

PARENDI: Thousand-Way Parallel RTL Simulation

Mahyar Emami mahyar.emami@epfl.ch EPFL Lausanne, Switzerland Thomas Bourgeat thomas.bourgeat@epfl.ch EPFL Lausanne, Switzerland James R. Larus james.larus@epfl.ch EPFL Lausanne, Switzerland

Abstract

Hardware development critically depends on cycle-accurate RTL simulation. However, as chip complexity increases, conventional single-threaded simulation becomes impractical due to stagnant single-core performance.

PARENDI is an RTL simulator that addresses this challenge by exploiting the abundant fine-grained parallelism inherent in RTL simulation and efficiently mapping it onto the massively parallel Graphcore IPU (Intelligence Processing Unit) architecture. PARENDI scales up to 5888 cores on 4 Graphcore IPU sockets. It allows us to run large RTL designs up to 4× faster than the most powerful state-of-the-art x64 multicore systems.

To achieve this performance, we developed new partitioning and compilation techniques and carefully quantified the synchronization, communication, and computation costs of parallel RTL simulation: The paper comprehensively analyzes these factors and details the strategies that PARENDI uses to optimize them.

CCS Concepts: • Hardware \rightarrow Simulation and emulation; Testing with distributed and parallel systems; Hardware description languages and compilation; • Computing methodologies \rightarrow Massively parallel and high-performance simulations; Distributed simulation; Simulation evaluation; Discrete-event simulation; • Computer systems organization \rightarrow Multiple instruction, multiple data; • Software and its engineering \rightarrow Compilers.

Keywords: Bulk-synchronous Parallel, RTL Simulation, Cycleaccurate, Partitioning, Submodular Load Balancing

ACM Reference Format:

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 (ASPLOS '25), March 30-April 3, 2025, Rotterdam, Netherlands. ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3676641.3716010

This work is licensed under a Creative Commons Attribution 4.0 International License. *ASPLOS '25, Rotterdam, Netherlands* © 2025 Copyright held by the owner/author(s). ACM ISBN 979-8-4007-1079-7/25/03 https://doi.org/10.1145/3676641.3716010

1 Introduction

Hardware developers spend as much as a quarter of their time *running* simulations [22, 23]. Cycle-accurate RTL (Register Transfer Level) simulation is an essential tool for debugging and validating an ASIC or FPGA design, but it can be time-consuming to run.

Unfortunately, its slow speed hampers the design process. Fig. 1 shows the increasing gap between single-thread performance and package transistor count. It shows that the single-thread simulation of the new generations of chips on existing computers is becoming less feasible.



Figure 1. Chip growth and single-thread performance [44]. The dashed line predicts the core count, assuming linear scaling, necessary to simulate a state-of-the-art chip at the same rate as in 2006.

One appealing solution is to exploit the inherent parallelism of RTL designs by simulating them on parallel computers [21, 35, 36, 58]. However, Fig. 1 shows that simulating today's chips at the same rate as we simulated chips in 2006 requires parallel simulation that can utilize hundreds or thousands of cores.

This paper presents a practical solution to the problem of parallelizing RTL simulation of large (e.g., 100-core SoCs) across a few *thousand* cores. To demonstrate, we build an RTL simulator running on the Graphcore IPU [4, 29], a 1472-core chip that is the building block of parallel machine-learning systems. Although the IPU is not well known, its architecture contains many features-high core count, fast synchronization, and high internal and external bandwidth-that are especially well-suited for large-scale RTL simulation.

A parallel RTL simulator on a massively parallel machine must balance synchronization, communication, and computation. We analyze these factors to clarify their relations. These axes are not independent, which makes it challenging to partition an RTL simulation across many cores.

We use these experimental insights to build PARENDI¹, the first scalable, multi-thousand-way parallel RTL simulator.

¹PARENDI is the female Zoroastrian angel of abundance.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands



Figure 2. The IPU processor and M2000 server blade.

PARENDI is open-source, facilitating further research. For large designs, PARENDI demonstrates performance and efficiency gains across multiple dimensions. It runs up to 4× faster than multithreaded Verilator (the fastest RTL simulator). In nightly cloud testing scenarios, PARENDI could reduce costs by more than 2×. It also compiles large designs 12× faster and uses 18× less memory than Verilator.

The contributions of this work are:

- A quantitative study of massively parallel RTL simulation.
- A new communication- and duplication-aware compilation strategy for large-scale RTL simulation.
- Implementation of PARENDI, the first open-source² RTL simulator that runs on thousands of cores.
- An evaluation of PARENDI on a Graphcore system with 5888 cores that shows that it cost-effectively exceeds Verilator's performance on a high-end x64 system by up 4.0×.

The paper is organized as follows: §2 provides background on the IPU. §3 details the parallel RTL simulation strategy. §4 presents a high-level performance study. §5 outlines PARENDI's design. §6 evaluates PARENDI on IPU and Verilator on x64. §7 reviews related work. §8 concludes.

2 Graphcore IPU

A Graphcore IPU is a single package containing 1472 tiles (physical cores) connected by a high-bandwidth network (IPU exchange) with 11 TiB/s all-to-all bandwidth [4]. The IPU is a multiple-instruction, multi-data (MIMD) architecture in which each tile runs an independent instruction stream. By contrast, GPUs use SIMD or SIMT execution, where groups of threads (Warps) simultaneously execute the same instruction on different data. The IPU is a message-passing machine. Each tile can only access its private memory and must explicitly communicate through the exchange fabric. An IPU has a total on-chip memory of approximately 900 MiB, with each tile having exclusive access to 624 KiB.

Fig. 2 displays a Graphcore M2000 IPU server, a 1U unit housing 4 IPUs with a 320 GiB/s exchange fabric, totaling 5888 tiles. Systems can scale to 16 or 64 IPUs using multiple boards. Our study utilized a single board. An IPU system (with any number of IPUs) is programmed in C++ using the bulk synchronous parallel (**BSP**) [56] programming model, supported by the *poplar* SDK [6] and clangbased C++ compiler *popc*. The IPU directly supports BSP communication and synchronization. The following section describes BSP and how to apply it to RTL simulation.

3 Parallel RTL Simulation

Hardware description languages (HDL) like Verilog express digital sequential circuits. An HDL program contains stateful, clocked elements called registers interconnected by wires and stateless combinational logic. Register transfer level (RTL) is a set of clocked registers and combinational logic.

This paper considers cycle-accurate RTL simulation, where combinational logic has zero delay. Also, we only use fullcycle (activity-oblivious) simulation, which evaluates an entire circuit at each RTL cycle. The alternative is event-driven (activity-aware) simulation. In general, full-cycle simulators perform better—sometimes by orders of magnitude—because tracking value changes in RTL is expensive [14].

3.1 Shared-Memory Simulation

Parallel RTL simulation poses challenges for cache-coherent, shared-memory computers. First, fine-grained parallelism requires frequent synchronization, which is costly on a sharedmemory multiprocessor [21, 58]. Second, the RTL tasks perform fine-grained, point-to-point communication. In an RTL design, a task may communicate only a few bytes of data to the known cores computing its neighbors in the RTL graph, but all transfers go through the last-level cache (LLC). Finally, when compiled into code, RTL can have a high data and instruction reuse distance, which makes caches perform poorly. Most data items and instructions are accessed once per simulated RTL cycle, which might span millions of machine cycles. When large designs do not fit the caches, these memory references incur cache misses each RTL cycle [64].

3.2 BSP RTL Simulation

To alleviate the first two problems, we use Valiant's bulk synchronous parallel (**BSP**) [56] model. BSP is a messagepassing model alternating two phases: (i) computation and (ii) communication. In the computation phase, parallel *processes* run, reading shared values and modifying only private data. Computation ends at a barrier. Then, the communication phase transfers newly computed private values to consuming processes. Communication also ends at a barrier, after which the next computation phase begins. The appeal of BSP is that it reduces synchronization to two global barriers per RTL clock cycle. ³

²https://github.com/epfl-vlsc/parendi

³Other parallel RTL simulation systems utilize this computation model [16–18, 21, 40, 57, 58], but it was only recently called out as *BSP* [21].



Figure 3. BSP Simulation of an RTL data dependence graph. The graph contains three fibers (f1, f2, f3), partitioned into two processes (p1, p2), running on two threads. a3 is duplicated. The run on the right shows the computation and communication phases, separated by barriers.

Fig. 3 contains a sample *data dependence graph* RTL circuit. This graph splits each RTL register into two values: a readonly value (*current*, at the leading clock edge) and a writeonly value (*next*, at the end). The *current* values at the top (e.g., read1) are fed into stateless combinational logic (circles) that computes the *next* register values (e.g., write1).

The dashed lines in Fig. 3 *partition* the graph into BSP processes p1 and p2. Each process reads a set of RTL registers and computes new values for one or more (e.g., write1 in p2). Since we only communicate register values at the end of the computation phase, processes may need to duplicate intermediate computations (e.g., a3 is in both p1 and p2).

The right side of Fig. 3 shows the evaluation of the example. The parallel processes synchronize at a barrier (dashed vertical lines). The first barrier marks the end of the computation, after which we exchange *next* values (write1, write2, and write3) and update the *current* values accordingly (read1, read2, and read3). We finish the communication with a barrier and conclude one RTL simulation cycle.

A parallel RTL compiler partitions the data dependence graph to minimize the time spent running a simulation cycle. p1 and p2 are not the only possible partitions of Fig. 3. We call the atoms of BSP simulation *fibers*. A fiber is the smallest set of operations that uniquely produces the *next* value of a single register. Fig. 3 contains three registers and three fibers f1 to f3, partitioned into p1 = {f1} and p2 = {f2, f3}.

It is worth noting that BSP is only one of many possible parallel simulation techniques. In BSP, nodes a1 and a2 belong to the same fiber, so they run one after the other. We could consider a fine-grain parallel execution in which individual nodes in Fig. 3 evaluate in parallel, with point-to-point synchronization. Verilator [48–50] uses this approach [46]. The advantage of fine-grained parallelism is avoiding duplicated work at the cost of more synchronization.

4 Analysis of BSP RTL Simulation

We now measure and analyze the principal performance factors in BSP RTL simulation using small benchmarks on the M2000 quad-IPU system and an Intel Xeon Gold 6348 56-core dual-socket processor (see Table 2 for details).

Parallel performance depends on synchronization, communication, and computation costs. In BSP, the sum of the three is the time to simulate one cycle, so the simulation rate (in a thousand RTL cycles per second or kHz) is

$$r_{cycle} = \frac{1}{t_{sync} + t_{comm} + t_{comp}},\tag{1}$$

where t_{sync} , t_{comm} , and t_{comp} are per RTL cycle synchronization, communication, and computation latencies. Reducing this sum increases the simulation rate. Below, we explore how each term behaves as we seek to increase parallelism. Our analysis reveals salient architectural features of the IPU and x64, providing insight into compilation strategies.

4.1 Synchronization

BSP requires two global synchronizations per simulated clock cycle, so t_{sync} is the cost of two barriers. Therefore, t_{sync} is independent of the simulated design. However, since the cost of a barrier increases with parallelism, it depends on the number of hardware threads used for a simulation.

To explore the relationship between t_{sync} and performance, we simulate a set of pseudo-random number generators (PRNG), each performing three XORs and three shifts [37]. The simulated PRNGs are independent; so, $t_{comm} = 0$, but $t_{sync} > 0$ as we still need to synchronize with the RTL clock. Therefore, if t_{sync} is small compared to the computation cost, we expect to observe a near-constant simulation rate.

We simultaneously increase the number of PRNGs and computation units—tiles (IPU) or threads (x64). In each set of experiments, we keep the amount of work per tile (thread) constant. Note that each PRNG consists of one fiber, but we can sequentially execute multiple fibers on one tile (thread) to vary the computation-to-synchronization cost ratio.

We use the IPU's built-in barrier and sweep the tiles from 64 to 5888 by 64. On x64, we use a user-space (atomic fetchand-add) barrier, measuring from 1 to 56 threads. This type of barrier performed better than OpenMP's built-in, MCS, or sense barriers.

Fig. 4 shows the measured rate on the IPU (normalized to the rate with 64 tiles) and on x64 (normalized to the rate with one thread). Each line shows the performance with a fixed quantity of fibers per tile: 7, 56, and 448 on the IPU and 736, 5888, and 47104 on the x64. The total work with 5888 tiles on IPU is the same as 56 threads on x64. We normalized each experiment to itself as we are not comparing absolute simulation rates between the machines.

With 7 fibers per tile on the IPU, synchronization causes performance to fall by almost 50% as the number of tiles increases. Synchronization latency becomes less detrimental as the computation per tile increases (with 448 fibers, performance falls by a few percent). The cost of synchronization on x64 is high, even with many fibers per thread. With 736



Figure 4. IPU and x64 PRNG rates

fibers per thread, performance drops by more than 75%, and even with 47104 fibers per thread, performance falls by 25%. The IPU has a native hardware barrier that consumes only a few hundred IPU cycles. By contrast, x64 barrier synchronization requires expensive atomic memory accesses that could require a few thousand cycles with 56 threads.

Fig. 4 reveals a simple rule-of-thumb. Masking synchronization overhead on x64 requires hundreds of thousands of instructions per thread (each fiber is roughly 6 instructions), whereas on the IPU, a few thousand instructions adequately hide synchronization overhead. The IPU supports very finegrain parallelism, whereas the x64 does not.

4.2 Communication

Similar to t_{sync} , we expect t_{comm} to increase as we increase parallelism, as adding tiles (threads) means more values are communicated among tile (thread). Unlike synchronization, communication depends on the specifics of an RTL design and the partition of fibers among tiles (threads). We summarize these considerations into two parameters: bytes sent from each tile (*b*) and number of tiles in the simulation (*m*).

To first order, we might expect $t_{comm} = \frac{m \times b}{bw}$, where bw is the communication bandwidth and $m \times b$ is the total communication volume. Additional parallelism can increase communication latency if it increases the inter-tile (thread) volume. Therefore, at some point, the increase in t_{comm} could outweigh the benefits of spreading computation among more tiles (threads). Alternatively, t_{comm} could be almost independent of m and depends primarily on b. In this case, performance would increase *monotonically* with parallelism.

We found that communication within a single IPU appears to depend primarily on b, but communication between IPUs depends on $m \times b$. We demonstrate this with two experiments.

First, consider 2m tiles running on one IPU. We *randomly* partition the 2m tiles into two sets of m tiles and send a fixed number of bytes in both directions between the sets. The left plot in Fig. 5 shows the measured IPU cycle counts (averaged over 10 random bi-partitions). The cycle counts include t_{sync} as an exchange requires synchronization. The horizontal axis is the number of bytes each tile sends and receives (b). The vertical axis is the number of tiles (m). The on-chip t_{comm} increases only in the direction of increasing b as shown by the arrow in the left chart of Fig. 5.

Mahyar Emami, Thomas Bourgeat, and James R. Larus



Figure 5. Measured communication cycles on the IPU

In the second experiment, one tile in each pair resides on one IPU and the other on another IPU, so all traffic goes off-chip. The right chart in Fig. 5 reports the results. It shows a vastly different behavior: t_{comm} increases with increasing parallelism *and* bytes per tile as it depends on $m \times b$ (the diagonal arrow delineates the direction of change). Furthermore, the increase is more pronounced.

At the plots' darkest points, we consume 13% and 82% of the maximum measured communication on- and off-chip bandwidth, respectively (7.7 TiB/s and 107 GiB/s). The onchip experiment is far from saturating the bandwidth, so latency is insensitive to tile count. By contrast, the off-chip experiment runs near the fabrics's maximum bandwidth, so additional communication increases contention and latency.

In conclusion, the difference between these communication fabrics means that minimizing off-chip communication volume is a first-class concern when the traffic is large.

It is worth noting a limitation of Graphcore BSP communication. The IPU's exchange fabric is *statically scheduled*; hence, communication *must* start with a barrier to ensure all tiles are at the same point in execution. Unfortunately, this precludes optimization such as overlapping computation and communication or dynamic load balancing.

4.3 Computation

At first glance, optimizing t_{comp} is similar to the *multiprocessor independent task scheduling (makespan minimization)* problem [25]. In this classic problem, we consider a set of tasks (fibers) $F = \{f_1, ..., f_n\}$ with corresponding execution times $t_i, ..., t_n$. The goal is to schedule these tasks on *m* tiles (threads) to minimize the longest execution time across all tiles. This problem is NP-hard [55], but polynomial-time approximations exist [25, 45].

In the classical problem, tasks (fibers) have fixed execution times, independent of where they run. However, in RTL simulation, two fibers might compute a shared intermediate value (for example, value a3 in Fig. 3). Collocating these two fibers in the same process enables optimization. However, it complicates the partitioning problem. Each fiber consists of a set of computation nodes (the nodes in Fig. 3). If we



(a) Stragglers impose a lower bound on *t_{comp}*. Fibers sorted for visualization.



(b) Fiber computation cycles in pico, bitcon, and rocket.



(c) Reducing t_{comp} through parallel execution (base-2 log scale). Dashed lines show a perfect scaling.

Figure 6. Straggler fibers and performance scaling regions.

		Par	ENDI			Verilator on ix3									
pico		bitcoin		roc	ket	F	ico	bit	coin	rocket					
par. kHz		par. kHz		par.	kHz	par.	kHz	par.	kHz	par. kHz					
1	168.7	1	14.5	2	17.7	1	14141.7	1	537.4	1	220.3				
111	629.4	270	935.2	1211	93.3	2	490.4	2	232	2	99.2				

Table 1. Simulation rate in kHz. **par** is the tile- or threadcount used to achieve the rate in **kHz**. See Table 2 for the technical specification of ix3.

denote the execution time of a BSP process using $\tau(.)$, then a process made up of fibers f_i and f_j would have $\tau(f_i \cup f_j) = t_i + t_j - \tau(f_i \cap f_j)$ since we need to execute the shared code only once. Moreover, merging fibers eliminates communication (t_{comm}) so $\tau(f_i \cup f_j) \leq t_i + t_j - \tau(f_i \cap f_j)$. This is a submodular function, and this variant of the scheduling problem is called submodular load balancing (SLB). SLB is inherently more complex than classic scheduling and challenging to get even modest approximation guarantees ($\sqrt{n/loq(n)}$ [51]).

In the trivial case, when fewer tasks exist than tiles (threads) $(n \le m)$, the optimal solution is to assign a fiber to each tile (thread). It is impossible to improve t_{comp} beyond max_i t_i as the slowest fiber (the straggler) bounds t_{comp} from below. Encountering this bound on x64 hardware is unlikely: even relatively small designs have a few hundred fibers, an order of magnitude more than available cores. However, a single IPU chip has 1472 tiles, sufficient for small to medium-sized designs so that a straggler can limit IPU performance.

Fig. 6a depicts the SLB problem: mapping fibers to tiles results in a linear region in which t_{comp} falls almost linearly with additional tiles. The benefits become less significant, and we eventually plateau at max_i t_i (with sufficient tiles) with m_{crit} as the minimum tiles needed to this point. To maximize the simulation rate, we only need m_{crit} tiles; having more would not help as the straggler is the limit.

To put this into perspective, consider three small RTL designs: (1) a multi-cycle RISC-V pico core [42], (2) a bitcoin miner [5], and (3) a *small* rocket pipelined RISC-V core [11]. These small designs contain more fibers than x64 systems cores: 111, 270, and 1211 fibers, respectively. We run each benchmark using PARENDI, described later in §5.

Fig. 6b shows fiber computation latency (t_i for each f_i) of the three benchmarks (in IPU machine cycles). Fig. 6c illustrates the corresponding scheduled execution times with the dashed diagonal representing a perfect linear reduction. We normalize machine cycle counts to the minimum parallel execution: 1 tile in pico and bitcoin, 2 tiles in rocket (a single tile cannot hold sufficient code and state for rocket).

First, t_{comp} in Fig. 6c follows the trend of Fig. 6a: imbalanced fibers yield a small linear scaling region. pico is the most imbalanced and settles to a final t_{comp} extremely quickly. rocket is slightly more scalable but bitcoin performs the best as its fibers are roughly balanced. Second, t_{sync} and t_{comm} increase with additional tiles. However, the t_{comp} reduction is always larger, so the execution cost decreases.

Table 1 compares the wall-clock simulation rate of the IPU, using PARENDI, against an Intel Xeon 6348 (see Table 2 for details), using Verilator. We show the simulation rate using a single tile (except for rocket, which needs more memory than available in one tile) and the maximum number of tiles, where we assign one fiber per tile. For x64, we report the single-thread and best multi-thread performance.

None of the three small benchmarks show any speedup on x64 from parallelism since the synchronization cost is too high (see §4.1). These results do not mean that Verilator cannot speed up RTL simulation. In §6, we show that Verilator does an excellent job of parallelizing code. However, our analysis of t_{sync} shows that a simulated design on the x64 must be large enough to mask synchronization overhead, and these three benchmarks need to be bigger.

Verilator's inability to scale these three benchmarks supports our claim that the straggler fiber is not a performance limit on x64 as synchronization latency dominates. On the other hand, stragglers are a fundamental concern for PARENDI for small designs. Table 1 shows that PARENDI's *parallel* performance does not manage to even match Verilator's *single-thread* performance for pico and rocket, despite modest gains from parallelism. PARENDI runs bitcoin using a single-tile at 14.5 kHz, far from Verilator's single-thread performance (537 kHz). However, with 270 tiles, PARENDI runs bitcoin at 935.2 kHz, faster than Verilator's single- and multi-thread performance. Lastly, note that single-tile execution of pico and bitcoin on the IPU are approximately $84 \times$ and $37 \times$ slower than x64. Consequently, the IPU has to significantly scale RTL simulation to reach Verilator's single-thread performance.

5 PARENDI Compiler

PARENDI is a Verilog compiler for the IPU systems. It is derived from Verilator to take advantage of its optimizations and maturity. However, PARENDI contains significant changes that target the IPU (message-passing) rather than the x64 (shared memory). PARENDI also includes new scheduling and partitioning passes and IPU-specific optimizations.

PARENDI generates a C++ BSP program that uses Graphcore's *poplar* programming framework. The code defines each tile's computation and how the tiles communicate.

At a high level, PARENDI's primary responsibility is to partition RTL across the tiles of an IPU system. The user specifies the number of tiles. PARENDI tries to maximize the simulation rate by finding an appropriate partitioning of fibers to tiles. We briefly describe our partitioning strategy.

5.1 Partitioning

After generating a data dependence graph (see Fig. 3), we find the fibers by collecting the nodes that transitively feed into each sink node by crawling the graph in reverse.

Once we form the set of fibers, we must solve the SLB problem (§4.3). It is also crucial to recognize the interdependence between computation (t_{comp}) and communication latency (t_{comm}). In addition, each tile has a finite memory. Consequently, our partitioning algorithm must consider duplication, communication, and memory limitations.

We solve this problem in multiple steps, each pursuing a different goal. At each algorithm step, we *merge* fibers into *processes*. On the IPU, a process is a collection of fibers that will eventually run on a tile (see Fig. 3). There are four stages in our algorithm: (1) Reduce data memory footprint, (2) minimize off-chip communication, (3) reduce t_{comm} while keeping t_{comp} unchanged, (4) match the number of fibers to the available hardware.

In the first stage, we merge fibers that reference the same RTL array but only for *very* large arrays (e.g., \geq 128 KiB, tunable). We do this to save memory at the cost of possibly increasing t_{comp} , so it is only worthwhile for large arrays.⁴

The second stage minimizes off-chip exchanges if PARENDI is compiling for multiple IPUs. We do this by partitioning a hypergraph of fibers, in which hypernodes represent fibers, and hyperedges represent RTL registers. If two fibers access the same register (read or write), their corresponding hypernodes share a hyperedge. The hyperedge weights are the number of words in an RTL register, and hypernodes are unweighted. For *k* IPUs, we use the KaHyPar [47] library to find a *k*-way balanced partition of the hypergraph that minimizes the *cut* (hyperedges crossing the partitions). KaHyPar produces *k* roughly equally-sized sets of fibers. However, each set may contain more than 1472 fibers, so we must further merge fibers to fit the available tiles.

In the third stage, we *conservatively* merge the smallest fibers to reduce communication within each target IPU without increasing computation latency. Intuitively, we move right to the left in the right subplot of Fig. 6a towards m_{crit} . If we reduce the number of fibers to fit the tiles in an IPU without crossing m_{crit} , then we can use the optimal $t_{comp} = \max_i t_i$ and a pseudo-optimal t_{comm} . We create one process per fiber and estimate its execution cost. Recall from §4.3 that the cost of a process is submodular with respect to its fibers. We use a dense bitset data structure to represent duplication across fibers and efficiently compute intersection and union in the submodular cost function $-\tau(f_i \cup f_j) = t_i + t_j - \tau(f_i \cap f_j)$. Moreover, we use another bitset to track the memory usage after merging, accounting for deduplication.

In each iteration, we select the process with the shortest execution time and try to merge it with another with which it communicates, so long as their merged time does not surpass the worst existing execution time. If we cannot perform the merge because of overflowing memory or exceeding the straggler execution time, we consider merging the two smallest processes. If that fails, we skip the candidate process and move to the next one. We merge processes until we process all of them or reach the desired tile count.

The final stage only runs if the third stage fails to reach the desired tile count. We follow the same strategy as in the third stage but allow worst-case execution time to increase. At the end of this stage, the number of processes must fit the available hardware. Otherwise, the compilation fails because the design is too large to fit the hardware resources.

5.2 IPU-Specific Optimizations

PARENDI extends Verilator's optimizations with a few IPUspecific ones. We briefly describe the most important ones.

Differential exchange. RTL arrays are common in hardware designs, e.g., a register file or a cache bank. If a process reads an array, it needs a full copy on its tile. We avoid sending whole arrays by using static analysis to determine the number of updates to an array, though not their location or condition (e.g., a 2-port SRAM with byte-strobes). With this analysis, we only send the changes instead of an entire array.

⁴Doing so reduces the probability of running out of tile memory later. Consider a design with one 256 KiB array and four 64-KiB ones. Assume each array references 3 equal-size fibers: a_1, a_2, a_3 by the 256 KiB array, b_1, b_2, b_3 by the first 64-KiB array, ..., and e_1, e_2, e_3 by the fourth 64 KiB array. The goal is to create 3 balanced processes. Because there are more 64-KiB arrays, we could end up merging fibers that reference distinct arrays, e.g., $\{a_1, b_1, c_1, d_1, e_1\}$. Such a process needs 512 KiB of data memory, which exceeds the available on-tile data memory. However, if we premerge $\{a_1, a_2, a_3\}$, in the worst case we will end up with a process such as $\{a_1, a_2, a_3, b_1, c_1\}$ requiring 384 KiB. No other balanced process would exceed 256 KiB either.

Aggressive block splitting. We extend Verilator's V3Split pass, favoring parallelism over code bloat, to maximally split all clocked code blocks.

Aggressive inline. PARENDI ensures that the simulation program on the IPU is free of function calls. Inlining can increase code size and produce excessive instruction cache pressure on x64, especially in RTL simulation, where nearly every instruction executes only once per RTL cycle, except for functions invoked multiple times. An IPU tile has no instruction cache but a 624 KiB local memory, of which 200 KiB holds executable code. So, a single IPU chip has \approx 300 MiB of on-chip instruction memory space, which allows PARENDI to aggressively inline code.

5.3 Limitations

Currently, PARENDI only supports a subset of Verilator's clocking capabilities. PARENDI can simulate an RTL design with one top-level (at the testbench) clock and an arbitrary number of gated or divided ones.⁵ PARENDI supports only a Verilog test driver, whereas Verilator allows both C++ and Verilog drivers. We do this for pragmatic reasons: host interactions are costly on the IPU, and a C++ testbench interacts at every simulated cycle, which would be impractically slow on the IPU.⁶ PARENDI may fail to compile very large design whose code and state exceed the on-chip memory capacity of an IPU board (≈4×900 MiB). Verilator could perhaps compile and run such massive designs, albeit, very slowly. PARENDI's philosophy is to scale out and use more IPUs for larger designs. That said, this work explored this path only up to 4 IPUs. Scaling simulation to 16, 64, or even 256 IPUs (225 GiB of SRAM) is left for future work. Additionally, a single IPU tile has about 400 KiB of data memory. So, if a design contains a single Verilog array larger than this amount, compilation fails. Such large Verilog arrays are unlikely to appear in reasonably real silicon since large SRAMs in RTL designs (e.g., caches) are banked into much smaller arrays (e.g., 64-KiB banks). However, these arrays might exist in non-synthesizable test benches. Currently, users have to manually split these large arrays into smaller ones. Finally, PARENDI's compilation fails if the design has a combinational loop.

6 Evaluation

We use the following benchmarks to evaluate PARENDI:

- mc [54] is stock option price predictor.
- vta [39] is an ML accelerator. We configure vta with BlockIn/Out=64 (larger than the default FPGA configuration) to expose more parallelism.
- srN is a N × N Constellation [62] mesh NoC consisting of N × N – 3 small Rocket cores [11] (64-bit, no FPU, no VM),

generated by the Chipyard [10] SoC generator (3 nodes connect to uncore). We changed N from 2 to 15.

 1rN is similar to srN, but we use large cores with an FPU and VM. We changed N from 2 to 10.

Note that srN differs from rocket in §4.3 as the latter is bus-based, whereas srN uses a NoC. These benchmarks resemble contemporary chip designs, including accelerators and multicore systems. Varying the mesh size in srN and 1rN explores PARENDI's performance on larger chips. Using a generic gate library, we estimated sr2 and 1r2 to have about 200 and 320 thousand gates, respectively, while sr15 and 1r10 have about 20 million gates (excluding SRAMs). Due to a bug in *popc*, we could not evaluate BOOM [63].

We wrap all benchmarks with simple Verilog drivers, without DPI calls.⁷ Chipyard's default simulation flow heavily uses DPI calls to connect the simulation to services provided by a software RISC-V front-end server. By avoiding non-RTL software, we ensure that our evaluation of both PARENDI and Verilator does not include extraneous performance influences from the simulator and front-end communications.

Baseline. Parallel Verilator is our baseline. Other possible baselines are research artifacts that also explore parallelism. Verilog is a complex language, so academic works, including ours, make concessions and focus on techniques rather than full language coverage and robust implementation. These systems, unfortunately, cannot run all benchmarks (§7), so using Verilator permits a fuller evaluation.

Evaluation Setup. Table 2 summarizes the hardware for our evaluation. For Verilator, we use two modern data center computers: ae4 is the latest generation AMD server with large caches and a high core count. The 64 cores in one socket are constructed from *chiplets* [1] containing 8 cores. ix3 is a recent Intel server with no chiplets, less cache, and fewer cores. For PARENDI, we use a 4-IPU M2000 server⁸.

We use Verilator v5.006 (PARENDI is forked from this version) with all optimizations enabled (-03). To find each design's best simulation performance on PARENDI, we consider 1472, 2944, 4416, and 5888 tiles (1, 2, 3, and 4 IPUs, respectively). On Verilator, we measured each design from 2 to 32 threads (step size of 2) because Verilator takes a long time to generate multi-threaded code for the larger designs. Table 2 reports the compilation time and memory usage.

6.1 PARENDI Vs. Verilator

Fig. 7 reports PARENDI's speedup compared with Verilator. Overall, PARENDI outperforms Verilator. The geometric mean speedups are 2.81 and 2.75 over ix3 and ae4. Table 3 details the performance of each platform. We also report size metrics for each benchmark: number of data dependence graph nodes, fibers, x64 instructions to simulate one RTL cycle on

⁵Other work handle a single clock without any driven ones [20, 21, 58]. ⁶PARENDI has experimental support for DPI to interface with C++.

⁷PLI calls such as \$readmemh, \$display, \$plusargs still exist.

⁸The M2000 is not the fastest IPU machine available. A newer BOW-2000 IPU clocks at 1.85 GHz (a 37% increase) with the same tile count [2].



Figure 7. IPU's speedup versus multi-thread Verilator.

Compiler	Name/Short	Cores	GHz	MiB	×	Date					
Verilator	EPYC 9554 / ae4 Xeon 6348 / ix3	64 28	3.75 3.5	2/128/256 2.2/35/42	2 2	Q4 2022 Q2 2021					
Parendi	M2000/ipu	1472	1.35	897	4	Q3 2020					
Ubuntu 20.04 popc 3.3 (clang 16.0.0) Verilator v5.006 g++ 10.5.0 PARENDI: tiles up to 5888 Verilator: threads up to 32											

C	compile	on	Intel	xeon	6132	1.5	118	Memo	ry

min / max	Compile time	Memory usage
Parendi	26s / 40m	335 MiB / 55 GiB
Verilator	3s / <mark>8h</mark>	223MiB / 1043 GiB

Table 2. Evaluation setup: **Cores** is the physical core count per socket. **MiB** is the cache capacity (L1/L2/L3) for x64 and the on-chip memory for the IPU. × is the number of sockets. We use **Short** names for brevity. We also report the min. and max. compilation time and compiler memory usage.

a single thread, and Verilator's code footprint. Furthermore, for multi-IPU points, we report the KiB size of the variables exchanged (actual exchange volume is higher due to fanout).

6.2 Verilator's Performance

Table 3 reports Verilator's best speedup relative to itself. Verilator benefits from parallelism when a design is large (up to $22 \times$ speedup). A few points are worth considering:

Synchronization. Fig. 8a shows that smaller designs see limited speedups. Per §4.1, we expected this behavior: synchronization cost outweighs the gains of parallelism.

Communication is non-uniform. From §4.1 and Fig. 4, we see that synchronization does not affect large designs. Fig. 8b shows Verilator achieves significant speedups for large designs. However, on ae4, speedups fade after 8 threads (chiplet boundary). On ix3, we see a significant drop after 28 threads (socket boundary). The increased communication latency across chip boundaries has a noticeable performance cost, and parallel simulators should be aware of it.

Architecture matters. Fig. 8c shows no clear advantage between the two x64 machines. In general, ae4 wins for smaller designs and ix3 for large ones. In some cases, ae4 shows superlinear improvement up to the chiplet boundary. Such gains are exciting but not uncommon in RTL simulation. Increasing cores means each core runs less code and accesses less data, reducing pressure on the local cache, which reduces cache misses and provides a performance bonus [13, 14, 21,



(a) Verilator's speedup diminishes quickly for smaller designs as synchronization is costly.



(b) Non-uniform communication (crossing chiplets or sockets) reduces speedups.



(c) ae4 and ix3 have different scaling profiles due to implementation differences.

Figure 8. Verilator's performance and scalability.



(a) Simulation speed on one IPU. We start at 184 tiles (1/8 of an IPU) and scale to a full IPU (1472 tiles).



(b) Breakdown of simulation time.

Figure 9. Single-IPU speedup and simulation time breakdown.



Figure 10. Performance scaling across multiple IPUs.

36, 58]. However, the superlinear gains disappear when the local caches cannot hold the working set or chip-cross costs diminish the value of increased parallelism.

6.3 PARENDI's Performance

On x64, synchronization and communication were the two causes for reduced performance. On the IPU, only off-chip communication is a bottleneck, similar to the x64's off-chiplet or -socket communication.

Single-IPU scaling. On x64, we cannot consistently use *all* cores to increase simulation speed since the synchronization or communication costs may limit parallelism gains. On a single IPU, performance monotonically increases with additional tiles. Fig. 9a shows the rate for three designs as a function of the fraction of the IPU. The IPU's limited local memory cannot fit a moderately large design on a single tile. So, we cannot compute speedup relative to a single tile. We use 184 tiles ($\frac{1}{8}$ of an IPU) as the baseline. This is a fundamentally different starting point than a single thread on x64, as it already uses significant parallelism. However, we still see improvements.⁹

Fig. 9b shows the breakdown of simulation time for each design. The vertical axis is normalized to $\frac{1}{8}$ of the IPU. Communication (t_{comm}) and synchronization (t_{sync}) remain roughly constant while computation time (t_{comp}) decreases with additional tiles. However, in sr3, the improvement in t_{comp} ends due to fiber imbalance (see §4.3). The IPU performance is non-decreasing because of its low-cost communication and synchronization, but, to achieve the best performance on x64, a hardware developer must select the parallelism for each design *and* machine (ix3 or ae4).

Multi-IPU scaling. Within one IPU, communication is relatively cheap, which facilitates scaling. However, communication and synchronization across IPU boundaries is expensive (§4.2). Therefore, preserving performance monotonicity across IPUs is challenging: crossing IPUs is similar to crossing chiplets or sockets since off-chip communication



Figure 11. Coping with increasing design size. The left axis shows the best simulation rate. The right axis shows the geomean speedup of PARENDI against Verilator (dashed lines).



Figure 12. Fiber imbalance allows us to keep the simulation rate constant despite increasing design size.

latency increases abruptly (see Fig. 5). However, non-uniform communication emerges much later on the IPU (after 1472 tiles rather than 8 or 28 threads). Fig. 10 shows the simulation speed across multiple IPUs. Even for very large designs, running at maximum parallelism may yield a poorer result so that fewer IPUs can produce marginal gains in some cases.

Improvements are also much smaller off-chip. Going from 1472 tiles to 5888 tiles $(4\times)$ in 1r9 improves performance by 60%. However, a 60% gain is still attractive: on x64, it is difficult to scale beyond 28 threads (ix3), but on the IPU, we increase performance to 5888 tiles (210× more parallel).

Performance resilience. PARENDI can strongly scale the simulation rate within and across IPUs. But can we can maintain a constant simulation rate as we scale the design size (weak scaling)? Fig. 11 shows the maximum simulation rate of PARENDI and Verilator as a function of mesh size in srN and lrN. Neither PARENDI nor Verilator can keep the simulation rate perfectly constant, but PARENDI is better. For instance, at the right of Fig. 11, there is a long period in which PARENDI simulates larger designs at the same rate. Verilator's rate slowly drops in this region, and the speedup (PARENDI vs. Verilator) increases.

While fiber imbalance severely limits the performance of a small or medium RTL design, limiting strong scaling, it actually *enables* better weak scaling. Fig. 12 shows how it happens. Consider an SoC with N cores and a sizable imbalance among its fibers, as shown on the left. If we double the size of the SoC, we double the number of fibers. Because only a small portion of fibers have significantly longer execution time, we can tolerate increasingly larger designs and keep the simulation rate constant using unused parallel resources (Fig. 12). However, the utilization of tiles starts to balance at some point, after which increasing the design size decreases the simulation rate.

⁹Fig. 9a shows that vta's performance remains flat between $\frac{4}{8}$ and $\frac{6}{8}$ of the IPU. Such *staircase* behavior is a characteristic of highly regular fine-grain parallelism where there are many equal-size straggler processes. Let us explain using a hypothetical but similar example with 12 equal-size fibers f_1 until f_{12} . Suppose at first, we run these fibers on 4 balanced processes: $p_1 = \{f_1, f_2, f_3\}, ..., p_4 = \{f_{10}, f_{11}, f_{12}\}$. Using 5 balanced processes, we will have: $p'_1 = \{f_1, f_2, f_3\}, ..., p'_3 = \{f_7, f_8\}, p'_4 = \{f_9, f_{10}\}, p'_5 = \{f_{11}, f_{12}\}$. In other words, increasing the number of processes, we will have: $p''_1 = \{f_1, f_2\}, ..., p_6'' = \{f_{11}, f_{12}\}$, i.e., a 33% improvement in execution time.

ASPLOS '25, March 30-April 3, 2025, Rotterdam, Netherlands

in	ix3				ae4				Parendi			Speedup							
Bett	st-kHz	mt-kHz	#T	gain	st-kHz	mt-kHz	#T	gain	kHz	#T	ix3	ae4	gmean	MiB	#I (M)	#N (K)	#F (K)	Int.	Ext.
vta	30.91	113.75	4	3.7	44.79	164.73	4	3.7	454.10	1472	3.99	2.76	3.32	1.5	0.17	23.5	6.0	28.7	_
mc	28.68	88.96	8	3.1	37.55	143.88	8	3.8	592.83	1472	6.66	4.12	5.24	1.0	0.15	26.9	7.5	24.2	_
sr2	123.40	76.22	2	0.6	176.49	145.75	4	0.8	91.20	1472	0.74	0.52	0.62	1.2	0.06	12.7	2.8	12.8	_
sr3	20.95	40.95	8	2.0	28.71	77.66	8	2.7	83.95	1472	2.05	1.08	1.49	3.1	0.17	36.3	8.1	33.9	_
sr4	8.79	30.93	22	3.5	7.23	54.79	8	7.6	85.09	1472	2.75	1.55	2.07	5.5	0.32	68.2	15.3	63.5	_
sr5	5.23	24.34	26	4.6	4.26	40.09	8	<u>9.4</u>	84.28	1472	3.46	2.10	2.70	8.6	0.50	107.9	24.2	101.5	_
sr6	3.53	23.11	20	6.5	2.90	30.51	8	10.5	76.63	1472	3.32	2.51	2.89	12.2	0.72	156.0	35.0	145.4	_
sr7	2.47	18.83	28	7.6	2.10	22.72	8	10.8	71.33	2944	3.79	3.14	3.45	16.6	0.99	212.6	47.7	199.2	0.9
sr8	1.82	17.94	26	9.9	1.58	13.66	8	8.7	57.39	2944	3.20	4.20	3.66	21.6	1.29	277.3	62.3	259.0	1.1
sr9	1.37	15.56	28	11.4	1.22	11.72	32	9.6	58.79	4416	3.78	5.02	4.35	27.4	1.65	351.4	78.8	328.8	1.7
sr10	1.06	15.03	24	14.1	0.97	10.83	32	11.1	52.77	2944	3.51	4.87	4.14	33.8	2.03	433.5	97.2	396.3	1.3
sr11	0.85	13.59	26	16.0	0.79	10.21	32	12.9	47.71	5888	3.51	4.67	4.05	40.9	2.47	524.6	117.5	488.0	3.3
sr12	0.70	12.98	28	18.5	0.65	8.79	32	13.5	43.30	5888	3.34	4.93	4.05	48.6	2.93	623.7	139.7	579.5	3.1
sr13	0.58	11.40	28	19.5	0.54	8.18	32	15.2	37.83	4416	3.32	4.62	3.92	56.9	3.44	731.1	163.9	665.7	2.3
sr14	0.50	10.37	28	20.8	0.44	7.09	32	16.1	34.98	5888	3.37	4.93	4.08	65.9	3.99	847.0	189.9	775.2	3.4
sr15	0.43	9.22	28	21.6	0.33	6.51	32	20.0	31.69	5888	3.44	4.86	4.09	75.6	4.58	972.2	217.9	886.9	3.7
lr2	69.07	70.69	2	1.0	123.55	132.09	8	1.1	64.58	1472	0.91	0.49	0.67	1.6	0.09	16.5	3.7	16.2	-
lr3	8.74	33.89	12	3.9	7.79	60.93	8	7.8	58.73	1472	1.73	0.96	1.29	5.7	0.36	59.4	13.3	55.5	-
lr4	4.13	25.27	22	6.1	3.61	38.97	8	10.8	50.93	5888	2.02	1.31	1.62	11.1	0.73	118.2	26.7	109.9	1.8
lr5	2.36	23.56	26	10.0	2.15	21.87	8	10.2	50.09	5888	2.13	2.29	2.21	17.8	1.20	192.4	43.4	178.4	2.0
lr6	1.50	17.86	28	11.9	1.43	13.15	30	9.2	39.84	1472	2.23	3.03	2.60	26.0	1.77	282.8	63.7	256.2	_
lr7	1.03	14.73	28	14.3	1.01	10.41	30	10.3	39.00	2944	2.65	3.74	3.15	35.8	2.45	389.4	87.7	354.8	1.3
lr8	0.74	12.52	28	16.9	0.74	8.60	32	11.6	39.02	2944	3.12	4.54	3.76	47.0	3.24	511.8	115.4	463.9	1.0
lr9	0.58	10.63	26	18.5	0.56	7.57	32	13.4	38.22	4416	3.60	5.05	4.26	59.8	4.14	651.3	146.7	595.8	1.6
lr10	0.45	9.27	28	20.6	0.37	6.27	32	17.0	38.24	5888	4.13	6.10	5.02	74.0	5.12	806.4	181.7	734.9	3.2
									gme	an	2.81	2.75	2.78						

Table 3. st-kHz, **mt-kHz** are single- and multi-thread Verilator performance (blue is best of ix3-ae4). **kHz** is best PARENDI rate. **gain** is Verilator's self-relative speedup (underscored superlinear). **#T** is threads or tile count. **Speedup** is PARENDI vs. Verilator (green ≥ 2 and red < 1). **gmean** is reported across machines and benchmarks. **MiB** is Verilator's binary size. **#I** is the millions of x64 instructions per RTL cycle (Verilator). **#N** is thousands of data dependence graph nodes. **#F** is thousands of fibers. **Int.** and **Ext.** are KiBs on- and off-chip cut size (lower than actual communication volume due to fanout).

6.4 Cost Comparison

The IPU's performance advantage for large designs makes it more cost-effective than other systems. The cloud hosting service GCore offered IPU-POD4 classic instances (an M2000, see Table 2) for \$2.13 per hour [2]. A Dv4 Microsoft Azure instance (Xeon 8272CL) costs \$0.048 per hour per core [3] or \$0.77 per hour for 16 cores. We use the sr15 design to briefly compare the cost of running long and short simulations on PARENDI and Verilator. We exclude compilation time and cost from our analysis.

Single Long Test. Consider simulating sr15 for 1 billion cycles. On Dv4, the simulation scales from 222 Hz (1 thread) to 4.88 kHz (16 threads, a superlinear 22× speedup, but slows down beyond 16 threads). The rate on IPU-POD4 scales from 22.94 kHz (1 IPU) to 31.69 kHz (4 IPUs). So, IPU-POD4 finishes the simulation in 9 hours, costing \$19.20. But, Dv4 takes 57 hours and costs \$43.78 (16 threads).

A back-of-the-envelope calculation shows that IPU-POD4 is always more cost-effective than Dv4 irrespective of the number of rented cores. Let t be the number of Verilator



Figure 13. Nightly test simulation time (in hours, sr15) on a 16-core Dv4 instance (x64) and the IPU-POD4 classic (ipu). Numbers on bars are total cost and N is number of run tests.

threads and *s* be its speedup (vs. single-thread performance). Four IPUs run 142.74× faster than single-thread Dv4. So long as $\frac{s}{t} < 142.74 \times \frac{0.048}{2.13} = 3.2$, Dv4 with *t* threads will cost more than IPU-POD4. Since linear scaling is $\frac{s}{t} = 1$, Verilator would become cost-effective only at a 3.2× superlinear scaling, which is very far from what we observe.

PARENDI: Thousand-Way Parallel RTL Simulation

Several Short Tests. We now run many short—1 million cycles—"nightly regression" tests using two strategies. First, the most straightforward strategy: On Dv4, we assign a core per test, running 16 tests in parallel. Since it is impossible to assign one IPU tile to each test (there is not enough memory), we conservatively assign one IPU to each test and run four tests in parallel at a time on IPU-POD4. We call this ad-hoc parallelism. Second, we run each test with an optimal number of threads or tiles, which is 16 on Dv4 and 5888 on IPU-POD4. We call this fine-grained parallelism, as for this benchmark, we end up exploiting parallelism within each test but running the tests one after the other.

Admittedly, these scheduling strategies do not explore the performance-cost space systematically; they illustrate the two obvious alternatives. Fig. 13 shows the time to finish N tests. The numbers on each bar show the total cost. On the IPU, ad-hoc parallelism is more cost-effective because fine-grained parallelism scales sublinearly. On the x64, we see the opposite trend since ad-hoc parallelism scales almost linearly, while fine-grained parallelism scales superlinearly ($22 \times$ on 16 threads). The fine-grained parallelism finishes faster. Regardless, PARENDI is cheaper for both approaches since its performance edge exceeds the cost difference.

Power and Energy Estimates. We measured that the 4 IPUs consume 185W. Due to security limitations, we could not measure power draw on our x64 baselines. The ae4 and ix3 TDP's are 320W and 235W, respectively. We estimate the power draw to be 80W and 118W as we use a quarter of the cores on ae4 and half on ix3. sr15 on the IPU is 4.86× faster than ae4 and 3.44× faster than ix3, so the IPU's energy draw is about 2× lower than x64.

6.5 Comparison with Other Systems

RepCut. RepCut [58] is a BSP RTL simulator (full-cycle) for firrtl [27] on x64. RepCut demonstrates superlinear speedups within and across sockets (we saw a similar effect with chiplets). It improved simulation up to 32 cores (48-core machine) but showed no gains beyond that.

We were unable to compare directly for frustrating practical reasons. Chipyard's CIRCT backend has replaced the first firrtl Scala compiler that RepCut is based on. The new version cannot ingest the srN and lrN designs produced by Chipyard—there is no reliable way to convert Verilog back to firrtl. However, we were able to simulate an older Rocket SoC (based on git hash 4276f17f9) with RepCut and compare it against PARENDI. We ported our lightweight Chipyard test driver, free of DPI calls, to the Rocket SoC and removed all internal print statements. We found that the stock test driver for Rocket SoC (and Chipyard) severely limits performance. RepCut reports a 1-core Rocket SoC simulates at \approx 10 kHz on Verilator and \approx 50 kHz on RepCut (single-thread). We reproduced these results. However, with our streamlined test driver, the benchmarks ran at 276 kHz on Verilator and



Figure 14. Performance of RepCut (rct) [58], Verilator (vlt), and ipu. Figure legend is on the bottom right.



Figure 15. Comparison of ipu and Manticore (mcr).

75 kHz on RepCut (on ae4), showing that the original Rocket test driver is a debatable baseline.

Fig. 14 compares Verilator, RepCut, and PARENDI for various SoC sizes. We ran Verilator and RepCut simulations on ae4 up to 32 threads. PARENDI ran on a single IPU. Verilator is fastest for smaller designs, RepCut gains a small advantage for medium SoCs, and PARENDI performs best for the largest. Code generated by RepCut for the 32-core SoC crashes clang.

Manticore. Manticore [21] is a 225-core, statically scheduled, deeply pipelined architecture designed for BSP RTL simulation and prototyped on an FPGA. Manticore's compiler frontend does not support Verilog's packed arrays used abundantly in 1rN and srN. Moreover, since FPGAs have limited memory, large designs do not fit on Manticore. Fig. 15 compares PARENDI (1472 tiles) to Manticore (225 cores) using the raw numbers reported in their work [21] (see [21] for the description of the designs). Manticore's huge register file lets it achieve a higher single-core rate than the IPU. So, the small bc design runs faster on it, but the larger vta and mc designs benefit from PARENDI's greater parallelism.

6.6 Partitioning Strategies

So far, we have used the partitioning strategy outlined in §5.1. This section considers alternative strategies for partitioning fibers within and across IPUs.

Single-IPU Partitioning. RepCut [58] formulates SLB as a hypergraph partitioning problem where hypergraph nodes fibers and hyperedges represent duplicated clusters across fibers. We implemented this strategy as an alternative. Fig. 16 compares the default bottom-up (**B**, §5.1) strategy against hypergraph partitioning (**H**) on a single IPU (1472-way partitioning). Neither strategy is uniformly better. Bottom-up performs best with srN, whereas hypergraph is sometimes better with lrN.



Figure 16. Comparison of PARENDI's bottom-up SLB algorithm (**B**) against RepCut's hypergraph approach (**H**) [58]. The vertical axis shows normalized IPU machine cycles per RTL cycle (lower is better).



Figure 17. Normalized simulation rate for 4-IPU partitioning strategies. Partitioning fibers pre merge performs better than partitioning processes post merge. Ignoring the muli-IPU configuration (none) yields vastly inferior results.

Multi-IPU Partitioning. Fig. 17 compares three strategies for multi-device partitioning of 4 IPUs: (pre) partition fibers across IPUs before merging them into processes (default PARENDI strategy), (post) partition processes across IPU, i.e., after merging fibers into processes, and (none) does not partition, i.e., multi-IPU oblivious.

Not partitioning fibers or processes across IPUs yields inferior performance. Partitioning fibers works better than partitioning processes. The former approach offers more degrees of freedom for partitioning since the earlier process merge may suboptimally absorb some *good cuts* and land in a region of the design space that is only locally optimal.

6.7 Discussion

Fast simulation requires a simulator that exploits the finegrain parallelism of RTL and effectively utilizes the features of the underlying hardware platform. Verilator fails to scale because its frequent fine-grain synchronization overuses the x64's costly synchronization and communication. PARENDI scales better, albeit from lower single-core performance, because an IPU efficiently supports BSP synchronization and low-latency communication. However, the IPU's high offchip latency demands effective RTL partitioning to minimize cross-chip traffic. When we started this project, we expected no speed gains from multiple IPUs and only planned to use additional IPUs when we ran out of memory. We were surprised to find speedups to even 5888 cores and believe compiler improvements would increase performance further.

VLSI design practices explain why speedups are possible on thousands of cores. Optimizing circuit performance for synthesis, placement, and routing requires a *floorplan* that is aware of physical constraints such as pin placement. A good design facilitates floorplanning and optimizes off-chip communication in its simulation—there is a natural minimal cut. The critical path length in a VLSI circuit limits the clock rate, just as the straggler fiber and its cone of logic limit the simulation rate. A fast circuit design minimizes the critical path length, indirectly minimizing the critical cone of logic (area) and producing many fibers. In general, faster circuits should simulate faster and utilize parallelism better.

The lessons from PARENDI may help apply BSP for RTL simulation on other parallel architectures. Low-latency memory (SRAM) capacity is the main enabler of high-speed medium-to-large RTL simulations (and a bottleneck on x86). Other architectures such as Groq [7, 8] or Cerebras [33] offer considerable low-latency memory and might be good platforms for RTL simulation. By contrast, the NVIDIA H100 GPU has only ≈50 MiBs of shared on-chip memory. As a result, GPUs are unlikely to perform well using the BSP execution model. Besides SRAM capacity, low-cost synchronization is essential for BSP RTL simulation on an accelerator. The IPU (and perhaps a Groq-like architecture) has predictably low synchronization costs, as computation and communication are almost entirely statically scheduled. Tentative experiments found (not presented in the paper) that full-device synchronization on GPUs does not share the same predictably low latencies.

7 Related Work

PARENDI is the first RTL simulator for a few thousand cores. Prior work used tens of CPU and GPU cores or a few hundred specialized cores. In addition, prior work uses software or FPGA emulation to simulate thousand-core SoCs.

7.1 Tens of Cores

Verilator and RepCut [58] are parallel, full-cycle RTL simulators that target commodity processors with few tens of cores. Both are limited by the x64's expensive synchronization and communication, as shown earlier. Other research on parallel, event-driven techniques focuses on finer granularity rather than full-cycle simulation. They employ smart concurrency techniques on x64 to avoid computation/communication [9, 31, 34, 61]. SAGA [57] achieves a 16× parallel speedup on GPUs by statically scheduling SystemC [41]. GCS [16–18] employs *levelization* [59, 60] and acclerates gate-level simulation on GPUs.

7.2 Hundreds of Cores

100+ core general-purpose machines are uncommon, except in GPUs and accelerators. Since RTL is irregular, SIMD execution on a GPU typically yields low thread utilization. Qian et al. [43] describe a GPU-accelerated, event-driven simulator with a single thread per GPU core (i.e., $\frac{1}{32}$ Warp utilization). RTLFlow [35] fully utilizes a GPU by independently simulating a single design driven by multiple test vectors. RTLFlow performs comparably to Verilator with 1K test vectors but runs 40× faster with 64K tests. RTLFlow is limited by available GPU memory, as described in [20]. Nexus [15] is an FPGA-based parallel RTL simulator with a systolic array of 240 8-bit processors. Like Manticore, Nexus suffers from limited SRAM resources on FPGAs. ASH [20] extends the Swarm architecture [28] with prioritized dataflow to accelerate RTL simulation. It comprises 256 simple x64 cores with dedicated task queues to support efficient event-driven simulation. ASH demonstrated a 32× speedup over Verilator running on a simulator (it has not been prototyped).

7.3 Thousands of Cores

Most previous work on thousand-way parallel simulation used C++ processor (architectural) models, not RTL. An RTL simulator parallelizes a model (the code), whereas an architectural simulator *implements* parallel models. Hence, the simulator developer must parallelize the code rather than our compiler. Moreover, many architectural simulators compromise modeling accuracy to enable efficient parallel execution [19, 38, 52, 65], which is incompatible with RTL's rigid semantics. Some architectural simulators use multiple machines for large-scale simulation [24, 26, 36, 38]. Like PARENDI, the motivation for distributed simulation is to use multiple machines' computing and memory resources.

Metro-MPI [36] is a framework for manually connecting independent simulations of hardware components that interact over a clearly defined interface, such as a NoC. It does not compile an RTL design into an executable that can simulate a design. Instead, it utilizes coarse-grain parallelism and concurrently simulates one or a few components on each core. Theoretically, Metro-MPI could exploit finegrain parallelism by dedicating several cores to a simulation and running Verilator in parallel. However, realistically, it requires implementation and fine-tuning on a per-project basis. By contrast, PARENDI automatically (without developer assistance) extracts fine-grain parallelism across an entire RTL design, groups it into appropriate-sized computations, and maps them to available computing resources. That said, Metro-MPI manages to evaluate a design that is $\approx 20 \times \text{larger}$ than the largest design we evaluated in this work (i.e., sr15 and 1r10). We estimate this difference: Metro-MPI evaluates a 1000-core chip containing 10 billion transistors, or perhaps 1 billion generic gates, including SRAM. As mentioned in §6, 1r10 has ≈ 20 million gates, excluding SRAM. Including SRAM in our estimation, 1r10 probably contains 50 million gates, 20× smaller than Metro-MPI's workload.

Emulating large systems on FPGAs is an alternative to simulation, but capacity and compile time are significant challenges. FireSim [30] and DIABLO [53] simulate warehousescale computers, but individual FPGAs limit the overall scale (e.g., 8-core processor). Other emulation platforms connect many FPGAs into a single logical FPGA [12, 32] to circumvent resource limits. These systems share problems, such as partitioning, with software-based RTL simulation but also suffer from protracted compilation time (hours to days) to map logic to FPGA primitives. However, they can emulate large systems as fast as 1 MHz at a high price.

8 Conclusion

Thousand-way parallel RTL simulation is becoming necessary. Current simulation techniques can adequately exploit only tens of cores in general-purpose processors because of their high synchronization and communication costs.

We used a 1472-core computer, the Graphcore IPU, to study the feasibility and challenges of massively parallel simulation. Our study analyzed three dimensions of parallel simulation: synchronization, communication, and computation. Using these results, we implemented PARENDI, an RTL compiler that can use up to 5888 cores effectively. Despite the IPU's almost 84× single-core performance disadvantage against x64 machines, PARENDI runs up to 4× faster on large designs.

Our work demonstrates that thousand-way parallel RTL simulation is practical and beneficial. It opens new avenues for future research that speeds up RTL simulation on massively parallel systems.

9 Acknowledgements

We are grateful to Graphcore for lending us the M2000 hardware. We thank the the Graphcore staff and engineers, especially Mark Pupilli, Dario Domizioli, Peter Birch, Svetlomir Hristozkov, Marie-Ann Le Menn, and David Bozier. They helped us throughout the project development, from getting us started to detailed explanations of the inner workings of poplar and popc, and providing feedback on our work.

At EPFL, Sahand Kashani and Rishabh Iyer's feedback on the writing and paper's narrative helped us make significant improvements. Furthermore, Sanidhya Kashyap generously allowed us to use their machines to benchmark Verilator. We thank Edouard Bugnion and Margaret Church for their support in the last year of the Very Large Scale Laboratory at EPFL. Last but not least, we thank Jiacheng Ma, who made us realize the potential of using the IPUs for RTL simulation.

References

- [1] 4th gen AMD EPYC Processor Archiecture. Technical report, AMD.
- [2] AI IPU Cloud Infrastructure. https://gcore.com/cloud/ai-platform. Accessed: 22-11-2023.
- [3] Azure pricing calculator. https://azure.microsoft.com/en-us/pricing/ calculator/. Accessed 24-06-2024.
- [4] Introducing the Colussus MK2 GC200 IPU. https://www.graphcore.ai/ products/ipu. Accessed: 2023-11-23.
- [5] Open-Source FPGA Bitcoin Miner. https://github.com/progranism/ Open-Source-FPGA-Bitcoin-Miner.
- [6] Poplar Graph Programming Framework. https://docs.graphcore.ai/en/ latest/child-pages/poplar.html#poplar.
- [7] Dennis Abts, Garrin Kimmell, Andrew C. Ling, John Kim, Matthew Boyd, Andrew Bitar, Sahil Parmar, Ibrahim Ahmed, Roberto DiCecco, David Han, John Thompson, Michael Bye, Jennifer Hwang, Jeremy Fowers, Peter Lillian, Ashwin Murthy, Elyas Mehtabuddin, Chetan

Tekur, Thomas Sohmers, Kris Kang, Stephen Maresh, and Jonathan Ross. A software-defined tensor streaming multiprocessor for largescale machine learning. In *Proceedings of the 49th International Symposium on Computer Architecture (ISCA)*, pages 567–580, 2022.

- [8] Dennis Abts, Jonathan Ross, Jonathan Sparling, Mark Wong-VanHaren, Max Baker, Tom Hawkins, Andrew Bell, John Thompson, Temesghen Kahsai, Garrin Kimmell, Jennifer Hwang, Rebekah Leslie-Hurd, Michael Bye, E. R. Creswick, Matthew Boyd, Mahitha Venigalla, Evan Laforge, Jon Purdy, Purushotham Kamath, Dinesh Maheshwari, Michael Beidler, Geert Rosseel, Omar Ahmad, Gleb Gagarin, Richard Czekalski, Ashay Rane, Sahil Parmar, Jeff Werner, Jim Sproch, Adrian Macias, and Brian Kurtz. Think Fast: A Tensor Streaming Processor (TSP) for Accelerating Deep Learning Workloads. In Proceedings of the 47th International Symposium on Computer Architecture (ISCA), pages 145–158, 2020.
- [9] Tariq Bashir Ahmad, Namdo Kim, Byeong Min, Apurva Kalia, Maciej Ciesielski, and Seiyang Yang. Scalable parallel event-driven HDL simulation for multi-cores. In 2012 International Conference on Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), pages 217–220, 2012.
- [10] Alon Amid, David Biancolin, Abraham Gonzalez, Daniel Grubb, Sagar Karandikar, Harrison Liew, Albert Magyar, Howard Mao, Albert J. Ou, Nathan Pemberton, Paul Rigge, Colin Schmidt, John Charles Wright, Jerry Zhao, Yakun Sophia Shao, Krste Asanovic, and Borivoje Nikolic. Chipyard: Integrated Design, Simulation, and Implementation Framework for Custom SoCs. *IEEE Micro*, 40(4):10–21, 2020.
- [11] Krste Asanović, Rimas Avižienis, Jonathan Bachrach, Scott Beamer, David Biancolin, Christopher Celio, Henry Cook, Palmer Dabbelt, John Hauser, Adam Izraelevitz, Sagar Karandikar, Benjamin Keller, Donggyu Kim, John Koenig, Yunsup Lee, Eric Love, Martin Maas, Albert Magyar, Howard Mao, Miquel Moreto, Albert Ou, David Patterson, Brian Richards, Colin Schmidt, Stephen Twigg, Huy Vo, and Andrew Waterman. The Rocket Chip Generator. Technical report, University of California, Berkeley, 2016.
- [12] Jonathan Babb, Russell Tessier, Matthew Dahl, Silvina Hanono, David M. Hoki, and Anant Agarwal. Logic emulation with virtual wires. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 16(6):609–626, 1997.
- [13] Scott Beamer. A Case for Accelerating Software RTL Simulation. IEEE Micro, 40(4):112–119, 2020.
- [14] Scott Beamer and David Donofrio. Efficiently exploiting low activity factors to accelerate RTL simulation. In 57th ACM/IEEE Design Automation Conference, DAC 2020, San Francisco, CA, USA, July 20-24, 2020, pages 1–6. IEEE, 2020.
- [15] Peter Birch. Open source FPGA-based emulation with Nexus. In Workshop on Open-Source EDA Technology (WOSET), number 1, 2022.
- [16] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. Eventdriven gate-level simulation with GP-GPUs. In Proceedings of the 46th Design Automation Conference, DAC 2009, San Francisco, CA, USA, July 26-31, 2009, pages 557–562. ACM, 2009.
- [17] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. GCS: High-performance gate-level simulation with GPGPUs. In Luca Benini, Giovanni De Micheli, Bashir M. Al-Hashimi, and Wolfgang Müller, editors, Design, Automation and Test in Europe, DATE 2009, Nice, France, April 20-24, 2009, pages 1332–1337. IEEE, 2009.
- [18] Debapriya Chatterjee, Andrew DeOrio, and Valeria Bertacco. Gate-Level Simulation with GPU Computing. ACM Trans. Design Autom. Electr. Syst., 16(3):30:1–30:26, 2011.
- [19] Jianwei Chen, Murali Annavaram, and Michel Dubois. SlackSim: a platform for parallel simulations of CMPs on CMPs. SIGARCH Comput. Archit. News, 37(2):20–29, 2009.
- [20] Fares Elsabbagh, Shabnam Sheikhha, Victor A. Ying, Quan M. Nguyen, Joel S. Emer, and Daniel Sánchez. Accelerating RTL Simulation with Hardware-Software Co-Design. In *Proceedings of the 56th Annual*

IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023, pages 153–166. ACM, 2023.

- [21] Mahyar Emami, Sahand Kashani, Keisuke Kamahori, Mohammad Sepehr Pourghannad, Ritik Raj, and James R. Larus. 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, ASPLOS '23, page 219–237, New York, NY, USA, 2024. Association for Computing Machinery.
- [22] Harry Foster. Part 4: The 2020 Wilson Research Group Functional Verification Study, FPGA Verification Effort Trends, 12 2020.
- [23] Harry Foster. Part 8: The 2020 Wilson Research Group Functional Verification Study, IC/ASIC Resource Trends, 1 2021.
- [24] Yaosheng Fu and David Wentzlaff. PriME: A parallel and distributed simulator for thousand-core chips. In Proceedings of the 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pages 116–125, 2014.
- [25] M. R. Garey, Ronald L. Graham, and David S. Johnson. Performance Guarantees for Scheduling Algorithms. Oper. Res., 26(1):3–21, 1978.
- [26] Steven Herbst, Noah Moroze, Edgar Iglesias, and Andreas Olofsson. Switchboard: An Open-Source Framework for Modular Simulation of Large Hardware Systems. *ArXiv*, abs/2407.20537, 2024.
- [27] Adam M. Izraelevitz, Jack Koenig, Patrick Li, Richard Lin, Angie Wang, Albert Magyar, Donggyu Kim, Colin Schmidt, Chick Markley, Jim Lawson, and Jonathan Bachrach. Reusability is FIRRTL ground: Hardware construction languages, compiler frameworks, and transformations. In 2017 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2017, Irvine, CA, USA, November 13-16, 2017, pages 209–216. IEEE, 2017.
- [28] Mark C. Jeffrey, Suvinay Subramanian, Cong Yan, Joel S. Emer, and Daniel Sánchez. A scalable architecture for ordered parallelism. In Proceedings of the 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), pages 228–241, 2015.
- [29] Zhe Jia, Blake Tillman, Marco Maggioni, and Daniele Paolo Scarpazza. Dissecting the Graphcore IPU Architecture via Microbenchmarking. *CoRR*, abs/1912.03413, 2019.
- [30] 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 Howard Katz, Jonathan Bachrach, and Krste Asanovic. FireSim: FPGA-Accelerated Cycle-Exact Scale-Out System Simulation in the Public Cloud. *IEEE Micro*, 39(3):56–65, 2019.
- [31] Dusung Kim, Maciej J. Ciesielski, and Seiyang Yang. A new distributed event-driven gate-level HDL simulation by accurate prediction. In Design, Automation and Test in Europe, DATE 2011, Grenoble, France, March 14-18, 2011, pages 547–550. IEEE, 2011.
- [32] Helena Krupnova and Gabriele Saucier. FPGA-based emulation: Industrial and custom prototyping solutions. In Proceedings of the The Roadmap to Reconfigurable Computing, 10th International Workshop on Field-Programmable Logic and Applications, FPL '00, page 68–77, Berlin, Heidelberg, 2000. Springer-Verlag.
- [33] Gary Lauterbach. The Path to Successful Wafer-Scale Integration: The Cerebras Story. IEEE Micro, 41(6):52–57, 2021.
- [34] Tun Li, Yang Guo, and Sikun Li. Design and Implementation of a Parallel Verilog Simulator: PVSim. In VLSI Design, pages 329–334, 2004.
- [35] Dian-Lun Lin, Haoxing Ren, Yanqing Zhang, Brucek Khailany, and Tsung-Wei Huang. From RTL to CUDA: A GPU Acceleration Flow for RTL Simulation with Batch Stimulus. In Proceedings of the 51st International Conference on Parallel Processing, ICPP 2022, Bordeaux, France, 29 August 2022 - 1 September 2022, pages 88:1–88:12. ACM, 2022.

- [36] Guillem López-Paradís, Brian Li, Adrià Armejach, Stefan Wallentowitz, Miquel Moretó, and Jonathan Balkind. Fast Behavioural RTL Simulation of 10B Transistor SoC Designs with Metro-Mpi. In Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023, pages 1–6. IEEE, 2023.
- [37] George Marsaglia. Xorshift RNGs. Journal of Statistical Software, 8(14):1–6, 2003.
- [38] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald III, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In Proceedings of the 16th IEEE Symposium on High-Performance Computer Architecture (HPCA), pages 1–12, 2010.
- [39] Thierry Moreau, Tianqi Chen, Luis Vega, Jared Roesch, Eddie Q. Yan, Lianmin Zheng, Josh Fromm, Ziheng Jiang, Luis Ceze, Carlos Guestrin, and Arvind Krishnamurthy. A Hardware-Software Blueprint for Flexible Deep Learning Specialization. *IEEE Micro*, 39(5):8–16, 2019.
- [40] Mahesh Nanjundappa, Hiren D. Patel, Bijoy Antony Jose, and Sandeep K. Shukla. SCGPSim: a fast SystemC simulator on GPUs. In Proceedings of the 15th Asia South Pacific Design Automation Conference, ASP-DAC 2010, Taipei, Taiwan, January 18-21, 2010, pages 149–154. IEEE, 2010.
- [41] OSCI. SystemC. https://www.systemc.org.
- [42] PicoRV32 A Size-Optimized RISC-V CPU. https://github.com/ YosysHQ/picorv32.
- [43] Hao Qian and Yangdong Deng. Accelerating RTL simulation with GPUs. In Joel R. Phillips, Alan J. Hu, and Helmut Graeb, editors, 2011 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2011, San Jose, California, USA, November 7-10, 2011, pages 687–693. IEEE Computer Society, 2011.
- [44] Karl Rupp. Microprocessor trend data. https://github.com/karlrupp/ microprocessor-trend-data, 2022. Accessed: 18-10-2023.
- [45] Sartaj Sahni. Algorithms for Scheduling Independent Tasks. J. ACM, 23(1):116–127, 1976.
- [46] Vivek Sarkar and John L. Hennessy. Compile-time partitioning and scheduling of parallel programs. In SIGPLAN Symposium on Compiler Construction, pages 17–26, 1986.
- [47] Sebastian Schlag, Tobias Heuer, Lars Gottesbüren, Yaroslav Akhremtsev, Christian Schulz, and Peter Sanders. High-Quality Hypergraph Partitioning. ACM J. Exp. Algorithmics, 27:1.9:1–1.9:39, 2022.
- [48] Wilson Snyder. Verilator, accelerated: Accelerating development, and case study of accelerating performance. 2nd Workshop on Open-Source Design Automation (OSDA).
- [49] Wilson Snyder. Verilator 4.0: Open simulation goes multithreaded. The OPen Source Digital Design Conference (ORConf), 2018.
- [50] Wilson Snyder. Your Big 4th Simulator: 2019 intro and roadmap. CHIPS Alliance, 2019.
- [51] Zoya Svitkina and Lisa Fleischer. Submodular Approximation: Sampling-based Algorithms and Lower Bounds. SIAM J. Comput., 40(6):1715–1737, 2011.
- [52] Daniel Sánchez and Christos Kozyrakis. ZSim: fast and accurate microarchitectural simulation of thousand-core systems. In *Proceedings* of the 40th International Symposium on Computer Architecture (ISCA), pages 475–486, 2013.
- [53] Zhangxi Tan, Zhenghao Qian, Xi Chen, Krste Asanovic, and David A. Patterson. DIABLO: A Warehouse-Scale Computer Network Simulator using FPGAs. In Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-XX), pages 207–221, 2015.
- [54] Xiang Tian and Khaled Benkrid. Design and implementation of a high performance financial Monte-Carlo simulation engine on an FPGA supercomputer. In Tarek A. El-Ghazawi, Yao-Wen Chang, Juinn-Dar Huang, and Proshanta Saha, editors, 2008 International Conference on Field-Programmable Technology, FPT 2008, Taipei, Taiwan, December 7-10, 2008, pages 81–88. IEEE, 2008.

- [55] Jeffrey D. Ullman. NP-Complete Scheduling Problems. J. Comput. Syst. Sci., 10(3):384–393, 1975.
- [56] Leslie G. Valiant. A Bridging Model for Parallel Computation. Commun. ACM, 33(8):103–111, 1990.
- [57] Sara Vinco, Debapriya Chatterjee, Valeria Bertacco, and Franco Fummi. SAGA: systemc acceleration on GPU architectures. In Patrick Groeneveld, Donatella Sciuto, and Soha Hassoun, editors, *The 49th Annual Design Automation Conference 2012, DAC '12, San Francisco, CA, USA, June 3-7, 2012*, pages 115–120. ACM, 2012.
- [58] Haoyuan Wang and Scott Beamer. 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 2023, page 572–585, New York, NY, USA, 2023. Association for Computing Machinery.
- [59] L.-T. Wang, Nathan E. Hoover, Edwin H. Porter, and John J. Zasio. SSIM: A software levelized compiled-code simulator. In A. O'Neill and D. Thomas, editors, *Proceedings of the 24th ACM/IEEE Design Automation Conference. Miami Beach, FL, USA, June 28 - July 1, 1987*, pages 2–8. IEEE Computer Society Press / ACM, 1987.
- [60] Zhicheng Wang and Peter M. Maurer. LECSIM: A levelized event driven compiled logic simulation. In Richard C. Smith, editor, Proceedings of the 27th ACM/IEEE Design Automation Conference. Orlando, Florida, USA, June 24-28, 1990, pages 491–496. IEEE Computer Society Press, 1990.
- [61] Seiyang Yang, Jaehoon Han, Doowhan Kwak, Namdo Kim, Daeseo Cha, Junhyuck Park, and Jay Kim. Predictive parallel event-driven HDL simulation with a new powerful prediction strategy. In Gerhard P. Fettweis and Wolfgang Nebel, editors, *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March* 24-28, 2014, pages 1–3. European Design and Automation Association, 2014.
- [62] Jerry Zhao, Animesh Agrawal, Borivoje Nikolic, and Krste Asanović. Constellation: An open-source SoC-capable NoC generator. In 2022 15th IEEE/ACM International Workshop on Network on Chip Architectures (NoCArc), pages 1–7, 2022.
- [63] Jerry Zhao, Ben Korpan, Abraham Gonzalez, and Krste Asanovic. SonicBOOM: The 3rd Generation Berkeley Out-of-Order Machine. May 2020.
- [64] Kexing Zhou, Yun Liang, Yibo Lin, Runsheng Wang, and Ru Huang. Khronos: Fusing Memory Access for Improved Hardware RTL Simulation. In Proceedings of the 56th Annual IEEE/ACM International Symposium on Microarchitecture, MICRO 2023, Toronto, ON, Canada, 28 October 2023 - 1 November 2023, pages 180–193. ACM, 2023.
- [65] Niko Zurstraßen, José Cubero-Cascante, Jan Moritz Joseph, Li Yichao, Xinghua Xie, and Rainer Leupers. par-gem5: Parallelizing gem5's Atomic Mode. In Design, Automation & Test in Europe Conference & Exhibition, DATE 2023, Antwerp, Belgium, April 17-19, 2023, pages 1–6. IEEE, 2023.