HeteroExcept: A CPU-GPU Heterogeneous Algorithm to Accelerate Exception-aware Static Timing Analysis

Zizheng Guo[1,2], Zuodong Zhang[1,2], Wuxi Li[5], Tsung-Wei Huang[6], Xizhe Shi[1], Yufan Du[1,3], Yibo Lin[1,2,4*], Runsheng Wang[1,2,4], Ru Huang[1,2,4]
[1]School of Integrated Circuits, Peking University [2]Institute of EDA, Peking University [3]School of EECS, Peking University
[4]Beijing Advanced Innovation Center for Integrated Circuits [5]AMD, Inc. [6]The University of Wisconsin at Madison

ABSTRACT

Static timing analysis (STA) for large-scale modern circuits requires extensive handling of false paths, multi-cycle paths, and other types of path exceptions. Despite the linear nature of timing propagation, we show that exception-aware STA is NP-hard and thus requires a long runtime to solve using conventional CPU-based methods. To overcome this runtime challenge, we propose a general CPU-GPU heterogeneous algorithm, HeteroExcept, that can handle common types of path exceptions and efficiently generate an accurate path report. Our algorithm targets runtime efficiency at the scale of thousands of exception rules and millions of circuit elements. To further improve the performance, we optimize our GPU implementation by introducing a cost-effective data exchange strategy between CPU and GPU. Experimental results demonstrate up to 6.84× and 12.93× speed-up compared to industrial timers, PrimeTime and OpenSTA.

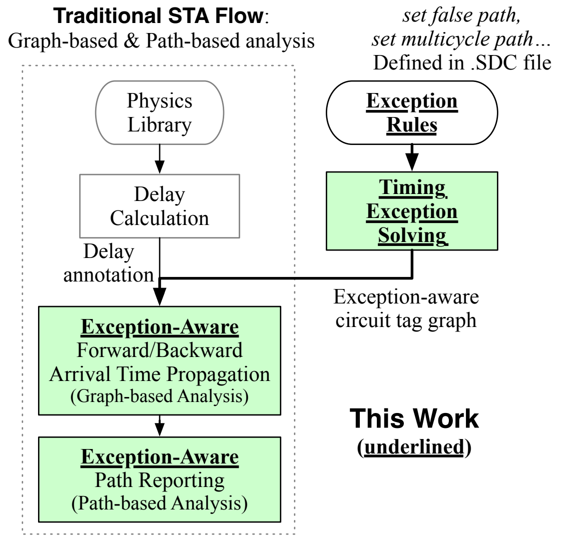

Traditional STA Flow:
Graph-based & Path-based analysis
set false path,
set multicycle path...
Defined in .SDC file
Physics Library
Exception Rules
Delay Calculation
Timing Exception Solving
Delay annotation
Exception-aware circuit tag graph
Exception-Aware Forward/Backward Arrival Time Propagation (Graph-based Analysis)
This Work (underlined)
Exception-Aware Path Reporting (Path-based Analysis)

Figure 1: An exception-aware STA flow.

1 INTRODUCTION

Static timing analysis (STA) is a critical component of the design flow as it verifies circuit functionality under the given clock frequency and timing constraints. A typical STA algorithm evaluates the circuit timing by propagating worst-case timing values (e.g., minimum slew, maximum delay, latest arrival time) from inputs through the circuit network to outputs. However, this approach can introduce errors because it does not consider the timing exceptions under real operating conditions. Examples of errors include false-positive critical paths that are logically impossible [1], multi-cycle or cross-domain paths that are not properly constrained [2], etc. These errors can prevent designers from obtaining real timing results, leading to wasted efforts in optimizing paths that are actually functioning correctly.

To address this issue, commercial STA engines incorporate a set of rules called *timing exceptions*, which can be specified by designers or optimization algorithms. Each timing exception acts on the paths that match a certain subgraph pattern by either stopping the timing check completely or applying special checking rules. As a result, exceptions enable us to address the limitations of STA, particularly in subgraphs where critical paths may be disregarded to accommodate the circuit functionality under real operating conditions. Exception-aware STA has become integral in high-performance circuit designs.

Existing exception-aware STA engines typically solve the exceptions before updating the timing on the circuit graph, as shown in Figure 1. An exception-aware circuit graph can usually be 10× larger [3] than the original circuit graph because pins must be split into tags to accommodate different rule-matching states. Creating this enlarged circuit graph as well as propagating exception-aware timing values through this graph is thus very time-consuming. For example, enabling timing exceptions with thousands of rules in OpenSTA [4] can slow down the runtime by 3×.

The efficiency of STA is critical for fast design closure since STA is used in the loop of many optimization stages, including logic synthesis, placement, routing, etc. To speed up STA, CPU-GPU heterogeneous algorithms have been proposed to accelerate various time-consuming STA tasks, such as delay calculation [5, 6], timing updates [7–12], path reports [13–16], and so on. Despite significant runtime improvement through CPU-GPU heterogeneous parallelism, these works are all *exception-oblivious*. Without the ability to handle timing exceptions, the practical use of these works is highly limited.

Nevertheless, GPU acceleration of graph algorithms like STA is known to be challenging because graph computing is highly irregular [2, 17–20]. Considering exceptions will make the problem even more challenging because it significantly complicates the STA graph (Challenge 1). As we shall present later in this paper, we formally prove that exception-aware STA is NP-hard in the worst case. Although practical exception-aware STA can be handled with pruning techniques, a comprehensive framework for handling various exception types is lacking (Challenge 2). Finally, to make the most

*Corresponding author: Yibo Lin (yibolin@pku.edu.cn). This work is supported in part by the National Science Foundation of China (Grant No. T2293700, T2293701), Natural Science Foundation of Beijing, China (Grant No. Z230002), NSF of the US grants 2235276, 2349144, 2349143, 2349582, 2349141, and the 111 Project (B18001).

ICCAD '24, October 27–31, 2024, New York, NY, USA
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1077-3/24/10...$15.00
https://doi.org/10.1145/3676536.3676651

of GPU acceleration, we need novel CPU-GPU heterogeneous data structures and kernel algorithms that can efficiently represent and solve exception states in a general setting (Challenge 3).

As a consequence, we present **HeteroExcept**, a CPU-GPU heterogeneous algorithm to accelerate exception-aware STA. We summarize our three key contributions as follows:

(1) We propose various novel pruning techniques for exception-aware STA targeting GPU runtime efficiency, including exception footprinting, aggressive storage reuse, and memory-hierarchy-aware algorithms (for Challenge 1).
(2) We present the first GPU-accelerated STA engine supporting the most common types of timing exceptions, including false paths, multi-cycle paths with multiple clock domains, path delay margins, and explicit delay overrides (for Challenge 2). With our proposed micro-exception framework, we achieve the best-in-class exception compatibility without complicating the design of the STA engine.
(3) We propose explicit improvements to CPU-GPU data exchange strategies for heterogeneous graph algorithms which we believe can be applied to a wide range of tasks (for Challenge 3).

We achieve significant speed-up over industrial standard timers. For full timing, HeteroExcept is up to 6.84× and 12.93× faster than PrimeTime and OpenSTA, respectively. Our runtime for *incremental timing updates* is even less than 0.2s for large designs with millions of gates and thousands of exceptions. We believe HeteroExcept will help designers and the EDA community greatly by providing an efficient and highly compatible STA toolset for modern designs.

The rest of this paper is organized as follows. Section 2 introduces the background of exception-aware STA. Section 3 formally shows that exception-aware STA is NP-hard. Section 4 presents our HeteroExcept engine. Section 5 demonstrates the experimental results. Finally, Section 6 concludes the paper.

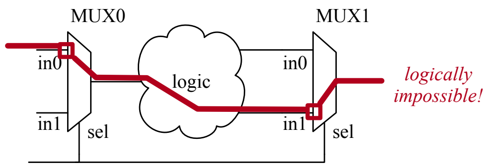
2 PRELIMINARIES

STA engine analyzes the signal delay and reports the most timing-critical signal paths inside the circuit. This information is then used by designers and VLSI CAD algorithms to optimize circuit performance and fix circuit race conditions during the entire design flow. During the STA process, the circuit is modeled as a directed acyclic graph (DAG) (V, E), in which nodes $v \in V$ represent circuit pins and edges $e \in E$ represent timing arcs of logic cells and wires [21]. Each timing arc on the DAG is annotated with a floating point delay value calculated by physics-aware delay models like Elmore, NLDM, and CCS before performing path extraction. This work assumes delay values as inputs and focuses on the exception-solving and path extraction steps which are key to the efficiency of the overall STA flow.

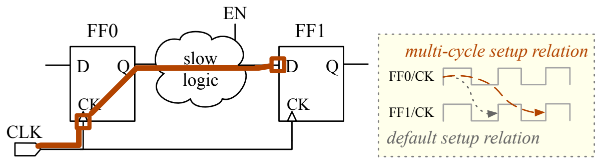
During path extraction, timing constraints are calculated and compared with the worst-case path delays to get path slacks. A path is a sequence of connected pins on the DAG from a timing startpoint (primary inputs, PI, or outputs of sequential elements) to a timing endpoint (primary outputs, PO, or inputs of sequential elements). Setup and hold constraints give the upper and lower bounds of path delays respectively conditioned on the clock domain involved. A number k is given by the designer and the STA engine searches for the top-k critical timing paths for setup and hold constraints separately.

Modern circuits exploit a variety of clocking and pipelining techniques during logic design. Due to logic properties and nontrivial path clocking, paths satisfying certain patterns in these circuits require special treatment in constraints calculation. Designers specify these patterns as *timing exceptions* in a Synopsys Design Constraints (.SDC, a Tcl dialect) file and feed it to signoff STA engines like PrimeTime and OpenSTA in order to obtain real timing-critical paths of such circuits. As shown in Figure 2, the most common types of timing exceptions are:

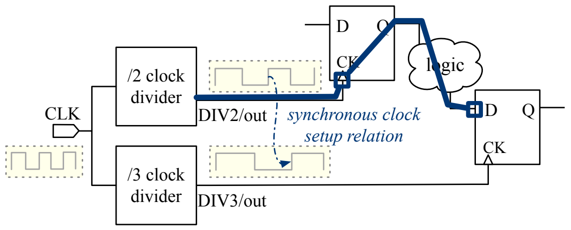
(a) **False paths**: paths satisfying given patterns are completely eliminated from timing reports because they are logically impossible to achieve.
(b) **Multi-cycle paths**: paths satisfying given patterns are marked as expanding multiple clock cycles during signal propagation. Their timing constraints are redefined accordingly.
(c) **Synchronous clock domains and generated clocks**: a design can have multiple synchronous clock domains with different waveforms, either explicitly defined or generated from clock frequency changers. The timing paths across different clock domains have special setup and hold relations.
(d) **Path margins**: a delay offset is applied to paths matching given patterns.
(e) **Delay overrides**: explicit delay assertions can be applied between arbitrary pairs of pins in the netlist, creating new timing startpoints and endpoints.



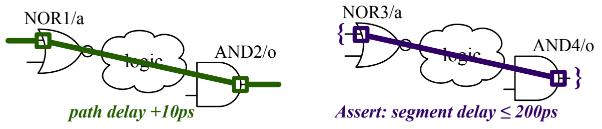
(a) **set_false_path** `-through MUX0/in0 -through MUX1/in1`



(b) **set_multicycle_path** `-setup 2 -from FF0 -to FF1`



(c) **create_generated_clock** `-divide_by 2 [get_pins DIV2/out]`
create_generated_clock `-divide_by 3 [get_pins DIV3/out]`



(d) **set_path_margin** `10 -through NOR1/a -through AND2/o`

(e) **set_max_delay** `200 -from NOR3/a -to AND4/o`

Figure 2: Different types of common timing exceptions.

Table 1: Overview of exception support in different academic and commercial STA engines.

STA engine	(a) false path	(b) multi-cycle path	(c) clock domain	(d) path margin	(e) delay override	GPU
OpenTimer [29]	×	×	×	×	×	×
False paths*	✓	×	×	×	×	×
[2, 28]	✓	✓	×	×	×	×
[30]	✓	✓	✓	×	✓	×
GPU Timers†	×	×	×	×	×	✓
OpenSTA [4]	✓	✓	✓	×	✓	×
PrimeTime [31]	✓	✓	✓	✓	✓	×
This work	✓	✓	✓	✓	✓	✓

*: False path only works include: [1, 17–20, 22–27].

†: GPU timer works include: [5–8, 13–16]. None of them supports any timing exceptions. [15] reports paths *inside* a subgraph, which is different from false path exceptions that globally *disable* a set of subgraphs (which turns out to be much harder, see Section 3).

Other types of timing exceptions like `set_disable_timing`, `set_case_analysis`, and `set_clock_groups` can be regarded as special cases of `set_false_path`. Signoff STA engines like PrimeTime and OpenSTA provide abundant support to various types of timing exceptions. Prior academic works [1, 2, 17–20, 22–28], on the other hand, mainly focused on false paths and multi-cycle paths only. Table 1 lists an overall comparison of exception support in previous STA engines.

3 NP-HARDNESS PROOF OF EXCEPTION

It has been known that the *identification* of all false paths in a circuit is NP-hard, due to its obvious equivalence to boolean satisfiability (SAT) and formal verification. To circumvent this, designers have been using knowledge of the frontend circuit design or automatic test pattern generation (ATPG) techniques to manually specify a set of subgraphs as false path exceptions to STA engines.

However, even with this *given* set of false path patterns as prior knowledge, STA engines still struggle to check circuit timing. Research on exception-aware STA has targeted pruning techniques like exhaustive path searching and tag-based algorithms that heuristically reduce the STA runtime, but none of these techniques have strong algorithm complexity guarantees. A very close attempt for provable complexity is made by Blaauw *et al* [24] who gave a polynomial time algorithm for special 2-stage false paths (e.g., without `-through` options), but their algorithm does not work for general false path patterns.

As a result, it has been wondered whether there exists a polynomial runtime STA method that can handle general exception patterns heavily used by designers nowadays [17, 30]. Contrary to the false path *identification* problem, STA with predefined simple path patterns to avoid seems much easier to handle as it is a pure graph problem and unrelated to circuit logic. Counter-intuitively and unfortunately, here we give a formal *negative* answer to the complexity conjecture for the first time in literature, by proving that STA with predefined timing exceptions is NP-hard, even only considering false path exceptions. This implies that no polynomial algorithm can exist for exception-aware STA unless P=NP.

Specifically, we have proved the following theorem:

Theorem 1. *Any 3-SAT problem instance can be polynomially reduced to an equivalent static timing analysis problem instance with a set of 3-stage false path patterns.*

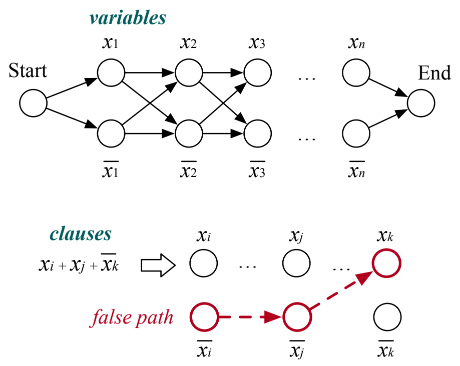


Figure 3: Illustration of our 3-SAT reduction: DAG represents variables and false path exceptions represent CNF clauses.

3-SAT is among the first combinatorial problems proved to be NP-hard [32]. It asks for the satisfiability of a boolean formula in conjunctive normal form (CNF) where every clause has at most 3 literals.

For any 3-SAT problem instance with n variables and m clauses, we create a timing DAG with $2n + 2$ nodes and $4n$ edges:

$$V = \{\text{Start}, \text{End}, x_1, x_2, ..., x_n, \overline{x_1}, \overline{x_2}, ..., \overline{x_n}\}, \tag{1}$$

$$E = \{\text{Start} \rightarrow x_1, \text{Start} \rightarrow \overline{x_1}, x_n \rightarrow \text{End}, \overline{x_n} \rightarrow \text{End}\} \cup$$
$$\{x_{i-1} \rightarrow x_i, \overline{x_{i-1}} \rightarrow x_i, x_{i-1} \rightarrow \overline{x_i}, \overline{x_{i-1}} \rightarrow \overline{x_i} \mid i = 2, 3, ..., n\}. \tag{2}$$

The edge delays are all set to 1. We then transform the m clauses to m false path exceptions on this graph. The false path pattern matches the nodes corresponding to the negation of literals in the clause. For example, for a clause $(x_i + x_j + \overline{x_k})$ with $i < j < k$, it is equivalent to $\overline{\overline{x_i} \cdot \overline{x_j} \cdot x_k}$ by de Morgan's law. We add the following false path exception:

$$\textbf{set\_false\_path}\ \texttt{-through}\ \overline{x_i}\ \texttt{-through}\ \overline{x_j}\ \texttt{-through}\ x_k. \tag{3}$$

These constructions are visualized in Figure 3. For completeness of timing definition, we add a dummy clock, set the arrival time of node Start to 0, and the required arrival time of node End to 0 relative to this clock.

```
create_clock -name dummy
set_input_delay -max 0 Start -clock dummy        (4)
set_output_delay -max 0 End -clock dummy
```

With the above construction, a 3-SAT problem is reduced to an exception-aware STA problem. We have the following lemma,

Lemma 1. *Every valid timing path in the constructed timing graph corresponds to a variable assignment that satisfies all 3-SAT clauses.*

Proof. By the initial conditions in Equation (4), every constrained timing path connects node Start and End. The graph constructed by Equation (2) has a layered structure, forcing every path to go through exactly one node in $\{x_1, \overline{x_1}\}$, then exactly one node in $\{x_2, \overline{x_2}\}$ and so forth. There are 2^n possible paths matching every assignment of the n boolean variables.

Among these assignments, the valid ones should satisfy all m clauses. For example, to satisfy one clause $(x_i + x_j + \overline{x_k})$, at least one of x_i, x_j, and $\overline{x_k}$ should be true. This is equivalent to ensuring $\overline{x_i}$, $\overline{x_j}$, and x_k are not simultaneously adopted. This constraint is exactly modeled by the false path exception in Equation (3). □

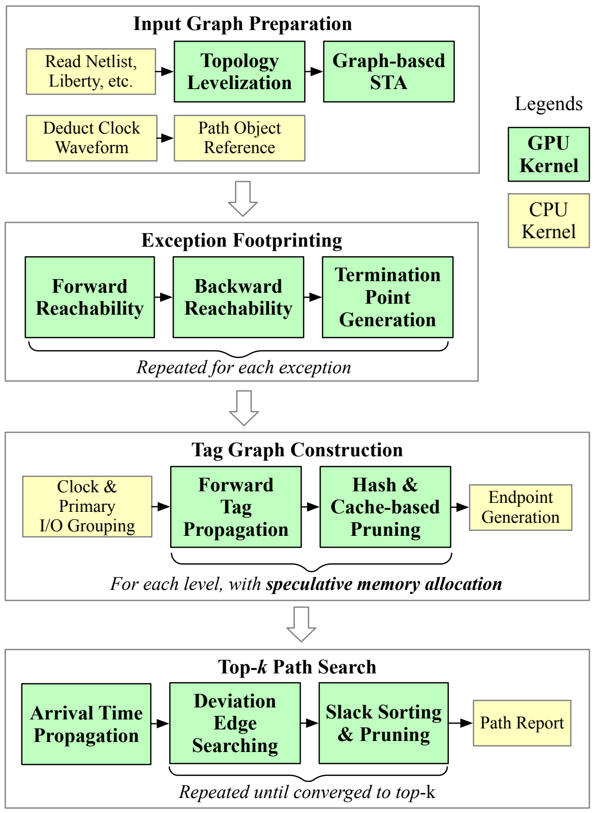


Figure 4: Our HeteroExcept algorithm flow.

The proof of Theorem 1 is straightforward given Lemma 1. By asking for the top-1 path, a valid critical path is reported if and only if the 3-SAT problem is satisfiable. The NP-hardness conclusion follows this reduction [33] as if there exists a polynomial timing exception solver, it can then be used to solve any 3-SAT problem in polynomial time. A similar NP-completeness conclusion can be easily derived for a decision version of exception-aware STA that only asks whether a critical path exists or not. Due to page limit, we skip the claim and proof for NP-completeness. We note that this reduction method also rules out the possibility of many approximation algorithms for exception-aware STA.

4 THE HETEROEXCEPT ALGORITHMS

Although NP-hardness is a negative conclusion to the complexity question, we emphasize that better pruning techniques and heterogeneous parallelism are still feasible that can solve *practical* timing exceptions a lot more efficiently.

In this work, we propose HeteroExcept, an efficient heterogeneous exception engine for STA. We build our flow on top of prior works on GPU-accelerated STA including levelization and graph-based STA algorithms in [7, 8] and path search algorithms in [13, 14, 16]. Our algorithm flow (Figure 4) consists of 4 major steps: input graph preparation, exception footprinting, tag graph construction, and top-k path search. We use a combination of CPU and GPU computation tasks (see legends in Figure 4) to accelerate exception handling on a heterogeneous platform, providing unprecedented runtime speedup. To achieve this, we employ a series of novel data structures and CPU-GPU interaction strategies.

4.1 Micro-Exception Framework to Handle General Exceptions

We introduce support to all 5 exception types mentioned in Section 2 in a unified framework. This makes HeteroExcept by far the most general exception engine in academic literature. HeteroExcept achieves this by extending the *tag*-based exception solver widely used in prior works.

A tag maintains the matching states of exception patterns to distinguish paths that fall under different exceptions, as illustrated in Figure 5. Each tag contains a set of partly-matched exception patterns called the *rule set*. A pin can have multiple tags each with its own signal arrival time (e.g., node 3 in Figure 5). During signal propagation, the rule sets in tags are also updated (e.g., from node 3 to nodes 4 and 5 in Figure 5).

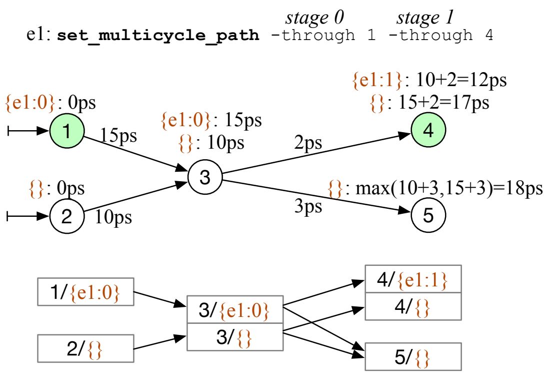


Figure 5: Example of tag-based exception handling, with a 2-stage exception pattern.

Upon a successful match, an exception modifies the timing check behavior of the path. With multiple exceptions matched, STA engine shall consider their interactions. For example,

- When there are multiple matching exceptions of one same type, the one with the highest priority should be adopted. A higher priority is assigned to exceptions with more specific path patterns.
- Different types of exceptions should be adopted simultaneously. For example, a path can cross different clock domains, be a multi-cycle path, and be affected by a slack margin all at the same time.
- Delay overrides (set min/max delay) might create new timing startpoints and endpoints if pins are specified in `-from` and `-to` options, respectively. Paths from these new startpoints have special arrival times starting from zero and are detached from the fanin pins. These paths no longer recognize multi-cycle exceptions, but can still be affected by false paths or path margins.

Correctly modeling these exception interactions is challenging and can easily complicate the implementation and acceleration of an exception engine. To address this complexity, we add one intermediate representation (IR) below the exception types which we call *micro-exceptions*. Micro-exceptions are to designer-visible timing exceptions as CPU microcodes are to programmer-visible CPU instructions. In HeteroExcept, we have the following 5 types of micro-exceptions:

(1) `KILL_AT_COMPLETE`: removes the whole tag when the exception pattern is matched.

(2) `KILL_AT_COMPLETE_IF_NOT_OWN_START`: removes the whole tag when the exception pattern is matched, if the tag is not from special tag generated by `START_OWN`.
(3) `NO_PROP_AFTER_COMPLETE`: marks the tag as final and prevents its propagation to fanout pins when the exception pattern is matched.
(4) `START_OWN`: when the exception pattern starts to match, creates a special tag at the start pin with zero arrival time.
(5) `CLEANUP_PRIORITY`(type, priority): when the exception pattern is matched, the rules with the same type but lower priority are pruned from the tag rule set.

Each exception is compiled into a set of micro-exceptions by Algorithm 1 that collectively implement its complex behavior.

Algorithm 1: Exception translation.

```
1  micro_exceptions ← {};
2  if exception is a false path then
3      Add KILL_AT_COMPLETE into micro_exceptions;
4  if exception is a delay override then
5      if exception has -from pin then
6          Add START_OWN into micro_exceptions;
7      if exception has -to pin then
8          Add NO_PROP_AFTER_COMPLETE into
            micro_exceptions;
9      if exception has both -from pin and -to pin then
10         Add
            KILL_AT_COMPLETE_IF_NOT_OWN_START
            into micro_exceptions;
11 if exception is a multi-cycle path, path margin, or delay
    override then
12     Add CLEANUP_PRIORITY into micro_exceptions;
```

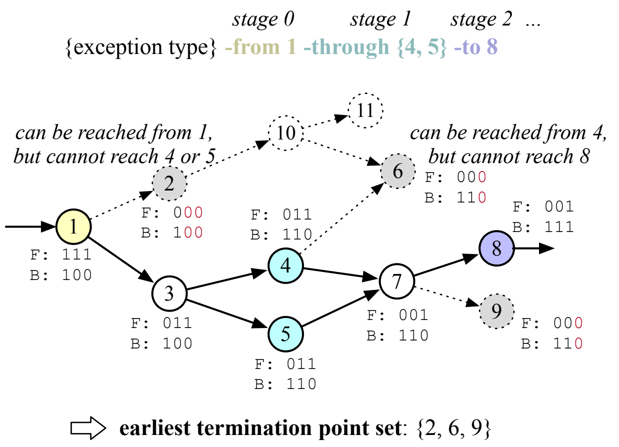
By executing the micro-exceptions, each tag at timing endpoints contains a rule set with clean matched exceptions readily to be considered for calculating required arrival time and slacks. Micro-exceptions help us control the complexity of our exception solver by making it general and flexible.

4.2 Exception Footprinting

With n exceptions, it is easy to see that the maximum number of distinct tag rule sets can exceed 2^n. In addition to modeling the interactions between exceptions, it is thus also essential to prevent unnecessary interactions by strictly limiting the range of effect for every exception with best practical effort. To this end, we present a novel preprocessing step called exception footprinting (the second step block in Figure 4) before constructing the tag graph. Exception footprinting provides a succinct representation of the affected regions for every exception pattern, which in turn greatly saves runtime during tag-based exception solving.

The basic idea of exception footprinting is to compute a set of region boundary pins, which we call an earliest termination point set. A termination point is defined as a pin that can be reached from the exception subgraph, but cannot reach the pins at next stages (see Figure 6). Once a tag reaches a termination point, the corresponding rule (if exists) will be pruned from the rule set of that tag because it will no longer be matched successfully.

To find the termination points, we perform a forward propagation and a backward propagation on the circuit topology. During both propagations, we compute the reachability of every node to all stages



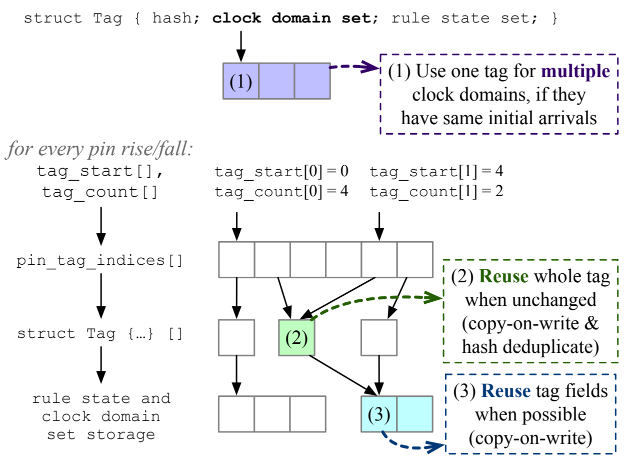
stage 0
stage 1
stage 2 ...
{exception type} -from 1 -through {4, 5} -to 8
can be reached from 1, but cannot reach 4 or 5
can be reached from 4, but cannot reach 8
1 F: 111 B: 100
2 F: 000 B: 100
3 F: 011 B: 100
4 F: 011 B: 110
5 F: 011 B: 110
6 F: 000 B: 110
7 F: 001 B: 110
8 F: 001 B: 111
9 F: 000 B: 110
10
11
earliest termination point set: {2, 6, 9}

Figure 6: An example of exception footprinting, illustrating the stages (in colors), earliest termination points (in gray), and the forward/backward reachability vectors (marked with `F:` and `B:`).

in the exception pattern. We obtain 2 bit vectors called forward and backward reachability vectors for every node. By OR-ing the 2 vectors and checking if the result vector contains 0, we can then determine whether the node is a termination point and obtain the earliest termination points.

4.3 Technical Highlights on Tag Pruning

To further reduce the runtime cost for the expensive tag propagation, we develop a multi-layer approach for tag pruning, as shown in Figure 7. These pruning techniques deliberately target a few frequent exception patterns in large multi-clock-domain circuit designs and turn them into optimization opportunities.



struct Tag { hash; clock domain set; rule state set; }
(1)
(1) Use one tag for multiple clock domains, if they have same initial arrivals
for every pin rise/fall:
tag_start[], tag_count[]
tag_start[0] = 0 tag_count[0] = 4
tag_start[1] = 4 tag_count[1] = 2
pin_tag_indices[]
struct Tag {…} []
(2)
(2) Reuse whole tag when unchanged (copy-on-write & hash deduplicate)
rule state and clock domain set storage
(3)
(3) Reuse tag fields when possible (copy-on-write)

Figure 7: Multiple stage design of tag database and our pruning techniques.

Our first observation is that multiple clocks can be defined on the same source pins with different frequencies by providing `-add` when creating the clock. This is often used in video decode designs. The arrival times for these different clock domains are the same regardless of the delay propagation. As a result, a single tag is sufficient to represent all of them. To this end, we add a clock domain set field alongside the rule set field in the tag structure (see (1) in Figure 7). A tag representing multiple clock domains will be split later when needed, e.g., when an exception matches only a part of the clock domains in the tag.

We further observe that only a very small fraction (less than 1%) of pins will modify the tag. Most of the pins instead just pass on the tags from their fan-ins. In HeteroExcept, we thus reuse tags and tag fields whenever possible. This technique is called copy-on-write which is often used in functional programming. We use pointers and indices to reference the whole tags and the tag fields (see (2) and (3) in Figure 7 respectively). We postpone the creation of new tag or allocation of tag field memory until there is a change.

In tag propagation, we always ensure there are no duplicate tags on every pin when merging the fan-in tags. Due to the multi-layer tag structure, the detection of tag duplication involves dozens of costly indirect memory references. We accelerate the duplication detection by hashing the tags. We need a strong collision-resistant hash function for tags that is agnostic to the order of elements in orderless sets like clock domains and rule states.

At the core of our hashing function (Algorithm 2) is the integer bit mixing [34] primitive (`hash_u64`). This integer hashing makes use of efficient bitwise operations modulo 2^{64}. The magic constants inside it are random odd numbers ensuring that the hashing is 1-1 invertible. The tag hasher (`hash_tag`) sums the mixed bits of all clocks and rules, with a last-bit distinction between the two.

Algorithm 2: Tag hashing.

```
1  Function hash_u64(x)            ▷ bit mixing primitive [34]:
2      $x \leftarrow x \oplus$ 0x1cb8b9d87bc84a7;
3      $x \leftarrow (x \oplus (x \gg 30)) \times$ 0xbf58476d1ce4e5b9;
4      $x \leftarrow (x \oplus (x \gg 27)) \times$ 0x94d049bb133111eb;
5      $x \leftarrow x \oplus (x \gg 31)$;
6      return x;
7  Function hash_tag(tag):
8      $h \leftarrow$ 0x1;
9      for $clock \in tag.clocks$ do
10         $h \leftarrow h +$ hash_u64($clock \ll 1$);
11     for $rule \in tag.rules$ do
12         $h \leftarrow h +$ hash_u64($rule \ll 1 \mid 1$);
13     return hash_u64($h$);
```

By fully exploiting the memory hierarchy of heterogeneous devices, we can even further boost the tag propagation efficiency. Both CPUs and GPUs have small and ultrafast cache near their computation units (L1 cache and thread-local memory respectively). In kernel implementation, we store the hashes of the first few tags inside a software cache array that fits in these ultrafast caches. We observe that the number of tags is less than 6 for most (75%+) pins even on our largest designs. Our tag propagation kernel on these pins will work completely inside caches and thus eliminate nearly all global memory accesses.

4.4 Efficient CPU-GPU Storage Management

Dynamic memory management is one major difficulty in porting graph algorithms to GPU. Graph algorithms often need to allocate small pieces of memory on every node for temporary or result storage purposes. A parallel graph algorithm thus needs to parallelize the per-node memory allocation. A good way to avoid thread race conditions for memory allocation is to pre-allocate a memory pool and distribute the memory pieces according to the memory demand

Algorithm 3: Local cache-based deduplication in kernels.

```
1  NUM_CACHE_LINE ← 6;
2  cache ← [empty × NUM_CACHE_LINE];
3  for every new tag propagated do
4      Search for same hash in cache lines;
5      if not found in cache but the cache is full then
6          Search again for the same hash in stored tag memory,
             by comparing it with all current tags;
7      if same hash found wherever then
8          Remap the new edges pointing to the existing tag;
9          Continue to process next tag;
10     Add new tag and edges to memory;
11     if there is empty line in cache then
12         Add tag hash into cache;
```

of threads. However, the demand for threads is usually highly imbalanced and hard to predict as it is determined by algorithm context.

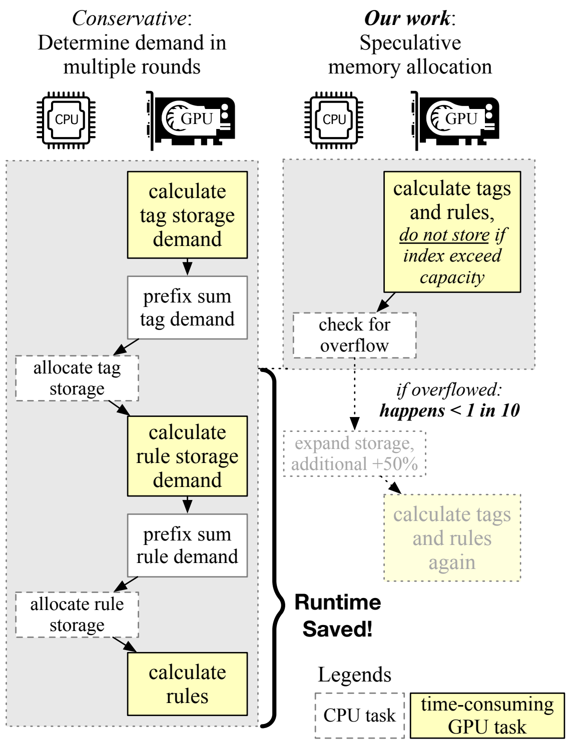
To solve the problem, a "compute-twice" approach is widely adopted in previous works [13, 14, 16, 35]. The first round of computation only figures out the memory demand without writing out the results. After calculating the demand, a parallel prefix sum is performed to obtain the total amount of memory to allocate, as well as the memory offsets for all threads. Finally, the memory is allocated and a second round of computation is performed which writes out the result to the memory. While this method is precise in memory allocation, it forces one to perform the same computation twice which is wasteful. For highly sophisticated graph algorithms like tag propagation, it may even need multiple rounds to calculate the memory demand which leads to unacceptable performance.

HeteroExcept adopts a novel *speculative* approach for memory allocation, as shown in Figure 8. Instead of calculating memory demand for tags and nested fields in advance which requires up to 3 rounds of computation, we compute tags and fields in a single round that may fail or succeed. The memory pool is managed heuristically like `std::vector` in C++. The GPU kernel is informed of its size and maximum capacity. When the memory writes exceed the capacity during GPU parallel execution, the size is advanced as usual but actual memory writes are not performed. After one round of calculation, CPU checks whether the final size exceeds the capacity to find if there has been an overflow. If an overflow has happened, CPU expands the memory pool with the precise demand now available, rolls back the size, and re-runs the GPU kernel. Thanks to the heuristic expansion, more than 90% of the time the kernel needs to run optimally once.

5 EXPERIMENTAL RESULTS

We implement our exception-aware STA flow as **HeteroExcept**, a heterogeneous exception engine for general timing exception analysis. HeteroExcept is written in high-performance C++, CUDA, and Rust, exporting user interface as a general STA engine.

To demonstrate the runtime benefit of HeteroExcept over cutting-edge commercial tools, we compare the runtime of HeteroExcept with two baselines, PrimeTime (2021.06) and OpenSTA (latest), on a set of million-sized circuit design originated from TAU timing analysis contest [36]. We do not compare with previous GPU-based timers because they do not support exception handling (Table 1). For each design, we randomly generate up to 4,216 exceptions of different types, each by randomly selecting a path and then randomly



Conservative: Determine demand in multiple rounds
Our work: Speculative memory allocation
CPU
GPU
CPU
GPU
calculate tag storage demand
prefix sum tag demand
allocate tag storage
calculate rule storage demand
prefix sum rule demand
allocate rule storage
calculate rules
calculate tags and rules, do not store if index exceed capacity
check for overflow
if overflowed: happens < 1 in 10
expand storage, additional +50%
calculate tags and rules again
Runtime Saved!
Legends
CPU task
time-consuming GPU task

Figure 8: Illustration of speculative memory allocation efficien is much more efficient than conservative multi-rounds memory allocation especially on tag propagation.

choosing 1–3 pins on the path as through points to form an exception pattern. We pre-generate delay annotations in SDF format for every design and feed them to the STA engines under test. We switch off common path pessimism removal (CPPR) in both baselines. All STA engines start with the same set of circuit graphs and delays which ensures a fair comparison. Our algorithm does not sacrifice any accuracy and the reported paths by HeteroExcept achieve *full match* on slack values compared to PrimeTime.

We conduct all experiments on an Ubuntu 22.04 Linux host with 64 cores AMD CPU and 1 RTX 3090 GPU. The main memory is 378GB for CPU and the GPU memory is 24GB. Wall-clock runtime is measured on the `report_timing` call after reading the design files. Each data point is the average of 3 runs.

5.1 Full Timing and Incremental Timing Performance

Table 2 lists benchmark statistics and a detailed runtime comparison. HeteroExcept outperforms both PrimeTime and OpenSTA on all benchmarks we have tested, either running on CPU or GPU. The speed-up ratio on GPU is 2.25×–6.84× over PrimeTime and 2.99×–12.93× over OpenSTA. The average speed-up is 4.53× and 8.16× on GPU over PrimeTime and OpenSTA respectively.

Even on the same CPU hardware without GPU, HeteroExcept algorithm can still outperform PrimeTime and OpenSTA by 2× on average, thanks to our novel tag data structures and pruning techniques designed to solve exceptions efficiently. HeteroExcept on GPU achieves significant speed-up over CPU except on one smallest case `aes_core` where the total runtime is already negligible.

It is known that STA engines are frequently invocated inside inner loops of the design process such as timing-driven placement and routing. In these scenarios, the performance requirement is even more critical as the timing analysis needs to be repeated thousands of times along with the design iterations. These scenarios are called *incremental timing update* where the delays are updated but path exceptions remain unchanged. As a result, the construction of tag graph becomes a one-time cost and the timing update on the fixed tag graph will instead dominate the flow runtime.

Thus, we also introduce support to incremental timing updates in HeteroExcept and measure the runtime in Table 3. Before the incremental update, we perform a full update and feed a new SDF file with randomly modified delay values. The incremental updates are drastically faster than full updates. For example, it only takes HeteroExcept less than 0.2 seconds (200 ms) to update the timing of a million-sized design with thousands of exceptions. On average, our GPU-accelerated incremental timing update costs only 2.1% of a full update runtime, which is over 40× speed-up. Compared with incremental updates on CPU, the GPU-accelerated incremental update is 20× faster. As a result, HeteroExcept is very helpful in timing-driven design optimization applications.

5.2 Runtime Scalability

We further analyze the runtime scalability of HeteroExcept and the baselines under different numbers of CPU threads (Figure 9) and paths (Figure 10).

The CPU parallelism of OpenSTA saturates at around 4 threads. PrimeTime generally runs faster with more threads, up to its maximum supported 32 threads. Our CPU version of HeteroExcept runs best under 16 threads where it can outperform PrimeTime with 32 threads. All of them eventually hit a performance wall due to the fundamental limitation of CPU parallelism. The heterogeneous GPU version of HeteroExcept successfully overcomes this limitation and introduces a large speed-up thanks to the power of heterogeneous parallelism and our efficient algorithms. HeteroExcept on GPU also benefits from increase in CPU threads because it also contains tasks on CPU (see Figure 4).

With an increasing number of reported paths up to k=10K, HeteroExcept shows a consistent high speed. HeteroExcept also shows a much smaller increase in runtime for each additional path searched. This efficiency aligns with previous GPU-accelerated STA works based on similar parallel prefix-suffix expansion algorithms [13, 14, 16], showing a successful migration of heterogeneous path searching to exception-aware tag graphs.

5.3 Ablation Study and Runtime Breakdown

We look deeper into the performance of HeteroExcept by conducting an ablation study for our speculative allocation technique (Section 4.4) and a runtime breakdown. Figure 11 shows the total tag propagation runtime for design `netcard` split into different steps. Speculative execution reduces our GPU kernel runtime by 2.4× compared to conservative allocation. The accelerated tag propagation step is now faster than exception footprinting (25% vs. 46%) in the runtime breakdown. Endpoint generation also takes 19% of runtime, in which further speed-up is possible through GPU acceleration in future work.

6 CONCLUSION

This paper presents HeteroExcept, an ultrafast heterogeneous STA engine for general exceptions including false paths, multi-cycle paths, and more. With fine-grained optimization techniques including exception footprinting, micro-exception translation, tag pruning, and speculative execution, HeteroExcept achieves up to 6.84× and 12.93×

Table 2: Runtime comparison between PrimeTime, OpenSTA, and HeteroExcept (Ours) on TAU 2015 benchmark.

Benchmark	Circuit Statistics			#Excepts	PrimeTime (16C)		OpenSTA (16C)		Ours CPU (16C)		Ours GPU (16C + 1G)	
	#Gates	#Nets	#Pins		RT (ms)	Ratio	RT (ms)	Ratio	RT (ms)	Ratio	RT (ms)	Ratio
aes_core	22938	23199	66221	66	625.13	3.90	479.69	2.99	**119.96**	0.75	160.36	1.00
b19_iccad	255278	255300	776320	782	3275.13	5.75	6494.89	11.40	1438.02	2.52	**569.67**	1.00
des_perf_ispd	138878	139112	371587	373	1904.74	6.66	2322.59	8.12	938.79	3.28	**286.10**	1.00
edit_dist_ispd	147650	150212	416609	419	2448.76	2.25	7823.66	7.18	2421.99	2.22	**1090.23**	1.00
fft_ispd	38158	39184	116139	116	894.92	4.71	634.33	3.34	237.90	1.25	**189.93**	1.00
leon2_iccad	1616369	1616984	4178874	4216	24732.50	4.27	44241.53	7.63	20392.20	3.52	**5796.63**	1.00
leon3mp_iccad	1247725	1247979	3267993	3297	18856.88	4.80	50775.08	12.93	12200.72	3.11	**3925.81**	1.00
matrix_mult_ispd	164040	167242	475186	478	2444.96	3.33	6294.99	8.57	2445.20	3.33	**734.78**	1.00
mgc_edit_dist_iccad	161692	164254	444693	448	2565.11	3.50	5971.82	8.14	1438.15	1.96	**733.28**	1.00
mgc_matrix_mult_iccad	171282	174484	489670	493	2538.92	3.76	8230.28	12.19	2034.64	3.01	**675.32**	1.00
netcard_iccad	1496719	1498555	3901343	3936	19872.67	4.84	37811.71	9.21	16566.81	4.03	**4106.54**	1.00
pci_bridge32_ispd	40790	40950	108172	108	804.50	4.26	1055.49	5.59	319.03	1.69	**188.78**	1.00
vga_lcd_iccad	259067	259152	662179	667	3514.43	6.84	4526.05	8.81	1633.48	3.18	**513.79**	1.00
Avg. Ratio		-		-	-	4.53	-	8.16	-	2.60	-	1.00

RT: runtime in milliseconds (10^{-3}s). **Ratio**: runtime ratio compared to HeteroExcept on GPU. **#Excepts**: number of exceptions in SDC.
All baselines are run with 16 CPU cores and 1 GPU, reporting $k = 100$ paths.

Table 3: Our runtime (ms) for incremental timing update.

Benchmark	Full CPU	Full GPU	Incr. CPU	Incr. GPU
aes_core	119.96	160.36	5.24	1.96
b19_iccad	1438.02	569.67	51.32	9.99
des_perf_ispd	938.79	286.10	27.65	4.46
edit_dist_ispd	2421.99	1090.23	176.55	23.12
fft_ispd	237.90	189.93	8.68	2.34
leon2_iccad	20392.20	5796.63	3048.17	182.50
leon3mp_iccad	12200.72	3925.81	1830.59	109.92
matrix_mult_ispd	2445.20	734.78	378.18	18.87
mgc_edit_dist_iccad	1438.15	733.28	121.85	16.53
mgc_matrix_mult_iccad	2034.64	675.32	323.50	17.15
netcard_iccad	16566.81	4106.54	2281.60	93.55
pci_bridge32_ispd	319.03	188.78	9.42	2.73
vga_lcd_iccad	1633.48	513.79	230.37	9.55
Avg. Ratio	2.605	1.000	0.279	0.021

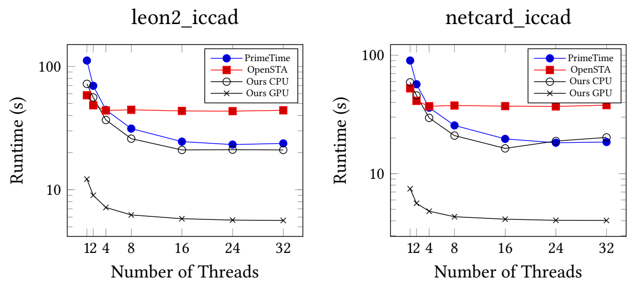


Figure 9: Runtime scalability of PrimeTime, OpenSTA, and HeteroExcept (Ours) under different numbers of CPU cores on design `leon2` and `netcard`, reporting $k = 10$ paths.

speed-up over industrial standard STA engines. We believe HeteroExcept can benefit a wide range of timing-driven chip design tasks by providing an ultrafast exception-aware timing engine.

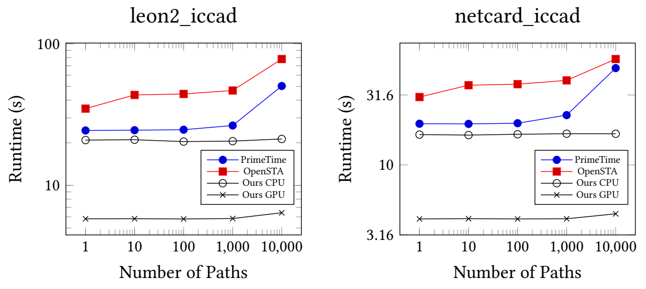


Figure 10: Runtime on different number of paths k for PrimeTime, OpenSTA, and HeteroExcept (Ours) on design `leon2` and `netcard`, all on 16 CPU cores.

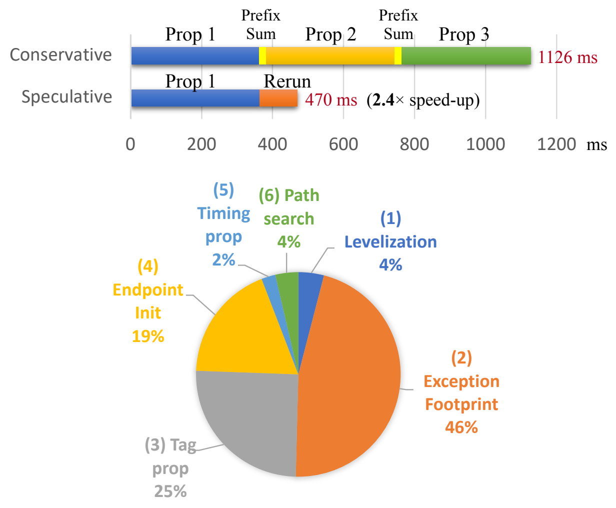


Figure 11: Ablation study on design `netcard` for speculative allocation and the runtime breakdown.

REFERENCES

[1] K. Belkhale and A. Suess, "Timing analysis with known false sub graphs," in *Proc. ICCAD*. San Jose, CA, USA: IEEE Comput. Soc. Press, 1995, pp. 736–739.

[2] Shuo Zhou, Bo Yao, Hongyu Chen, Yi Zhu, Chung-Kuan Cheng, and M. Hutton, "Efficient static timing analysis using a unified framework for false paths and multi-cycle paths," in *Proc. ASPDAC*. Yokohama, Japan: IEEE, 2006, pp. 73–78.

[3] W. Li, Y. Kukimoto, G. Servel, I. Bustany, and M. E. Dehkordi, "Calibration-based differentiable timing optimization in non-linear global placement," in *Proc. ISPD*. ACM, 2024, p. 31–39.

[4] "OpenSTA," https://github.com/The-OpenROAD-Project/OpenSTA.

[5] Z. Guo, T.-W. Huang, Z. Jin, C. Zhuo, Y. Lin, R. Wang, and R. Huang, "Heterogeneous static timing analysis with advanced delay calculator," in *Proc. DATE*, 2024.

[6] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. F. Young, and M. D. Wong, "GCS-Timer: Gpu-accelerated current source model based static timing analysis," in *Proc. DAC*, 2024.

[7] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proc. ICCAD*. ACM, 2020.

[8] ——, "Accelerating static timing analysis using cpu-gpu heterogeneous parallelism," *IEEE TCAD*, pp. 1–1, 2023.

[9] ——, "A provably good and practically efficient algorithm for common path pessimism removal in large designs," in *Proc. DAC*. ACM, 2021.

[10] T.-W. Huang, D.-L. Lin, C.-X. Lin, and Y. Lin, "Taskflow: A Lightweight Parallel and Heterogeneous Task Graph Computing System," in *IEEE TPDS*, vol. 33, no. 6, 2022, pp. 1303–1320.

[11] T.-W. Huang, C.-X. Lin, G. Guo, and M. D. F. Wong, "Cpp-Taskflow: Fast Task-based Parallel Programming using Modern C++," in *Proc. IPDPS*. IEEE, 2019, pp. 974–983.

[12] T.-W. Huang and M. D. Wong, "OpenTimer: A high-performance timing analysis tool," in *Proc. ICCAD*. IEEE, 2015, pp. 895–902.

[13] Z. Guo, T.-W. Huang, and Y. Lin, "HeteroCPPR: Accelerating common path pessimism removal with heterogeneous cpu-gpu parallelism," in *Proc. ICCAD*. ACM, 2021.

[14] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *Proc. DAC*. ACM, 2021.

[15] ——, "Gpu-accelerated critical path generation with path constraints," in *Proc. ICCAD*, 2021, pp. 1–9.

[16] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. D. F. Wong, "A gpu-accelerated framework for path-based timing analysis," *IEEE TCAD*, pp. 1–1, 2023.

[17] H. Chen, "Static Timing Analysis with False Paths and Combinational Loops," PhD Thesis, University of Minnesota, USA, 2004.

[18] S. Tsai and C.-Y. R. Huang, "A false-path aware formal static timing analyzer considering simultaneous input transitions," in *Proc. DAC*. San Francisco CA: ACM, 2009, pp. 25–30.

[19] Chul Rim, Soo-Hyun Kim, Joo-Hyun Park, Myung-Soo Jang, Jin-Yong Lee, Kyu-Myong Choi, and Jeong-Taek Kong, "Fast and practical false-path elimination method for large SoC designs," in *Proc. SOCC*. Portland, OR, USA: IEEE, 2003, pp. 397–400.

[20] S. Zhou, C.-K. Cheng, B. Yao, H. Chen, Y. Zhu, M. Hutton, T. Collins, S. Srinivasan, N.-C. Chou, and P. Suaris, "Efficient timing analysis with known false paths using biclique covering," *IEEE TCAD*, vol. 26, no. 5, pp. 959–969, 2007.

[21] J. Bhasker and R. Chadha, *Static Timing Analysis for Nanometer Designs: A Practical Approach*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[22] Haizhou Chen, Bing Lu, and Ding-Zhu Du, "Static timing analysis with false paths," in *Proc. ICCD*. Austin, TX, USA: IEEE Comput. Soc, 2000, pp. 541–544.

[23] J.-J. Liou, A. Krstic, L.-C. Wang, and K.-T. Cheng, "False-path-aware statistical timing analysis and efficient path selection for delay testing and timing validation," in *Proc. DAC*, Jun. 2002, pp. 566–569.

[24] D. Blaauw, R. Panda, and A. Das, "Removing user specified false paths from timing graphs," in *Proc. DAC*. Los Angeles, CA, USA: ACM Press, 2000, pp. 270–273.

[25] Shuo Zhou, Bo Yao, Hongyu Chen, Yi Zhu, Chung-Kuan Cheng, M. Hutton, T. Collins, and S. Srinivasan, "Improving the efficiency of static timing analysis with false paths," in *Proc. ICCAD*. San Jose, CA: IEEE, 2005, pp. 527–532.

[26] D. H. Du, S. H. Yen, and S. Ghanta, "On the general false path problem in timing analysis," in *Proc. DAC*. New York, NY, USA: ACM, 1989, pp. 555–560.

[27] S. Tsukiyama, M. Tanaka, and M. Fukui, "Techniques to remove false paths in statistical static timing analysis," in *Proc. ASICON*. Shanghai, China: IEEE, 2001, pp. 39–44.

[28] G. Lucas, C. Dong, and D. Chen, "Variation-Aware Placement With Multi-Cycle Statistical Timing Analysis for FPGAs," *IEEE TCAD*, vol. 29, no. 11, pp. 1818–1822, 2010.

[29] T. Huang, G. Guo, C. Lin, and M. D. F. Wong, "OpenTimer v2: A New Parallel Incremental Timing Analysis Engine," *IEEE TCAD*, vol. 40, no. 4, pp. 776–789, 2021.

[30] M. Hutton, D. Karchmer, B. Archell, and J. Govig, "Efficient static timing analysis and applications using edge masks," in *Proc. FPGA*. Monterey CA USA: ACM, 2005, pp. 174–183.

[31] "Synopsys PrimeTime," http://www.synopsys.com.

[32] R. M. Karp, *Reducibility among combinatorial problems*. Springer, 1972.

[33] R. G. Michael and S. J. David, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., 1979.

[34] G. L. Steele, D. Lea, and C. H. Flood, "Fast splittable pseudorandom number generators," in *Proceedings of the 2014 ACM International Conference on Object Oriented Programming Systems Languages &amp; Applications*, ser. OOPSLA '14. New York, NY, USA: ACM, 2014, p. 453–472.

[35] Z. Guo, F. Gu, and Y. Lin, "Gpu-accelerated rectilinear steiner tree generation," in *Proc. ICCAD*. ACM, 2022.

[36] J. Hu, G. Schaeffer, and V. Garg, "TAU 2015 contest on incremental timing analysis," in *Proc. ICCAD*. IEEE, 2015, pp. 882–889.