A Java Backend for ESSENT

Augie Eriksson and Maanuj Vora
Department of Electrical Engineering and Computer Science
University of California, Berkeley
{raeriksson, maanujvora}@berkeley.edu

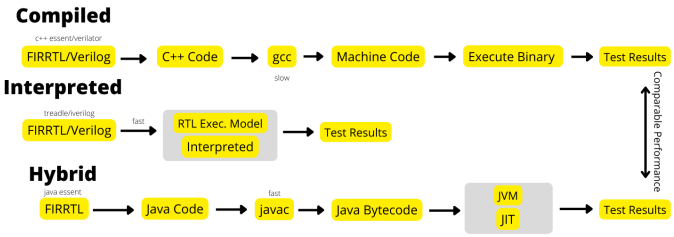

Fig. 1: Types of hardware simulators

***Abstract*—We propose a new open-source RTL simulator that achieves faster compilation and startup speed compared to compiled simulators, such as Verilator, but also achieves higher simulation performance than interpreted simulators, such as treadle [1]. We build on the work in ESSENT [2], an optimizing open-source FIRRTL simulator, which has an existing C++ backend, and extend it with a new Java backend. We take advantage of fast Java bytecode compilation and the JVM's JIT compilation to simultaneously deliver fast simulator startup time and high simulation throughput.**

I. Introduction

RTL simulators fall in one of two categories (see Figure 1):

1) Compiled simulators: RTL is turned into an optimized C++ model, and gcc is invoked to compile the model to a binary. Examples include Verilator, VCS, and ESSENT.
2) Interpreted simulators: RTL is parsed into an in-memory model. The simulator runtime interprets that model. Examples include treadle (an interpreted FIRRTL simulator) and Icarus Verilog.

We build our hybrid approach on the ESSENT FIRRTL simulator. ESSENT ingests FIRRTL and partitions the RTL into subgraphs that are executed only when their inputs change. ESSENT emits C++ code and invokes gcc to compile the code to a simulation binary, similar to Verilator.

Invoking gcc for compilation and linking introduces a significant startup overhead. To mitigate this, we designed a Java code emitter for ESSENT, which reuses ESSENT's optimization passes. To run an RTL testbench, we load the compiled class files into a JVM and invoke a test written in Java.

II. Benchmarks

We evaluated four simulators: Verilator, treadle, ESSENT (with default C++ and custom Java backends). On these simulators, we evaluated four designs with corresponding test benches: a GCD module, the bit-serial RISC-V core Serv [3], a TileLink RAM, and the riscv-mini 3-stage RISC-V core [4].

	Java ESSENT		C++ ESSENT		Verilator		treadle
	CG	Comp	CG	Comp	CG	Comp	
GCD	1.3	0.3	1.4	0.4	1.5	1.1	1.1
TL RAM	2.1	0.6	2.3	0.8	2.4	1.5	1.9
SERV	1.9	0.5	2.2	0.7	2.1	1.6	1.7
riscv-mini	2.7	0.6	3.0	1.6	2.9	2.7	2.5

TABLE I: Compilation times (in seconds). CG = code generation time, Comp = compilation time

	Java ESSENT	C++ ESSENT	Verilator	treadle
GCD	0.4	0.1	0.1	9.8
TL RAM	4.5	0.4	0.4	100
SERV	0.8	0.4	0.3	29
riscv-mini	0.4	0.02	0.01	12

TABLE II: Simulation times (in seconds)

A. Compilation Performance

We benchmark the startup latencies in Table I. For the compiled simulators, we measure code generation time (compiling FIRRTL into intermediate code (C++ / Java)), and compilation time (compiling intermediate code into a runnable binary or bytecode). For the interpreted simulator, we measure the time to load a FIRRTL file and be ready to run testbench code.

Java ESSENT is faster to spin up than the other compiled simulators, and achieves similar startup time to treadle.

B. Simulation Performance

We also benchmark the testbench execution times (Table II). Java ESSENT achieves simulation performance comparable to other compiled simulators, while having a low startup time.[1]

III. Conclusion

Java ESSENT delivers the "best of both worlds" for short to medium length tests on moderately sized designs. This is achieved by utilizing the Java Compiler for fast compilation and the Java JIT for optimized execution. We are optimizing to increase simulation speed, and adding VCD dumping and chiseltest integration.

Find our open source repository here (https://github.com/ekiwi/essent/tree/java-backend).

References

[1] C. Markley, https://github.com/chipsalliance/treadle
[2] S. Beamer and D. Donofrio, "Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation," DAC 2020.
[3] O. Kindgren, https://github.com/olofk/serv
[4] D. Kim, https://github.com/ucb-bar/riscv-mini

[1]The slower execution in TL RAM and riscv-mini are due to inefficiencies in wide signal operations that will be optimized in the future.