# Path-Based Timing Analysis Acceleration via Segment-Level Timing Arc Reuse


Yuyang Ye
CUHK
Hong Kong, China

Xiangfei Hu
Southeast University
Nanjing, China

Leyun Tian
Southeast University
Nanjing, China

Chang Meng*
Eindhoven University of Technology
Eindhoven, The Netherlands

Qing He
Tongji University
Shanghai, China

Longxing Shi
Southeast University
Nanjing, China



## Abstract

Path-Based Static Timing Analysis (PBA) ensures sign-off accuracy but suffers from excessive runtime. We propose a novel PBA acceleration framework that reduces computational complexity through segment-level timing arc reuse, rather than merely speeding up per-arc computations. By decomposing timing paths into multi-fanin-bounded segments, we enable delay reuse across structurally identical contexts. A dual-indexed *SegHashMap* caches segment delays under varying input slews using red-black trees. To ensure error-bound reuse, we train a slew-sensitive timing model with multi-view NLDM embedding and Sobolev-aligned supervision to preserve delay gradients. Our segment-centric engine reconstructs path delays via reusable segments with bounded pessimism. Experiments show our method achieves an average 195× speedup over parallel PBA baselines while preserving accuracy and pessimism, and 2–3× faster than GPU-based methods on CPU alone.


## 1 Introduction

Static Timing Analysis (STA) is a critical step in chip design to ensure timing closure. It can operate in either graph-based mode (GBA) or path-based mode (PBA). While GBA is fast but pessimistic, PBA eliminates excessive pessimism by analyzing timing on a per-path basis [1–5]. However, PBA is notoriously time-consuming (i.e., often 10–1000× slower than GBA) due to the exponential number of timing paths that must be analyzed [4–6]. As design sizes grow, PBA becomes a major bottleneck in industrial sign-off flows.

To address the runtime challenge, prior work has explored CPU and GPU acceleration. Multi-threaded PBA improves throughput but quickly saturates due to limited core scalability. State-of-the-art (SOTA) task-parallel PBA achieves modest gains beyond 16 threads. Block-based methods yield only ~1.63× speedup due to sequential bottlenecks [1]. Recently, GPU-based frameworks have been introduced to exploit massive parallelism [4, 6–11]. GPU-PBA [4] demonstrates 46× speedup on a 1.6M-gate design using GPU-accelerated arc evaluation. Nonetheless, these approaches retain the original arc/path workload. In this case, fundamental computational complexity remains unchanged, and path explosion problem persists.

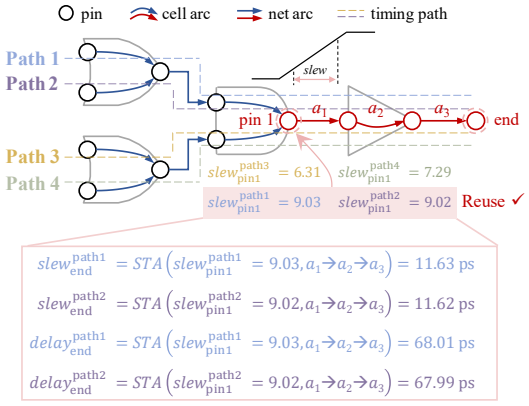



**Figure 1: Example of segment-level timing arc reuse.**

This work takes a complementary approach: instead of merely accelerating arc-level delay evaluations, we reduce the fundamental computational complexity of PBA by reusing timing results across structurally overlapping timing path segments. As illustrated in Fig. 1, many critical paths share common subpaths. We decompose each path into reusable *timing path segments* (e.g., $a_1 \to a_2 \to a_3$), each bounded by a *multi-fanin node* (e.g., `pin1`), and reuse timing evaluations under similar input slews. For example, four critical paths (`path1`–`path4`) all traverse the arc sequence $\{a_1, a_2, a_3\}$ from `pin1` to a common endpoint. In `path1` and `path2`, the output slews at `pin1` are 9.02 ps and 9.03 ps, resulting in only 0.02 ps and 0.01 ps difference in delay and slew. Such negligible deviation permits error-bounded reuse of segment delay on `path1`, eliminating redundant evaluations on `path2` and reducing overall PBA complexity.

Despite its promise, segment-level timing arc reuse introduces several challenges:
(1) *How can reuse be applied efficiently?* Reuse is only beneficial if timing segments are frequently shared and can be accessed efficiently. Arc-level reuse suffers from high sensitivity and low redundancy, leading to costly tracking and recomputation. A coarser reuse granularity—via structurally defined segments and level-aware scheduling—is needed to maximize reuse and minimize overhead.
(2) *How can accuracy be guaranteed?* Input slew mismatch across paths may cause non-negligible timing errors. We need a principled deviation model that captures slew-induced delay variations and ensures conservative reuse with bounded error.
(3) *How can reusable segments be integrated into path-based timing analysis?* A reuse-centric engine must support fast delay lookup, slew propagation, and end-to-end path reconstruction without ever falling back to redundant arc-by-arc traversal.


*Corresponding author.




*DAC '26, July 26–29, 2026, Long Beach, CA, USA*



ACM ISBN 979-8-4007-2254-7/2026/07…$15.00
https://doi.org/10.1145/3770743.3804358

To address the above challenges, we propose a fast path-based timing analysis framework that reduces computational complexity via segment-level timing arc reuse. Given a timing graph and critical paths, the framework decomposes paths into reusable segments, stores timing results in a structured *SegHashMap*, and schedules them level-by-level. The framework consists of four stages: (1) Levelized Timing Graph Segmentation decomposes the timing graph into topologically ordered path segments bounded by multi-fanin nodes. Each segment is assigned a level based on its backward distance from endpoints. (2) Level-wise Segment Timing Calculation precomputes segment delays in parallel under descending input slew values. All results are stored in a dual-indexed *SegHashMap*, enabling scalable reuse across levels. (3) Level-crossing Segment Timing Arc Reuse applies a slew-sensitive deviation model to determine whether existing segment delays can be reused under perturbed slew inputs. This ensures bounded pessimism while minimizing redundant computation. (4) Fast Path Timing Analysis reconstructs end-to-end delays by chaining reusable segments and propagating slews across levels. This enables high-throughput, pessimism-preserving evaluation over the entire path. Our contributions are summarized as follows:

- We accelerate PBA by reducing its computational complexity through segment-level timing arc reuse, offering a distinct alternative to traditional per-arc acceleration methods.
- We design a dual-indexed *SegHashMap* combining segment hashing and slew-keyed red-black trees, supporting scalable caching and $O(\log n)$ nearest-slew lookup for efficient and high-throughput timing arc reuse under slew variations.
- We develop a slew-sensitive, error-aware segment timing model by embedding Non-linear delay model (NLDM) patches in timing library through multi-view encoding. A Sobolev-aligned supervision strategy preserves delay gradient sensitivity to slew variations, enabling robust reuse under bounded error.
- By reconstructing path delays from reusable segments with fast chaining and slew propagation, we eliminate redundant arc evaluations, achieving **195× speedup** over SOTA parallel CPU PBA.

## 2 Preliminaries

### 2.1 Timing Graph

In static timing analysis (STA), the circuit netlist is modeled as a directed acyclic graph (DAG), where each node represents a pin (e.g., primary input/output or cell pin), and each directed edge represents a timing dependency between pins.

**Definition 1** (Timing arc)**.** A *timing arc* is the minimal unit of delay propagation. It describes the delay from a source pin to a destination pin, and is categorized into two types: (1) **cell arcs**, representing delay from input to output pin of a cell, and (2) **net arcs**, representing wire delay between cell pins.

A *timing path* is defined as follows:

**Definition 2** (Timing path)**.** A *timing path* is a sequence of pins and timing arcs that begins at a **startpoint** (a primary input or clock pin of a register) and ends at an **endpoint** (a primary output or data pin of a register). It is the fundamental unit for delay checking in STA.

To support reuse-based optimization, we introduce a key node concept that determines timing segment boundaries:

**Definition 3** (Multi-fanin pin node)**.** A *multi-fanin pin node* is a node that has two or more fan-in timing arcs. Such nodes typically act as convergence points for multiple timing paths and are selected as natural boundaries for segment decomposition.

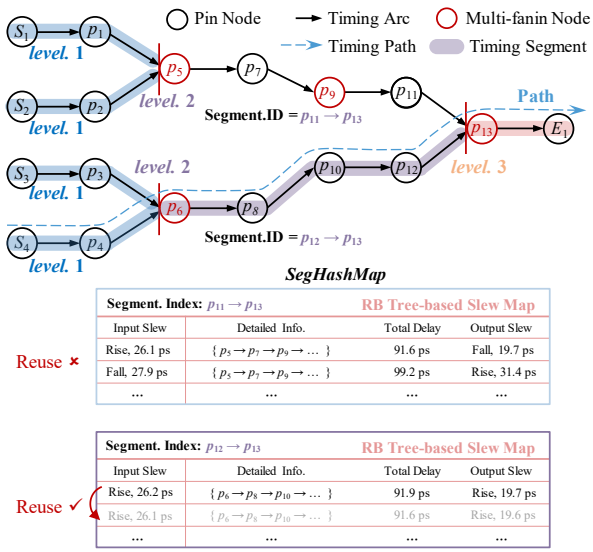



**Figure 2: Top: Timing graph is partitioned into timing segments based on timing arcs and multi-fanin nodes. Bottom: *SegHashMap* organizes segment timing results with two-level indexing: segment ending arc and input slew.**

Based on these definitions, we define the key reusable structure:

**Definition 4** (timing segment)**.** A *timing segment* is a maximal contiguous sub-path within a timing path, bounded by two multi-fanin pin nodes (inclusive of the former). All internal nodes of the segment have exactly one fan-in arc. Timing information (slew, capacitance, etc.) can be reused based on the segment.

To illustrate the concepts more clearly, Fig. 2 provides an example that includes timing arcs, timing paths, multi-fanin pin nodes, and timing segments.

### 2.2 Problem Formulation

Given that path-based timing analysis (PBA) must evaluate a large number of critical paths, many of which share common timing segments, it is desirable to reuse arc-level computations across such overlaps. We formalize this objective as follows:

**Problem 1** (Reuse-based PBA acceleration)**.** Given a timing graph $\mathcal{G}$ extracted from a digital circuit netlist, and its decomposition into timing segments $\{\boldsymbol{seg}_1, \boldsymbol{seg}_2, \dots\}$, the goal is to minimize redundant arc timing computations across all timing paths by: (1) reusing precomputed delay and slew information for segments under similar input conditions, and (2) ensuring bounded error in path delay while maximizing reuse coverage.

## 3 Proposed Framework

We now introduce a scalable timing analysis pipeline based on segment-level reuse and level-wise graph scheduling. Below, we first summarize the core idea and briefly preview each technical component. To enable scalable and accurate timing analysis, we propose a segment-level reuse framework that transforms fine-grained timing graphs into reusable computation units.

In **Section 3.1 (Levelized Timing Graph Segmentation)**, we decompose the timing graph into directed acyclic timing segments using reverse BFS traversal. Each segment corresponds to a single-fan-in arc chain terminated at a multi-fanin boundary node, which ensures acyclic segmentation and enables level-wise grouping into sets $\{\mathcal{S}_k\}_{k=0}^{K}$ for parallel evaluation.

In **Section 3.2 (Level-wise Segment Timing Calculation)**, we selectively compute exact delay–slew responses at representative segments and store them in a hierarchical container, the *SegHashMap*—a two-level structure with hash indexing and red-black tree lookup. It ensures fast and scalable storage during timing analysis.

In **Section 3.3 (Level-Crossing Segment Timing Arc Reuse)**, we propose a reuse mechanism driven by a neural delay predictor trained with Sobolev alignment, which provides both value and gradient estimates of segment delay under varying slews. This enables error-bounded, pessimism-preserving reuse of timing results.

In **Section 3.4 (Fast Path Timing Analysis)**, we describe the full path-level procedure that allocates per-segment error budgets, evaluates segments level-by-level, applies learned reuse, and reconstructs total path delay via segment accumulation.

## 3.1 Levelized Timing Graph Segmentation

The segmentation of the timing graph is performed via reverse traversal using breadth-first search (BFS) starting from each endpoint. Since timing paths originating from different endpoints are independent, the segmentation process can be parallelized.

To efficiently construct timing segments based on timing path structure and multi-fanin pin nodes, we follow the steps below: Step 1: Initialize a new timing segment from each unassigned node. Step 2: Extend the segment backward along its unique fan-in arc, collecting a sequence of nodes and timing arcs. Step 3: Terminate the extension upon reaching a multi-fanin pin node, i.e., a node with two or more incoming timing arcs (see Definition 3). Step 4: Include this boundary node as the starting point of the segment. This ensures that each internal node in the segment has exactly one fan-in, and that any potential slew merging occurs only at segment boundaries.

After all reverse BFS traversals are complete, the timing graph is partitioned into timing segments, as illustrated in Fig. 2. To facilitate timing propagation in later stages, we assign a level to each segment, defined as the maximum number of segments between it and any connected startpoint. All segments with the same level $k$ are grouped into the level-specific set $\mathcal{S}_k$.

## 3.2 Level-wise Segment Timing Calculation

To reduce redundant computations in path-based timing analysis, we selectively precompute timing information for a subset of *representative* timing segments. Instead of exhaustively enumerating all possible input slews for every segment, we compute and store only those timing states that are *not replaceable* via reuse (see Section 3.3). This selective precomputation strategy preserves accuracy while significantly lowering memory and computation overhead.

**Segment Timing Calculation**. Given a representative input slew $s_{\text{in}}$, the timing of a timing segment is computed by forward delay–slew propagation through its constituent arcs:

(1) For **cell arcs**, the delay $d_{\text{cell}}$ and output slew $s_{\text{out}}$ are obtained via:

$$(d_{\text{cell}}, s_{\text{out}}) = \mathcal{T}_{\text{NLDM}}(s_{\text{in}}, c_{\text{load}}), \tag{1}$$

where $\mathcal{T}_{\text{NLDM}}$ is a 2D NLDM LUT indexed by input slew $s_{\text{in}}$ and load capacitance $c_{\text{load}}$, with bilinear interpolation.

(2) For **net arcs**, we employ a reduced-order RC abstraction $\mathcal{N}_{\text{RC}}$ (e.g., via moment matching) and compute

$$(d_{\text{net}}, s_{\text{out}}) = \mathcal{F}_{\text{M}}(s_{\text{in}}, \mathcal{N}_{\text{RC}}), \tag{2}$$

where $\mathcal{F}_{\text{M}}$ is an analytical delay model (e.g., Elmore).

The input slew for each arc is derived from the output slew of its immediate predecessor, forming a forward propagation chain. As the delay accumulates across arcs, we obtain the total delay of all arcs in one segment. Finally, for each input slew $s_{\text{in}}$, the output timing information $(d, s_{\text{out}})$ of the segment is stored into the slew map under its corresponding ending arc index.

**Segment Timing Information Storage**. Precomputed timing results are organized in a two-level *SegHashMap*: (1) The first level is a hash map that stores the *slew map* of the second level, indexed by the ending timing arc of each segment. From a circuit view, the ending arc is uniquely defined by a single fan-in chain (see Section 3.1), ensuring stable indexing. (2) The second level (i.e., slew map) is a red-black tree that uses representative input slews (with value and polarity) as keys to store their corresponding timing results. This supports $\mathcal{O}(\log n)$ insertion and nearest-key lookup for reuse.

To illustrate this structure clearly, we provide an example in Fig. 2, where each segment endpoint is hashed to an entry, and its slew-dependent timing values are organized for fast retrieval via the red-black tree. The *SegHashMap* provides a compact yet fast-access storage mechanism, ensuring constant-time segment-level lookup and logarithmic-time slew-level resolution, which together support scalable and fine-grained timing reuse during analysis.

**Level-wise Parallelization**. As introduced in Section 3.1, timing segments are organized into level-specific sets $\{\mathcal{S}_k\}_{k=0}^{K}$ based on their topological distance from startpoints. Segments within the same level set $\mathcal{S}_k$ exhibit no interdependencies, enabling conflict-free and parallel scheduling across multiple threads. During parallel timing calculation, all segments in $\mathcal{S}_0$ are processed first, followed by those in $\mathcal{S}_1$, and so on. This level-wise execution order guarantees that all required predecessor information (e.g., the output slew of the terminal arc in fan-in segments from $\mathcal{S}_{k-1}$) is already available in the container before evaluating any segment in $\mathcal{S}_k$.

## 3.3 Level-crossing Segment Timing Arc Reuse

To suppress redundant delay computations in path-based analysis, we propose a level-crossing timing arc reuse mechanism. Here, *level-crossing* refers to timing propagation across segment levels, where the output slew of segments in $\mathcal{S}_{k-1}$ serves as the input to a segment in $\mathcal{S}_k$. A segment in $\mathcal{S}_k$ may thus receive multiple input slews due to fan-in from different predecessors. To reduce redundancy, we examine whether some slews can be merged (i.e., treated as equivalent) and whether timing results for one input slew can be reused under bounded error for another. Such reuse is allowed only when the slew-induced delay difference is within a predefined tolerance, ensuring negligible error and guaranteed pessimism. This is enabled by a learned, slew-sensitive delay predictor that captures fine-grained variations and guides reuse under bounded error. Our three-stage framework consists of: (1) multi-view segment embedding to encode local slew-delay patterns, (2) Sobolev-aligned delay prediction to enhance sensitivity modeling, and (3) error-bounded segment reuse under slew perturbation.

**Multi-View Segment Embedding from NLDM Tables**. Each timing segment ***seg*** consists of a topologically ordered sequence of cell

arcs $\{a_i\}_{i=1}^{N_{\text{arc}}}$. For reuse decisions, we focus on cell arcs because their delay is highly sensitive to slew, while net arcs are typically insensitive and already handled via analytical models. To capture local timing behavior, we construct a dual-view segment embedding $e_{\text{seg}}$ by aggregating arc-level representations $\{e_{a_i}\}_{i=1}^{N_{\text{arc}}}$, where each $e_{a_i}$ is derived from two complementary perspectives. An illustrative example of arc-level timing information is shown in Fig. 3.

**(1) Graph-structured table view:** For each cell arc $a_i$, we extract a 3×3 local patch from its NLDM table centered at the nominal query point $(s_{\text{in}}, c_{\text{load}})$ and instantiate a grid graph $\mathcal{G}_{a_i} = (\mathcal{V}_{a_i}, \mathcal{E}_{a_i})$ over the nine surrounding slew–load combinations. Each node in $\mathcal{V}_{a_i}$ represents one grid point, connected to its 8-neighbors (both axial and diagonal); edge types distinguish axial from diagonal transitions. Each node $v \in \mathcal{V}_{a_i}$ is assigned a feature vector:

$$\boldsymbol{x}_v = [d_v,\ \Delta s_v,\ \Delta c_v,\ \phi(\Delta s_v, \Delta c_v)], \tag{3}$$

where $d_v$ is the NLDM delay at that query point, $(\Delta s_v, \Delta c_v)$ are the relative offsets from the patch center, and $\phi(\cdot)$ denotes a positional one-hot encoding that explicitly encodes node position within the patch. This spatial encoding enhances the model's ability to perceive asymmetric curvature and local shape of the delay surface.

A lightweight GNN is then applied over $\mathcal{G}_{a_i}$ using $K$=2 layers of edge-aware multi-head GAT [12], with residual connections, LayerNorm, and dropout for stability. Final node embeddings are aggregated via attentive pooling and projected into a segment vector:

$$\boldsymbol{e}_{a_i}^{\text{tab}} = \text{Proj}\left(\text{AttnPool}\left(\text{GAT}_{v\in\mathcal{V}_{a_i}}(\boldsymbol{x}_v)\right)\right). \tag{4}$$

This yields a curvature-aware embedding that captures fine-grained slew–capacitance sensitivity.

**(2) Metadata view:** Each arc $a_i$ is also encoded using a complementary set of serialized metadata that reflects circuit-level semantics and GBA timing values. We consider the following four categories: (1) Arc type: including cell name, pin direction. (2) Intrinsic capacitance: output pin capacitance as defined in the NLDM library. (3) GBA-reported timing outputs: including output delay and output slew under nominal operating conditions. (4) GBA-reported timing input: the actual input slew (including value and rise/fall) observed at this arc during gate-based analysis. These tokens are embedded via a shared MLP and concatenated to form a metadata vector $\boldsymbol{e}_{a_i}^{\text{text}}$.

To ensure cross-view consistency, we introduce a contrastive loss:

$$L_{\text{cons}} = -\log \frac{\exp(\langle \boldsymbol{e}_{a_i}^{\text{tab}},\ \text{sg}[\boldsymbol{e}_{a_i}^{\text{text}}]\rangle)}{\sum_j \exp(\langle \boldsymbol{e}_{a_i}^{\text{tab}},\ \text{sg}[\boldsymbol{e}_{j}^{\text{text}}]\rangle)}. \tag{5}$$

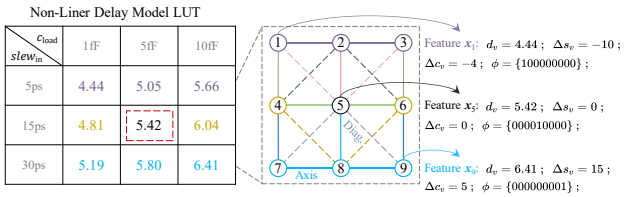
A Transformer encoder [13] aggregates arc embeddings to obtain the full segment representation:

$$\boldsymbol{e}_{\text{seg}} = \text{Transformer}(\{e_{a_i}^{\text{tab}} || e_{a_i}^{\text{text}}\}_{i=1}^{N_{\text{arc}}}). \tag{6}$$

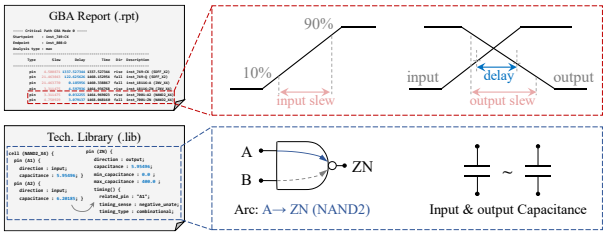
**Slew-Sensitive Delay Prediction via Sobolev Alignment**. Given $\boldsymbol{e}_{\text{seg}}$ and input slew $s$, a neural predictor estimates both the delay $\hat{d}$ and its gradient $\hat{g} = \widehat{\partial d/\partial s}$. To improve sensitivity to subtle slew variations, we apply a Sobolev-style loss composed of three terms for regularization:

(1) **Value term:** Huber loss [14] between predicted and true delay:

$$L_{\text{delay}} = \text{Huber}(\hat{d}, d) = \begin{cases} \frac{1}{2}(\hat{d}-d)^2 & \text{if } |\hat{d}-d| \le \delta, \\ \delta\cdot(|\hat{d}-d| - \frac{1}{2}\delta) & \text{otherwise,} \end{cases} \tag{7}$$




**(a) Graph-structured NLDM table view**




**(b) Metadata view**

**Figure 3: Example of timing arc information used in our slew-sensitive segment timing model from multiple views, including NLDM table view and timing-related metadata view.**

where $\delta = 0.05$ ps, which reflects the tolerable segment delay deviation under reuse and balances smooth regression and robustness.

(2) **Derivative term:** Difference between predicted and finite gradients, enforcing consistency in local slew sensitivity:

$$g^\star = \frac{d(s+\delta) - d(s-\delta)}{2\delta}, \quad L_{\text{deriv}} = \|\hat{g} - g^\star\|_1. \tag{8}$$

(3) **Perturbation consistency:** Reconstructed perturbation should match predicted linear response:

$$L_{\text{pert}} = \left|\hat{d}(s+\delta s) - \hat{d}(s) - \hat{g}\cdot\delta s\right|, \quad \delta s \sim \mathcal{N}(0, \sigma^2). \tag{9}$$

The total loss is:

$$L = \alpha L_{\text{delay}} + \beta L_{\text{deriv}} + \gamma L_{\text{pert}} + \lambda_{\text{cons}} L_{\text{cons}}, \tag{10}$$

where $\alpha, \beta, \gamma, \lambda_{\text{cons}}$ are tunable (equal to 0.25 in our work). This Sobolev alignment makes the predictor explicitly slew-sensitive and locally linearizable, enabling reliable reuse under slew variations.

**Error-Bounded Reuse Across Slew Variants**. To decide reuse, we evaluate a segment ***seg*** at two input slews $s_1 > s_2$, both with the same structural configuration. Let $\hat{d}_1 = \hat{d}(e_{\text{seg}}, s_1)$ and $\hat{d}_2 = \hat{d}(e_{\text{seg}}, s_2)$. We define the absolute delay deviation: $\varepsilon = |\hat{d}_1 - \hat{d}_2|$. If $\varepsilon \le \tau$, we reuse the timing computed at $s_1$ for the smaller slew $s_2$.

## 3.4 Fast Path Timing Analysis

To enable accurate yet efficient path timing analysis, we propose an evaluation framework that integrates reuse-aware segment processing with path-level accumulation. Given a timing graph, a timing-path set $\mathcal{P}$ and a total error tolerance $\tau_{\text{total}}$, the procedure is shown in Algorithm 1 and proceeds in three sequential phases.

**Phase 1: Segment-level Error Budget Allocation**. To manage reuse-induced approximation error, we allocate the total error budget $\tau_{\text{total}}$ to individual segments proportionally to their reuse potential. Each segment receives a local error bound:

$$\tau_{\text{seg}} = \tau_{\text{total}} \times \frac{N_{(\text{seg})}}{N}, \tag{11}$$

**Algorithm 1:** Path Timing Analysis via Timing Arc Reuse

**Input:** Timing graph $G$, timing path set $\mathcal{P} = \{P^{(1)}, P^{(2)}, \ldots, P^{(N)}\}$, total error tolerance $\tau_{\text{total}}$
**Output:** Path delays $\{D^{(i)}\}_{i=1}^{N}$ and output slews $\{S_{\text{out}}^{(i)}\}_{i=1}^{N}$

```
/* P1:Segment-Level Error Budget Allocation */
```
Partition graph $G$ into segments $\{\boldsymbol{seg}\}$ `// Section 3.1`
**foreach** *segment* $\boldsymbol{seg}$ **do**
  $N_{(\text{seg})} \leftarrow$ count number of paths in $\mathcal{P}$ that traverse $\boldsymbol{seg}$;
  Allocate local error budget $\tau_{\text{seg}}$ via Eq. (11);
```
/* P2:Level-wise Segment Evaluate and Reuse */
```
**for** $k \leftarrow 1$ **to** $K$ **do**
  **forall** *segment* $\boldsymbol{seg} \in \mathcal{S}_k$ ***in parallel*** **do**
    Get input slews $\{s_1, \ldots, s_M\}$ from level-$(k-1)$ fan-ins;
    Sort slews in descending order: $s_1 > \cdots > s_M$;
    Compute exact delay $d_1$ at $s_1$; insert into *SegHashMap*;
    `// Described in Section 3.2`
    **for** $m \leftarrow 2$ **to** $M$ **do**
      Find nearest larger computed slew $s_{\text{ref}}$;
      $d_{\text{ref}} \leftarrow \textit{SegHashMap}[\boldsymbol{seg}][s_{\text{ref}}]$;
      $\hat{d}_m \leftarrow \mathcal{F}_{\text{pred}}(e_{\text{seg}}, s_m)$ `// Section 3.3 model`
      $\varepsilon \leftarrow |d_{\text{ref}} - \hat{d}_m|$;
      **if** $\varepsilon \le \tau_{seg}$ **then**
        Reuse $d_{\text{ref}}$ for $s_m$;
      **else**
        Compute delay at $s_m$; insert into *SegHashMap*;
```
/* P3:Path-wise Timing Accumulation */
```
**foreach** *timing path* $P^{(i)} = \{\boldsymbol{seg}_1, \ldots, \boldsymbol{seg}_L\} \in \mathcal{P}$ **do**
  $D^{(i)} \leftarrow 0$, $s_1 \leftarrow$ actual input slew for $P^{(i)}$;
  **for** $j \leftarrow 1$ **to** $L$ **do**
    $d_j \leftarrow \textit{SegHashMap}[\boldsymbol{seg}_j][s_j]$;
    $D^{(i)} \leftarrow D^{(i)} + d_j$;
    $s_{j+1} \leftarrow$ output slew of $\boldsymbol{seg}_j$;
  $S_{\text{out}}^{(i)} \leftarrow s_{L+1}$;
**return** $\{D^{(i)}\}_{i=1}^{N}$, $\{S_{out}^{(i)}\}_{i=1}^{N}$

where $N_{(\text{seg})}$ and $N$ denote the number of timing paths traversing segment $\boldsymbol{seg}$ and the total number of timing paths. Segments shared by more paths receive more budget to maximize reuse efficiency.

**Phase 2: Level-wise Segment Evaluation and Level-Crossing Reuse**. Based on the level decomposition $\{\mathcal{S}_1, \mathcal{S}_2, \ldots\}$ in Section 3.1, we evaluate segments in a level-by-level order. Segments within the same level are data-independent and evaluated in parallel. For each segment $\boldsymbol{seg} \in \mathcal{S}_k$, we collect all input slews propagated from level-$(k-1)$ fan-ins. After deduplication and rise/fall separation, we obtain a descending-ordered list $\{s_1, s_2, \ldots, s_M\}$ with $s_1 > s_2 > \cdots > s_M$.

We first compute segment delay $d_1$ at the largest slew $s_1$ using the NLDM + net model from Section 3.2, and insert the result into *SegHashMap*, i.e., a two-level structure indexed by segment-ending arc and keyed in a red-black tree (slew map) by $s_1$. Then, for each smaller slew $s_m$ $(m = 2, \ldots, M)$, we: (1) locate the nearest larger computed slew $s_{\text{ref}}$ in slew map and retrieve its delay $d_{\text{ref}}$; (2) query the learned predictor from Section 3.3, $\hat{d}_m = \mathcal{F} * \text{pred}(e * \text{seg}, s_m)$; (3) compute the deviation $\varepsilon = |d_{\text{ref}} - \hat{d}_m|$; (4) if $\varepsilon \le \tau_{\text{seg}}$, reuse $d_{\text{ref}}$ for $s_m$, otherwise compute the delay at $s_m$ and insert it into the slew map. This descending-slew reuse policy preserves pessimism (reusing a larger slew's delay) while maximizing reuse by selecting the nearest dominating key instead of always comparing to the global maximum.

**Table 1: Benchmark statistics.**

| Benchmarks | #I/O | #Pins | #Cell Arcs | #Net Arcs |
|---|---|---|---|---|
| aes_core | 260/129 | 62281 | 81802 | 40639 |
| des_perf | 235/64 | 233685 | 307794 | 153572 |
| vga_lcd | 89/109 | 260146 | 344230 | 171323 |
| pca_bridge32_ispd | 162/201 | 84450 | 107202 | 53553 |
| cordic_ispd | 34/64 | 119362 | 155692 | 77495 |
| edit_dist_ispd | 2562/12 | 376509 | 495874 | 241185 |
| netcard_iccad | 1836/10 | 2725999 | 3534820 | 1765158 |
| leon2_iccad | 615/85 | 2384781 | 3180958 | 1589807 |
| leon3mp_iccad | 254/79 | 1961871 | 2625280 | 1312388 |

**Phase 3: Path-wise Timing Query and Accumulation**. Once all relevant segment entries are stored in the *SegHashMap*, we evaluate full path delays. Each critical path in $\mathcal{P}$ is decomposed into segments $\{\boldsymbol{seg}_1, \boldsymbol{seg}_2, \ldots, \boldsymbol{seg}_L\}$. For each segment $\boldsymbol{seg}_i$ and input slew $s_i$, we retrieve delay $d_i$ via nearest-key lookup in the slew map. Delays are accumulated sequentially, with each output slew propagated as the next input $s_{i+1}$ to ensure timing consistency across the path.

This framework ensures per-segment bounded error, pessimism-preserving reuse, and cumulative savings via shared timing segments. Together with level-wise parallelism, it delivers scalable, near-signoff timing evaluation with minimal redundant arc computations.

## 4 Experimental Results

Our PBA acceleration framework is implemented in C++, and the auxiliary slew-sensitive models are implemented using PyTorch. All training and runtime experiments are conducted on a 32-core Linux workstation with 128 GB memory and four NVIDIA Tesla V100 GPUs. Experiments are conducted on circuits selected from TAU15 contest benchmarks (unseen for our slew-sensitive model), using reference results produced by IBM Einstimer in static mode [15]. The detailed information of these circuits are listed in Table 1. To evaluate the effectiveness of our framework, we use OpenTimer v2 [1] as the baseline and compare three aspects: (1) the total number of arcs actually calculated during analyzing critical paths. (2) The analysis error in path slack, where the exact values are obtained by OpenTimer v2. Since $R^2$ score approaches 1 when the number of paths is large, we adopt the maximum absolute error (MAXE) as metric. (3) Analysis runtime. For each circuit, the total error budget $\tau_{\text{total}}$ is set to $0.1 \times N$, where $N$ is the number of paths under analysis.

### 4.1 Performance Evaluation

The performance evaluation is detailed as follows.

**Reuse effectiveness**. Table 2 reports the number of timing arcs actually calculated by different methods when analyzing $10^3$ to $10^6$ timing paths. Compared to OpenTimer v2, our framework markedly reduces the number of timing arcs that must be calculated. As the number of analyzed paths increases, the proportion of arc calculations saved by our framework climbs from an average of 90.1% for $10^3$ paths to an impressive 98.8% for $10^6$ paths. This trend shows that as more paths are considered, the growing overlap among them amplifies redundant arc timing analysis, which is effectively capitalized by our framework to reduce computational cost.

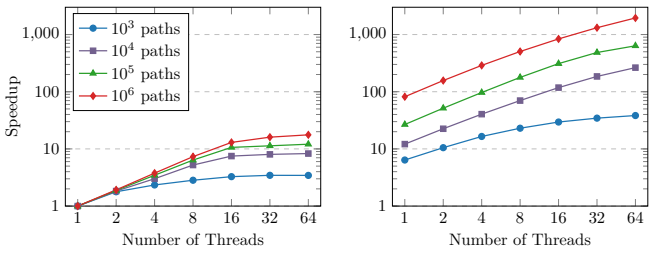
**Analysis efficiency**. Table 3 presents the accuracy and runtime efficiency of our framework. It achieves a significant speedup over OpenTimer v2 across all benchmark circuits, from an average of 13.25× on 1K paths to 195× on 5M paths. This acceleration trend is correlated with the arc calculation savings reported in Table 2. For each benchmark circuit, the speedup becomes more pronounced as the proportion of saved arc calculations increases, demonstrating the

**Table 2: The total number of timing arcs actually calculated when analyzing different numbers of paths. The values in parentheses indicate the proportion of arc calculations saved by ours compared with OpenTimer v2 [1].**

| Circuit | #Calc. Arcs ($10^3$ Paths) | | #Calc. Arcs ($10^4$ Paths) | | #Calc. Arcs ($10^5$ Paths) | | #Calc. Arcs ($10^6$ Paths) | |
|---|---|---|---|---|---|---|---|---|
| | OpenTimer | Ours | OpenTimer | Ours | OpenTimer | Ours | OpenTimer | Ours |
| aes_core | 40.3K | 2.51K (-93.8%) | 410K | 12.5K (-97.0%) | 4.04M | 85.9K (-97.9%) | 38.9M | 441K (-98.9%) |
| des_perf | 29.0K | 3.62K (-87.5%) | 283K | 30.7K (-89.2%) | 2.82M | 229K (-91.9%) | 28.1M | 1.53M (-94.6%) |
| vga_lcd | 22.4K | 6.95K (-69.0%) | 263K | 48.8K (-81.5%) | 2.86M | 285K (-90.0%) | 30.3M | 879K (-97.1%) |
| pci_bridge32_ispd | 30.5K | 5.49K (-82.0%) | 332K | 31.5K (-90.5%) | 3.67M | 155K (-95.8%) | 42.6M | 510K (-98.8%) |
| cordic_ispd | 130K | 1.09K (-99.2%) | 1.25M | 4.86K (-99.6%) | 12.2M | 11.8K (-99.9%) | 117M | 86.3K (-99.9%) |
| edit_dist_ispd | 132K | 16.6K (-87.4%) | 1.29M | 97.5K (-92.4%) | 12.7M | 446K (-96.5%) | 131M | 1.46M (-98.9%) |
| netcard_iccad | 32.2K | 1.40K (-95.7%) | 319K | 10.7K (-96.6%) | 3.20M | 85.3K (-97.3%) | 30.7M | 215K (-99.3%) |
| leon2_iccad | 46.3K | 3.95K (-91.5%) | 445K | 19.4K (-95.6%) | 4.31M | 106K (-97.5%) | 44.9M | 288K (-99.4%) |
| leon3mp_iccad | 26.4K | 6.70K (-74.6%) | 259K | 28.1K (-89.2%) | 3.76M | 136K (-96.4%) | 35.2M | 359K (-99.0%) |
| Average | 54.3K | **5.37K (-90.1%)** | 539K | **31.6K (-94.1%)** | 5.51M | **171K (-96.9%)** | 55.4M | **641K (-98.8%)** |

**Table 3: The accuracy of our framework and its speedup over OpenTimer v2 [1]. Accuracy is evaluated against OpenTimer v2 as the golden reference, and both methods are executed with 64-thread parallelism.**

| Circuit | $1 \times 10^3$ Paths | | $1 \times 10^4$ Paths | | $1 \times 10^5$ Paths | | $1 \times 10^6$ Paths | | $5 \times 10^6$ Paths | |
|---|---|---|---|---|---|---|---|---|---|---|
| | MAXE (ps) | Speedup | MAXE (ps) | Speedup | MAXE (ps) | Speedup | MAXE (ps) | Speedup | MAXE (ps) | Speedup |
| aes_core | 0.0431 | 10.53× | 0.0507 | 26.68× | 0.0542 | 35.52× | 0.0763 | 52.17× | 0.0894 | 218.37× |
| des_perf | 0.1780 | 9.25× | 0.2373 | 11.57× | 0.4880 | 9.14× | 0.5270 | 5.29× | 0.6741 | 61.54× |
| vga_lcd | 0.0000 | 3.96× | 0.0000 | 4.64× | 0.0481 | 7.62× | 0.0738 | 21.91× | 0.0862 | 121.10× |
| pci_bridge32_ispd | 0.0473 | 5.99× | 0.8051 | 11.46× | 1.6100 | 25.13× | 1.8300 | 57.15× | 2.3600 | 186.93× |
| cordic_ispd | 0.1839 | 41.22× | 0.1830 | 82.39× | 0.2501 | 104.41× | 0.3767 | 115.83× | 0.4686 | 188.62× |
| edit_dist_ispd | 0.1481 | 8.43× | 0.2256 | 14.51× | 0.3046 | 31.88× | 0.3392 | 67.54× | 0.3971 | 273.25× |
| netcard_iccad | 0.1847 | 20.17× | 0.2571 | 37.65× | 0.3673 | 105.72× | 0.3890 | 128.36× | 0.3895 | 206.72× |
| leon2_iccad | 0.1280 | 11.14× | 0.2190 | 31.82× | 0.3110 | 52.59× | 0.5170 | 110.61× | 0.9420 | 212.93× |
| leon3mp_iccad | 0.0212 | 8.60× | 0.0369 | 16.43× | 0.0435 | 57.55× | 0.0961 | 73.69× | 0.1622 | 285.50× |
| Average | 0.1038 | **13.25×** | 0.2239 | **26.35×** | 0.3863 | **47.73×** | 0.4695 | **70.28×** | 0.6188 | **195.00×** |




(a) Opentimer v2 (b) Our framework

**Figure 4: Parallel scalability on `leon2_iccad`. Baseline for speedup is the single-thread runtime of OpenTimer v2 [1].**

effectiveness of reuse in improving PBA efficiency. Meanwhile, the above performance gain is achieved while maintaining high accuracy. The MAE in path slack is kept to a minimum, remaining below *1ps* in the vast majority of circuits and averaging a mere *0.6188ps* at the largest analysis scale. This indicates that the errors from timing reuse are effectively controlled under the guidance of slew-sensitive model.

**Parallelism evaluation**. We evaluate the parallel scalability of our framework on one of the largest circuits `leon2_iccad`, with results shown in Fig. 4. Unlike OpenTimer v2, whose speedup saturates quickly (around 16 threads), our framework exhibits consistent scaling with increasing thread counts. This stems from our effort to minimize the serial bottleneck of PBA. Although inherent serial components prevent ideal speedup, the benefits of additional threads grow with larger workloads (i.e., more timing paths), making this especially promising for large-scale timing analysis.

**Comparison with GPU Works**. Due to the lack of publicly available GPU-accelerated PBA implementations, a direct head-to-head comparison is infeasible. However, we draw an indirect comparison with GPU-PBA [4], which reports speedup results on the same large-scale designs using OpenTimer v2 [1] with 32 threads as the baseline—matching our experimental setup. On three of the largest benchmark cases (all with over 1M cells), namely `netcard`, `leon2`, and `leon3mp`, GPU-PBA [4] achieves speedups of 50.65×, 24.32×, and 46.10×, respectively. In contrast, our proposed work attains speedups of 57.82×, 64.71×, and 111.16× on the same benchmarks. These findings highlight the effectiveness of our proposed reuse-centric methodology in fundamentally reducing PBA computational complexity. Even compared to state-of-the-art GPU-based acceleration, our method demonstrates superior performance while maintaining timing sign-off fidelity.

**Potential for GPU Acceleration**. The inherent data parallelism of our framework makes it well-suited for GPU acceleration. The level-wise segment timing calculation aligns naturally with GPU's level-synchronized execution model, where each segment level is processed by a dedicated kernel (e.g., the kernel design in [9]) without inter-level dependencies. Meanwhile, all intermediate results (e.g., *SegHashMap* entries) can be stored in global memory throughout execution, avoiding frequent and costly PCIe transfers. The CPU merely orchestrates the kernel launch sequence and receives a compact subset of final timing data for aggregation. This architecture minimizes host-device communication and highlights GPU acceleration as a highly promising direction for future extensions of our framework.

## 5 Conclusion

We proposed a segment-level timing arc reuse framework that accelerates Path-Based Timing Analysis (PBA) by reducing its inherent computational complexity. By decomposing timing paths into multi-fanin-bounded segments and reusing delay computations across identical segments with similar input slews. Our method avoids redundant computations while rigorously preserving pessimism and accuracy. A dual-indexed *SegHashMap* and a slew-sensitive timing model enable safe, efficient reuse across paths. Experiments show that our method achieves up to **195× speedup** over state-of-the-art parallel CPU PBA and **2–3×** gains over recent GPU-based methods, which highlights that complexity reduction is a new path toward scalable sign-off.

## References


[1] T.-W. Huang, G. Guo, C.-X. Lin, and M. D. Wong, "Opentimer v2: A new parallel incremental timing analysis engine," *IEEE transactions on computer-aided design of integrated circuits and systems*, vol. 40, no. 4, pp. 776–789, 2020.

[2] T.-W. Huang and M. D. Wong, "Opentimer: A high-performance timing analysis tool," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.

[3] A. B. Kahng, U. Mallappa, and L. Saul, "Using machine learning to predict path-based slack from graph-based timing analysis," in *2018 IEEE 36th International Conference on Computer Design (ICCD)*, 2018.

[4] G. Guo, T.-W. Huang, Y. Lin, Z. Guo, S. Yellapragada, and M. D. Wong, "A gpu-accelerated framework for path-based timing analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 11, pp. 4219–4232, 2023.

[5] Y. Ye, T. Chen, Y. Gao, H. Yan, B. Yu, and L. Shi, "Graph-learning-driven path-based timing analysis results predictor from graph-based timing analysis," in *Proceedings of the 28th Asia and South Pacific Design Automation Conference*, 2023.

[6] G. Guo, T.-W. Huang, Y. Lin, and M. Wong, "Gpu-accelerated path-based timing analysis," in *2021 58th ACM/IEEE Design Automation Conference (DAC)*, 2021.

[7] Z. Guo, T.-W. Huang, and Y. Lin, "Gpu-accelerated static timing analysis," in *Proceedings of the 39th international conference on computer-aided design*, 2020.

[8] G. Guo, T.-W. Huang, and M. Wong, "Fast sta graph partitioning framework for multi-gpu acceleration," in *2023 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2023.

[9] Z. Guo, T.-W. Huang, and Y. Lin, "Accelerating static timing analysis using cpu–gpu heterogeneous parallelism," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 42, no. 12, pp. 4973–4984, 2023.

[10] Z. Guo, Z. Zhang, W. Li, T.-W. Huang, X. Shi, Y. Du, Y. Lin, R. Wang, and R. Huang, "Heteroexcept: A cpu-gpu heterogeneous algorithm to accelerate exception-aware static timing analysis," in *Proceedings of the 43rd IEEE/ACM International Conference on Computer-Aided Design*, 2024.

[11] S. Lin, G. Guo, T.-W. Huang, W. Sheng, E. Young, and M. Wong, "Gcs-timer: Gpu-accelerated current source model based static timing analysis," in *Proceedings of the 61st ACM/IEEE Design Automation Conference*, 2024.

[12] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, "Graph attention networks," in *International Conference on Learning Representations (ICLR)*, 2018, arXiv:1710.10903.

[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems (NeurIPS)*, 2017.

[14] P. J. Huber, "Robust estimation of a location parameter," in *Breakthroughs in statistics: Methodology and distribution*. Springer, 1992, pp. 492–518.

[15] J. Hu, G. Schaeffer, and V. Garg, "Tau 2015 contest on incremental timing analysis," in *2015 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, 2015.