A Seamless Hardware/Software Switching Technique for Embedded Systems Using HDLRuby

Lovic Gauthier

Information System Course
National Institute of Technology, Ariake College
Omuta, Fukuoka 836-8585 (JAPAN)
lovic@ga.ariake-nct.ac.jp

Sachi Yoshigai

Information System Course
National Institute of Technology, Ariake College
Omuta, Fukuoka 836-8585 (JAPAN)
s57204@ga.ariake-nct.ac.jp

Abstract— This paper presents a new hardware/software (HW/SW) co-design technique for HDLRuby, a Register Transfer Level (RTL) Hardware Description Language. This technique enables automatic switching between HW and SW implementations for HDLRuby processes that describe finite state machines. It is designed to facilitate smooth HW/SW exploration and accelerated HW simulation. Experimental results show that, with this technique, simulation can be orders of magnitude faster than standard RTL simulation.

I. Introduction

HDLRuby [1] is a Register Transfer Level (RTL) hardware description language based on the Ruby programming language [2]. Compared to RTL languages like VHDL and Verilog HDL, HDLRuby provides object-oriented programming, reflection, unconstrained genericity (anything can be a generic parameter), and meta-programming. HDLRuby descriptions can also include Ruby code that is executed at compilation time. This SW code can be used for analysis or HW generation. In addition to the standard processes of RTL (e.g., the `always` blocks of Verilog HDL), HDLRuby also supports processes we call *sequencers* that describe finite state machines using SW-like constructs.

Nowadays, even the simplest devices include both HW and SW parts, which renders HW/SW co-design a necessity. HDLRuby supports HW/SW co-simulation through a construct, called the `program`, that executes C or Ruby SW code on the edge of a signal. Yet this construct is intended solely for embedding software code into RTL simulations. On the other hand, the sequencers, while similar to SW in appearance, remain HW components.

This paper focuses on embedded systems and presents a technique for generating SW — in Ruby or C — from HDLRuby sequencers. The generated code can then be executed within the HDLRuby RTL co-simulation tool to reduce validation time or used directly as the final SW to run on the target embedded system. Overall, this contribution provides:

- Functionally equivalent HW and SW code generated from the same sequencer description.
- Easy switch at any time during the design between HW and SW implementations.
- High-speed simulation for HDLRuby sequencers.

The rest of the paper is organized as follows: the next section presents some related works, then the contribution is presented in section III and section IV shows how to combine sequencer and `program` constructs for HW/SW smooth co-design. Then section V gives some experiments results and section VI concludes the paper.

II. Related Works

HW/SW co-design has been an important research topic for decades. Over the years, multiple approaches have been and are still proposed, [7, 10]. A major approach, called High Level Synthesis (HLS) [15, 18], gained some popularity and is now integrated into commercial synthesis tools [11, 14]. It consists of generating HW from SW code like C. HLS is promising, but it still cannot compete with RTL design for HW [15, 18] and it still lacks reliability [14, 16]. A few other approaches are using abstract languages like Matlab/Simulink or more HW-oriented languages like SystemC [18]. Some of the most recent works focus on using artificial intelligence for helping with HLS [12, 13]. Compared to these existing works, the approach of this paper is original in that it proposes to generate SW from HW descriptions. Generating SW from HW is not new, though, since approaches like [17] can be cited. However, the objective was different: it was to keep using legacy RTL code in a modern HW/SW environment.

Increasing the speed of RTL simulation and co-simulation is not new, and several approaches have been proposed, including abstracting interfaces [6], focusing on specific architectures [8] or generating aggressively optimized simulator code [9]. The originality of the approach proposed in this paper is that the optimization focuses on designated parts of the HW description only.

III. SW Implementation of HDLRuby Sequencers

From this point on, we call HW sequencer and SW sequencer the respective HW and SW implementations of sequencers. The objectives of the SW sequencers are to accelerate RTL simulation as well as easily switch between HW and SW during the design of an HW/SW device. Given these objectives, C may look as the natural target language for SW sequencers — and indeed, it is a supported target. Nevertheless, in this section, the focus is on the Ruby language. While dynamic languages like Ruby are not considered efficient in terms of speed, memory footprint, and power consumption, several arguments are in favor of using Ruby. First, Ruby benefits from an embedded system-oriented implementation, mruby[3]. Second, preliminary experiments indicated that if accuracy is to be preserved for the SW sequencer, the performance gap between C and Ruby is small. Third, integrating Ruby code within the HDLRuby framework is much easier than for C.

A. *About HDLRuby Sequencers*

HDLRuby sequencers were introduced to facilitate the design of finite state machines. All the usual SW constructs are reproduced in sequencers, but instead of executing SW instructions, they generate RTL descriptions of states and transitions of finite state machines. More precisely, the sequencers can include two kinds of statements: combinatorial statements and state statements.

The combinatorial statements comprise:

- The assignments using the arrow "<=" operator.
- The arithmetic and logic operations.
- The combinatorial conditional statements: `hif` for *if*, and `hcase`/`hwhen` for *case/when*.

In a sequencer, successive combinatorial statements are combined into a single dataflow whose outputs are stored in registers.

The state statements comprise:

- Explicit states: with the `step` keyword, which forces the beginning of a new state. It is used for splitting dataflows whose path is too long for a single clock cycle.
- State-based conditionals: `sif` for *if*, and `scase`/`swhen` for *case/when*. A state is created for each conditional branch.
- Loops: `sfor` for *for*, `swhile` for *while*, and `sloop` for infinite loops. A state is created for the body of the loops.
- Iterators: for applying algorithms on each element of a range, a bit-vector, or an array.

These constructs are all similar to sequential SW, but the generated HW remains parallel, so that no performance is lost compared to hand-made finite state machines. The generated finite state machine runs according to a clock signal, which is to be provided when declaring a sequencer. In addition, the start or restart of a sequencer execution is controlled by an additional reset signal.

```
sequencer(clk,start) do
  count <= 0
  sloop do
    swhile(button) { count <= count + 1 }
    result <= count
  end
end
```

Fig. 1. Example of a simple sequencer

For illustration, the code in Fig. 1 describes a simple sequencer, transitioning states on the rising edge of the signal `clk`, and starting execution when the signal `start` is set to one. This sequencer counts the clock cycles as long as a button is pressed and outputs its value to `result` when the button is released (in the example, all the signals are assumed to be declared earlier in the code).

Note: As in Ruby, code blocks can be delimited either by the { and } characters or by the `do` and `end` keywords.

While dedicated to describing finite state machines, sequencers can be used for describing various HW architectures. For example, a pipeline architecture can be described by writing the assignments in a loop in reverse order of dependency.

B. *The Technique used for Implementing the SW Sequencers*

The model of computation of HDLRuby sequencers is compatible with software and sequencer descriptions are, in practice, implemented as a set of Ruby methods. Hence, it would be straightforward to replace these methods for HW descriptions with ones that interpret the sequencers for SW execution. For example, `sif` would be interpreted as the execution of an if statement, `swhile` as a while statement, and so on. In addition, since the number of clock cycles required for each sequencer statement is known by construction, the total number of clock cycles can be counted for the SW implementation too, for obtaining an accurate estimate of the execution time of the corresponding HW sequencer.

This straightforward approach suffers from low performance, though, because method calling has a high performance cost in dynamic languages like Ruby. The technique proposed in this paper targets much better performance by relying on ahead of time translation: a first step generates high-performance Ruby code, or C code, before a second step that will execute the resulting code.

Contrary to HW sequencers, which run in parallel with the other components of the electronic device, SW sequencers' code must be explicitly executed through a function (or method) call. In Ruby, executing this code is done using the `call` method (or the `.()` operator). For example, the code of Fig. 2 describes two sequencers, one

```
1  [7].inner :val
2  my_seq = sequencer do
3    val <= 0
4    swhile(val<100) { val <= val + 1 }
5  end
6
7  sequencer do
8     val <= 100
9     swhile(val > 0) { val <= val - 1 }
10 end.()
11
12 my_seq.()
```

Fig. 2. Example of two SW sequencers

executed through the variable `my_seq`, and the other executed at declaration time. In the code, `val` is a 7-bit unsigned signal that is decremented from 100 to 1 in the second sequencer — executed first — and incremented up to 99 in the first one — executed afterward — (line 12).

Details about the code generation are given in a further subsection, after the handling of values, signals, and synchronizations is explained.

C. *Values and Signals for SW Sequencers.*

The possible bit widths for values in SW languages like C are usually limited (e.g., 32 bits), whereas in HW any bit width can be used. However, Ruby natively uses arbitrary-precision integers. Such integers can be used to model any bit width using bit masking techniques. For the sake of performance, we use them with a lazy approach that applies bit-width constraints only when strictly necessary, i.e., when performing comparisons or memory accesses — to handle cases of overflow — or when passing values outside the sequencers. For the comparisons, the bits exceeding the bit-width are masked out, and in case of signed types, the sign bit is reversed to normalize with unsigned comparisons. This processing is given by Eq. (1) where $\wedge$ is the bitwise and and $\oplus$ is the bitwise xor, and *width* is the bit-width of the value:

$$value \wedge (2^{width} - 1) \oplus 2^{width-1} \tag{1}$$

For memory accesses, the procedure is similar, but the sign is not processed, and the resulting index is considered as unsigned. For passing values outside sequencers, the same masking is used, but this time the sign is reencoded to match the target. For example, in Eq. (2), Ruby language-compatible sign is enforced.

$$\begin{array}{rl} value \wedge (2^{width} - 1) & if \quad value \wedge (2^{width-1}) = 0 \\ (value \wedge (2^{width} - 1)) - 2^{width} & if \quad value \wedge (2^{width-1}) \neq 0 \end{array} \tag{2}$$

This approach limits the type-specific processing to a minimum, taking advantage of the fact that Ruby already handles the majority of the work by default.

Note: For generating C code instead of Ruby, we can use an arbitrary-precision integer library like GMP [4], to achieve to same result.

The drawback of using arbitrary-precision integers is that bit values other than 0 or 1 (e.g., `Z` or `X`) are not supported. If a SW sequencer is meant to be used for fast HW simulation, it may be a limitation in some cases.

Sequencers are originally designed for HW and use signals. Yet, SW uses variables. For the sake of compatibility of both HW and SW sequencers, variables used in the latter are declared exactly like the signals of the former. The behavior of these SW "signals" is equivalent to their HW counterpart as long as they are used within a sequencer. For accessing them from external Ruby code, the `value` method is to be used. It ensures conversion between Ruby and Sequencer formats. For example, the value of `val` can be set to 50 in Ruby code as follows: `val.value = 50`.

D. *Synchronization of SW Sequencers*

HW sequencers, like the standard RTL processes, run in parallel with the rest of the circuit. However, each SW sequencer is executed to its end before the next SW code is executed. As long as the sequencers do not interact with others or their environment, the functional equivalence between SW and HW implementations is preserved. However, if a sequencer needs to share values with another component at a specific time, inconsistencies between the HW and SW implementations may occur.

One possible solution would be to synchronize the SW sequencers with their environment at each clock cycle, but it would defeat the purpose of achieving high performance and purely SW-based execution. Hence, a lazy approach is used again: no synchronization is done by default, but the user can insert synchronization points when required. For that purpose, a new sequencer command called `sync` is provided, which can be inserted in the code of a sequencer. As long as there is no such command, a SW sequencer runs standalone like any SW function. It is only when a `sync` is encountered that the underlying work for synchronization is performed.

To that end, the SW sequencers are implemented as fibers [5], i.e., very lightweight cooperative tasks. On low-end mono-processors, fibers are more efficient than threads, and in many cases, it remains better on high-end ones too. The function of the `sync` command is then to stop the execution of the current fiber, and consequently the current sequencer. It can be resumed afterward by calling for execution again using the `call` method (or the `.()` operator). For example, the code given in Fig. 3 describes two sequencers communicating in a ping-pong fashion 100 times.

In the example, `stimes` is an HDLRuby iterator, equivalent to the `times` iterator of Ruby. The loop of line 20 is Ruby code that controls the execution of the sequencers. It checks if either sequencer has completed using the `alive?` method. If not, the executions of `seq0` and `seq1` are resumed. Both sequencers stop their ex-

```
 1  inner :ping, :pong
 2  seq0 = sequencer do
 3    ping <= 0
 4    100.stimes do
 5      ping <= 1
 6      swhile(pong != 1) { sync }
 7      ping <= 0
 8    end
 9  end
10
11  seq1 = sequencer do
12    pong <= 0
13    100.stimes do
14      pong <= 1
15      swhile(ping != 1) { sync }
16      pong <= 0
17    end
18  end
19
20  while(seq0.alive? or seq1.alive?) { seq0.(); seq1.() }
```

Fig. 3. Example of two synchronised SW sequencers

```
 1  # Sequencer               # Generated Ruby code
 2  [32].inner :clk           $__clk ||= 0
 3  [8].input :din            __din = RubyHDL.din
 4  [8].output :dout          __dout ||= 0
 5  [8].inner :reg            __reg ||= 0
 6  sequencer(clk) do         Fiber.new do
 7    100.stimes do             __i = 100
 8                              while(i>0) do
 9    reg <= din                  __reg = __din
10      dout <= 0                 __dout = 0
11                                $__clk = $__clk + 1
12      swhile(reg !=0) do        while(__reg & 255 != 0) do
13                                  $__clk = $__clk + 1
14        hif(reg & 1) do           if(reg&1 != 0) do
15          dout <= dout + 1          __dout = __dout + 1
16        end                       end
17        reg <= reg >> 1           __reg = __reg >> 1
18      end                       end
19    end                       end
20  end                       end
21                            RubyHDL.dout = __dout & 255
```

Fig. 4. Sequencer and the corresponding SW code

ecution every time they encounter the sync command, respectively in lines 6 and 15.

If sync commands are correctly placed, the user can achieve functional equivalence between SW and HW implementations. Moreover, sync is ignored in HW sequencers, and conversely, the step command is ignored in SW sequencers — except for counting the clock cycles — so that both implementations can still coexist in a single description.

E. Code Generation for the SW Sequencers

In HDLRuby, the HW sequencers are implemented as a set of methods that generate the RTL code of state machines. The same can be done for SW sequencers by replacing the library of RTL code generators with Ruby or C code generator. The resulting SW code can then be executed for simulation or saved in a separate file for being executed on an embedded processor. In case of Ruby code, this code can be executed using mruby[3], for example.

Generating Ruby code from the sequencer is straightforward, since all the concepts used, e.g., arbitrary-precision integers or fibers, are natively supported by Ruby. Generating the clock cycle count is done using a global variable which is increased each time a state transition is expected for the corresponding HW sequencer.

However, maintaining high performance is crucial. Nowadays languages like Ruby benefits from JIT [2] so that the interpretation cost is low. Additionally, this language is efficient at handling the stack and arrays. Its handling of arithmetic and logic computation suffers from its use of arbitrary-precision integers, but the SW sequencers use them for efficient bit-level processing. However, message passing (i.e., method or function calls), as well as object's variables accesses, are inherently slow because dynamic languages like Ruby must look up their names at run time. By contrast, local variables are stored on the stack and can be accessed quickly.

Therefore, the priority when generating code is to use as many local variables as possible and to limit method calls to the strict minimum. In practice, all the signals in a sequencer, as well as the loop indexes, are converted to local variables. Regarding memories, converting them to Ruby arrays is enough, since these data structures are optimized. No bound check is required: Ruby returns nil values in case of out-of-bound accesses, which is enough for detecting an error. The handling of computations and synchronizations has already been presented in the previous sections. Fig. 4 shows an example of a sequencer and the resulting optimized Ruby code when the clock cycles are counted. In the figure, din is an input signal and dout is an output signal for the circuit. Both are transmitted from and to the environment using the RubyHDL package (details are skipped for the sake of brevity) as shown in lines 3 and 21.

The same approach can be used for C code generation and has been implemented. Two possible implementations have been considered. One, a pure C version: used when signals are guaranteed to be smaller than 64 bits. Experiments show this version is the fastest by at least one order of magnitude as long as no synchronization command (sync) is used. Two, C with GMP version: GMP [4] is a library for arbitrary-precision integer computation.

Finally, we opted for version one for two main reasons: (1) GMP is not a standard C library, and (2) the performance difference between Ruby and C when using GMP is small.

Note: For the implementation of the sync command a C fiber library like Libco is considered.

```
1  ar = [clk,start]
2  program(ruby,run) do
3    actport start.posedge
4    inport din: din
5    outport dout: dout
6     code do
7       activate_sequencer_sw(binding)
8       input :din
9       output :dout
10      ar = []
11
12      my_seq = sequencer(*ar) do
13        dout <= din
14      end
15
16      my_seq.()
17    end
18 end
```

Fig. 5. Syntax of a program component

IV. SW Sequencers for Co-Design

The SW code generated from SW sequencers can be used in co-simulation environments like any other SW component. This includes HDLRuby's co-simulation environment, which uses the `program` construct. This construct can execute an external SW file, on a rising or falling edge of a signal, declared using the command `actport`, and supports data transfer between HW and SW through registers declared by the `inport` and `outport` commands. If the SW language is Ruby, the code can be written inline in the `program` construct.

For easing the integration of SW sequencers with co-simulation, a Ruby command has been added that allows converting sequencers to Ruby within a HDLRuby `program` construct. With this command, a SW sequencer can be used as is, and it becomes easy to explore HW/SW partitioning, as well as speeding up only parts of a HW implementation: it is enough to comment in or out the `program` description. Fig. 5 gives such an example. In the figure, sequencer `my_seq` returns to `dout` the values it receives from `din`. It is implemented as a SW sequencer executed in co-simulation, but if the program part is commented out, lines 2-11 and 15-17, it becomes a HW sequencer. Line 7 is the command that gives support to the automatic generation and execution of SW code from the sequencers by the co-simulation engine.

Note: The clock and start signals, respectively `clk` and `start` required for the HW sequencer are provided through the array `args`, which is overwritten in the program code on line 10.

V. Experiments

For the experiments, we implemented sequencers for several benchmarks, including sample codes to individually check specific cases, as well as several applications. We compared their simulation or execution speed in HW, Ruby SW, and C SW implementations. The samples include checks on arithmetic operators (*arith*), logic operations (*logic*), control statements (*ctrl*), and memory accesses (*mem*). The applications include the following: rectangular (*recwav*), sawtooth (*sawwav*), triangular (*triwav*) and sine (*sinwav*) sound wave generation, gaussian (*gauss*) and sobel (*sobel*) image filtering, greater common divider (*gcd*) and modular exponentiation (*modexp*). The simulations have been done on an iMac 15,5 computer with an Apple M3 8-core 4.05 GHz processor.

For the C version of the SW sequencers, the resulting C code must be compiled separately, whereas the Ruby version could be used seamlessly using the HDLRuby standard simulation tool. Moreover, various easy-to-use standard libraries are available in Ruby for loading or storing images and sounds that could be used for both HDLRuby HW and Ruby SW sequencers. Doing the same with C requires additional installation and compilation steps. For design exploration, the ease of use of the HDLRuby/Ruby environment is an important advantage.

Table I summarizes the execution times of each benchmark for HDLRuby RTL simulation, Ruby SW sequencers, and C SW sequencers. For a better grasp of the actual performance, we compared the performance of the HDLRuby RTL simulator with that of Icarus Verilog in a previous publication [19], which showed that the HDLRuby simulator was generally faster. In the table, N/A indicate the result could not be obtained, either for lack of memory for RTL, or non-supported bit-width for C. While the results vary a lot depending on the kind of computation, SW sequencers in Ruby are, in the majority of cases, more than one order of magnitude faster than the simulation of HW sequencers. At the same time, the SW sequencers in C are in a majority of the cases, more than one order of magnitude faster than the ones in Ruby. Still, SW sequencers in Ruby are fast enough for reasonable time evaluation of the HW, e.g., a convolution on a 1024x1024 image only took about 4 seconds. Moreover, the C versions only work if the used bit width of each signal is inferior to 64, e.g., *gcd* for 200 and 1024 bits were not supported by the C SW sequencers. These performance considerations, combined with the fact that the Ruby SW sequencers can be used inline in the co-simulation constructs of HDLRuby makes us think that this should be the default choice for accelerating RTL simulation, switching to C only for very specific heavy computation cases, like *modexp* (module exponentiation), whose 32-bit version could not be simulated in a reasonable time with Ruby.

As a preliminary estimation for selecting the computation method for C code, the *arith* sample has also been implemented using the GMP library for supporting any bit width. The execution time was 21.96s, i.e., more than twice the Ruby version. While optimizations may be possible, the benefits of using C for full bit width support are limited compared to using Ruby, especially if we consider the flexibility of the latter.

TABLE I
SIMULATION SPEED RESULTS

Algorithm / Size	RTL Time	Ruby Time	C Time
arith / $3*10^9$ ops.	124.6s	10.62s	0.81s
logic / $3*10^9$ ops.	65.1s	14.78s	7.79s
ctrl / $9*10^6$ ops.	2.70s	0.21s	0.004s
mem / $5*10^8$ ops.	329.6s	3.89s	0.48s
recwav / 1s sound	0.51s	0.11s	$1.62*10^{-3}$s
sawwav / 1s sound	0.52s	0.11s	$1.30*10^{-3}$s
triwav / 1s sound	0.49s	0.11s	$1.44*10^{-3}$s
sinwav / 1s sound	0.66s	0.11s	$1.06*10^{-3}$s
gauimg / 256x256	15.87s	0.33s	$0.36*10^{-3}$s
sobimg / 256x256	16.16s	0.34s	$0.74*10^{-3}$s
gauimg / 1Kx1K	N/A	4.06s	$5.69*10^{-3}$s
sobimg / 1Kx1K	N/A	4.15s	$13.69*10^{-3}$s
gcd / 32 bit	0.12s	0.06s	$2.41*10^{-7}$s
gcd / 200 bit	0.12s	0.06s	N/A
gcd / 1024 bit	0.15s	0.06s	N/A
mexp / 24 bit	200.60s	13.32s	0.14s
mexp / 32 bit	N/A	4324.38s	190.83s

VI. Conclusion

This paper presented a new hardware/software (HW/SW) co-design technique for HDLRuby, which consists of generating a SW implementation of the sequencer construct, called SW sequencer. A sequencer is a kind of process specific to HDLRuby that allows simple design of finite state machines. The SW sequencer can be executed on a processor, for a HW/SW device, or used for fast simulation, where the exact number of clock cycles can still be measured. Two versions were developed: a Ruby-based implementation and a C-based one. The Ruby version supports signals of any bit-width and is well integrated within the HDLRuby environment, whereas the C version is faster, but is limited to a maximum 64-bit signal widths. Switching between HW and SW implementation at any stage of the design is straightforward and safe, requiring no modification of the sequencer code. Moreover, the Ruby SW sequencers can be used inline in the HDLRuby co-simulation environment, so that switching between HW and SW requires only minor declaration changes in the embedding RTL.

Experimental results show that Ruby SW sequencers are typically more than one order of magnitude faster than RTL simulation and require little enough memory to simulate large designs. The C implementation further accelerates execution by often more than one additional order of magnitude. Although the Ruby SW sequencers are significantly slower than the C ones, they remain reasonably fast, and in a majority of cases, they are enough for simulating large devices, while benefiting from their higher flexibility and integration.

Future works will focus on simplifying further the integration of SW sequencers in the co-simulation environment of HDLRuby, and on supporting 4-state logic in SW sequencers simulation, only when required for functional accuracy, without compromising performance.

Acknowledgements

This research is funded by the Japanese Grants-in-Aid for Scientific Research (KAKENHI) No. 22K11965.

References

[1] "HDLRuby: a Hardware Description language based on Ruby" `https://github.com/civol/HDLRuby`

[2] Y. Matsumoto "The Ruby Programming Language," `http://https://www.ruby-lang.org/`

[3] "mruby, the lightweight implementation of the Ruby language," `https://mruby.org/`

[4] "The GNU Multiple Precision Arithmetic Library" `https://gmplib.org/`

[5] "Fibers in Ruby" `https://docs.ruby-lang.org/en/master/Fiber.html`

[6] Y. Sungjoo, A. A. Jerraya. "Hardware/software cosimulation from interface perspective," *Computers and Digital Techniques*, vol. 152.3, pp. 369–379, 2005.

[7] D. M. Giovanni, R. K. Gupta "Hardware/Software Co-Design," *Proceedings of the IEEE*, vol. 85.3, pp. 349–365, 1997.

[8] A. Akram, L. Sawalha, "A survey of computer architecture simulation techniques and tools," *Ieee Access*, vol. 7, pp. 78120–78145, 2019.

[9] "Verilator, the fastest Verilog/SystemVerilog simulator." `https://www.veripool.org/verilator/`

[10] A. Sampson, J. Bornholt, L. Ceze, "Hardware–Software Co-Design: Not Just a Cliché," *International Journal of Emerging Trends in Computer Science and Information Technology*, pp. 262–273, 2015.

[11] R. Nane, V. M. Sima, C. Pilato, J. Choi, B. Fort, A. Canis, Y.T. Chen, H. Hsiao, S. Brown, F. Ferrandi, J. Anderson, "A survey and evaluation of FPGA high-level synthesis tools," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 35.10, pp. 1591–1604, 2015.

[12] S. Dai, Y. Zhou, H. Zhang, E. Ustun, EFY. Young, Z. Zhang, "Fast and accurate estimation of quality of results in high-level synthesis with machine learning," *IEEE 26th Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 129–132, 2018.

[13] Y. Liao, T. Adegbija, R. Lysecky, "Are llms any good for high-level synthesis?," *43rd IEEE/ACM International Conference on Computer-Aided Design*, pp. 1–8, 2015.

[14] Y. Herklotz, Z. Du, N. Ramanathan, J. Wickerson, "An empirical study of the reliability of high-level synthesis tools," *IEEE 29th Annual International Symposium on Field-Programmable Custom Computing Machines*, pp. 219–223, 2021.

[15] BC. Schafer, Z. Wang, "High-level synthesis design space exploration: Past, present, and future," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 39.10, pp. 2628–2639, 2019.

[16] N. Pundir, F. Farahmandi, M. Tehranipoor, "Secure high-level synthesis: Challenges and solutions," *22nd International Symposium on Quality Electronic Design*, pp. 164–171, 2021.

[17] M. I. Rashid, B. C. Schafer, "MIRROR: MaxImizing the Reusability of RTL thrOugh RTL to C CompileR," *Design, Automation & Test in Europe Conference*, pp. 1–6, 2023.

[18] S. Lahti, P. Sjövall, J. Vanne, TD. Hämäläinen, "Are we there yet? A study on the state of high-level synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 38.5, pp. 898–911, 2018.

[19] "Implementation and Comparison of Several Register Transfer Level Simulation Engines for the HDLRuby Language" *Proceedings of the 11th IIAE International Conference on Industrial Application Engineering*, 2023.