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

Blaise-Pascal Tine
Georgia Tech
Atlanta, Georgia
blaise.tine@gatech.edu

Sudhakar Yalamanchili
Georgia Tech
Atlanta, Georgia
sudah@gatech.edu

Hyesoon Kim
Georgia Tech
Atlanta, Georgia
hyesoon@cc.gatech.edu

***Abstract*—With Moore's law coming to an end, 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. This trend is further accentuated by rapid-pace of innovations in Machine Learning and Graph Analytic, calling for a faster product development cycle for hardware accelerators and the importance of addressing the increasing cost of hardware verification. The productivity of software-hardware co-design relies upon better integration between the software and hardware design methodologies, but more importantly in the effectiveness of the design tools and hardware simulators at reducing the development time. In this work, we developed Tango, an Optimizing compiler for Just-in-Time RTL simulation. Tango implements unique hardware-centric compiler transformations to speed up runtime code generation in a software-hardware co-design environment where hardware simulation speed is critical. Tango achieves a 6x average speedup compared to the state-of-the-art simulators.**

I. Introduction

The end of Moore's law is motivating the push for hardware specialization and systems-on-chip as a solution for scaling, delivering orders of magnitude performance and power benefits compared to traditional general-purpose architectures. But this transition is slowed down by the high cost of hardware verification [1] which accounts for more than 40% of the total hardware development cycle. The traditional approaches at addressing this problem have mainly focused on integrating hardware verification early during software development with various hardware modeling levels 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 this process is very time-consuming, mainly due to the complexity of emulating hardware behavior at low-level register states transitions.

Several solutions have been proposed to improve the performance of RTL simulation [2] [3] [4] 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 single-thread low-level code-generation, relying on the effectiveness of existing compilers for generating the final binary.

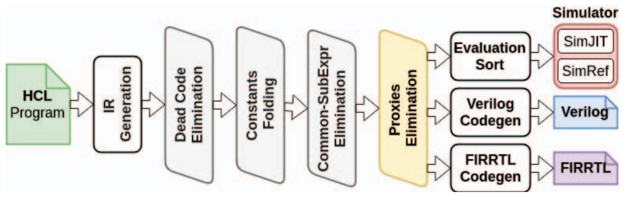


Fig. 1: Tango Compiler Infrastructure

We present Tango (Figure 1), an optimizing just-in-time (JIT) RTL simulator compiler. Tango was designed to be used as a back-end of any hardware construction language (HCL) [5] [6] to enable direct high-speed simulation and debugging of hardware models. Tango code generator uses LLVM to implement runtime code specialization optimized for the target architecture.

Tango implements an RTL-friendly IR that captures high-level abstractions of the described hardware blocks to lower them directly into native instructions for runtime execution. Tango identifies unique compiler transformations based on hardware-centric information in its IR that provides significant speed improvement during RTL simulation. The main contributions of this paper are the following:

- We introduce Tango just-in-time compilation infrastructure, highlighting the major transformations from a high-level hardware description to its final executable.
- We introduce proxy coalescing dataflow optimization for eliminating hidden indirections in the RTL code.
- We introduce new hardware-centric codegen optimization techniques for lowering shift registers, sequential nodes, and switch tables.
- Tango simulation achieves a 6x average speedup over state-of-the-art simulators.

II. Background and Related Work

A. Event-Driven Hardware Simulation

Prominent RTL simulators [7] [8] [9] employ event-driven simulation as their simulation methodology to evaluate RTL netlists during verification. Event-driven simulation provides an efficient mechanism for processing large and complex networks like RTL netlist by only scheduling the execution of components whose inputs have changed after

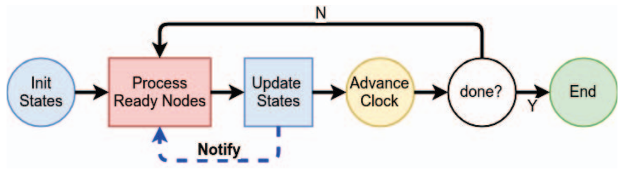


Fig. 2: Event-Driven Simulation Loop

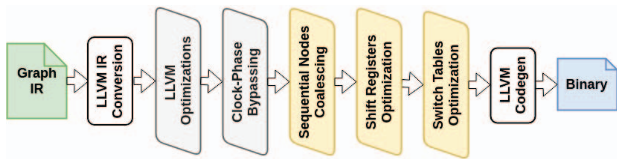


Fig. 3: SimJIT Optimization Pipeline

each iteration. Figure 2 illustrates a typical event-driven simulation loop. The RTL simulator starts by initializing all the states in the system, the loop begins with the process stage where all nodes whose input signals have been asserted in the prior iteration are scheduled for execution, then an update stage follows where changes are processed and affected nodes are notified, then the clock is updated to advance the simulation. This loop iterates until the desired simulation time is reached or a custom event occurs.

The time-consuming portion of the RTL simulation is generally in the process stage where updated nodes are evaluated. Prior works [10] [11] have looked into partitioning the computational graph to distribute it on parallel architectures like GPUs which can provide effective performance improvement for large models, others [12] have explored inputs prediction to improve distributed parallel simulation. In this work, we are particularly interested in single-threaded CPU performance. Prominent RTL simulators in use today translate the operations in the RTL netlist into native C++ leaving it to the native compiler to lower the source code into native binary for execution, leaving potential opportunities for optimization that the native compiler cannot perform since not having access to additional context information only available in the original RTL source. Some of these context information includes the original bit-size of the data types, their scope, their update semantic, and aliasing information. Tango optimizing compiler leverages some of this information to improve the performance of RTL simulation.

B. Related Work

1) Verilator [13] is an open-source RTL simulator that that translates Verilog [14] into a C/C++ program that the user can compile with any native compiler for execution. It implements some high-level transformations on the Verilog netlist to partition the computation graph and reduce the update cost during the simulation. To the best of our knowledge, Verilator is currently the fastest RTL simulator in use today and has shown great adoption with recent hardware construction languages such as Chisel [6], PyMTL [5]. Verilator's main speed advantage comes from the fact that it only simulates binary signal states '01' instead of using the four '01XZ' Verilog states, allowing it to efficiently map RTL primitives to C++.

2) Icarus Verilog [9] is an open-source RTL simulator for Verilog. It supports the full four-state Verilog model giving the simulator an edge for accuracy over Verilator. Icarus Verilog doesn't use native compilation to generate its binaries, it uses highly optimized code blocks for the various RTL primitives in the Verilog Netlist, allowing it to execute interpreted code at a reasonable speed.

3) Synopsys VCS [8] is a proprietary RTL simulator for Verilog. It is an industry-standard fast simulator with support for the complete Verilog specification. VCS uses native compilation to lower its netlist into an executable which allows it to leverage additional low-level optimizations to improve its performance. VCS is on average 10x faster than Icarus Verilog.

4) SystemC [7] is a C++ library for hardware modeling and simulation. It supports multiple modeling abstraction levels which include RTL level, System level, and Functional level. SystemC is widely used for systems and functional modeling where it excels remarkably because of its runtime efficiency and its effective application in software-hardware co-design. SystemC implements an event-driven simulator based on user-defined processes and threads emulating the parallel behavior of the target hardware and a sensitivity list of signals that trigger the processes to execute when the value of a signal change. SystemC simulator is efficient at rendering very large models leveraging its built-in threading system. The simulator has a high setup cost that makes it sub-optimal for average-size models.

III. Tango Compilation Pipeline Overview

A. Tango Compilation Pipeline Overview

Figure 1 illustrates the different phases of Tango compilation pipeline; The first stage, IR Generation, converts a given hardware block description into Tango's intermediate representation (IR) suitable for optimization in subsequent 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 indirections in the Graph IR. After the graph optimization phase, the resulting pruned IR can be used to generate Verilog or FIRRTL [6], suitable for export to other EDA tools.

The last stage of the pipeline implements a JIT simulator for RTL, SimJIT, that consumes the optimized IR and converts it into native code for runtime execution. There is also an optional non-JIT simulation engine, SimRef, for platforms where JIT is prohibited or the target architecture is not supported by the JIT engine. SimJIT is implemented using LLVM compiler infrastructure to leverage its efficient code generator and its flexible extension API for adding new optimization passes. Figure 3 illustrates the optimization pipeline inside SimJIT, starting with optimized

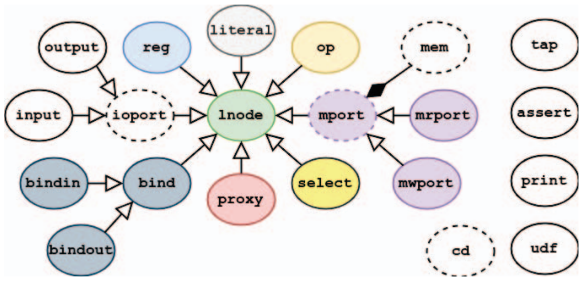


Fig. 4: The Tango IR Data Structure

Graph IR being converted to LLVM IR a part of the first stage. Then SimJIT applies all LLVM optimizations on the generated IR in the second stage. In the third stage, SimJIT implements Clock-Phase Bypassing optimization to only update the netlist on clock transitions. Stages four-five-six implement our custom optimizations, Sequential Nodes Coalescing (SNC), Shift Registers optimization (SRO), and Switch Table Optimization (SWO) which we will describe in the following section.

B. The Tango IR Description

Figure 4 summarizes the data-structure of the Graph-IR capturing the connections between hardware primitives contained in the source hardware block description. *lnode* is the base element from which all other nodes in the Graph IR inherit from. The IR node types include I/O ports, registers, memory ports, literals. Arithmetic operations are captured by the *op* nodes, Muxes are captured by the *select* nodes, Bits slicing, subscript, and concatenation operations are captured by *proxy* nodes, and sub-modules binding by *bind* nodes. There is a dedicated clock domain node, *cd*. User-defined functions also have their IR node, *udf*. Debugging functions for tap, assertion, and printing are captured by the *tap*, *assert*, and *print* nodes respectively. A unique feature of this IR is the addition of high-level abstractions like shift registers, memory blocks, switch statements as extensions of the existing nodes to provide additional information that are useful for the graph optimizer and code generator.

C. Walk-thru Fifo Example

To model various hardware designs targeting Tango IR, we implemented a C++ DSL front-end for Tango that easily maps the Graph IR to a programmable API supporting C++ generics and compile-time reflection. The detail about the software implementation of this library is outside the scope of this paper. Listing 1 illustrates the design of generic Fifo using the C++ DSL. Our generic Fifo is modeled using a C++ template class, taking a data type *T* and the number of entries *N* as template parameters. The I/O ports are described as member variables using *__in* and *__out* attributes to provide their input and output directions (see lines 4-9). The *describe()* method implements the actual body of our hardware block. Reading the code, we are able to identify some hardware primitives available in the Tango IR, including registers *ch_reg* (line 13), memory

```
1  template <typename T, int N, int A = log2ceil(N)>
2  class Fifo {
3    __io(
4      __in (T)         din,
5      __in (ch_bool)   push,
6      __in (ch_bool)   pop,
7      __out (T)        dout,
8      __out (ch_bool)  empty,
9      __out (ch_bool)  full
10   );
11
12   void describe() {
13     ch_reg<ch_uint<A+1>> rd_p(0), wr_p(0);
14
15     auto rd_a = ch_slice<A>(rd_p);
16     auto wr_a = ch_slice<A>(wr_p);
17
18     auto read = io.pop && !io.empty;
19     auto write = io.push && !io.full;
20
21     rd_ptr->next = ch_sel(read, rd_p + 1, rd_p);
22     wr_ptr->next = ch_sel(write, wr_p + 1, wr_p);
23
24     ch_mem<T, N> mem;
25     mem.write(wr_a, io.din, write);
26
27     io.dout  = mem.read(rd_a);
28     io.empty = (wr_p == rd_p);
29     io.full  = (wr_a == rd_a) && (wr_p[A] != rd_p[A]);
30   }
31 };
```

Listing 1: Generic Fifo Description

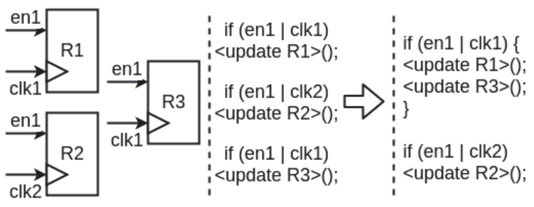
objects *ch_mem* (line 24), and multiplexers *ch_sel* (lines 21-22). There are also bit slicing operations (lines 15-16), and subscript operations (line 29), that we model in the Graph IR using *proxy* nodes.

IV. Tango Optimizations

A. Proxy-Coalescing Dataflow Optimization

A unique characteristic of hardware programs is the prevailing presence of bitwise operations involving slicing or concatenation that we capture via *proxy* nodes. Different types of redundant operations can exist across proxy nodes which can introduce unnecessary computations during simulation if the compiler is not able to resolve the aliasing effects. Figure 8 shows the three types of transformations executed during this stage: *A)* This transform eliminates identity proxies which are direct copies of their source node by simply deleting the node and connecting its source node to the destination nodes that were using the original proxy. *B)* This transform eliminates shadow proxies which are proxies that are used as source to other proxies and have a bit range that supersedes those destination proxies. In figure 8, proxy *P1* has proxy *P2* as source but *P2*'s bit range overlaps *P1*. We use the partial blue coloring on the *P1* and *P2* blocks to show the partial bit range they copy from their source node. There is no need to have *P2* in the middle since its bit range overlaps *P1*, simply eliminating it removes the unnecessary indirection. *C)* This transform coalesces adjacent proxies which are source proxies that do not overlap and when merged together can produce the full bit range of their destination node. In figure 8, proxy *P1* has two source nodes *P2* and *P3* that are also proxies, each encapsulating a partial range of node B. When put together, *P2* and *P3* is simply a direct copy of node B and therefore can both be eliminated.

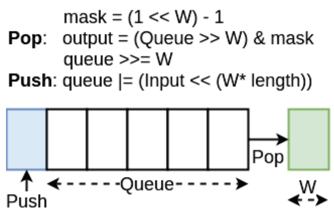
We found PCX to be particularly effective in programs where bits manipulation is prevalent and this is especially the case for most encryption and compression algorithms.



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

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

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

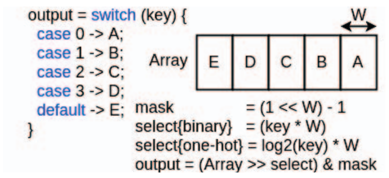
```
if (en1 | clk1) {
<update R1>();
<update R3>();
}

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

Fig. 5: Coalescing Register Update



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

Fig. 6: 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. 7: Switch Table Lowering

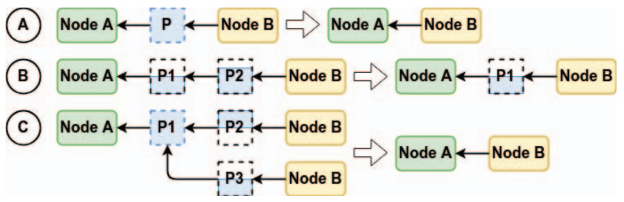


Fig. 8: Proxies Elimination

B. Sequential Node Coalescing

Sequential components in hardware such as registers, memory are modeled separately in the graph IR, each node capturing the behavioral properties of the component with their dependent signals, generally, the *clock* and the *reset* signals. These components can be individually lowered to native code, updating their state transition in their unique control flow block. However, we observed that many of these sequential objects share identical control signals that make it possible to group them together and perform their state update as a batch. SNC optimization traverses the graph IR sorted in topological order, groups sequential nodes based on their shared signals and determines the optimal scheduling of these nodes update that leverage the sharing of the control flow blocks.

Figure 5 illustrates a simple example of three register nodes that are grouped into two control flow blocks for update based on their matching input signals. This optimization leverage the runtime behavior of sequential nodes in the IR to schedule their update efficiently. This change can have a considerable impact on performance by not only eliminating unnecessary control flow but also reducing the overall binary blob size. This change affected pretty much all the models we evaluated in our analysis.

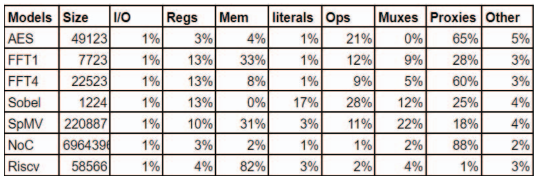
C. Shift-Register Optimization

Shift registers are prevalent hardware components used to delay signal values for the desired amount of clock cycles. They are generally implemented using a cascading chain of connected flip-flop registers. A shift register is not a native RTL primitive and has to be explicitly described using registers and connecting wires. The Tango IR explicitly captures shift-registers in its IR to enable Tango to exploit unique optimizations during the simulator code generation. A naive implementation of shift register involves copying data between the various stages of the shift registers pipeline in sequence every cycle. SRO optimization lowers small shift register using simple shift arithmetic. Figure 6 illustrates a typical shift register operation with a new input value pushed in the right of the queue and a new output value pop-ed from the queue every cycle. For short shift registers, the 'Pop' operation can be implemented by masking out the lower *W* bits of queue scalar where *W* is the data bit width. The 'Push' operation can be implemented by shifting the input value *W* x queue *length* bits to the right and Or-ing with the current queue scalar value. Using a 64-bit scalar for storing the shift register queue value was enough to capture the major subset of shift registers in our sample models. For instance, it is possible to store a 8-bit shift register of length 8 into a 64-bit scalar.

D. Switch Table Optimization

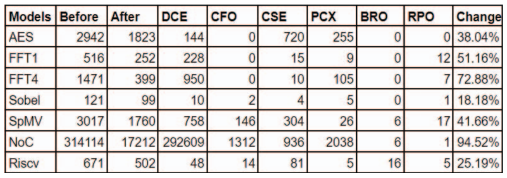
Switch statements in Tango are modeled as multiplexers. By analyzing various designs we observed that a considerable set of multiplexers consist of simple small lookup tables with constant values. SWO optimization attempts to lower these small switch tables using simple arithmetic operations, to avoid unnecessary branch computations. Traditional compiler switch optimizations [15] will not always be able to optimize such switch statement because the language C/C++ language datatypes only capture byte-wise element size. Since the graph IR also capture the bit-level size information for the switch statements, more elements can be packed into a switch table with small destination bit width (less than 8-bits) which were fairly common in our sample models. A typical switch node destination of 4-bit could be stored in 64-bit scalar supporting up to 16 values.

Figure 7 illustrates Tango's switch table lowering scheme. Similar to the shift register implementation, switch table lowering also involves native shift operations over a scalar constant variable containing an ordered array representation of the switch destination values. In the example, the case constant values A, B, C, D, and E of the provided switch are appended into a scalar array and are extracted based on the predicate key. We implemented two methods for computing the select key used to extract values out of the scalar array based on the type of multiplexer if it is a binary multiplexer or a one-hot multiplexer. For one-hot multiplexers, we compute the *log2* of the switch predicate key using the native *popcount* instruction on the target architecture. It is important to point out that SWO optimization, in particular, the support for one-hot multiplexers is a hardware-centric optimization that only



Models	Size	I/O	Regs	Mem	literals	Ops	Muxes	Proxies	Other
AES	49123	1%	3%	4%	1%	21%	0%	65%	5%
FFT1	7723	1%	13%	33%	1%	12%	9%	28%	3%
FFT4	22523	1%	13%	8%	1%	9%	5%	60%	3%
Sobel	1224	1%	13%	0%	17%	28%	12%	25%	4%
SpMV	220887	1%	10%	31%	3%	11%	22%	18%	4%
NoC	6964396	1%	3%	2%	1%	1%	2%	88%	2%
Riscv	58566	1%	4%	82%	3%	2%	4%	1%	3%

Fig. 9: Models Netlist Summary



Models	Before	After	DCE	CFO	CSE	PCX	BRO	RPO	Change
AES	2942	1823	144	0	720	255	0	0	38.04%
FFT1	516	252	228	0	15	9	0	12	51.16%
FFT4	1471	399	950	0	10	105	0	7	72.88%
Sobel	121	99	10	2	4	5	0	1	18.18%
SpMV	3017	1760	758	146	304	26	6	17	41.66%
NoC	314114	17212	292609	1312	936	2038	6	1	94.52%
Riscv	671	502	48	14	81	5	16	5	25.19%

Fig. 10: Dataflow Optimization Summary

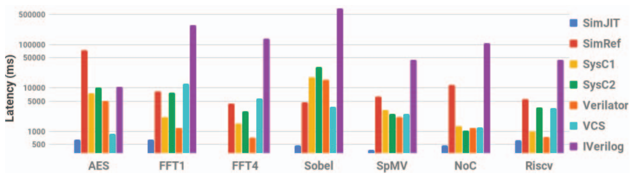


Fig. 11: Runtime Latency Comparison

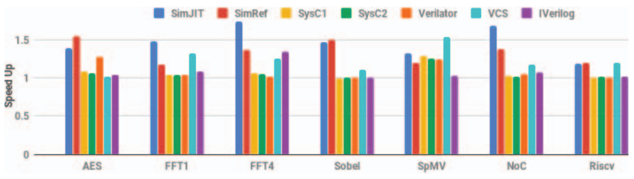


Fig. 12: Proxy Coalescing Speed Up Comparison

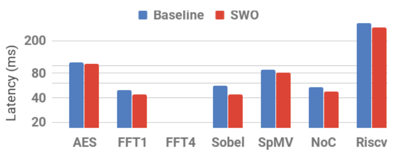


Fig. 13: Sequential Node Opt.

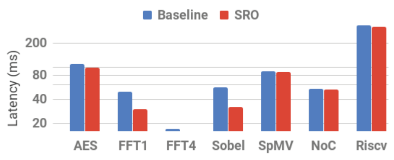


Fig. 14: Shift Register Opt.

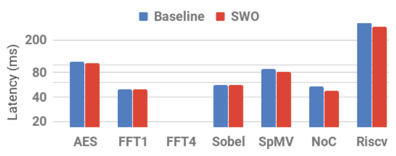


Fig. 15: Switch Table Opt.

makes sense in this application domain.

V. Evaluation

A. Benchmarks Description

For evaluation, we use various hardware blocks that we implemented as part of a test benchmark which include; (1) a 128-bit Advanced Encryption Standard Encryption engine (AES), (2) a radix 2^2 single-path delay feedback (SDF) FFT, (3) a 4-lanes parallel radix 2^2 multi-path delay feedback (MDF) FFT, (4) an 8-bit pipelined Sobel filter (Sobel), (4) a 16.16 fixed-point Sparse Matrix Multiplier (SpMV), (5) A 32-nodes Network-on-Chip router (NoC), (6) a 5-stage RiscV RV32I processor core (RiscV).

The table in figure 9 presents the breakdown Graph IR nodes distribution for each model. The first column, Size, contains the total size in term of total number bits of all the nodes in the IR graph for a given model. The following columns provide the average percentage in bits count for the associated node in the graph. The Registers column provides the total percentage of register bits relative to the graph total size. This information is a proxy for the complexity and area cost of the various models. The NoC is the largest model with 6964396 IR nodes, mainly dominated by 'proxy' nodes which are used to connect all the routers in the design. The AES model has a high concentration of ALU operations, about 21%, which come from the large amount bit shifting and XOR operations involved in its computation. The RiscV model has the highest concentration of memory object, about 82%, coming from its register file and caches. The SpMV model has the highest concentration of muxes, about 22%, which comes from the FSM blocks of its internal controllers.

B. Experimental Setup

We ran our benchmarks on a 32-cores 2.3 GHz Intel Xeon E5-4610 processor with 32KB L1, 256KB L2, and 16384KB L3 caches. The on-board total system memory was 64 GBytes. For comparison, we compared the Tango simulator with other RTL simulator implementations, including Icarus Verilog (IVerilog [9]), the industry-grade Synosys VCS [8], SystemC [7] single-threaded (SysC) and multi-threaded (SysC2) with 2 threads, and Verilator [13]. We used Verilator's SystemC generator to convert our Verilog models to SystemC.

C. Data-flow Optimization

The table in figure 10 presents a breakdown report of Tango data-flow optimization pipeline for our evaluated benchmark models. The 'Before' column shows the total number of nodes in the Graph IR before optimization. The 'After' column shows the final total nodes count after the optimization. The following columns 'DCE', 'CFO', 'CSE', 'PCX', provides the total number of deleted nodes affected by the associated optimization pass. In this analysis, we will ignore DCE from the evaluation since this a common artifact of code elaboration. AES model reduction is mainly affected by CSE and PCX optimizations, we observe a similar trend also for FFT1 and FFT4. CFO has major impact on the SpMV, NoC models which are the largest models in our suite.

D. End-to-end Performance

Figure 11 presents the average performance comparison between Tango and other RTL simulators as runtime latency in milliseconds. Tango JIT simulator (SimJIT) is on average 6.9x faster than Verilator, 7.8x faster than VCS and 225x faster than IVerilog across all models. Tango Reference simulator (SimRef) is on average 29x faster than IVerilog across all models. Tango is about 7x faster than Verilator for the AES computation, also 4x faster in SpMV and 3x faster on FFT computation. For the RiscV processor, Tango is only 1.2x faster, this is mainly because for RiscV graph IR did not benefit much from the unique Proxy dataflow optimization or other sequential nodes coalescing, shift register, switch table codegen optimizations that we will discuss in later sections. Except for SpMV and NoC models, SysC2's performance was worse than SysC, which is why we limited the number of threads to two.

E. Proxy Coalescing Optimization

Figure 12 presents the average speed-up of enabling proxy coalescing optimization (PCX) across all test models. In this evaluation, we compare the performance of the models when PCX optimization is disabled versus when it is enabled. We also use Tango's generated Verilog on other RTL simulators to also see by how much PCX improves their performance. We can observe that across the aboard the Tango simulator benefit from enabling PCX, A 1.5x speed up on average across all test models for SimJIT. Verilator's highest speed improvement is for the AES model, about 20% average speed up. VCS and IVerilog simulators also shows a similar speed improvement for FFT1 and FFT4 models.

F. Codegen Optimizations

Figure 13 presents the average performance improvement of Sequential Node Coalescing (SNC) codegen optimization for Tango. The graph compares the simulation latency when SNC is enabled alone versus a baseline where all three codegen optimizations SNC, SRO and SWO are disabled, only showing the performance contribution of the optimization in isolation to other optimizations. SNC optimization shows great impact for the Sobel filter, with about 43% average speed up. This is not surprising since we know that the Sobel graph IR contains the highest concentration of registers across all test models (see figure 9). AES and FFT1 also show considerable performance improvement, a 5% and 25% speed up, respectively.

Figure 14 presents the average performance improvement of Shift-Register (SRO) codegen optimization for Tango. The graph compares the simulation latency when SRO is enabled alone versus a baseline where all three codegen optimizations SNC, SRO and SWO are disabled, only showing the performance contribution of the optimization in isolation to other optimizations. SRO optimization shows great impact again for the Sobel filter, with about 75% average speed up. Once again, this is because of the high registers count existing in the model Graph IR. SRO optimization also has a great impact on FFT1, with about 66% average speedup. AES also shows considerable performance improvement with SRO, a 11% average speed up, respectively.

Figure 15 presents the average performance improvement of Switch Table (SWO) codegen optimization for Tango. The graph compares the simulation latency when SWO is enabled alone versus a baseline where all three codegen optimizations SNC, SRO and SWO are disabled, only showing the performance contribution of the optimization in isolation to other optimizations. SWO optimization shows considerable impact for the three largest models in our benchmark suite, SpMV, NoC, and RiscV, with an average speed up of 8%, 12%, 14%, respectively. We did identify in section A that SpMV had the highest count of multiplexer elements in its graph IR, accounting for the multitude of FSMs in its hardware description.

VI. Conclusion

In this paper, we presented a new compilation framework for just-in-time RTL simulation. We described the unique optimization techniques that Tango implements at both the dataflow level and the native codegen level to improve RTL simulation performance. These techniques leverage the unique attributes of hardware primitives and their semantic to optimize code generation, allowing Tango to out-perform other simulators. This work has also enabled us to identify the correlations between simulation speed and hardware description, giving us a better insight into the elements of the IR that directly impact runtime performance during simulation. Tango is being made available as an open-source project for public use.

References

[1] DARPA, "Intelligent design of electronic assets (idea)." https://www.darpa.mil/attachments/eri_design_proposers_day.pdf, 2017.

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

[3] A. Kupriyanov, D. Kissler, F. Hannig, and J. Teich, "Efficient event-driven simulation of parallel processor architectures," SCOPES '07.

[4] F. H. Alexey Kupriyanov and J. Teich, "Automatic and optimized generation of compiled high-speed rtl simulators," 2004.

[5] D. Lockhart, G. Zibrat, and C. Batten, "Pymtl: A unified framework for vertically integrated computer architecture research," MICRO-47.

[6] J. Bachrach, H. Vo, B. Richards, Y. Lee, and A. Waterman, "Chisel: Constructing hardware in a scala embedded language," DAC'12.

[7] P. R. Panda, "Systemc - a modeling platform supporting multiple design abstractions," in *System Synthesis, 2001.*

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

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

[10] L. Li and C. Tropper, "A design-driven partitioning algorithm for distributed verilog simulation," PADS'07.

[11] H. Qian and Y. Deng, "Accelerating rtl simulation with gpus," ICCAD '11.

[12] D. Kim, M. Ciesielski, and S. Yang, "A new distributed event-driven gate-level hdl simulation by accurate prediction," DATE'11.

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

[14] "Ieee standard for verilog hardware description language," *IEEE Std 1364-2005 (Revision of IEEE Std 1364-2001)*, pp. 0_1–560, 2006.

[15] A. Korobeynikov, "Improving switch lowering for the llvm compiler system," 2007.