SimBricks: End-to-End Network System Evaluation with Modular Simulation

Hejing Li
Max Planck Institute for Software Systems (MPI-SWS)
Saarbrücken, Germany
hejingli@mpi-sws.org

Jialin Li
National University of Singapore
Singapore
lijl@comp.nus.edu.sg

Antoine Kaufmann
Max Planck Institute for Software Systems (MPI-SWS)
Saarbrücken, Germany
antoinek@mpi-sws.org

Abstract

Full system "end-to-end" measurements in physical testbeds are the gold standard for network systems evaluation but are often not feasible. When physical testbeds are not available we frequently turn to simulation for evaluation. Unfortunately, existing simulators are insufficient for end-to-end evaluation, as they either cannot simulate all components, or simulate them with inadequate detail.

We address this through modular simulation, flexibly combining and connecting multiple existing simulators for different components, including processor and memory, devices, and network, into virtual end-to-end testbeds tuned for each use-case. Our architecture, SimBricks, combines well-defined component interfaces for extensibility and modularity, efficient communication channels for local and distributed simulation, and a co-designed efficient synchronization mechanism for accurate timing across simulators. We demonstrate SimBricks scales to 1000 simulated hosts, each running a full software stack including Linux, and that it can simulate testbeds with existing NIC and switch RTL implementations. We also reproduce key findings from prior work in congestion control, NIC architecture, and in-network computing in SimBricks.

CCS Concepts

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

Keywords

Modular Simulation, End-to-End Evaluation, Network Systems

ACM Reference Format:
Hejing Li, Jialin Li, and Antoine Kaufmann. 2022. SimBricks: End-to-End Network System Evaluation with Modular Simulation. In *ACM SIGCOMM 2022 Conference (SIGCOMM '22), August 22–26, 2022, Amsterdam, Netherlands.* ACM, New York, NY, USA, 17 pages. https://doi.org/10.1145/3544216.3544253

1 Introduction

Our community expects research ideas to be implemented and evaluated as part of a complete system "end-to-end" in realistic conditions. End-to-end evaluation is important as many factors in each system component affect the overall behavior in subtle and unpredictable ways.

SIGCOMM '22, August 22–26, 2022, Amsterdam, Netherlands

ACM ISBN 978-1-4503-9420-8/22/08.
https://doi.org/10.1145/3544216.3544253

Yet evaluation in full physical testbeds is frequently infeasible. Work might require cutting edge commercial hardware that is not yet available at the time of publication [32, 33, 35, 50], develop hardware extensions to existing proprietary hardware [51], or propose entirely new ASIC hardware architectures [9, 13, 25, 27, 34, 36, 53, 55]. The trend towards increasingly specialized hardware, including SmartNICs, programmable switches, and other accelerators, further exacerbates this. Finally, work on network protocols and congestion control necessitates evaluation in large scale networks with hundreds to thousands of hosts.

When a full evaluation in a physical testbed is not possible, simulation has long offered an alternative. In networking, we use ns-2 [43], ns-3 [44], and OMNeT++ [57] to evaluate protocols and algorithms; computer architects rely on system simulators such as gem5 [8], while hardware designers employ RTL simulators such as Modelsim [52] or Verilator [54]. While network systems do benefit from these simulators [4, 28, 41], they do not enable end-to-end evaluation, as no existing simulator simulates all required components in a testbed: hosts, devices, and the full network.

In this paper, we demonstrate how to enable end-to-end network system simulation by combining different simulators to cover the necessary functionality. Instead of building a new simulator, throwing away decades of work, we connect existing and new simulators – for hosts, hardware devices, and networks – into full system simulations capable of running unmodified operating systems, drivers, and applications. Existing simulators, however, are standalone and not designed to be combined with other simulators. To achieve modular end-to-end simulation, we thus need to overcome three technical challenges: 1) no interfaces to connect simulators together, 2) efficient, scalable, and correct synchronization of simulator clocks, and 3) combining mutually incompatible simulation models.

We present the design and implementation of *SimBricks, a modular simulation framework for end-to-end network system simulations.* SimBricks defines interfaces for interconnecting simulators based on natural component boundaries in physical systems, specifically PCIe and Ethernet links. Individual component simulators run in *parallel* as separate processes, and communicate via message passing only between connected peers through optimized shared memory queues. With this message transport, we co-design a protocol that leverages simulation topology and latency at component boundaries for *efficient and accurate synchronization* of simulator clocks. For scaling out simulations across physical hosts, we introduce a proxy to forward messages over TCP or RDMA.

Currently, SimBricks integrates QEMU [46] and gem5 [8] as host simulators, Verilator [54] as an RTL hardware simulator for hardware devices, and ns-3 [44], OMNeT++ [57], as well as the Intel

Tofino simulator [23] for network simulation. Further, we have integrated open source RTL designs for the Corundum FPGA NIC [16] and the Menshen switch pipeline [58] to showcase SimBricks's generality. We have also implemented fast behavioral simulators, e.g. for the Intel X710 40G NIC [22], and ported the FEMU NVMe SSD model [31] into SimBricks. In combination, these simulators enable a broad range of end-to-end configurations for different use-cases.

In our evaluation, we demonstrate that SimBricks enables end-to-end simulation of existing network systems at small and large scales. We also reproduce key results from congestion control [3], in-network compute [33], and FPGA NIC design [16] in SimBricks. SimBricks obtains more realistic results compared to ns-3 in isolation (§3). SimBricks also scales to 1000 hosts and NICs with only a 14% increase in simulation time compared to a 40-host simulation (§7.4). Finally, SimBricks provides deep visibility and control of low-level system behaviors, facilitating evaluation and performance debugging (§8.1).

We make the following technical contributions:

- *Modular architecture for end-to-end system simulation* (§5.1) combining host, device, and network simulators.
- *Co-designed message transport and synchronization mechanism for parallel and distributed simulations* (§5.5, §5.2) leveraging pairwise message passing to efficiently ensure correct simulation, even at scale.
- SimBricks, a *prototype implementation* of our architecture (§6) with integrations for existing and new simulators.

SimBricks is available open source at https://simbricks.github.io
This work does not raise any ethical issues.

2 Simulation Background

Simulators employ techniques such as discrete event simulation, binary translation, and hardware virtualization, to simulate system components at various scales and levels of detail. Network simulators, such as ns-2 [43], ns-3 [44], and OMNeT++ [57], use discrete event simulation to model packets traversing network topologies. Computer architecture simulators, such as gem5 [8], QEMU [46], and Simics [37], simulate full computer systems capable of running unmodified guest software, including operating systems, with different and sometimes configurable degrees of detail. These simulators also include I/O devices, but often only implement the minimum features for basic functionality. Hardware RTL simulations, such as xsim [59] and Verilator [54], help test and debug hardware designs cycle by cycle against testbenches. In all three cases *individual components are simulated in isolation.*

Advantages. The main motivation for simulation is that a physical implementation is often not feasible. Simulations are also *portable* as they decouple the simulated system from the host system. Many are deterministic (with explicit seeds for randomness), providing *reproducible results.* Simulators are also *flexible*; implemented as software they can be modified, and frequently offer parameters representing a broad range of configurations. Finally, simulations provide great *visibility*, and can log details about the system, without affecting behavior.

Disadvantages. Simulations also have some common drawbacks. *Long simulation times* are common – architectural simulators often only simulate hundreds or thousands of system cycles a second [26, 55], and simulating a few milliseconds of a large scale topology in ns-3 can take many hours. Different simulators strike different trade-offs between accuracy and simulation time, depending on the intended use-case. Finally, simulation results are only as good as the simulator, and may not be representative unless *validated* against a physical testbed.

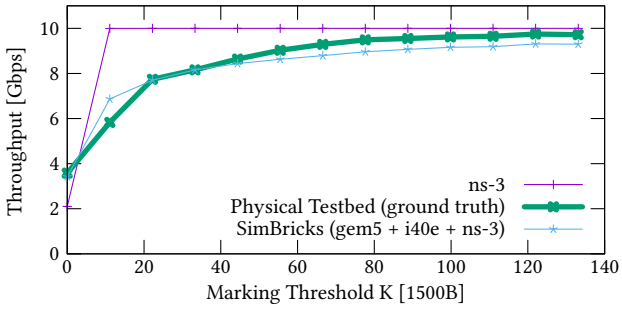

Figure 1: Throughput for two dctcp flows in ns-3, a physical testbed, and a SimBricks end-to-end simulation.

Comparison to Emulation. Emulations replicate externally visible behavior of a system without modeling internal details, and typically run at close to interactive speeds. For example, Mininet [30] emulates OpenFlow networks with multiple end-hosts running real Linux applications at near native speed on a single physical host, by using Linux containers and kernel network features. However, as emulation uses wall-clock time, it only works as long as all components can keep up in real time. Simulations, in contrast, rely on virtual time which can slow down without affecting simulated behavior. Additionally, emulation does not model internals of a system that could affect system behavior, e.g., interactions between NIC and drivers. As such, emulation is primarily useful for interactive testing or performance evaluation when fidelity is not crucial.

3 Systems Research Challenges

Systems research faces additional challenges that complicate using simulation during prototyping and evaluation.

Not end-to-end. First and foremost, *no existing simulator covers all required components for network systems with sufficient features and detail*, precluding end-to-end evaluation. While existing simulators cover individual components, such as computer architecture, hardware devices, and networks, they only do so in isolation with no mechanism for combining them into complete systems. As a result, we are left with non-end-to-end "piecemeal" evaluation, where different components are evaluated in isolation [4, 20, 41].

We illustrate the pitfalls of piecemeal evaluation by comparing dctcp [3] congestion control behavior in the ns-3 network simulator to a physical testbed. As network speed increases and bottlenecks move to end-hosts, congestion control incurs small variations in timing in the host hardware and software which can affect behavior [3, 29, 40]. However, ns-3 only models network and protocol behavior, and as a result, does not capture these factors. We set up two clients and two servers sharing a single 10G bottleneck

link with a 4000B MTU, and one large TCP flow generated by iperf for each client-server pair. Fig. 1 shows the throughput for varying dctcp marking thresholds K. The marking threshold balances queuing latency and throughput; a lower threshold reduces queue length but risks under-utilizing links. ns-3 underestimates the necessary threshold [3] to achieve line rate, as it does not model host processing variations, particularly processing delay caused by OS interrupt scheduling. Only an end-to-end evaluation of the full system captures such intricacies.

Not scalable. Network and distributed systems frequently require evaluation on clusters beyond tens of hosts to demonstrate scalability. But for most simulators, already long simulation times increase super-linearly with the size of the simulated system, making simulation of a large network system an infeasible task.

Not modular. Using simulators for systems research often requires extending existing simulators with additional functionality, e.g., adding a new NIC to an architecture simulator. These extensions are tied to a particular simulator, as different simulators lack common internal interfaces, This complicates apples-to-apples comparisons for future work that may use a different simulator, e.g., to simulate a host with a different NIC, forcing the same simulator to be used throughout the project cycle. Finally, this tight integration complicates the implementation and releasing of such extensions, as they often require maintaining a fork of the full simulator.

4 Modular Simulation

We argue that *end-to-end simulations can be effectively assembled from multiple different interconnected and synchronized simulators for individual components.* To demonstrate this, we present SimBricks, a new modular simulation framework that aims to provide end-to-end network system simulation.

End-to-end simulations are better. Returning to the dctcp example from earlier, Fig. 2 shows the simulation setup that produces the result shown in Fig. 1. We combine four instances of gem5 with four instances of the Intel i40e NIC simulator we developed, each pair connected through PCIe; all NIC simulators are in turn connected to an instance of ns-3. The gem5 instances are running a full Ubuntu image with unmodified NIC drivers and iperf. Fig. 1 shows that our SimBricks simulation approximates the behavior of the physical testbed much more closely than ns-3, and yields the same insight. We conclude that end-to-end evaluation with SimBricks improves accuracy for network system evaluation over non-end-to-end simulators.

4.1 Design Goals

To address the challenges for using simulations in systems research, (§3), we have the following design goals for SimBricks:

- **End-to-end**: simulate full network systems, with hosts, existing or custom devices, network topologies, and the full software stack, including unmodified OS and applications.
- **Scalable**: simulate large network systems consisting of tens or hundreds of separate hosts and devices.
- **Fast**: keep simulation times as low as possible.
- **Modular**: enable flexible composition of simulators, where components can be added and swapped independently.

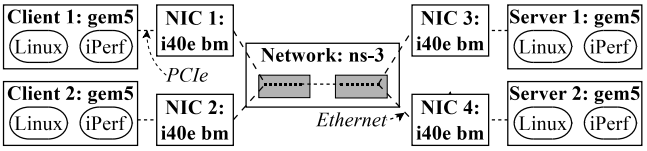

Figure 2: SimBricks configuration for the dctcp experiment in Fig. 1, combining gem5, ns-3, and an Intel NIC simulator. Each simulator runs in a separate process.

- **Accurate**: preserve accuracy of constituent simulators, correctly interface and synchronize components to behave equivalent to a monolithic simulator with the same models.
- **Deterministic**: keep end-to-end simulation deterministic when all individual simulators are deterministic.
- **Transparent**: provide deep and detailed visibility into end-to-end performance without affecting simulation behavior, to support debugging and performance analysis.

4.2 Technical Challenges

Achieving our design goals incurs the following challenges:

Simulation interconnection interfaces. Unfortunately, existing simulators are standalone and provide no suitable interfaces for interconnecting with other external simulators. Moreover, enabling modular "plug-and-play" configurations, where components can be independently swapped out, requires common, well-defined interfaces between different component types.

Scalable synchronization and communication. Individual component simulators maintain their own virtual simulation clocks that progress at different rates. To accurately connect simulators, we need to synchronize their virtual clocks. However, this synchronization comes at a performance cost, especially with increasing system scale. For example, we measure a 3.7× increase in runtime for the dist-gem5 [42] simulator when scaling from 2 to 16 simulated hosts, due to synchronization overhead (§7.3.1). Prior work shows synchronization overhead can be reduced by sacrificing accuracy and determinism through lax synchronization. [12, 19]. Since this violates two of our design goals, we do not consider this.

Incompatible simulation models. Finally, different simulators often employ mutually incompatible simulation models. For example, QEMU has a synchronous device model where calls in device code block until complete, while ns-3 schedules asynchronous events to model networks, and Verilator simulates hardware circuits cycle by cycle. We therefore need an interface compatible with all of these simulation models.

4.3 Design Principles

We address these challenges through four design principles:

Fix natural component simulator interfaces. To enable modular composition of simulators, SimBricks defines an interface for each *component type* (§5.1). We base these interfaces on the *point-to-point* component boundaries in real systems: PCI express (PCIe) connects today's hardware devices to servers, while network devices typically connect through Ethernet networks. We choose

these interfaces as a starting point, but our approach generalizes to other interconnects and networks. These component interfaces form narrow waists, decoupling innovation on both sides: To integrate a simulator into SimBricks, developers need to add an adapter that implements the component interface, without needing to modify other simulators. We assume a static topology of components throughout a simulation.

Loose coupling with message passing. Instead of tightly integrating multiple simulators into one simulation loop, SimBricks runs component simulators as separate processes that communicate through message passing (§5.1) across our defined interfaces. This drastically simplifies integrating simulators into SimBricks, as we treat each simulator as a black-box that only needs to implement our interfaces. Using asynchronous message passing also maximizes compatibility with different simulation models: Discrete event and cycle-by-cycle simulations can issue requests and process responses at the scheduled times, while blocking simulations can block till the response message arrives — for peer simulators this is transparent. Message passing channels also provide inspection points for debugging and tracing system behavior without modifying component simulators.

Parallel execution with shared memory queues. We run simulators in parallel on different host cores and connect them through optimized shared-memory queues (§5.2). As simulators run on separate cores and only communicate when necessary, this avoids unnecessary cache-coherence traffic and hidden scalability bottlenecks. These mechanisms allow us to (i) *scale up* to large simulations: Instead of simulating the complete system in one simulation instance, we simulate different components of the system in separate simulators running in parallel (§5.3). (ii) *scale out* with distributed simulations: We use a separate proxy that transparently forwards messages on shared memory queues over the network to and from simulators running on remote hosts (§5.4).

Accurate and efficient synchronization. We ensure accurate simulation through correct time synchronization among simulators, but with minimum runtime overhead. *Synchronization is optional*, and the user can disable it for unsynchronized emulations. For this, we combine three key insights: 1) *Global synchronization is not necessary* as our simulator boundaries at point-to-point interfaces limit which simulators directly communicate. As long as events at these pairwise interfaces are processed in a time-synchronized manner, simulation behavior is correct. 2) *Latency at component interfaces provides slack*, reducing frequency of component having to wait for others to coordinate [12] and thus synchronization overhead. An event sent at time T only arrives at $T + \Delta$, as our component interfaces have an inherent latency Δ in physical systems that we model. 3) By *inlining synchronization with efficient polled message transfers*, synchronization overheads can be minimized and sometimes completely avoided. We combine these observations to design an accurate, efficient, and scalable synchronization mechanism for parallel end-to-end simulations (§5.5).

4.4 Non-Goals

SimBricks is not a panacea. We explicitly view the following aspects as out of scope for this paper and leave them for future work:

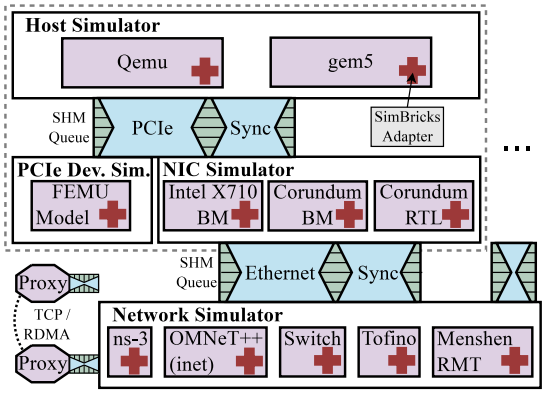

Figure 3: SimBricks architecture. *Double hour glass* with narrow waists between hosts and devices, and NICs and networks.

Accelerating component simulators. SimBricks does not generally aim to reduce simulation times for individual component simulators as we only modify simulators to add SimBricks adapters. Simulation times for synchronized end-to-end SimBricks simulations are at least as high as the slowest component simulator, and may increase due to synchronization and communication overhead. However, in a few cases, SimBricks interfaces enable developers to decompose an existing component simulator into multiple smaller parallel pieces, thereby reducing simulation time (§7.3.2).

Avoiding need for validation. To obtain representative results, users need to validate component simulation configurations in SimBricks as with any other simulation. Validation effort is no higher in SimBricks than it would be in an equivalent monolithic simulator, as SimBricks forwards timestamped events accurately from one simulator-internal interface to another without modifying them (except for the configured link latency). We expect, however, that SimBricks could reduce validation effort by allowing users to re-combine validated component simulator configurations without validating from scratch. (§9)

Interfacing semantically incompatible simulators. While SimBricks can combine simulators that use different models for simulation, it cannot bridge semantic gaps between simulators. For example, SimBricks cannot connect a gem5 host sending packets through an RTL NIC with a flow-based network simulator. Such conversions may be possible in special cases, but are specific to the concrete simulators, and as such could be integrated as part of a SimBricks adapter in such a simulator.

5 Design

Using our design principles, we have built SimBricks, a modular, end-to-end simulation framework shown in Fig. 3. In this section, we detail the design of SimBricks, including simulator interfaces, fast message transport, techniques to scale up and out to larger simulations, and the synchronization mechanism.

PCIe: Device → Host	
Message Type	**Message Fields**
INIT_DEV	PCI vendor, device id, class, subclass, revision, # of MSI vectors, # of MSI-X vectors, table/PBA bar and offset
DMA_READ, DMA_WRITE	request ID, memory address, length, payload data (optional)
MMIO_COMPL	request ID, payload data (optional)
INTERRUPT	interrupt type, MSI/MSI-X: vector #, legacy: level

PCIe: Host → Device	
Message Type	**Message Fields**
DMA_COMPL	request ID, payload data (optional)
MMIO_READ, MMIO_WRITE	request ID, BAR # and offset, length, payload data (optional)
INT_STATUS	interrupts enabled: legacy, MSI, MSI-X

Ethernet: NIC ↔ Net / Net ↔ Net	
Message Type	**Message Fields**
PACKET	packet length, packet data

Figure 4: SimBricks's simulator interfaces: PCIe between host and device, and Ethernet between network components.

5.1 Component Simulator Interfaces

SimBricks achieves modularity through well-defined interfaces between component simulators: Host simulators connect to device simulators through a PCIe interface; NIC and network simulators interconnect through an Ethernet interface. This results in a double hourglass architecture (Fig. 3) with narrow waists at component boundaries. In physical systems both interfaces are asynchronous and incur propagation delay (Δ_i). We replicate both aspects.

5.1.1 PCIe: Host-Device Interface

PCIe itself is a layered protocol, ranging from the low-level physical layer to the transactional layer for data operations. We define SimBricks's host-device interface (Fig. 4) based on the PCIe *transactional layer*, and abstract away physical attributes of the PCIe link with simple parameters – link bandwidth and latency. Low-level complexity such as encoding and signaling are unnecessary for most system simulations and would incur substantial cost and complexity for each simulator. Should future use-cases need to model this, a detailed PCIe simulator could be integrated as an interposed component (§5.3).

Discovery and Initialization. A key PCIe feature is that hosts can enumerate and identify connected devices and the features they support. To this end, our interface defines the INIT_DEV message for registering device simulators with the host simulator. The device simulator includes device information in the message, such as the PCI vendor, device identifiers, base address registers (BARs), the number of MSI(-X) interrupt vectors, and addresses of the MSI-X table and PBA. The host simulator uses this information to expose a corresponding PCIe device to the system.

Data transfers: MMIO & DMA. PCIe data transfers are symmetrical: both sides can initiate reads and writes, which the other side completes. SimBricks's PCIe interface defines DMA_READ / WRITE messages for DMA transfers initiated by device simulators, and MMIO_READ / WRITE for MMIO accesses initiated by host simulators. As in PCIe, all data transfer operations are *asynchronous*. Once a request is finished, the device simulator issues a MMIO_COMPL completion message, while the host simulator adapter sends a DMA_COMPL. PCIe allows multiple outstanding operations and only guarantees that they will be issued to the memory system in the order of arrival. Completion events, however, may arrive out-of-order. To match completions with outstanding requests, all requests carry an identifier that the receiving simulator includes in the response.

Interrupts. Our interface supports all PCIe interrupt signaling methods: legacy interrupts (INTX), message signaled interrupts (MSI), and MSI-X. Physical PCIe devices implement MSI (including configuration, masking, and generating signalling operations) completely on the device side. To reduce repeated implementation effort in device simulators and integration challenges in host simulators, we instead opt to keep this functionality inside the host simulator. Device issues INTERRUPT messages to either trigger an interrupt vector for MSI(-X) or to set interrupt pin state for INTX. To support devices that require knowledge about which interrupt mechanisms the OS has enabled, our interface provides the INT_STATUS message which the host simulator sends on configuration changes.

5.1.2 Ethernet: Network Component Interface

In SimBricks's network interface, we similarly abstract away low-level details of the Ethernet standard, and only expose Ethernet frames, as PACKET messages, to NIC and network simulators. A PACKET message carries the length of the packet alongside packet payload, but omits CRCs to reduce overhead as none of our network simulators models them and most NICs strip them after validation. If future network or NIC simulators require CRCs, their SimBricks adapter can transparently generate and strip the checksums, as we currently do not model data corruption. We leave support for hardware flow control as future work.

5.2 Inter-Simulator Message Transport

SimBricks runs component simulators as separate processes communicating through message passing. Thus, efficient inter-process communication is critical for the overall performance. We use optimized shared memory queues with polling for efficient message transport between simulators. For parallel processes on separate cores, shared memory queues enable low-latency communication with minimal overhead [5, 7]. Between any pair of *communicating* simulators, SimBricks establishes a bidirectional message channel consisting of a pair of unidirectional queues in opposite directions. During channel initialization, SimBricks uses a Unix socket to provide a named endpoint for connection setup and for communicating queue parameters and shared memory file descriptors.

SimBricks uses concurrent, circular, single-producer and consumer queues. They comprise an array of fixed-sized, cache line

aligned message slots. The last byte in each slot is reserved for metadata: one bit indicating the current owner of the slot (`consumer` or `producer`) and the rest for the message type. As queues are single-producer and single-consumer, we store the tail pointer *locally* at the producer and the head pointer at the consumer. Consumers poll for a message in the next slot, until the ownership flag indicates `consumer`. After processing the message the consumer resets the ownership flag. Producers similarly wait for the next slot to be available, fill it, and switch the ownership flag.

The SimBricks message transport design avoids cache coherence overhead unless it is fundamentally necessary. Since head and tail pointers are local to consumer and producer respectively, only accesses to shared message slots result in coherence traffic. Moreover, as long as a consumer does not poll in between the producer writing a message to the corresponding slot and setting the ownership bit, all coherence traffic carries necessary data from producer to consumer [5]. We include additional detail and pseudocode in §A.2.

5.3 Scaling Up with Decomposition

SimBricks can scale to larger simulations by adding more component simulators. For instance, a network simulator connecting to many devices may become a bottleneck as it needs to synchronize with all peers. We leverage the SimBricks architecture to improve scalability, by decomposing the network simulator into multiple processes that connect and synchronize via SimBricks Ethernet interfaces. Other simulators, such as a gem5 simulated host, can be accelerated in a similar fashion by decomposing into connected components. We will demonstrate the scalability benefit of our decomposition approach in §7.4.

5.4 Scaling Out with Proxies

Running simulators in parallel on dedicated cores maximizes parallelism, but the number of available cores in a single machine limits simulation size. Message passing and modular simulation in SimBricks enables us to scale out simulations by partitioning components to multiple hosts and replacing message queues between simulators on different hosts with network communication. However, directly implementing this in individual component simulators has two major drawbacks. First, it increases the complexity for integration, as each simulator adapter needs to implement an additional message transport. Second, it increases communication overhead in component simulators, leaving fewer processor cycles for simulators and increasing simulation time. To avoid these drawbacks, we instead implement network communication in proxies. SimBricks proxies connect to local component simulators through existing shared memory queues and forward messages over the network to their peer proxy which operates symmetrically. This requires an additional processor core for the proxy on each side, but is fully transparent to component simulators and does not increase their communication overhead.

5.5 Simulator Synchronization Mechanism

To ensure accurate interconnection of component simulators, we design a synchronization mechanism that that guarantees correctness while minimizing overhead, even when scaling to large simulations.

5.5.1 *Naive Synchronization Mechanisms do not Scale*

A conceptual straw-man for synchronizing components are global barriers at each time step, keeping simulators in lockstep. When components are connected by communication links with non-zero latency, frequency of global barriers can be reduced by dividing simulation time into *epochs* no larger than the lowest link latency. Global barriers are only required at epoch boundaries, since all cross-component events will be delivered after the end of the current epoch [1, 42, 47]. Unfortunately, epoch-based synchronization still relies on non-scalable global barriers across all simulators, with the barrier frequency determined by the lowest link latency in the whole simulation, incurring substantial synchronization overhead.

5.5.2 *Scalable synchronization in SimBricks*

We avoid global synchronization while *guaranteeing accurate simulator interconnection* by relying on properties specific to the SimBricks architecture. Fig. 5 shows pseudocode for the SimBricks synchronization protocol.

Enforcing message processing times is sufficient. In SimBricks, all communication between simulators is explicit through message passing along statically created point-to-point channels. Thus, the only requirement for accurate simulation is that *messages are processed at the correct time* [10, 11]. Additional synchronization does not affect the simulation, as simulators cannot otherwise observe or influence each other. To enforce this guarantee, senders tag messages with the time when the receiver must process the message. For determinism, simulators with multiple peers must order messages with identical timestamps consistently.

Pairwise synchronization is sufficient. All SimBricks message passing channels are point-to-point and statically determined by the simulation structure. This is where we differ from most prior synchronization schemes: they do not assume a known topology and thus require global synchronization. SimBricks only needs to implement pairwise synchronization, between each simulator and its a priori known peers [10].

Per-channel message timestamps are monotonic. Our message queues deliver messages strictly in order. Since each SimBricks connection between two simulators incurs a propagation latency $\Delta_i > 0$, a message sent at time T over interface i arrives at $T + \Delta_i$. Assuming simulator clocks advance monotonically, message timestamps on each channel are thus monotonic.

Message timestamps ensure correctness. A corollary of monotonic timestamps is that a message with timestamp t is an implicit promise that no messages with timestamps $< t$ will arrive on that channel later. Therefore, once a simulator receives messages with timestamps $\geq T$ from *all* its peers, it can safely advance its clock to T without more coordination.

Ensuring liveness with sync messages. The above conditions ensure accuracy, but do not guarantee liveness. Simulations can only make progress when every channel carries at least one message in each direction in every Δ_i time interval [10, 11]. To ensure progress, we introduce `SYNC` messages that simulators send if they have not sent any messages for $\delta_i \leq \Delta_i$ time units. `SYNC` messages allow connected peers to advance their clocks in the absence of data messages. In our simulations we set $\delta_i = \Delta_i$; lower values of δ_i are

```
procedure Init
    for if in interfaces do
        SyncTimer(if)
        msg ← PollMsg(if)
        Reschedule(msg.timestamp, RxTimer, msg, if)
procedure SyncTimer(if)
    msg ← AllocMsg(if)
    msg.type ← SYNC
    SendMsg(msg)
procedure RxTimer(msg, if)
    if msg.type ≠ SYNC then
        ProcessMsg(msg)
    msg ← PollMsg(if)
    Reschedule(msg.timestamp, RxTimer, msg, if)
procedure SendMsg(msg, if)
    msg.timestamp ← T + Δ_if
    EnqueueMsg(msg, if)
    Reschedule(T + δ_if, SyncTimer)
```

Figure 5: SimBricks synchronization protocol pseudocode for a discrete event-based simulator. Reschedule schedules a callback for the specified time, cancelling earlier instances. ProcessMsg and SendMsg interface with the upper layer PCI or Network protocol. Δ_{if} is the link latency and δ_{if} the synchronization interval.

valid, but we have not found configurations where the benefit of more frequent clock advances outweighed the cost of sending and processing additional SYNC messages.

Link latency provides synchronization slack. Non-zero link latencies further reduce synchronization overhead, since not even peer simulators need to execute in lockstep. Specifically, a message sent at T allows its peer to advance to $T + \Delta_i$. At that point, the peer's clock is guaranteed to lay between $T - \Delta_i$ (otherwise the local clock would not be at T) and $T + \Delta_i$. Different channels in a SimBricks configuration can use different Δ_i values. While synchronized simulations are fundamentally only as fast as the slowest component, this slack improves efficiency by absorbing small transient variation in simulation speed, without immediately blocking all simulators.

6 Implementation

SimBricks is implemented in 4206 of C/C++ and 2102 lines of Python for core functionality, 5348 lines for adapters in existing simulators, and 4556 lines for new simulators we built (details in §A.3).

6.1 Core SimBricks Components

Libraries. To reduce integration effort for simulators, we develop a common library that implements the SimBricks messaging interfaces, and helper functions to parse and generate synchronization messages. We also implement a helper library with common C++ components for behavioral NIC simulators (`nicbm`) that we use for our NIC simulators below.

Proxies. To scale out SimBricks simulations, we have implemented two proxies, one uses TCP sockets for network communication and the other one uses RDMA. Both implement adaptive batching by forwarding multiple messages at once if more than one is available in the queue. The RDMA proxy minimizes communication latency and CPU overhead by directly writing messages to remote queues.

Orchestration. Configuring and running SimBricks simulations is a challenge due to the multitude of interconnected components involved. We streamline simulation setups with our orchestration framework. Users can assemble complete simulations in compact python scripts, and the framework is responsible for running individual components (details in §A.1).

6.2 Host Simulation

We have integrated two host simulators, gem5 and QEMU, that are capable of running unmodified operating systems and applications. For both, we implement the SimBricks adapter as a regular PCIe device within the simulator's device abstractions.

gem5. gem5 is a flexible full system simulation with configurable level of detail for memory and CPU. We use version v20.0.0.1, extend it with a patch for Intel DDIO support [2], and implement support for the functional and timing memory protocols. The functional protocol is blocking, i.e., it expects device accesses and DMA to synchronously return results, and does not model timing. The timing protocol models accesses as asynchronous request and response messages. To reduce simulation time, we can configure gem5 to boot up with a fast functional CPU, and then switch to a detailed synchronized CPU. We also implement an Ethernet adapter to connect the built-in NICs in gem5 to SimBricks for comparison.

QEMU. We use QEMU version 5.1.92 with KVM CPU acceleration for fast functional simulation. We also implement support for synchronized simulation with instruction counting (`icount`), in which QEMU controls the rate of instruction execution relative to a virtual clock. The key challenge is modelling MMIO delays, as QEMU's device interface does not model timing and expects accesses to return immediately. We work around this by aborting execution of the instruction from the MMIO handler and stopping the virtual CPU, only re-activating it when the SimBricks PCIe completion event arrives. QEMU will then re-try the instruction. Unfortunately we have found that this QEMU version is no longer fully deterministic even with instruction counting.

6.3 NIC Simulation

We integrate three NIC simulators, a detailed hardware RTL model, and two less detailed but faster behavioral simulators.

Corundum RTL. To demonstrate realistic RTL device simulation, we use the unmodified Verilog implementation of the open source Corundum FPGA NIC [16]. We use Verilator [54] to simulate the `interface` module implementing Corundum's data path, including RX, TX, descriptor queues, checksums, and scheduling. As Verilator cannot simulate vendor IP Corundum uses for PCIe, DMA, and Ethernet, we implement them directly in the C++ testbench.

Corundum behavioral. To enable a fair comparison with other simulators, we also implement a fast behavioral simulator for Corundum in C++. Both Corundum simulators are fully compatible with the unmodified upstream Linux driver [17].

Intel i40e behavioral. Many recent network systems require a modern NIC compatible with Linux or kernel-bypass frameworks such as DPDK [15]. We implement a behavioral simulator for the common `i40e` Intel 40G X710 NIC. This simulator is compatible with unmodified drivers, and it implements important NIC features such as multiple descriptor queues, TCP and IP checksum offload, receive-side scaling, large segment offload, interrupt moderation, and support for MSI and MSI-X.

6.4 Network Simulation

ns-3 and OMNeT++. To integrate with ns-3.31, we implement a SimBricks Ethernet adapter class extending `NetDevice`, the ns-3 base abstraction for host network interfaces. When receiving packets from our Ethernet interface, the adapter pushes them to the connected network channel, and vice-versa. The adapter also implements our synchronization protocol (Fig. 5). We integrate OMNeT++ with INET [21] analogously.

Ethernet switch. We also implement a fast simulator for a basic Ethernet switch. In the simulation loop, the switch polls packets from each port, performs MAC learning, switches each packet to the corresponding egress port(s) according to the MAC table, and sends synchronization messages as necessary.

Tofino. We integrate the Tofino [24] simulator provided by Intel [23], as the most popular programmable switch. This simulator includes a cycle accurate model of the switch pipeline and an approximate model for queuing. The simulator is closed source, communicates through Linux Kernel virtual Ethernet interfaces (`veth`), and only allows minimal control over timing. To implement a synchronized adapter, we parse the output log of the simulator and generate packet timestamps accordingly.

Menshen RTL. Finally, we integrate the Verilog implementation of the Menshen RMT pipeline [58] using Verilator and the C++ Ethernet MAC adapter we implemented for Corundum.

6.5 Limitations

Incompatible simulation models. We do not support the gem5 atomic memory protocol where memory operations, including DMA and MMIO, are implemented as synchronous function calls that return how long the operation should take. This is incompatible with SimBricks's asynchronous PCIe interface. For example, while the SimBricks PCIe adapter is waiting for an MMIO completion message, no other events, such as incoming DMA requests can be scheduled and executed.

Single-core hosts. Both gem5 and synchronized QEMU simulate multiple cores sequentially, resulting in a super-linear increase in simulation time. As host simulator internals are orthogonal, we pragmatically opt to restrict our evaluation to single-core hosts. The scalable x86 simulators we found [18, 39, 49] only simulate applications and cannot run operating systems, precluding end-to-end simulation. As future work, we envision applying our techniques to scale out existing full system simulators, as modern multi-cores are essentially networked systems [6] with message latencies.

7 Evaluation

We now evaluate if SimBricks meets our design goals (§4.1):

- Can SimBricks *modularly* combine simulators into *end-to-end* simulations? How do these simulations perform? (§7.2)
- How *efficient* is the SimBricks synchronization mechanism? How does the overhead compare to prior approaches? (§7.3.1)
- Can SimBricks enable *faster* simulations by breaking down large simulators into smaller, parallel simulators? (§7.3.2)
- How do larger SimBricks simulations *scale* on a single physical host and distributed across multiple physical hosts? (§7.4)
- Does SimBricks *accurately* combine simulators? (§7.5)
- Are SimBricks simulations *deterministic*? (§7.6)

7.1 Experimental Setup

Unless otherwise stated we use the following setup: We run simulations on physical hosts with two 22-core Intel Xeon Gold 6152 processors at 2.10 GHz with 187 GB of memory, hyper-threading disabled, and 100 Gbps Mellanox ConnectX-5 NICs.[1] All simulated hosts have a single core and 8 GB of memory, and each runs Ubuntu 18.04 with kernel 5.4.46 where we disabled unneeded features and drivers to reduce boot time. All device drivers and applications are unmodified. For synchronized QEMU we set a clock frequency of 4GHz. For gem5, we use `DDR4_2400_16x4` memory and the `TimingSimple` CPU model, which simulates an in-order CPU with the timing memory protocol, and configure cache sizes and latencies to match those of the testbed. We set gem5 parameters (e.g., in-order CPU clock frequency of 8 GHz [2]) to achieve the same effective instruction execution performance as a representative physical testbed [28], for a Linux networking benchmark at 1.3 cycles/inst = 0.43 ns/inst. Further, we set the PCIe latency, Ethernet link latency and synchronization interval all to 500 ns, network bandwidth to 10 Gbps, and frequency for the Corundum RTL model to 250 MHz.

7.2 SimBricks is Modular

Navigating speed-accuracy trade-offs. We start by evaluating modular combinations of component simulators in SimBricks. As a workload, we use the `netperf` TCP benchmark to run a 10s throughput test (`TCP_STREAM`) followed by a 10s latency test (`TCP_RR`) between two simulated hosts. We focus on four configurations for common systems research use-cases: debugging and performance evaluation of hardware and software prototypes. Debugging HW & SW is most productive when fast and interactive, while accurate performance is not the primary concern. Here we combine QEMU with KVM for fast host simulation, our fast switch model, and either the `i40e` NIC for SW testing or Verilator with Corundum as a HW example. Performance evaluation on the other hand requires accurate results, but it can tolerate longer simulation times. We use a detailed gem5 host simulator and ns-3 for SW performance evaluation, while choosing a less detailed but time-synchronized QEMU simulator for benchmarking our HW prototype.

Our results in Tab. 1 confirm the expected trade-off between simulation time and simulator detail: simulation times range from 31s to 18 hours. The results show that, SimBricks can effectively help navigate this trade-off by only using detailed simulators when details matter for the use-case. Even combining fast QEMU-kvm

[1] The testbed only affects simulation time and unsynchronized experiments.
[2] Gem5 also supports an out-of-order CPU, but with $4-6\times$ higher simulation time, so we use the TmingSimple CPU as a compromise.

Use-case Simulator Combination	netperf T'put	Latency	Sim. Time
SW debugging	4.37 G	71 μs	00:00:32
QEMU-kvm + behavioral i40e NIC + behavioral switch			
SW perf. evaluation*	8.92 G	20 μs	12:49:46
gem5 + behavioral i40e NIC + ns-3			
HW debugging	81 M	3.4 ms	00:00:31
QEMU-kvm + Corundum Verilog + behavioral switch			
HW perf. evaluation*	6.55 G	32 μs	04:13:10
QEMU-timing + Corundum Verilog + behavioral switch			

Table 1: SimBricks configurations for different use-cases, with measured simulation time and application performance. Configurations with * are synchronized and deterministic, while the others are unsynchronized emulation.

with an unsynchronized RTL simulation is fast enough (31s) to test and debug the full system. Modularity also allows us to late bind simulator choices, e.g. if we later realize that QEMU-timing is not sufficiently accurate, we can replace it with gem5 without additional changes.

All combinations are functional. Besides these four configurations, we also evaluated the full cross-product of simulator choices (§6) and confirm SimBricks supports all combinations (subset of performance results in §A.4).

SimBricks interfaces are general. SimBricks interfaces are generic and serve as narrow waists between simulators. To further demonstrate its generality, we extracted gem5's `e1000` Intel NIC model, adapted it to SimBricks's PCIe interface without other modifications, and verified that it is compatible with gem5 and QEMU. To show that SimBricks's PCIe interface generalizes beyond NICs, we have adapted FEMU [31]'s NVMe SSD model from their QEMU fork into a separate simulator. This simulator also works in combination with QEMU and gem5.

7.3 SimBricks is Fast

We now show SimBricks does not significantly slow down simulators through synchronization, and can even speed up simulations through decomposition into parallel components.

7.3.1 Synchronization

Overhead. We measure synchronization overhead by comparing simulation time for gem5 standalone and in SimBricks. The experiment does not use the network, but for synchronization, we connect the gem5 to `i40e` NIC in SimBricks and to our switch. We first compare a low-event workload in gem5: executing `sleep 10`. The simulation takes 2.25 min standalone and 2.91 min in SimBricks, a 30% overhead. This is the worst case – gem5 is almost exclusively handling SimBricks synchronization events (every 500 ns), as the CPU is mostly halted. For a high-event workload we use `dd` to read from `/dev/urandom` to keep the CPU busy. This simulation takes 100.26 min standalone and 101.06 min in SimBricks, a mere 0.8% overhead. *SimBricks incurs manageable synchronization overhead, and does not significantly slow down already slow simulations.*

Comparison to dist-gem5. Next, we compare to dist-gem5 [42] which interconnects multiple gem5 instances and employs conventional epoch-based global synchronization over TCP. We configure 2 to 32 instances of gem5 that communicate pairwise using iperf, through the `e1000` NIC in gem5 and a single switch. For SimBricks we use our gem5 Ethernet adapter to connect to our switch model. Our simulation time measurements in Fig. 6 show that *SimBricks is more efficient than dist-gem5, especially with increasing scale.* SimBricks reduces simulation time by 27% for 2 hosts, and by 74% for 32 hosts.

Sensitivity to link latency. SimBricks synchronization overhead is linked with the configured link latency, which places a lower bound on sync message frequency. We measure how link latency affects synchronization overhead, with a pair of gem5 hosts running `netperf` for 1 s of throughput and latency measurements each, connected to `i40e` NICs and a shared switch. We vary the configured PCIe latency and sync interval, and report our results in Fig. 9. While synchronization time does increase, *lowering the link latency by three orders of magnitude (from 1 μs to 1 ns) only increases simulation time by 59%*, demonstrating that SimBricks can effectively parallelize simulations across low-latency interconnects.

7.3.2 Decomposition for Parallelism

Extracting NIC from gem5. When connecting synchronized simulators, the best SimBricks can achieve is to not slow them down beyond the slowest component simulator. However, SimBricks enables developers to decompose monolithic simulators into connected components (§7.2) running in parallel, thereby accelerating simulation. We evaluate this by comparing two gem5 configurations in SimBricks: first, gem5 with the built-in `e1000` NIC connected via our Ethernet adapter, and second, gem5 connected to our `i40e` NIC model through the PCIe interface. In both cases we run a pair of hosts connected to our switch model. The first configuration takes 350 minutes, while the second only takes 138 minutes: *Parallelism from the external NIC simulator reduces simulation time by 60%.*

Network simulator as scalability bottleneck. Network simulators are potential scalability bottlenecks in SimBricks, as they often connect many NICs, while hosts and NICs typically only connect one and two peers, respectively. To demonstrate this bottleneck, we develop a packet generator as a dummy NIC that implements the SimBricks Ethernet interface and the synchronization mechanism. The dummy NIC simply injects packets at a configured rate. We now measure simulation time for 2 and 32 dummy NICs connected to one switch for 1 second of virtual time. First we set the packet rate to 0 (to only measure synchronization overhead) and measure an increase from 2.6 s to 17.6 s of simulation time. Next, we set the packet rate to 100 Gbps on each NIC, and measure the simulation time increases from 12.6 s to 211.6 s. This experiment confirms that *a single network simulator can become a bottleneck for fast simulations.* We have so far not observed this outside of this microbenchmark.

Parallelizing network simulation. To address this bottleneck in SimBricks, we can decompose the network into multiple network simulators carved up at natural boundaries (e.g. switches or groups thereof). We demonstrate this by modifying the microbenchmark to divide the 32 hosts to 4 "ToR" switches, connected through a fifth "core" switch. With this configuration, the simulation time for packet

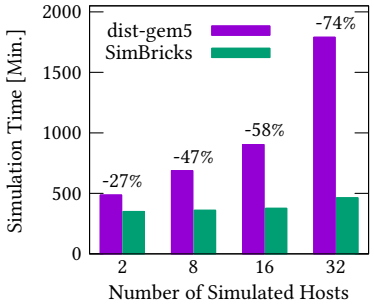

Figure 6: dist-gem5 vs. SimBricks.

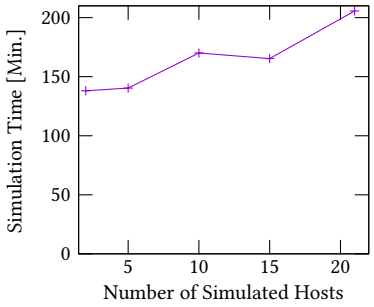

Figure 7: SimBricks local scalability.

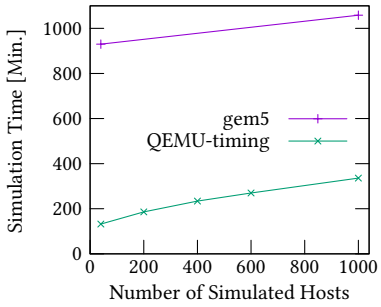

Figure 8: Distributed scalability.

rate 0 is 9.6 s down by 45% compared to the single switch setup, and 96.8 s at 100 Gbps packet rate, a 53% reduction. *Decomposing network simulators, therefore, can effectively reduce simulation time at scale.*

7.4 SimBricks is Scalable

We now evaluate scalability for local and distributed simulations.

7.4.1 Scaling Up

First, we measure simulation time as we vary the number of simulated gem5 hosts and `i40e` NICs connected to a single switch, running on a single physical host. We set up one server and a variable number of client hosts, running the same UDP iperf benchmark. To avoid overloading the server, we fix the aggregate throughput to 1 Gbps. The results in Fig. 7 show the simulation time increases with the number of clients, from 138 min with 2 hosts to 205 min with 21 hosts (48% increase).

Surprisingly, the longer simulation time is *not* caused by scalability bottlenecks in SimBricks synchronization. Instead, we discovered that this increase is due to thermal throttling of our host CPU slowing down all cores as more active. To confirm this, we run multiple independent instances of the 1-client experiment and measure how this affects simulation time. When running 4 independent instances of the 2-host simulations (5 cores each), using a total of 20 cores in the same NUMA node, the simulation takes 171 min. This matches the runtime of the 10-host simulation above, which uses 21 cores in total. We conclude that *SimBricks scales at least to the moderate cluster sizes typical for many of our evaluations.*

7.4.2 Scaling Out

We now move on to SimBricks simulations running across multiple physical hosts, using our RDMA and TCP proxies (Fig. 11).

Overhead of distributed simulation. First we compare performance for local simulations to equivalent distributed simulations with the SimBricks proxies, to measure overheads. We use two qemu-kvm hosts running `netperf` connected to `i40e` NICs which connect to the same switch. Locally, this unsynchronized simulation yields a throughput of 4.4 Gbps, and a latency of 71 μs. Next we distribute the simulation by running one pair of QEMU and NIC on a second server and proxying the Ethernet connection to the switch running locally. With the sockets proxy the latency increases to 305 μs and throughput remains constant, and with RDMA both remain constant. Next we measure simulation time for the same configuration but with QEMU timing and gem5, and find that simulation time does not change with either proxy. We conclude that *SimBricks proxies are no bottleneck for synchronized simulations.*

Large-scale memcache cluster. To evaluate scalability to larger systems, we next run multiple distributed simulations ranging from 40 to 1000 simulated hosts, on 1 to 26 physical servers. We run these simulations on Amazon ec2 `c5.metal` (spot) instances, with 96 hyperthreads each, and 20 Gbps network connectivity in a single proximity placement group. We simulate a varying number of racks of 40 hosts with `i40e` NICs and a top of rack (ToR) switch each, that then connect to a single core switch, as shown in Fig. 11. We assign the core switch and each rack to a dedicated server. A separate sockets proxy pair (Amazon ec2 does not offer RDMA) connects each ToR to the core switch. We run `memcached` on half of the hosts in each rack, and the `memaslap` client on the other half. Each client randomly connects to the 20 servers on the same rack, and to 20 random servers in other racks.

Fig. 8 shows the measured simulation time for 10 s of virtual time as we increase the number of hosts. From one rack and 40 hosts to 25 racks and 1000 hosts, simulation time with gem5 hosts increases by 13.8% from 15.5 h, to 17.6 h. With QEMU-timing, simulation time increases from 2.2 h to 5.6 h by 2.5×. With profiling we found the cause to be QEMU's dynamic binary translation. When an instance misses in its code cache and has to recompile a block, the instance blocks for a while. While rare, at scale these occurrences grow more frequent, and slow down other hosts due to synchronization. We conclude that *SimBricks scales to simulate systems with 100s of hosts.*

7.5 SimBricks is Accurate

We now show *SimBricks Ethernet and PCIe interfaces accurately connect and synchronize simulators.* For Ethernet, we first run a pure ns-3 simulation of two communicating nodes connected by a network link with our default parameters, and log packet timestamps on each node. Next, we repeat the experiment with two ns-3 instances each containing one node and a SimBricks Ethernet adapter, and connect the two. For PCIe, we run two gem5 instances running netperf with the built-in `e1000` NIC connected through the SimBricks Ethernet adapter to a switch. We rerun this experiment with our standalone version of gem5's `e1000` connected to both simulators through the SimBricks PCIe adapter. In both cases we find that the timestamped logs match *exactly*, demonstrating the correctness of our synchronization.

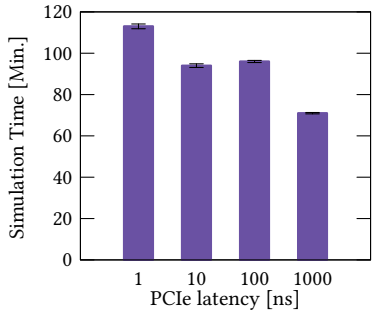

Figure 9: Sensitivity of SimBricks simulation time to link latency.

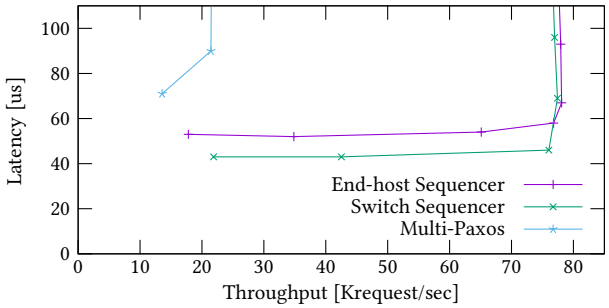

Figure 10: NOPaxos in SimBricks with a Tofino switch sequencer and with a sequencer on a simulated host.

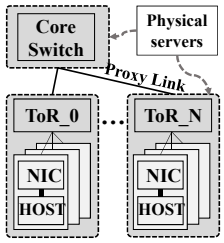

Figure 11: Large scale simulation configuration.

7.6 SimBricks is Deterministic

Finally, we verify that *SimBricks simulations with deterministic component simulators yields deterministic end-to-end simulations.* To this end, we have repeated the two configurations combining only deterministic simulators in Tab. 1 5 times on different machines. We then compared event timestamps in the simulation logs and found that they match *exactly*.

8 SimBricks for System Evaluation

Finally, we show SimBricks end-to-end simulations can aid evaluation, by providing more visibility and control than a physical testbed, and by accurately simulating unavailable hardware.

8.1 Use-Case: NIC Hardware Architecture

Using Corundum as an example, we show that SimBricks simulations can provide insights that are challenging to obtain from physical testbeds. The original Corundum evaluation shows significantly lower throughput for a 1500B MTU compared to the ConnectX-5 NIC they compare to. While developing our Corundum simulators, we found the root cause reason for this. Corundum relies on reading the head index registers of receive descriptor queues to identify new entries, while for most other NICs, drivers instead directly poll descriptors in memory. MMIO reads stall the processor until the device returns a result, while with DDIO descriptor reads typically hit in the L3 cache. For CPU-bound workloads this degrades performance.

Leveraging simulation visibility & flexibility. Our debugging effort was greatly facilitated by the simulator logs provided by SimBricks. Synchronized simulations can produce detailed logs without affecting system behavior. We leveraged this to trace PCI activity, NIC activity, and CPU activity, and combined those into an end-to-end view of the RPC latency. We further confirm this by doubling the simulated PCIe latency to 1 μs in gem5 with the Corundum and Intel behavioral simulators. When PCIe latency doubles, Corundum throughput reduces by 21.2%, while the Intel NIC throughput remains unchanged. Our experience demonstrates that *simulators can offer greater visibility and the flexibility to change key parameters that are fixed in physical systems.*

8.2 Use-Case: In-Network Processing

Work leveraging programmable switches for application acceleration requires end-to-end measurements for a meaningful evaluation. However, many of these works rely on functionality not (yet) available in off-the-shelf hardware at publication time. We use Network-Ordered Paxos (NOPaxos) [33] as an example to demonstrate that SimBricks can serve as a virtual testbed for such systems. NOPaxos introduces a new network-level primitive, the Ordered Unreliable Multicast (OUM), which requires a single sequencer device in the network. Implementing the sequencer in a programmable network switch offers the best performance. However, as the required network hardware was not yet available, the authors relied on sequencer emulation on a network processor or an end-host implementation. We implement switch support for OUM both in ns-3 and the now available Tofino simulator, and combine them with gem5 and the Intel NIC. On the simulated hosts, we run the unmodified NOPaxos open source code.

Reproducing results. We use SimBricks to simulate two NOPaxos configurations: a P4 switch sequencer running on Tofino, and an end-host sequencer implementation. Similar to the original work, we also simulate the classic Multi-Paxos state machine replication protocol. We compare the throughput-latency curves (Fig. 10) to figure 6 in the NOPaxos paper, where the switch sequencer configuration of NOPaxos achieves a latency of 110 μs, while the end-host sequencer configuration has 35% higher latency; both configurations achieve similar throughput (230 K/s). The original paper also shows that NOPaxos (switch sequencer) achieves a 370% increase in throughput and a 54% reduction in latency compared to Multi-Paxos. In SimBricks we find a lower baseline latency of 43 μs for the switch sequencer setup, and 23% higher latency for the end-host sequencer configuration. This is expected as the authors used a slower network processor to emulate switch functionality. We also find that both systems saturate at the same throughput of 78 K/s. The lower throughput is because we are measuring on a single-core host, where application and packet processing share a core. We confirmed this in a physical testbed by disabling all but one core, and measured throughput within 10%. When comparing to Multi-Paxos running in SimBricks, NOPaxos with switch sequencer attains a 270% throughput increase and 40% latency reduction. We conclude that *SimBricks can accurately evaluate in-network processing systems.*

9 Looking Forward

Validation. To obtain representative simulation results, users have to validate simulators and configuration parameters against physical testbeds. While SimBricks cannot avoid this, we argue that our approach can reduce validation effort. Instead of validating each combination of simulators, components can be validated individually and then composed. This enables users to combine previously validated component configurations into a full system. We propose a public repository of validated component simulator configurations to simplify re-use. To ensure validity of these configurations over time, we imagine a continuous-integration system, periodically re-running configurations and recording the results.

Beyond networking. While we evaluate SimBricks for network systems, our approach generalizes beyond networking. We have already demonstrated that our PCIe interface can support an NVMe simulator. Going forward, simulating PCIe attached accelerators, which are also attracting growing interest in our community, should not require changes to SimBricks. SimBricks can also be easily extended with additional components or interfaces, such as CXL [14]. We expect the emergence of further use-cases as architecture and systems researchers continue to investigate specialized hardware.

Evaluating ASIC designs. Finally, we see evaluation of systems that include new ASIC components as a driving use-case in the future. While small ASIC designs with lower clock rates can often be evaluated in physical testbeds with FPGAs, this is not possible for larger designs or designs with fast clock speeds. SimBricks, on the other hand, can simulate ASIC RTL with arbitrary frequencies, although FPGA accelerated RTL simulations [26] may be required for manageable simulation times.

10 Related Work

Parallel & distributed simulation. dist-gem5 [42] and pd-gem5 [1] connect multiple gem5 instances for parallel and distributed simulations and synchronize with global barriers. Graphite [39] also parallelizes a multi-core simulation across cores and machines, but uses approximate synchronization where causality errors are possible. Similar to gem5, Simics [37] also supports full system simulation and runs unmodified operating systems and applications, and multiple Simics processes can be connected to simulate networked systems. SimBricks connects multiple different simulators together using fixed interfaces, and synchronizes them accurately with a synchronization protocol that leverages the simulation structure.

ns-3 adds support for distributed simulation in version 3.8 [45]. It uses a similar conservative look-ahead protocol with explicit synchronization for correctness,and relies on the Message Passing Interface (MPI) to connect multiple ns-3 processes. MPI decouples ns-3 from the choice of message transport, directly supporting distributed simulations over various interconnects, but incurs the cost of this abstraction in every process. SimBricks instead closely couples synchronization and adapters to our optimized shared memory queues (implementation is inlined from shared headers), minimizing communication overhead in simulator adapters. SimBricks scales out through proxies that decouple individual simulators from the choice and overhead of distributed transport (RDMA, sockets), at the cost of typically one core per physical simulation host.

Co-simulation of multiple simulators. gem5 supports the integration of systemC code [38] to implement hardware models, by linking them into the gem5 binary and embedding the systemC event loop with the gem5 event loop. SimBricks instead interconnects multiple heterogeneous simulators with potentially completely different simulation models. The Structural Simulation Toolkit (SST) [48] is a modular simulation framework for HPC clusters, uses a parallel discrete event simulation with global epoch synchronization, and defines common interfaces to link in various *component* simulators. Unlike SimBricks, SST requires deep integration of simulators into one simulation loop resulting in integration challenges. SST does also not define fixed component interfaces for specific components, instead compatibility is up to individual simulators.

Full system emulation. Prior work on emulation has provided a path closer to end-to-end evaluation without matching physical testbeds. Mininet [30] emulates network topologies and hosts through Linux networking and container features, running real applications and using the host kernel for protocol processing. ns-3 direct code execution (DCE) [56] integrates a Linux Kernel instance as a libOS into ns-3 and connects its network interface to ns-3 topologies. Both systems offer lower run-times compared to SimBricks, but at the cost of not modeling low-level details, such as caches or PCIe interactions with devices, and other bottlenecks on the physical system. Finally, other work has relied on emulating NIC or switch functionality on dedicated processors, while running the rest of the system natively [27, 33]. Simulations incur higher run-times but can control the level of details in the model, and enable adjustment of relative performance of components by operating on virtual time.

11 Conclusion

We described and evaluated SimBricks, a novel modular framework enabling full end-to-end simulation of network systems by combining multiple tried-and-true simulators for different system components. SimBricks is fast and scalable, and accurately and deterministically connects and synchronizes simulators. We also demonstrated SimBricks can replicate key findings from prior work, including congestion control, in-network compute, and NIC hardware architecture. End-to-end simulations are a valuable tool for systems research, especially in the era of specialized hardware.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and feedback, and the anonymous artifact evaluation committee for reviewing our artifact. We also thank Jeff Mogul, Peter Druschel, Simon Peter, Trevor E. Carlson, Aastha Mehta, Ming Liu, Katie Lim, Pratyush Patel, for their input on earlier drafts of this paper. Keon Jang contributed the dctcp experiment idea and physical testbed implementation, and joined many discussions on SimBricks. We thank Jonas Kaufmann for his help with preparing our artifact and open source release, and Zhiqiang Xie for profiling SimBricks. Finally, we thank Huaicheng Li for help with integrating FEMU, and Tao Wang and Anirudh Sivaraman for help with Menshen. Jialin Li is supported by a MOE Tier 1 grant A-0008452-00-00 and a ODPRT grant A-0008089-00-00.

References

[1] 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.

[2] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. 2020. Data Direct I/O Characterization for Future I/O System Exploration. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*. 160–169. https://doi.org/10.1109/ISPASS48437.2020.00031

[3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. 2010. Data Center TCP (DCTCP). In *2010 ACM SIGCOMM Conference on Data Communication* (New Delhi, India) *(SIGCOMM)*.

[4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. 2020. Enabling Programmable Transport Protocols in High-Speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation* (Santa Clara, CA) *(NSDI)*.

[5] Andrew Baumann, Paul Barham, Pierre-Evariste Dagand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhania. 2009. The Multikernel: A New OS Architecture for Scalable Multicore Systems. In *22nd ACM Symposium on Operating Systems Principles* (Big Sky, MT) *(SOSP)*.

[6] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhania, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. 2009. Your computer is already a distributed system. Why isn't your OS?. In *12th Workshop on Hot Topics in Operating Systems* (Monte Verità, Switzerland) *(HOTOS)*.

[7] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. 1991. User-Level Interprocess Communication for Shared Memory Multiprocessors. *ACM Transactions on Computer Systems* 9, 2 (May 1991), 175–198.

[8] 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.

[9] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. 2013. Forwarding Metamorphosis: Fast Programmable Match-action Processing in Hardware for SDN. In *2013 ACM SIGCOMM Conference on Data Communication* (Hong Kong, China) *(SIGCOMM)*.

[10] Randal Everitt Bryant. 1977. *Simulation of packet communication architecture*. Master's thesis. Massachusetts Institute of Technology, Laboratory for Computer Science.

[11] K. Mani Chandy and Jayadev Misra. 1979. Distributed simulation: A case study in design and verification of distributed programs. *IEEE Transactions on software engineering* SE-5, 5 (1979), 440–452.

[12] Jianwei Chen, Murali Annavaram, and Michel Dubois. 2009. SlackSim: A Platform for Parallel Simulations of CMPs on CMPs. *SIGARCH Computer Architecture News* 37, 2 (July 2009).

[13] Sharad Chole, Andy Fingerhut, Sha Ma, Anirudh Sivaraman, Shay Vargaftik, Alon Berger, Gal Mendelson, Mohammad Alizadeh, Shang-Tse Chuang, Isaac Keslassy, Ariel Orda, and Tom Edsall. 2017. dRMT: Disaggregated Programmable Switching. In *2017 ACM SIGCOMM Conference on Data Communication* (Los Angeles, CA) *(SIGCOMM)*.

[14] CXL Consortium. 2020. Compute Express Link (CXL). https://www.computeexpresslink.org/spec-landing. Revision 2.0.

[15] DPDK Project. 2022. Data Plane Development Kit. http://www.dpdk.org/. Retrieved Feb 2, 2022.

[16] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. 2020. Corundum: An Open-Source 100-Gbps NIC. In *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines* (Fayetteville, AR) *(FCCM)*.

[17] Alex Forencich, Alex C. Snoeren, George Porter, and George Papen. 2022. Corundum GitHub Repository. https://github.com/corundum/corundum. Retrieved Feb 2, 2022.

[18] Yaosheng Fu and David Wentzlaff. 2014. PriME: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software* (Monterey, CA) *(ISPASS)*.

[19] Richard M. Fujimoto. 1999. Exploiting Temporal Uncertainty in Parallel and Distributed Simulations. In *Proceedings of the Thirteenth Workshop on Parallel and Distributed Simulation* (Atlanta, GA) *(PADS)*.

[20] Mark Handley, Costin Raiciu, Alexandru Agache, Andrei Voinescu, Andrew W. Moore, Gianni Antichi, and Marcin Wójcik. 2017. Re-Architecting Datacenter Networks and Stacks for Low Latency and High Performance. In *2017 ACM SIGCOMM Conference on Data Communication* (Los Angeles, CA) *(SIGCOMM)*.

[21] INET Authors. 2022. INET Framework. https://inet.omnetpp.org/. Retrieved Feb 2, 2022.

[22] Intel Corporation. 2020. Intel Ethernet Controller X710/ XXV710/XL710 Datasheet. https://cdrdv2.intel.com/v1/dl/getContent/332464. Revision 3.7.

[23] Intel Corporation. 2022. Intel P4 Studio. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html. Retrieved Feb 2, 2022.

[24] Intel Corporation. 2022. Intel Tofino Series Programmable Switch ASIC. https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/tofino-series.html. Retrieved Feb 2, 2022.

[25] Norman P. Jouppi, Cliff Young, Nishant Patil, David Patterson, Gaurav Agrawal, Raminder Bajwa, Sarah Bates, Suresh Bhatia, Nan Boden, Al Borchers, Rick Boyle, Pierre-luc Cantin, Clifford Chao, Chris Clark, Jeremy Coriell, Mike Daley, Matt Dau, Jeffrey Dean, Ben Gelb, Tara Vazir Ghaemmaghami, Rajendra Gottipati, William Gulland, Robert Hagmann, C. Richard Ho, Doug Hogberg, John Hu, Robert Hundt, Dan Hurt, Julian Ibarz, Aaron Jaffey, Alek Jaworski, Alexander Kaplan, Harshit Khaitan, Daniel Killebrew, Andy Koch, Naveen Kumar, Steve Lacy, James Laudon, James Law, Diemthu Le, Chris Leary, Zhuyuan Liu, Kyle Lucke, Alan Lundin, Gordon MacKean, Adriana Maggiore, Maire Mahony, Kieran Miller, Rahul Nagarajan, Ravi Narayanaswami, Ray Ni, Kathy Nix, Thomas Norrie, Mark Omernick, Narayana Penukonda, Andy Phelps, Jonathan Ross, Matt Ross, Amir Salek, Emad Samadiani, Chris Severn, Gregory Sizikov, Matthew Snelham, Jed Souter, Dan Steinberg, Andy Swing, Mercedes Tan, Gregory Thorson, Bo Tian, Horia Toma, Erick Tuttle, Vijay Vasudevan, Richard Walter, Walter Wang, Eric Wilcox, and Doe Hyun Yoon. 2017. In-Datacenter Performance Analysis of a Tensor Processing Unit. In *44th Annual International Symposium on Computer Architecture* (Toronto, ON, Canada) *(ISCA)*.

[26] Sagar Karandikar, Howard Mao, Donggyu Kim, David Biancolin, Alon Amid, Dayeol Lee, Nathan Pemberton, Emmanuel Amaro, Colin Schmidt, Aditya Chopra, Qijing Huang, Kyle Kovacs, Borivoje Nikolic, Randy Katz, Jonathan Bachrach, and Krste Asanović. 2018. FireSim: FPGA-accelerated Cycle-exact Scale-out System Simulation in the Public Cloud. In *45th Annual International Symposium on Computer Architecture* (Los Angeles, CA) *(ISCA)*.

[27] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. 2016. High Performance Packet Processing with FlexNIC. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems* (Atlanta, GA) *(ASPLOS)*.

[28] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. 2019. TAS: TCP Acceleration as an OS Service. In *14th ACM European Conference on Computer Systems* (Dresden, Germany) *(EuroSys)*.

[29] Gautam Kumar, Nandita Dukkipati, Keon Jang, Hassan M. G. Wassel, Xian Wu, Behnam Montazeri, Yaogong Wang, Kevin Springborn, Christopher Alfeld, Michael Ryan, David Wetherall, and Amin Vahdat. 2020. Swift: Delay is Simple and Effective for Congestion Control in the Datacenter. In *2020 ACM SIGCOMM Conference on Data Communication* (Virtual Event, USA) *(SIGCOMM)*.

[30] Bob Lantz, Brandon Heller, and Nick McKeown. 2010. A Network in a Laptop: Rapid Prototyping for Software-Defined Networks. In *10th ACM Workshop on Hot Topics in Networks* (Monterey, CA) *(HotNets)*.

[31] Huaicheng Li, Mingzhe Hao, Michael Hao Tong, Swaminathan Sundararaman, Matias Bjørling, and Haryadi S. Gunawi. 2018. The CASE of FEMU: Cheap, Accurate, Scalable and Extensible Flash Emulator. In *2018 USENIX Annual Technical Conference* (Boston, MA) *(ATC)*.

[32] Jialin Li, Ellis Michael, and Dan R. K. Ports. 2017. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP '17)*. Association for Computing Machinery, Shanghai, China. https://doi.org/10.1145/3132747.3132751

[33] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. 2016. Just Say NO to Paxos Overhead: Replacing Consensus with Network Ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation* (Savannah, GA) *(OSDI)*.

[34] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. 2020. PANIC: A High-Performance Programmable NIC for Multi-tenant Networks. In *14th USENIX Symposium on Operating Systems Design and Implementation* (Virtual Event) *(OSDI)*.

[35] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. 2017. IncBricks: Toward In-Network Computation with an In-Network Cache. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS)*.

[36] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. 2016. ASIC Clouds: Specializing the Datacenter. In *43rd Annual International Symposium on Computer Architecture* (Seoul, Republic of Korea) *(ISCA)*.

[37] 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.

[38] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. 2017. System simulation with gem5 and SystemC: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation* (Pythagoreio, Greece) *(SAMOS)*.

[39] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. 2010.

Graphite: A distributed parallel simulator for multicores. In *16th IEEE International Symposium on High-Performance Computer Architecture* (Bangalore, India) *(HPCA)*.

[40] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. 2015. TIMELY: RTT-based Congestion Control for the Datacenter. In *2015 ACM SIGCOMM Conference on Data Communication* (London, United Kingdom) *(SIGCOMM)*.

[41] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. 2018. Revisiting Network Support for RDMA. In *2018 ACM SIGCOMM Conference on Data Communication* (Budapest, Hungary) *(SIGCOMM)*.

[42] Alian Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. 2017. dist-gem5: Distributed simulation of computer clusters. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software* (Santa Rosa, CA) *(ISPASS)*.

[43] ns-2 Authors. 2022. The Network Simulator - ns-2. https://www.isi.edu/nsnam/ns/. Retrieved Feb 2, 2022.

[44] nsnam. 2022. ns-3 | a discrete-event network simulator for internet systems. https://www.nsnam.org/. Retrieved Feb 2, 2022.

[45] Joshua Pelkey and George Riley. 2011. Distributed Simulation with MPI in Ns-3. In *Proceedings of the 4th International ICST Conference on Simulation Tools and Techniques* (Barcelona, Spain) *(SIMUTools '11)*. ICST (Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering), Brussels, BEL, 410–414.

[46] QEMU Authors. 2022. QEMU – the FAST! processor emulator. https://www.qemu.org/. Retrieved Feb 2, 2022.

[47] Steven K Reinhardt, Mark D Hill, James R Larus, Alvin R Lebeck, James C Lewis, and David A Wood. 1993. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In *1993 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science* (Santa Clara, CA) *(SIGMETRICS)*.

[48] 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.

[49] Daniel Sanchez and Christos Kozyrakis. 2013. ZSim: Fast and Accurate Microarchitectural Simulation of Thousand-Core Systems. In *40th Annual International Symposium on Computer Architecture* (Tel-Aviv, Israel) *(ISCA)*.

[50] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. 2017. Evaluating the Power of Flexible Packet Processing for Network Resource Allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation* (Boston, MA) *(NSDI)*.

[51] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. 2018. Approximating Fair Queueing on Reconfigurable Switches. In *15th USENIX Symposium on Networked Systems Design and Implementation* (Renton, WA) *(NSDI)*.

[52] Siemens. 2022. ModelSim HDL Simulator. https://eda.sw.siemens.com/en-US/ic/modelsim/. Retrieved Feb 2, 2022.

[53] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. 2016. Packet Transactions: High-Level Programming for Line-Rate Switches. In *2016 ACM SIGCOMM Conference on Data Communication* (Florianopolis, Brazil) *(SIGCOMM)*.

[54] Wilson Snyder. 2022. Verilator – the fastest Verilog HDL simulator. https://www.veripool.org/wiki/verilator. Retrieved Feb 2, 2022.

[55] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. 2020. The NeBuLa RPC-Optimized Architecture. In *47th Annual International Symposium on Computer Architecture* (Worldwide) *(ISCA)*.

[56] Hajime Tazaki, Frédéric Uarbani, Emilio Mancini, Mathieu Lacage, Daniel Camara, Thierry Turletti, and Walid Dabbous. 2013. Direct Code Execution: Revisiting Library OS Architecture for Reproducible Network Experiments. In *ACM Conference on Emerging Networking Experiments and Technologies* (Santa Barbara, CA) *(CoNEXT)*.

[57] András Varga and Rudolf Hornig. 2008. An Overview of the OMNeT++ Simulation Environment. In *1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops* (Marseille, France) *(Simutools)*.

[58] Tao Wang, Xiangrui Yang, Gianni Antichi, Anirudh Sivaraman, and Aurojit Panda. 2021. Isolation mechanisms for packet-processing pipelines. *CoRR* abs/2101.12691 (2021). arXiv:2101.12691 https://arxiv.org/abs/2101.12691

[59] Xilinx. 2022. Vivado 2021.2 Logic Simulation. https://www.xilinx.com/support/documentation-navigation/design-hubs/dh0010-vivado-simulation-hub.html. Retrieved Feb 2, 2022.

Appendices are supporting material that has not been peer-reviewed.

A Appendix

A.1 Modular Simulation Orchestration

Finally, an operational challenge arises for running simulations with SimBricks. Because we design SimBricks without any centralized control, a simulation consists entirely of interconnected component simulators. Thus, to run a complete end-to-end simulation, a user has to start each individual component simulator, while providing unique paths for the Unix sockets and shared memory regions for each channel. While this is manageable with very small simulations, the complexity rapidly grows with simulation size, along with the additional challenges of cleanup, collecting simulation logs, and monitoring for crashes. An additional challenge, especially when running multiple simulations in parallel, is that performance drastically degrades when overcommitting cores or memory. SimBricks addresses both challenges with an orchestration framework for assembling, running, and, if necessary, scheduling simulations.

```
from simbricks import *
for rate in [10, 100, 200, 500, 1000]:
  e = Experiment('udp-' + str(rate))
  net = SwitchBM(e)

  s = Gem5Host(e, 'server')
  s.nic = I40eNIC(e)
  s.node_config = I40eLinuxNode()
  s.node_config.ip = '10.0.0.1'
  s.node_config.app = IperfUDPServer()

  c = Gem5Host(e, 'client')
  c.nic = I40eNIC(e)
  c.node_config = I40eLinuxNode()
  c.node_config.ip = '10.0.0.2'
  c.node_config.app = IperfUDPClient()
  c.node_config.app.server = '10.0.0.1'
  c.node_config.app.rate = rate

  experiments.append(e)
```

Figure 12: An example of a simulation configuration in the SimBricks orchestration framework.

Similar to other simulators with modular configuration we also implement our orchestration in a scripting language. The SimBricks orchestration framework is designed as a collection of python modules, and simulation experiments can be assembled by relying on arbitrary python features. In addition to the previously mentioned tasks, we also integrate functionality to automatically generate customized disk images for host simulators, e.g. with different IP address configurations or to run applications with separate parameters in individual hosts. In Fig. 12 we show an example script.

A.2 Inter-Simulator Message Transport

Fig. 13 shows pseudocode for the SimBricks queue implementation. To enable zero-copy implementation in simulators producer and consumer each have separate functions for getting access to an available queue slot, PollMsg for the consumer and AllocMsg for the producer, and then releasing in when processing is complete, ReleaseMsg for the consumer and EnqueueMsg for the producer. The consumer uses its local head pointer to determine the slot the next message is or will be in and then checks the type and ownership byte, re-trying if the slot is marked by as owned by the producer. After the consumer completes processing a message it marks the message as owned by the consumer. Symmetrically, the producer uses its local tail pointer to determine the slot for the next message, if necessary waits until the slot is marked as producer-owned, and resets the ownership bit to consumer after it places the message in the slot. Compiler memory barriers are necessary to prevent the compiler from reordering memory accesses across accesses to the ownership bit, but with the strong X86 memory model no CPU memory barriers are necessary.

```
rxQueue, rxLen ← MapQueue(rx)
rxHead ← 0
txQueue, txLen ← MapQueue(tx)
txTail ← 0

procedure PollMsg
    msg ← &rxQueue[rxHead]
    while msg->owner ≠ CONSUMER do
        Spin()
    ReadMemoryBarrier()
    rxHead ← (rxHead + 1) % rxLen
    return msg
procedure ReleaseMsg(msg)
    msg->owner ← PRODUCER
procedure AllocMsg
    msg ← &txQueue[txTail]
    while msg->owner ≠ PRODUCER do
        Spin()
    txTail ← (txTail + 1) % txLen
    return msg
procedure EnqueueMsg(msg)
    WriteMemoryBarrier()
    msg->owner ← CONSUMER
```

Figure 13: SimBricks multi-core shared memory message passing queue. ReadMemoryBarrier and WriteMemoryBarrier are compiler barriers to prevent re-ordering during optimization.

A.2.1 Coherence Behavior To understand the performance properties, consider three key cases, the queue is empty, the queue is full, and the queue is neither empty nor full. When the queue is empty, the consumer will spin on the last cache line, which will be in the local L1 after the first access, and only incurs an additional when the producer updates that cache line. When the queue is full, the producer similarly waits for the next slot to free up with the same coherence behavior. Finally, when neither is the case, the consumer immediately finds a message when polling and incurs a necessary miss that will fetch the message. Further, the CPU

	SimBricks Component	Lines
SimBricks core	Message transport library	1411
	NIC behavioral model library	715
	Distributed simulation proxy	2080
	Runtime orchestration	2102
Host simulators	gem5 integration	1265
	QEMU integration	676
NIC simulators	Corundum Verilator	1315
	Intel i40e model	2900
	Corundum model	911
	gem5 e1000 model	2952
Network simulators	ns-3 integration	158
	OMNeT++ integration	208
	Tofino simulator integration	330
	Ethernet switch model	399
	Menshen RMT Verilator	391
	Packet generator	415
Dev sims.	FEMU SSD integration	1005

Table 2: Lines of code for the various components in SimBricks, excluding blank lines and comments. For integrated simulators we only count adapter code.

hardware prefetcher will likely already fetch the next message as they are laid out sequentially in memory, thereby avoiding a demand miss (but of course incurring the same coherence traffic). The producer does have to read the ownership flag incurring a miss, but also immediately finds the empty slot, and the same prefetcher behavior applies.

A.3 SimBricks Implementation Effort

Tab. 2 shows a per-component breakdown of the implementation effort for SimBricks, listing the number of lines of code.

A.4 Performance for SimBricks Configurations

Tab. 3 contains a cross-product of different simulators in SimBricks for host, NIC, and the network. This is an extended version of Tab. 1 with the same experimental setup. Note that with recent versions of QEMU we have found QEMU + timing (QT) no longer to be fully deterministic and have instead observed minor variations in simulation results.

Simulators					Sim.	
Host	NIC	Net	T'put	Latency	Time	Det.
QK	IB	SW	4.37 G	71 μs	00:00:32	
QK	IB	NS	409 M	141 μs	00:00:32	
QK	IB	TO	1.92 M	6.6 ms	00:00:33	
QK	CB	SW	1.84 G	211 μs	00:00:29	
QK	CB	NS	429 M	294 μs	00:00:30	
QK	CB	TO	2.18 M	6.7 ms	00:00:33	
QK	CV	SW	81 M	3.4 ms	00:00:31	
QK	CV	NS	82 M	3.4 ms	00:00:32	
QK	CV	TO	2.31 M	23 ms	00:00:33	
QT	IB	SW	8.85 G	17 μs	01:05:03	(✓)
QT	IB	NS	8.88 G	17 μs	01:06:43	(✓)
QT	CB	SW	3.74 G	28 μs	01:00:24	(✓)
QT	CB	NS	3.74 G	28 μs	00:59:41	(✓)
QT	CV	SW	6.55 G	32 μs	04:13:10	(✓)
QT	CV	NS	6.39 G	32 μs	04:13:13	(✓)
G5	IB	SW	8.84 G	20 μs	12:51:41	✓
G5	IB	NS	8.92 G	20 μs	12:49:46	✓
G5	CB	SW	3.05 G	33 μs	09:20:48	✓
G5	CB	NS	3.06 G	33 μs	09:26:13	✓
G5	CV	SW	6.70 G	37 μs	10:23:26	✓
G5	CV	NS	6.43 G	37 μs	10:21:28	✓

Table 3: Performance for combinations of some of our component simulators. Checkmarks mark deterministic combinations. Host: QK is QEMU with KVM (functional simulation), QT is QEMU with timing, and G5 is gem5. NIC: IB is the Intel behavioral model, CB the Corundum behavioral model, and CV the Corundum verilator model. Network: SW is the switch behavioral model, NS is ns-3, TO is the Tofino model.

B Artifact Appendix

Abstract

The SimBricks artifact comprises two components, the source code of the main simulator, and paper-specific parts (artifact scripts, documentation, and data) to replicate the results in this paper.

Scope

Users interested in using SimBricks in their work should refer to the former, as this will continue to evolve over time, while the latter remains stable (modulo bug fixes) to ensure reproducible results.

The artifact scripts can run all major and minor experiments in the paper, except for the physical testbed baseline for the dctcp experiments. For deterministic simulations, results should be exactly reproducible. Other measurements, especially simulation times, will vary based on the hardware, but should be approximately reproducible on similar hardware to what we describe.

Contents

The artifact contains everything required to reproduce the results in the paper: source code, instructions for building and running SimBricks, scripts for running experiments, and plotting scripts for the graphs in the paper. We also include most of the execution logs we generated for the experiments in this paper.

Hosting

Both the main SimBricks repo and the artifact package are hosted on GitHub:

- **Main SimBricks source:** https://github.com/simbricks/simbricks
- **Artifact package:** https://github.com/simbricks/sigcomm22-artifact

For both we have tagged the version submitted for evaluation with `sigcomm22-ae-submission`, and a stable version potentially receiving bug-fixes will remain in the `sigcomm22-ae` branch. The `main` branch will evolve and might contain breaking changes.

We have also built docker specifically for the artifact that we link to in the artifact README file.

Requirements

The precise hardware requirements for each experiment vary significantly and are detailed in the artifact repository. All non-distributed experiments only require a single machine, but require sufficient processor cores (varies per experiment up to 44). The largest experiments also require around 192 GB of RAM.

We have tested SimBricks on Linux. The specific software dependencies are provided by the documentation in the artifact repo.