

Enabling End-to-End Simulation for Host Networking Evaluation using SimBricks

Hejing Li

Max Planck Institute for Software Systems

Jialin Li

National University of Singapore

Keon Jang

Rubrik

Antoine Kaufmann

Max Planck Institute for Software Systems

Abstract

Full system "end-to-end" measurements in physical testbeds are the gold standard for evaluation of network systems but are fraught with challenges. Adequate testbeds are often not available, as projects target next generation devices, propose new hardware, or require larger scale. Further, evaluations in testbeds limit what we can observe without affecting system behavior, are frequently hard to reproduce, and are only available to groups with sufficient funding. Yet, we lack an accepted alternative, leaving us with ad-hoc non-end-to-end evaluations that do not form a solid basis for future work.

We argue that full system simulations enable comparable end-to-end evaluation and are the next best alternative when a physical testbed is not available. To this end, we present SimBricks, a modular full system simulation framework for network systems. SimBricks combines multiple existing simulators for individual components, including processor and memory, NIC, and network, into full virtual testbeds running unmodified software. The architecture combines well-defined component interfaces for extensibility with new simulators, efficient communication channels for local and distributed simulators, and a novel efficient synchronization protocol for accurate timing across different simulators. We demonstrate that SimBricks simulations reproduce key findings from prior work in congestion control, NIC architecture, and in-network computing, and show scalability to 1000 simulated hosts.

1 Introduction

The systems and networking community expects research ideas to be implemented and evaluated as part of a complete system "end-to-end" in realistic conditions, if at all possible. End-to-end evaluation is important as many factors affect system behavior in ways that are hard to reason about a priori.

Yet evaluation in full physical testbeds is frequently difficult or outright infeasible for academic research projects, because adequate hardware testbeds are not available. Work might require cutting edge commercial hardware that is

not available yet at the time of publication [26, 27, 29, 44], other work designs hardware extensions to existing proprietary hardware [45], finally there is much work proposing entirely new hardware architectures to be implemented as ASICs [12, 13, 21, 23, 28, 30, 46, 47]. Evaluation on a real hardware implementation of the system is beyond academic researchers for these projects. This trend is getting worse and worse in the post-Moore era, as we move towards increasingly specialized hardware, including SmartNICs, programmable switches, and other hardware accelerators. Finally, especially in domains such as network protocols and congestion control, proposals need to be evaluated at large scale in networks with hundreds or thousands of hosts. In all of these cases, a full end-to-end evaluation in a physical testbed is infeasible.

When a full evaluation in a physical testbed is not possible, simulation has long offered an alternative means of system evaluation. In networking, we use ns-2 [39], ns-3 [40], and OMNeT++ [48] to evaluate protocols and algorithms; computer architects rely on cycle accurate system simulators such as gem5 [9], while hardware designers employ RTL simulators such as Modelsim [36] or Verilator [49] for testing hardware components. While network systems do benefit from these simulators [4, 24, 35], they do not enable end-to-end evaluation, as no existing simulator simulates all required components in a testbed: host, NIC devices, full network.

In this paper, we demonstrate end-to-end evaluation in simulation can be achieved by combining and connecting multiple different existing simulators to cover the necessary functionality. Instead of building a new simulator, throwing away decades of work on existing simulators, we integrate existing and new simulators, for hosts, NICs, and networks, into full system simulations of complete network testbeds capable of running unmodified operating systems, drivers, and applications. Existing simulators, however, are not designed for connecting to other simulators. To achieve modular end-to-end simulations we need to overcome three technical challenges: 1) no interfaces to connect simulators together, 2) efficient and scalable 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 that is scalable, accurate, and enables reproducible evaluation. SimBricks defines interfaces for simulators based on natural component boundaries in physical systems. We combine these with a novel protocol that leverages latency at component boundaries for efficient and accurate synchronization of simulator clocks. Individual component simulators run in parallel as separate processes, and communicate via message passing through optimized shared memory queues.

Currently, SimBricks integrates QEMU [41] and gem5 [9] as host simulators, Verilator [49] as an RTL hardware simulator for NICs with the open source Corundum FPGA NIC [16] as a NIC simulation, and ns-3 [40] as well as the Intel Tofino simulator [18] for network simulation. We have also implemented fast behavioral simulators for Corundum and the Intel X710 40G NIC [19], as well as a basic network switch. In combination, these simulators enable a broad range of different testbed configurations for end-to-end evaluation. In our evaluation we demonstrate that SimBricks enables flexible end-to-end evaluation of network systems at small and large scales. We reproduce key results from congestion control [3], in-network compute [27], and FPGA NIC design [16] in SimBricks. SimBricks obtains more realistic results compared to ns-3 in isolation (§2.2). SimBricks also scales to 1000 hosts and NICs with only a 14% increase in simulation time from 40 hosts (§6.5). Finally, SimBricks also provides deeper visibility and increased control off low-level system behavior compared to physical testbeds (§6.8).

We make the following technical contributions:

- *Modular architecture for network system simulation* (§4) with interfaces for host, NIC, and network component simulators.
- *Synchronization protocol for parallel and distributed simulations* (§4.2) using message passing to efficiently ensure correct simulation, even at scale.
- SimBricks, a *prototype implementation* of our architecture (§5) and with integration of multiple existing and new component simulators. We plan for a full open source release prior to publication.¹

2 Background and Motivation

We start by providing background on simulations for computer systems components, before discussing the shortcomings of existing simulators for computer systems research.

¹Along with simulation and plotting scripts to reproduce our results.

2.1 Simulations — Virtual Testbeds

Simulators employ various techniques, including discrete event simulation, binary translation, and hardware virtualization, to simulate networks, computer architectures, and hardware components. Network simulators, such as ns-2 [39], ns-3 [40], and OMNeT++ [48], rely on discrete event simulation using timestamped events to model packets traversing network topologies assembled from models of individual components. Computer architecture simulators, such as gem5 [9], QEMU [41], and Simics [31], simulate full computer systems capable of running unmodified guest software, including operating systems, with different and sometimes configurable degrees of detail. Full system simulators also include I/O devices, but typically only implemented with the bare minimum of required features. Detailed RTL simulations, such as xsim [51] and Verilator [49], are employed to test and debug hardware designs using accurate timing against testbenches. In all three cases – network, system, and hardware simulators – *individual components are simulated in isolation*.

Advantages. The primary motivation for simulating a system is typically that a physical implementation is not feasible. Simulations are also *portable* and can be easily shared as they completely decouple the simulated system from the host system. A large class of simulators, e.g., discrete event simulators, are deterministic (with explicit seeds for randomness), enabling truly *reproducible evaluation*. Simulators also provide *flexibility*; they are implemented as software that can be modified, and they usually offer parameters representing a broad range of system configurations. Finally, simulations offer superior *visibility* compared to physical hardware, as they can be easily instrumented to log minute details about the system, without affecting system behavior.

Problems. Simulations also have issues that limit their applicability. *Long simulation times* are common – detailed architectural simulators typically only simulate hundreds or thousands of system cycles a second [22, 47], 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. Developing new simulators and maintaining or extending existing ones require significant *development effort*. Finally, results obtained from simulations are only as good as the simulator, and may not be representative unless *validated* against a physical testbed.

2.2 Simulations for Systems Research

For systems research there are additional challenges that often preclude using simulation during prototyping and evaluation. First off, network and distributed systems frequently require evaluation on tens or hundreds of hosts to demonstrate *scalability*. But for most simulators, already long simulation time

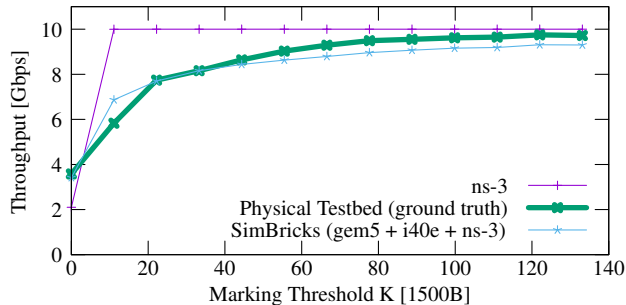


Fig. 1: Measured throughput for two dctp flows in ns-3, a physical testbed, and an end-to-end simulation with SimBricks.

increases super-linearly with the size of the simulated system. Next, systems projects typically have substantial software, and sometimes hardware, implementation components. Unfortunately, prototyping productivity decreases significantly with long edit-compile-test cycles. While there exist fast simulators that omit various levels of details, prototyping in these simulators defeats the purpose if the implemented system can then not be evaluated properly because of insufficient accuracy. Finally and most importantly, *no existing simulators covers all required components for networked systems with sufficient features and details*, precluding full *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, systems researchers are left with non-end-to-end "piecemeal" evaluation, where different components are evaluated in isolation, or to build ad hoc simulators which have low accuracy.

Motivational example. To demonstrate the pitfalls of piecemeal evaluation, we compare the behavior of the dctp [3] congestion control algorithm in the ns-3 network simulator to a physical testbed. As network speed increases and bottlenecks move to end-hosts, congestion control behavior is influenced by small variations in timing in the host hardware and software [3, 25, 34]. However, ns-3 only models network- and protocol-level behavior, and as a result, does not simulate the host side accurately. Concretely, we set up two clients and two servers sharing a single 10G bottleneck link with a 4000B MTU, and with one large TCP flow generated by iperf for each client-server pair. Fig. 1 shows the measured throughput for varying dctp marking thresholds K . The dctp marking threshold K balances queuing latency and throughput; a lower threshold reduces queue length but risks under-utilizing links. As expected [3], the ns-3 simulation underestimates the necessary threshold to achieve line rate, as it does not model host processing variations, such as processing delay caused by OS interrupt scheduling. Only an end-to-end

evaluation of the full system captures such intricacies.

3 Modular Simulation

We argue that *rigorous end-to-end evaluation can be efficiently achieved through simulations assembled from multiple existing simulators for individual components that are interconnected and synchronized*. To demonstrate this, we present SimBricks, a new modular, end-to-end simulation framework that aims to provide accurate simulation and evaluation of network systems.

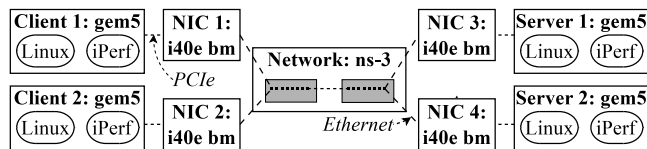


Fig. 2: SimBricks configuration for the dctp experiment in Fig. 1, combining gem5, ns-3, and an Intel NIC simulator.

End-to-end simulation is better. Fig. 2 shows the SimBricks end-to-end simulation that produces the result shown in Fig. 1. We combine four instances of gem5 with four instances of Intel i40e NIC simulator we developed, and 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 configuration approximates the behavior of the physical testbed closely, and capture the same insight. We conclude that end-to-end evaluation with SimBricks improves accuracy for congestion control evaluation over ns-3.

3.1 Design Goals

SimBricks has the following design goals to enable prototyping and evaluation of systems research projects using simulation:

- **End-to-end:** simulate complete network systems, including hosts, NICs, network hardware, and the full software stack including OS and applications.
- **Scalable:** simulate large network systems consisting of hundreds of hosts and devices.
- **Fast:** keep simulation times low to allow for simulation of real systems.
- **Modular:** enable flexible configuration that can be modified and extended with new simulators without modification to the rest of the system.
- **Accurate:** Preserve the accuracy of the constituent simulators when combining them. Ensure that validation from individual simulators carries over to the combined simulation.

- **Deterministic:** keep end-to-end results reproducible when individual simulators are deterministic.

3.2 Technical challenges

Achieving these goals, however, requires us to address the following set of technical challenges:

Simulation interconnection interfaces. Only by connecting multiple simulators for different components together can we cover the necessary functionality for end-to-end simulation. Unfortunately, existing simulators provide no suitable interfaces for connecting to other external simulators. Moreover, enabling modular "plug-and-play" configurations, where individual components can be swapped out, requires well-defined interfaces for different component types.

Scalable synchronization and communication. Individual component simulators maintain their own separate virtual simulation clocks. To obtain meaningful performance measurements, we need a mechanism to synchronize these clocks. However, synchronizing clocks of multiple simulators comes at a cost, especially with increasing system scale. For example, we measure a $3.7\times$ increase in runtime for the dist-gem5 [37] simulator when increasing simulation size from 2 to 16 gem5 hosts, due to synchronization overhead (§6.3). Prior work shows that synchronization overhead can be reduced by sacrificing accuracy and determinism through lax synchronization. However, results of partially synchronized simulations are hard to reason about and less reproducible as they depend on specific runtime factors. Modular combination of different simulators further compounds this and, we argue, requires validation of each combination of simulators.

Incompatible simulation models. Interconnecting different simulators is a challenge as different simulators employ mutually incompatible simulation models. For example, QEMU has a blocking device model where calls in device code are expected to block till completed, while ns-3 asynchronously schedules events to model networks, and Verilator simulates hardware cycle by cycle. Attempting to connect these simulators together requires us to address these impedance mismatches.

3.3 Design principles

We address these challenges through four principles that underpin the design of SimBricks.

Fix natural component simulator interfaces. To enable modular composition of simulators, SimBricks defines an interface for each *component type* (§4.1). We base these interfaces on the component boundaries in real systems: PCI express (PCIe) connects today's NICs to servers, while data cen-

ter NICs typically connect to Ethernet networks. We choose these interfaces as a starting point, but our approach generalizes to other interconnects and network technologies. These component interfaces form narrow waists decoupling innovation on both sides: To integrate a simulator into SimBricks, developers only need to add an adapter that implements the component interface, otherwise the simulator remains unmodified.

Accurate and efficient synchronization. In SimBricks, we want to ensure accurate results and avoid requiring validation of each combination of simulators, and demonstrate that accurate time synchronization does not have to come at the cost of performance. To this end, we leverage three insights in combination to design a novel synchronization protocol (§4.2): 1) *Global synchronization is not necessary* as using natural interfaces limits which simulators actually communicate. As long as events at these pairwise interfaces arrive on time the simulation behavior is correct. 2) *Latency at component interfaces provides slack* and helps tolerate communication overheads between simulators. An event sent at time T only arrives at $T + \Delta$, as our natural component interfaces have an inherent latency in physical systems that we need to model. 3) *By including synchronization in-line with data transfers*, synchronization overheads can be reduced and sometimes completely avoided. The SimBricks synchronization protocol only increases simulation time of two gem5 instances by 2% compared to unsynchronized operation. Even when scaling up to a distributed simulation of 1000 hosts, simulation time only increases by $3\times$ compared to a simulation of 40 hosts.

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 (§4.1) across our defined interfaces. This drastically simplifies integration of 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 naturally issue requests and process responses at the scheduled times, while blocking simulations can block till the response message arrives – for peer simulators this is fully transparent.

Parallel execution with shared memory queues. We run simulators in parallel on different cores and connect them through optimized shared-memory queues (§4.3). As simulators run on separate cores and only communicate explicitly when necessary, this avoids unnecessary cache-coherence traffic and hidden scalability bottlenecks. These mechanisms allow us to (i) scale up to large network simulations: Instead of simulating the complete network in one simulation

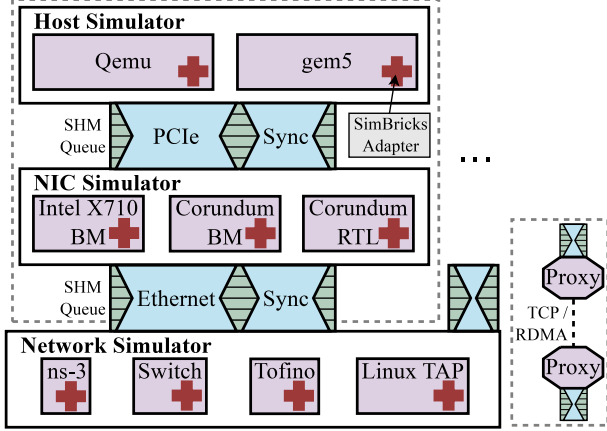


Fig. 3: SimBricks architecture. *Double hour glass* with narrow waists between hosts and NICs, and NICs and networks.

instance, we simulate different parts of the network in separate simulators running in parallel (§4.4), and (ii) scale out using 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 (§4.5).

4 Design

Following the design principles in §3, we have built a modular, end-to-end simulation framework SimBricks, with its high-level system architecture shown in Fig. 3. In this section, we detail the design of SimBricks’s major components, including component simulator interfaces, a novel synchronization protocol, fast message transport, and techniques to scale up and out to larger simulations.

4.1 Component Simulator Interfaces

SimBricks’s modularity is based on well-defined interfaces between component simulators: Host simulators connect to NIC simulators through an interface based on PCIe; NIC simulators interface with network simulators through an Ethernet interface. This results in a double hourglass architecture (Fig. 3) with narrow waists at component boundaries. Both interfaces are inherently asynchronous in physical systems, and SimBricks replicates this. We also model the physical propagation delay at each interface as a per-interface latency Δ_i .

4.1.1 PCIe: Host-NIC interface.

PCIe itself is architected as a layered protocol, ranging from a low-level physical layer to the transactional layer responsible for data operations. We define SimBricks’s host-NIC interface (Fig. 4) solely based on the PCIe *transactional layer*,

PCIe: NIC → Host	
Message Type	Message Fields
INIT_NIC	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 (and Payload data)
MMIO_COMPL	Request ID (and Payload data)
INTERRUPT	Interrupt type, MSI/MSI-X: vector #, Legacy: level

PCIe: Host → NIC	
Message Type	Message Fields
DMA_COMPL	Request ID (and Payload data)
MMIO_READ, MMIO_WRITE	Request ID, BAR # and offset, Length (and Payload data)
INT_STATUS	Interrupts enabled: legacy, MSI, MSI-X

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

Fig. 4: SimBricks defines a PCI interface between host and NIC simulators, and an Ethernet interface between NIC and network simulators.

and abstract physical attributes of the PCIe link away with simple parameters – link bandwidth and latency. Low-level complexities such as encodings and signaling are unnecessary for most system-level simulations and would introduce excessive complexity. Should future use-cases require low-level modelling of PCIe, a detailed PCIe simulator could be integrated modularly akin to how we scale up network simulations (§4.4).

Discovery and Initialization. A key feature of PCIe is that hosts identify connected devices and the specific features they support without relying on administrator’s input. To this end, our interface defines an `INIT_NIC` message for initializing and registering NIC simulators with a host simulator. The NIC simulator adapter includes device information in the message, including the PCI vendor, device identifiers, base address registers (BARs), and the number and addresses of MSI(-X) interrupt vectors. 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 de-

finishes `DMA_READ/WRITE` messages for DMA transfers initiated by NIC simulators, and `MMIO_READ/WRITE` for MMIO accesses initiated by host simulators. All data transfer operations are completed *asynchronously*. Once a request is finished, the NIC simulator issues a corresponding `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 asynchronously and out-of-order. To match completion notifications with outstanding requests, requests carry an opaque request identifier that the target simulator includes in the corresponding response.

Interrupts. The final mechanism covered by the host-NIC interface is interrupts. Our interface supports all PCI interrupt signaling methods, including legacy interrupt pins (INTX), message signaled interrupts (MSI), and MSI-X. Physical PCIe devices implement MSI (including configuration, masking, and generating signalling operations) completely on the device. To reduce repeated implementation effort in NIC simulators and integration challenges in host simulators, we instead opt to keep this functionality inside the host simulator. NIC simulators can issue `INTERRUPT` messages to either trigger an interrupt vector for MSI(-X) or (de-)assert interrupt pins for INTX. To incorporate NICs that require knowledge about which interrupt mechanisms the OS has enabled, our interface also provides a `INT_STATUS` message issued by the host simulator on interrupt configuration changes.

4.1.2 Ethernet: NIC-network interface.

In SimBricks’s NIC-network interface, we similarly abstract away low-level details of the Ethernet standard, and only expose Ethernet frames – in the form of `PACKET` messages – to the 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 simulator 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 do not explicitly model data corruption on the wire. We leave an extension of the interface with Ethernet hardware flow control as future work.

4.2 Simulator Synchronization Protocol

To ensure accurate simulation result when connecting parallel simulation components that run at different speeds, we design a novel synchronization protocol that minimizes synchronization overhead even when scaling to large simulations.

```

procedure INIT
  for if in interfaces do
    SYNCTIMER(if)
    msg ← POLLMSG(if)
    RESCHEDULE(msg.timestamp, RXTIMER, msg, if)
procedure SYNCTIMER(if)
  msg ← ALLOCMMSG(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 + Δif
  ENQUEUEMSG(msg, if)
  RESCHEDULE(T + δif, SYNCTIMER)

```

Fig. 5: SimBricks synchronization protocol pseudo-code 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.

4.2.1 Prior synchronization mechanisms do not scale

A naive approach to synchronizing components is to use a global barrier at each time step, keeping them completely 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. Within each epoch, no synchronization is required since all cross-component events will only be delivered after the end of the current epoch. The synchronization protocol, therefore, only needs to perform global barriers at epoch boundaries [1, 37, 42]. Unfortunately, epoch-based synchronization still relies on non-scalable global barriers across all simulators, and the barrier frequency is constrained by the lowest link latency in the whole simulation, resulting in substantial synchronization overhead.

4.2.2 Scalable synchronization in SimBricks

We avoid global synchronization while *guaranteeing simulation accuracy* by leveraging key properties specific to the SimBricks architecture. Fig. 5 shows pseudocode for the SimBricks synchronization protocol described below.

Enforcing correct message processing times is sufficient. In SimBricks, *all communication between simulators are explicitly* done through message passing. Thus, the only requirement for accurate simulation is that *messages are processed at*

the correct time. Additional synchronization would not affect simulation behavior, as simulators cannot observe or influence each other through other channels. To enforce this guarantee, senders tag each message with its arrival time indicating when the message should be processed at the receiver.

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

Per-channel message timestamps are monotonic. Our message queues deliver messages in the order of transmission. As each SimBricks connection between two simulators is modeled with a fixed (non-zero) per-link propagation latency Δ_i , a message sent at time T over interface i arrives at $T + \Delta_i$. By assuming that each simulator’s clock advances monotonically, message timestamps on each channel are therefore also monotonic.

Message timestamps ensure accuracy. A corollary of the monotonicity guarantee is that, a message with timestamp t is an implicit promise that no more messages with timestamps $< t$ will arrive on that channel. Therefore, if a simulator has received messages with timestamps larger than T from *all* of its peers, it can safely advance its clock up to T .

Ensuring liveness with sync messages. These conditions ensure accuracy of simulations, but do not guarantee liveness. Simulations can only make progress as long as every channel carries at least one message in each direction in every Δ_i time interval. To ensure progress, we introduce SYNC messages that simulators send if they have not sent a message for $\delta_i \leq \Delta_i$ time units. SYNC messages allow peers to safely advance their clocks in the absence of data messages. In our simulations we set $\delta_i = \Delta_i$; lower values of δ_i are legal, but we have not found configurations where the benefit of more frequent clock advances outweighed the cost of sending and processing the additional SYNC messages. For other simulation configurations this trade-off might be different.

Link latency provides synchronization slack. Non-zero link latencies further reduce synchronization overhead, as even peer simulators do not 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 $T + \Delta_i$. While synchronized simulations are fundamentally only as fast as the slowest component simulator, this slack improves efficiency by absorbing small transient

variation in simulation speed, without immediately blocking all simulators.

4.3 Inter-Simulator Message Transport

As SimBricks runs component simulators as separate processes communicating through message passing, inter-process communication plays a critical role on the overall efficiency. We use optimized shared memory queues with polling to implement efficient message transport for inter-simulator communication. For parallel processes running on separate cores, shared memory queues enable low-latency communication with minimal overhead [6, 8]. Between any pair of *communicating* simulators, SimBricks establishes a bidirectional message channel consisting of a pair of unidirectional queues in opposite directions. For initialization, SimBricks uses a Unix socket for each channel to provide a named endpoint for the peers to connect to, and to communicate queue parameters and shared memory file descriptor.

SimBricks message queues are single-producer single-consumer concurrent circular 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 remaining bits for the message type. Each message queue has a separate head and a tail pointer. As queues are single-producer and single-consumer, we minimize false sharing by storing the tail pointer locally at the producer, while consumers have a local exclusive head pointer. We include pseudo-code for the queue implementation in the appendix (§A.2) for reference.

The SimBricks message transport design avoids coherence overhead unless it is fundamentally necessary. The head and tail pointers are local to consumer and producer respectively, thus only accesses to shared message slots result in coherence traffic. Moreover, as long as a consumer does not poll in between its paired producer writing a message to the corresponding slot and setting the ownership bit, all coherence traffic carries necessary data from the producer to the consumer [6] (more detail in §A.2).

4.4 Scaling Up by Decomposing Networks

SimBricks scales to larger simulations by adding more component simulators. For Host and NIC simulators there is no direct scalability limit, beyond the number of available cores. A single network simulator that all hosts connect to can become a bottleneck as it has to synchronize with each NIC simulator. We leverage the SimBricks architecture to improve scalability, by decomposing the network simulation into multiple processes that connect and synchronize via SimBricks network interfaces. For example, in our evaluation we demonstrate large scale simulations that simulate each switch as a separate process (§6.5).

4.5 Scaling Out with Proxies

Running simulators in parallel on dedicated cores while communicating via shared memory queues, maximizes parallelism and minimizes communication overheads. But this limits simulation size to the number of available cores. For larger simulations SimBricks needs to scale out across multiple hosts. The combination of message passing and modular interfaces enables a natural means of scaling out simulations to multiple hosts: partition simulators to hosts and replace shared between simulators on different hosts with network communication.

However, directly implementing network communication in individual component simulators has two drawbacks. First, it increases complexity for integrating component simulators, as each individual simulator adapter needs to implement an additional message transport mechanism. Second, and more importantly, it increases communication overhead in component simulators, leaving fewer processor cycles for the many already slow simulators, further increasing simulation time.

To avoid these drawbacks SimBricks instead implements network communication in proxies. SimBricks proxies connect to local component simulators through the existing shared memory queues and forward messages over the network to the peer proxy which operates symmetrically. This approach 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 Implementation

SimBricks is implemented in 8960 lines of C/C++, 1237 lines of Python, and 1146 lines for the gem5 adapter, 724 for QEMU, and 1303 for ns-3. (more detail in §A.3).

5.1 Core SimBricks Components

Libraries. To reduce programming effort when integrating individual simulators, we develop a common message transport library that implements the SimBricks messaging interfaces. The library additionally implements helper functions for the synchronization protocol; specifically, functions to construct and send synchronization messages to the connected peers. We also implement a helper library with common components for behavioral NIC simulation models (`nicbm`) that we use in both our behavioral model implementations.

Proxies. The SimBricks design scales out simulations through proxies that translate between shared memory queues and the network on both sides. We have implemented two proxies, one uses standard sockets for network communication and the other one RDMA. Both proxies implement batching and try to forward multiple messages at once if multiple are in the queue. The RDMA proxy minimizes latency

and overhead by directly writing messages to remote queues with RDMA writes.

Orchestration. Configuring and running SimBricks simulations is a challenge due to the many interconnected components involved. We streamline this with our orchestration framework where users can assemble complete simulations in compact python scripts, and the framework runs them (details in §A.1).

5.2 Host Simulator

We integrate two host component simulators, gem5 and QEMU, capable of running unmodified operating systems and applications. We implement the integration as a regular PCIe device within the respective abstractions in the simulators.

gem5. gem5 is a flexible full system simulation with configurable level of detail for memory and CPU simulation. We use version v20.0.0.1 and extend it with a patch for Intel DDIO support [2]. We implement support for the functional and timing memory access protocols. The functional model is blocking, i.e. expects accesses to immediately return results, and does not model timing. The timing protocol is a natural fit for SimBricks as it includes asynchronous request and response messages for each access. We do not support the atomic protocol, where operations immediately return and indicate how long the operation should take, as this is incompatible with SimBricks's asynchronous interfaces. To reduce simulation time, we leverage flexibility in gem5 to boot up with a fast functional CPU, and switching to a detailed model after. We additionally implement an Ethernet adapter to also connect the few basic NICs in gem5 to SimBricks for comparison.

QEMU. We use QEMU (version 5.1.92) with KVM CPU acceleration for fast functional simulation, and also implement support for synchronized simulation with QEMU's instruction counting feature. With instruction counting (`icount`), QEMU controls the rate of instruction execution relative to a virtual clock. The key challenge is to model MMIO access timings, 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 completion event arrives on the PCIe interface. At that point QEMU will re-try executing the instruction.

5.3 NIC Simulations

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

Corundum RTL simulation. To demonstrate the feasibility of integrating realistic RTL simulation into SimBricks, we use the unmodified Verilog implementation of Corundum [16], an open-source, FPGA NIC. We use the Verilator [49] RTL simulator to simulate the `interface` module implementing Corundum’s core data path (which includes rx, tx, descriptor queues, checksumming, packet scheduling etc.). When implementing on an FPGA, Corundum instantiates vendor IP for the remaining components, including the PCIe interfaces, DMA engine, and Ethernet MAC. As Verilator cannot simulate vendor IP, we implement this functionality directly in the C++ testbench, where we interface with SimBricks.

Corundum behavioral model. To enable an apples-to-apples comparison, we also implement a behavioral model for Corundum. The model is fully compatible with the open-source Corundum Linux driver [15]. The implementation currently only supports a single interface and one receive and transmit queue pair. Advanced features such as checksum offloading and receive-side scaling are future work.

i40e behavioral model. A modern NIC simulator compatible with Linux and kernel-bypass frameworks such as DPDK [20] is needed for simulating today’s network systems. We implement a behavioral model of the common i40e Intel 40G X710 NIC. This simulator is compatible with unmodified drivers and, while not feature complete, implements the most important 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.

5.4 Network Simulations

ns-3. To integrate with ns-3.31, we implement a derived class of `NetDevice`, implementing the SimBricks Ethernet interface. `NetDevice` is the ns-3 base abstraction for all end-host network hardware. When receiving Ethernet packets from the NIC, the adapter pushes these to the connected network channel. When the adapter receives a packet from the ns-3 network channel, it simply forwards it to the SimBricks channel. The adapter also implements the synchronization protocol, as shown in Fig. 5. We use various ns-3 configurations for our experiments. S

Ethernet switch. Similar to NIC behavioral models we also implement a fast model of simple Ethernet switch. The model uses L2 MAC learning with infinite MAC table size. Each switch port is implemented using SimBricks Ethernet interface. In each iteration of 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 to each port if necessary.

Tofino. Finally, we integrate the proprietary Intel Tofino [5] simulator provided by Intel in the development toolkit (SDE) [18]. This simulator includes a cycle accurate model of the switch pipeline and an approximate model of the queuing subsystem. Unfortunately our version of the simulator is closed source and only communicates through Linux Kernel virtual Ethernet interfaces (`veth`), precluding synchronization. We thus only implement a functional adapter that transfers packets between SimBricks Ethernet interfaces and `veth` ports.

6 Evaluation

We have already shown that SimBricks reproduces behavior of a physical testbed for congestion control, and improves accuracy over ns-3. We now aim to answer the following questions:

- Can SimBricks modularly combine component simulators? How do different combinations perform? (§6.2)
- What is the synchronization overhead for component simulators? How does it compare to prior approaches? (§6.3)
- Can SimBricks reduce simulation time by breaking down components into smaller pieces and parallelizing them? (§6.4)
- How does simulation time scale with number of components combined? (§6.5)
- Can SimBricks effectively and scalably distribute large-scale simulations across multiple physical hosts? (§6.6)
- Can SimBricks reproduce key results of recent work on in-network compute (§6.7), and NIC architecture (§6.8)?
- Does SimBricks enable additional insights that cannot be obtained from physical testbeds? (§6.8)

6.1 Experimental Setup

We use servers 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 RoCE NICs.

Unless otherwise stated use following simulator parameters. All simulations are running Ubuntu 18.04 with a Linux version 5.4.46 kernel where we disabled unneeded features and drivers to reduce boot time. No drivers or applications were modified. Each host has one core and 8 GB of memory. For QEMU with synchronization (QT) 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 required for SimBricks synchronization (§5.2). We set parameters to achieve the

Use-case Simulator Combination	netperf		Sim. Time
	T'put	Latency	
SW debugging QEMU-kvm + behavioral i40e NIC + behavioral switch	4.37 G	71 μ s	00:00:32
SW perf. evaluation gem5 + behavioral i40e NIC + ns-3	9.05 G	20 μ s	18:18:49
HW debugging QEMU-kvm + Corundum Verilog + behavioral switch	81 M	3.4 ms	00:00:31
HW perf. evaluation QEMU-timing + Corundum Verilog + behavioral switch	6.27 G	33 μ s	04:23:07

Tab. 1: SimBricks configurations for four typical use-cases, with the resulting simulation time and measured app. performance.

same effective instruction execution performance as a representative physical testbed [24], for a Linux network stack benchmark at 1.3 cycles/inst = 0.43 ns/inst. We match cache sizes and latencies of the testbed. With an in-order CPU clock frequency 8 GHz we get the same effective instruction rate as the testbed. Gem5 also supports an out-of-order CPU, but at a simulation time of 4 – 6 \times higher, so we use the timing CPU as a compromise. Network links are set to 10 Gbps by default. The Corundum verilator model runs at 250 MHz. Further, we set the PCIe latency, NIC-network link latency and synchronization interval all to 500 ns, unless otherwise reported.

6.2 Modular Simulation

We start by evaluating modular combinations of component simulators in SimBricks. For this, we use the netperf TCP benchmark to run a 10s throughput test (TCP_STREAM) followed by a 10s latency test (TCP_RR) on two simulated hosts. We focus on a subset of four combinations for common systems research use-cases in: debugging and performance evaluation of hardware and software prototypes respectively.

For debugging HW & SW we are most productive when we can interactively use the system, while accurate performance, and thus synchronization is not required. Here we combine QEMU with kvm for fast host simulation with the fast basic switch, and either the intel NIC model for SW testing or Verilator with Corundum as a HW example. During performance evaluation we do want accurate performance results, but can tolerate longer simulation times. For software performance measurements we rely on a more detailed host simulator with gem5, while here we assume that we need less detail of the host for benchmarking our HW prototype.

Navigating speed-accuracy trade-offs with modularity. Our results reported in Tab. 1 confirm the trade-off between

simulation time and model detail; simulation times range from 31s to 18 hours for the same workload. The results show that, SimBricks can effectively help navigate this trade-off by only using detailed simulators for components where detail matters. The measured performance for unsynchronized configurations, especially for the QEMU-kvm and Corundum, while not representative, are still sufficient to test and debug the 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. We evaluate the full cross-product of simulator choices and confirm that SimBricks supports all combinations (performance results in §A.4).

6.3 SimBricks Synchronization

We now use gem5 as a case study to measure synchronization behavior in three experiments.

Synchronization overhead. To measure the overhead of SimBricks synchronization, we run two workloads with gem5, once standalone and once in SimBricks, and compare simulation time. Neither experiment uses the network, but for SimBricks we connect the the Intel NIC model, and that to the switch. The SimBricks adapter thus exchanges sync messages every 500 ns.

The first workload has a low-event density in gem5: executing sleep 10. The standalone simulation takes 2.25 min and in SimBricks 2.91 min, a 30% overhead. Here gem5 is almost exclusively handling SimBricks synchronization events, as the CPU is mostly halted. As a high-event density 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. In both cases, *SimBricks integration incurs manageable synchronization overhead, and is unlikely to significantly slow down already slow simulators.*

Comparison to dist-gem5. Next, we compare to dist-gem5 [37] as a baseline system that employs a conventional epoch-based global synchronization protocol over TCP sockets, to interconnect multiple gem5 instances. We configure simulations with 2 to 32 instances of gem5 that communicate pairwise using iperf, through the e1000 NIC in gem5 and a 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 significantly more efficient than dist-gem5, especially at scale.* With 2 hosts SimBricks reduces simulation time by 27% and for 32 hosts by 74%.

Sensitivity to link latency. SimBricks synchronization overhead is fundamentally linked with the configured link latency, as a lower bound for how frequently sync messages are required. So does SimBricks synchronization grow unreasonably expensive at lower latencies? To answer this, we measure synchronization time for a pair of gem5 hosts running `netperf` connected to the Intel NIC and the switch, while we vary the configured PCIe latency and sync interval. We report our results in Fig. 9, and find that while synchronization time does increase, *lowering the latency by two orders of magnitude from 1 μ s to 10 ns, only increases simulation time by 38%.*

6.4 Decomposition for Performance

In SimBricks, we can partition individual simulators into multiple connected smaller simulators for increased parallelism and scalability, in two separate instances.

Running NIC outside of gem5 is faster. We simulate NICs separately for modularity, while gem5 and qemu also directly include NICs. We evaluate the performance implications of this design choice, 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 Intel NIC. In both cases we run a pair of hosts connected to our switch. The first configuration takes 350 minutes, and the second only 138 minutes. Despite simulating a more complex NIC with additional features, *the 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 to many NICs, while hosts and NICs only connect to one and two peers, respectively. We first confirm this with a microbenchmark. We implement a packet generator as a dummy NIC that implements the SimBricks Ethernet interface and synchronization protocol, and simply injects packet 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. Next, we set the packet rate to 100 Gbps, and measure an increase from 12.6 s to 211.6 s. *A single network simulator can become a bottleneck for fast simulations,* but we have so far not observed this outside of this microbenchmark.

Scaling by parallelizing network simulation. To address this bottleneck in SimBricks, we can run multiple network simulators carved up at natural boundaries (e.g. switches or groups thereof). We demonstrate this by modifying the previous microbenchmark to divide the 32 hosts to 4 "ToR"

switches that each connect to the fifth "core" switch. With this configuration the simulation time for rate 0 is 9.6 s down by 45% compared to the single switch, and 96.8 s at 100 Gbps, 53%. *Decomposing networks reduces simulation time at scale.*

6.5 Local Scalability

A design goal for SimBricks is to scale to simulate systems with many hosts. Here we measure how simulation time changes as we vary the number of simulated gem5 hosts and Intel NICs on a single physical host, connected to a single switch. We set up one server and a variable number of client hosts, running the same UDP 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 by 48% to 205 min with 21 hosts.

Surprisingly this is *not* because of scalability bottlenecks in SimBricks synchronization. Instead, we discovered that this increase is due to thermal throttling in 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, 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.*

6.6 Distributed Simulations

So far the limiting factor for simulating larger systems is the number available cores. Now we move on to SimBricks simulations running across multiple physical hosts, using our proxies. Fig. 11 shows the configuration.

Overhead of distributed simulation. First we compare performance for small local simulations to distributed simulations with the SimBricks sockets and RDMA proxy, to measure overheads. We use two qemu-kvm hosts running `netperf` connected to Intel NICs which connect to the same switch. Locally, this unsynchronized simulation again yields a throughput of 4.4 Gbps, and a latency of 71 μ s. 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 μ s 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. *The SimBricks proxies are no bottleneck for synchronized simulations.*

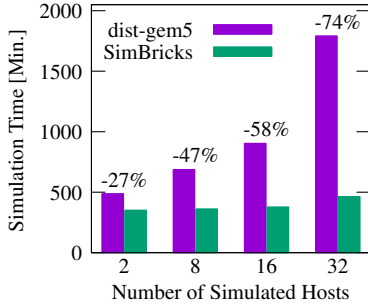


Fig. 6: dist-gem5 vs. SimBricks.

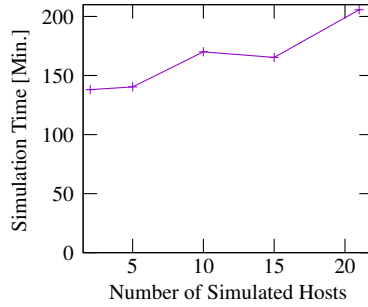


Fig. 7: SimBricks local scalability.

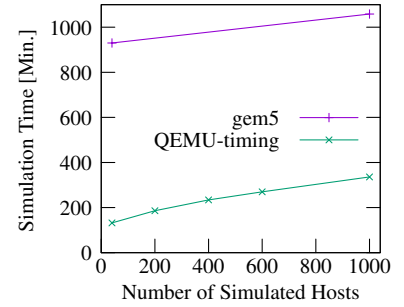


Fig. 8: Distributed memcached.

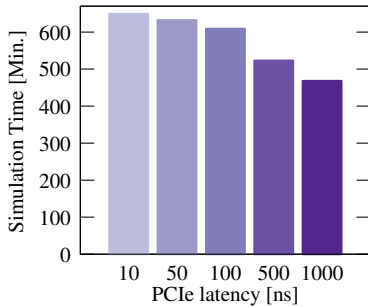


Fig. 9: Sensitivity to link latency.

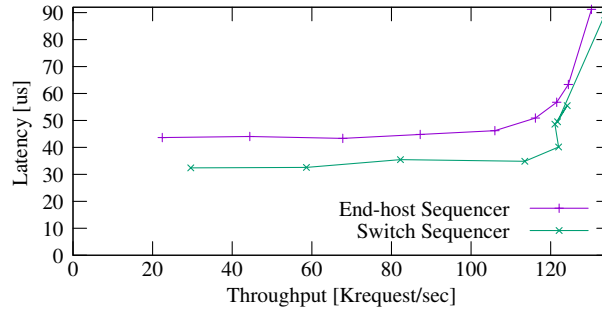


Fig. 10: NOPaxos in SimBricks with ns-3 switch sequencer and sequencer on a simulated host.

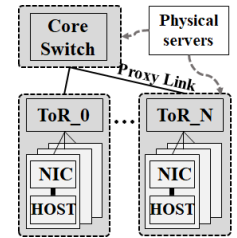


Fig. 11: Large scale simulation configuration

Large-scale memcache cluster. To evaluate scalability to realistic data center systems, we next run multiple distributed simulations ranging from 40 to 1000 simulated hosts, on up to 26 physical servers. We run these simulations on Amazon ec2 c5.metal instances, with 96 hyperthreads each, and 20 Gbps network connectivity in a single proximity placement group. We structure the simulation with a varying number of racks of 40 hosts with Intel NICs and a top of rack (ToR) switch each, that then connect to a single core switch, as shown in Fig. 11. When distributing the core switch gets assigned to a dedicated server, and each rack is assigned a dedicated server. A separate sockets proxy pair (ec2 does not have RDMA support) connects each ToR to the core switch. As the workload, we run memcached on half of all hosts in each rack, and the memslap 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. The experiment runs for 10 (simulated) seconds.

Fig. 8 shows the measured simulation time as we vary the number of simulated hosts by adding racks. As we increase from one rack and 40 hosts to 25 racks and 1000 hosts, the simulation time with gem5 hosts increases by 13.8% from 15.5 h, to 17.6 h. With QEMU-timing hosts, simulation time increases from 2.2 h to 5.6 h by 2.5 \times . With profiling and instrumentation we found that this is a "straggler" problem to do with QEMU's of dynamic binary translation. When a

QEMU instance misses in its code cache and has to recompile a block, the instance sometimes blocks for a while. While rare, at scale these occurrences are more frequent and because of the synchronization can slow down other hosts. But even then QEMU simulation time remains well below gem5. We conclude that *SimBricks scales to simulate systems with 100s of hosts.*

6.7 Use-Case: In-Network Processing

Leveraging programmable switches for application acceleration is another use-case that requires end-to-end measurements. Much of this work relies on switch features that are not yet available in off-the-shelf hardware, but can be easily implemented in simulation. We use Network-Ordered Paxos (NOPaxos) [27] to demonstrate that SimBricks can serve as a virtual testbed for these systems too. NOPaxos introduces a new network-level primitive, the ordered unreliable multicast (OUM), which requires a single sequencer device in the network. The ideal place to implement the sequencer is in a programmable network switch. But as the required hardware was not yet available, the authors relied on emulation on a network processor or implementation on a network end-host. We implement switch support for OUM in ns-3 and combine this with gem5 and the Intel NIC to evaluate NOPaxos. Inside the simulation we run the unmodified open source NOPaxos

code.

Performance evaluation with ns-3. In Fig. 10, we show throughput-latency curves for two configurations, with the switch sequencer and with the end-host-sequencer (included in the NOPaxos source). We compare this to figure 6 in the NOPaxos paper, where NOPaxos achieves a latency of $110\ \mu\text{s}$, while the end-host sequencer has latency about 35% higher and both systems achieve the same throughput of 230 K/s. In SimBricks we find a lower baseline latency of $32\ \mu\text{s}$ and 34% higher for the end-host sequencer. This is expected as the authors used a slower network processor to emulate functionality. Both systems saturate at the same throughput of 130 K/s. The lower throughput is because we are measuring on a single-core host, where application and packet processing run on the same core. We confirmed this in a physical testbed by disabling all but one core, and measured a throughput within 10%. We conclude that *SimBricks can evaluate in-network processing systems.*

Testing P4 implementation on Tofino. Now that P4 switches are readily available, the NOPaxos authors have implemented OUM in P4. We have obtained the implementation from the authors, and deploy it in the Tofino simulator. In combination with QEMU-kvm and the Intel NIC we verify that we can run the full unmodified NOPaxos from the physical testbed in SimBricks. As the Tofino simulator does not support synchronization we cannot use it for performance measurements.

6.8 Use-Case: NIC Hardware Architecture

We have already demonstrated that both the simulation of the unmodified Corundum RTL and our behavioral model support integration into end-to-end systems with unmodified drivers. Next, we show that SimBricks simulations can provide insights that are challenging to obtain from physical testbeds. The original Corundum evaluation shows that, especially for 1500B MTU, Corundum achieves significantly throughput than the ConnectX-5 NIC they compare to. While developing our Corundum simulators, we found a likely reason for this: unlike other NICs, Corundum relies on reading the head index registers of receive descriptor queues to identify new entries. For most NICs drivers instead directly poll descriptors in memory. This incurs overhead as 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 can significantly degrade performance.

Leveraging simulation visibility & flexibility. We observed this behavior from inspecting detailed simulator logs to debug high latencies we observed. For synchronized simulations, logging can be enabled without affecting system behavior. We leveraged this to log detailed PCI activity, NIC

activity, and CPU activity, and combined those into an end-to-end view of the latency. We further confirm this hypothesis by doubling the simulated PCIe latency to $1\ \mu\text{s}$ in gem5 with the corundum behavioral model and the Intel behavioral model. When PCIe latency doubles, Corundum throughput reduces by 21.2%, while the Intel NIC throughput remains consistent. *Simulators have two key advantages here, visibility and the flexibility to change key parameters that are fixed in physical systems.*

7 Discussion

Going forward we see three challenges for SimBricks: reducing validation effort, efficiently simulating multi-core hosts, and moving beyond our current focus of networking.

7.1 Validation

Simulations are only useful for system performance evaluation if they produce meaningful results representative of an equivalent physical system. To obtain meaningful results, users have to validate the combination of simulators and configuration parameters against physical testbeds. Initial validation effort creates a hurdle for using simulation, and when simulators or configuration parameters are modified, the complete simulation needs to be re-validated. This is further complicated when a comparable physical testbed is not available.

Modular validation. While SimBricks cannot avoid the need for validation, we argue that our approach reduces validation effort through modularity. Instead of validating each combination of simulators, components can be validated individually and then composed. As SimBricks ensures correct synchronization, composition does not affect accuracy of individual components, as long as the interfaces have also been validated. This enables users to combine previously validated component configurations into a full system. And when changing a component simulator, users only need to re-validate that component.

Repository of validated configurations. We propose a public repository of validated component simulator configurations to simplify re-use. This allows users to find configurations to start with, and contribute their own validated configurations, both reducing the effort for building simulations and for validating them. Ensuring correctness and validity of such configurations over time is a challenge. We are planning to address this in an automated continuous-integration system, periodically re-running configurations, recording the results, and making the history of results available.

7.2 Multi-Core Hosts

We limit our evaluation to single-core hosts for a pragmatic reason: for gem5 and QEMU with synchronization, simulation time increases super-linearly with the number of simulated cores, as both simulate multi-cores sequentially. The scalable X86 simulators we are aware of [17, 33] only simulate applications and do not include an operating system and I/O devices.

Scalable multi-core simulation. For scalable multi-core simulation we envision applying our techniques inside the boundaries of a single machine. Today’s multi-cores are fundamentally distributed systems [7] with cache interconnects acting as the network. Cache interconnects incur latencies in the range of what we have already evaluated (§6.3). We thus believe our techniques could be applied to simulating multi-cores with greater parallelism and reduced simulation time.

7.3 Beyond Networking

While we evaluate SimBricks for network systems, our design is generally applicable to other types of systems. For example, integration of PCIe attached accelerators does not require changes to SimBricks, as our PCIe protocol is not specific to networking. The SimBricks architecture can also be easily extended with additional components or interfaces, such as CXL [14].

A natural first extension of SimBricks we envision is to include simulators for accelerators, which are also attracting growing interest in the systems and networking community. Especially for ASIC accelerator designs, such as Google’s TPU [21], simulation is often the only way to evaluate these systems. As these accelerators typically form parts of larger distributed systems, our approach that enables end-to-end evaluation is essential. We also expect the emergence of further use-cases as computer architects and systems researcher delve deeper into the realm of specialized hardware.

8 Related Work

Parallel Simulation. `dist-gem5` [37] and `pd-gem5` [1] connect multiple gem5 instances to enable simulation of distributed systems and use global barriers for synchronization. Graphite [33] also parallelizes a multi-core simulation across cores and machines, but uses approximate synchronization where causality are possible. Similar to gem5, Simics [31] 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.

Co-Simulation of Multiple Simulators. gem5 supports the integration of systemC code [32] to implement hardware models, by linking them into the gem5 binary and embedding the systemC event loop with the gem5 event loop. SimBricks instead aims to interconnect multiple heterogeneous simulators with potentially completely different simulation models. The Structural Simulation Toolkit (SST) [43] is a modular simulation framework for HPC clusters, uses a parallel discrete event simulator as the core, 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.

Evaluating network systems in simulation. Simulations and emulations have been used by past systems to facilitate end-host networking research. SINIC [11] is a NIC design that leverages M5 [10] to perform full-system simulation running network workloads. Scale-Out NUMA [38] uses the Flexus [50] full-system simulator to evaluate their remote memory controller design in a distributed memory system. NeBuLa [47] similarly uses Flexus to simulate their new NIC and memory architecture design targeting RPC systems. Instead of ad-hoc simulations tied to a particular simulator, SimBricks enables component simulators that can be combined with different host and network simulators.

9 Conclusion

End-to-end simulation offer an alternative for rigorous evaluation when a physical testbed is not available. With SimBricks, we have presented a novel modular simulation framework targeted at end-to-end evaluation of network systems by combining multiple tried-and-true simulators for system components. SimBricks can replicate key findings from a broad range of prior work, including congestion control, in-network compute, and NIC hardware architecture. Moreover, SimBricks simulations are portable and reproducible. As we increasingly to look to hardware for performance improvements, we believe that end-to-end simulations are an essential tool for systems research.

Acknowledgments

We would like to thank the anonymous reviewers for their comments and feedback so far. We also thank Jeff Mogul, Peter Druschel, Simon Peter, Trevor E. Carlson, Aastha Mehta, and Katie Lim, for their input on early drafts of this paper.

References

- [1] Mohammad Alian, Daehoon Kim, and Nam Sung Kim. Pd-Gem5: Simulation infrastructure for parallel/distributed computer systems. *IEEE Computer Architecture Letters*, 15(1):41–44, January 2016.
- [2] Mohammad Alian, Yifan Yuan, Jie Zhang, Ren Wang, Myoungsoo Jung, and Nam Sung Kim. Data direct i/o characterization for future i/o system exploration. In *2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pages 160–169, 2020.
- [3] Mohammad Alizadeh, Albert Greenberg, David A. Maltz, Jitendra Padhye, Parveen Patel, Balaji Prabhakar, Sudipta Sengupta, and Murari Sridharan. Data center TCP (DCTCP). In *2010 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2010.
- [4] Mina Tahmasbi Arashloo, Alexey Lavrov, Manya Ghobadi, Jennifer Rexford, David Walker, and David Wentzlaff. Enabling programmable transport protocols in high-speed NICs. In *17th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2020.
- [5] Barefoot Networks. Barefoot Tofino. <https://barefootnetworks.com/products/product-brief-tofino/>.
- [6] Andrew Baumann, Paul Barham, Pierre-Evariste Daggand, Tim Harris, Rebecca Isaacs, Simon Peter, Timothy Roscoe, Adrian Schüpbach, and Akhilesh Singhanian. The Multikernel: A new OS architecture for scalable multicore systems. In *22nd ACM Symposium on Operating Systems Principles*, SOSP, 2009.
- [7] Andrew Baumann, Simon Peter, Adrian Schüpbach, Akhilesh Singhanian, Timothy Roscoe, Paul Barham, and Rebecca Isaacs. Your computer is already a distributed system. why isn’t your OS? In *12th Workshop on Hot Topics in Operating Systems*, HOTOS, 2009.
- [8] Brian N. Bershad, Thomas E. Anderson, Edward D. Lazowska, and Henry M. Levy. User-level interprocess communication for shared memory multiprocessors. *ACM Transactions on Computer Systems*, 9(2):175–198, May 1991.
- [9] 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. The Gem5 simulator. *SIGARCH Computer Architecture News*, 39(2):1–7, August 2011.
- [10] Nathan L. Binkert, Ronald G. Dreslinski, Lisa R. Hsu, Kevin T. Lim, Ali G. Saidi, and Steven K. Reinhardt. The M5 simulator: Modeling networked systems. *IEEE Micro*, 26(4):52–60, July 2006.
- [11] Nathan L. Binkert, Ali G. Saidi, and Steven K. Reinhardt. Integrated network interfaces for high-bandwidth TCP/IP. In *12th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2006.
- [12] Pat Bosshart, Glen Gibb, Hun-Seok Kim, George Varghese, Nick McKeown, Martin Izzard, Fernando Mujica, and Mark Horowitz. Forwarding metamorphosis: Fast programmable match-action processing in hardware for SDN. In *2013 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2013.
- [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. dRMT: Disaggregated programmable switching. In *2017 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2017.
- [14] CXL Consortium. Compute express link (CXL). <https://www.computeexpresslink.org/spec-landing>, October 2020. Revision 2.0.
- [15] Alex Forencich, Alex C. Snoeren, George Porter, and George Papan. Corundum github repository. <https://github.com/corundum/corundum>.
- [16] Alex Forencich, Alex C. Snoeren, George Porter, and George Papan. Corundum: An open-source 100-Gbps NIC. In *28th IEEE Annual International Symposium on Field-Programmable Custom Computing Machines*, FCCM, 2020.
- [17] Yaosheng Fu and David Wentzlaff. PriME: A parallel and distributed simulator for thousand-core chips. In *2014 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2014.
- [18] Intel. Intel P4 Studio. <https://www.intel.com/content/www/us/en/products/network-io/programmable-ethernet-switch/p4-suite/p4-studio.html>.
- [19] Intel Corporation. Intel Ethernet controller X710/XXV710/XL710 datasheet. <https://cdrdv2.intel.com/v1/dl/getContent/332464>, October 2020. Revision 3.7.
- [20] Intel data plane development kit. <http://www.dpdk.org/>.

- [21] 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 Gotipati, 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. In-datacenter performance analysis of a tensor processing unit. In *44th Annual International Symposium on Computer Architecture*, ISCA, 2017.
- [22] 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. FireSim: FPGA-accelerated cycle-exact scale-out system simulation in the public cloud. In *45th Annual International Symposium on Computer Architecture*, ISCA, 2018.
- [23] Antoine Kaufmann, Simon Peter, Naveen Kr. Sharma, Thomas Anderson, and Arvind Krishnamurthy. High performance packet processing with FlexNIC. In *21st International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2016.
- [24] Antoine Kaufmann, Tim Stamler, Simon Peter, Naveen Kr. Sharma, Arvind Krishnamurthy, and Thomas Anderson. TAS: TCP acceleration as an OS service. In *14th ACM European Conference on Computer Systems*, EuroSys, 2019.
- [25] 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. Swift: Delay is simple and effective for congestion control in the datacenter. In *2020 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2020.
- [26] Jialin Li, Ellis Michael, and Dan R. K. Ports. Eris: Coordination-Free Consistent Transactions Using In-Network Concurrency Control. In *Proceedings of the 26th Symposium on Operating Systems Principles*, SOSP '17, Shanghai, China, 2017. Association for Computing Machinery.
- [27] Jialin Li, Ellis Michael, Naveen Kr. Sharma, Adriana Szekeres, and Dan R. K. Ports. Just say NO to paxos overhead: Replacing consensus with network ordering. In *12th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2016.
- [28] Jiaxin Lin, Kiran Patel, Brent E. Stephens, Anirudh Sivaraman, and Aditya Akella. PANIC: A high-performance programmable NIC for multi-tenant networks. In *14th USENIX Symposium on Operating Systems Design and Implementation*, OSDI, 2020.
- [29] Ming Liu, Liang Luo, Jacob Nelson, Luis Ceze, Arvind Krishnamurthy, and Kishore Atreya. Incbricks: Toward in-network computation with an in-network cache. In *22nd International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2017.
- [30] Ikuo Magaki, Moein Khazraee, Luis Vega Gutierrez, and Michael Bedford Taylor. Asic clouds: Specializing the datacenter. In *43rd Annual International Symposium on Computer Architecture*, ISCA, 2016.
- [31] Peter S Magnusson, Magnus Christensson, Jesper Eskilson, Daniel Forsgren, Gustav Hallberg, Johan Hogberg, Fredrik Larsson, Andreas Moestedt, and Bengt Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2):50–58, August 2002.
- [32] Christian Menard, Jeronimo Castrillon, Matthias Jung, and Norbert Wehn. System simulation with gem5 and SystemC: The keystone for full interoperability. In *2017 International Conference on Embedded Computer Systems: Architectures, Modeling, and Simulation*, SAMOS, 2017.
- [33] Jason E. Miller, Harshad Kasture, George Kurian, Charles Gruenwald, Nathan Beckmann, Christopher Celio, Jonathan Eastep, and Anant Agarwal. Graphite: A distributed parallel simulator for multicores. In *16th IEEE International Symposium on High-Performance Computer Architecture*, HPCA, 2010.
- [34] Radhika Mittal, Vinh The Lam, Nandita Dukkipati, Emily Blem, Hassan Wassel, Monia Ghobadi, Amin Vahdat, Yaogong Wang, David Wetherall, and David Zats. TIMELY: RTT-based congestion control for the datacenter. In *2015 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2015.

- [35] Radhika Mittal, Alexander Shpiner, Aurojit Panda, Eitan Zahavi, Arvind Krishnamurthy, Sylvia Ratnasamy, and Scott Shenker. Revisiting network support for RDMA. In *2018 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2018.
- [36] ModelSim ASIC and FPGA design – Mentor Graphics. <https://www.mentor.com/products/fv/modelsim/>.
- [37] Alian Mohammad, Umur Darbaz, Gabor Dozsa, Stephan Diestelhorst, Daehoon Kim, and Nam Sung Kim. dist-gem5: Distributed simulation of computer clusters. In *2017 IEEE International Symposium on Performance Analysis of Systems and Software*, ISPASS, 2017.
- [38] Stanko Novakovic, Alexandros Daglis, Edouard Bugnion, Babak Falsafi, and Boris Grot. Scale-out NUMA. In *19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS, 2014.
- [39] The network simulator - ns-2. <https://www.isi.edu/nsnam/ns/>.
- [40] ns-3 | a discrete-event network simulator for internet systems. <https://www.nsnam.org/>.
- [41] QEMU – the FAST! processor emulator. <https://www.qemu.org/>.
- [42] Steven K Reinhardt, Mark D Hill, James R Larus, Alvin R Lebeck, James C Lewis, and David A Wood. The Wisconsin Wind Tunnel: virtual prototyping of parallel computers. In *1993 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Science*, SIGMETRICS, 1993.
- [43] 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. The structural simulation toolkit. *ACM SIGMETRICS Performance Evaluation Review*, 38(4):37–42, March 2011.
- [44] Naveen Kr. Sharma, Antoine Kaufmann, Thomas Anderson, Arvind Krishnamurthy, Jacob Nelson, and Simon Peter. Evaluating the power of flexible packet processing for network resource allocation. In *14th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2017.
- [45] Naveen Kr. Sharma, Ming Liu, Kishore Atreya, and Arvind Krishnamurthy. Approximating fair queueing on reconfigurable switches. In *15th USENIX Symposium on Networked Systems Design and Implementation*, NSDI, 2018.
- [46] Anirudh Sivaraman, Alvin Cheung, Mihai Budiu, Changhoon Kim, Mohammad Alizadeh, Hari Balakrishnan, George Varghese, Nick McKeown, and Steve Licking. Packet transactions: High-level programming for line-rate switches. In *2016 ACM SIGCOMM Conference on Data Communication*, SIGCOMM, 2016.
- [47] Mark Sutherland, Siddharth Gupta, Babak Falsafi, Virendra Marathe, Dionisios Pnevmatikatos, and Alexandros Daglis. The NeBuLa RPC-optimized architecture. In *47th Annual International Symposium on Computer Architecture*, ISCA, 2020.
- [48] András Varga and Rudolf Hornig. An overview of the OMNeT++ simulation environment. In *1st International Conference on Simulation Tools and Techniques for Communications, Networks and Systems & Workshops*, Simutools, 2008.
- [49] Verilator – the fastest verilog HDL simulator. <https://www.veripool.org/wiki/verilator>.
- [50] Thomas F Wenisch, Roland E Wunderlich, Michael Ferdman, Anastassia Ailamaki, Babak Falsafi, and James C Hoe. SimFlex: statistical sampling of computer system simulation. *IEEE Micro*, 26(4):18–31, August 2006.
- [51] Xilinx. Vivado simulator. <https://www.xilinx.com/products/design-tools/vivado/simulator.html>.

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 channels 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 = IperfUDPSTerver()

    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)
```

Fig. 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.

```
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
```

Fig. 13: SimBricks multi-core shared memory message passing queue. READMEMORYBARRIER and WRITEMEMORYBARRIER are compiler barriers to prevent re-ordering during optimization.

A.2 Inter-Simulator Message Transport

Fig. 13 shows pseudo-code 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 it 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.

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 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 Implementation effort for SimBricks

	SimBricks Component	Lines
common	message transport library	766
libraries	NIC behavioral model library	484
host	gem5 integration	1146
simulators	QEMU integration	724
NIC	Corundum Verilator	1394
simulation	i40e model	3055
	Corundum model	979
network	ns-3 integration	1303
simulation	Ethernet switch model	165
runtime	runtime orchestration	1237
proxy	distributed simulation proxy	2117

Tab. 2: Lines of code for the various components in SimBricks, excluding blank lines and comments.

A.4 Performance for SimBricks Simulator Configurations

Tab. 3

Simulators					Sim.
Host	NIC	Net	T'put	Latency	Time
QK	IB	SW	4.37 G	71 μ s	00:00:32
QK	IB	NS	409 M	141 μ s	00:00:32
QK	IB	TO	1.92 M	6.6 ms	00:00:33
QK	CB	SW	1.84 G	211 μ s	00:00:29
QK	CB	NS	429 M	294 μ s	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	8.88 G	17 μ s	02:03:17
QT	IB	NS	8.87 G	17 μ s	02:14:43
QT	CB	SW	6.39 G	25 μ s	02:01:36
QT	CB	NS	6.41 G	25 μ s	02:02:41
QT	CV	SW	6.27 G	33 μ s	04:23:07
QT	CV	NS	6.52 G	33 μ s	04:37:17
G5	IB	SW	9.05 G	20 μ s	18:18:49
G5	IB	NS	9.02 G	20 μ s	17:01:53
G5	CB	SW	3.01 G	33 μ s	11:50:48
G5	CB	NS	3.00 G	33 μ s	12:09:29
G5	CV	SW	6.69 G	37 μ s	11:57:34

Tab. 3: Complete cross-product combinations of our component simulators. Host simulators: QK is QEMU with KVM (functional simulation), QT is QEMU with timing, and G5 is gem5. NIC Simulators: IB is the Intel behavioral model, CB the Corundum behavioral model, and CV the Corundum verilator model. Network Simulators: SW is the switch behavioral model, NS is ns-3.