

SplitSim: Towards Practical Large-Scale Full-System Simulation for Systems Research

HEJING LI, Max Planck Institute for Software Systems, Germany

MARVIN MEIERS, Max Planck Institute for Software Systems, Germany

JIALIN LI, National University of Singapore, Singapore

ANTOINE KAUFMANN, Max Planck Institute for Software Systems, Germany

When physical testbeds are out of reach for evaluating a networked system, we frequently turn to simulation. In today's datacenter networks, bottlenecks are often not at the network protocol level, but instead in end-host software or hardware components. Here, current protocol-level simulations are an inadequate means of evaluation. Detailed end-to-end simulations covering these components on the other hand, simply cannot achieve the required scale with feasible simulation performance and computational resources.

It is fundamentally difficult to simultaneously achieve high simulation fidelity, low simulation time, and minimal resource consumption—especially for large-scale systems. With limited time and resource budgets, users must find the right compromise across these dimensions. Unfortunately, existing simulation frameworks do not offer flexible trade-offs among these aspects and are unable to make effective use of the time and resources users can afford.

In this paper, we address this with SplitSim, a simulation framework for practical end-to-end evaluation for large-scale network and distributed systems. SplitSim provides the user abstractions to specify the system to be evaluated, and easily & flexibly instantiate different simulations for the same system to navigate these trade-offs. Based on these abstractions, SplitSim allows users to flexibly trade off simulation fidelity, time, and resource usage by mixing simulation models with different fidelity and controlled parallelization. Finally, SplitSim includes a profiler that identifies simulation performance bottlenecks and helps users make informed decisions about parallelization, resource allocation, and simulation detail. With these capabilities, we demonstrate that SplitSim can simulate a 1200-node datacenter network using 24 cores running end-to-end applications in 175 min for 20 s of system time.

CCS Concepts: • **Networks** → **Network simulations**; *Data center networks*; **Network servers**; **Network adapters**; *Bridges and switches*; • **Hardware** → *Networking hardware*; *Buses and high-speed links*.

Additional Key Words and Phrases: Large-Scale Modular Simulation, End-to-End Evaluation, Network Systems

ACM Reference Format:

Hejing Li, Marvin Meiers, Jialin Li, and Antoine Kaufmann. 2025. SplitSim: Towards Practical Large-Scale Full-System Simulation for Systems Research. *Proc. ACM Netw.* 3, CoNEXT4, Article 52 (December 2025), 19 pages. <https://doi.org/10.1145/3768999>

Code: <https://github.com/simbricks/conext25-artifact>

Authors' Contact Information: Hejing Li, Max Planck Institute for Software Systems, Saarbrücken, Germany, hejingli@mpi-sws.org; Marvin Meiers, Max Planck Institute for Software Systems, Saarbrücken, Germany, mmeiers@mpi-sws.org; Jialin Li, National University of Singapore, Singapore, lijl@comp.nus.edu.sg; Antoine Kaufmann, Max Planck Institute for Software Systems, Saarbrücken, Germany, antoinek@mpi-sws.org.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2025 Copyright held by the owner/author(s).

ACM 2834-5509/2025/12-ART52

<https://doi.org/10.1145/3768999>

1 Introduction

Research on large-scale network and distributed systems often faces the challenge of access to adequate testbeds. Most researchers and even many practitioners do not have access to testbeds that are large enough and/or provide the necessary flexibility, control, and hardware. For example, a new datacenter congestion control algorithm might require specific configuration parameters at each network switch in the datacenter, or a distributed system accelerated with in-network processing requires new programmable switches deployed at specific points in the network.

In these cases we typically rely on a patchwork evaluation that combines end-to-end measurements in a small physical testbed with protocol-level simulations for evaluating at scale. However, this methodology compromises accuracy of end-to-end system behaviors at scale. The physical testbed is by necessity too small and protocol-level simulations do not model many system components, from NIC behavior, to host-interconnects, memory hierarchy, and the whole OS and application-level software stack.

We argue that *simulated end-to-end testbeds can bridge this gap*. In this paper, we take a top-down approach, starting from detailed but slow small-scale end-to-end simulations, we tackle the practical and fundamental challenges in scaling up.

We use existing work on modular end-to-end simulation [11, 18] as a starting point. Modular end-to-end simulations combine and connect different best-of-breed simulators for different system components, and through modularity flexibly cover a broad range of use-cases. These simulations scale up by running separate components as parallel processes communicating and synchronizing them efficiently, locally or distributed across physical machines. SimBricks [11] has demonstrated that this approach scales to simulate a network of 1000 single-core hosts and NICs running Memcached on Linux. However, while technically feasible, this simulation of 10 s of application workload required 6–20 h of simulation time, depending on configuration, on 26 machines with 96 vCPUs, for \$600-\$2000 today on ec2.

With SplitSim, we enable large-scale simulations with more reasonable cost-benefit ratios through a combination of techniques. First, we provide a *configuration and orchestration framework* for end-to-end simulations that simplifies simulating a concrete system flexibly in different ways, by separating the configuration of the simulated system from concrete simulation instantiation choices. We then leverage modularity to implement *mixed fidelity simulations* where some system component instances are simulated in less accurate simulators to drastically reduce CPU resources needed, while key instances remain in accurate simulators. Next, we design generic building blocks for reducing simulation time by *parallelizing bottleneck simulators by decomposing* them into multiple parallel, connected, and synchronized processes. Finally, we introduce *lightweight synchronization and communication profiling* to inform the user about bottlenecks and resource efficiency across component simulator instances.

In our evaluation we demonstrate that SplitSim enables evaluation of large scale systems in networks of up to 1200 hosts, while running complete OS and application stacks for key nodes. SplitSim simulations thus enable full end-to-end application evaluation in large networks. By combining mixed fidelity simulation with parallelization through decomposition, SplitSim can simulate 20 seconds of a large-scale system in less than 3 hours while running on a single machine. The modular approach and flexible configuration and orchestration framework makes SplitSim suitable for a broad range of evaluation use-cases. SplitSim is open source and the full paper artifact is available here: <https://github.com/simbricks/conext25-artifact>.

This work does not raise ethical issues.

	End-to-End	Scalability	Fidelity	Flexibility	Engineering Effort
Estimators	✗	✓	✗	✗	High
Sequential DES	✗	✗	✓	✗	Low
Parallel DES	✗	✓	✓	✗	Low
Modular Sim.	✓	✗	✓	✓	Low
SplitSim	✓	✓	✓	✓	Low

Table 1. Overview of existing network simulation approaches and their characteristics.

2 Background and Motivation

2.1 Requirements for Evaluating Large-Scale Network Systems

We argue tooling for practical evaluation of large scale systems should be:

End-to-End: Obtain full-system measurements with all relevant hardware (switches, topology, NICs, server-internals, etc.) and software (application, OS, library) components.

Scalable: Support evaluation of systems with realistic scale, e.g. 100–1000s of hosts.

Efficient: Evaluation within feasible resource limits, in particular processor cycles.

Fast: Keep evaluation and measurement times manageable.

Flexible: Support a broad range of different system configurations.

Easy to Use: Make it easy for users to configure and run evaluations and measurements.

2.2 Existing Approaches Fall Short

We now overview typical evaluation approaches used in network research when physical testbeds are out of reach, and explain why they are insufficient. Table 1 shows an overview.

Performance Estimators. Theoretical models [6, 22] have long served as valuable tools for describing network states and estimating key metrics such as throughput and latency. While they are helpful when network behavior can be precisely articulated through equations, they lack the fine-grained packet-level visibility offered by other approaches. These approaches also do not enable an end-to-end evaluation but instead rely on coarse-grained modeling for a system.

Recent efforts emulate networks using deep neural networks to estimate user-relevant metrics such as delay and packet loss. MimicNet [21], for instance, learns performance metrics at a cluster granularity level and generates estimated packet traces. DeepQueueNet [20] refines this approach by learning device-level performance metrics, thereby enhancing packet visibility within the cluster. These estimations are derived through inferencing input data, which lends itself well to massive parallelization, enabling rapid results for large-scale networks. However, the deep neural net’s behavior is not interpretable. Additionally, to model different network configurations, the model has to be reconstructed and re-trained, incurs substantial computing and engineering effort.

Discrete Event Network Simulation. DES-based simulators such as ns-3 [15] and OMNeT++ [10] are extensively used in networking as they are well suited to model packets in networks. They provide detailed insights into network behavior at the packet and protocol level, enabling in-depth analysis. These simulators model network events with timestamps, such as a packet being generated at a host, transmitted through a link, and received at the other end. Events are processed in chronological order to update the state of the simulated network. Other simulators leverage DES to simulate

other components, such as gem5 [4] or Simics [13] for computers including detailed processor and memory hierarchy.

However, sequential DES struggles with scalability when modeling large systems. For example, simulating a few seconds of a modern datacenter network—with vast traffic from thousands of end hosts—can take hours or even days. Parallelizing DES [2, 3, 14, 17] can reduce simulation times. The primary challenge is the difficulty of efficiently parallelizing while accurately synchronizing DES. For example, with ns-3’s native parallelization, we found partitioning a network simulation across 16 processes on 16 cores resulted in only a 3.8x speedup. Clean-slate parallel designs [8, 19] further improve parallelism, but lack the rich feature sets developed over decades of research and necessarily focus on simulating specific components rather than entire end-to-end systems.

More fundamentally, a DES-based simulation only models behavior explicitly included in the models of the simulator. Given the growing complexity of modern network systems and all the components involved, it appears infeasible to add all necessary models to one single simulator. Individual simulation models also fundamentally need to choose what to model and what to skip to keep simulation overhead manageable. As such, an individual simulator may be well suited for one use, but inadequate for another.

Modular Simulation. To address this, modular simulation frameworks such as SimBricks [11] and SST [18] allow users to combine multiple simulators into one complete system. For example, SimBricks combines simulators for host, hardware device, and network to construct an end-to-end simulation, and run in parallel. By combining different best-of-breed simulators, this approach leverages the substantial engineering effort that has gone into developing simulators for different components, and combines this effort across components. Modular simulation is also flexible, as depending on the choice of simulators, a simulation can be configured to be accurate but slow, or faster and less accurate, by using different simulators. This works well, but for large scale systems, these simulations use computational resources inefficiently. Each system component needs an additional simulator instance running on a separate core, and many cores will waste precious cycles waiting for bottleneck simulators.

SplitSim. With SplitSim we aim to address these shortcomings, by pragmatically leveraging existing pieces and the decades of engineering effort, but augmenting their capabilities to make practical end-to-end simulation of large scale systems possible. We make it easy to simulate a system configuration of interest in different ways, to flexibly explore different simulator choices trade-offs. SplitSim also supports the user in locating simulation bottlenecks, and parallelizing bottleneck simulators while consolidating others. Overall, SplitSim thereby enables *practical & flexible end-to-end evaluation of large-scale systems* in simulation.

2.3 Technical Challenges

Based on the observations above, we use modular end-to-end simulations as a starting point, and examine the compounding challenges in scaling up to meet the requirements for large-scale network systems.

High Resource Requirements for Detailed Simulators. In general, detailed simulators are slower and less resource efficient compared to less-detailed simulators. To obtain meaningful end-to-end measurements, we generally require functionally and timing accurate simulators for all component types in the system, be it processor and memory subsystem, hardware devices such as NICs, or the actual network topology. As a result, modular end-to-end simulations of large-scale systems are prohibitively expensive, requiring hundreds or thousands of processor cores for many hours to simulate seconds.

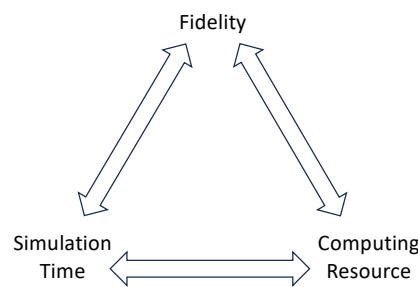


Fig. 1. Inherent three-way trade-off landscape for system simulations.

Simulations Bottlenecked by the Slowest Component. Modular simulation comprising multiple synchronized components can, by construction, only proceed as fast as the slowest component in the system. Slow bottleneck simulators cause two separate problems. First, overall simulation times will be long for simulations that include even just a single slow simulator component. Second, as typical end-to-end simulation will naturally contain component simulators that simulate with different speeds. This results in faster components wasting a lot of processor cycles waiting for slower simulators. At scale, this reduces in substantial waste.

Hard to Understand Simulation Performance. To make matters worse, finding bottlenecks in simulations comprising tens to thousands of communicating and synchronized components is a challenge. Most efficient simulation synchronization mechanisms rely on polling shared memory state for efficiency. Thus, all components will commonly show 100% CPU utilization, and a regular profiler will indicate lots of time spend in the functions that poll for messages. Based on these indicators it is hard to tell if a simulator is bottlenecked or communicating heavily, especially when also combined with heavy compiler optimization. Blocking will also naturally propagate through dependent system components.

Navigating Fidelity, Time, and Cost Trade-offs. Achieving high simulation fidelity while minimizing simulation time and computational resource usage is inherently challenging. Simulating large-scale systems, necessarily requires some compromise across three key dimensions: fidelity, time, or resource demand. Users want to make the best use of their available time, computational budget and simulation fidelity by navigating this trade-off space effectively. Here there is a large space, of different options, different choices of simulation models, parallelization strategies, and strategies for mapping onto available physical resources. Exploring this fundamentally requires exploration and trail-and-error with different configurations. Unfortunately, this is difficult to navigate with existing tools.

Complex Configuration and Execution. More generally, configuring and running simulations for large-scale end-to-end system is a complex task. Many instances of different simulators for different components need to be configured, connected, and then executed in a coordinated manner. The first problem is the complexity: each simulator has its own mechanism and abstractions for configuring it, and there is a substantial learning curve whenever a user looks to use a new simulator. Second, this is complicated by the fact that any non-trivial evaluation typically will need to simulate multiple different configurations of its system, and often needs to explore different simulators and simulator configurations to identify suitable configurations. Finally, once the user has chosen a system and simulation configuration, all components need to be connected together, started in the correct order respecting dependencies, outputs need to be collected, and finally all simulators need to be cleanly terminated. Even with more than a handful of components a manual approach is prohibitively complex and laborious.

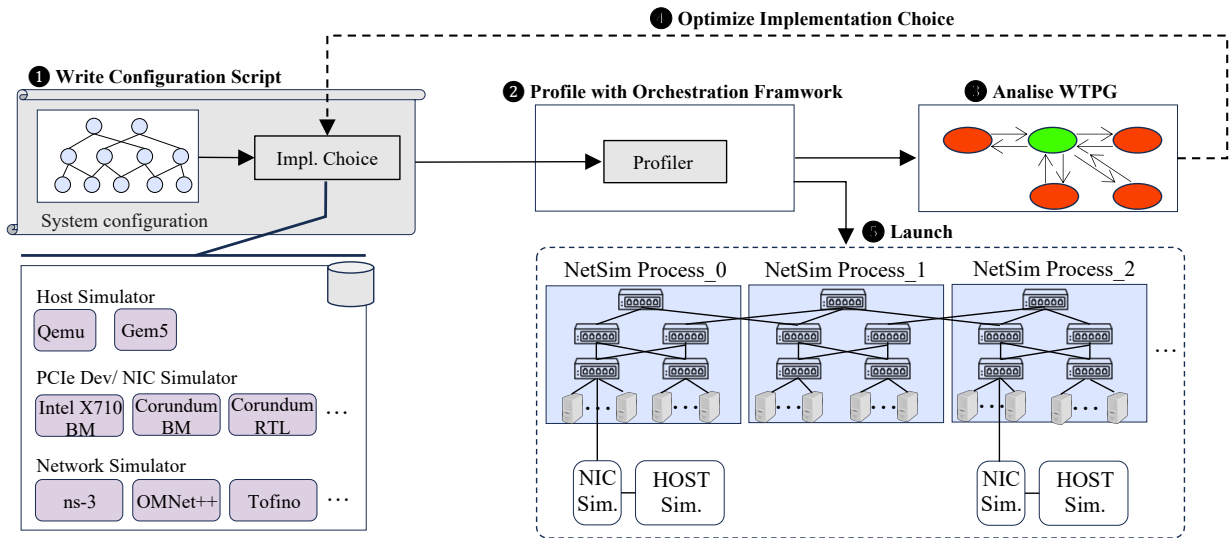


Fig. 2. Overview of SplitSim demonstrating its key components and how they combine.

3 Design and Implementation

SplitSim combines four techniques to address these technical challenges. Figure 2 shows an overview. First, SplitSim reduces the resources needed for large scale simulations by enabling *mixed-fidelity simulations* (3.1), where expensive detailed simulators are replaced with faster, less resource intensive simulations in part of the system, while keeping detailed simulation in other parts. To increase simulation speed and avoid poorly utilized processor cores, SplitSim provides generic building blocks for *parallelizing bottleneck component simulators by decomposing them into parallel processes* (3.2). SplitSim helps users identify bottleneck component simulators and largely idle component simulators with a *cross-simulator synchronization and communication profiler* (3.3). Finally, SplitSim streamlines configuring and running a broad range of different system and simulation configurations, with *programming abstractions for configuration and communication*. (3.4).

3.1 Mixed-Fidelity Simulations

To reduce the computational resources necessary for large-scale end-to-end simulations, we propose mixed-fidelity simulation. The idea is basically to retain a subset of full detailed end-to-end components for parts of the system, while using less resource-intensive simulations for less critical areas of the system.

Reducing Simulation Detail in non-Critical Components. The underlying insight is that typically full detail is not required in every component of the system. A common example is running a system as part of a larger network to evaluate the effect of other background traffic, congestion, etc. in the network; here protocol-level simulation of hosts generating this background traffic is completely sufficient. However, where detailed simulation is required and where detail can be sacrificed depends on the system and evaluation goal. When evaluating peak system throughput for a client-server system, modeling internal client detail is not essential – as long as client requests arrive at the required rate and with the correct protocol or format, the server behavior will be the same. When evaluating end-to-end request latency, on the other hand, client internal behavior is likely to significantly affect measured latency, thus here at least the clients that measure the latency need to be simulated in full detail. For all three of these examples, instead of simulating all hosts end-to-end with detailed architectural simulators, such as qemu or gem5, we can instead simulate a

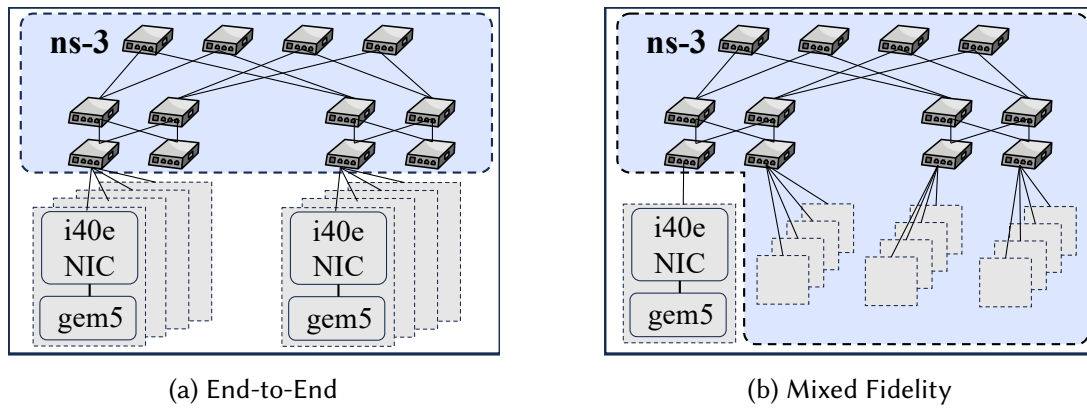


Fig. 3. Changing an end-to-end simulation into a mixed-fidelity simulation by simulating clients at the protocol-level in ns-3 instead using individual host and NIC simulator instances.

specific subset of them at the protocol level, e.g. in ns-3 or OMNet++ (Figure 3). A similar approach applies to other system components, e.g. instead of running expensive RTL-level simulations for all NICs or Switches for projects that propose new hardware design, a judicious combination of RTL simulations with faster and more efficient lightweight simulation models, drastically reduces cycles needed.

Enabling Mixed-Fidelity End-to-End Simulations. At a technical level, SplitSim enables mixed fidelity simulations through modular composition, inherited from SimBricks. Components simulators are connected through fixed message passing interfaces, primarily Ethernet packets and PCI, and individual simulators are thus decoupled from how these messages are generated.

New Challenges. However, while mixed fidelity simulations can drastically reduce computational cost for large-scale simulations, configuring and running mixed fidelity simulations gives rise to or exacerbates the other three challenges. First, these simulations often result in heavy bottlenecks for simulation speed, thereby also introducing significant imbalance leading other simulators to waste cycles waiting and leaving cores idle – with most SimBricks simulations we have run, the end-host simulators (qemu or gem5) are the slowest component by a significant margin, however once we move a few hundred or thousand hosts into the ns-3 network, ns-3 slows down the whole simulation by 3–5×. In the following two subsections we discuss how SplitSim enables users to locate (3.3) and mitigate (3.2) such bottlenecks. Finally, configuring mixed-fidelity simulations and exploring different levels of detail in different system components, is particularly complicated and laborious for users. In addition to building host disk images with applications and configurations, and setting up commands for each host to run etc., a mixed fidelity simulation now also requires configuring additional simulators, e.g. ns-3, to also implement similar functionality through their abstractions. SplitSim simplifies this, in part, through the configuration and orchestration framework (3.4).

3.2 Parallelizing Through Decomposition

In general, parallelizing simulators is a challenging problem with different approaches for different types of simulators. These are well-studied but at least the few major relevant simulators for end-to-end simulations are either sequential (gem5) or scale poorly (ns-3, OMNeT++) [21]. Moreover, the existing parallelization approaches often require intrusive changes to simulators. In SplitSim, we instead propose simpler easy-to-integrate building blocks to parallelize system simulators with modular architectures, such as ns-3, OMNeT++, and gem5.

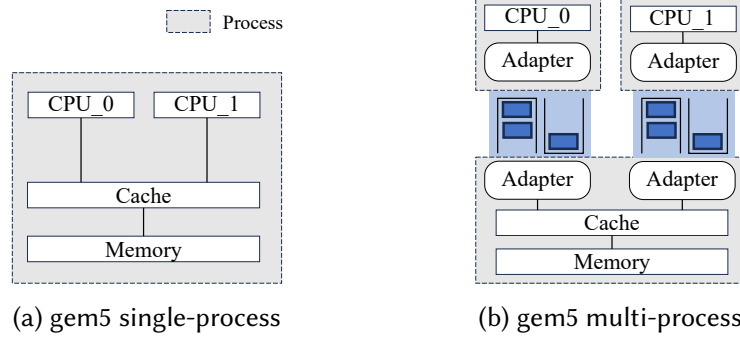


Fig. 4. Parallelizing a sequential multicore architecture simulation by splitting it into parallel processes interconnected with SplitSim adapters.

The key idea in the SplitSim parallelization approach is to decompose these simulators at component boundaries into multiple separate processes.

We then leverage the well-defined module interfaces for connecting and synchronizing the parallel processes with SplitSim adapters that translate these interfaces into messages on SimBricks channels, the same channels also used to interconnect other SplitSim simulator components. Using the same mechanisms enables re-use and provides SplitSim with visibility into the simulation structure for effective orchestration and also enables use the SplitSim profiler for these newly parallel components.

3.2.1 Building Blocks.

Base Adapter. We build on SimBricks adapters in gem5, ns-3, and OMNeT, that are all implemented simply within the device abstractions of each simulator, and implement synchronization through the channel, as well as communication. Based on these, we define an abstract SplitSim base adapter for each simulator, that implements initialization and synchronization, but is not specific to a particular SimBricks channel type. This base adapter can then be used to implement multiple specific protocol adapters without needing to re-implement the common functionality. This includes adapters for the existing SimBricks protocols, but also makes it easy to implement adapters for internally connecting and synchronizing pieces of a simulator.

Trunk Adapter. Many non-trivial partitions will require multiple connections between some pairs of processes. In principle here multiple instances of the SplitSim adapter can be used and this will just work. However, this will unnecessarily incur the synchronization overhead once for each adapter. To address this, SplitSim introduces trunk channels, that multiplex messages for multiple upper layer channels over one synchronized SimBricks channel. The implementation tags messages going across with the sub-channel identifier, for demultiplexing at the receiver.

3.2.2 Examples.

Multi-Core gem5. Figure 4 demonstrates how we use SplitSim adapters to parallelize multi-core simulations in gem5. Changes to gem5 are limited to 1) implementing the adapters as simulation object in gem5, and serializing the already message-based memory packet interface to messages, and 2) changing the gem5 python configuration script to only instantiate the relevant components for each process.

Parallel ns-3 and OMNeT++. We also implemented SplitSim parallelization for the ns-3 and OMNeT++ network simulators. Here we also instantiate different parts of the overall network

topology in separate processes, and replace links going across components with SplitSim trunk link adapters. For network simulators we rely on the user to configure the partitioning and create the adapters, either manually or through our configuration framework (3.4).

3.3 Lightweight Profiling for Synchronization and Communication

To address the challenges in understanding simulation performance, deciding what to parallelize, consolidate, and generally find bottlenecks, SplitSim includes profiling infrastructure. The SplitSim profiler measures metrics related to SplitSim cross-simulator synchronization and communication in each component simulator. The profiler comprises two components: instrumentation in each simulator, and post-processing to aggregate the collected metrics and present them to the user.

3.3.1 Lightweight Instrumentation. SplitSim instruments each adapter, both for communication across simulator components and within the processes of a particular component, with lightweight metric collection and logging. First, each adapter continuously counts the number of 1) *CPU cycles blocked waiting for a synchronization message* from the peer to allow the simulation to proceed, 2) *sending data messages* to peer simulators, and 3) *processing incoming data messages*. Second, each simulator can be set to periodically, e.g. every 10 s, log the values of these counters for each adapter and the current time stamp counter as well as that simulator’s current simulation time.

3.3.2 Profiler Post-Processing. After the simulation terminates, either because it completes or because the user stops it, the profiler post processor ingests and parses these logs. As each simulator logs absolute totals for each value, we calculate the difference between a late entry towards the end and an early entry towards the beginning, dropping a configurable number of warm-up and cool-down lines. Each log entry contains both the simulation time and processor time stamp counter, thereby providing a reference for simulation time and physical system time.

Metrics Calculated. The post processor first calculates a global metric, *simulation speed*, by dividing the difference in simulation time by the difference in time stamp counter cycles (as all simulators are synchronized, this value is the same for each simulator). For each simulator we also calculate their *efficiency* as the fraction of cycles not spent on receive, transmit, or synchronization in the SplitSim adapter. This metric is useful to determine when diminishing returns for parallelizing SplitSim simulations set in.

Wait-Time Profile Graph. The main output for understanding SplitSim simulation performance and for localizing bottlenecks, is the wait-time profile graph (WTPG). The WTPG contains a node for each simulator instance, and a pair of opposite directed edges for each SplitSim channel connecting two simulators. The profiler annotates each edge with the fraction of cycles that the simulator at the source of the edge has spent waiting for synchronization messages from the destination simulator of the edge. As such, the graph visualizes “who waits for who”. Additionally, the profiler annotates each node with the total number of cycles that node spends waiting across simulators. Based on this value, we also color nodes on a spectrum from green to red, with red for nodes that spend few cycles waiting for other nodes, and green for nodes that spend many cycles waiting for other nodes. Typically, nodes that spend little time waiting, are the bottleneck simulators and will stand out in red. If in doubt, the edge labels allow users to confirm that their neighbors spend significant cycles waiting on them. Figure 5 shows an example of a WTPG for a SplitSim simulation.

3.4 Configuration and Orchestration

Finally, we are left with addressing the complexity of configuring and running a broad range of large-scale simulations. SplitSim addresses this with an orchestration framework. The orchestration framework aims to reduce the user configuration complexity by providing natural abstractions for

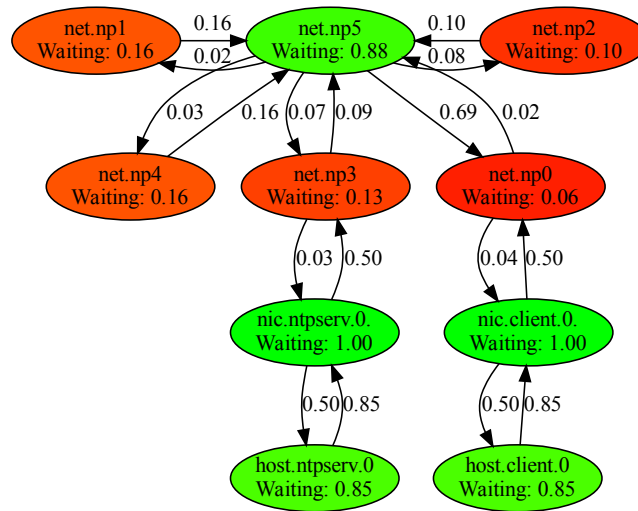


Fig. 5. Example of a generated wait-time-profile graph. Here the `net.np0` process is the immediate bottleneck, but `np1-3` are close behind, judging from their waiting numbers.

separately specifying the configuration of the simulated system from the implementation choices for how to simulate the system. Finally, SplitSim will apply the specified implementation choices and coordinate the execution of the simulation, including starting up each component simulator, connecting them up, collecting outputs, and, eventually, clean up after termination.

Crucially, with the SplitSim configuration abstractions, users can solve many simulation configuration tasks fully within SplitSim, without resorting to manually configuring specific simulators through their specific configuration mechanism. For such tasks, SplitSim abstractions also provide a level of portability, in fully separating system configurations from concrete simulator choices. At the same time, the SplitSim orchestration framework can easily be bypassed where necessary and users can resort to manually configuring specific simulators. The SplitSim orchestration aims to make easy tasks easy, and complex tasks possible.

3.4.1 System Configuration Abstraction. The goal for the SplitSim system configuration abstraction is to specify the *configuration of the simulated system* separate from concrete choices of how to simulate it. We represent the system configuration as a hierarchy of Python objects. At the root we have the `SystemConfiguration` object, that contains a list of all system components. A system component can be a host, a NIC, a switch, or a PCI or Ethernet link. Each component object carries the expected attributes. For example, a host object may specify the number of cores, memory, disk image, applications to run, IP address, etc. A link object specifies latency and bandwidth, along with its two endpoints. The key design consideration for this abstraction is to specify system characteristics while abstracting simulation details.

Since we use Python for defining SplitSim system configurations, users can use all python language features for assembling this configuration, in particular relying on loops for instantiating repeated patterns or using functions and modules for factoring out re-usable configuration parts. For example, for the experiments in this paper we use the same parameterizable large-scale network topology across multiple of our experiments and have defined this in a common function across experiments.

3.4.2 Implementation Choices. After a user has assembled a system configuration in the simulation Python script, the second step is to generate one or more different concrete simulation instantiations. Users instantiate a SplitSim simulation by choosing specific simulators and translating

the system configuration for the corresponding system components into configurations for these concrete simulators. We specify the resulting instantiated simulation using the existing SimBricks abstractions for describing interconnected instances of component simulators.

In general, there are many different instantiation strategies. Instead of trying to automate this inherently complex step with a one-size-fits-all approach, SplitSim instead opts for an extensible and flexible approach by merely providing library routines for common instantiation strategies. For example, one strategy we commonly use is to instantiate all hosts as separate processes for a specified host simulator, qemu or gem5, all NICs of a particular type, and simulate the whole network topology in one ns-3 process. A generalized version of this strategy instead first applies a partition function provided as a parameter to divvy up the network topology components into different partitions to run in separate ns-3 processes. For the network topology above, we have implemented a couple of different partition strategy functions that we use across the experiments in this paper.

As instantiated simulation configurations is just a regular SimBricks configuration, comprising SimBricks orchestration python objects, SplitSim users can manually modify this configuration afterwards when the need arises.

3.4.3 Running Simulations. Finally, to actually run SplitSim simulations, we leverage the existing SimBricks orchestration framework runtime. Since the instantiation above produces a SimBricks configuration as its output, this can directly be passed through for execution.

3.5 Workflow

The usage model and workflow of SplitSim are illustrated in Figure 2. The user begins by writing a configuration script that specifies the configuration of the system to be evaluated. Next, based on the system configuration, the user specifies possible simulation instantiation choices. Since it is difficult to predict bottlenecks and determine how to divide the simulation workload across processes in advance, the user typically prepares multiple candidate configurations with different fidelity and performance properties. Next, each configuration is executed with the SplitSim profiler for a short duration to evaluate performance. The profiler generates a WTPG for each configuration, informing the user of bottlenecks and idle components. Based on these insights, the user then refines the configuration script by further splitting bottleneck components into additional processes or consolidating idle ones. This iterative process continues until the workload is sufficiently balanced. Finally, the user executes the final configuration to collect the full simulation results.

4 Evaluation

In our evaluation we aim to answer the following questions:

- Does SplitSim efficiently support trade-offs between simulation time and computing resource requirement?
- Does SplitSim facilitate an effective trade-off between computational resource consumption and simulation accuracy?
- With a given configuration, does SplitSim facilitate the better resource utilization?
- How much user effort is required to configure and run simulations with SplitSim, and to explore the optimal settings of trade-offs?
- Does SplitSim efficiently simulate large-scale of networked systems while providing end-to-end visibility?

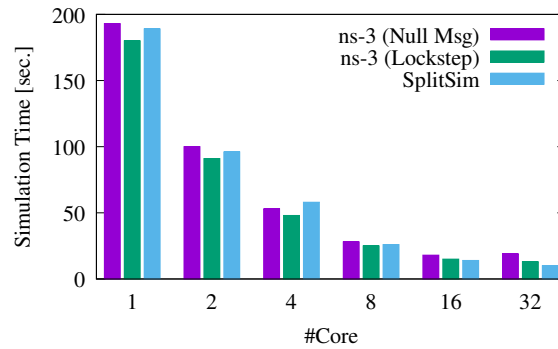


Fig. 6. Simulation time for SplitSim-parallelized ns-3 compared to ns-3 native parallelization.

4.1 Methodology

Our measurements are performed on machines with double Intel Xeon Gold 6336Y CPUs for a total of 48 physical cores and 256 GB RAM. For resource efficiency, we opt for the smallest simulations that substantiate each evaluation point. While SplitSim supports SimBricks proxies for distributed simulations and inherits their demonstrated scalability, by relying on mixed fidelity simulations for our evaluation we have not found the need to scale out to multiple machines.

Unless otherwise mentioned, we use the following simulation parameters. We configure up single-core hosts and qemu with instruction counting for time synchronization. Simulated hosts are configured with 4 GHz clock frequency and 1 GB of memory. For NICs we use the SimBricks `i40e_bm` simulator for the Intel X710 NIC.

4.2 Trade-Offs Between Simulation Time and Computing Resources

We first demonstrate how SplitSim enables flexible trade-offs between simulation time and computing resource requirements. There are two primary approaches: the first reduces simulation time by parallelizing existing sequential simulators and executing them on multiple cores; the second reduces computing resource usage by consolidating simulation components into a single process.

4.2.1 Parallelizing Existing Sequential DES. Simulators composed of multiple components and the channels that interconnect them can be parallelized using SplitSim building blocks. This applies not only to network simulators such as OMNeT++ and ns-3, but also to systems like gem5, which—though not a network simulator per se—models a networked architecture where multiple host components communicate through an interconnection network.

Figure 6 shows the simulation time of SplitSim-parallelized ns-3 compared to its native parallelization. In this experiment, we simulate a $k = 8$ FatTree topology consisting of 32 racks and 128 hosts. Within each rack, one server acts as a sink while the remaining hosts send packets to it. The network is partitioned into 1, 2, 4, 8, 16, and 32 components, where each component contains the corresponding number of racks, all connected to the partition that includes the spine switches. We then compare the simulation time of SplitSim-parallelized ns-3 with that of its native parallelization. As shown in Figure 6, SplitSim-parallelized ns-3 achieves comparable or slightly better speedup. At 32 partitions, SplitSim attains higher simulation time than the native approach, due to the bottleneck at the partition containing the spine switches, which must coordinate with all other partitions. We expect this performance to improve further by splitting the spine switches into multiple partitions with guidance from the SplitSim profiler.

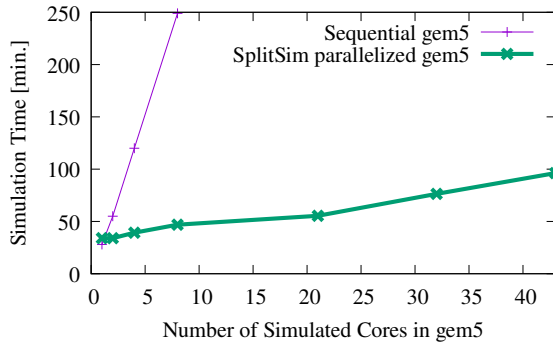


Fig. 7. Simulation time for SplitSim-parallelized multi-core gem5 compared to sequential gem5.

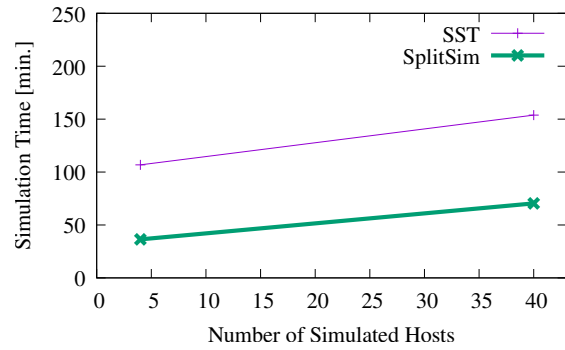


Fig. 8. Simulation time for multiple gem5 nodes connected through a network switch using SplitSim compared to SST.

The second case study is using SplitSim to parallelize gem5, a widely used architectural simulator. The gem5 architectural simulator [4] operates sequentially. Consequently, when simulating multi-core machines, the simulation time increases at least linearly with the number of simulated cores, as illustrated in Figure 7.

In our experiment, we configured gem5 with varying numbers of cores, each running a compute-intensive workload consisting of continuous arithmetic operations. Using SplitSim, we parallelized the simulation by separating each core, along with its associated L1 and L2 caches, into individual gem5 processes connected via SplitSim adapters. This parallelization required approximately 1,000 lines of additional code in gem5, implemented without intrusive modifications.

As shown in Figure 7, the parallelized version significantly reduces simulation time compared to the original sequential gem5. For an 8-core configuration, we observed approximately a 5 \times speedup. Moreover, the parallelized gem5 exhibits good scalability, with simulation time increasing by only a factor of 2 when scaling from 8 to 44 cores.

4.2.2 Comparison with SST. The efficiency of SplitSim parallelization is further highlighted through a comparison with the Structural Simulation Toolkit (SST) [18], a widely used parallel discrete-event simulator that supports modular composition of full-system simulations. In contrast to SST's use of global lockstep synchronization and MPI-based message passing, SplitSim employs loosely coupled synchronization and optimized shared-memory message passing. This design enables SplitSim to achieve significantly higher simulation speeds.

We configured a network composed of a variable number of gem5 nodes connected via a central network switch, with each client node transmitting UDP packets to a server. As shown in Figure 8, SplitSim outperforms SST with more than a 2 \times speedup.

4.2.3 Consolidation of Simulation Components. In addition to parallelization, SplitSim supports the consolidation of multiple simulation components into a single process, thereby reducing overall resource consumption. Specifically, users can configure components such as multiple host simulators or NICs to run within the same process under a unified event loop. This feature is particularly beneficial for balancing computational load across processes and optimizing resource utilization in constrained environments.

4.3 Trade-off Between Computational Resource Requirements and Fidelity

Next, we demonstrate how SplitSim enables trade-offs between simulation fidelity and computational resource through mixed-fidelity simulations. We further evaluate how these trade-offs affect

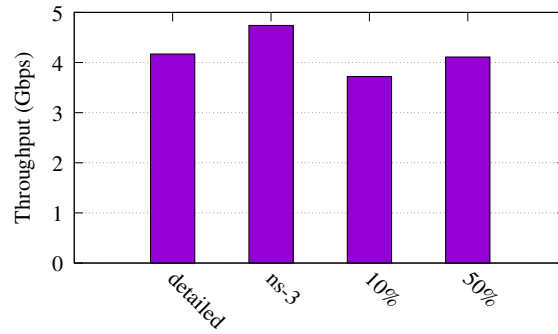


Fig. 9. Measured system throughput in detailed end-to-end simulation, protocol-level ns-3, and with mixed-fidelity of 10% and 50% detailed hosts.

simulation results and propose heuristic guidelines for applying mixed-fidelity simulations under varying fidelity tolerance requirements using a concrete case study.

In this experiment, we simulate a network system using a FatTree-4 topology [1] with 64 servers. We run 32 client-server pairs, where each client transmits TCP packets to its corresponding server, and we measure the client-side throughput as the performance metric.

Using mixed-fidelity simulation, we configure a subset of nodes with gem5 hosts and I40E NIC simulators to provide detailed modeling, while the remaining nodes are simulated as simplified dummy hosts in ns-3. The dummy nodes do not model end-host behavior and introduce no delays, which causes them to saturate links more aggressively than both physical testbeds and fully detailed simulations [11]. We treat the results from the fully detailed gem5 nodes as the ground truth.

Figure 9 presents the simulation results for three configurations other than the ground truth: a pure ns-3 simulation, and mixed-fidelity simulations with 10% and 50% of the hosts modeled in detail. With only 10% of the hosts simulated in detail, throughput deviates from the ground truth by approximately 10%. At 50% detailed hosts, the deviation drops to about 1%, whereas the pure ns-3 simulation exhibits a 12% deviation. Importantly, this improvement in accuracy comes with a significantly lower resource cost: the fully detailed simulation requires 129 CPU cores, while the 10% and 50% mixed-fidelity simulations require only 13 and 65 cores, respectively—representing 10× and 2× reductions in resource usage.

We also note that the placement of detailed hosts could affect the simulation outcome, and we leave a systematic exploration of this factor for future work.

4.4 SplitSim Profiler

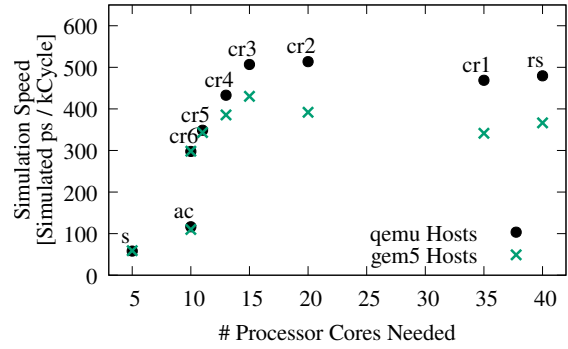
With SplitSim, users can split or consolidate simulation components to find an optimal balance between simulation speed and resource consumption. However, identifying the right partitioning strategy is not always straightforward. Poor decisions in component splitting may lead to increased resource inefficiency, even if simulation time is reduced. This is because, in parallel simulation, the overall performance is often constrained by the slowest component. Therefore, identifying the performance bottleneck is critical for optimizing simulation efficiency.

We demonstrate the SplitSim profiler can help users efficiently pinpoint bottleneck components with minimal effort. And provides insights that guide users in refining their partitioning strategies and improving overall resource utilization.

Complexity in Predicting Simulation Performance. We first explore different partitioning strategies as shown in Figure 10a for the 1 200 node network topology with background traffic from subsection 4.6. Here we connect a pair of qemu or gem5 hosts with two Intel x710 NICs. Figure 10b

Part.	Description
s	Whole network as one process.
ac	One process per aggregation block, plus one for the core switch.
crN	Aggregate N racks per process, +1 for the aggregation and core switches.
rs	One process per rack, one process each per aggregation and core switch.

(a) Partition strategies.



(b) Simulation speeds and core counts.

Fig. 10. Different network partition strategies and their resulting simulation speed and processor core requirements for system with qemu and gem5 hosts.

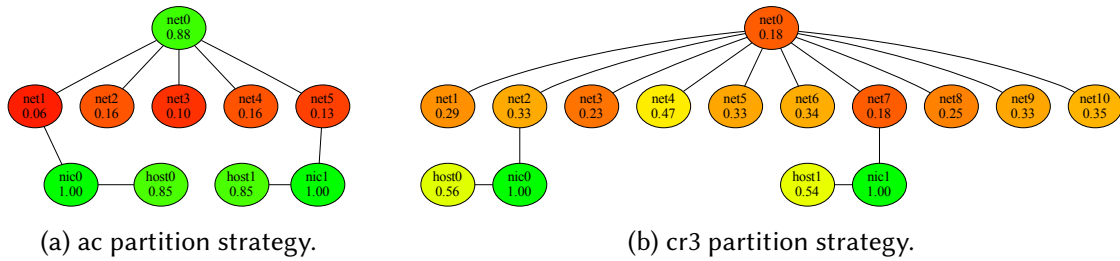


Fig. 11. SplitSim profile graphs for two partition strategies in Figure 10b with qemu hosts.

shows the partition strategies along with their achieved simulation speeds and cores used for the whole simulation (including 4 cores for hosts and NICs). The results show that different partitioning strategies achieve significantly different simulation speeds across strategies but also with qemu compared to gem5, in some cases even with identical number of cores. Additionally, the results show that past a point adding more cores results in lower simulation speeds again.

Profiling to Locate Bottlenecks. Next we pick the *ac* and *cr3* partition strategies and examine their SplitSim profile graphs in Figure 11. Here, we run the simulations for 5 min for the most reliable results, but we found typically even 60-90 s is sufficient to record the profile to inform partition decisions. Profiling and post-processing to generate the graphs are fully automatic, and simply require adding the flag to enable profiling when running SplitSim, and then running the post-processing script. For compactness and readability, we simplify the graph here to only show the node color representing the fraction of cycles each simulator spends waiting for messages in the SplitSim adapters.

High waiting cycles, shown in green, imply the simulator is not computation bound and thus not the bottleneck, while few waiting cycles, shown in red, imply a bottleneck. Figure 11a shows that for the coarse-grain *ac*, primary bottlenecks are the *ns-3* instances with the 6 racks, rather than the *ns-3* instance with the core, or the qemu or NIC instances. Thus, as expected, a much more fine-grain partition of the network into 15 processes with *cr3*, Figure 11b shows that here the bottleneck are starting to shift towards the two qemu instances.

Bottleneck Component Placement. Simulation performance is not only affected by how the user partitions system components into simulator processes, but also by how these are placed on the machine. Sometimes these results are not intuitive and difficult to predict a priori. For example,

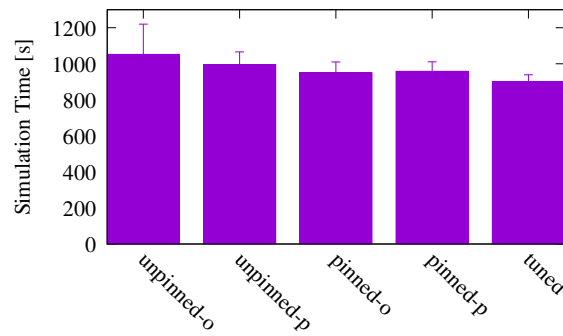


Fig. 12. Impact of CPU Pinning and Frequency Scaling. “Unpinned” refers to all simulators without CPU affinity. “O” and “P” represent the use of the ondemand and performance CPU frequency governors, respectively. “Tuned” denotes a configuration where bottleneck simulators are assigned the performance governor, while underloaded simulators use powersave.

during testing we found that spreading simulators across the two processor sockets resulted in faster simulation performance than pinning them onto one processor socket. We eventually found that this was because of the CPU frequency scaling, where using fewer cores per socket with others idle, allowed them to run at higher turbo boost frequencies.

SplitSim Profiling results can guide users in identifying these bottlenecks and in optimizing their placement and system configuration. To demonstrate this, we evaluate three configurations: (1) no explicit CPU core affinity (i.e., unpinned simulators), (2) simulators running under either the ondemand or performance CPU frequency governors, and (3) a tuned configuration where cores with bottleneck simulators are assigned the performance governor, while underloaded simulators are assigned the powersave governor.

As shown in Figure 12, simulation time varies significantly across these configurations. The tuned setup achieves a 14.5% speedup compared to the unpinned configuration using ondemand.

4.5 SplitSim Configuration and Orchestration

Finally, we demonstrate the flexibility and ease of use of SplitSim, by comparing the necessary effort for configuring and running some simulations used in our evaluation.

SplitSim Simulations are Easy to Configure. Even complex simulation configurations are relatively easy to configure in SplitSim. For example, the configuration for running all the clock synchronization simulations in subsection 4.6 comprises 252 lines of Python, 195 of which are responsible for generating configuration files and commands to run for Chrony, ptp4l, and CockroachDB. The other simulation configurations in this evaluation section are more compact. Other than the initial extension to ns-3 for PTP transparent clocks in switches, no changes outside of the python configuration are required for configuring these simulations.

Re-use of Configuration through Python. SplitSim configurations are just Python scripts; ordinary language features such as loops, functions, and modules, can be used as meta-programming for generating configurations. We commonly use loops to generate different structurally similar, configurations. We also abstract out common building blocks into re-usable python modules. For example, the large (parameterized) background network topology used in multiple experiments is defined in a separate python module of 195 lines and is imported and used for multiple of our simulations.

Running Simulations is Fully Automatic. After writing a SplitSim simulation configuration, execution, is fully automatic. The SplitSim orchestration framework starts processes, wires up channels between different simulator instances, collects output, and cleanly terminates simulations.

4.6 Case-Study: Clock Synchronization

As a case study, we aim to compare NTP vs PTP host clock synchronization accuracy and its effect on application performance for distributed systems that rely on clock-bounds for consistency [7, 16] in large-scale networks. We use a modified version of CockroachDB [12] that uses the dynamic clock bound from the Chrony NTP server [5] for its commit-wait period, used in prior work [9]. We configure the following end-to-end host machines: 2 CockroachDB replica servers, 4 CockroachDB clients running the social workload, and a single clock server, either NTP server or PTP grand master. Clients and server run Chrony, for PTP alongside ptp41. For the NTP configuration, we configure Chrony to synchronize to the NTP server. For the PTP configuration, we configure Chrony to use the local NIC's PTP hardware clock (PHC) as a reference clock.

We integrate these machines into a large-scale network topology comprising 1200 hosts total, 7 qemu hosts, and 1193 background hosts simulated in ns-3. The background hosts are randomized pairs of hosts performing bulk transfers. The network topology has a single core switch, connected through 100 Gbps links to 4 aggregation switches, that each connect to 6 racks with a ToR and 40 machines. We extended ns-3 with a switch that implements a PTP transparent clock (TC).

In the simulation, we measure the clock accuracy bound that Chrony reports on the servers. As expected, we see that PTP, with its NIC hardware timestamping and transparent clocks in switches, improves the clock bound from 11 μ s with NTP, to 943 ns with PTP. Note that this includes a full end-to-end simulation of the PTP synchronization, with ptp41 running on Linux, using the NIC hardware receive and transmit timestamping, as well as the transparent clock switch support adding corrections for queue residence time. As reported by prior work [9, 16], this improved clock bound improves application request throughput and latency for the application relying on commit-wait to ensure consistency. We measure a 38% throughput improvement for write operations, and a 15% reduction in latency for writes. This simulation simulates 20s in 175 min and 227 min for NTP and PTP respectively.

5 Looking Forward

In this paper we have introduced SplitSim, a system and methodology for enabling end-to-end evaluation for large scale systems in simulation for when physical testbeds are out of reach. SplitSim enables users to easily navigate the inherent trade-offs when fidelity, simulation time, and computational resources. In our evaluation we have shown SplitSim can simulate multiple hosts with full OS and application software stacks, NICs, as part of a large scale network of 1200 hosts, run this simulation on a single physical machine, and complete a 20s simulation run in under four hours. Finally, SplitSim drastically lowers the barrier to entry for such simulations, by enabling users to configure such simulations comprising multiple different simulators etc. without needing expertise for how to configure each and every simulator.

Navigating Fidelity Trade-offs. We have shown that mixed-fidelity simulation can significantly reduce both simulation time and computational resource usage while maintaining reasonable accuracy compared to full-fidelity simulation. However, finding the ideal mixed-fidelity configuration with maximal performance but adequate fidelity for a concrete evaluation use case a priori remains an open problem. In our evaluation we have relied on intuition to determine plausible configurations, by relaxing fidelity where it seems less critical, and then validated these configurations against much shorter full-fidelity simulations (where possible). We expect developing a robust methodology

for determining this a priori will require substantial measurement studies across a broad range of different systems, and simulation models, that we hope our community will develop over time. Until then the current process is somewhat laborious, but since evaluations in our community require many evaluations of similar system configurations and parameters, SplitSim still enables drastic time and resource savings.

Towards Automating Simulation Configuration. With SplitSim, we have introduced abstractions that separate configuration of the system, the “what”, from the choice of simulation configuration, the “how”. Further, through its profiler, SplitSim also provides a feedback mechanism to determine where problems lie. This leaves a typically large search space, but also an automated means of testing different points and obtaining feedback. In future work, based on these building blocks, we plan to develop automated methods to assist users in configuring large-scale simulations. Additionally, we expect that by collecting this data across many configurations, it will become possible to automatically provide implementation suggestions a priori based on the system configuration.

6 Acknowledgments

We thank our shepherd, Soudeh Ghorbani, and the anonymous reviewers for their constructive and insightful feedback. We also thank Praneeth Balasubramanian for his contributions to the SplitSim implementation on ns-3 and OMNeT++. Finally, we thank Jakob Gorgen and Jonas Kaufmann for their support in upstreaming SplitSim functionality to the SimBricks codebase.

References

- [1] Mohammad Al-Fares, Alexander Loukissas, and Amin Vahdat. 2008. A scalable, commodity data center network architecture. *SIGCOMM Comput. Commun. Rev.* 38, 4 (Aug. 2008), 63–74. doi:10.1145/1402946.1402967
- [2] Mohammad Alian, Daehoon Kim, and Nam Sung Kim. 2016. Pd-Gem5: Simulation Infrastructure for Parallel/Distributed Computer Systems. *IEEE Computer Architecture Letters* 15, 1 (Jan. 2016), 41–44.
- [3] Songyuan Bai, Hao Zheng, Chen Tian, Xiaoliang Wang, Chang Liu, Xin Jin, Fu Xiao, Qiao Xiang, Wanchun Dou, and Guihai Chen. 2024. Unison: A Parallel-Efficient and User-Transparent Network Simulation Kernel. In *Proceedings of the Nineteenth European Conference on Computer Systems (Athens, Greece) (EuroSys '24)*. Association for Computing Machinery, New York, NY, USA, 115–131. doi:10.1145/3627703.3629574
- [4] Nathan Binkert, Bradford Beckmann, Gabriel Black, Steven K. Reinhardt, Ali Saidi, Arkaprava Basu, Joel Hestness, Derek R. Hower, Tushar Krishna, Somayeh Sardashti, Rathijit Sen, Korey Sewell, Muhammad Shoaib, Nilay Vaish, Mark D. Hill, and David A. Wood. 2011. The Gem5 Simulator. *SIGARCH Computer Architecture News* 39, 2 (Aug. 2011), 1–7.
- [5] chrony project. 2024. chrony. <https://chrony-project.org/>. Retrieved Oct 7, 2025.
- [6] Florin Ciucu and Jens Schmitt. 2012. Perspectives on network calculus: no free lunch, but still good value. *SIGCOMM Comput. Commun. Rev.* 42, 4 (aug 2012), 311–322. doi:10.1145/2377677.2377747
- [7] James C. Corbett, Jeffrey Dean, Michael Epstein, Andrew Fikes, Christopher Frost, J. J. Furman, Sanjay Ghemawat, Andrey Gubarev, Christopher Heiser, Peter Hochschild, Wilson Hsieh, Sebastian Kanthak, Eugene Kogan, Hongyi Li, Alexander Lloyd, Sergey Melnik, David Mwaura, David Nagle, Sean Quinlan, Rajesh Rao, Lindsay Rolog, Yasushi Saito, Michal Szymaniak, Christopher Taylor, Ruth Wang, and Dale Woodford. 2013. Spanner: Google’s Globally Distributed Database. *ACM Transactions on Computer Systems* 31, 3 (aug 2013).
- [8] Kaihui Gao, Li Chen, Dan Li, Vincent Liu, Xizheng Wang, Ran Zhang, and Lu Lu. 2023. DONS: Fast and Affordable Discrete Event Network Simulation with Automatic Parallelization. In *Proceedings of the ACM SIGCOMM 2023 Conference*. 167–181.
- [9] Jacob Gunnarsson and Fabian Lindfors. 2022. *Utilizing highly synchronized clocks in distributed databases*. Master’s thesis. Lund University.
- [10] INET Authors. 2022. INET Framework. <https://inet.omnetpp.org/>. Retrieved Feb 2, 2022.
- [11] Hejing Li, Jialin Li, and Antoine Kaufmann. 2022. SimBricks: End-to-End Network System Evaluation with Modular Simulation. In *2022 ACM SIGCOMM Conference on Data Communication (Amsterdam, Netherlands) (SIGCOMM)*.
- [12] Fabian Lindfors and Jacob Gunnarsson. 2022. GitHub fabianlindfors/cockroach. <https://github.com/fabianlindfors/cockroach/>. Retrieved Oct 7, 2025.

- [13] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. 2002. Simics: A full system simulation platform. *IEEE Computer* 35, 2 (Aug. 2002), 50–58.
- [14] Alian Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. 2017. distgem5: Distributed simulation of computer clusters. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software* (Santa Rosa, CA) (ISPASS).
- [15] nsnam. 2022. ns-3 | a discrete-event network simulator for internet systems. <https://www.nsnam.org/>. Retrieved Feb 2, 2022.
- [16] Oleg Obleukhov and Ahmad Byagowi. 2022. How Precision Time Protocol is being deployed at Meta. <https://engineering.fb.com/2022/11/21/production-engineering/precision-time-protocol-at-meta/>.
- [17] George F Riley and Thomas R Henderson. 2010. The ns-3 network simulator. In *Modeling and tools for network simulation*. Springer, 15–34.
- [18] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. Cooper-Balis, and B. Jacob. 2011. The Structural Simulation Toolkit. *ACM SIGMETRICS Performance Evaluation Review* 38, 4 (March 2011), 37–42.
- [19] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and accurate microarchitectural simulation of thousand-core systems. *ACM SIGARCH Computer architecture news* 41, 3 (2013), 475–486.
- [20] Qingqing Yang, Xi Peng, Li Chen, Libin Liu, Jingze Zhang, Hong Xu, Baochun Li, and Gong Zhang. 2022. DeepqueueNet: Towards scalable and generalized network performance estimation with packet-level visibility. In *Proceedings of the ACM SIGCOMM 2022 Conference*. 441–457.
- [21] Qizhen Zhang, Kelvin K. W. Ng, Charles Kazer, Shen Yan, João Sedoc, and Vincent Liu. 2021. MimicNet: Fast Performance Estimates for Data Center Networks with Machine Learning. In *Proceedings of the 2021 ACM SIGCOMM 2021 Conference (Virtual Event, USA) (SIGCOMM '21)*. Association for Computing Machinery, New York, NY, USA, 287–304. doi:10.1145/3452296.3472926
- [22] Kevin Zhao, Prateesh Goyal, Mohammad Alizadeh, and Thomas E Anderson. 2023. Scalable Tail Latency Estimation for Data Center Networks. In *20th USENIX Symposium on Networked Systems Design and Implementation (NSDI 23)*. 685–702.

Received June 2025; accepted September 2025