# Activity-Aware Partitioning for Effective Multi-Threaded Event-Driven RTL Simulation

Kexing Zhou
School of Integrated Circuits
Peking University
Beijing, China
zhoukexing@pku.edu.cn

Youwei Zhuo
School of Integrated Circuits
Peking University
Beijing, China
youwei@pku.edu.cn

Yibo Lin
School of Integrated Circuits
Peking University
Beijing, China
yibolin@pku.edu.cn

Weikang Qian
Shanghai Jiao Tong University
Shanghai, China
qianwk@sjtu.edu.cn

Pengpeng Ren
Shanghai Jiao Tong University
Shanghai, China
renpp@sjtu.edu.cn

Yun Liang*
School of Integrated Circuits
Peking University
Beijing, China
ericlyun@pku.edu.cn

## Abstract

Register-transfer-level (RTL) simulation is commonly accelerated through multi-threading or event-driven techniques. Ideally, combining them should achieve both high utilization and high efficiency. However, severe load imbalance arises: in event-driven execution, circuit regions exhibit highly uneven activity, leaving many threads idle with very low utilization. Existing multi-threaded RTL partitioning models often optimize for full-cycle execution while ignoring the dynamic activity variation that dominates event-driven workloads.

We present P$^2$SIM, a multi-threaded event-driven RTL simulator that incorporates runtime activity information into its partitioning model. Our model augments hypergraph weights with measured activity frequency and correlation, which is efficiently estimated through a lightweight MinHash profiler. By aligning thread assignment with true activity patterns, P$^2$SIM substantially reduces idle time and improves utilization. Across experiments on large benchmarks, P$^2$SIM achieves simulation speeds of 200-400 kHz and up to 8.9× speedup over Verilator.

**ACM Reference Format:**
Kexing Zhou, Youwei Zhuo, Yibo Lin, Weikang Qian, Pengpeng Ren, and Yun Liang*. 2026. Activity-Aware Partitioning for Effective Multi-Threaded Event-Driven RTL Simulation. In *63rd ACM/IEEE Design Automation Conference (DAC '26), July 26–29, 2026, Long Beach, CA, USA.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3770743.3803973

## 1 Introduction

Register-transfer-level (RTL) simulation is a fundamental step in pre-silicon verification [17]. It provides cycle-accurate functional validation before synthesis and prototyping, but it also dominates verification time and computational cost. To speedup RTL simulation, designers rely on a spectrum of acceleration platforms, including hardware emulators [5, 9, 10, 19, 24], FPGA prototypes [14, 23], GPU-based simulators [12, 15, 16], and CPU-based software simulators [4, 6, 18, 20–22, 26]. Despite the impressive performance of specialized hardware, CPU-based simulators remain widely used because they offer short compile time and easy integration into verification flows.

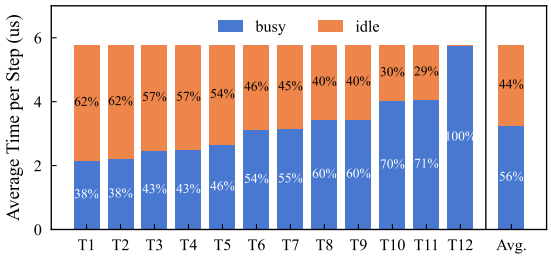



**Figure 1: Thread-level utilization breakdown of a 12-thread event-driven simulation (booting Linux on LargeBoom 1C).**

To accelerate CPU-based software RTL simulation, two orthogonal directions have been widely explored: *multi-threading* and *event-driven* simulation. Multi-threaded simulators, such as RepCut [21] and Verilator [18], exploit spatial parallelism by decomposing the circuit into structural partitions that can be evaluated concurrently on multiple CPU cores. Event-driven simulators, including ESSENT [4] and GSIM [6], improve temporal efficiency by skipping inactive logic and evaluating only blocks affected by signal changes. Ideally, combining them should achieve both high parallelism and high efficiency, with each thread processing only the active portion of the design.

However, this combination exposes a fundamental scalability bottleneck that must be addressed. Event-driven execution exhibits strong *temporal sparsity* and *spatial skew*: in each cycle, only a small and rapidly changing subset of blocks is active, and activity tends to cluster within certain regions of the design. When such behavior is mapped onto a fixed thread partition, a severe *thread-level load imbalance* problem arises. Fig. 1 illustrates this imbalance: more than half of the threads remain idle for over 45% of the total simulation time, and some spend less than 5% of their cycles in useful computation. This severe imbalance prevents multi-threaded

* Corresponding author.


*DAC '26, Long Beach, CA, USA*

ACM ISBN 979-8-4007-2254-7/2026/07
https://doi.org/10.1145/3770743.3803973

event-driven simulators from achieving good scaling on modern multi-core CPUs.

Existing partitioning methods are largely ineffective under such temporal sparsity because they assume uniform activity across regions. Full-cycle partitioners such as RepCut [21] model only static structure, which works well when every node is evaluated in each cycle, but fails when activity varies significantly between regions. In event-driven simulation, accurate load balancing requires not only estimating how frequently each block becomes active but also understanding *which blocks tend to be active together*. Two regions with similar activation patterns should reside in the same partition to minimize cross-thread synchronization.

To address the thread load imbalance problem, this paper presents **P$^2$SIM**, a CPU-based framework for parallel cycle-accurate event-driven RTL simulation. P$^2$SIM organizes the simulation into a *profiling–partitioning–execution* pipeline. It first runs a lightweight profiling round to record block activity statistics and correlations, building compact MinHash-based signatures that approximate which blocks are co-active without materializing a wave trace. This signature allows P$^2$SIM to model higher-order activity correlations, such as the cost of cross-thread activation of two nearby blocks and long-range dependency contributions in a signal path. P$^2$SIM formulates thread assignment as a hypergraph partitioning problem augmented with activity information. To reduce the synchronization cost, P$^2$SIM converts cross-thread communication into a cache-line forwarding task embedded into the event-driven scheduling pipeline.

Within this framework, we make the following contributions:

- We design *P$^2$SIM*, a unified framework for CPU-based multi-threaded cycle-accurate event-driven RTL simulation that integrates profiling, activity-aware partitioning, and a synchronization model.
- We introduce a lightweight *MinHash-based activity profiler* that records block-level activity patterns and enables constant-time estimation of activity correlation between arbitrary blocks.
- We propose an *activity-aware hypergraph reweighting* model that incorporates runtime activity into the partitioning objective, guiding the partitioner to assign co-active blocks to the same thread and to balance the event-driven workload.

We evaluate P$^2$SIM on large-scale SoC designs, including multi-core Rocket [2] and BOOM [25] running full software stacks. Across these benchmarks, P$^2$SIM achieves 200–400 kHz simulation speed and delivers up to 8.9× speedup over multi-threaded Verilator [18]. These results show that a framework that explicitly models runtime activity and cross-thread communication is essential for scalable multi-threaded event-driven RTL simulation on CPUs.[1]

## 2 Background and Challenge

### 2.1 CPU-Based RTL Simulation

Most CPU-based RTL simulators internally lower the RTL netlist into a directed acyclic graph (DAG) or a network of basic blocks, where nodes represent combinational logic or state updates and edges represent signal dependencies. The central design choice

[1] P$^2$SIM is open-sourced at https://github.com/pku-liang/p2sim.

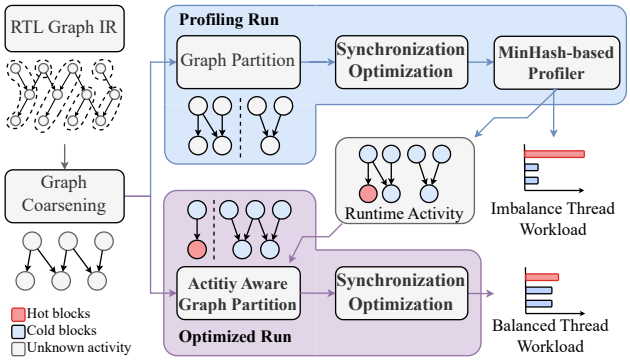



**Figure 2: Overview of the P$^2$SIM framework.**

is the *scheduling strategy*, i.e., how to determine which nodes to evaluate at each simulated cycle and in what order.

*Full-cycle scheduling.* Full-cycle simulators evaluate all nodes in a fixed topological order every cycle. Tools such as Verilator [18] generate optimized C++ code for this schedule with aggressive inlining and dataflow optimizations. Full-cycle schedulers are simple and predictable, making them well-suited to static analysis and multi-threaded partitioning. However, they fundamentally ignore temporal sparsity: even if only a small portion of the design is active, the entire DAG is still evaluated.

*Event-driven scheduling.* Event-driven simulators exploit temporal sparsity by evaluating only logic affected by signal changes. Modern event-driven CPU-based simulators, such as ESSENT [4] and GSIM [6], coarsen the DAG into larger *blocks* or *supernodes*. Simulation proceeds by activating blocks whose inputs have changed, evaluating them, and propagating events to successors. This approach can significantly reduce work per cycle for sparse workloads, but introduces overhead to track active blocks and manage the event queue. Existing event-driven CPU simulators are primarily single-threaded and do not address thread-level load balancing or synchronization.

*Multi-threading.* To leverage multi-core CPUs, several works have explored multi-threaded full-cycle simulation [18, 20–22, 26]. These simulators partition the DAG into regions evaluated concurrently, often using hypergraph partitioning with replication to reduce synchronization [21]. A global barrier or a levelized schedule synchronizes threads at each cycle. Because full-cycle simulation evaluates every node each cycle, the workload is relatively uniform and static structural partitioning can achieve good load balance. However, full-cycle multi-threaded simulators do not exploit temporal sparsity.

Overall, existing CPU-based simulators tend to optimize either temporal sparsity (single-threaded event-driven) or spatial parallelism (multi-threaded full-cycle), but not both simultaneously.

### 2.2 Challenges

Combining multi-threading with event-driven scheduling on CPUs is attractive but challenging. Event-driven execution introduces two key properties that complicate static partitioning:

- *Temporal sparsity*: in each cycle, only a small subset of blocks becomes active, and the set of active blocks changes over time.

- *Spatial skew and correlation*: activity tends to cluster within certain regions of the design, and groups of blocks often fire together in correlated bursts.

When the design is partitioned across threads without modeling these properties, two dominant costs arise. First, *load imbalance*: threads mapped to mostly inactive regions remain idle, while threads responsible for active regions become bottlenecks. Second, *cross-thread synchronization*: if blocks that frequently trigger each other are assigned to different threads, cross-thread activations become frequent, causing cache coherence overhead.

Existing multi-threaded partitioners for RTL simulation [20–22, 26] are designed for full-cycle workloads and typically assume that all nodes are evaluated every cycle. They construct hypergraphs that model structural connectivity and replication cost, but they do not incorporate runtime activity frequency or correlation. Conversely, existing event-driven simulators [4, 6] focus on single-threaded scheduling and do not provide mechanisms for activity-aware thread assignment or scalable cross-thread communication.

These limitations motivate a CPU-based framework that: (i) observes runtime activity patterns through a lightweight profiling mode; (ii) embeds this activity information into a hypergraph partitioning model to balance true event-driven workloads; and (iii) provides a synchronization mechanism tailored to event-driven cross-thread communication. P$^2$SIM is designed to fill this gap.

## 3 Overview

Fig. 2 shows the overall workflow of P$^2$SIM. P$^2$SIM accepts RTL input, translates it into a graph-based IR for optimization, and finally generates C++ code. On top of the compilation flow, we execute a short *profiling run* (blue region) and the *optimized run* (purple region).

In the profiling run, a baseline graph partition assigns blocks to threads without activity information. Then P$^2$SIM performs synchronization optimization on the partitioned design. Before code generation, a MinHash-based profiler is embedded to record per-block activation signatures. The profiling results reveal both the runtime activity and the correlation structure of the design. P$^2$SIM then uses the collected activity information to drive an *optimized run*. An activity-aware partitioner reweights the coarsened graph using activation frequency and correlation and produces a new thread assignment. This profiling-guided partition produces a more balanced distribution of active work across threads, enabling scalable performance on multi-core CPUs.

Event-driven optimization in P$^2$SIM is split into two stages. The first stage is the graph coarsening step in Fig. 2, where compile-time known co-active logics are merged to reduce the graph size. In the second stage, after graph partitioning, a more aggressive graph coarsening is applied. This two-stage event-driven optimization is critical for multi-threading because graph coarsening complicates data dependencies and can break parallelism. At a high level, P$^2$SIM adds three components to a standard cycle-accurate event-driven compilation pipeline: (i) a MinHash-based profiler; (ii) an activity-aware graph partition model; (iii) a runtime using cache-line tasks for cross-thread communication. The following subsections describe these components and their interactions.

## 4 Method

In this section, we explain the core profiling and optimization techniques used in P$^2$SIM.

### 4.1 MinHash-based Activity Profiler

In event-driven simulation, we would like to preserve as much information as possible about each block's temporal activity while keeping the profiling overhead low. Simple scalar statistics, such as the number of active cycles, lose most of the temporal structure that determines how blocks correlate with each other, whereas recording the exact active set for every block is prohibitively expensive. To strike a balance, P$^2$SIM encodes each block's activity pattern using a compact MinHash signature that approximates its activation set at low space and time cost.

*Activity representation.* We represent the activity of each block $i$ as the set of cycles in which it becomes active in Eqn. 1. A naive way of recording $A_i$ would require storing all activity cycles for all blocks, leading to an $O(N \times C)$ memory footprint for $N$ blocks and $C$ cycles, easily reaching gigabytes or even tens of gigabytes.

$$A_i = \{\, c \mid \text{block } i \text{ is active at cycle } c \,\}. \tag{1}$$

*MinHash encoding.* To compactly capture activity patterns, P$^2$SIM employs a *MinHash*-based profiler. Given $k$ independent hash functions $\{h_1, h_2, \ldots, h_k\}$, we define the MinHash signature of block $i$ as Eqn. 2. This signature is fixed in size and does not grow with simulation time.

$$\text{MinHash}(A_i) = [H_{i,1}, H_{i,2}, \ldots, H_{i,k}] \tag{2}$$

$$H_{i,k} = \min_{c \in A_i} h_k(c). \tag{3}$$

MinHash provides an unbiased estimator of the Jaccard similarity between any two activity sets:

$$\text{Jaccard}(A_i, A_j) = \frac{|A_i \cap A_j|}{|A_i \cup A_j|} \approx \frac{1}{k} \sum_{t=1}^{k} [H_{i,t} = H_{j,t}]. \tag{4}$$

This allows us to estimate the similarity of activity patterns between arbitrary blocks in $O(1)$ time. During graph partitioning, this activity similarity serves as a weight adjustment factor, guiding the partitioner toward an activity-balanced graph division.

*Efficient implementation.* In the recording process, the cycle number $c$ serves directly as the input to the hash functions. Therefore, the hash computation can be replaced by generating pseudo-random values from a fast random number generator. As shown in Algorithm 1, for each active block, the profiler updates its MinHash vector by taking the element-wise minimum between the current value and the newly generated random vector (Line 8). This operation can be vectorized using only one or two AVX2 instructions, introducing negligible runtime overhead even for millions of cycles.

### 4.2 Activity-Aware Partition Model

Inspired by RepCut [21], we use hypergraphs for cost modeling in multi-threaded task partitioning. In the hypergraph model, each node represents a sink node to be assigned to a thread, and each hyperedge connects multiple sink nodes to represent a shared resource. Fig. 3 demonstrates our model. It consists of four components: (1) node weight model; (2) replication cost; (3) communication cost,

**Algorithm 1** MinHash-based Activity Recording

**Require:** Number of blocks $N$, total cycles $C$, activity flags `active[N]`, signature length $K$
**Ensure:** MinHash signatures `MH[N][K]`

```
Initialize MH[blk] ← [UINT_MAX; K] for all blk
rng ← PseudoRandom(seed=0)
for cycle ← 0 to C − 1 do
    Run event-driven simulation step to obtain active[N]
    cycle_hash ← rng.next::<[u32; K]>()
    for blk ← 0 to N − 1 do
        if active[blk] is true then
            MH[blk][:] ← min(MH[blk][:], cycle_hash[:])
        end if
    end for
end for
```

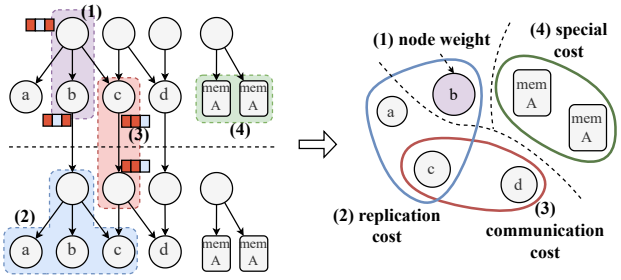



**Figure 3: Activity-Aware Partition Model**

and (4) special cost. The hypergraph partitioner divides this graph into multiple parts with roughly equal node weight for balanced thread workload, and minimizes the cost of cut edges for lower communication or replication cost.

*Node weight model.* The weight of the node represents the cost of evaluating it. We estimate the evaluation cost of each sink node as the sum of contributions from its non-sink predecessors. The contribution of a non-sink node $i$ to sink node $s$ is evaluated by its Jaccard similarity in Eqn. 5, where $P(i)$ is the profiled number of active cycles of node $i$, $\text{cost}(i)$ is its evaluation cost, and $A_i$ is its activity set.

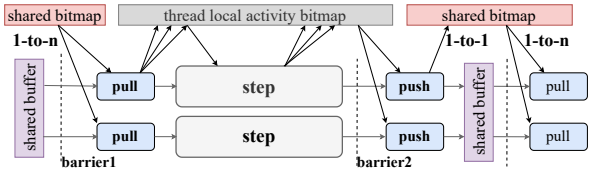
$$\text{Contrib}(i,s) = P(i) \cdot \text{cost}(i) \cdot \frac{\text{Jaccard}(A_i, A_s)}{\sum_{s' \in S} \text{Jaccard}(A_i, A_{s'})} \tag{5}$$

The total weight of a sink node $s$ is the sum of contributions from all of its predecessors, as shown in Eqn. 6.

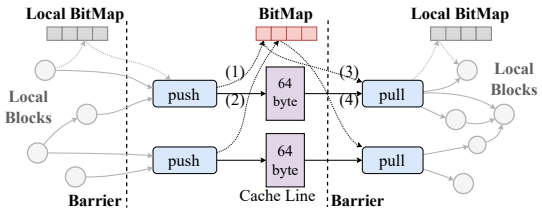
$$\text{Weight}(s) = \sum_{p \in \text{pred}(s)} \text{Contrib}(p,s), \tag{6}$$

These activity-aware weights guide the partitioner to keep strongly correlated and frequently active nodes in the same thread, improving balance and locality.

*Replication cost.* When a logic block is replicated across multiple threads, it increases both the instruction footprint (in I-cache) and the memory footprint (in D-cache). For each logic block $u$, we introduce a hyperedge that groups all sink nodes reachable from $u$, as defined in Eqn. 7, to model replication cost. The edge weight is defined in Eqn. 8, where $\text{InstCnt}(u)$ is the estimated number of instructions to evaluate block $u$, and $\lambda_{\text{inst}}$ is an adjustable parameter




**(a) Synchronization pipeline**




**(b) Cross thread communication**

**Figure 4: Synchronization Model of P²SIM**

to balance the replication cost.

$$HE_u = \{s \in \text{Sink} \mid \exists \text{ path } u \to s\} \tag{7}$$

$$\text{Weight}(HE_u) = \lambda_{\text{inst}} \cdot \text{InstCnt}(u) \tag{8}$$

For a state $x$, we build its hyperedge as in Eqn. 9 by combining the hyperedges of all blocks that access $x$ and weighting the edge by the size of $x$.

$$HE_x = \{s \in \text{Sink} \mid \exists \text{ block } u \text{ accessing } x \wedge \exists \text{ path } u \to s\} \tag{9}$$

*Communication cost.* We model cross-thread communication between a sink block and its consumers in the next cycle using hyperedges. For each sink node $s$ and each block $u$ that reads the values produced by $s$ in the next cycle, we create a binary hyperedge

$$HE_{s,u} = \{s\} \cup HE_u. \tag{10}$$

Its weight reflects both the amount of data transferred and how often $s$ and $u$ are simultaneously active in Eqn. 11, where $X_{s,u}$ is the set of state variables or signals that $u$ reads from $s$, and $A'_u$ is constructed in the profiling round by applying MinHash to the shifted set $\{c-1 \mid c \in A_u\}$.

$$\text{weight}(HE_{s,u}) = \lambda_{\text{comm}} \cdot \text{Jaccard}(A_s, A'_u) \cdot \sum_{x \in X_{s,u}} \text{Size}(x), \tag{11}$$

*Special cost.* A small number of RTL nodes, such as memories with multiple write ports, must preserve the order of updates to maintain correctness. For such nodes, P²SIM assigns their corresponding hyperedges a very large weight, preventing them from being distributed across threads. This constraint guarantees functional correctness while keeping the remainder of the graph freely partitionable.

## 4.3 Synchronization Method

Beyond partition quality, scalable event-driven parallelism also hinges on how cross-thread activations are communicated. Event-driven simulators propagate a large number of activation signals: every write to a state may trigger many dependent blocks. When these activations cross thread boundaries, they introduce fine-grained dependences and heavy cache-coherence traffic. Inspired by the Gluon graph processing framework [7], we merge fine-grained activations

**Table 1: Best Performance over Different Thread Counts (Verilator [18] is abbreviated as VLT)**

| Design | # | Benchmark | Simulation Speed (kHz) | | | | | Speedup over Verilator | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | | P²SIM(A) | P²SIM(U) | RepCut | GSIM | VLT | P²SIM(A) | P²SIM(U) | RepCut | GSIM | VLT |
| LargeBOOM | 1 | Linux | **233.4** | 203.5 | 152.7 | 66.9 | 43.5 | **5.37** | 4.68 | 3.51 | 1.54 | 1.00 |
| | 2 | Linux | **205.7** | 164.6 | 155.2 | 42.2 | 26.6 | **7.73** | 6.19 | 5.83 | 1.59 | 1.00 |
| | 4 | Linux | **175.8** | 132.0 | 156.5 | 14.9 | 19.8 | **8.88** | 6.67 | 7.90 | 0.75 | 1.00 |
| | 1 | CoreMark | **278.3** | 240.4 | 158.5 | 66.9 | 43.6 | **6.38** | 5.51 | 3.64 | 1.53 | 1.00 |
| | 2 | CoreMark | **229.2** | 177.7 | 156.0 | 46.7 | 27.1 | **8.46** | 6.56 | 5.76 | 1.72 | 1.00 |
| | 4 | CoreMark | **186.6** | 148.4 | 157.3 | 16.3 | 19.8 | **9.42** | 7.49 | 7.94 | 0.82 | 1.00 |
| MediumBOOM | 1 | Linux | **252.0** | 209.4 | 182.6 | 102.5 | 47.0 | **5.36** | 4.46 | 3.89 | 2.18 | 1.00 |
| | 2 | Linux | **240.2** | 202.7 | 169.8 | 70.9 | 38.2 | **6.29** | 5.31 | 4.45 | 1.86 | 1.00 |
| | 4 | Linux | **214.5** | 187.9 | 125.5 | 39.4 | 25.8 | **8.31** | 7.28 | 4.86 | 1.53 | 1.00 |
| | 1 | Coremark | **315.5** | 302.0 | 184.5 | 129.2 | 46.9 | **6.73** | 6.44 | 3.93 | 2.75 | 1.00 |
| | 2 | Coremark | **289.2** | 245.9 | 167.2 | 81.9 | 39.0 | **7.42** | 6.31 | 4.29 | 2.10 | 1.00 |
| | 4 | Coremark | **248.1** | 212.1 | 125.5 | 43.1 | 26.0 | **9.54** | 8.16 | 4.83 | 1.66 | 1.00 |
| Rocket | 1 | Linux | **437.7** | 384.0 | 281.9 | 263.3 | 99.3 | **4.41** | 3.87 | 2.84 | 2.65 | 1.00 |
| | 2 | Linux | **404.4** | 379.7 | 227.2 | 252.9 | 70.8 | **5.71** | 5.36 | 3.21 | 3.57 | 1.00 |
| | 4 | Linux | **376.7** | 344.3 | 182.3 | 211.1 | 47.5 | **7.93** | 7.25 | 3.84 | 4.44 | 1.00 |
| | 1 | CoreMark | **435.5** | 380.1 | 285.4 | 253.5 | 96.3 | **4.52** | 3.95 | 2.96 | 2.63 | 1.00 |
| | 2 | CoreMark | **407.5** | 379.7 | 230.6 | 244.8 | 73.7 | **5.53** | 5.15 | 3.13 | 3.32 | 1.00 |
| | 4 | CoreMark | **401.4** | 340.2 | 181.4 | 203.2 | 49.1 | **8.18** | 6.93 | 3.69 | 4.14 | 1.00 |

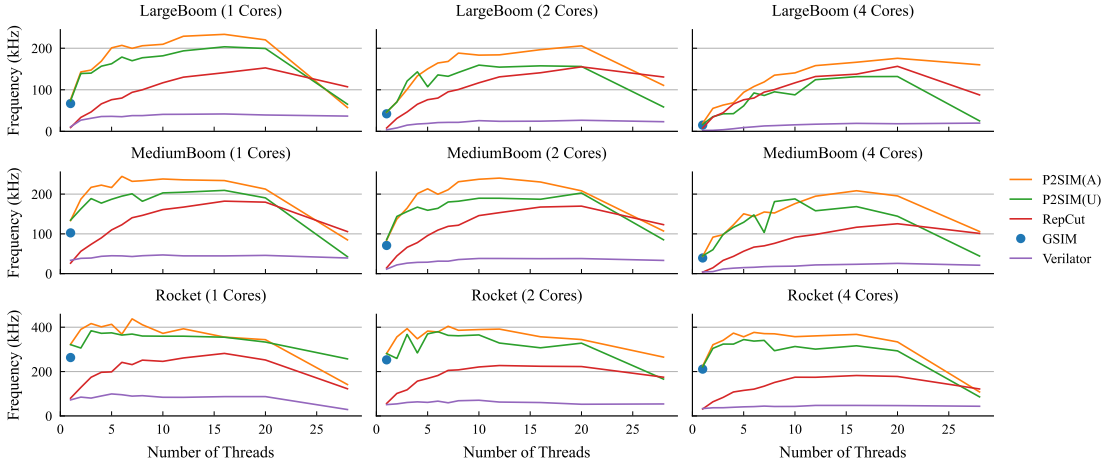



**Figure 5: Scaling Curve of Multi-threaded RTL Simulators (Booting Linux)**

into coarse-grained memory transfers to match the granularity of the cache hierarchy.

*Execution pipeline.* As shown in Fig. 4a, P²SIM divides each thread's simulation into five stages: *pull*, *step*, *barrier*$_1$, *push*, and *barrier*$_2$. Cache-line tasks are divided into push tasks and pull tasks in the corresponding stage. The push task groups local state into a cache line and writes it to the shared buffer, while the pull task performs the inverse operation. To avoid false sharing, the active-bit bitmap is partitioned into a private region and a shared region. The private bitmap is densely packed for fast intra-thread access, whereas the shared bitmap is padded to separate cache lines accessed by different threads.

*Pull–push tasks.* Each *push task* corresponds to at least one *pull task*, as illustrated in Fig. 4b. A push task executes after a group of sink nodes and writes updated values to the shared state and the shared activity bitmap. Its corresponding pull task reads these values in the next cycle and activates the related local blocks. Because the two tasks are separated by a barrier, mutual exclusion between read and write is guaranteed, and no atomic operation is required. This eliminates the cache-line thrashing caused by atomic updates and greatly reduces inter-thread activation latency.

# 5 Evaluation

## 5.1 Evaluation Setup

We implement P²SIM based on GSIM [6]. P²SIM takes FIRRTL [13] as input and generates optimized C++ code as output. We use MT-KaHyPar [11] as the hypergraph partitioner. The event-driven graph coarsening algorithm is a simple dynamic programming procedure, the same as in GSIM.

All experiments are conducted on a dual-socket workstation equipped with two Intel Xeon Platinum 8575C CPUs (48 cores, 2.4 GHz) and 512 GB of DDR5 memory, running Ubuntu 22.04 with `clang` 21.1.2. Unless otherwise stated, all reported runtimes are the average of three executions under identical conditions. We evaluate P²SIM using a diverse suite of open-source and industrial RTL designs, including BOOM [25] and RocketChip [2]. All Chisel [3] generated FIRRTL [13] sources are compiled into SystemVerilog using firtool 1.058 in CIRCT [1].

We compare against the state-of-the-art simulators Verilator [18], RepCut [21], and GSIM [4]. Each tool is configured with its recommended parallel settings. For fair comparison, all generated sources are compiled with clang++ with identical optimization flags (-O2) and run on the same CPU affinity layout. We choose CoreMark [8]

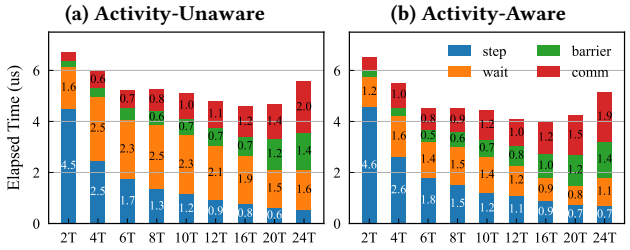



**Figure 6: Average Thread Utilization Breakdown on LargeBOOM 1C with different thread counts**

and Linux as software workloads. For each design and benchmark, we only profile the first 2% of total cycles.

## 5.2 Overall Performance

Table 1 summarizes the best performance achieved by each simulator across all benchmarks. For multi-threaded simulators, we report the highest frequency obtained by enumerating thread counts and selecting the optimal configuration. P$^2$SIM(A) denotes the full version with activity-aware partitioning; P$^2$SIM(U) uses activity-unaware static partition. Overall, P$^2$SIM achieves a 1.56× speedup over RepCut, a 3.09× speedup over GSIM, and a 7.01× speedup over Verilator on average. The activity-aware partition provides a consistent 20–30% advantage over the activity-unaware partition. For large designs such as `LargeBOOM 4C`, the gap between P$^2$SIM and RepCut narrows due to instruction-cache pressure. Addressing this bottleneck requires more structured partitioning techniques [22], which we leave for future work.

## 5.3 Scalability Comparison

Fig. 5 shows the scalability of P$^2$SIM compared with GSIM, RepCut, and Verilator. P$^2$SIM(A) consistently achieves higher peak speed and better scalability than the baselines. Compared to P$^2$SIM(U), the activity-aware strategy requires fewer threads to reach peak throughput, yet delivers roughly 20% higher peak performance, confirming that profiling-guided partitioning reduces load imbalance and synchronization overhead. Compared to RepCut and Verilator, P$^2$SIM starts from a much higher single-thread performance and reaches its peak faster, highlighting the advantage of event-driven scheduling.

## 5.4 Thread Utilization Breakdown

Fig. 6 shows the average thread-utilization breakdown on `LargeBoom` as the thread count increases. We report four components: `step` (logic evaluation), `comm` (push/pull tasks), `barrier` (synchronization), and `wait` (idle time). For both schemes, `step` decreases roughly inversely with thread count, confirming high parallelism. Under the activity-unaware partition (Fig. 6a), scalability is largely offset by growing idle time. The activity-aware partition (Fig. 6b) significantly reduces this imbalance: at 16 threads, idle time is roughly halved compared to the activity-unaware case, improving simulation speed from about 200 kHz to 250 kHz.

**Table 2: Ablation Study on Optimization Techniques**

| Design | # | Simulation Speed (kHz) | | | |
|---|---|---|---|---|---|
| | | NoOpt | CLTask | Rep Cost | Full |
| LargeBOOM | 1 | 110.2 | 216.6 | 217.4 | 233.4 |
| LargeBOOM | 2 | 66.4 | 156.8 | 159.7 | 205.7 |
| LargeBOOM | 4 | 56.4 | 121.1 | 131.5 | 175.8 |
| MediumBOOM | 1 | 114.1 | 224.2 | 241.0 | 252.0 |
| MediumBOOM | 2 | 102.8 | 200.5 | 232.5 | 240.2 |
| MediumBOOM | 4 | 66.2 | 170.8 | 182.6 | 214.5 |
| Rocket | 1 | 239.8 | 378.2 | 423.8 | 437.7 |
| Rocket | 2 | 249.0 | 344.6 | 363.7 | 404.4 |
| Rocket | 4 | 144.9 | 320.2 | 345.6 | 376.7 |

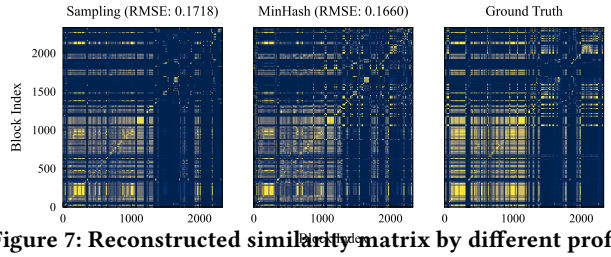



**Figure 7: Reconstructed similarity matrix by different profiling methods (MediumBoom 1C, Booting Linux)**

## 5.5 Ablation Study

*Performance Breakdown.* Table 2 shows the contribution of each optimization. Starting from the baseline `NoOpt`, adding cache-line tasks (`CLTask`) yields approximately 2× better performance, confirming that coarse-grained cache-line communication effectively replaces fine-grained activations. The replication-cost model (`Rep Cost`) further improves performance by 8%. The `Full` configuration consistently delivers the highest speed. Overall, each component provides a meaningful benefit and their combination is essential for achieving peak performance.

*Profiling Accuracy.* We evaluate MinHash accuracy against a sampling-based baseline. Fig. 7 compares the reconstructed Jaccard similarity matrices. The baseline records activity once every 256 cycles, producing a 256-bit signature. Both methods profile 65,536 cycles of MediumBOOM. The sampling approach preserves coarse trends but misses bursty behavior. MinHash provides a much finer approximation, yielding substantially lower RMSE and preserving both high- and low-similarity regions more faithfully.

## 6 Conclusion

This work presents **P$^2$SIM**, a profiling-guided multi-threaded event-driven RTL simulator that integrates runtime activity modeling into partitioning. By recording lightweight MinHash-based activity profiles and reweighting the hypergraph accordingly, P$^2$SIM achieves balanced workload distribution and high thread utilization. Experiments on large-scale SoC designs demonstrate up to 8.9× speedup over Verilator and 1.56× over RepCut with minimal profiling overhead. The results highlight the effectiveness of incorporating dynamic activity information into RTL simulation.

## Acknowledgments

This work was supported in part by the National Science Foundation of China (Grant No. T2293700 and T2293701).

## References


[1] [n. d.]. Circuit IR Compilers and Tools. https://circt.llvm.org/

[2] Krste Asanović, Rimas Avizienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Ben Keller, Donggyu Kim, and John Koenig. 2016. *The Rocket Chip Generator.* Technical Report.

[3] Jonathan Bachrach, Huy Vo, Brian Richards, Yunsup Lee, Andrew Waterman, Rimas Avižienis, John Wawrzynek, and Krste Asanović. 2012. Chisel: constructing hardware in a Scala embedded language. In *Proceedings of the 49th Annual Design Automation Conference*. ACM, San Francisco California, 1216–1225. doi:10.1145/2228360.2228584

[4] Scott Beamer and David Donofrio. 2020. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *2020 57th ACM/IEEE Design Automation Conference (DAC)*. IEEE, San Francisco, CA, USA, 1–6. doi:10.1109/DAC18072.2020.9218632

[5] Cadence. [n. d.]. Palladium Emulation. https://www.cadence.com/en_US/home/tools/system-design-and-verification/emulation-and-prototyping/palladium.html

[6] Lu Chen, Dingyi Zhao, Zihao Yu, Ninghui Sun, and Yungang Bao. 2025. GSIM: Accelerating RTL Simulation for Large-Scale Designs. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, Moscone West, United States, 1–7. doi:10.1109/DAC63849.2025.11133142

[7] Roshan Dathathri, Gurbinder Gill, Loc Hoang, Hoang-Vu Dang, Alex Brooks, Nikoli Dryden, Marc Snir, and Keshav Pingali. 2018. Gluon: a communication-optimizing substrate for distributed heterogeneous graph analytics. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM, Philadelphia PA USA, 752–768. doi:10.1145/3192366.3192404

[8] EEMBC. [n. d.]. CoreMark: An EEMBC Benchmark. https://www.eembc.org/coremark/

[9] Mahyar Emami, Thomas Bourgeat, and James R. Larus. 2025. Parendi : Thousand-Way Parallel RTL Simulation. In *Proceedings of the 30th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 2*. ACM, Rotterdam Netherlands, 783–797. doi:10.1145/3676641.3716010

[10] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 2023. Manticore: Hardware-Accelerated RTL Simulation with Static Bulk-Synchronous Parallelism. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ACM, Vancouver BC Canada, 219–237. doi:10.1145/3623278.3624750

[11] Lars Gottesbüren, Tobias Heuer, Nikolai Maas, Peter Sanders, and Sebastian Schlag. 2024. Scalable High-Quality Hypergraph Partitioning. *ACM Trans. Algorithms* 20, 1 (Jan. 2024), 9:1–9:54. doi:10.1145/3626527

[12] Zizheng Guo, Yanqing Zhang, Runsheng Wang, Yibo Lin, and Haoxing Ren. 2025. GEM: GPU-Accelerated Emulator-Inspired RTL Simulation. In *2025 62nd ACM/IEEE Design Automation Conference (DAC)*. IEEE, Moscone West, United States, 1–7. doi:10.1109/DAC63849.2025.11132713

[13] Adam Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. 2017. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In *2017 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, Irvine, CA, 209–216. doi:10.1109/ICCAD.2017.8203780

[14] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanovic. 2018. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. In *2018 ACM/IEEE 45th Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Los Angeles, CA, 29–42. doi:10.1109/ISCA.2018.00014

[15] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2022. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *Proceedings of the 51st International Conference on Parallel Processing*. ACM, Bordeaux France, 1–12. doi:10.1145/3545008.3545091

[16] Hao Qian and Yangdong Deng. 2011. Accelerating RTL simulation with GPUs. In *2011 IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. IEEE, NW Washington, DCUnited States, 687–693. doi:10.1109/ICCAD.2011.6105404 ISSN: 1558-2434.

[17] Louis Scheffer, Luciano Lavagno, and Grant Martin. 2006. *EDA for IC System Design, Verification, and Testing (Electronic Design Automation for Integrated Circuits Handbook)*. CRC Press, Inc., USA.

[18] Wilson Snyder. 2018. Verilator 4.0: Open Simulation Goes Multithreaded.

[19] Synopsys. [n. d.]. ZeBu Emulation Systems. https://www.synopsys.com/verification/emulation-prototyping/emulation.html

[20] Jie Tong, Zhengxiong Li, Umit Yusuf Ogras, and Tsung-Wei Huang. 2025. A Scalable Code Generation Flow for Heterogeneous Parallel RTL Simulation using MLIR. In *2025 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, Dresden, Germany, 1–9. doi:10.1109/HPEC67600.2025.11196471 ISSN: 2643-1971.

[21] Haoyuan Wang and Scott Beamer. 2023. RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 3*. ACM, Vancouver BC Canada, 572–585. doi:10.1145/3582016.3582034

[22] Haoyuan Wang, Thomas Nijssen, and Scott Beamer. 2024. Don't Repeat Yourself! Coarse-Grained Circuit Deduplication to Accelerate RTL Simulation. In *Proceedings of the 29th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 4*. ACM, Hilton La Jolla Torrey Pines La Jolla CA USA, 79–93. doi:10.1145/3622781.3674184

[23] Joonho Whangbo, Edwin Lim, Chengyi Lux Zhang, Kevin Anderson, Abraham Gonzalez, Raghav Gupta, Nivedha Krishnakumar, Sagar Karandikar, Borivoje Nikolić, Yakun Sophia Shao, and Krste Asanović. 2024. FireAxe: Partitioned FPGA-Accelerated Simulation of Large-Scale RTL Designs. In *2024 ACM/IEEE 51st Annual International Symposium on Computer Architecture (ISCA)*. IEEE, Buenos Aires, Argentina, 501–515. doi:10.1109/ISCA59077.2024.00044

[24] Ruifan Xu, Jin Luo, Yawen Zhang, Yibo Lin, Runsheng Wang, Ru Huang, and Yun Liang. 2024. Hestia: An Efficient Cross-Level Debugger for High-Level Synthesis. In *Proceedings of the 2024 57th IEEE/ACM International Symposium on Microarchitecture*. IEEE Press, 765–779. doi:10.1109/MICRO61859.2024.00062

[25] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. 2020. *SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine.* Technical Report.

[26] Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru Huang. 2023. Khronos: Fusing Memory Access for Improved Hardware RTL Simulation. In *56th Annual IEEE/ACM International Symposium on Microarchitecture*. ACM, Toronto ON Canada, 180–193. doi:10.1145/3613424.3614301