RepCut: Superlinear Parallel RTL Simulation with Replication-Aided Partitioning

Haoyuan Wang
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
hwang208@ucsc.edu

Scott Beamer
University of California, Santa Cruz
Santa Cruz, CA 95064, USA
sbeamer@ucsc.edu

ABSTRACT

Register transfer level (RTL) simulation is an invaluable tool for developing, debugging, verifying, and validating hardware designs. Despite the parallel nature of hardware, existing parallel RTL simulators yield speedups unattractive for practical application due to high communication and synchronization costs incurred by typical circuit topologies.

We present RepCut, a novel parallel RTL simulation methodology. RepCut is enabled by our replication-aided partitioning approach that cuts the circuit into balanced partitions with small overlaps. By replicating the overlaps, RepCut eliminates problematic data dependences between partitions and significantly reduces expensive synchronization overhead between parallel threads. RepCut outperforms state-of-the-art simulators, and when simulating a large system-on-chip with multiple out-of-order cores, it achieves a 27.10× speedup (superlinear) using 24 threads with only a 3.81% replication cost.

CCS CONCEPTS

• **Hardware → Hardware description languages and compilation**; • **Computing methodologies → Massively parallel and high-performance simulations**.

KEYWORDS

RTL simulation, parallel simulation, full-cycle simulation, replication-aided partitioning

ACM Reference Format:
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 (ASPLOS '23), March 25–29, 2023, Vancouver, BC, Canada.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3582016.3582034

1 INTRODUCTION

The time-consuming nature of RTL simulation is a burdensome problem for large-scale system-on-chip development, especially for verification signoff. Developers need to create multidimensional tests in RTL simulation with various input stimuli to cover use-case scenarios for design space exploration. Unfortunately, that slow speed of RTL simulation can often be the bottleneck [31], particularly when the complexity of an IC design reaches the point where the number of tests necessary to achieve full coverage cannot be completed within the development schedule. Despite the parallelism intrinsic to digital ICs, parallelizing individual simulations in practice has proven challenging for decades [6]. Available offerings typically provide sub-linear speedups, which reduce overall simulation throughput.

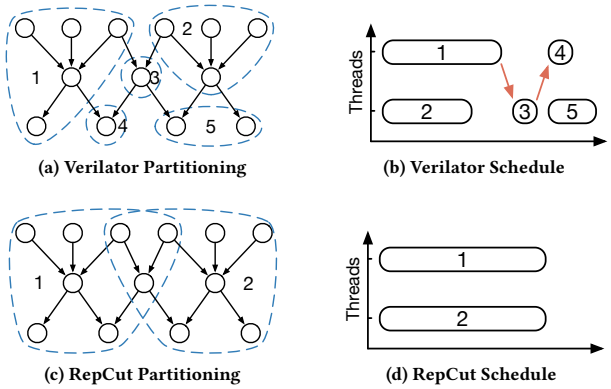


Figure 1: Comparison of partitioning and scheduling a design by leading prior work (Verilator) and this work (RepCut). RepCut reduces synchronization by removing inter-partition dependences with replication.

Distributing work among threads is a crucial aspect of parallelization efforts, since it impacts both load balance and inter-thread synchronization. Ideally, RTL simulation should execute in a single uninterrupted phase per simulated cycle. Accomplishing this requires partitioning the design such that each thread's work can be executed without dependences during that simulated cycle. Unfortunately, partitioning the hardware design in such a manner is practically impossible due to traits common to circuit topologies.

In this work, we propose a method that uses a trivial amount of replication to break intra-cycle dependences between partitions to enable such an efficient partitioning. This results in partitions that can be executed in parallel which generally achieve much higher utilization because they only need to synchronize once per simulated cycle (Figure 1). In addition to extracting more parallelism than prior approaches, we also create accurate work estimates to improve load balance across threads to achieve even higher speedups.

ASPLOS '23, March 25–29, 2023, Vancouver, BC, Canada

ACM ISBN 978-1-4503-9918-0/23/03.
https://doi.org/10.1145/3582016.3582034

Our approach successfully reduces the amount of synchronization, but for a sufficiently high degree of parallelism, synchronization can still limit performance [34, 37]. Thus, we anticipate there will be practical limits to how fast a modestly-sized design's simulation can be parallelized (strong scaling), and as a consequence, parallelization will be most effective on increasingly larger designs (weak scaling).

A surprise benefit of using a statically scheduled parallelization for this workload is that superlinear speedups are possible. Using additional host cores shrinks the amount of work assigned to each core such that each core performs better due to improved locality (caches and branch prediction). Our approach can benefit from nearly doubled instruction throughput as a result of this phenomenon.

We present *RepCut*, a parallel RTL simulator built on our novel partitioning approach, and we contribute the following:

- A parallel RTL simulation strategy that reduces synchronization between threads which increases scalability.
- Our partitioning method which makes use of a proxy problem, a hypergraph partioner, and our simulation effort model to generate independent and nearly balanced partitions.
- Evaluation including analysis with hardware performance counters in which RepCut achieves up to a 27.10× speedup with 24 cores and it outperforms the state-of-the-art parallel simulator Verilator with PGO enabled by up to 3.09×.
- We release RepCut as open-source with a BSD license.[1]

2 HARDWARE SIMULATION BACKGROUND

RTL simulators can be broadly classified as either *event-driven* [2, 16, 28–30, 46, 47] or *full-cycle* [10, 43]. Event-driven simulators propagate activity (value changes) through the hardware design as events. They can efficiently model arbitrary delays by using a priority queue to determine which event to simulate next. For cycle-accurate simulation, this timing precision is unnecessary, and the effort to track and prioritize events adds substantial overhead. A full-cycle simulator removes that overhead by using a static (precomputed) schedule. It compiles the design into a custom program that simulates only that design. Full-cycle simulators are termed *oblivious*, since they simulate the entirety of the design every cycle independent of the actual amount of activity. In practice, full-cycle simulation is much faster than fully event-driven, even though real-world designs often have a fair amount of inactivity.

Full-cycle simulators have uncommon workload characteristics that can stress the host processor in unusual ways [9]. The code within a full-cycle simulator is nearly straight-line (activity oblivious), as it simulates the entirety of the design every cycle. This results in extremely consistent execution times for each cycle. For small designs, this commonly achieves reasonably high instruction throughputs [35]. However, as the size of the design grows, so too does the size of the simulator program. With a sufficiently large design, the host processor's frontend can become overwhelmed. Although a large fraction of the program is re-executed every cycle, the reuse interval can become too large for the instruction cache and branch predictors to exploit. Surprisingly, the data working sets tend to be modest and enjoy reasonable locality. Even when full-cycle simulators are frontend-bound due to the size of a large design, they are still faster than event-driven.

Verilator is a high-performance open-source Verilog simulator that uses a full-cycle approach and includes a number of optimizations [43, 44]. It is commonly used in industry and academia due to its speed and non-existent licensing cost. We use it in this work as a high-performance baseline, and in the next section, we analyze its parallelization approach.

ESSENT is an open-source RTL simulator whose research contribution is exploiting low-activity factors to avoid unnecessary computation [10, 11]. We use ESSENT to prototype our approach due to its small modular codebase that eases our development. We disable its activity optimizations since they are orthogonal to this work. An unoptimized ESSENT-generated simulator is full cycle. ESSENT benefits greatly from being built on top of the mature FIRRTL [23] ecosystem.

3 PARALLEL SIMULATION CHALLENGES

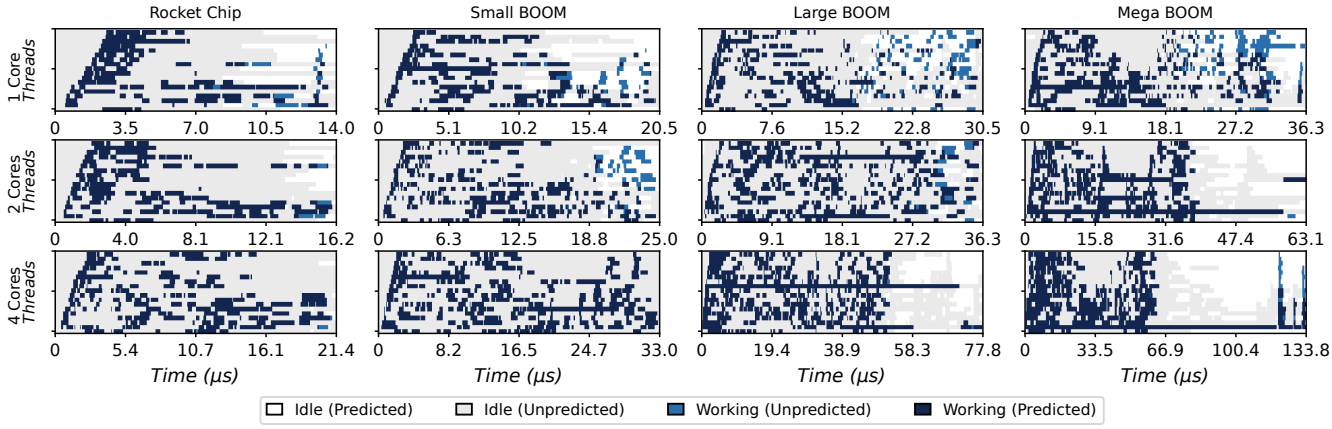
Partitioning a hardware design across threads for parallel simulation is a daunting task. Ideally, partitions will be perfectly balanced and communicate only once per simulated cycle. Doing so will reduce the time threads spend idling awaiting each other. Unfortunately, such a partitioning is typically impossible for real-world designs. Partitioning the circuit along boundaries that only need to communicate once a cycle (e.g. registers) results in wildly imbalanced partitions. Thus, parallel simulators need to cope with intra-cycle communication to balance work across partitions. Intra-cycle communication requires more synchronization, as there must also be a method to indicate when a dependence is ready (e.g. condition variable).

The approach taken by Verilator and other parallel RTL simulators is to intentionally *excessively partition* the design into far more partitions than threads [38]. Having many partitions increases parallelism and allows for partitions to execute completely instead of needing to pause midway for data dependences. With many partitions available, Verilator statically allocates partitions to threads to balance the workload. Since these partitions have intra-cycle data dependences, Verilator estimates the execution time of the partitions and schedules them so dependences complete before they are needed in order to reduce thread waiting. This method leads to worthwhile speedups and is a significant improvement over prior work. It is worth appreciating that this parallelization is done in an automated fashion by Verilator and it does not require user parallelization, partitioning, or annotations [39].

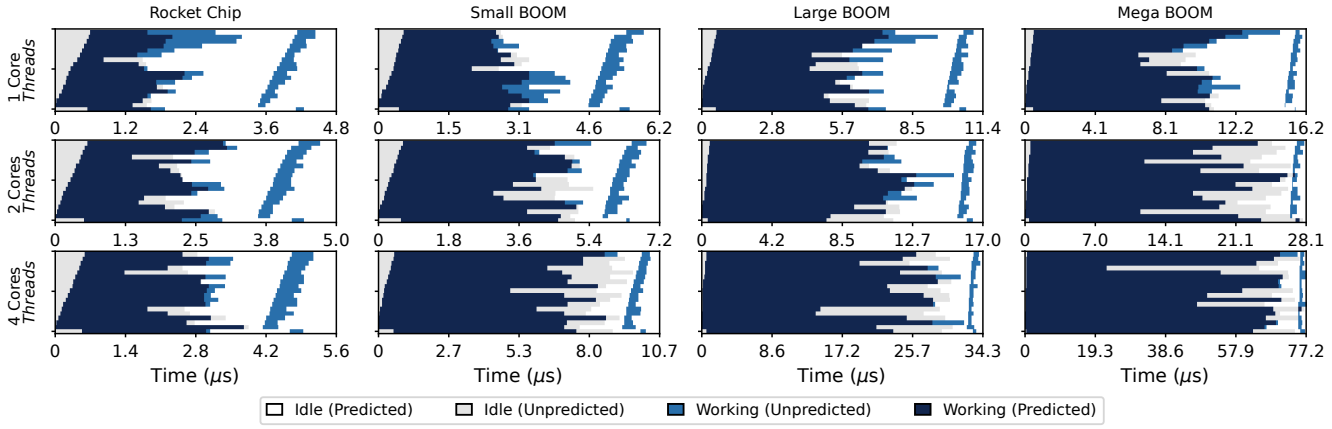
We profile Verilator (Figure 2a) and observe despite obtaining parallel speedups, many threads spend a significant amount of time idle. Idle threads are caused by waiting on data dependences, but there are a number of root causes for why those dependences are produced late:

- Some *partitions are far too large* for a balanced schedule because Verilator's partitioner does not limit partition sizes.
- Other *partitions are excessively small*, and although they can break up data dependences, they are not worth the overhead.
- *Inaccurate execution time predictions* prevent the scheduler from being able to ensure data dependences are ready before

[1] GitHub: https://github.com/ucsc-vama/essent/tree/repcut



(a) Verilator Thread Profile



(b) RepCut Thread Profile

Figure 2: Thread activity for Verilator and RepCut using 18 threads (profiling method in Section 6.5) across all designs (Table 1). RepCut's reduced synchronization keeps threads busier (more filled in) and delivers an overall speedup (shorter cycle time). Includes each simulator's thread's execution time predictions, but RepCut does not predict the global update phase at the end.

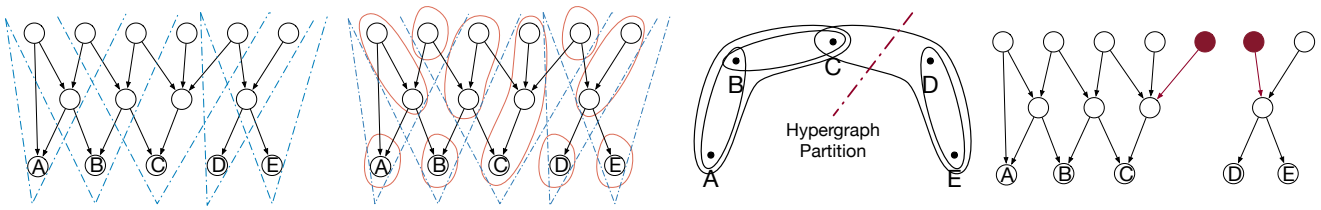
their dependents need them, which cause "bubbles" during execution (Figure 2a). This cost can be reduced by enabling Verilator's Profile-Guided Optimization (PGO), which collects partition execution times during execution and recompiles. We evaluate Verilator PGO in Section 6.

- Having *intra-cycle data dependences* in general runs the risk that there could be a stall for anything less than a near-perfect schedule.

Due to the lack of execution time variance between cycles, statically partitioning work is a natural optimization to avoid dynamic scheduling overheads. However, dynamic scheduling could bring practical advantages. First, it could allow for the number of threads used to be chosen after compilation time, and it could even be changed during execution. Second, it could load balance dynamically, which would obviate the need to have accurate work predictions. We suspect a parallel commercial simulator uses dynamic scheduling since according to its documentation, it allows for the number of threads to be set after compilation. Unfortunately, we are unable to access the parallel feature with the commercial simulator. Other experiments report Verilator outperforming the commercial simulator, both in serial or parallel [1, 8, 44].

A statically-scheduled simulator running at full utilization should be able to outperform a dynamically-scheduled simulator. Not only does static scheduling avoid the overhead of scheduling during simulation, but it will also have better thread locality for the caches. Changing the location where a task executes on different cycles reduces its locality, but pinning a task to the same location for better locality loses the advantage of dynamic scheduling.

From our analysis, we observe that efficient parallel RTL simulation will require statically-allocated partitions that are balanced and only need to communicate once per simulated cycle (Figure 1). Achieving such a balance with a single statically-allocated partition per thread will require accurate execution time predictions. Additionally, keeping the communication to a single round will require innovation to outfox limitations from the design topology.



(a) Traverse Cones (b) Divide Clusters (c) Build & Partition Hypergraph (d) Acquire Partitions

Figure 3: Replication-aided Partitioning Steps – Starting from sink nodes, crawl upwards to identify cones (a). Cluster nodes that belong to same set of cones (b). After building and partitioning the hypergraph (c), use replication (shown in dark red) to make partitions independent (d). Does not show the source nodes representing register reads since they are not partitioned.

4 REPLICATION-AIDED GRAPH PARTITIONING

Our novel design partitioning approach achieves our goals of balanced partitions that only require a single round of communication per simulated cycle. To overcome the topological challenges, our insight is to replicate a small portion of the design to break the dependences. In this section, we describe how we prepare the design, create a proxy problem for the partitioner, and how to accurately estimate simulation work to better balance our partitions.

To leverage existing abstractions, terminology, and algorithms, we view a hardware design as a directed graph. Each node represents a component (e.g. a logic gate, adder, multiplexor, or register) and each edge represents a wire connecting components.

4.1 Exploiting Chronological Independence of Registers and Memories

Before attempting to partition the design, we first split registers and memories into two nodes in the graph. Registers are an essential component in synchronous digital designs as they not only store data, but they also regulate the flow of data updates in a circuit. Every cycle, data is read from registers, fed through combinational logic, and the results are written back to registers. We represent each register with two nodes in the graph: one for reading the register (a source) and one for writing the register (a sink). This simplifies scheduling the simulation to have the correct data flow, e.g. no read should be able to see the result of a write in the same cycle. However, our main motivation for splitting registers and memories is to significantly reduce the graph's connectivity.

Partitioning the design purely along register boundaries is desirable, since partitions would only need to communicate once per simulated cycle. Unfortunately, our experiments find simply splitting registers generally does not yield balanced disconnected components in the graph. It typically creates a gigantic component along with a few tiny fragments, especially in complex designs. Further analyses reveal that typically only a handful of paths between large components make them connected. Fortunately, splitting registers creates a considerable amount of source and sink nodes in the graph, which significantly benefits the next step (partitioning). Splitting registers also guarantees that no internal state exists in the graph except at source and sink vertices, which makes each partition a pure function, which eases memory-related optimizations.

4.2 Generating the Intersection Hypergraph

We obviate edges between partitions that do not pass through registers by using replication to "cut" them, which also yields the name *RepCut*. Specifically, we only replicate combinational logic nodes (computation), since we have shared memory and thus have no need to copy data. Our approach succeeds because the cost of recomputing a small portion of the design is less than the time spent waiting for inter-thread synchronization. Prior work has considered using replication to avoid computation (Section 7), however, our approach to determine what to replicate and how to minimize that replication is novel.

We translate the novel problem of partitioning with replication into a proxy problem an existing partitioner can solve. In particular, we create a hypergraph that uses edge weights to convey the replication costs and node weights to convey the simulation costs. This proxy problem encourages the partitioner to minimize the weight of the edges cut (which minimizes replication) while also balancing partitions by node weight (which balances simulation time).

```
1  def TraverseCone(g, seed, cone_id):
2      fringe = g.vtxs[seed].predecessors()
3      while fringe.notEmpty():
4          vtx = fringe.pop()
5          if not visited(vtx):
6              g.vtxs[vtx].cone_ids.append(cone_id)
7              fringe.extend(g.vtxs[vtx].predecessors)
8
9  def TraverseAllCones(g): # g: Circuit DAG
10     seeds = g.sink_vertices()
11     for seed in seeds:
12         TraverseCone(g, seed, new_id())
```

Algorithm 1: Traverse Cones

To build our proxy problem, we use *hypergraphs* which generalize the graph abstraction. While a graph uses edges to represent connections between pairs of nodes, a hypergraph uses *hyperedges* to represent connections between sets of nodes. Conceptually, a hypergraph is a natural fit for a hardware design graph (netlist) as a single hyperedge can represent a wire driven by one component that broadcasts to many readers (net). A hypergraph can be losslessly transformed into a bipartite graph by replacing each hyperedge with a node and the necessary pairwise connections. However, keeping hyperedges intact prevents the partitioner from cutting a hyperedge's constituent edges independently.

We build the hypergraph for our proxy problem over multiple steps. We group vertices based on the topology to reduce the size of

the graph. First, for each vertex, we identify which cones it belongs to (Figure 3a). The *cone* of a vertex v is the set of its ancestors and itself, i.e. the vertices that can determine the value of v [40]. For each sink vertex, we crawl bottom-up until reaching source vertices and annotate all of its ancestors reached with the cone identifier (Algorithm 1). A non-sink vertex can belong to multiple cones.

We group vertices into a *cluster* if they belong to the same set of cones (Figure 3b). We then collapse each cluster into a single node. The resulting cluster graph greatly reduces the size of the design while grouping vertices together if they can impact the same sink vertices. A vertex in the original graph belongs to exactly one cluster. If a cluster contains a sink vertex from the original graph, it must be a sink vertex in the cluster graph. Otherwise, if the cluster has descendants (i.e. not a sink), it would contradict the sink in the original graph being descendant-free.

```
def BuildHypergraph(clusters):
    hg = HyperGraph()
    for cluster in clusters:
        if is_sink_cluster(cluster):
            weight = hg_vtx_weight(cluster)
            hg.addVertex(cluster.cid, weight)
        else:
            weight = hg_edge_weight(cluster)
            heVtx = cluster.cone_ids
            hg.addEdge(cluster.cid, weight, heVtx)
    return hg
```

Algorithm 2: Build Hypergraph

The final transformation (Algorithm 2) converts the cluster graph into a *intersection hypergraph* which captures the replication cost between overlapping cones. The intersection hypergraph $H = (V, E, \omega_v, \omega_e)$ is weighted and undirected. The vertices (V) represent sink clusters and the hyperedges (E) represent non-sink clusters, which connect to all of their descendant cones (Figure 3c). We denote the set of hyperedges connected to a vertex v as $\Gamma(v)$, and the number of endpoints (pin count) of a hyperedge e as $|e|$. We assign weights to vertices ($\omega_v(v)$) and hyperedges ($\omega_e(e)$) with Formula 1. Each node in the cluster graph is given a weight η based on its predicted simulation time (next subsection).

$$\omega_v(v) = \eta(v) + \sum_{e \in \Gamma(v)} \frac{\eta(e)}{|e|} \qquad \omega_e(e) = \eta(e) \tag{1}$$

The weight of a hyperedge is the weight of the cluster node it represents from the cluster graph. The weight of a vertex is the weight of itself in the cluster graph as well as the sum of its proportional share of all of its ancestors. A hyperedge cut implies its corresponding cluster needs to be replicated across different partitions.

Partitioning the intersection hypergraph creates a useful proxy problem, but it does not perfectly capture the weight of the resulting partitions. When we assign vertex weights ($\omega_v(v)$) before partitioning, we do not yet know how hyperedges will be cut. Thus, we distribute their weight evenly to their descendants. This assumption introduces minor error which we analyze in the evaluation (Section 6.6).

Graph partitioning is known to be NP-hard [5, 13, 26, 27, 45], and hypergraph partitioning is no exception. Fortunately, hypergraph partitioning is a well-studied problem, and well-tuned heuristic-driven frameworks can produce good results in practical amounts of time [14, 25, 42]. In this work we use KaHyPar [42] as our hypergraph partitioner because of its speed, high-quality results, and support for our objective *cost* function (Formula 2).

4.3 Simulation Cost Model (Node Weights)

Accurately predicting the simulation time of a partition is essential for improving load balance with a static partitioning. Verilator assigns a weight to each Verilog AST node and accumulates them for each task [43], but in practice the predictions can be inaccurate (Figure 2). Compiler optimizations may confound such simple models, and inaccurate predictions worsen load balance and ultimately speedup.

RepCut predicts cluster simulation times with FIRRTL's low-level IR [23] instead of at the Verilog AST level. This enhancement brings multiple benefits that help improve prediction accuracy. First, low-level FIRRTL is closer to the generated C++ code, allowing us to more easily consider compiler optimizations like constant propagation. Second, low-level FIRRTL provides considerable information such as the IR type, operation type, reference type, and data width. The additional information makes accurate prediction feasible while bypassing interference from high-level language constructs. A crucial advantage is the ability to determine if a reference is connected to a wire or a register. We assume wire references are zero-cost since the C++ compiler is capable of optimizing circuit references.

RepCut's simulation time predictor is a simple linear model. We build the weights for our model based on a least squares linear regression on the aforementioned attributes and simulation times for a variety of circuit partitions.

4.4 Using the Hypergraph Partitioning Result to Guide Replication

After partitioning, cones in the same partition[2] remain connected, and clusters required by those cones can be reconstructed to satisfy data dependences (Figure 3d).

We formulate the overall replication cost where *cut* denotes the set of all hyperedges cut by the partitioning:

$$cost = \sum_{e \in cut} (|e| - 1)\omega_e(e) \tag{2}$$

We use replication to break dependences, but it can create additional computational overhead and even deteriorate partition balance. In the worst case, the replication burden is imposed solely on a single partition, which would cause the imbalance introduced by replication to be the *cost* function. In practice, the replication is more disperse and less unbalanced.

5 PARALLELIZATION APPROACH

With our balanced and independent partitions available, we turn our attention to the overall parallel execution strategy as well as its memory layout. Our partitioning approach greatly reduces inter-thread communication and synchronization, which in turn makes these factors crucial to solve to scale to higher parallel speedups.

[2] A typical definition for a partitioning requires the partitions to be disjoint. We argue our partitions do not violate this property, because after the necessary replication, any vertex is in only one partition.

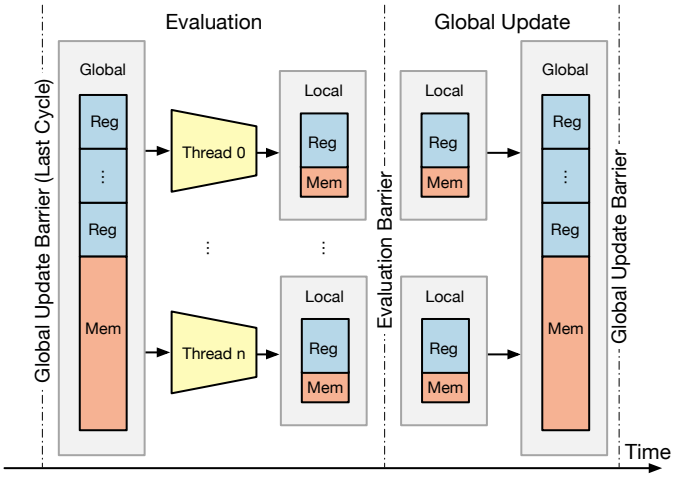


Figure 4: Execution Phases – In the Evaluation Phase, threads access global values to compute the new values in local (private) memory. After a barrier, threads copy their updated local values to global data in the Global Update Phase. A second barrier completes the cycle and the next cycle can start.

5.1 Parallelizing with a Single Update per Cycle

During RTL simulation, the only data saved from one cycle to the next are the values of the state elements (registers and memories). Thus, to simulate a cycle is to use the current state to compute the next state. We store these state elements in a shared global data structure to ease access. Our partitioning scheme ensures there are no dependences between partitions during a cycle, but there are data dependences between partitions between cycles (e.g. registers on boundaries). To regulate these data exchanges, we simulate each cycle with two barrier-separated phases: *Evaluation* and *Global Update*:

- **Evaluation Phase**: Each thread reads registers and memory values from the global copy, evaluates logic in the partition, and writes registers into a local copy. It buffers (delays) memory write requests.
- **Evaluation Barrier**: Wait for all threads to complete their evaluation stage.
- **Global Update Phase**: Each thread overwrites the global copy using its local copy, and performs deferred memory write requests.
- **Global Update Barrier**: Wait for all threads to complete updates to global data before moving onto the next cycle.

During the evaluation phase, each thread records the new values of registers in its partition in thread-private memory (Figure 4). Keeping these values thread private prevents other threads from seeing new values prematurely or forcing threads to wait on each other within a cycle. To speed up the global update phase, we arrange the registers in the host's memory such that each thread controls a single contiguous portion. Each thread's private registers use the same layout, so a single `std::memcpy` (typically vectorized) can speedily copy the values from thread private to shared memory during the global update phase.

Since simulated memories may have much larger capacities than registers, we treat them differently. We simulate each memory with a single shared representation. During the evaluation phase, we record the details of the memory writes and delay updating global memory until the global update phase. The buffering prevents a read from prematurely seeing the value of a write.

5.2 Optimizing the Memory Layout to Avoid False Sharing

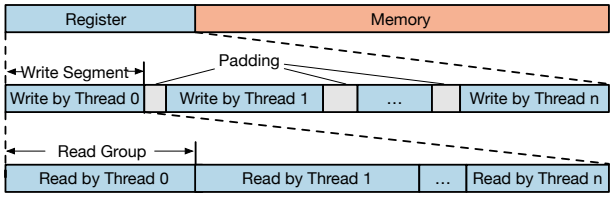


Figure 5: Memory Layout – Register locations in memory are chosen to improve atomicity and locality. Each thread updates a single contiguous portion of memory and padding is added to align it before the next thread. Within a region a thread writes, registers are grouped by which thread will read them. For a simulated memory, there is only one copy and updates to it are deferred until the Global Update Phase.

We perform additional memory layout optimizations beyond ensuring all of a thread's registers are adjacent in the global copy (Figure 5). Within each thread's segment, registers are further sorted by readers (threads) and are topologically ordered within each group. Since ESSENT emits statements in a topological order, sorting registers this way effectively increases spatial locality.

False sharing can be an extreme performance limiter for multithreaded programs [12, 22]. Multiple objects that are not shared but reside in the same cache line can cause that cache line to constantly thrash unnecessarily from core to core. In our work, false sharing could happen due to writes during the global update phase. We eliminate false sharing for simulated registers by ensuring only one thread can be a writer for each cache line. During the evaluation phase, writes are to thread-private memory, and in the global update phase, we pad the thread segment boundaries. False sharing on simulated memory updates is possible but rare. Most memories in hardware have a limited number of write ports, and they typically do not access nearby addresses.

6 EVALUATION

We implement RepCut by extending ESSENT [10] to generate multithreaded simulators. Our work only requires around 2,000 additional lines of Scala code, including partitioning and code generation. The C++ code generated by RepCut requires the C++ standard library to provide threading and synchronization primitives (C++11).

We use the following designs to benchmark (Table 1):

- **RocketChip** is an open-source system-on-chip generator written in Chisel [3]. We select the default in-order core (Rocket) and generate designs with 1, 2, or 4 cores.

- **BOOM** is an open-source RISC-V superscalar out-of-order core generator [4, 49]. BOOM is highly parameterized and can be flexibly configured. We utilize three common configurations: *SmallBoomConfig* (1-wide with 32 ROB entries), *LargeBoomConfig* (3-wide with 96 ROB entries), and *MegaBoomConfig* (4-wide with 128 ROB entries), each generated with 1, 2 or 4 cores.

Table 1: Evaluated Designs

Design	IR Nodes	Edges	Sink Vtx	Sink (%)	Reg Writes
RocketChip-1C	69,995	116,246	11,137	15.91	3,149
RocketChip-2C	101,154	168,765	14,508	14.34	4,682
RocketChip-4C	164,266	276,114	21,150	12.88	7,710
SmallBOOM-1C	118,704	214,009	17,100	14.41	8,244
SmallBOOM-2C	198,520	364,247	26,434	13.32	14,872
SmallBOOM-4C	361,899	671,439	45,598	12.60	28,178
LargeBOOM-1C	229,699	459,582	25,957	11.30	15,483
LargeBOOM-2C	421,429	856,125	44,238	10.50	29,345
LargeBOOM-4C	803,911	1,648,664	80,320	9.99	56,893
MegaBOOM-1C	335,678	703,258	32,320	9.63	19,604
MegaBOOM-2C	632,292	1,341,756	56,680	8.96	37,523
MegaBOOM-4C	1,224,804	2,618,373	105,068	8.58	73,213

We evaluate the following simulators:

- **RepCut** implements our partitioning and code generation approach. Since we start from ESSENT's codebase with optimizations disabled, a single-threaded simulation is equivalent to ESSENT with the `-O0` flag.
- **RepCut UW (Unweighted)** removes the benefit of our simulation cost prediction model (Section 4.3) and simply uses the partition size for balancing partitions.
- **Verilator** is an open-source Verilog simulator that provides performance competitive with commercial simulators.
- **Verilator PGO** enables Verilator's PGO optimization (Section 2). Verilator's static scheduler is able to obtain accurate execution time information as a result of PGO.

We use a dual-socket server with 24 cores per socket for our evaluation (Table 2). We carefully control thread placement and pinning (via `numactl`) to guarantee each simulator achieves its maximum performance. We assign at most one thread per core. We compile all simulators with clang++ 10 unless specifically mentioned. In one experiment, we consider the benefit of using clang++ 14.

Table 2: Evaluation Environment

Field	Value
CPU	2× Intel Xeon Platinum 8260
L1 Cache	48× private 32 KB L1I, 32 KB L1D
L2 Cache	48× private 1 MB L2, inclusive
L3 Cache	2× shared 35.75 MB L3, non-inclusive
OS	Ubuntu 20.04 LTS (default), 22.04 LTS (Clang 14)
Compiler	Clang++ 10.0 (default) and Clang++ 14.0, -O3
Verilator	Verilator 4.226, -O2

6.1 Opportunity for Partitioning Design Graphs

We first consider the evaluated designs topological characteristics to determine their parallelizability. After splitting the registers, the designs contain a significant fraction of sink vertices (Table 1). A large fraction of sink vertices makes the design graph wider and shallower, which creates more opportunities for the hypergraph partitioner.

6.2 Replication Costs Are Worthwhile

Replication is the unavoidable cost of our partitioning strategy. We measure replication cost by considering the combined weight of the extra work (Formula 3).

$$replication\ cost = \frac{\sum_{p \in Partitions}(weight(p))}{weight(entire\ circuit)} - 1 \tag{3}$$

Fortunately, if the replication cost is modest, it can be easily overcome for an overall speedup (Figure 6). We achieve an overall replication cost of less than 25% for all designs we test when partitioning up to 24 ways. Replication costs increase as the number of partitions grows, while the rate of replication decreases with larger designs and more cores per design, suggesting that independence between different circuit blocks (cores in this case) can be efficiently exploited. Replication costs are typically modest for a reasonable number of partitions relative to the design size.

6.3 Performance & Superlinear Scalability

We first measure the *scalability* of each simulator, that is the speedup of each simulator relative to a single-threaded execution of itself (Figure 7). Since each simulator has different internal algorithms and implementations, we use speedup to evaluate the effectiveness of its parallelization methodology while ignoring performance differences from their baseline implementations. For a small design such as *RocketChip-1C*, RepCut and Verilator scale similarly. As the size of the design grows, RepCut scales significantly better (up to 6.31×) than Verilator, and it can even yield a superlinear speedup on certain configurations which we discuss in the next subsection. RepCut's simulation time predictions provide a noticeable speedup improvement (over RepCut UW).

Larger design sizes normally present scalability challenges. Unlike other simulators, RepCut benefits from larger designs (Figure 8), as they provide more opportunity to compensate for overheads (weak-scaling). Verilator is unable to benefit from large designs as its partitioner often yields a few gigantic partitions (Figure 2a) which cause cores to frequently stall waiting for data dependences, and thus ultimately undermine its parallel speedup. Those gigantic partitions also diminish any potential benefit that Verilator's scheduler could obtain from PGO data.

We also compare the simulators' absolute simulation performance in terms of simulation speed measured in thousands of simulated cycles per second (Figure 9). With a single thread, RepCut is faster for smaller designs but is outperformed by Verilator as the design size increases. However, using additional threads greatly accelerates RepCut, and it achieves a maximum simulation speed of 149.15 KHz when using 6 threads to simulate the smallest design (*RocketChip-1C*). Overall, RepCut at its best thread count is the

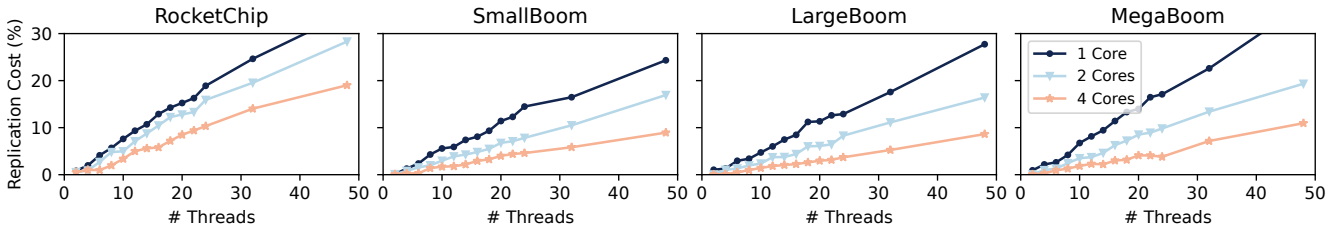


Figure 6: Replication Cost (Formula 3) – Larger designs, especially with more cores, require less replication.

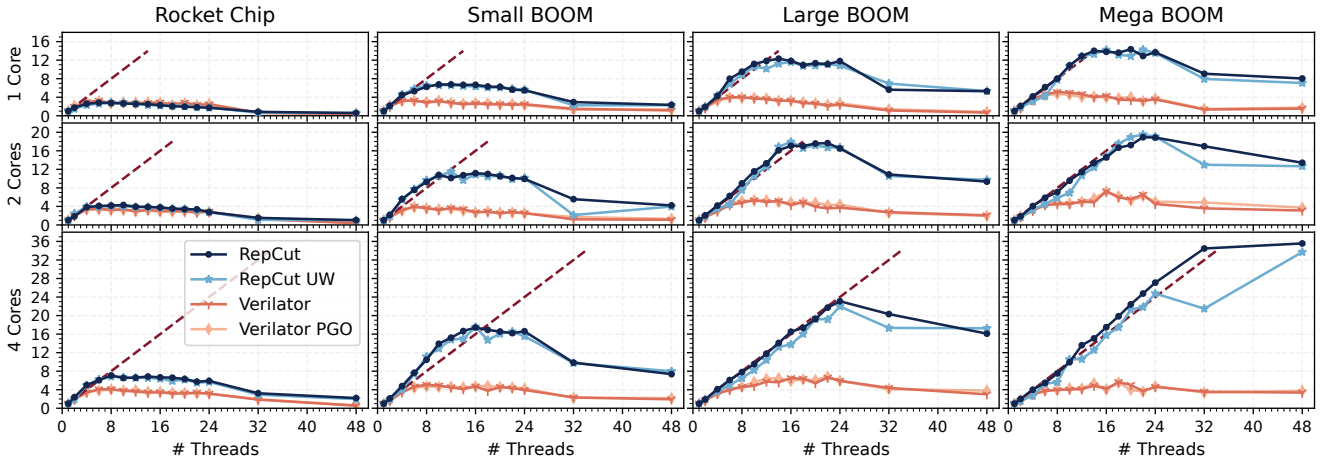


Figure 7: Speedups relative to singled-threaded execution of self (Scalability) – All implementations benefit from larger designs (moving right and down), but RepCut benefits even more from increased scalability. Figure 9 shows absolute performance. The hitch for RepCut UW on *MegaBOOM-4C* (bottom right) is reproducible and most likely caused by imbalance caused by significant skew in the computational cost of nodes assigned to each partition, since it does not have a cost model.

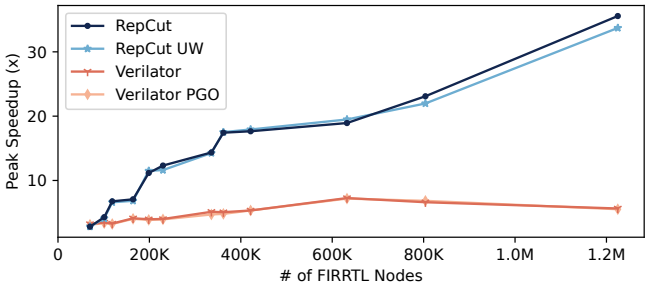


Figure 8: Peak (self-relative) speedup for every design – Larger designs (moving right) enable higher speedups.

fastest for every design, and is significantly faster (2.22 - 3.98×) than Verilator.

The performance of simulators generated by RepCut are significantly influenced by compiler optimizations, and using a more recent and thus more advanced compiler can result in significant speedups. We compare the performance of RepCut, RepCut UW, and Verilator using different versions of the clang compiler (Figure 10). RepCut benefits greatly from better optimizations, and using the latest version of clang (14) nearly doubles the simulation performance on the largest design (*MegaBOOM-4C*). Furthermore, the newer compiler better aligns with our simulation time prediction model, as demonstrated by a significant performance advantage over RepCut UW (Figure 9). However, unlike RepCut, using the newer compiler has a limited improvement on Verilator's performance.

6.4 Analysis to Understand Performance

A superlinear speedup is not typical in parallel programming and is worth investigating. It implies the parallelized program expends less aggregate CPU time than the sequential version.

To dig deeper, we collect performance counter information using `perf`[3] regarding caches, branches, and pipeline stalls for an increasing number of threads (and cores) within a socket and even interleaved across sockets (Table 3). We analyze *MegaBOOM-4C*

[3] `perf` competes for cache resources with the RepCut simulator and significantly slows down only a single-threaded simulation of *MegaBOOM-4C*. For consistency, we use measurements without `perf` for computing speedups.

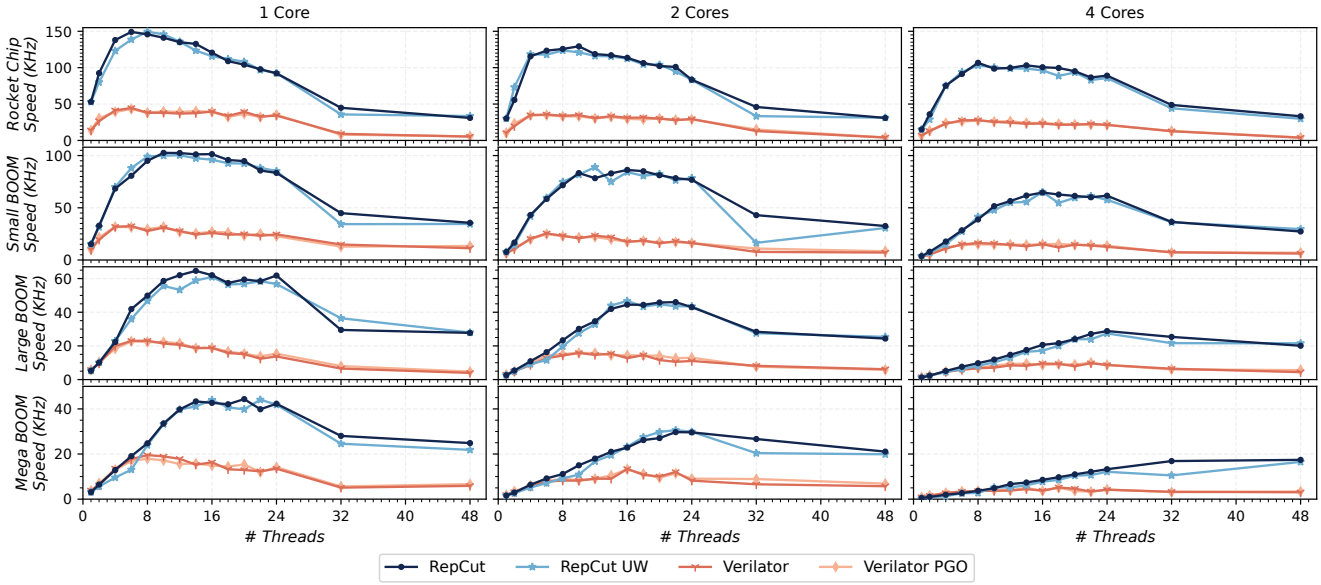


Figure 9: Simulation Speed (KHz) vs. Number of Simulation Threads – RepCut enjoys a significant performance advantage, especially for larger designs or designs with more cores. Despite having comparable scalability (Figure 7) on the difficult to scale smallest design (*RocketChip-1C*), RepCut provides greater absolute performance.

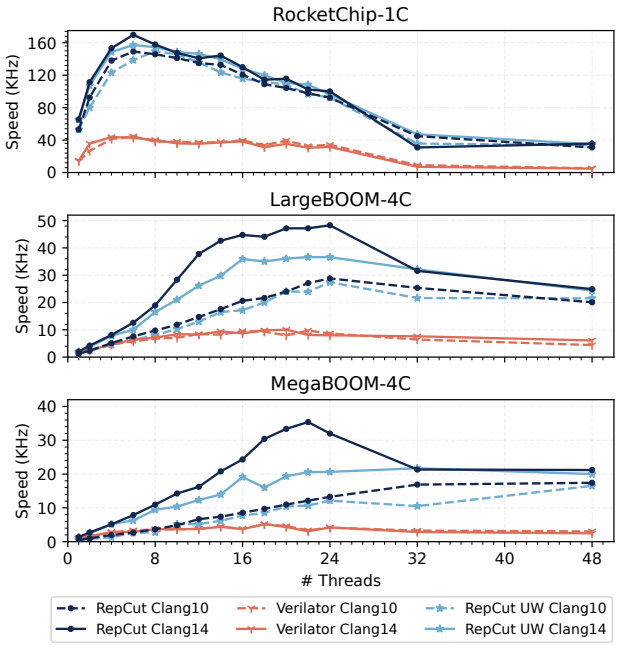


Figure 10: Compiler Impact on Simulation Speed (KHz) – The performance of RepCut is significantly improved by using the Clang 14 compiler (solid line) compared to Clang 10 (dashed line). Moreover, our simulation time prediction model delivers an even greater performance benefit when using the newer compiler (RepCut vs. RepCut UW).

since it is the largest design in this work. When compiled as a simulator, it has a program binary size of 31 MB to 36 MB, depending on the thread count. The large binary size leaves insufficient last-level cache (LLC) space for the simulator when running on a single socket.

The instruction caches are overwhelmed by the large design, but the data caches enjoy frequent hits. Increasing the number of threads used shrinks the partition sizes and thus the corresponding program segments assigned to each thread. Using more threads does little to improve the L1 instruction cache hit rate, as the smaller partitions are still much larger than the L1 cache size (Table 3). Most misses for the L2 cache are code read misses (*l2_rqsts.code_rd_miss*) and they drop as we use more threads. The lowest L2 cache code read miss rate occurs with 24 cores (a full socket) when the amount of code allocated to each core approaches the L2 cache size. A reduced code footprint also contributes to improved processor memory prefetcher accuracy. Interestingly, using more cores also decreases the number of L3 misses, since there is additional free space in the non-inclusive L3 cache as more code resides uniquely in the L2 caches.

The variation in the number of instructions executed is due in large part to threads waiting at thread barriers and the small amount of replication introduced by our partitioning approach. Using more cores reduces the overall branch miss rate from 5.29% with a single-thread to 0.31% with 8 threads, and ultimately 0.18% with 48 threads (2 sockets).

Using more threads results in a considerable reduction in front-end bubbles because more processor resources become available and the size of the code allocated to each core decreases. It is worth

Table 3: Performance counter measurements for RepCut simulating *MegaBOOM-4C*. Considers various thread counts and allocation entirely on the *same* socket or *interleaved* across both sockets. Event counts are total across all cores.

		1 Socket					2 Sockets, Interleaved				
	Perf Event	1 Thread	4 Threads	8 Threads	16 Threads	24 Threads	4 Threads	8 Threads	16 Threads	24 Threads	48 Threads
Cache	L1-icache-load-misses	543 B	563 B	543 B	550 B	556 B	565 B	543 B	549 B	556 B	586 B
	L1-dcache-load-misses	137 B	108 B	85 B	77 B	74 B	107 B	85 B	77 B	74 B	67 B
	L1-dcache-loads	1,786 B	1,830 B	1,854 B	1,862 B	1,881 B	1,807 B	1,842 B	1,865 B	1,883 B	2,061 B
	L1-dcache-stores	1,522 B	1,530 B	1,558 B	1,577 B	1,592 B	1,531 B	1,558 B	1,577 B	1,590 B	1,677 B
	l2_rqsts.code_rd_miss	536 B	551 B	537 B	453 B	256 B	551 B	536 B	456 B	259 B	14 B
	l2_rqsts.code_rd_hit	4 B	4 B	5 B	96 B	300 B	5 B	5 B	92 B	297 B	571 B
	l2_rqsts.pf_miss	36 B	38 B	28 B	10 B	13 B	36 B	26 B	10 B	14 B	26 B
	l2_rqsts.pf_hit	58 B	47 B	35 B	31 B	34 B	44 B	33 B	26 B	24 B	14 B
	l2_rqsts.all_demand_miss	550 B	569 B	553 B	462 B	264 B	571 B	553 B	465 B	267 B	26 B
	l2_rqsts.all_demand_data_rd	70 B	48 B	33 B	29 B	27 B	48 B	34 B	29 B	28 B	27 B
	l2_rqsts.miss	586 B	608 B	581 B	472 B	278 B	608 B	580 B	476 B	281 B	53 B
	LLC-load-misses	331 M	212 M	146 M	25 M	15 M	335 M	1,214 M	1,404 M	1,416 M	4,075 M
	LLC-loads	5,915 M	7,348 M	5,974 M	3,997 M	4,906 M	7,574 M	6,154 M	4,363 M	5,369 M	10,413 M
	LLC-store-misses	426 M	217 M	101 M	34 M	23 M	48 M	85 M	184 M	228 M	703 M
	LLC-stores	7,105 M	7,432 M	6,793 M	1,491 M	1,553 M	8,699 M	7,332 M	1,205 M	950 M	1,309 M
Branch	instructions	5,937 B	6,129 B	6,266 B	6,349 B	6,442 B	6,033 B	6,212 B	6,367 B	6,445 B	7,110 B
	branch-misses	4,537 M	668 M	413 M	364 M	360 M	664 M	391 M	361 M	350 M	385 M
	branches	85 B	124 B	133 B	122 B	131 B	101 B	121 B	128 B	133 B	216 B
Pipeline	topdown-fetch-bubbles	50,648 B	28,988 B	24,302 B	14,930 B	10,227 B	15,154 B	13,260 B	11,699 B	8,861 B	2,548 B
	icache_16b.ifdata_stall	10,634 B	5,691 B	4,724 B	2,519 B	1,543 B	2,446 B	2,127 B	1,720 B	1,204 B	88 B
	icache_64b.iftag_stall	3,656 B	930 B	373 B	216 B	193 B	688 B	356 B	239 B	205 B	174 B
Misc	Wall Clock Time	3,774.80 s	752.33 s	361.11 s	144.48 s	101.07 s	453.94 s	259.19 s	130.10 s	87.56 s	71.56 s
	CPU Time	3,773.63 s	3,004.75 s	2,873.12 s	2,251.22 s	2,292.70 s	1,807.52 s	2,040.81 s	1,999.87 s	1,939.03 s	2,925.87 s
Analysis	IPC	0.41	0.55	0.61	0.85	0.95	0.87	0.83	0.89	0.93	0.79
	Branch Miss Rate	5.29%	0.54%	0.31%	0.30%	0.27%	0.66%	0.32%	0.28%	0.26%	0.18%
	Extra Instructions (vs. 1 thread)	-	3.23%	5.55%	6.95%	8.51%	1.61%	4.63%	7.24%	8.56%	19.75%
	Replication Cost	-	0.27%	1.30%	3.00%	3.81%	0.27%	1.30%	3.00%	3.81%	10.95%

noting our performance events are reported in aggregate, so a metric that appears largely unchanged actually occurs fewer times per core. As the amount of work per core becomes sufficiently small, the core crosses a performance threshold when going from frequent cache misses to frequent cache hits. Achieving a 27.10× speedup using 24 threads is understandable when considering that the IPC doubles from 0.41 using 1 thread to 0.95 with 24 threads.

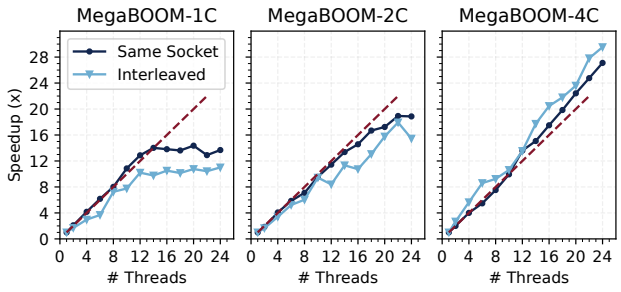


Figure 11: Socket Allocation Impact on RepCut's Speedup on *MegaBOOM* Design – Typically, performance is better if threads are allocated on the same socket due to inter-socket latency penalties. However, for the biggest design (*MegaBOOM-4C*), interleaving the threads across sockets performs better due to effectively twice the L3 cache capacity.

RepCut is able to take advantage of additional processor resources for sufficiently large designs, even if they come from multiple sockets. Parallel RTL simulators are not unlike most parallel programs that frequently access main memory and perform synchronizations across host cores. They prefer their threads to be allocated within the same NUMA node to avoid overhead incurred by NUMA (inter-socket) communication latency. Though RepCut-generated simulators are not heavily communication bound, for most benchmark designs, the simulation speed is slower when we manually interleave them across sockets using `numactl` (Figure 11). However, we also observe that *MegaBOOM-4C* is sufficiently large to be an exception. The identical simulator generated by RepCut achieves a remarkable 8.60× speedup using 8 threads, 20.46× using 16 threads, and even 29.53× using 24 threads when running interleaved across two sockets, significantly outperforming itself running within a single socket, which already achieves a super-linear speedup. When running with both sockets, the simulator benefits from double the L3 cache capacity which greatly reduces the number of pipeline bubbles caused by instruction fetch hazards (Table 3). The inevitable inter-socket communication occurs only once per cycle, and it can be largely diluted for a sufficiently large design.

6.5 Profiling RepCut Reveals High Utilization

To analyze the parallelization efficiency of our implementation, we profile our RepCut-generated simulators by collecting precise timestamps[4] at the start and end of each simulation phase (Figure 2b). We record and process the timestamps offline in order to minimize the performance impact introduced by profiling.

[4]We utilize x86's `rdtsc` instruction which returns the value of the time stamp counter that increases at a constant rate across all cores.

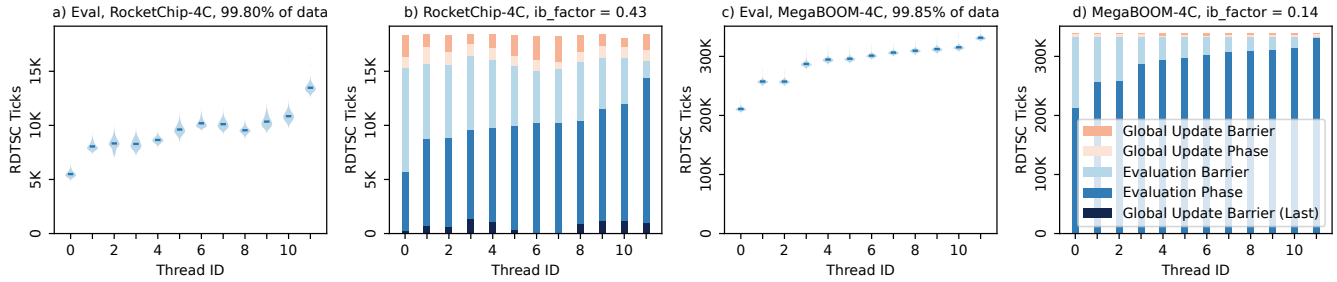


Figure 12: Thread Timing Profiles per Simulated Cycle for *RocketChip-4C* (small) and *MegaBOOM-4C* (large) – Both executions demonstrate little variance in the execution time of a partition (a & c). The larger design has more useful work to amortize the fixed amount of synchronization that limits the smaller design (b & d). Threads reordered for illustration.

We present data for only two configurations due to page limitations (Figure 12). We visualize a smaller design (*RocketChip-4C*) and the largest design (*MegaBOOM-4C*) which when using 12 threads, provide a modest 6.61× speedup or an impressive 13.60× speedup, respectively. We drop extreme outliers for analysis of the evaluation stage, as most of those outliers are inevitably introduced by the operating system (e.g. context switches).

There is typically little variance in the evaluation time for each thread (Figure 12a & Figure 12c), as work is statically assigned to threads and that work has few data-dependent branches. Looking at how each thread spends its time within a simulated cycle gives a better view of efficiency (Figure 12b & Figure 12d). The threads in the larger design (*MegaBOOM-4C*) spend a greater fraction of the time doing useful work (evaluation). Although some threads spend a large fraction of time waiting at the barrier (e.g. thread 0), they also complete their evaluation promptly. This is in contrast to the results from the smaller design (*RocketChip-4C*) which enjoys a more modest speedup since most threads spend less time doing useful work (evaluation). This is partly an inevitable challenge for smaller designs, as in this case, the execution time for a simulated cycle is nearly an order of magnitude less. That shorter simulated cycle time provides less time to amortize the near-fixed costs for thread synchronization and the global update phase. Additionally, thread 11 (Figure 12b) is a clear straggler which causes other threads to spend more time waiting at the barrier. We investigate workload imbalance in the next subsection.

6.6 Quantifying Workload Imbalance

Our approach uses barrier synchronization for each simulated cycle to update global data, and thus workload imbalance can greatly degrade parallelization efficiency. More specifically, a tiny partition will not severely hinder performance, because a single thread waiting is not significant. However, a partition that is much larger than average should be avoided, as all of the other threads will be waiting for it at the barrier. We quantify imbalance and parallelization efficiency as:

$$imbalance\ factor = \frac{max(part\ size) - avg(part\ size)}{avg(part\ size)} \quad (4)$$

$$parallelization\ efficiency = \frac{achieved\ speedup}{ideal\ speedup} \quad (5)$$

There is a strong trend between workload balance and parallelization efficiency (Figure 13). We trace the balance through various stages of our tool flow and execution to find the source of imbalance (Figure 14). KaHyPar partitions the intersection hypergraph in an almost perfectly balanced manner (marked as *Excluding Replication*), but the resulting design partitions exhibit significant imbalance (marked as *Including Replication*). Once those partitions actually execute, there can be additional imbalance (marked as *Measured*). Imbalances exist in our approach as a consequence of our partitioning strategy and inaccurate execution time predictions. Since our simulator is activity-oblivious, circuit activity is unlikely to introduce noticeable variance.

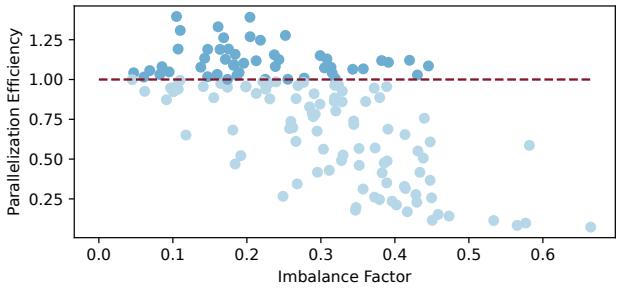


Figure 13: Parallelization efficiency degrades with load imbalance, suggesting load imbalance is a performance limiter. Formulas 4 & 5 define metrics. Uses profiled executions of RepCut for all designs and thread counts.

The proxy problem we provide to the hypergraph partitioner encodes the cost of replication as weights on the hyperedges. However, the partitioner attempts to balance the vertex weights while it attempts to reduce the total weight of the hyperedges cut. Minimizing the cut will minimize the replication, but how that replication is distributed is not controlled and can thus cause partition imbalance. Furthermore, the execution time of a partition on real hardware can be challenging to predict, as the compiler may be able to optimize some structures better than others and the execution may be more efficient on the host core's microarchitecture. Despite an imperfect balance, our approach is still able to provide breakthrough scalability and overall performance for RTL simulation.

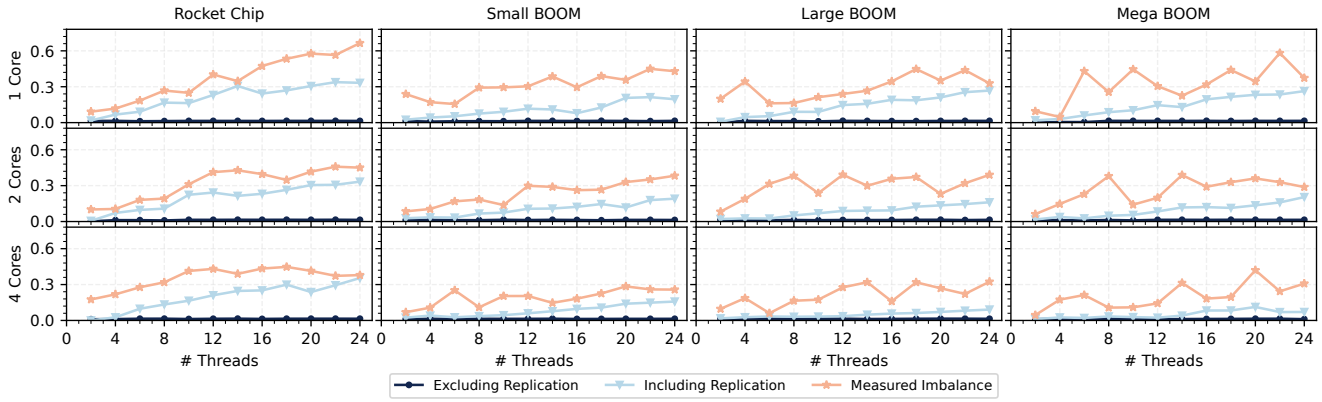


Figure 14: Imbalance Factor - A greater number of partitions worsens the *Imbalance Factor* (Formula 4) which is caused both by uneven replication and inaccurate execution time predictions. We exclude data beyond 24 cores to avoid inaccuracy introduced by NUMA latency.

7 RELATED WORK

We focus our discussion of related work specifically to parallel RTL simulation. Both event-driven and full-cycle approaches have been parallelized, and both struggle with low parallelism caused by circuit connectivity. Event-driven approaches can be limited by waiting or fallback, while full-cycle approaches can be limited by synchronization and load imbalance.

Using conventional parallelization methods on event-driven simulation demonstrate limited scalability [1, 15, 32]. Alternatively, MULTES uses simplified high-level simulation to quickly generate checkpoints it then simulates in parallel with detailed simulation [29]. Cascade executes different simulation methods (CPU and FPGA) in parallel and dynamically switches over to mask long compilation times [41]. Parallelizing event-driven simulation, especially for a distributed context, requires additional parallelism to cope with the increased latency. Common techniques include optimistic concurrency or predicting future input stimuli [2, 28, 47]. However, both checkpoints or accurate predictions require output generated from a higher abstraction level simulation, which is not always available for RTL simulation.

Efficient parallelization can be eased with the help of a user or hardware designer manually intervening [20]. ArchHDL achieves promising speedups with a customized language and user-provided parallel hints [39]. Though converting ArchHDL to Verilog is feasible, rewriting existing HDL designs and providing adequate hints will require extensive labor.

Using GPUs to accelerate RTL simulation requires more workload regularity than the multicore methods we explore in this work. Coping with this constraint, Qian et al. translate RTL code into GPU kernels using only one thread per warp (to limit divergence) and they achieve considerable speedups when running benchmark designs with regular, repetitive patterns such as adder arrays and AES encryption [36]. RTLflow batches multiple simulations with different stimuli on the same design to exploit the GPU's SIMT capabilities, and it achieves much higher aggregate simulation throughput than a single stimulus [33]. Gate-level simulation has successfully been parallelized on GPUs by regularizing the design by simulating logic gates via table lookups [16, 17, 48]. Gate-level simulation has an order of magnitude more logic nodes than RTL simulation, but RTL simulation has more diverse internal behaviors, which make truth table lookups impractical.

Hardware support can accelerate parallel hardware simulation by enabling more sophisticated scheduling. The Swarm architecture uses speculative execution to accelerate ordered irregular parallelism [24]. An architectural simulation of Swarm demonstrates a linear speedup on up to 64 cores on a circuit simulation benchmark. Unlike Swarm, RepCut does not rely on any specialized hardware and can thus be more easily deployed on conventional systems.

Using replication to avoid communication is an established optimization for parallel computing, especially in distributed contexts. When replicating, one can replicate data and/or computation. The appropriately named *communication-avoiding linear algebra* work uses replication to avoid communication in both multicore (to reduce memory bandwidth) and distributed (to reduce network bandwidth) contexts [7]. Mirroring or caching remote vertices is commonly done in distributed graph frameworks, and PowerGraph [19], CuSP [21], and Gluon [18] are notable examples. By contrast, our work only replicates computation and not data since we use shared memory. Furthermore, our partitioning seeks to fully disconnect a directed graph to require only one round of communication per simulated cycle, which is different than what is typically encountered by general-purpose distributed graph frameworks.

Chatterjee et al. demonstrate the parallelization benefit of overlapping partitions (replication) for gate-level simulation [16, 17]. However, their approach must limit the depth of partitions to keep replication costs manageable. As a result, they must partition the design into layers of parallel partitions, and synchronization and communication at layer boundaries still limits performance. A high amount of replication also adds considerable extra computation and degrades performance.

Throughout this work, we discuss (Section 2) and evaluate (Section 6) Verilator [43] and we build our experiments on ESSENT [10].

8 CONCLUSION

In this work, we introduce RepCut. With our novel partitioning scheme and optimization opportunities that emerge from our reduced synchronization, RepCut achieves superlinear speedups on large complex designs. Such speedups make parallel RTL simulation attractive in production as both individual simulations as well as overall simulation throughput benefit. Our analysis with performance counters reveals that parallel RTL simulation becomes a processor frontend-bound application once synchronization is largely optimized. This is a productive outcome, as this issue is more tractable to address, whether by more parallelization (to reduce the code footprint per core) or by building more capable cores. Further simulation efficiency improvements will be most appreciable in terms of decreasing the size of the design necessary for parallel benefit.

Our key insight is to generate independent partitions at the cost of duplicating overlapping areas. We quantify the replication cost and demonstrate it is not only acceptable, but we are able to achieve large speedups in spite of it. Furthermore, our method of using a proxy problem ensures the hypergraph partitioner attempts to minimize the amount of replication. Our heuristic-based execution time estimates noticeably improve RepCut's partition balance. As demonstrated by our evaluation, our method is applicable to RTL simulation and may easily be applied to related problems such as gate-level simulation.

ACKNOWLEDGMENTS

We would like to thank the anonymous reviewers for their insightful feedback on this work. This material is based upon work supported by, or in part by, the Army Research Laboratory and the Army Research Office under contract/grant W911NF1910466.

A ARTIFACT APPENDIX

A.1 Abstract

This artifact contains the source code for RepCut, as well as other open source projects that are required to reproduce the results in the paper. We include Verilator 4.226 as a baseline. In addition, this artifact also contains scripts and a Makefile to compile and run the generated simulators, as well as to reproduce every figure and table from experimental data. This artifact provides a *quick compilation*, which compiles and run a single design (*DualSmallBoomConfig*) with 1, 2, 4, 6 and 8 threads, as well as *full compilation* that compiles everything and reproduces figures (Figure 2a, 2b, 6, 7, 8, 9, 11, 12, 13, 14) and tables (Table 1, 3) in this work. Please note, a full compilation takes a significant amount of time and compute resources, as it compiles many variants (every thread count by 2), and like most parallel scaling experiments, much of the time is spent on the low thread count instances.

A.2 Artifact check-list (meta-information)

- **Compilation:** Clang++, Java (clang 14, OpenJDK 11 recommended).
- **Run-time environment:** Linux. This artifact is tested under Ubuntu 20.04 LTS and Ubuntu 22.04 LTS (Table 2).
- **Hardware:** Multi-core x86 platform is required. Please leave sufficient memory for compilation (See later). To fully reproduce result in this paper, we recommend platforms that have at least 48 cores.
- **How much disk space required (approximately)?:** 60 GB for a quick compilation, 500 GB for a full compilation.
- **How much time is needed to prepare workflow (approximately)?:** 2 hours for a quick compilation, 54 hours for a full compilation.
- **How much time is needed to complete experiments (approximately)?:** 5 minutes for a quick compilation, 38 hours for a full compilation.
- **Publicly available?:** Yes, GitHub[5].
- **Code licenses (if publicly available)?:** BSD
- **Archived (provide DOI)?:** Yes, https://doi.org/10.5281/zenodo.7621336

A.3 Description

A.3.1 How to access. The artifact can be downloaded from Zenodo[6]. A docker image[7] is also avaiable with all dependencies installed. Extracting this artifact requires 2.1 GB disk space. More disk space is needed to compile and run the artifact.

A.3.2 Hardware dependencies. No special hardware is required. However, we recommend running this artifact on a system with at least 8 cores to observe parallel speedup. Please also make sure have sufficient memory to compile this artifact.

A.3.3 Software dependencies. Please check `RepCut-AE/README.md`, Section 1.

A.4 Installation

Compilation script is provided as `RepCut-AE/Makefile`. Please read `RepCut-AE/README.md` Section 4 for more information.

A.5 Experiment workflow

Experiment script is provided as `RepCut-AE/Makefile`. Please read `RepCut-AE/README.md` for more information.

A.6 Evaluation and expected results

Please read `RepCut-AE/README.md` for more information.

A.7 Experiment customization

Please read `RepCut-AE/README.md` Section 4 for more information. Changing the number of threads may break the plotting scripts.

REFERENCES

[1] Tariq B. Ahmad and Maciej Ciesielski. 2014. Parallel Multi-core Verilog HDL Simulation Using Domain Partitioning. In *IEEE Computer Society Annual Symposium on VLSI*. 619–624. https://doi.org/10.1109/ISVLSI.2014.47

[2] Tariq Bashir Ahmad, Namdo Kim, Byeong Min, Apurva Kalia, Maciej Ciesielski, and Seiyang Yang. 2012. Scalable parallel event-driven HDL simulation for multi-cores. In *International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD)*. 217–220. https://doi.org/10.1109/SMACD.2012.6339456

[3] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Daniel Dabbelt, John Hauser, Adam Izraelevitz, et al. 2016. The Rocket Chip Generator. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2016-17* 4 (2016).

[4] Krste Asanović, David A Patterson, and Christopher Celio. 2015. The Berkeley Out-of-Order Machine (BOOM): An Industry-Competitive, Synthesizable, Parameterized RISC-V Processor. *EECS Department, University of California, Berkeley, Tech. Rep. UCB/EECS-2015-167* (2015).

[5] https://github.com/ucsc-vama/essent/tree/repcut
[6] https://doi.org/10.5281/zenodo.7621336
[7] https://hub.docker.com/r/haooozi/repcut-ae

[5] David A Bader, Henning Meyerhenke, Peter Sanders, and Dorothea Wagner. 2013. *Graph partitioning and graph clustering*. Vol. 588. American Mathematical Society Providence, RI. https://doi.org/10.1090/conm/588

[6] Mary L. Bailey, Jack V. Briner, and Roger D. Chamberlain. 1994. Parallel Logic Simulation of VLSI Systems. *Comput. Surveys* 26, 3 (Sep. 1994), 255–294. https://doi.org/10.1145/185403.185424

[7] Grey Ballard, James Demmel, Olga Holtz, and Oded Schwartz. 2011. Minimizing communication in numerical linear algebra. *SIAM J. Matrix Anal. Appl.* 32, 3 (2011), 866–901. https://doi.org/doi.org/10.1137/090769156

[8] Pete Bannon, Ganesh Venkataramanan, Debjit Das Sarma, and Emil Talpes. 2019. Computer and Redundancy Solution for the Full Self-Driving Computer. In *IEEE Hot Chips 31 Symposium (HCS)*. https://doi.org/10.1109/HOTCHIPS.2019.8875645

[9] Scott Beamer. 2020. A Case for Accelerating Software RTL Simulation. *IEEE Micro* 40, 4 (2020), 112–119. https://doi.org/10.1109/MM.2020.2997639

[10] Scott Beamer and David Donofrio. 2020. Efficiently Exploiting Low Activity Factors to Accelerate RTL Simulation. In *Design Automation Conference (DAC)*. https://doi.org/10.1109/DAC18072.2020.9218632

[11] Scott Beamer, Thomas Nijssen, Krishna Pandian, and Kyle Zhang. 2021. ESSENT: A High-Performance RTL Simulator. In *Workshop on Open-Source EDA Technology (WOSET), International Conference on Computer Aided Design (ICCAD)*.

[12] William J Bolosky and Michael L Scott. 1993. False Sharing and its Effect on Shared Memory Performance. In *Symposium on Experimental Distributed and Multiprocessor Systems*. 57–71.

[13] Aydın Buluç, Henning Meyerhenke, Ilya Safro, Peter Sanders, and Christian Schulz. 2016. Recent Advances in Graph Partitioning. (2016), 117–158. https://doi.org/10.1007/978-3-319-49487-6_4

[14] Ümit Çatalyürek and Cevdet Aykanat. 2011. *PaToH (Partitioning Tool for Hypergraphs)*. Springer US, Boston, MA, 1479–1487. https://doi.org/10.1007/978-0-387-09766-4_93

[15] K.M. Chandy and J. Misra. 1979. Distributed Simulation: A Case Study in Design and Verification of Distributed Programs. *IEEE Transactions on Software Engineering* SE-5, 5 (1979), 440–452. https://doi.org/10.1109/TSE.1979.230182

[16] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. 2009. Event-Driven Gate-Level Simulation with GP-GPUs. In *Design Automation Conference (DAC)* (San Francisco, California). 557–562. https://doi.org/10.1145/1629911.1630056

[17] Debapriya Chatterjee, Andrew Deorio, and Valeria Bertacco. 2011. Gate-Level Simulation with GPU Computing. *ACM Trans. Des. Autom. Electron. Syst.* 16, 3, Article 30 (Jun. 2011). https://doi.org/10.1145/1970353.1970363

[18] 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 *Conference on Programming Language Design and Implementation (PLDI)* (Philadelphia, PA, USA). 752–768. https://doi.org/10.1145/3192366.3192404

[19] Joseph E Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. Powergraph: Distributed Graph-Parallel Computation on Natural Graphs. In *Symposium on Operating Systems Design and Implementation (OSDI)*. 17–30.

[20] J. P. Grossman, Brian Towles, Joseph A. Bank, and David E. Shaw. 2013. The Role of Cascade, a Cycle-Based Simulation Infrastructure, in Designing the Anton Special-Purpose Supercomputers. In *Design Automation Conference (DAC)* (Austin, Texas). Article 122, 9 pages. https://doi.org/10.1145/2463209.2488884

[21] Loc Hoang, Roshan Dathathri, Gurbinder Gill, and Keshav Pingali. 2021. CuSP: A Customizable Streaming Edge Partitioner for Distributed Graph Analytics. *SIGOPS Operating Systems Review* 55, 1 (June 2021), 47–60. https://doi.org/10.1145/3469379.3469385

[22] Randall L. Hyde and Brett D. Fleisch. 1996. An Analysis of Degenerate Sharing and False Coherence. *J. Parallel and Distrib. Comput.* 34, 2 (1996), 183–195. https://doi.org/10.1006/jpdc.1996.0054

[23] 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 *International Conference on Computer-Aided Design (ICCAD)*. 209–216. https://doi.org/10.1109/ICCAD.2017.8203780

[24] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel Emer, and Daniel Sanchez. 2015. A scalable architecture for ordered parallelism. In *International Symposium on Microarchitecture (MICRO)*. 228–241. https://doi.org/10.1145/2830772.2830777

[25] George Karypis. 1998. hMETIS 1.5: A Hypergraph Partitioning Package. *http://www. cs. umn. edu/˜ metis* (1998).

[26] George Karypis and Vipin Kumar. 1998. A Fast and High Quality Multilevel Scheme for Partitioning Irregular Graphs. *SIAM Journal on Scientific Computing* 20, 1 (1998), 359–392. https://doi.org/10.1137/S1064827595287997

[27] G. Karypis and V. Kumar. 1998. Multilevel Algorithms for Multi-Constraint Graph Partitioning. In *Conference on Supercomputing (SC)*. 28–28. https://doi.org/10.1109/SC.1998.10018

[28] Dusung Kim, Maciej Ciesielski, and Seiyang Yang. 2011. A New Distributed Event-Driven Gate-Level HDL Simulation by Accurate Prediction. In *Design, Automation & Test in Europe (DATE)*. https://doi.org/10.1109/DATE.2011.5763280

[29] Dusung Kim, Maciej Ciesielski, and Seiyang Yang. 2013. MULTES: Multilevel Temporal-Parallel Event-Driven Simulation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 32, 6 (2013), 845–857. https://doi.org/10.1109/TCAD.2013.2237769

[30] Alexey Kupriyanov, Frank Hannig, and Jürgen Teich. 2004. High-Speed Event-Driven RTL Compiled Simulation. In *Computer Systems: Architectures, Modeling, and Simulation*, Andy D. Pimentel and Stamatis Vassiliadis (Eds.). Springer Berlin Heidelberg, 519–529. https://doi.org/10.1007/978-3-540-27776-7_53

[31] Luciano Lavagno, Igor L Markov, Grant Martin, and Louis K Scheffer. 2017. *Electronic Design Automation for IC System Design, Verification, and Testing*. CRC Press. https://doi.org/10.1201/b19569

[32] Tun Li, Yang Guo, and Si-Kun Li. 2004. Design and Implementation of a Parallel Verilog Simulator: PVSim. In *International Conference on VLSI Design*. 329–334. https://doi.org/10.1109/ICVD.2004.1260944

[33] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. 2023. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In *International Conference on Parallel Processing (ICPP)* (Bordeaux, France). Article 88. https://doi.org/10.1145/3545008.3545091

[34] Jonas Oberhauser, Rafael Lourenco de Lima Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: Push-Button Verification and Optimization for Synchronization Primitives on Weak Memory Models. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 530–545. https://doi.org/10.1145/3445814.3446748

[35] Clément Pit-Claudel, Thomas Bourgeat, Stella Lau, Arvind, and Adam Chlipala. 2021. Effective Simulation and Debugging for a High-Level Hardware Language Using Software Compilers. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*. 789–803. https://doi.org/10.1145/3445814.3446720

[36] Hao Qian and Yangdong Deng. 2011. Accelerating RTL Simulation with GPUs. In *International Conference on Computer-Aided Design (ICCAD)*. 687–693. https://doi.org/10.1109/ICCAD.2011.6105404

[37] Alberto Ros and Stefanos Kaxiras. 2015. Callback: Efficient Synchronization without Invalidation with a Directory Just for Spin-Waiting. In *International Symposium on Computer Architecture (ISCA)* (Portland, Oregon). 427–438. https://doi.org/10.1145/2749469.2750405

[38] Vivek Sarkar. 1987. *Partitioning and Scheduling Parallel Programs for Execution on Multiprocessors*. Ph. D. Dissertation. Stanford University.

[39] Shimpei Sato, Ryohei Kobayashi, and Kenji Kise. 2018. ArchHDL: A Novel Hardware RTL Modeling and High-Speed Simulation Environment. *IEICE Transactions on Information and Systems* E101.D, 2 (2018), 344–353. https://doi.org/10.1587/transinf.2017RCP0012

[40] G. Saucier, D. Brasen, and J.P. Hiol. 1993. Partitioning with Cone Structures. In *International Conference on Computer Aided Design (ICCAD)*. 236–239. https://doi.org/10.1109/ICCAD.1993.580063

[41] Eric Schkufza, Michael Wei, and Christopher J. Rossbach. 2019. Just-In-Time Compilation for Verilog: A New Technique for Improving the FPGA Programming Experience. In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)* (Providence, RI, USA). 271–286. https://doi.org/10.1145/3297858.3304010

[42] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. 2022. High-Quality Hypergraph Partitioning. *ACM Journal Experimental Algorithmics* (Apr. 2022). https://doi.org/10.1145/3529090

[43] Wilson Snyder. 2017. Verilator: Speedy Reference Models, Direct from RTL. *Presentation to University of Massachusetts Amherst* (2017). https://www.veripool.org/papers/Verilator_Modeling_UMass2017b_pres.pdf

[44] Wilson Snyder. 2018. Verilator 4.0 – Open Simulation Goes Multithreaded. In *Open Source Digital Design Conference (ORConf)*.

[45] Chris Walshaw and Mark Cross. 2007. JOSTLE: Parallel Multilevel Graph-Partitioning Software – An Overview. *Mesh Partitioning Techniques and Domain Decomposition Techniques* 10 (2007), 27–58. https://doi.org/10.4203/csets.17.2

[46] Stephen Williams and Michael Baxter. 2002. Icarus Verilog: Open-Source Verilog More than a Year Later. *Linux Journal* 99 (2002), 3.

[47] Seiyang Yang, Jaehoon Han, Doowhan Kwak, Namdo Kim, Daeseo Cha, Junhyuck Park, and Jay Kim. 2014. Predictive Parallel Event-Driven HDL Simulation with a New Powerful Prediction Strategy. In *Design, Automation & Test in Europe (DATE)*. https://doi.org/10.7873/DATE.2014.329

[48] Yanqing Zhang, Haoxing Ren, and Brucek Khailany. 2020. Opportunities for RTL and Gate Level Simulation Using GPUs. In *International Conference on Computer-Aided Design (ICCAD)*. Article 166. https://doi.org/10.1145/3400302.3415773

[49] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanović. 2020. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. In *Workshop on Computer Architecture Research with RISC-V (CARRV), International Symposium on Computer Architecture (ISCA)*.

Received 2022-10-20; accepted 2023-01-19