TaroRTL: Accelerating RTL Simulation Using Coroutine-Based Heterogeneous Task Graph Scheduling

Dian-Lun Lin(✉), Umit Ogras, Joshua San Miguel, and Tsung-Wei Huang

University of Wisconsin-Madison, Madison, WI, USA
dianlun.lin@wisc.edu

Abstract. RTL simulation is critical for validating hardware designs. However, RTL simulation can be time-consuming for large designs. Existing RTL simulators have leveraged task graph parallelism to accelerate simulation on a CPU- and/or GPU-parallel architecture. Despite the improved performance, they all assume atomic execution per task and do not anticipate multitasking that can bring significant performance advantages. As a result, we introduce TaroRTL, a coroutine-based task graph scheduler for efficient RTL simulation. TaroRTL enables non-blocking GPU and I/O tasks within a task graph, ensuring that threads are not blocked waiting for GPU or I/O tasks to finish. It also designs a coroutine-aware work-stealing algorithm to avoid unnecessary context switches. Compared to a state-of-the-art GPU-accelerated RTL simulator, TaroRTL can further achieve 40–80% speed-up while using fewer CPU resources to simulate large industrial designs.

Keywords: RTL simulation · Heterogeneous task graph · Scheduling

1 Introduction

The time-consuming nature of Register-transfer level (RTL) simulation poses a significant challenge for verifying today's highly complex SoCs, processors, and accelerators [11,13,16]. As SoC complexity continues to grow, achieving industry-quality functional verification signoff typically demands a significant and growing amount of simulation tests on the same Design-Under-Test (DUT) with different input stimuli, all in preparation for tapeout. For a comprehensive analysis of the design's behavior, SoC designers even require a Value-Change-Dump (VCD) file, resulting in substantial long runtime associated with input and output (I/O) operations to capture and process traces [2]. Speeding up RTL simulation is crucial for coping with the rapidly increasing design complexity and the shorter time-to-market demands.

State-of-the-art RTL simulators have leveraged *task graph parallelism* to accelerate simulation on a CPU- and/or GPU-parallel architecture [11–13]. This task graph consists of various tasks performed per simulation cycle, such as evaluating logic elements, setting inputs, or I/O VCD dump. Through task graph scheduling, multiple tasks can be scheduled and executed concurrently once the dependency constraints are met. While all these approaches have shown runtime or throughput improvements, the performance is far from optimal. Specifically, existing task graph scheduling solutions for RTL simulation all assume *atomic* execution per task (i.e., a thread runs or blocks until its assigned task is complete) and do not anticipate multitasking that can reduce CPU waiting time on awaiting GPU and I/O tasks to finish. For instance, Fig. 1 shows an RTL task graph with different task types [11]. Without multitasking as shown in (b), when a CPU thread invokes a GPU task (task B) or an I/O task (task C), it will block until the task finishes.

Recently, the new C++20/23 standard has introduced Coroutine [1]. Coroutine offers a new mechanism for programming multitasking by allowing suspension and resumption of a function from its running thread. This mechanism has inspired us to design a new coroutine-based task graph scheduling solution with significantly improved performance. As shown in Fig. 1(c), after invoking task B and task C, the CPU thread suspends those tasks and multitask to task D without being blocked. Compared to (b), using Coroutine enables better utilization of computing resources and reduces the total runtime.

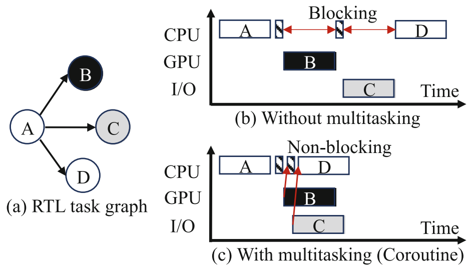

Fig. 1. Performance comparison with and without multitasking using one CPU and one GPU. The patterned rectangle represents the kernel call overhead (GPU and I/O).

However, designing a coroutine-based task graph scheduler is very challenging for three reasons. First, coroutines present a different execution mechanism compared to traditional function calls, primarily due to their ability to suspend and resume execution at certain points rather than executing to completion like traditional functions. This difference poses challenges for existing task graph schedulers [6,9–13], as they are typically designed with the assumption of traditional function calls and lack support for coroutine-specific features. Second, Coroutine's suspension and resumption ability requires a specially designed scheduling algorithm to minimize the cost of context switches. Furthermore, a coroutine does not automatically resume after suspension; instead, it requires a scheduler to track and control its execution. Managing and tracking the execution status of each coroutine becomes complicated when dealing with complex workloads.

To overcome these challenges, we introduce TaroRTL, an efficient coroutine-based task graph scheduler for RTL simulation. We summarize three key contributions as follows:

– We design a coroutine-based task graph scheduling model to enable non-blocking GPU and I/O tasks within a task graph.

– We design a coroutine-aware work-stealing algorithm to avoid unnecessary context switches and cache misses.
– We design an execution control strategy to track and control the execution of each invoked GPU and I/O task.

We have evaluated TaroRTL on industrial designs and demonstrated its promising performance compared to the state-of-the-art RTLflow (CPU- and GPU-based) [11] and Verilator (CPU-based) [12]. As an example, TaroRTL can speed up RTLflow by 40–80% while using fewer CPU resources to simulate large industrial designs. We will make TaroRTL open-source to benefit RTL simulation research.

2 The Motivation of Using Coroutine in RTL Simulation

Existing task graph scheduling solutions for RTL simulation all assume atomic execution, resulting in significant CPU waiting time on awaiting GPU and I/O tasks to finish. This problem prevents us from fully unleashing the power of heterogeneous simulation. Figure 2 gives an example of CPU waiting and active time growth over increasing input stimuli in RTLflow [11] (CPU- and GPU-based). The CPU waiting time inevitably reduces the overall efficiency of an RTL simulator, especially when simulating a design with numerous input stimuli. The increasing CPU waiting time over increasing number of input stimuli indicates untapped performance potential within the RTL simulation. Furthermore, CPUs need to keep spinning until GPU completes its task, wasting a lot of unnecessary CPU resources.

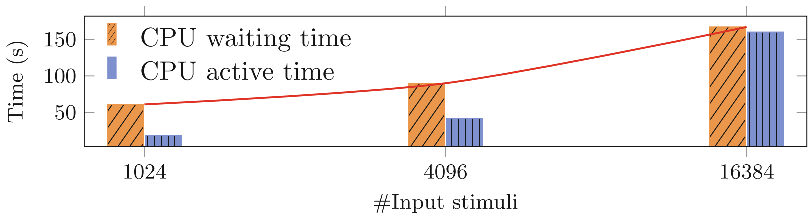

Fig. 2. CPU waiting and active time growth over increasing numbers of input stimuli in RTLflow [11]. The relative ratio of waiting time gets smaller as the number of input stimuli increases because a large number of input stimuli induce a significant amount of CPU computation for setting inputs.

Unlike a traditional function that runs to completion and returns a value, a coroutine can be suspended and resumed at specific points without losing its state. Specifically, modern C++ Coroutine allows a CPU thread to suspend its current task and resume other tasks (i.e., multitasking) while awaiting GPU or I/O operations to finish. This property makes coroutines particularly useful for parallel RTL simulation. Listing 1.1 gives a CPU-GPU simulation example for simulating a design with two input stimuli using Coroutine. The code simulate an input design *dut* cycle by cycle with a scheduler *s*. At each cycle iteration, we first set the inputs of *dut* using the given input stimulus (Line 5). When a CPU thread offloads `eval` to GPU (Line 7 and 9), it can multitask to another input stimulus for `set_inputs`. Without Coroutine, existing RTL simulators such as RTLflow [11] require a CPU thread to wait for `eval` on GPU to finish.

```
 1  void sim(Stimulus& stim) {
 2    Design dut;
 3    size_t c{0};
 4    while(!dut.stop and c <= NUM_CYCLES) {
 5      dut.set_inputs(stim, c);
 6      dut.set_clock(0);
 7      co_await dut.eval();    // offload to GPU and multitask
 8      dut.set_clock(1);
 9      co_await dut.eval();    // offload to GPU and multitask
10      c += 1;
11    }
12  }
13  int main() {
14    Scheduler s;
15    Stimuli stim = get_stimuli(); // get input stimuli
16    s.emplace(sim, stim[0]); // emplace a sim task for stim 0
17    s.emplace(sim, stim[1]); // emplace a sim task for stim 1
18    s.schedule();            // schedule the two sim tasks
19    return 0;
20  }
```

Listing 1.1. An example of RTL simulation using Coroutine. When co_await, a CPU thread multitasks to another input stimulus. The scheduler needs to track and control the execution of each invoked task.

3 TaroRTL

At a high level, TaroRTL enables multitasking within a task graph through Coroutine. We allow CPU threads to multitask without being blocked by GPU or I/O tasks. We introduce a coroutine-aware work stealing to minimize context switches. Our execution control strategy effectively tracks and controls the execution of each GPU and I/O task.

3.1 Overview

Figure 3 shows an example of TaroRTL scheduling a task graph using two CPU threads (workers), one GPU stream, and one I/O buffer. Each worker maintains a high-priority queue (HPQ) and a low-priority queue (LPQ). HPQ stores suspended tasks that have been lately executed by the worker, while LPQ stores new tasks that have met task dependency constraints. During scheduling, each worker extracts tasks from its HPQ to LPQ, ensuring that a suspended task is prioritized over a new task. Such prioritization allows efficient caching by ensuring that suspended tasks, which have been recently executed, take precedence over new tasks in the scheduling process. We leverage work-stealing queues [7] to support our scheduling architecture. Only the queue owner [7] can pop/push a task from/into one end of the queue, while multiple workers can steal a task from the other end at the same time. When both of a worker's queues are empty, that worker tries to steal a task from another worker's LPQ to HPQ. This strategy not only balances the workload among the workers, but also reduces the chances of different threads stealing (i.e., resuming) the suspended task. As shown in Fig. 3, the algorithm follows these steps:

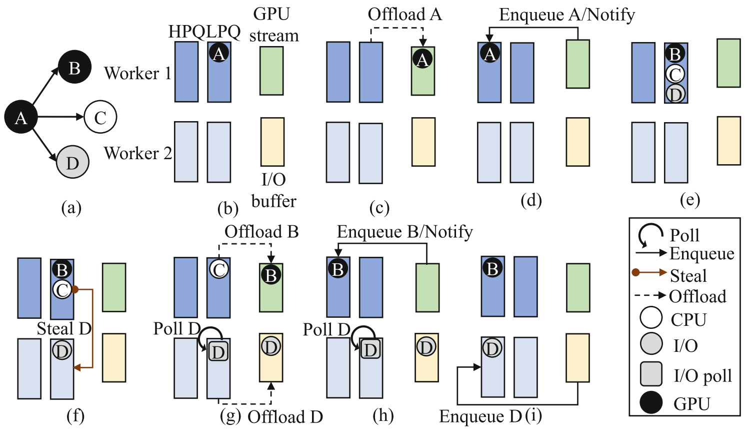

Fig. 3. TaroRTL schedules a task graph using two CPU workers, one GPU stream, and one I/O buffer. Each worker owns a high-priority task queue (HPQ) and a low-priority task queue (LPQ) to prioritize resuming a suspended task over a new task.

(b) **Enqueue A**: TaroRTL enqueues A into Worker 1's LPQ.
(c) **Offload A and register a CUDA callback**: Worker 1 executes and offloads A into the CUDA stream. Worker 1 then registers a CUDA callback for A.
(d) **Invoke the CUDA callback for A**: After A finishes, CUDA runtime invokes the callback that enqueues A back into Worker 1's HPQ and notifies Worker 1. This strategy ensures Worker 1 resumes A rather than Worker 2, avoiding unnecessary context switches.
(e) **Enqueue B, C, and D**: Worker 1 resumes A for cleanup and resolving task dependencies, and enqueues B, C, and D into its LPQ.
(f) **Steal task from Worker 1**: Worker 2's queues are empty. Worker 2 steals D from Worker 1's LPQ. Since D has not yet started, this steal does not induce context switches.
(g) **Offload B and D**: Worker 1 and 2 offload B and D into CUDA stream and I/O buffer, respectively. Worker 1 registers a CUDA callback for B.

(g) **Poll the status of D**: Worker 2 creates a query task and enqueues the task into its LPQ to poll the execution status of D.
(g) **Multitask to C**: Worker 1 multitasks to C.
(h) **Invoke the CUDA callback for B**: After B finishes, CUDA runtime invokes the callback that enqueues B back into Worker 1's HPQ and notifies Worker 1.
(i) **Resume D**: Worker 2 resumes D for cleanup after verifying that D has finished.
(i) **Continue until complete**: The scheduling process continues until each worker completes its assigned task.

3.2 Coroutine-Aware Work Stealing

Conventional work-stealing algorithms cannot be used out of the box due to the distinct performance characteristics between atomic and suspendable executions. For instance, when a suspended task is ready, conventional work-stealing algorithms notify any available CPU threads to resume that task. This strategy may resume a task using different CPU threads, resulting in frequent context switches and cache misses.

Recently, C++20 released a new synchronization primitive of *atomic wait and notify*, which allows a thread to wait on an atomic variable until other threads change its value and notify that thread. This new feature has inspired us to tackle this challenge by assigning each worker an atomic variable to communicate with a specific worker while tracking each worker's state. This approach aligns with the goal of reducing context switches and cache misses by ensuring that a task is mostly resumed by the same worker. Furthermore, the new atomic features have shown improved performance compared to condition variables, which are commonly used by existing schedulers for synchronization purposes.

Algorithm 1 presents the pseudocode of our work-stealing algorithm for each worker. Each worker has an atomic variable, *state*, with three possible states: BUSY, SIGNALED, and SLEEP. BUSY indicates a worker is actively processing tasks. SIGNALED signifies a worker has been notified by other workers. SLEEP represents a worker who is inactive and waiting for other workers to notify. Initially, each worker waits on the SLEEP (line 2), ensuring its inactivity until scheduled by the scheduler. When the schedule function is called, the scheduler evenly distributes the source tasks to each worker's LPQ. It then changes each worker's state to SIGNALED and notifies them, indicating they are ready to execute tasks.

Once a worker wakes up, it changes its state to BUSY and starts popping tasks from its own HPQ to LPQ (lines 4–9). The worker first attempts to pop a task from its HPQ. Since CUDA runtime or other workers can simultaneously enqueue suspended tasks into HPQ (e.g., Fig. 3(d) and (h)), we use `steal` to extract a task from the HPQ that avoids data races. If the HPQ is empty, the worker proceeds to pop a task from its LPQ. The LPQ is managed by the worker, and enqueuing/popping a LPQ by other workers requires a lock [7]. In the event that both of the worker's queues are empty, the worker randomly selects another worker and steals a task from its LPQ to HPQ (lines 10–22). The iteration continues until we successfully steal a task or fail to steal a task after MAX_STEAL times. To keep track of the overall progress, we maintain an atomic variable, *pending_tasks*, that represents the total number of tasks ready to be invoked. If no tasks are available at a given point, resulting in *pending_tasks* becoming zero, the worker changes its state to SLEEP and checks if its original state has been changed by another worker (lines 35–37). If the worker's state has changed, indicating at least one other worker has changed the state to SIGNALED, the worker continues to work. Otherwise, the worker waits until other workers change its state and notify it.

After a worker invokes a task, there are two situations: 1) The task is complete (Fig. 3(e)). 2) The task is suspended (Fig. 3(g)). If the task is complete, the worker checks the task's successors and enqueues them into its LPQ if the dependency constraints are met (lines 26–32). On the other hand, if the task is suspended, indicating the worker has invoked GPU or I/O tasks, the worker continues without blocking. Once invoked GPU or I/O tasks are complete, CUDA runtime or a worker will enqueue the suspended task back into the worker's HPQ and notify the worker.

Algorithm 2 outlines how CUDA runtime or a worker enqueues a task and notifies the worker. If *worker* is NULL, it means we want to enqueue a new successor task. The current worker enqueues the task into its LPQ and notifies one available worker (lines 1–14). This strategy allows an available worker to steal a new task from the worker to avoid under-utilization. We iterate through each worker and check if its state is SLEEP using the Compare and Swap (CAS) operation. If a worker's state is SLEEP and its state is successfully changed to SIGNALED (i.e., the CAS operation returns true), it is notified. Otherwise, we iterate to the next worker. The process continues until either a worker is successfully notified or all workers have been checked. If no worker is in SLEEP, we exit the loop without notifying any worker. If *worker* is not NULL, it means we want to enqueue the task into a specific worker's HPQ and notify that worker (lines 15–21). We require a lock since HPQ may be simultaneously enqueued by other workers. After the task is enqueued, we update the worker's state to SIGNALED, indicating this task is ready to resume. If the original state of the worker was SLEEP, it implies that the worker is inactive and waiting to be notified. We notify the worker using its state. However, if the worker's state was not SLEEP, implying that the worker is already active, we skip the notification. This organization minimizes unnecessary notification overhead and helps improve overall performance.

3.3 Execution Control Strategy

After a GPU or an I/O task finishes, we need to resume the task for cleanup, such as freeing up GPU or I/O resources, releasing memory, and resolving task dependencies. While Coroutine allows for the suspension of a task, it does not automatically resume a suspended task. We need an execution control strategy to track the execution status of a suspended task and trigger its resumption.

Algorithm 1: Coroutine-aware work-stealing algorithm

```
worker ← this_worker();/* current worker */
worker.state.wait(SLEEP);
do
    worker.state.store(BUSY);
    do
        /* get from worker's own HPQ to LPQ */
        task ← worker.HPQ.steal();
        if task == NULL then
            task ← worker.LPQ.pop();
        end
        /* steal from another worker's LPQ to HPQ */
        if task == NULL then
            cnt ← 0;
            while cnt++ < MAX_STEAL do
                aworker ← random_select();
                task ← aworker.LPQ.steal();
                if task == NULL then
                    task ← aworker.HPQ.steal();
                end
                if task != NULL then
                    break;
                end
            end
        end
        /* invoke the task and enqueue successors */
        if task != NULL then
            pending_tasks.fetch_sub();
            invoke(task);
            if task.is_done() then
                for succ : task.successors do
                    if succ.dependency.is_met() then
                        enqueue_notify(succ, NULL);
                    end
                end
            end
        end
    while pending_tasks.load() > 0;
    if worker.state.exchange(SLEEP) == BUSY then
        /* wait to be notified */
        worker.state.wait(SLEEP);
    end
while !stop;
```

Non-blocking I/O Tasks. We leverage *io_uring* [3], a new asynchronous I/O framework in Linux that provides efficient and scalable support for asynchronous

Algorithm 2: enqueue_notify(task, worker)

```
   Input: $task$: a suspended task to be enqueued
   Input: $worker$: a worker to be notified
1  if $worker$ == NULL then
       /* enqueue to current worker's own LPQ */
2      $worker \leftarrow$ this_worker();
3      $worker.LPQ$.push($task$);
4      $pending\_tasks$.fetch_add();
       /* notify one SLEEP worker */
5      $cnt \leftarrow 1$;
6      do
7          $idx \leftarrow (worker.idx + cnt)\%$NUM_WORKERS;
8          $tmp \leftarrow$ SLEEP;
9          if $workers[idx].state$.CAS($tmp$, SIGNALED) then
10             $workers[idx].state$.notify_one();
11             return;
12         end
13     while $++cnt <$ NUM_WORKERS;
14 end
15 else
       /* enqueue and notify a specific worker */
16     lock{$worker.HPQ$.push($task$)};
17     $pending\_tasks$.fetch_add();
18     if $worker.state$.exchange(SIGNALED) == SLEEP then
19         $worker.state$.notify_one();
20     end
21 end
```

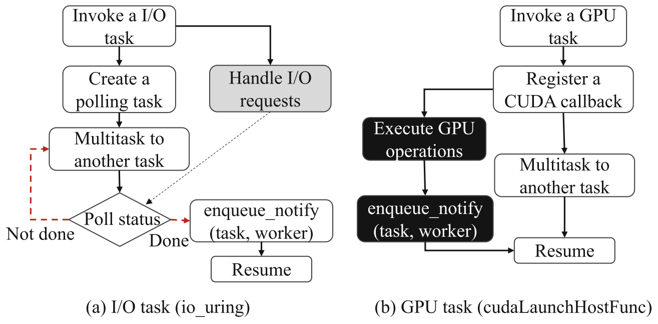

Fig. 4. A flowchart of our execution control strategy for (a) I/O and (b) GPU tasks. Gray (Black) blocks represent actions performed by io_uring (CUDA runtime).

I/O operations. io_uring implements a ring buffer structure to manage I/O requests. This ring buffer allows for the efficient submission and retrieval of I/O requests without the need for blocking system calls or copies. By incorporating

C++ coroutine with io_uring, we are able to submit non-blocking I/O tasks to the ring buffer and seamlessly multitask to a different task.

Figure 4(a) illustrates our strategy for an I/O task. After invoking and suspending an I/O task, the worker creates a polling task and multitasks to another task. The polling task is also a coroutine and can be stolen by other workers to repeatedly check the I/O status. It is a lightweight task that incurs little CPU migration overhead when stolen. Once the status becomes done, the worker that executes this polling task enqueues the suspended task back into the invoked worker's HPQ and notifies that worker.

Non-blocking GPU Tasks. Figure 4(b) illustrates our strategy for a GPU task. To probe the execution status of an offloaded GPU task, we utilize CUDA's API, `cudaLaunchHostFunc`, which allows us to register a callback for the offloaded GPU task. The worker then multitasks to other tasks without being blocked. Once the offloaded GPU task is complete, CUDA runtime invokes the callback to enqueue the suspended task back into the worker's HPQ and notify the worker. CUDA callback has a certain cost. In cases where the cost of a GPU task is negligible (e.g., simulate a small design) or CUDA callback is not applicable, we utilize a CUDA event to record the execution status of an offloaded GPU task and poll the status, similar to non-blocking I/O tasks.

3.4 Performance Improvement Analysis

In this section, we analyze the time complexity of TaroRTL. As different designs have different task graph structures and task runtimes, it is not practical to analyze the time complexity in a universal manner. Instead, we focus on a more constrained scenario where TaroRTL can achieve the best efficiency over non-coroutine-based approaches. Assuming on a CPU-GPU simulation workload at timeframe P:

- There are n_c CPU threads and n_g GPU streams available.
- There are N identical tasks ready to be executed, where N is larger than n_c and n_g.
- Each task consists of a CPU subtask s_c with a cost of t_c, followed by a GPU subtask s_g with a cost of t_g.

By these assumptions, we can compute a lower bound on the time difference between TaroRTL ($T_{TaroRTL}$) and a scheduler without multitasking (T) at a specific timeframe. In the beginning, all n_c CPU threads simultaneously execute the CPU subtask s_c within n_c tasks, incurring a cost of t_c. Subsequently, all n_g GPU streams execute the GPU subtask s_g within these n_c tasks, resulting in $t_g \cdot \lceil n_c/n_g \rceil$. In TaroRTL, CPUs can multitask to the next n_c CPU subtasks while simultaneously waiting for the GPU to complete the current n_c GPU subtasks. As shown in Fig. 5, the overlapped time of s_c and s_g is $\min\{t_c, t_g \cdot \lceil n_c/n_g \rceil\}$ per n_c tasks. Specifically, TaroRTL saves at least $\min\{t_c, t_g \cdot \lceil n_c/n_g \rceil\}$ per n_c tasks. With $\lceil N/n_c \rceil - 1$ times,

$$T - T_{TaroRTL} \geq (\left\lceil \frac{N}{n_c} \right\rceil - 1) \cdot \min\{t_c, t_g \cdot \left\lceil \frac{n_c}{n_g} \right\rceil\}.$$

The time difference expands as the number of tasks increases or the CPU and GPU subtasks' cost becomes larger.

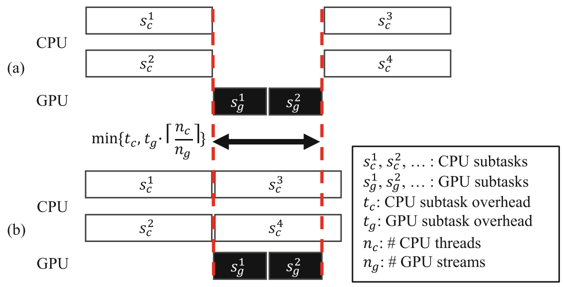
(a)
(b)
CPU
GPU
s_c^1
s_c^2
s_c^3
s_c^4
s_g^1
s_g^2
$\min\{t_c, t_g \cdot \lceil \frac{n_c}{n_g} \rceil\}$
$s_c^1, s_c^2, \ldots$: CPU subtasks
$s_g^1, s_g^2, \ldots$: GPU subtasks
t_c: CPU subtask overhead
t_g: GPU subtask overhead
n_c: # CPU threads
n_g: # GPU streams

Fig. 5. The time difference between (a) a scheduler without coroutine and (b) TaroRTL at a specific timeframe. In this example, n_c is 2, n_g is 1, and $t_c > t_g \cdot \left\lceil \frac{n_c}{n_g} \right\rceil$.

4 Experimental Results

We evaluate the performance of TaroRTL on three industrial designs: Spinal, riscv-mini, and NVDLA. We conducted our experiments on a 3.2 GHz 64-bit Linux machine with one NVIDIA RTX 3080 ti GPU and ten Intel i9-12900KF CPU cores. We compiled our programs with CUDA NVCC 12.1 on a GCC 12.1 host compiler and enabled optimization flag `-O2` and `-std=c++20`. We use an equal number of CUDA streams and CPU threads. All data is an average of ten runs. TaroRTL adopts heap allocation by default to store the stack of a C++ coroutine.

4.1 CPU-GPU RTL Simulation

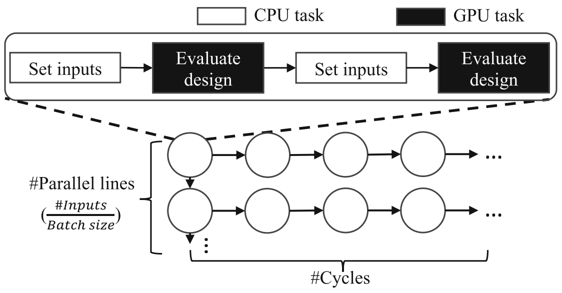

Fig. 6. The heterogeneous RTL task graph in RTLflow. Each task contains two CPU and GPU subtasks.

In this section, we analyze the performance benefits of TaroRTL with non-blocking GPU tasks. We consider RTLflow as our baseline. RTLflow [11] improves the throughput performance by running multiple input stimuli simultaneously using both CPU and GPU parallelisms. As shown in Fig. 6, RTLflow describes the RTL simulation workload as a heterogeneous task graph, where each task consists of four dependent CPU/GPU subtasks. The size of the task graph is determined by the number of inputs, the number of simulation cycles, and the chosen batch size. RTLflow splits inputs into batches to allow more parallelism (i.e., more parallel lines). For our experiments, we fixed the default batch size of 1024. However, RTLflow lacks support for multitasking, resulting in a CPU thread waiting for GPU tasks to finish. By scheduling RTLflow's heterogeneous task graph using TaroRTL, we are able to improve total runtime while using fewer CPU resources.

Table 1. Comparison between RTLflow and TaroRTL on Spinal, riscv-mini, and NVDLA designs using different numbers of threads for completing 32768 input stimuli. ELOC and SLOC represent lines of code for evaluation and lines of code for setting inputs, respectively. Bold texts represent the best results. All simulation results match the golden reference provided by RTLflow.

Design	ELOC	SLOC	#Threads	RTLflow	TaroRTL	Speed-up
Spinal	9654	6	2	36 s	**35 s**	2.9%
			4	**20 s**	21 s	-
			6	**16 s**	17 s	-
riscv-mini	10935	340	2	79 s	**44 s**	79.5%
			4	61 s	**34 s**	79.4%
			6	55 s	**35 s**	57.1%
			8	49 s	**35 s**	40.0%
			10	50 s	**36 s**	38.9%
NVDLA	560412	860	2	1082 s	**598 s**	80.9%
			4	600 s	**337 s**	78.0%
			6	482 s	**284 s**	69.7%
			8	376 s	**242 s**	55.4%
			10	379 s	**236 s**	60.6%

Table 1 compares overall runtime between RTLflow and TaroRTL on riscv-mini and NVDLA designs using different numbers of threads. TaroRTL outperforms RTLflow in almost all scenarios. TaroRTL achieves at least 1.4× speed-up and 1.6× speed-up on riscv-mini and NVDLA designs. We can clearly see the proposed coroutine-based task graph scheduling brings significant performance benefits to the CPU-GPU RTL simulation workload. Compared to RTLflow, TaroRTL achieves 1.8× speed-up on NVDLA. In the case of the Spinal design, RTLflow and TaroRTL exhibit similar runtimes, and the advantage of coroutines is less pronounced compared to other designs. When SLOC (lines of setting inputs) is small, where the CPU overhead is extremely small, TaroRTL does not benefit much from multitasking. However, modern designs typically have many thousands of ELOC and hundreds of SLOC [16] where TaroRTL can stand out.

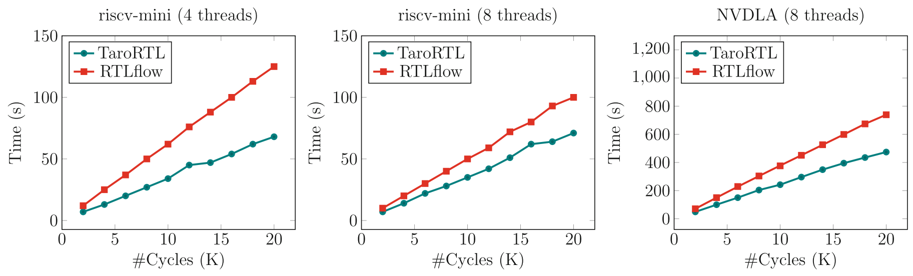

Fig. 7. Runtime growth over increasing numbers of cycles for TaroRTL and RTLflow using four and eight threads.

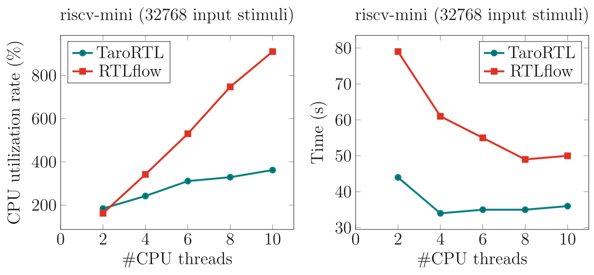

Fig. 8. Average CPU utilization rate reported by `/usr/bin/time` and runtime decrease over increasing numbers of CPU threads for TaroRTL and RTLflow on the riscv-mini design.

Figure 7 shows the runtime growth over increasing numbers of cycles for TaroRTL and RTLflow on different designs using four and eight threads. TaroRTL outperforms RTLflow in all scenarios. Compared to RTLflow, TaroRTL using four threads achieves 1.8× speed-up for riscv-mini design at 20K cycles. The significant improvement on runtime demonstrates the promise of our multitasking techniques. We can also clearly see the results are aligned with the speedup analysis. For example, the performance gap between TaroRTL and RTLflow continues to enlarge as we increase the number of cycles (i.e., the number of tasks N). Figure 8 shows the CPU utilization rate and runtime decrease over increasing numbers of threads on the riscv-mini design. TaroRTL using 362% CPU achieves 1.4× faster than RTLflow using 910% CPU. riscv-mini is a midsize design that does not induce large CPU computation. However, RTLflow requires CPUs to keep spinning until GPU finishes its operations, resulting in an unnecessarily high CPU utilization rate.

4.2 RTL Simulation with I/O

In this section, we study the performance benefits of RTL simulation with non-blocking I/O tasks. We consider Verilator, which supports VCD file dumping, as our baseline. Verilator is a single-stimulus simulator. For multi-stimulus simulation, the de facto way is to create multiple instances of Verilator and run independent input stimulus in parallel [11]. After an evaluation of the design, each Verilator stores traces in a buffer and dumps the buffer to a file once it is full. Since Verilator does not support multitasking, it requires a CPU thread to wait until I/O dumping finishes. By enabling our non-blocking I/O using TaroRTL, we are able to improve the simulation efficiency.

Figure 9 illustrates the achieved speed-up by TaroRTL over Verilator at different numbers of input stimuli using eight threads on riscv-mini. When the number of input stimuli equals 8, Verilator is faster due to the limited parallelism available for eight threads. However, as the number of input stimuli exceeds 8, where parallelism becomes more abundant (i.e., more independent tasks), TaroRTL starts to outperform Verilator. Since RTL simulation typically involves many input stimuli on the same design [11,16], TaroRTL's ability to handle larger parallelism provides a significant advantage.

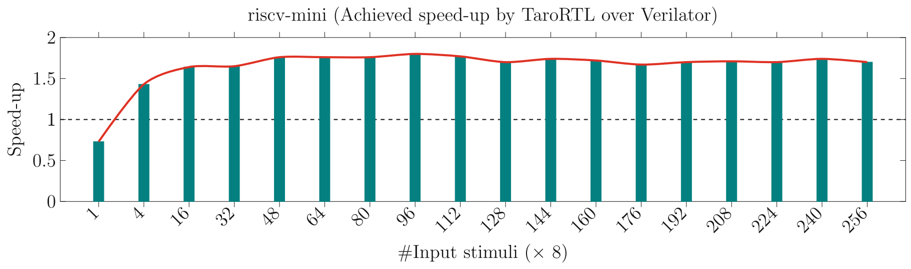

Fig. 9. Achieved speed-up by TaroRTL over Verilator at different numbers of input stimuli using eight threads for 3K cycles.

5 Related Work

Verilator [12], an open-source RTL simulator, aggregates adjacent logic elements into macro tasks, which are then scheduled using a static multi-threaded algorithm, achieving optimal performance with 8-10 CPU cores. RepCut [13] improves upon this by partitioning circuits into balanced segments with minimal overlap, reducing synchronization overhead and achieving superlinear speed-ups through task replication. However, it is restricted to strong scaling for single input stimuli. RTLflow [11] further innovates by executing multiple independent input stimuli in parallel on a GPU, utilizing a heterogeneous task set and a work-stealing scheduling algorithm. Despite its advancements, RTLflow encounters significant CPU idle time, as CPU threads must await the completion of GPU tasks within each simulation cycle. Taskflow [6] is a task graph scheduling system that has been adopted by many EDA algorithms [4,5,8,15], including RTLflow. However, Taskflow does not support multitasking in a task graph. Libfork [14] is a coroutine-tasking library that is primarily an abstraction for fully-portable, strict, fork-join parallelism. However, it does not support heterogeneous parallelism.

6 Conclusion

In this paper, we have introduced TaroRTL, a coroutine-based task graph scheduler for RTL simulation. TaroRTL has introduced a coroutine-based task graph scheduling model to enable multitasking in a task graph. TaroRTL has also introduced a coroutine-aware work-stealing algorithm to reduce unnecessary context switches. Compared to RTLflow, TaroRTL can further achieve 40–80% speed-up while using fewer CPU resources to simulate large industrial designs. Future work includes generalizing TaroRTL into a generic library with a new programming model that can be easily integrated with various applications.

Acknowledgment. This project is supported by NSF grants 2235276, 2349144, 2349143, 2349582, and 2349141.

References

1. C++ Coroutine (2020). https://en.cppreference.com/w/cpp/language/coroutines
2. Appello, D., et al.: Accelerated analysis of simulation dumps through parallelization on multicore architectures. In: DDECS (2021)
3. Axboe, J.: Efficient io with io_uring (2019)
4. Chang, C., Huang, T.W., Lin, D.L., Lin, S., Guo, G.: Ink: Efficient k-critical path generation. In: IEEE/ACM DAC (2024)
5. Huang, T.W., Guo, G., Lin, C.X., Wong, M.D.F.: Opentimer v2: a new parallel incremental timing analysis engine. IEEE TCAD (2021)
6. Huang, T.W., Lin, D.L., Lin, C.X., Lin, Y.: Taskflow: a lightweight parallel and heterogeneous task graph computing system. In: IEEE TPDS (2022)
7. Lê, N.M., Pop, A., Cohen, A., Zappa Nardelli, F.: Correct and efficient work-stealing for weak memory models. In: PPoPP (2013)
8. Lee, W.L., et al.: G-kway: multilevel GPU-Accelerated k-way Graph Partitioner. In: IEEE/ACM DAC (2024)
9. Lin, D.L., Huang, T.W.: Efficient GPU computation using task graph parallelism. In: Euro-Par (2021)
10. Lin, D.L., Huang, T.W.: Accelerating large sparse neural network inference using GPU task graph parallelism. IEEE TPDS **33**(11), 3041–3052 (2022)
11. Lin, D.L., Ren, H., Zhang, Y., Khailany, B., Huang, T.W.: From RTL to CUDA: a GPU acceleration flow for RTL simulation with batch stimulus. In: ICPP (2023)
12. Snyder, W.: Verilator 4.0: open simulation goes multithreaded (2018)

13. Wang, H., Beamer, S.: Repcut: superlinear parallel RTL simulation with replication-aided partitioning. In: ASPLOS (2023)
14. Williams, C.J., Elliott, J.: Libfork: portable continuation-stealing with stackless coroutines. arXiv preprint arXiv:2402.18480 (2024)
15. Zhang, B., et al.: G-PASTA: GPU accelerated partitioning algorithm for static timing analysis. In: IEEE/ACM DAC (2024)
16. Zhang, Y., Ren, H., Khailany, B.: Opportunities for RTL and gate level simulation using GPUs. In: ICCAD (2020)