POSTER: Tango: An Optimizing Compiler for Just-In-Time RTL Simulation

Blaise-Pascal Tine, Sudhakar Yalamanchili, Hyesoon Kim, Jeff Vetter
Georgia Institute of Technology, Oak Ridge National Laboratory
btine3@gatech.edu, sudha@gatech.edu, hyesoon@cc.gatech.edu, vetter@ornl.gov

***Abstract*—The end of Moore's law with the advent of hardware specialization presents a unique challenge for a much tighter software and hardware co-design environment to exploit domain-specific optimizations and increase design efficiency. The productivity of software-hardware codesign relies not on only in better integration between the software and hardware design methodologies but more importantly in the effectiveness of the design tools at reducing the development time. In this work, we developed Tango, an Optimizing compiler for a Just-in-Time RTL simulator. Tango implements unique hardware-centric compiler transformations to speed up runtime code generation in a software-hardware codesign environment where hardware simulation speed is critical. Tango achieves a 3x average speedup compared to the state-of-the-art RTL simulators.**

***Index Terms*—hardware description language, Just-in-time compilation, hardware simulation.**

I. Introduction

Hardware-software codesign methodology is about improving the integration between software and hardware development processes with the goal of reducing the total development cost. The traditional approaches [1] have mainly looked at addressing the high cost of hardware verification by integrating that process early during software development with higher hardware modeling abstractions to balance accuracy versus productivity. RTL simulation remains one of the most important steps in hardware verification for guaranteeing the quality of the final design, but it is very time-consuming, mainly due to the complexity of emulating hardware behavior at the RTL description level.

Several solutions have been proposed to improve the performance of RTL simulation [2] using event-driven simulation to schedule the execution of various components in the hardware when a change of properties affecting the component occurs. This certainly carves out a large portion of the RTL simulation performance bottlenecks, however, these solutions do not look at improving actual low-level code-generation, relying on the effectiveness of existing compilers to generate the final binary.

We present Tango, an optimizing just-in-time (JIT) compiler for RTL simulation. Tango was designed to be used as a back-end of hardware construction languages (HCL) [3] enabling direct high-speed simulation and debugging of described models. Tango implements unique compiler transformations based on hardware-centric information in its IR that provides significant speed improvement for RTL simulation. The main contributions of this paper are the following: (1) We introduce Tango just-in-time compilation infrastructure, highlighting the major transformations from its high-level IR to executable binary, (2) introduce proxy coalescing dataflow optimization for eliminating hidden indirections in the RTL netlist, and (3) We introduce new hardware-centric codegen optimization techniques for lowering sequential nodes, shift registers, and switch tables.

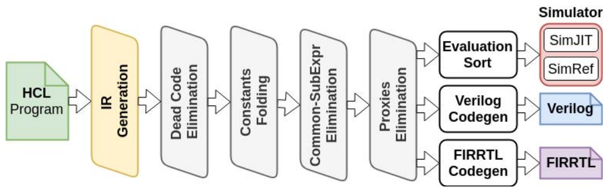


Fig. 1: Tango Compiler Infrastructure

II. Tango Back-end Compilation

Figure 1 illustrates the different phases of Tango compilation system; The first stage parses the program by recursively traversing the netlist in the source program to generate a Dataflow IR for the following optimization stages. Stages two, three, and four perform standard compiler optimizations on the Graph IR, which include dead-code elimination (DCE), constant folding (CFO), and common sub-expression elimination (CSE). Stage five, proxies elimination (PCX), implements our custom transformation to prune out hidden indirection in the graph. After the graph optimization phase, the resulting IR can be used to export Verilog or FIRRTL [3] to use with other EDA tools for synthesis to FPGAs or ASIC. The IR is also converted to native instructions for runtime simulation, using JIT (SimJIT), or non-JIT (SimRef) for platforms where JIT is not supported.

A. Tango IR Description

Tango IR captures the connections between the various hardware blocks in the source hardware description. Those hardware blocks include I/O ports, registers, RAM blocks, arithmetic operations, multiplexers, etc. Nodes performing bit slicing or concatenation are named 'proxy' nodes.

978-1-7281-3613-4/19/$31.00 ©2019 IEEE
DOI 10.1109/PACT.2019.00055

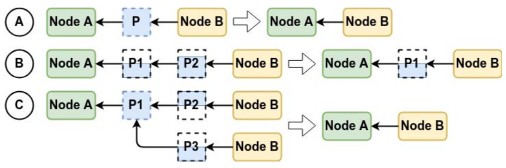


Fig. 2: Proxies Elimination

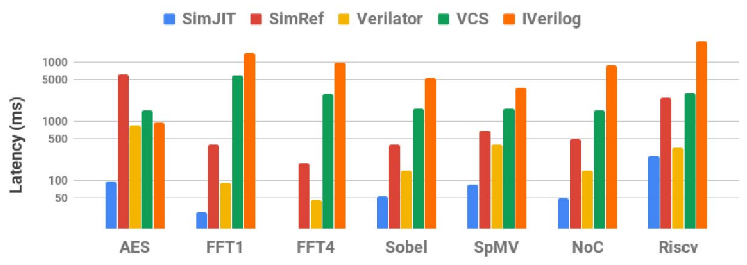
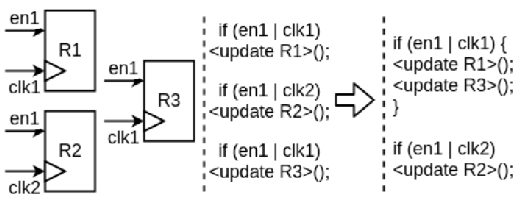


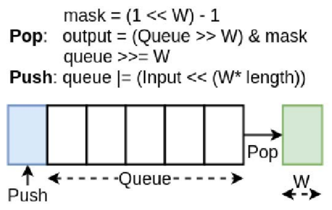
Fig. 3: Runtime Latency Comparison



```
if (en1 | clk1)
<update R1>();

if (en1 | clk2)
<update R2>();

if (en1 | clk1)
<update R3>();
```

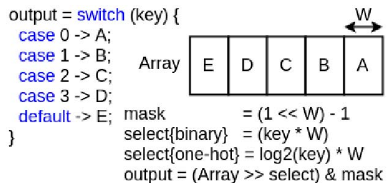
```
if (en1 | clk1) {
<update R1>();
<update R3>();
}

if (en1 | clk2)
<update R2>();
```

Fig. 4: Coalescing Register Update



```
      mask = (1 << W) - 1
Pop:  output = (Queue >> W) & mask
      queue >>= W
Push: queue |= (Input << (W* length))
```

Fig. 5: Shift Register Lowering



```
output = switch (key) {
  case 0 -> A;
  case 1 -> B;
  case 2 -> C;
  case 3 -> D;
  default -> E;
}
```

```
mask            = (1 << W) - 1
select{binary}  = (key * W)
select{one-hot} = log2(key) * W
output = (Array >> select) & mask
```

Fig. 6: Switch Table Lowering

B. *Proxy-Coalescing Optimization (PCX)*

Redundant operations can exist across proxy nodes which can introduce unnecessary computation during simulation. Figure 2 shows the three transforms executed during this stage: (A) eliminates proxies that are direct copies of their source node, (B) eliminates proxies that are used as source to other proxies and have a bit range that supersedes those destination proxies, and (C) coalesces proxies that do not overlap and when merged together can produce the full bit range of the destination node.

C. *Sequential Node Coalescing (SNC)*

This optimization (see Figure 4) traverses the graph IR sorted in topological order, grouping sequential nodes (reg, mem) into shared control flow blocks based on their shared signals (clk, reset, enable) and determines the optimal scheduling of these nodes's update logic that reduces the number of control flow blocks.

D. *Shift-Register Optimization (SRO)*

SRO attempts to pack for small shift registers into single integer scalars using shift arithmetic as illustrated in Figure 5. A *Push* operation add a new entry to the scalar queue and the *Pop* operation removes it. Shift registers are prevalent in streaming accelerators such as encryption, compression, FFT engines.

E. *Switch Table Optimization (SWO)*

SWO attempts to pack small switch statement tables into single integer scalars using shift arithmetic as illustrated in Figure 6, therefore avoiding unnecessary branches generation. This compaction technique supports both binary and one-hot switches. SWO particularly affects hardware blocks with control logic such as FSMs, encoders, multiplexers, crossbars.

III. Results

We evaluated our optimizations using a benchmark with a diverse mix of hardware components including (1) 128-bit Advanced Encryption Standard Encryption engine (AES), (2) single-path delay radix 2^2 FFT, (3) 4-lanes parallel radix 2^2 FFT, (4) 8-bit pipelined Sobel filter (Sobel), (4) 16-bit fixed-point Sparse Matrix Multiplier (SpMV), (5) 32-nodes Network-on-Chip router (NoC), (6) 5-stage RiscV RV32I processor core (RiscV). We ran our experiment on Intel Xeon E5-4610 processor with 32KB L1, 256KB L2, 16384KB L3 caches, and 64 GB of system memory. We compared Tango simulator performance with prominent RTL simulators including Icarus Verilog (IVerilog [4]), the industry-grade Synosys VCS [5], and state-of-the-art open-source simulator Verilator [6].

Figure 3 shows the average runtime latency on all simulators. SimJIT is on average 3.8x faster than Verilator, 72.8x faster than VCS and 225x faster than IVerilog across all models. Also, SimRef is on average 5.8x faster than VCS and 18.7x faster than IVerilog across all models. We observed that PCX optimization contributed for about 50% of the overall speedup, followed by SNC with 20%, SRO 17%, and SWO 13%.

References

[1] D. Kroening and N. Sharygina, "Formal verification of systemc by automatic hardware/software partitioning," in *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, MEMOCODE.*, pp. 101–110, 2005.

[2] F. H. Alexey Kupriyanov and J. Teich, "High-speed event-driven rtl compiled simulation," 2004.

[3] P. S. Li, A. M. Izraelevitz, and J. Bachrach, "Specification for the firrtl language," Tech. Rep. UCB/EECS-2016-9, EECS Department, University of California, Berkeley, Feb 2016.

[4] S. Williams, "Icarus verilog." http://iverilog.icarus.com.

[5] S. inc., "Vcs: Industrys highest performance simulation solution." https://www.synopsys.com/verification/simulation/vcs.html.

[6] W. Snyder, "Verilator." https://www.veripool.org/wiki/verilator.