Late Breaking Results: Efficient Timing Propagation with Simultaneous Structural and Pipeline Parallelisms

Cheng-Hsiang Chiu and Tsung-Wei Huang
Department of ECE, University of Utah, USA
{cheng-hsiang.chiu;tsung-wei.huang}@utah.edu

ABSTRACT

Graph-based timing propagation (GBP) is an essential component for all static timing analysis (STA) algorithms. To speed up GBP, the state-of-the-art timer leverages the task graph model to explore structural parallelism in an STA graph. However, many designs exhibit linear segments that cause the parallelism to serialize, degrading the performance significantly. To overcome this problem, we introduce an efficient GBP framework by exploring both structural and pipeline parallelisms in an STA task graph. Our framework identifies linear segments and parallelizes their propagation tasks using pipeline in an STA task graph. We have shown up to 25% performance improvement over the state-of-the-art task graph-based timer.

1 INTRODUCTION

The state-of-the-art parallel static timing analysis (STA) algorithm is based on *task graph parallelism* (TGP) [2]. Unlike the traditional loop-based parallelism (level-by-level propagations using parallel loops), TGP formulates the graph-based timing propagation problem into an *STA task graph* where each node encapsulates a sequence of linearly dependent propagation tasks (e.g., RC update, slew and delay lookup) and each edge denotes a dependency between two nodes. This formulation explores *structural parallelism* from the circuit graph and delegates scheduling details, including dynamic load balancing and concurrency controls, to an established task graph runtime [3]. The result of TGP outperforms loop-based parallel timers up to 5× in large designs of millions of gates [2].

While TGP is effective on structural parallelism, the gain is also limited by the structure itself. Specifically, STA graphs can have several *linear segments* induced by constrained regions (e.g., through pins, false paths) and serial chains of gates. These linear segments prohibit the parallelization of their propagation tasks under the TGP model. Consider the task graph in Figure 1, exhibiting a maximum structural parallelism of two tasks, as shown at the top in Figure 2. Each node (task) encapsulates five linearly dependent timing propagation tasks. From the structural view, the linear chains, `B-D-F` and `C-E-G`, have no parallelism over their encapsulated tasks. Depending on the propagation algorithm, some of these tasks can be time-consuming (e.g., computing CPPR credits), and the serial execution of them can degrade the performance largely. A clever way is to overlap these tasks using *pipeline parallelism*, as shown at the bottom in Figure 2.

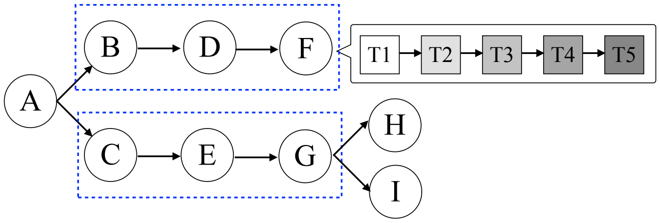

Figure 1: An STA task graph of nine nodes. Arrows represent dependencies between nodes. Each node has a sequential execution of five tasks, T1 to T5. Nodes `B`, `D`, and `F`, depicted in a blue rectangular, form a linear segment. Nodes `C`, `E`, and `G` form another linear segment.

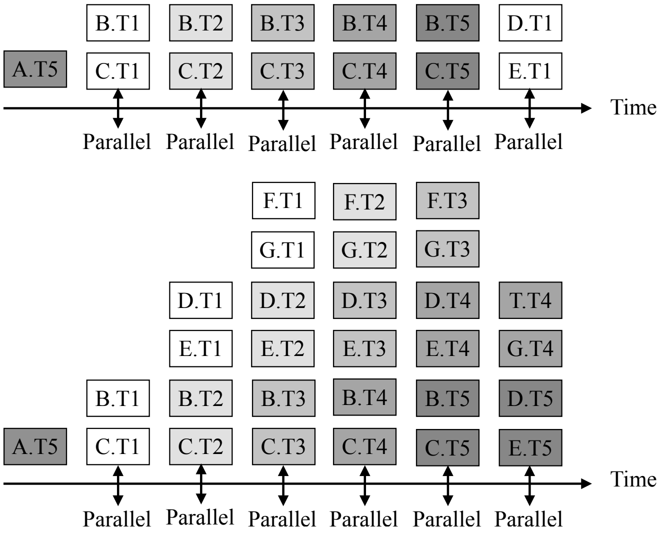

Figure 2: The execution timeline of Figure 1. The top describes structural parallelism to a degree of two, while the bottom describes both structural and pipeline parallelisms of up to six parallel tasks.

Consequently, we propose an efficient graph-based timing propagation framework by exploring both structural and pipeline parallelisms from an STA graph. Our framework enhances the performance of the TGP-based model by parallelizing propagation tasks in each linear segment using pipeline. Compared with a TGP-only baseline, our pipeline parallelism can further improve the performance by 16–25%.

DAC'22, July 10–14, 2022, San Francisco, CA, USA

ACM ISBN 978-1-4503-9142-9/22/07…$15.00
https://doi.org/10.1145/3489517.3530616

2 ALGORITHM

To explore both structural and pipeline parallelisms from an STA task graph, our framework identifies the linear segments and constructs a pipeline task for each segment. We leverage Taskflow [3] and its pipeline facility Pipeflow [1] to implement our pipeline. Based on Pipeflow's model, we declare one *pipe* for each *propagation task* encapsulated in each node of the STA task graph. The overall idea is to consider one node as a data token and propagate these tokens through a linear sequence of propagation tasks.

Algorithm 1 briefly explains the construction of a pipeline task for each linear segment, using the language in [1]. We declare a vector `pipes` to store each pipe (line 1). We call function `build_callable` and get a callable `pipe_callable` (line 2). In the function `build_callable`, we define the work of every propagation task in a node and set the length of the linear segment to be the termination condition of the pipeline task. Next, we specify the pipe type to be `tf::PipeType::SERIAL` as each task is executed sequentially in a node, and emplace `pipe_callable` into `pipes` up to `num_Tasks` times as there are `num_Tasks` propagation tasks per node (lines 3:5). Since the number of propagation tasks can change, we use `tf::ScalablePipeline` class, which allows variable assignment of pipes, to construct the pipeline task `pipeline_task` for the linear segment by specifying the length of the segment and two iterators (line 6). After constructing all pipeline tasks, we modify the dependencies in the task graph accordingly. For example, in Figure 1, task `A` precedes tasks `B` and `C` which would precede tasks `H` and `I`.

Algorithm 1: build_pipeline_task(linear_segment, num_Tasks)

```
1 $std :: vector < Pipe > pipes$;
2 $pipe\_callable \leftarrow$ build_callable( $linear\_segment, Pipeflow$ ) ;
3 foreach $i \in num\_Tasks$ do
4     $pipes.emplace( PipeType :: SERIAL, pipe\_callable)$;
5 end
6 $ScalablePipeline\ \ pipeline\_task($
      $linear\_segment.size(), pipes.begin(), pipes.end())$;
7 return $pipeline\_task$;
```

3 EXPERIMENTAL RESULTS

We evaluate the performance of our framework using real STA benchmarks from TAU18 Contest [2]. Each benchmark has a different ratio of linear segments in its STA task graph. We compile our programs using g++ 10.2 with C++17 standard `-std=c++17` and optimization flag `-O2` enabled. We run all the experiments on a Linux machine with Intel i7-9700K 8 Cores at 3.60GHz and 32 GB RAM. All data is an average of five runs. We consider the structural parallelism-only model in [2] as our baseline (denoted as "SP"). Our framework with pipeline is denoted as "SP+PP."

Table 1 shows the statistics and performance comparison of eight benchmarks between SP and SP+PP. The second column denotes the coverage of linear segments of lengths longer than 4 and 8, respectively. The coverage is defined as the percentage of the number of nodes in linear segments over the total number of nodes. The last column states the performance improvement of SP+PP over SP.

Table 1: Benchmark statistics and performance comparison between baseline (SP) and ours (SP+PP).

circuit	≥ 4/8	$\|V\|$	$\|E\|$	SP	SP+PP	Impr
s526_1	22.9/10.9%	911	1096	31ms	25ms	19%
s526_2	30.5/12.3%	971	1156	33ms	25ms	25%
s526_3	26.9/16.2%	951	1136	32ms	25ms	24%
s526_4	22.7/19.1%	921	1106	32ms	27ms	16%
vga_lcd_1	26.9/10.2%	412K	513K	13s	11s	20%
vga_lcd_2	27.3/15.1%	418K	519K	13s	10s	20%
wb_dma_1	28.0/14.5%	13K	17K	456ms	355ms	22%
wb_dma_2	31.3/14.1%	14K	17K	472ms	357ms	24%

We can see that SP+PP outperforms SP across all benchmarks. For example, in *vga_lcd_2*, SP+PP is 20% faster than SP.

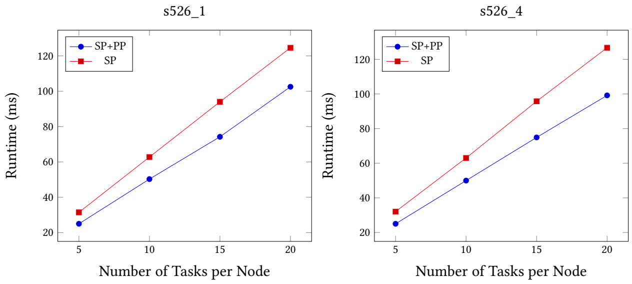

Figure 3: Runtime comparison between SP and SP+PP at increasing numbers of propagation tasks in each node.

Next, we demonstrate the benefits of pipeline parallelism when the number of sequential tasks encapsulated in each node increases. This pattern is prevalent in a graph-based analysis as algorithms can incorporate many tasks (e.g., CPPR, tags) during the linear propagation [2]. Here, we duplicate the propagation tasks to emulate this pattern. As shown in Figure 3, SP+PP outperforms SP in two selected benchmarks. The performance difference increases as we increase the number of tasks. When more propagation tasks are encapsulated in a node, the benefit of pipeline parallelism becomes significant.

4 ACKNOWLEDGEMENT

The project is supported by the NSF grants CCF-2126672 and CCF-2144523 (CAREER).

REFERENCES

[1] Cheng-Hsiang Chiu, Tsung-Wei Huang, Zizheng Guo, and Yibo Lin. 2022. Pipeflow: An Efficient Task-Parallel Pipeline Programming Framework using Modern C++. https://arxiv.org/abs/2202.00717

[2] Tsung-Wei Huang, Guannan Guo, Chun-Xun Lin, and Martin Wong. 2021. OpenTimer 2.0: A New Parallel Incremental Timing Analysis Engine. *IEEE TCAD* 4, 776–789.

[3] Tsung-Wei Huang, Dian-Lun Lin, Chun-Xun Lin, and Yibo Lin. 2022. Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System. In *IEEE TPDS*, Vol. 33. 1303–1320.