Modeling, and Simulation*, Springer, 2004.

[6] S. Karandikar and etc., "FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud," ISCA '18.

[7] M. Emami, S. Kashani, K. Kamahori, M. S. Pourghannad, R. Raj, and J. R. Larus, "Manticore: Hardware-accelerated RTL simulation with static bulk-synchronous parallelism," 2023. arXiv preprint arXiv:2301.09413.

[8] L. Cai and D. Gajski, "Transaction level modeling: an overview," in *Codes+ISSS'2003*.

[9] N. Bombieri, F. Fummi, and G. Pravadelli, "Automatic abstraction of RTL IPs into equivalent tlm descriptions," *IEEE Transactions on Computers*, vol. 60, no. 12, pp. 1730–1743, 2011.

[10] Y. Zeng, A. Gupta, and S. Malik, "Automatic generation of architecture-level models from RTL designs for processors and accelerators," in *DATE'2022*.

[11] C. Celio, D. A. Patterson, and K. Asanović, "The berkeley out-of-order machine (boom): An industry-competitive, synthesizable, parameterized risc-v processor," Tech. Rep. UCB/EECS-2015-167, EECS Department, University of California, Berkeley, Jun 2015.

[12] J. Bachrach, H. Vo, B. Richards, Y. Lee, A. Waterman, R. Avižienis, J. Wawrzynek, and K. Asanović, "Chisel: Constructing hardware in a scala embedded language," in *DAC'2012*.

[13] T. Chen, T. Moreau, Z. Jiang, L. Zheng, E. Yan, M. Cowan, H. Shen, L. Wang, Y. Hu, L. Ceze, C. Guestrin, and A. Krishnamurthy, "TVM: An automated end-to-end optimizing compiler for deep learning," OSDI'18.

[14] A. Izraelevitz, J. Koenig, P. Li, R. Lin, A. Wang, A. Magyar, D. Kim, C. Schmidt, C. Markley, J. Lawson, and J. Bachrach, "Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations," in *ICCAD'2017*.

[15] C. Wolf, "Yosys open synthesis suite." http://www.clifford.at/yosys/.

[16] T. Moreau, T. Chen, L. Vega, J. Roesch, E. Yan, L. Zheng, J. Fromm, Z. Jiang, L. Ceze, C. Guestrin, and A. Krishnamurthy, "A hardware–software blueprint for flexible deep learning specialization," *IEEE Micro*, vol. 39, no. 5, pp. 8–16, 2019.

[17] M. N. Wegman and F. K. Zadeck, "Constant propagation with conditional branches," *ACM Trans. Program. Lang. Syst.*, 1991.

[18] L. De Moura and N. Bjørner, "Z3: An efficient smt solver," TACAS'08/ETAPS'08, (Berlin, Heidelberg), p. 337–340, Springer-Verlag, 2008.

[19] N. Bombieri, F. Fummi, and S. Vinco, "A methodology to recover RTL IP functionality for automatic generation of sw applications," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 20, June 2015.

[20] W. Snyder, D. Galbi, and P. Wasson, "Introduction to verilator," 2009.

[21] D. Becker, M. Moy, and J. Cornet, "Parallel simulation of loosely timed systemc/tlm programs: Challenges raised by an industrial case study," *Electronics*, vol. 5, no. 2, 2016.

[22] "aes." https://github.com/yaozhaosh/chisel-aes, Accessed on March'23.

[23] Berkeley, "berkeley-hardfloat." https://github.com/ucb-bar/berkeley-hardfloat, 2021. Accessed on March 25, 2023. Accessed on March 25, 2023.

[24] freechipsproject, "ip-contributions." https://github.com/freechipsproject/ip-contributions, 2021. Accessed on March 25, 2023. Accessed on March 25, 2023.

[25] "sodla." https://github.com/soDLA-publishment/soDLA, 2019. Accessed on March 25, 2023. Accessed on March 25, 2023.

[26] ucb bar, "riscv-mini." https://github.com/ucb-bar/riscv-mini, 2023. Accessed on March 25, 2023.

[27] H. Qian and Y. Deng, "Accelerating RTL simulation with GPUs," in *ICCAD'2011*.

[28] D. Chatterjee, A. Deorio, and V. Bertacco, "Gate-level simulation with gpu computing," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 16, jun 2011.

[29] G. M. S. S. Thorsten Grötker, Stan Liao, *System Design with SystemC*. New York, NY: Springer, 1st ed., 2007.

[30] B.-Y. Huang, H. Zhang, P. Subramanyan, Y. Vizel, A. Gupta, and S. Malik, "Instruction-Level Abstraction: A uniform specification for system-on-chip (SoC) verification," *TODAES'2018*.

[31] A. Guerre, N. Ventroux, R. David, and A. Merigot, "Approximate-timed transactional level modeling for mpsoc exploration: A network-on-chip case study," in *DSD'2009*.