Journal of Parallel and Distributed Computing 204 (2025) 105136

Contents lists available at ScienceDirect

Journal of Parallel and Distributed Computing

journal homepage: www.elsevier.com/locate/jpdc

The CAMINOS interconnection networks simulator

Cristóbal Camarero [*], Daniel Postigo, Pablo Fuentes

Universidad de Cantabria, Facultad de Ciencias, Avda. Los Castros S/N, Santander, Spain

ARTICLE INFO

Keywords:
CAMINOS
Network simulator
Rust
Performance
Router microarchitecture

ABSTRACT

This work presents CAMINOS, a new interconnection network simulator focusing on router microarchitecture. It was developed in Rust, a novel programming language with a syntax similar to C/C++ and strong memory protection.

The architecture of CAMINOS emphasizes the composition of components. This allows new designs to be defined in a configuration file without modifying source code, greatly reducing effort and time.

In addition to simulation functionality, CAMINOS assists in managing a collection of simulations as an experiment. This includes integration with SLURM to support executing batches of simulations and generating PDFs with results and diagnostics.

We show that CAMINOS makes good use of computing resources. Its memory usage is dominated by in-flight messages, showing low overhead in memory usage. We attest that CAMINOS can effectively use CPU time, as scenarios with little contention execute faster.

1. Introduction

In the development of large computer systems, such as the *Frontier* supercomputer [1] that achieved the *Exaflop* milestone, a high-performance interconnect that delivers high throughput and low latency becomes indispensable, lest it become a performance bottleneck for the system. For such systems, estimating the interconnect's behavior accurately in various scenarios with high reproducibility is extremely useful. A network simulator provides a valuable tool to handle different evaluations without the constraints and costs of developing a hardware prototype. This allows for easy analysis of different characteristics of the interconnect, including the network topology that describes its connections, the routing protocol, the flow-control mechanism, or the low-level details of the router microarchitecture. This approach eliminates the need to deploy a physical prototype to test a new feature, significantly reducing manufacturing costs for new designs [2]. A comparable scenario arises for the communications between multiple cores within a chipset across a Network-on-Chip (NOC). At this level, network simulators avoid the specifics of higher-level network protocols or the physical implementation of the links. Instead, reference values, where appropriate, are employed to assess performance under different communication scenarios and traffic loads. The main purpose is to establish the impact of the given aspects of the router microarchitecture on the general performance of an interconnect.

This paper introduces CAMINOS (Cantabrian Adaptable and Modular Interconnection Network Open Simulator),[1] a network simulator written in the Rust language [3] that models the underlying router microarchitecture. It is oriented toward analyzing and evaluating lossless network interconnects at the High-Performance Computing (HPC) and Datacenter domains. It is an event-driven simulator that operates at the level of physical digits called *phits*, the smallest data unit that can move in a cycle. CAMINOS has already been employed in published research, such as [4–6]. CAMINOS can be found at [7].

The paper is organized as follows. The introduction explains the need for a new network simulator and the benefits of using the Rust language. Section 2 presents the state of the art and describes the main characteristics of comparable network simulators. Section 3 describes the structure of the simulator, while Section 4 covers the workflow, the input parameters, and the main output statistics. An example of network simulation with CAMINOS against a known scenario to validate its accuracy is presented in Section 5. Section 6 assesses the simulator's performance in terms of resource utilization. Finally, Section 7 concludes the work and remarks on the most important aspects.

* Corresponding author.
E-mail addresses: camareroc@unican.es (C. Camarero), daniel.postigo@unican.es (D. Postigo), fuentesp@unican.es (P. Fuentes).

[1] *Caminos* is also the Spanish word for paths.

https://doi.org/10.1016/j.jpdc.2025.105136
Received 12 March 2024; Received in revised form 24 February 2025; Accepted 12 June 2025

Available online 18 June 2025

1.1. Need for a new network simulator

As stated, a network simulator represents a useful tool for estimating the performance of a network interconnect and discovering pathological issues. By their very nature, network simulators are usually purposely developed for a given original target and require significant code additions to evaluate new proposals. This tends to introduce limitations that drag out further developments, hindering the work of network architects. For example, a simulator with a plain configuration may introduce parameters such as pairs `x/y`, which can later become confusing when extending the simulator for networks with more dimensions. This philosophy matches the benefits of the Zero-Cost Abstractions provided by the Rust language, which allow users to benefit from many high-level constructs without incurring performance costs.

CAMINOS has been developed with a focus on HPC and data center domains, although it can also analyze the performance of Networks-on-Chip. In these domains it is typical the use of lossless networks, where packets cannot be dropped, hence requiring deadlock avoidance mechanisms to avoid network stalls introduced by cyclic dependencies. CAMINOS focuses solely on lossless networks and does not model link errors due to network noise, that may cause information to be discarded and retransmitted, since the focus is on the topological characteristics and performance degradation introduced by the contention for network resources.

CAMINOS employs a discrete event simulation model, in which all occurrences are modeled by events [8]. It is clear that in the real world, changes are continuous but can be adequately modeled as discrete events with no changes in between. Moreover, as we consider systems with a well-defined clock, the time itself can be discretized using integers for the cycle in which an event occurs. Examples of these events may be to introduce a packet into the buffer of the next router at the time indicated by the link latency or to process the heads of buffers for access to the crossbar. The executions of these events may cause new events to be enqueued, driving the simulation.

CAMINOS models the network at a phit level, taking into consideration details of the router microarchitecture, such as the structures for handling resources (crossbar, allocators) and the ability to transmit or store information (buffers, links), to evaluate the performance impact of contention on the use of shared resources. Communications in the network are modeled through synthetic workloads, where the most common model is to follow a Bernoulli process. Section 3.2 provides a detailed description of the supported communication models in the simulator.

1.2. Benefits of the Rust language

Rust is a modern programming language born within Mozilla Research and reached its first stable release version in 2015. Its most essential aspects are similar to C or C++ but include constructs such as algebraic data types. The general philosophy of Rust is to allow the resulting executable to be as optimized as if it were hand-coded assembly and to allow the compiler to verify the correctness of the program as much as possible. A fundamental difference between C and C++ is the memory safety guaranteed in Rust's code, which prevents many common programming errors during compile time. Notably, Rust statically avoids `Out-of-bounds Write or Read`, `Use After Free` and `NULL Pointer Dereference`, which are among the 25 most dangerous weaknesses listed in the Common Weakness Enumeration of 2023 [9]. Rust helps prevent other mistakes beyond memory safety; for example, a function returning a `Result<T, E>` is less error-prone than one in C, where a negative number or NULL pointer could indicate the error.

In cases where Rust's lifetime-based analysis is not enough to prove access to be valid, a runtime check or the promise of its correctness using the unsafe keyword can be added. Thus, Rust does not sacrifice safety, performance, or abstraction and modeling capabilities. Safety is guaranteed by protecting all accesses in regular mode. Performance is achieved by making available a C-layout and other low-level features. Strong types with generics, traits, abstract data types, type inference, and lambda functions provide abstraction and modeling capabilities. Yet there is a cost, as it sometimes forces the programmer to reconsider the design choices, encouraging a more rigorous choice that can be statically proved safe.

A network simulator that includes large systems in its scope needs a language that provides good performance. Note that the requirements from the Rust programming model should not be conflated with those of functional programming, which has a more evident impact on performance. In a functional language, such as Haskell, some performance may be lost due to the stringent requirements on the mutability of data structures. Rust is designed to closely represent the machine's workings, annotating extra requirements for verification when possible. Only for very specific constructions must one decide whether to write precise but not automatically verifiable code or automatically verifiable code with some slight indirection. There are also some potential runtime checks, such as out-of-bound checks that the compiler fails to remove, for which an aware programmer can use the unchecked variants to remove their overhead [10]. Furthermore, the declarations that allow the automatic verification of memory safety can also be exploited for the optimization process. This effect is dominated by providing clear aliasing rules, whose lack in languages such as C can prevent the application of some low-level optimizations [11]. This allows Rust to excel over other languages to perform some specific tasks. Different benchmarks show Rust times to range between 50% to 400% from that of C/C++, depending on the implementation effort and tools employed [10,12,13].

In our experience, Rust imposes a higher cognitive load on the programmer than similar performant languages. Nevertheless, this is more than offset by the reduced time dedicated to debugging memory errors and the more seamless inclusion of higher-level designs. This approach is well suited for network simulators focusing on the router microarchitecture. Since evaluating new proposals frequently entails coding new features, greater trust in their validity speeds up the evaluation process.

1.3. The Rust toolchain

While not technically part of the language, the Rust toolchain is crucial in supporting development. In particular, the `rustc` compiler provides notably helpful error messages, aiding developers in identifying and resolving issues effectively. The cargo tool, akin to a Makefile but with additional features, stands out for its capacity to manage dependencies, simplifying the development process.

The cargo.io website extends the toolchain's functionality by offering a vast repository of libraries that can be easily used as dependencies. This facilitates immediate access to many functionalities, streamlining the development of projects like CAMINOS. Additionally, the `clippy` tool contributes by providing tips for correctness, simplicity, and performance, enhancing the overall quality of the code.

One notable feature is the standardized documentation comments, enabling the use of the `cargo doc` command to generate documentation pages. These pages are typically made available at https://doc.rs, significantly easing the process of creating well-documented libraries. This emphasis on documentation positively affects the quality of available libraries.

The Rust toolchain is known for its user-friendly nature and comprehensive set of features, including error management, dependency handling, and documentation support. This contributes to a robust development ecosystem. In our experience, the Rust toolchain and CAMINOS can be installed on quite old systems without issues. This is due to great efforts from the Rust team to make every version of the toolchain available for a wide range of machines. Rust's commitment to support various systems was demonstrated in a 2022 update. This update increased the minimum Linux requirements to glibc 2.17 (2012-12-25) and kernel

3.2 (2012-01-04).[2] This contrasts with other network simulators, such as SuperSim [14], where the Bazel toolchain demands several modern infrastructure components, some of which are missing in specific old systems. Although systems without internet access might pose challenges, the `cargo vendor` tool already builds a packed collection of project dependencies, ensuring usability in an offline environment.

2. State of the art

Network simulators are commonly used in network analysis and development to evaluate the performance under complex scenarios [15,16]. To the extent the authors are aware, there is no previous phit-accurate network simulator developed in the Rust language; all other similar tools described in this section are written in C or C++. BookSim [17], which bears its name for being the reference simulator used for the analyses presented in [16], has been extensively used in the literature to evaluate new network proposals [18–21]. Its second iteration (BookSim 2.0) is mainly intended to estimate the performance of NoCs, but it can and has been applied to evaluating HPC system interconnects. Apart from its extensive use in the research community, the accuracy of this simulator has also been validated against a *Register-Transfer Level* (RTL) implementation.

In contrast to CAMINOS, BookSim is a time-driven simulator that can increase execution time for evaluations with a low simulated traffic load. BookSim also presents a limitation related to the size of the simulated data units, being unable to model flow-transfer units (*flits*) with a size of multiple physical units (*phits*), that is, that require more than a single network cycle to transfer the information from the beginning to the end of the flit.

FSIN, part of the INSEE (*Interconnection Network Simulation and Evaluation Environment*) [22] framework developed at the Universidad del País Vasco, is very similar to BookSim, but with a lower adoption rate [23,24].

SuperSim [14] is a more modern simulator that is closer to CAMINOS in terms of a flexible and modular design. SuperSim was initially developed by Hewlett-Packard Labs and has been employed in analyzing a routing proposal for the HyperX network topology intended for HPC systems [25]. However, it is currently singlehandedly maintained by one of the original developers. Furthermore, this support is offered through a separate, private repository that is different from the official GitHub repository provided by Hewlett-Packard Labs. Similar to CAMINOS, SuperSim is an event-driven simulator that supports the simulation of flits composed of multiple phits. It also employs a hierarchical and structured syntax to define simulation experiments.

FOGsim [26] is another simulator of similar characteristics but designed exclusively for the simulation of Dragonfly networks, either under synthetic traffic loads or using execution traces of parallel applications. Despite its limitations, it has been used in several Ph.D. theses and multiple publications [27–29].

Other simulators provide less detail to focus on different aspects of the system. INRFlow [30] simulates traffic flows instead of network packets, which allows the simulation of much larger systems aimed at the data center and HPC domains but comes at the expense of being blind to many congestion problems. Garnet2.0 [31] is designed to be a module of the GEM5 full-system simulator [32] to assess the performance of a more complex system modeling a complete CPU with the memory subsystem and the execution of real applications. However, Garnet uses a simplified model of the router microarchitecture. It must be noted that BookSim can also be integrated into GEM5 through external work [33].

SST/Macro [34,35] does not provide flit-accurate simulations but is only accurate up to the packet level. Instead, it focuses on performing the communications of an actual MPI (the Message Passing Interface standard) application. Alternatively, it can simulate an application skeleton where the computations in the code have been removed.

Table 1
Comparison of network simulators.

Simulator	Detail level	Language	License	Latest commit
BookSim	flit	C++	privative	2017
CAMINOS	phit	Rust	MIT	2024
FSIN	phit	C	GPL 2.0	2017
FOGSim	phit	C++	GPL 2.0	2021
HNoCs	flit	C++	GPL 3.0	2022
Garnet	flit	C++	Berkeley-alike	2023
INRFlow	Traffic flow	C	GPL 2.0	2016
Noxim	flit	C++	GPL 2.0	2024
ROSS/CODES	flit	C	Open Source	2022
SST/Macro	Traffic flow	C++	privative	2023
SuperSim	phit	C++	Apache 2.0	2022

CODES (*Co-Design of Multi-layer Exascale Storage Architecture*) [36] is a flit-level simulation framework for the torus and Dragonfly topologies. CODES uses ROSS (*Rensselaer Optimistic Simulation System*) [37] as its discrete event simulator. In contrast to the other simulators, it can run in parallel with multiple CPU cores working together, providing greater execution flexibility and lower execution time.

HNoCs [38] and Noxim [39] also present flit-level accuracy but focus on the analysis of Networks-on-Chip. Noxim can model wireless communications between the network nodes, and HNoCs can model heterogeneous networks. Both tools are based on existing simulation frameworks written in C++: HNoCs use OMNET++, whereas Noxim leverages SystemC.

Table 1 compares these simulators, including our new entrant, CAMINOS. The table presents the detail level of the simulator, the programming language used in its development, the type of software license, and the date of the latest *commit*.

As mentioned at the beginning of this section, other simulation tools exist written in the Rust language, including network simulators. However, those tools focus on different aspects of the system, such as the impact of network protocols [40–43] or to simulate a network for testing network-oriented code [44] and a discrete event simulation framework meant to be built on top of MPI [45].

As far as the authors are concerned, CAMINOS remains the first network simulator written in Rust that focuses on the router microarchitecture and provides phit-level accuracy.

3. Simulator architecture

CAMINOS is a phit-accurate simulator, which decomposes packets between servers into physical digits or *phits*, representing the smallest data unit that can move in a single cycle. Traditionally, a phit was identified with the number of bit lanes in a network cable. Nowadays, they are more aptly associated with the width of the Serializer/Deserializer (SerDes). Phit accuracy is achieved by modeling the traversal of phits through the router components (buffers, crossbar) and links.

By using phits and cycles to model the router behavior, the simulator allows a layer of abstraction from actual hardware that can be easily translated into units of time (e.g., seconds) and bandwidth rates (e.g., bits per second) by considering the router clock frequency and link bitrate. All statistics expressed in cycles can be translated into seconds by dividing the number of cycles by the router clock frequency. Similarly, the throughput rates expressed in phits per cycle can be translated into bits per second by establishing the size of a phit in bits, considering that a phit needs to traverse through the link in a single cycle of the model. This approach is usual in network simulators that model the router microarchitecture, such as those described in Section 2.

This section describes the different classes of components modeled by CAMINOS and how they can be combined.

[2] The current requirements for the rustc compiler can be found at https://doc.rust-lang.org/nightly/rustc/platform-support.html.

3.1. Events and time

At its core, CAMINOS operates with an event queue that encapsulates scheduled occurrences at specific times. The smallest unit of time represented in CAMINOS is called a cycle. When all components operate at the same frequency, this cycle should be the period of operation. Generally, each component can have a period that is a multiple of the cycle. Each event is scheduled to a cycle and denoted as occurring either at the start or end of the cycle. This is a simplification of the delta delay used in VHDL and more general time models such as epsilon delay and superdense time; see [46]. In our case, this removes the ambiguity when some data is inserted into a queue, and the head of the queue is inspected in the same cycle. We schedule all insertions at the start of that cycle, during which other events can inspect them. Since the required event engine is relatively simple, we have implemented our own event engine to have finer control instead of using an existing framework.

The main events are the movement of phits and the processing of a network component. Additional events can be implemented as needed by providing a procedure to be executed when the event is processed. As notable example, a router can be modeled as a monolithic component, generating a single event for each router. A more elaborated router model would include events for different subcomponents. Such design reduces unnecessary computation by avoiding the reevaluation of a component when its state has not changed. This is helpful when the queues are empty or full, as they may cause deterministic stalls. Moreover, having subcomponents with independent events allows one to run each subcomponent at a different frequency, enabling an internal crossbar speed-up. Currently, two routers are available, a basic monolithic router and another with the described finer granularity; both are described in Section 3.4.

3.2. Traffic

During a simulation, servers generate their traffic in the form of messages. These messages are segmented into packets as they enter the network, following an MTU (Maximum Transmission Unit) equivalent. These packets are further divided into physical digits or *phits*, that require several simulation cycles to traverse from one network element to the next. Each packet can consist of one or more flow-management units known as *flits*, which can be stored separately in each router, depending on the flow control mechanism employed by the routers. CAMINOS considers a unit of 1 flit to refer the amount of data space required in the next buffer to allow its traversal; if Virtual Cut-Through (VCT) switching is used, this entails a packet size of 1 flit. CAMINOS assumes that the packet header can be fully stored in the first phit of the packet and therefore, under VCT switching, allows the packet to be retransmitted as soon as its first phit has been received, even when all the phits in the packet correspond to the same single flit.

A traffic component in the simulator describes the message generation process. These components allow their composition to simulate applications in sequence or parallel. The potential of composition is elaborated in Section 3.5. Messages can be generated in the network servers by following a Bernoulli process, with a given traffic load determining the probability of new message generation. Whenever a message generation is attempted, the available space at the server generation queue is checked; if available space is insufficient to fully host the message, the generation attempt is aborted and may be later reattempted. The number of instances in which this occurs, also named *missed generation*, is tracked through a functional metric later described in Section 4.2.

The traffic is also responsible for the size of messages, making it possible to mix messages of different sizes. There are more possible traffic types than generating load continuously. It is possible to provide each process with an initial load and wait for it to be consumed. It is also possible to define how new messages are generated in response to received messages. This makes it possible to implement traffic such as the wavefront found in the LU solver from the NAS Parallel Benchmarks. In this kernel, a corner of a mesh starts sending a message to its neighbors, and when a node receives the messages from its preceding neighbors, it generates new messages for its other neighbors. There are also functions to help place multiple applications across the topology and combine different traffic in multiple ways.

Traffic patterns in CAMINOS are built upon a more general concept of *pattern* as an algorithm that maps elements from one set into another, defining the possible destinations for any given source in a pattern. Significantly, these patterns are not limited to pure mathematical functions and may employ state or use a random number generator. This design allows for the composition of patterns, for instance, by building a global pattern over groups of varying sizes distinct from the total number of servers. The sources and destinations in the final composition are then mapped to the servers in the network. Within this context, a traffic pattern is a pattern over the network servers. For example, different permutations are patterns, such as transpose, cyclic or random permutation, or random involution. This permutation can even be loaded from a file instead of generating it on the fly. Other patterns include uniform traffic, where the destination changes per packet, or hotspot patterns, where some destination has multiple sources. Patterns can be composed to make new patterns, which is elaborated in Section 3.5.

Patterns can be used to define the destination of messages and to help define other mappings. They are also useful for managing the placement of multiple applications in the network.

3.3. Topologies

The *topology* of a network defines whether each switch port is connected, to which server or switch port it is connected, which is the port in the other endpoint, and a class for that link. We call *radix* of a switch to the number of its ports and *degree* to the number of other switches to which it is connected. These connections between switches make a graph in which the vertices are the switches, and the edges are the links. Thus, we may talk indistinctly about topology or the graph of switches for those matters where the servers, port indices, and classes are irrelevant. Topologies may be irregular, allowing for switches of different degrees, radix, non-connected ports, or connected servers. In particular, CAMINOS allows for irregular graphs, such as meshes, and indirect networks, such as Fat Trees. Links can be associated with a class, giving them special latency and frequency, for example, when including electrical and optical wires. Moreover, the link class is also employed in some algorithms. For instance, for the Dragonfly topology [47], CAMINOS defines three classes: one for links to servers, another for switches within the same group, and the last one connecting switches in different groups. Then, a weighted minimal routing can prefer to use a local-global-local route over a possible global-global one.

At present, CAMINOS offers a variety of topologies, including meshes, tori, and low-diameter topologies such as Dragonflies [47], projective networks [48] and Slim Flies [49]. In particular, the Dragonfly topology is presented with multiple aspect ratios and arrangements of global links. Among the available multi-stage network topologies are the eXtended Generalized Fat Tree (XGFT) [50], the Orthogonal Fat Tree (OFT) [51], Random Folded Clos (RFC) [52], and the flexibility to compose custom stages. Furthermore, CAMINOS incorporates the Megafly [53] (also known as Dragonfly+ [54]), featuring a diameter-3 indirect network of two levels, sharing similarities with both the Dragonfly and the Fat Tree.

In this paper we focus in another topology supported by CAMINOS, the Hamming Graph, which we employ in our evaluations in subsequent sections. In a Hamming Graph, which is the Cartesian product of complete graphs, nodes are organized in a multi-dimensional lattice where all adjacent nodes differ in a single coordinate, pertaining to a different position in one of the dimensions. This topology is also known as Generalized Hypercube [55], Flattened Butterfly [56] (FB) or HyperX [57]. The FB is achieved by "flattening" a regular Butterfly topology by coalescing routers from all the stages of the network into a single router, which results in a direct topology with all-to-all connectivity between the routers in each dimension. It can thus be argued that the FB topology is a particular case of a Hamming Graph where the number of endpoints per router equals the number of routers per dimension.

In addition to these families with particular definitions, CAMINOS provides for more general ones, such as random regular networks [58]. Moreover, CAMINOS allows you to read the topology from a file and export any created topology into a file. For example, the following command would export a 3D HyperX.

```
caminos --special=export --special_args='Export{
        topology:Hamming{sides:[4,4,4],servers_per_router:4},
        seed:5,
        filename:"hyperx4x4x4"}'
```

Working over arbitrary topologies gives patent a certain degree of abstraction that eases the implementation of new topologies. The construction of the topology is automatically verified, indicating the kind of problem, if any.

3.4. Router models

A router has the task of moving the incoming packets from its input ports to its output ports. It has to enforce a flow control, that is, to ensure the receiver has space for a flit before sending data. CAMINOS employs a credit counter on the emitter as the primary mechanism, tracking the available space on the receiver. The receiver sends notifications upstream when it gains more space. Upon reception, the sender updates its associated counter. The router also needs to include mechanisms such as routing and management of virtual channels (VCs).

Two router implementations are available in CAMINOS, called `Basic` and `InputOutput`. The `Basic` router is intended to be more simplistic, although they share many features. Both routers have multiple buffers per port, specifically one input and output buffer for each VC. The depth of these buffers can be controlled. In the `Basic` router, setting the size of the output buffers to 0 removes them, connecting the crossbar directly to the output ports. Traversing the `Basic` router does not add latency; if it is otherwise empty, a packet that enters an input queue at a cycle may go across the router and be scheduled at the next router with the link delay. The `InputOutput` router may set the crossbar delay and frequency. In the `InputOutput`, the allocator can be chosen between different implementations, while the `Basic` is fixed to a pure random allocation.

CAMINOS allows three different types of speed-up: internal, input, and output speed-up. Internal speed-up (only for `InputOutput` router) is achieved by forwarding data through the crossbar faster than through the links, hence requiring output buffers to decouple transmission rates. Input speed-up corresponds to having a greater number of inputs to the crossbar than the number of input ports to the router; in both routers of CAMINOS, the number of input ports to the crossbar equals the total number of input buffers, which is the product of the number of physical ports and the number of VCs in each port. Output speed-up is equivalent to input speed-up but is applied to the output ports of the crossbar and router.

In CAMINOS, a routing algorithm is characterized mainly by the function that provides valid output ports for a packet in an input queue. More specifically, given the current router, the destination, the topology, and the information potentially stored in the packet, the routing must build a set of candidate output ports. Each of these candidates is described by its output port and its VC, and it may optionally include indications for prioritizations or estimations. The routing specifies the routing information stored per packet. In particular, we aim to minimize costs for those routings that do not require extra information. Some examples of basic routing blocks supported by CAMINOS are minimal routing, Dimension-Order Routing (DOR), OmniWAR [25], and Up/Down. Nevertheless, nesting is allowed for composition. For example, when using Valiant routing [59], the information used for the subroutes is stored inside the information used for the Valiant routing. Some adaptive routing mechanisms such as UGAL can be expressed as the composition of other routing blocks.

Finally, many strategies exist to manage the valid candidates the routing indicates. A VC policy is a scheme that may combine the routing information with router information. A straightforward policy prevents deadlock by restricting the selected VC to a hop count or ladder [60,61]. Other policies may choose the minimum function of queue occupancy and route priority. For example, in UGAL routing [62], a packet must choose at source between using a minimal or Valiant route. As the average Valiant route has a base network latency twice that of a minimal route, it may select the shortest queue with the occupancy of non-minimal ones multiplied by a factor of 2. There are many possible variations: the length estimation could be used, the occupancy may include the output buffer of the current router and/or the credits of the next router, etc.

3.5. Component encapsulation and composition

CAMINOS is designed as a library, providing clear APIs for each component. Its design favors component composition, so many mechanisms can actually be defined at configuration.

Within the role of a library, CAMINOS allows a small project to add a new component without modifying the source code of CAMINOS. Such project would consist of an import declaration, the code of new components, and a call to the entry point function from the library. These steps only require the following lines of code.

```
// Include the library of CAMINOS.
use caminos_lib::{self,Plugs};
// Create an empty collection of extra components.
let mut plugs = Plugs::default();
// Insert the locally defined component.
plugs.add_topology("MyLocalTopology".to_string(),
        |arg|Box::new(topology::MyLocalTopology::new(arg)));
// Call the simulator.
caminos_lib::terminal_main_normal_opts(&args,&plugs,
    option_matches);
```

A project following this workflow would consist exclusively of code for the new components being developed instead of being a clone of the whole simulator. This also allows us to easily update the employed version of `caminos-lib` without significant revisions to the new components. This differs from many simulators, such as BookSim and FSIN, where several files must be modified to indicate new components. It also deviates from SuperSim, which via *Factories* avoids the need for that micro-management of files but still requires working over the whole source tree.

This approach is not without limitations. If a user has a component with complex connections that do not work within our defined API, it is unavoidable to clone the library to make the required changes. Additionally, the API is yet subject to changes when new components need to be added, which means the easy upgrade of the version of `caminos-lib` could require a little refactorization when it happens. A further possibility would be to provide a way to dynamically load components at execution time, although the usability of this approach is yet unclear.

A Rust trait defines the API for each component. This encapsulation is extended to the configuration file, which has the syntactical structure of a tree. Each component is built from a subtree, effectively making it independent from the other components. This encapsulation also facilitates the use of composition, which is applied extensively, allowing the definition of many novel components at the configuration level.

A helpful abstraction across several components is the interpretation of a set as a block with Cartesian coordinates. A region of size n with factorization $n = n_1 n_2 \cdots n_d$ can be interpreted as a block of d dimensions with side n_i for the dimension i. Then, any integer m, with $0 \le m < n$ can be converted into coordinates for such block by finding d integers m_i such that $0 \le m_i < n_i$ and $m = \sum_{i=1}^{n} m_i \prod_{k=1}^{i-1} n_i$. This is a natural representation for some topologies, such as the mesh and the Hamming

Graph, and can be usefully exploited in their routing algorithms. Another notable application is for the definition of traffic patterns. Many sensible patterns interpret the set of nodes as a Cartesian block and then shift or permute coordinates to find the destination. This alone allows defining a transpose pattern at configuration that is not required to be a power of 2 elements. Furthermore, the set of transposed nodes can be easily established to be the set of switches, servers, or others.

3.5.1. *Examples of composition*

We now present a non-exhaustive list of composition examples to illustrate the potential of this approach. Some are instances of a final mechanism with another component as a parameter; others are fundamental bricks to build others.

Valiant's randomization scheme is a prominent parametrized routing that can be used with various underlying routing schemes. For instance, the segment from injection to the intermediate switch can be routed using DOR, while the intermediate switch to the destination can be routed using fully adaptive minimal routing.

The most natural pattern composition is the product of two patterns: uniform destinations inside groups and some permutations among groups. A group may be the servers attached to a router, the routers in a Dragonfly group, or a row in a Cartesian-based topology.

An example of traffic composition is to build a sequence of other traffic, each with a given duration. A similar scenario is to make a sequence, where every traffic entry has an intrinsic finalization criteria, such as sending a given number of messages. In this case, each traffic in the sequence starts when the previous one has ended.

Some operations on routing candidates are defined as the composition of VC-management policies. For instance, some thresholds of UGAL-like mechanisms are set for packets excluding a defined escape channel.

As a simple operation for a topology, it is possible to relabel the servers according to a given pattern.

4. Usability

The usability of CAMINOS is built around the concept of experiment. An *experiment* is a collection of cohesive simulation instances that aim to answer some question. Those simulations will vary in a few parameters and their results can be reasonably represented in a figure. In CAMINOS, an experiment is represented as a directory with several files. The definition of the simulations belonging to that experiment is in the `main.cfg` file at the experiment's root. To comply with the encapsulation, this configuration is syntactically a tree, following a syntax designed for it. CAMINOS also allows executions without an entire experiment folder, just a configuration file. This mode is intended for quick simulations at an interactive shell and allows for command-line overrides of the configuration.

The working methodology with CAMINOS is to create a single folder for the whole experiment, with one file defining the simulations' parameters and a separate file describing the output to be generated, including the plots. This allows a simplified behavior where a single command can run the experiment and produce PDF files containing the desired plots and any other preferred output, such as a CSV (Comma-Separated Values) file. This methodology is appropriate for local runs and debugging since it aggregates all the tool's functionality and simplifies the user's process. For experiments run in computing clusters, the behavior is split into two phases: one to launch the experiments as jobs through a workload manager and another to aggregate the results and generate plots. This allows the division of the experiment into several steps, aggregates results from different partial executions of the same experiment, and decouples the output generation from the model execution to account for execution runtime.

4.1. *Configuration syntax*

In CAMINOS, experiments are declared in a domain-specific language employing a syntax similar to JSON but with notable enhancements. The same syntax is also used to describe outputs and some internal results.

Most syntax elements are like JSON, including double-quoted strings and bracketed lists. Dictionaries are tagged: writing `routing:DOR { order:[0,1] }` indicates the field `routing` of the parent is a `DOR` object, which has key/value pairs. This is different than JSON, where the `DOR` tag would be indicated in some other way, like `routing:{type: DOR, order:[0,1]}` or `routing:{tag:DOR,data:{order:[0,1]}}`. Most significantly, using a hierarchical syntax instead of a plain configuration allows for multiple uses of the same object with different parameters. For example, `Valiant{first:DOR{order:[0,1]}, second:DOR{order:[1,0]}}` expresses a randomized routing where the segment from the source to a random switch is made in dimension order, and the routing from the intermediate to the destination uses the reverse order.

The syntax further differs from JSON by including the elements `![a,b,c]` and `name![a,b,c]` which determine the extent of the set of experiments. Specifically, a tree containing an element `![a,b,c]` is expanded into three trees, each containing one of the variants. This simplifies the launch of experiments with sweeps in one or more parameters, as is common in evaluating new proposals and mechanisms.

For example, an experiment with `load:![0.25,0.5,0.75,1.0]` will expand into four trees containing respectively `load:0.25`, `load:0.50`, `load:0.75`, and `load:1.0`. When there are multiple elements, the resulting experiment has the Cartesian product of all combinations as its set of simulations. The named variant of this syntax allows for an expansion that must match between them; this is, all expansions with the same non-empty name must have the same number of elements. For instance, a tree containing both `size![1,10]` and `size!["small","large"]` will expand into two trees, the first one with `1` and `"small"`. The syntax also includes support for evaluating functions. This describes output processing to generate readily usable data (such as a graphic) and is described in Section 4.3. This repeating operator provides an easily manageable way to express large experiments. To illustrate this, note, for example, the case of SuperSim, where a battery of experiments is represented by a JSON file plus a Python script. In other simulators, such as BookSim, these batches are not supported, predisposing its users to make their own scripts, leading to unnecessary repeated efforts.

4.2. *Experiment metrics and statistics*

A simulation in CAMINOS can be divided into a warmup phase and a measured phase, with durations specified in the configuration file. Some traffic may finish before the specified duration, ending the simulation. In classical experiments, the warmup phase should be long enough for the network to reach a steady state. It should be noted that it is not always possible to achieve such a state, depending on the properties of the network and the workload. Tracked statistics for each phase are reported separately, with the measurements from the warmup phase mainly used for diagnosis.

Metrics can be classified into two main categories: functional metrics related to the experiment and metrics that consider the resource usage from the simulator itself.

There are two main groups of functional metrics, focusing on the amount of data per time unit and the latency. In the first group, we can find the *throughput* or *accepted load*, which is the rate at which packets reach their destination, and the *injected load*, which is the average rate at which packets are introduced into the network. Under stable conditions, the values for these two metrics converge as simulation time increases.

Latency can be measured through several different metrics, such as the *network latency*, which is the time since a packet enters the network until it is consumed at its destination server. Additionally, we can track the total message delay, the number of network cycles since the whole message was created until it is consumed at its destination. This includes all time spent on the server generation queues plus the link delay to the first router. It can be helpful to track other latency metrics under particular traffic loads or to understand pathological behavior. For instance, CAMINOS reports the cycle in which the simulation is finished for traffic that finishes after completing a load. Other metrics of interest in this traffic that can be tracked are the cycle in which the last message was consumed and the cycle in which the last phit was created, providing more information on the relative termination of the servers.

Under specific scenarios, traffic behavior between servers can be asymmetrical, which makes it interesting to check network fairness. Among the aggregated reported metrics in CAMINOS is the Jain index [63], a balance between the load of individual servers, either generated or consumed.

There are also metrics to assess the validity of a network model and its proper operation. The maximum link utilization reports the maximum rate of utilization across the links. If it does not reach a value of 1, every link in the network has been unused at some point in the simulation. The average number of missed generations in the servers and the number of servers with missed generations track the missed generations. In these instances, the traffic requested to generate a message at a server at some cycle, but the server generation queue did not have enough space to host the whole message. Depending on the traffic, the message may be generated later or skipped. Values greater than 0 may correspond with communications with infinite latency for the typical traffic with Bernoulli-distributed generation.

As information on the routes taken, CAMINOS provides the average number of hops per packet. This information is also tracked through a list with the count of packets of each route length, allowing the generation of path length histograms. There is also a metric with the average link utilization, which measures the average number of *phits* per cycle that traverse the links; for minimal routes, this metric should track the accepted load, and a deviation from that value may correspond to an inefficient use of the network resources. Such information, complemented with the accepted load and the average path length, allows us to assess the behavior of the network accurately. Link usage can also be analyzed per *virtual channels*, which may provide insight into the routes or other aspects, depending on how virtual channels are managed.

Other reported measures are related to the performance of CAMINOS, such as its memory footprint or execution time. The reported memory footprint is the process's maximum resident size obtained from `VmHWM` (high watermark in KB). The reported times are the user time and the system time in seconds. CAMINOS spends most of its time in user mode, so any high system time represents more of a problem in the node running CAMINOS than in the simulation.

4.2.1. Optional statistics

By default, CAMINOS only reports the aggregate measurements of the functional metrics over the whole measured phase. CAMINOS supports statistics across different regions of time, servers, or packets, but they require opt-in since they may negatively affect the simulation's performance.

It is possible to query for values at different percentiles to complement the default average values for given metrics. For example, for accepted load and percentile 75%, you get the value of accepted load such that 75% of servers have accepted less load than that value. This has little extra cost since servers already have to store some information. Similarly, percentile statistics can be gathered per packet; however, this is notably more expensive, as data are stored for each packet generated during the measuring phase.

CAMINOS supports a powerful option to add aggregated statistics per packet that are a function of certain packet properties. For example, it can be used to track the average delay per packet for each possible combination of hops given and the last VC used. This can be useful to identify particular behavior or network properties, particularly when evaluating new proposals. Employing this option impacts execution time, which should be considered.

4.3. Working methodology

For most simulators, the tools for launching simulations and processing results appear as satellites growing around them. These tools are also usually built-in scripting languages such as Python. We consider that a mistake, even if for some small experiments that may get you running, the final tools can have a size comparable to the simulator itself. For CAMINOS, we have opted to integrate these tools in the simulator. This makes the work methodology more uniform, with all actions operating over the same objects. Specifically, we define an experiment as a folder that includes some files. The file of major importance is the configuration file, describing all simulations that comprise the experiment. Each simulator action is executed against this folder with the proper `--action=` flag. In addition to the configuration file, the experiment folder contains a `journal` file, a description of outputs (`main.od`), a description of a `remote` host, obtained results (in `binary.results` or within `runs/`), and the generated outputs (within `outputs`). The journal file tracks the history of the experiment, with entries added whenever any action starts or finishes, plus the possibility of additional messages.

CAMINOS can manage executions on a SLURM system. For this purpose, it invokes externally the commands from the workload manager. For example, the action `slurm` enqueues the selected pending simulations into the SLURM queue system by calling the system's `sbatch`. Jobs can be launched or canceled, and their status can be checked directly from CAMINOS.

CAMINOS only executes simulations for which it does not have results already. If some results are known to be incorrect, they may be discarded with the `discard` action. Selective control for most actions is possible with a series of flags, restricting to a range or a set of conditions through a provided expression; for example, this allows separate simulations with the highest load to be run. There is support for merging experiments: if an experiment is defined as containing simulations already executed as part of other experiments, they can be imported using a flag. An empty copy of an available experiment is also possible, which helps make a new experiment with only a few differences from a previous one.

CAMINOS also helps manage remote copies of experiments by the actions `push` and `pull`, among others. These actions establish an SSH connection by requesting a password or using the user's public key file. The `remote` file manages the host to which these actions connect.

4.3.1. Generation of outputs

CAMINOS can generate CSV files, which can be helpful when interfacing with external tools. However, CAMINOS also seamlessly generates outputs from currently available results, such as direct plots, which are more common for ordinary usage.

CAMINOS allows seamlessly generating outputs from currently available results, e.g., by calling the system's `pdflatex` to generate PDFs. These outputs can be considered the end product of the simulation tool, presenting the user with more easily parsable data. In the case of generated TeX/PDF, the fraction of completed simulations is indicated, together with a lot of other diagnostic information that helps to replicate results in later experiments. For example, it includes all versions of CAMINOS that have been involved in the simulations. The outputs to be generated are described in the `main.od` file using the same syntax employed to define the experiments, including the option to evaluate expressions. This allows users to create their definitions of outputs to be generated, particularizing the information for their use cases of interest. By default, outputs include the value from the configuration or the results, but they can be renamed to make the output more easily integrated into documents.

It is more common to generate plots directly for ordinary usage. CAMINOS supports generating pgfplot-backed figures on LaTeX code, and PDFs are generated from them. These LaTeX fragments can be incorporated into other documents, providing some methods to ease tuning the presentation when working directly on such a document.

This infrastructure of CAMINOS can also be used without an actual CAMINOS experiment, beginning with a foreign CSV. It is thus possible to have a CSV file obtained with a different simulator, declare what plots to generate, and invoke CAMINOS with the `--foreign` and `--use_csv` flags to generate them.

All the forthcoming figures in Sections 5 and 6 are obtained with this methodology. For simulations with BookSim, their results have been exported to CSV and managed with the `--use_csv` flag. The LaTeX code of the figures was later retouched for a cleaner presentation.

5. Validation

To assess CAMINOS' accuracy, we selected a well-known scenario. We evaluated it in CAMINOS against BookSim, which was described in the state-of-the-art analysis performed in Section 2. We selected BookSim for its widespread adoption in the scientific community, as evidenced by the following works [49,64,33,65,66].

We initially intended to include SuperSim in this validation. However, preliminary simulations showed senseless results, such as negative delays. Since the source of the issues could not be traced back conclusively, we discarded it from the comparison.

The section first presents the methodological approach to this validation process, including a description of the network scenario considered. Next, a set of simulation results is presented and contextualized to understand the accuracy of CAMINOS and compare its output to that of BookSim for the same network scenario.

5.1. Validation methodology

This subsection provides an in-depth explanation of the methodological approach followed to assess CAMINOS's accuracy by analyzing its output for a given scenario. The main purpose of this process is to show that results from CAMINOS are consistent with analyses in previous works and with an outcome that does not diverge significantly from BookSim. To this extent, the scenario must be similar in all cases, with a target hindered by the limitations presented by the syntax and model restrictions in BookSim.

As described in Section 3.2, CAMINOS considers 1 flit as the amount of data that needs to be able to be hosted in a neighbor buffer to allow its forwarding and, therefore, has a packet size of 1 flit when Virtual Cut-Through switching is employed. BookSim employs the same approach but is only capable of *flit*-level simulations, transmitting the whole *flit* in a single network cycle; therefore, in the case of Virtual Cut-Through switching, the whole packet is transmitted in the same network cycle. In CAMINOS, non-trivial packet sizes with VCT switching can be modeled by setting a flit size of several phits, requiring several simulation cycles to traverse from one network element to the next, and allowing the packet to traverse to the next network element after receiving the first phit. To harmonize the scenarios given to each simulator, we considered only communications where both packets and flits are restricted to a size of 1 phit. Notice that, for this scenario, Virtual Cut-Through switching coincides with Store-and-Forward switching. Moreover, BookSim does not allow the generation of messages composed of multiple packets sharing the same source and destination. Accordingly, in CAMINOS, the size of messages has been set to one packet.

5.1.1. Network scenario

Table 2 presents the scenario employed in validating CAMINOS. A network following a *regular* HyperX topology [57] has been simulated since the topology is modeled in the two simulators. A regular HyperX network follows a multi-dimensional structure where the routers in each dimension present all-to-all connectivity and are directly connected to a given number of endpoints, as discussed in Section 3.3. This network is defined by the number of endpoints per router (T), the *network side* or number of routers per dimension (S), and the number of dimensions (L).

Table 2
Parameters of the network scenario fed to the simulation tools to perform the validation process.

Parameter		Value	Comment
Topology		HyperX	The topology is named `flatfly_onchip` in BookSim and `Hamming` in CAMINOS, but they are equivalent for the values employed.
Network side	S	16	Number of routers per network dimension.
Dimension	L	2	Number of network dimensions.
Concentration	T	16	Number of endpoints connected to each router.
Radix	r	46	Total number of input/output ports per router: $r = T + (S-1) \cdot L$.
Endpoints		4096	Total number of servers/endpoints in the network.
Router architecture		Input-Output Queueing (IOQ)	Router with buffers at ingress and egress points.
Virtual channels (VC)		8	Number of virtual channels (buffers) per physical channel (link).
Input (VC) buffer size		8	Buffer size for each virtual channel at the router input ports, expressed in flits.
Output (VC) buffer size		4	Buffer size for each virtual channel at the router output ports, expressed in flits.
Message/packet size		1	Size of the message/packet size, expressed in flits.
Routing algorithm		XY-YX	DOR-based routing.
Traffic pattern		UN, RandPerm	
Offered load		0.1 to 1 with 0.1 steps	Measured in phits/cycle/endpoint.
Simulation extent		23000 cycles	Split into warmup (20000 cycles) and measured (3000 cycles) phases.
Crossbar delay		2 cycles	
Link delay		1 cycle	All network links have the same delay.

CAMINOS models HyperX networks where all dimensions can have different sizes, and those sizes can be any arbitrary value, as well as different from the number of servers per router. However, BookSim only allows modeling regular 2D HyperX networks where S is the same for every dimension and is restricted to a square integer. Moreover, the number of servers per router, T, needs to be equal to S. Fig. 1 displays a sample diagram for a 2-dimensional HyperX with four endpoints per router and four routers per dimension.

The HyperX network provides good topological characteristics, such as performance scaling with the number of endpoints and a small diameter that allows for short, minimal paths. The HyperX topology receives different names in each tool, and particular parameters differ slightly. In CAMINOS, this topology is known as the Hamming Graph, the Cartesian product of complete graphs, which is the mathematical definition of the HyperX topology. Instead, BookSim refers to this topology as the Flattened Butterfly.

A network with a side length of $S = 16$ has been established, resulting in a non-trivial network size of 256 network routers and 4096 end nodes. This size is large enough to be representative of behaviors observed in larger networks. It ensures that we can evaluate network behavior without encountering pathological artifacts that may appear in smaller networks, which may not scale effectively and are, therefore, less representative for evaluation using the simulator.

The duration of the warmup phase has been chosen to reach a stable state. Booksim employs a convergence criterion to finish the measured phase, and we use the same duration for the simulation with CAMINOS.

The router microarchitecture follows an Input-Output Queuing approach, where buffers are located at the router's ingress and egress.

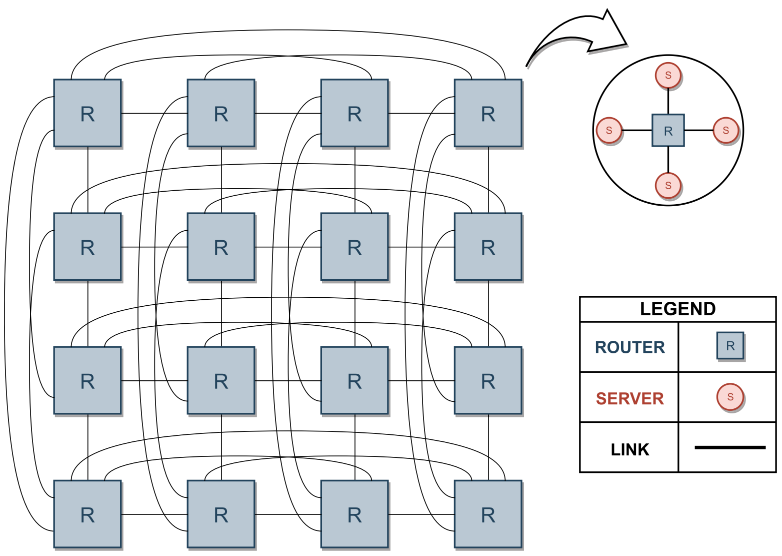

Fig. 1. Diagram of a 2-dimensional HyperX network with side $S = 4$ and concentration $T = 4$.

```
routing: Sum {
    policy: Random, // randomly select XY or YX routing
    first_routing: DOR {order: [0,1]}, // XY routing
    second_routing: DOR {order: [1,0]}, // YX routing
    first_allowed_virtual_channels: [0,1,2,3], // first 4 VCs
    second_allowed_virtual_channels: [4,5,6,7] // last 4 VCs
}
```

Listing 1: Definition with the syntax of CAMINOS of the XY-YX routing from BookSim.

Buffering at the ingress ensures storage if the crossbar resources cannot be immediately allocated. Buffers at the egress decouple router speed-up from the link bandwidth. Routers have an input and output speed-up factor of 8, as they can concurrently route data between each pair of input/output buffers (*virtual channels*, VCs), as defined in Section 3.4. However, the link connection limits the actual data transmission speed from the router. Router resources are allocated using the *iSLIP* algorithm, as described in [67], with 1 iteration to maintain consistency across simulators.

Packets are routed using a variant of DOR, a deterministic method that ensures the absence of network deadlock. In particular, we employ XY-YX routing, which has a specific implementation in BookSim and is supported in CAMINOS. It should be noted that BookSim has a `ran_min_flatfly` routing function. This function allegedly uses a random minimal route but never uses a random number, actually implementing DOR. To avoid confusion, we employ instead BookSim's `xy_yx_flatfly` function that randomly uses either XY or YX order, where nomenclature and code agree.

For coherence, this routing has been declared in CAMINOS and built upon the existing DOR. Remarkably, this routing configuration is entirely defined within the CAMINOS syntax without the need to code new functions. Listing 1 provides a detailed description of the routing approach employed in CAMINOS to implement DOR.

This routing does not cause deadlock, as each part uses a separate set of VCs. To mitigate head-of-line blocking, we employ a total of 8 VCs, which are separated into two blocks of 4 VCs, one for each DOR order. All four channels for each set are requested and resolved by the iSLIP allocator.

The analysis has considered two distinct traffic patterns. The first is Random Uniform Traffic (UN), which exhibits a homogeneous temporal and spatial distribution of communications between all network endpoints. Each endpoint has an equal probability of sending traffic to any other server in this pattern. The implementation of UN is identical in both simulators and can be considered benign for low-diameter networks.

The second pattern, RandPerm, involves monotonous flows of messages between the same pairs of source and destination endpoints across the whole simulation, following random permutations of endpoints established at the beginning of the simulation. Similar to the UN, this pattern demonstrates spatial homogeneity but models persistent connections without temporal variability throughout the simulation. The RandPerm pattern can induce more stress on the network depending on the specific flows generated by the permutations used. This increased stress can lead to the presence of *hotspots* where communications overlap, potentially affecting performance. We perform simulations with nine different permutations in both simulators to remove the bias for any particular permutation. Empirical results indicate little difference across the permutations, with a maximum deviation in throughput of under 4% for both simulators.

5.2. *Results for the network scenario*

This subsection describes the performance metrics utilized to assess the accuracy of CAMINOS in a scenario with known results beforehand. It presents the outcome of CAMINOS for that scenario and compares them to the results obtained from BookSim.

5.2.1. *Average throughput*

The throughput metric and average latency provide good evidence to detect network saturation or congestion. Fig. 2 presents the throughput values in CAMINOS and BookSim for the two traffic patterns described in Section 5.1.1, as a function of the traffic load offered to the network.

The graph displays nearly identical behavior for both simulators for the Uniform pattern and approximately the same for RandPerm. RandPerm presents a saturation point around 0.4 phits/node/cycle (40% of the maximum achievable), and UN can reach a significantly higher point of 0.9 phits/node/cycle. Apart from being the same in both simulators, these results adjust to a theoretical analysis of the scenario.

Theoretically, a perfectly uniform and homogeneous pattern of communications would allow every endpoint to inject and eject a phit in every cycle. However, the actual distribution of the communications in the UN subtly differs in several ways. Since the destinations are randomly independent, some servers are chosen more times than others across the simulation. This makes it impossible to reach the full rate, as those destinations would be asked to consume more load than possible. UN can also display heterogeneity for limited time periods, conducing

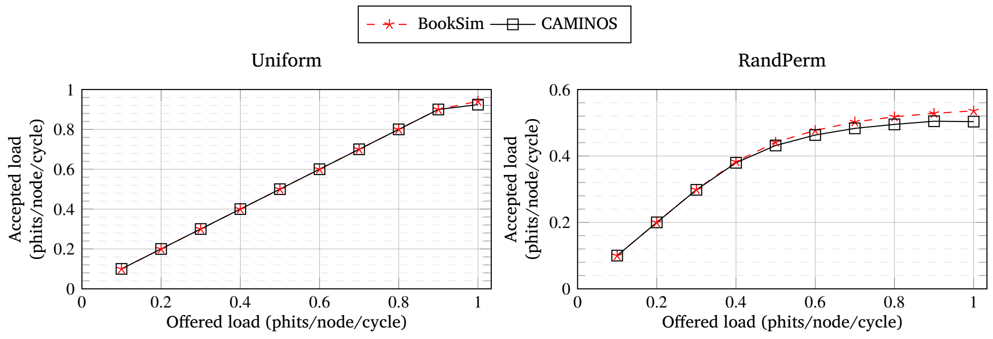

Fig. 2. Average throughput for the Random Uniform (left) and RandPerm traffic patterns (right).

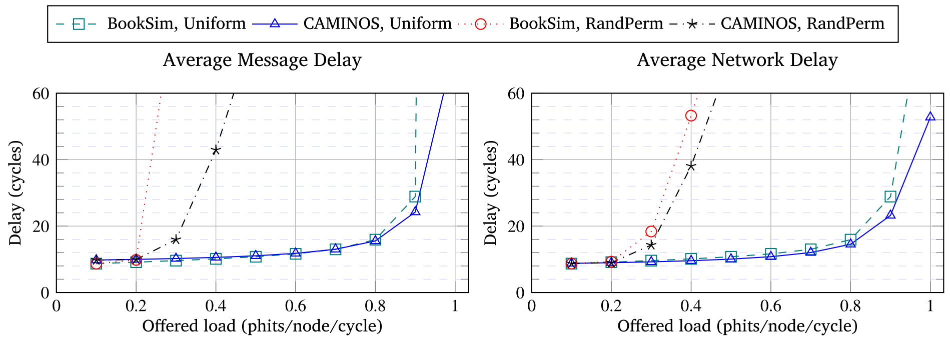

Fig. 3. Average latency, measured in cycles, as a function of offered traffic load and the traffic pattern. In the left the total message latency including time spent in server generation queues. In the right only the network latency.

to small areas of congestion (also called *hotspots*). Similarly, the pattern distribution cannot guarantee a perfect equilibrium that prevents collisions between resources (links, buffers, endpoints) and thus further reduces the maximum throughput that can be achieved.

On the other hand, RandPerm strongly depends on the distribution established at the beginning of the communications. The randomness is reflected on some links being part of the route of a higher number of source/destination pairs. Those links become bottlenecks and cause significant throughput degradation, but only affecting a subset of the endpoints. This conduct explains how the throughput achieved at the peak offered load is around 0.54 phits/node/cycle, notably above the saturation point. At peak load, there is a difference of about 0.04 phits/node/cycle between simulators, but they are nearly identical up to the saturation point.

The small differences between simulators in both traffic patterns come from a non-identical implementation of the same scenarios in each simulator, due to small architectural differences in the router components. For example, both simulators allow output speedup which, as described in Section 3.4, consists of a greater number of output ports in the crossbar than physical ports to the outside of the router. However, in CAMINOS this output speedup is modeled using one buffer per output port in the crossbar, and then arbitrating between all the buffer ports of the same physical output port. In BookSim there is a single output buffer in the physical port, but it can be filled by multiple output ports from the crossbar concurrently. Both approaches render a similar performance improvement, since they allow the crossbar to forward packets faster, but the approach in CAMINOS, which needs less complex logic, does not allow multiple packets targeting the same output port and Virtual Channel to advance at the same time.

5.2.2. *Average latency*

Latency measurements can be characterized for each size of information unit: phit, flit, packet, and message. As stated at the beginning of the section, we model a size of 1 phit for flit, packet, and message. Therefore, all information units have the same size, and any latency statistics are the same at each level. BookSim presents a model of the router microarchitecture with finer granularity, which introduces an increase in the number of cycles invested in traversing the router due to resource conflicts that reduce performance. However, a similar effect can be easily modeled in CAMINOS by establishing a 2-cycle delay to traverse the crossbar, as is done in all the simulations presented in the paper.

Latency results can be split into two components: injection latency, which establishes the time between a packet being generated and entering the network, and network latency, which measures the time between the instant the packet enters the network and the time it is delivered to its destination.

Fig. 3 shows the average latency, measuring the network component and the aggregated packet latency for each traffic pattern. Latency behaves differently before and after reaching the saturation point. Both simulators present similar behavior below saturation, with a steady increase in latency as the network load grows.

Above saturation, the latency rapidly increases due to injection latency; the buffer size limits this increase at the server generation buffers. In BookSim, these buffers are considered to have an infinite length. However, the injection of any packet behind the head of the buffer is not modeled until that packet is forwarded to a router output. This allows the simulation of server generation queues of unbounded size without requiring excessive memory resources and can lead to a rapid increase in packet latency for simulations beyond saturation. As described in Section 3.2, CAMINOS applies a different approach, limiting the size of the server generation buffer through an input parameter; when a traffic model attempts to generate a message, and it cannot be entirely stored in the queue at generation time, the message is discarded. Its generation is re-attempted at a later network cycle. This approach does not significantly affect the accuracy of the results with synthetic traffic loads, as can be observed. Modeling infinite queues may result in a correct open-loop simulation, as those discarded packets would be a case of the network influencing the traffic. Nevertheless, actual networks have buffers, and real applications frequently introduce waits. If messages accumulate, traffic adapts its shape (e.g., one application becomes slower than the other) or slows down.

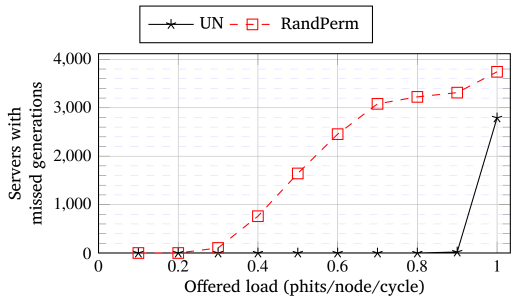

Fig. 4. Amount of severs which have failed to generate a message at some point due to limited space on the server generation queue.

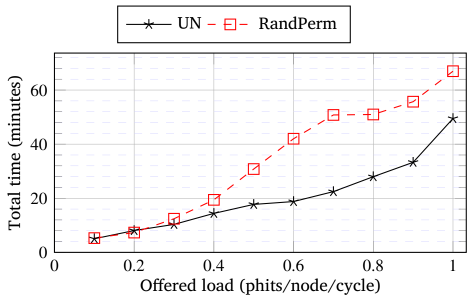

Fig. 5. Execution time as a function of offered network load and the traffic pattern.

To establish if such behavior is present, CAMINOS also outputs a metric of missed generations which, as described in Section 4.2, accounts for generated messages that could not be allocated enough buffer space to be hosted at the server generation queues of the server NIC and, therefore, had to be dismissed. If a network model renders a non-zero value for missed generations, this would imply an increase in the completion time of communications. The value of this metric is shown in Fig. 4. It can be observed that in RandPerm, some endpoints are already unable to complete the generation of all their messages for a load of 0.3, likely due to some network links being used by packets from a large number of sources. The saturation is already prominent for a load of 0.4, with 19% (760 from 4,096) of the servers being limited. The amount of capped communications grows progressively until affecting 91% of the servers at peak load. The impact of the difference in the modeling of the server generation queue is removed in the right part of Fig. 3, which only displays the average latency between the ingress and egress of the packets from the network. In this case, the outcome is similar for both simulators, and the increase after saturation has the same slope since network latency above saturation is limited by the size of the transit buffers in every router, which are finite in both simulators.

For the total message latency in BookSim, one cycle of delay from the server to the first switch has been added, as it is not included by default.

6. Performance of CAMINOS

This section describes the process followed to evaluate the performance of CAMINOS, including the performance metrics employed and the computational environment where the evaluation has been conducted.

6.1. *Evaluation methodology*

A set of metrics needs to be selected to analyze the performance of CAMINOS. All performance results have been achieved with the scenario from the simulator validation in Section 5.1.1 unless otherwise stated.

Three performance metrics have been selected for this evaluation: execution time, memory footprint, and scalability. Execution time measures the CPU time invested in running a single simulator execution for a given network scenario. It can be split into two components: system time and user time. System time details the amount of time invested in running OS tasks to complete the execution of the process (in this case, the simulator), and user time concerns only the execution of code in user mode. Memory footprint is measured through the *peak resident set size* metric, also known as the *high water mark*. Its value indicates the maximum amount of memory that a process has referenced. Scalability is defined as the ability to simulate a given network scenario when the network size is increased, leaving all other parameters unchanged. This last metric allows us to determine the maximum network size that can be evaluated with the simulator in a given set of resources, which would enable us to analyze better the impact of a given network feature in bigger networks.

To evaluate execution time and memory footprint, CAMINOS offers, by default, a set of performance measurements using the `procfs` crate, which reads directly from `/proc/`. These results are then presented as part of the default simulator output, allowing users to accurately allocate the resources needed if the execution is performed in a large system through a job scheduler.

All evaluations performed in this paper have been conducted on *Triton*, a small homogeneous computing cluster belonging to the Computer Architecture and Technology (ATC) group at the Universidad de Cantabria. This cluster consists of 5 identical computing nodes, each equipped with 2 Intel Xeon processor sockets and 32 GB of RAM, resulting in a total of 200 cores available in the cluster.

6.2. *Performance results*

This section analyses the performance results from CAMINOS for the network scenario described in Section 5.1.1; in the case of the scalability analysis, different values for the network size parameters have been considered to assess the tool's capability to handle a larger network size.

6.2.1. *Simulation time*

Simulation time refers to the duration of the execution of a single simulation. It strongly depends on several factors of the simulation being run: the complexity of the network scenario, the amount of network activity simulated, or the structure of the simulation. Since the number of messages traversing the network strongly correlates to the applied load, it is helpful to represent the simulation time as a function of the amount of traffic load. Section 6.1 described the two components of execution time for a program; however, the most important metric for the user is the invested time into completing a simulation. Since system time is three orders of magnitude below user time in all cases examined, this evaluation does not split the aggregated simulation time into user and system time.

Fig. 5 displays the execution time of CAMINOS under uniform (UN) and RandPerm traffic patterns, expressed in minutes. Results under UN traffic show a close to linear increase with the applied load for points below saturation and a rapid increase thereafter. Under RandPerm traffic, the increase is close to linear with the applied load until a value of 0.7 phits/node/cycle of applied load is reached. That point coincides with the small plateau for servers with missed generations seen in Fig. 4, showing a parallelism between simulation time and the number of source/destination pairs with a saturated path. This behavior is caused by a larger number of stalled packets at the head of the queues, which need to be checked to assess whether they can be forwarded. Maximum execution time sits around 80 minutes for RandPerm, with UN traffic at a lower maximum of 50 minutes. Since queues under UN traffic become less congested due to a higher saturation point, there are fewer packets stalled at the head of buffers; since traversing the crossbar and every element of the router microarchitecture consumes several network cycles, packets that are in motion represent a lower CPU usage than those that cannot advance, since their target output port needs to be

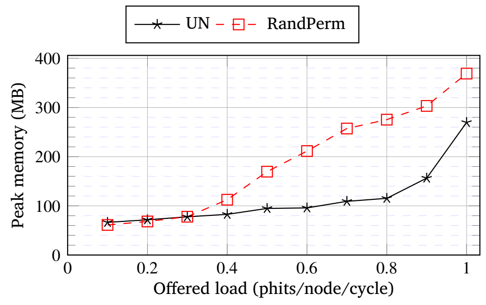

Fig. 6. Memory usage as a function of the offered network load and the traffic pattern.

recomputed every cycle. Note that this recomputation is not strictly necessary for deterministic routing; however, CAMINOS follows a common approach that accounts for non-deterministic routing, such as statistical and adaptive mechanisms. It must be noted that, for this network size, there is a significant divergence between minimum and maximum execution times for each traffic, which start at less than 10 minutes and present an average value of approximately 30 minutes across all network loads.

6.2.2. *Memory footprint*

Memory usage has been measured through the *memory peak* metric, which represents the highest use of memory during the execution of the simulator and establishes the minimal amount of memory required to run without halting execution. This metric is also known as the *high water mark* (VmHWM). Fig. 6 displays the amount of memory used by CAMINOS under UN and RandPerm traffic.

It can be observed that, for both traffic patterns, the peak of memory usage follows the same curve as the execution time, whose evolution was attributed to the number of packets in the modeling of the network. Since the lowest observed value, for a traffic load of 0.1 phits/node/cycle (10% of the maximum traffic load), sits around 50 MB, this implies an efficient use of resources: the size of the packet model relates to the amount of statistical information it provides, and therefore cannot be reduced without loss of insight, and a low base value proves an efficient model of the router architecture. Part of the efficient use of resources can be attributed to the use of the Rust programming language, which is supposed to avoid some of the costs of some abstractions that could appear in similar languages as C++.

More importantly, the analysis beyond the saturation point is interesting, but mainly for different scenarios. Throughput levels beyond saturation are mainly relevant to assess the lack of abnormal network behavior and performance losses from peak when the network is overly stressed. Evaluation of when the network links are saturated is also of interest when assessing the impact of congestion control mechanisms. A degradation of the simulator performance is expected when simulating scenarios with high congestion since there is a higher amount of packets and more contention for the network resources, leading to multiple attempts to forward the same packet. In general, low use of resources when the modeled network is not saturated is desirable and allows for more complex or detailed scenarios to be run for a given set of computing resources.

6.2.3. *Scalability analysis*

As described at the beginning of Section 6.1, the scalability of the tools is defined as the ability to perform a simulation of a given network scenario for increasing network sizes. Results show an analysis of CAMINOS for the two-dimensional HyperX network described in Section 5.1.1, where the network side ranges from 4 routers per dimension (64 servers in total) to 36 routers per dimension (46,656 servers in total in the range of modern HPC systems). Table 3 shows the different network sizes modeled.

Table 3
Size parameters of the HyperX network employed in the scalability evaluation.

Network side	Concentration	Endpoints
$S = 4$	$T = 4$	64
$S = 6$	$T = 6$	216
$S = 8$	$T = 8$	512
$S = 9$	$T = 9$	729
$S = 12$	$T = 12$	1,728
$S = 16$	$T = 16$	4,096
$S = 25$	$T = 25$	15,625
$S = 32$	$T = 32$	32,768
$S = 36$	$T = 36$	46,656

Figs. 7 and 8 show the total execution time and peak resident memory employed by CAMINOS under UN and RandPerm traffic for different network sizes. Each figure displays the average execution time and peak memory usage for a simulation of a given network size, as well as the minimum and maximum values. Minimum values represent the behavior when modeling low traffic loads, whereas maximum values correspond with a model of the network at peak traffic loads (1 phit/node/cycle). As described in Section 6.2.1, the performance presents two distinct behavior patterns corresponding to simulations where the load is below or above the saturation point. Minimum execution times are representative of the behavior observed below the saturation point, whereas maximum execution times correspond to scenarios above the saturation point.

It can also be observed that both metrics grow linearly with the number of endpoints. Each modeled network has a number of endpoints defined as the cube of the side of the topology since it is a 2D Hyper-X network with the same concentration factor as the network side. The number of server generation queues and generated packets scale with the number of endpoints in the network, which explains the trend. It can be observed that, for peak memory usage, the increase of average, minimum, and maximum values is a near-perfect straight line for over a thousand endpoints, further assuring efficient memory use in the tool. For very small sizes, the curve gets flattened, as there is about a 10MB constant overhead. Execution times show a less consistent trend but strongly correlate with the number of endpoints (and thus, messages) modeled in the network and no additional overhead for a small number of endpoints.

Execution time scales similarly for both traffic patterns, although higher under RandPerm, as explained in Section 6.2.1. Peak memory shows similar trends under both patterns for minimum and maximum values, but average peak memory shows a faster increase under RandPerm than UN traffic. This implies that most values for a given network size are closer to the maximum, which can be explained by the lower saturation point: values for offered network loads above saturation result in higher memory values due to the increase in packets in the network. These packets sit at in-transit, and server generation queues, but packets behind the head of a buffer do not strain the computational resources since they are not evaluated until they can be forwarded to the next buffer. This explains the similar evolution of the execution time under both patterns compared to the differences in average peak memory usage.

In general, it can be asserted that CAMINOS has good scalability, with a memory usage and execution time that strongly depend on the number of in-flight packets modeled in the network.

7. Conclusions

Network simulators that consider the details of the router microarchitecture are valuable tools for analyzing and developing large high-performance systems. They can assess the adequacy of the interconnection to the system or evaluate new proposals such as routing or deadlock avoidance mechanisms. This paper introduces CAMINOS, a modular network simulator written in the Rust programming language. CAMINOS

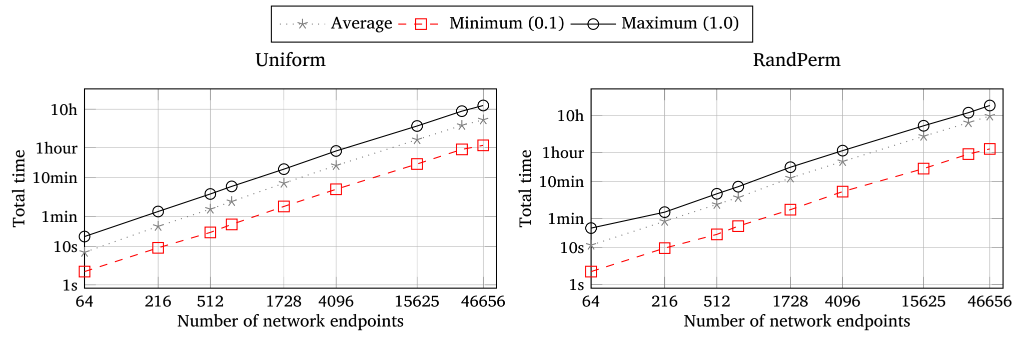

Fig. 7. Execution time as a function of the number of endpoints in the network, under Uniform and RandPerm traffic patterns.

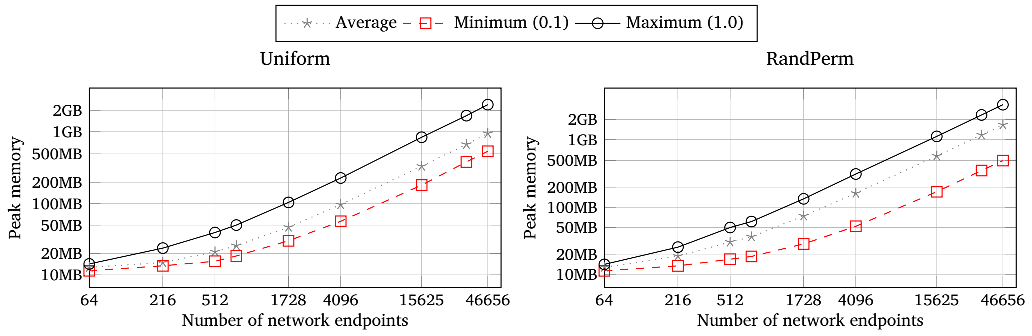

Fig. 8. Peak memory as a function of the number of endpoints in the network, under Uniform and RandPerm traffic patterns.

leverages the use of the Rust programming language and infrastructure to avoid common programming pitfalls, have C-like performance, install anywhere, and provide accessible documentation.

CAMINOS has a powerful configuration syntax that allows a wide variety of scenarios to be fully defined without modifying the source code. Additionally, CAMINOS offers a large array of available out-of-the-box metrics, including performance measures about the simulator itself, such as execution time and memory consumed. This, coupled with the integration SLURM job managers and generation of graphic outputs without external scripts or add-ons, allows a smoother and more productive workflow. CAMINOS presents a good use of resources, with the execution time and memory mainly driven by the amount of detail requested from the simulation; base use for both metrics is significantly low. This allows high scalability and, therefore, the analysis of large systems.

CAMINOS has been validated using a network scenario with a HyperX topology under different traffic patterns. Results are coherent with previous analyses in the literature and match the behavior of the same scenario in the BookSim network simulator.

CRediT authorship contribution statement

Cristóbal Camarero: Writing – review & editing, Writing – original draft, Software, Methodology, Investigation, Conceptualization. **Daniel Postigo:** Writing – review & editing, Writing – original draft, Software, Methodology, Investigation. **Pablo Fuentes:** Writing – review & editing, Writing – original draft, Methodology, Investigation.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgment

This work has been supported by the Spanish Ministry of Science, Innovation and Universities MICIU/AEI/ 10.13039/501100011033 under contracts PID2019-105660RB-C22, TED2021-131176B-I00 and PID2022-136454NB-C21, Ramón y Cajal contract RYC2021-033959-I and predoctoral grant with ref FPU23/03878.

We would like to show our gratitude to the reviewers for helping us improve the quality of the discourse.

Data availability

Data will be made available on request.

References

[1] ORNL, Frontier supercomputer press release, https://www.ornl.gov/news/frontier-supercomputer-debuts-worlds-fastest-breaking-exascale-barrier, 2022.
[2] F.J. Andújar, J.A. Villar, F.J. Alfaro, J.L. Sánchez, J. Escudero-Sahuquillo, An open-source family of tools to reproduce MPI-based workloads in interconnection network simulators, J. Supercomput. 72 (12) (2016) 4601–4628, https://doi.org/10.1007/s11227-016-1757-0.
[3] N.D. Matsakis, F.S. Klock, The rust language, Ada Lett. 34 (3) (2014) 103–104, https://doi.org/10.1145/2692956.2663188.
[4] C. Camarero, C. Martínez, R. Beivide, Polarized routing: an efficient and versatile algorithm for large direct networks, in: 2021 IEEE Symposium on High-Performance Interconnects, HOTI '21, 2021, pp. 52–59.
[5] C. Camarero, C. Martínez, R. Beivide, Polarized routing for large interconnection networks, IEEE MICRO 42 (2) (2022) 61–67.
[6] A. Cano, C. Camarero, C. Martínez, R. Beivide, Analysing mechanisms for virtual channel management in low-diameter networks, in: 2023 IEEE 35th International Symposium on Computer Architecture and High Performance Computing (SBAC-PAD), 2023, pp. 12–22.
[7] C. Camarero, CAMINOS: a modular interconnection network simulator, Common repository: https://crates.io/crates/caminos, Documentation: https://docs.rs/caminos-lib/latest/caminos_lib/. GitHub mirror: https://github.com/nakacristo/caminos-lib.
[8] A. Gavrilovska (Ed.), Attaining High Performance Communications: A Vertical Approach, Chapman and Hall/CRC, 2010.
[9] CWE/CAPEC Board, 2023 CWE top 25 most dangerous software weaknesses, https://cwe.mitre.org/top25/archive/2023/2023_top25_list.html, 2023.
[10] Y. Zhang, Y. Zhang, G. Portokalidis, J. Xu, Towards understanding the runtime performance of Rust, in: Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering, 2022, pp. 1–6.
[11] K. Chitre, P. Kedia, R. Purandare, The road not taken: exploring alias analysis based optimizations missed by the compiler, Proc. ACM Program. Lang. 6 (OOPSLA2) (2022) 786–810.

[12] W. Bugden, A. Alahmar, Rust: the programming language for safety and performance, in: 2nd. International Graduate Studies Congress. IGSCONG'22, 2022.

[13] J. Abdi, G. Posluns, G. Zhang, B. Wang, M.C. Jeffrey, When is parallelism fearless and zero-cost with Rust?, in: Proceedings of the 36th ACM Symposium on Parallelism in Algorithms and Architectures, 2024, pp. 27–40.

[14] N. McDonald, A. Flores, A. Davis, M. Isaev, J. Kim, D. Gibson, Supersim: extensible flit-level simulation of large-scale interconnection networks, in: 2018 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '18, IEEE, 2018, pp. 87–98.

[15] J.L. Hennessy, D.A. Patterson, A.C. Arpaci-Dusseau, Computer Architecture: A Quantitative Approach, 6th edition, Morgan Kaufmann, 2017, https://dl.acm.org/doi/book/10.5555/3207796.

[16] W.J. Dally, B.P. Towles, Principles and Practices of Interconnection Networks, Morgan Kaufmann, 2004.

[17] N. Jiang, D.U. Becker, G. Michelogiannakis, J. Balfour, B. Towles, D.E. Shaw, J. Kim, W.J. Dally, A detailed and flexible cycle-accurate network-on-chip simulator, in: 2013 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '13, 2013, pp. 86–96.

[18] W. Myung, Z. Qi, M. Cheng, Performance analysis of routing algorithms in mesh based network on chip using booksim simulator, in: 2019 IEEE International Conference of Intelligent Applied Systems on Engineering, ICIASE '19, 2019, pp. 297–300.

[19] H. Kasan, G. Kim, Y. Yi, J. Kim, Dynamic global adaptive routing in high-radix networks, in: Proceedings of the 49th Annual International Symposium on Computer Architecture, ISCA '22, Association for Computing Machinery, 2022, pp. 771–783.

[20] A.Q. Ansari, M.R. Ansari, M.A. Khan, Performance evaluation of various parameters of network-on-chip (NoC) for different topologies, in: 2015 Annual IEEE India Conference (INDICON), 2015, pp. 1–4.

[21] J. Marri, S. Manishankar, D. Radha, M. Moharir, Implementation and analysis of adaptive odd-even routing in booksim 2.0 simulator, in: 2019 International Conference on Communication and Electronics Systems (ICCES), 2019, pp. 76–83.

[22] J. Navaridas, J. Miguel-Alonso, J.A. Pascual, F.J. Ridruejo, Simulating and evaluating interconnection networks with INSEE, Simul. Model. Pract. Theory 19 (1) (2011) 494–515.

[23] F. Ridruejo, A. Gonzalez, J. Miguel-Alonso, TrGen: a traffic generation system for interconnection network simulators, in: 2005 International Conference on Parallel Processing Workshops (ICPPW'05), 2005, pp. 547–553.

[24] J. Miguel-Alonso, J. Navaridas, F. Ridruejo, Interconnection network simulation using traces of mpi applications, Int. J. Parallel Program. 37 (2009) 153–174.

[25] N. McDonald, M. Isaev, A. Flores, A. Davis, J. Kim, Practical and efficient incremental adaptive routing for hyperx networks, in: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC '19, ACM, 2019, pp. 1–13.

[26] M. García, P. Fuentes, M. Odriozola, E. Vallejo, R. Beivide, FOGSim interconnection network simulator, https://github.com/fuentesp/fogsim, 2014.

[27] M. Benito, P. Fuentes, E. Vallejo, R. Beivide, ACOR: adaptive congestion-oblivious routing in dragonfly networks, J. Parallel Distrib. Comput. 131 (2019) 173–188, https://doi.org/10.1016/j.jpdc.2019.04.022.

[28] P. Fuentes, E. Vallejo, R. Beivide, C. Minkenberg, M. Valero, FlexVC: flexible virtual channel management in low-diameter networks, in: 2017 IEEE International Parallel and Distributed Processing Symposium (IPDPS), 2017, pp. 842–854.

[29] M. García González, E. Vallejo Gutiérrez, R. Beivide Palacio, C. Camarero Coterillo, M. Valero, G. Rodríguez, C. Minkenberg, On-the-fly adaptive routing for dragonfly interconnection networks, J. Supercomput. 71 (3) (March 2015) 1116–1142, https://doi.org/10.1007/s11227-014-1357-9, http://hdl.handle.net/10902/6480.

[30] J. Navaridas, J.A. Pascual, A. Erickson, I.A. Stewart, M. Luján, Inrflow: an interconnection networks research flow-level simulation framework, J. Parallel Distrib. Comput. 130 (2019) 140–152, https://doi.org/10.1016/j.jpdc.2019.03.013.

[31] N. Agarwal, T. Krishna, L.-S. Peh, N.K. Jha, Garnet: a detailed on-chip network model inside a full-system simulator, in: 2009 IEEE International Symposium on Performance Analysis of Systems and Software, ISPASS '09, IEEE, 2009, pp. 33–42.

[32] J. Lowe-Power, A.M. Ahmad, A. Akram, M. Alian, R. Amslinger, M. Andreozzi, A. Armejach, N. Asmussen, B. Beckmann, S. Bharadwaj, G. Black, G. Bloom, B.R. Bruce, D.R. Carvalho, J. Castrillon, L. Chen, N. Derumigny, S. Diestelhorst, W. Elsasser, C. Escuin, M. Fariborz, A. Farmahini-Farahani, P. Fotouhi, R. Gambord, J. Gandhi, D. Gope, T. Grass, A. Gutierrez, B. Hanindhito, A. Hansson, S. Haria, A. Harris, T. Hayes, A. Herrera, M. Horsnell, S.A.R. Jafri, R. Jagtap, H. Jang, R. Jeyapaul, T.M. Jones, M. Jung, S. Kannoth, H. Khaleghzadeh, Y. Kodama, T. Krishna, T. Marinelli, C. Menard, A. Mondelli, M. Moreto, T. Mück, O. Naji, K. Nathella, H. Nguyen, N. Nikoleris, L.E. Olson, M. Orr, B. Pham, P. Prieto, T. Reddy, A. Roelke, M. Samani, A. Sandberg, J. Setoain, B. Shingarov, M.D. Sinclair, T. Ta, R. Thakur, G. Travaglini, M. Upton, N. Vaish, I. Vougioukas, W. Wang, Z. Wang, N. Wehn, C. Weis, D.A. Wood, H. Yoon, E.F. Zulian, The gem5 simulator: version 20.0+, https://arxiv.org/abs/2007.03152, 2020.

[33] I. Pérez, E. Vallejo, M. Moretó, R. Beivide, BST: a booksim-based toolset to simulate NoCs with single- and multi-hop bypass, in: 2020 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2020, pp. 47–57.

[34] A. Rodrigues, K. Hemmert, B. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, B. Jacob, The structural simulation toolkit, ACM SIGMETRICS Perform. Eval. Rev. 38 (2011) 37–42, https://doi.org/10.1145/1964218.1964225.

[35] C.L. Janssen, H. Adalsteinsson, S. Cranford, J.P. Kenny, A. Pinar, D.A. Evensky, J. Mayo, A simulator for large-scale parallel computer architectures, Int. J. Distrib. Syst. Technol. 1 (2010) 57–73, https://doi.org/10.4018/jdst.2010040104.

[36] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, R. Ross, Codes: enabling co-design of multilayer exascale storage architectures, in: Proceedings of the Workshop on Emerging Supercomputing Technologies, 2011, pp. 303–312.

[37] C.D. Carothers, D. Bauer, S. Pearce, Ross: a high-performance, low-memory, modular time warp system, J. Parallel Distrib. Comput. 62 (11) (2002) 1648–1669, https://doi.org/10.1016/S0743-7315(02)00004-7.

[38] Y. Ben-Itzhak, E. Zahavi, I. Cidon, A. Kolodny, HNoCS: modular open-source simulator for heterogeneous nocs, in: 2012 International Conference on Embedded Computer Systems (SAMOS), IEEE, 2012, pp. 51–57.

[39] A.S. Hassan, A.A. Morgan, M.W. El-Kharashi, An enhanced network-on-chip simulation for cluster-based routing, in: The 11th International Conference on Future Networks and Communications (FNC 2016) / The 13th International Conference on Mobile Systems and Pervasive Computing (MobiSPC 2016) / Affiliated Workshops, Proc. Comput. Sci. 94 (2016) 410–417, https://doi.org/10.1016/j.procs.2016.08.063.

[40] Y. Shen, A multitask parallel executor for ns-3 (network simulator), https://github.com/BobAnkh/ns3-parallel, 2022.

[41] D. Craven, Netsim-embed - a small network simulator, https://github.com/ipfs-rust/netsim-embed, 2023.

[42] M. Wagner, Kipa: key to ip address, https://github.com/mishajw/kipa, 2020.

[43] E. Harris-Braun, N. Luck, sim2h - a secure centralized "switchboard" implementation of lib3h, https://github.com/holochain/sim2h, 2019.

[44] A. Cann, P. Balciunas, netsim - network simulation in rust, https://github.com/canndrew/netsim, 2019.

[45] D. Sorokin, Discrete event simulation library (the generalized network interface), https://crates.io/crates/dvcompute_network, 2022.

[46] B.D.S. Dilinila, A behavioral model for simultaneous event execution in sequential discrete event system simulations, Master's thesis, Old Dominion University, 2021.

[47] J. Kim, W.J. Dally, S. Scott, D. Abts, Technology-driven, highly-scalable dragonfly topology, in: Proceedings of the 35th Annual International Symposium on Computer Architecture, ISCA '08, IEEE Computer Society, USA, 2008, pp. 77–88.

[48] C. Camarero, C. Martínez, E. Vallejo, R. Beivide, Projective networks: topologies for large parallel computer systems, IEEE Trans. Parallel Distrib. Syst. 28 (7) (2017) 2003–2016, https://doi.org/10.1109/TPDS.2016.2635640.

[49] M. Besta, T. Hoefler, Slim fly: a cost effective low-diameter network topology, in: SC '14: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2014, pp. 348–359.

[50] S.R. Öhring, M. Ibel, S.K. Das, M.J. Kumar, On generalized fat trees, in: Proceedings of 9th International Parallel Processing Symposium, 1995, pp. 37–44, https://api.semanticscholar.org/CorpusID:26494144.

[51] M. Valerio, L. Moser, P. Melliar-Smith, Recursively scalable fat-trees as interconnection networks, in: Proceeding of 13th IEEE Annual International Phoenix Conference on Computers and Communications, 1994, pp. 40–46.

[52] C. Camarero, C. Martínez, R. Beivide, Random folded clos topologies for datacenter networks, in: 2017 IEEE International Symposium on High Performance Computer Architecture (HPCA), 2017, pp. 193–204.

[53] M. Flajslik, E. Borch, M.A. Parker, Megafly: a topology for exascale systems, in: R. Yokota, M. Weiland, D. Keyes, C. Trinitis (Eds.), High Performance Computing, Springer International Publishing, Cham, 2018, pp. 289–310.

[54] A. Shpiner, Z. Haramaty, S. Eliad, V. Zdornov, B. Gafni, E. Zahavi, Dragonfly+: low cost topology for scaling datacenters, in: 2017 IEEE 3rd International Workshop on High-Performance Interconnection Networks in the Exascale and Big-Data Era (HiPINEB), 2017, pp. 1–8.

[55] Bhuyan, Agrawal, Generalized hypercube and hyperbus structures for a computer network, IEEE Trans. Comput. 100 (4) (1984) 323–333.

[56] J. Kim, W.J. Dally, D. Abts, Flattened butterfly: a cost-efficient topology for high-radix networks, in: Proceedings of the 34th Annual International Symposium on Computer Architecture, ISCA '07, Association for Computing Machinery, 2007, pp. 126–137.

[57] J.H. Ahn, N. Binkert, A. Davis, M. McLaren, R.S. Schreiber, HyperX: topology, routing, and packaging of efficient large-scale networks, in: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis, 2009, pp. 1–11.

[58] A. Singla, C.-Y. Hong, L. Popa, P.B. Godfrey, Jellyfish: networking data centers randomly, in: Proceedings of the 9th USENIX Conference on Networked Systems Design and Implementation, NSDI'12, USENIX Association, Berkeley, CA, USA, 2012, p. 17, http://dl.acm.org/citation.cfm?id=2228298.2228322.

[59] L.G. Valiant, A scheme for fast parallel communication, SIAM J. Comput. 11 (2) (1982) 350–361, https://doi.org/10.1137/0211027.

[60] K.D. Günther, Prevention of deadlocks in packet-switched data transport systems, IEEE Trans. Commun. 29 (4) (1981) 512–524, https://doi.org/10.1109/TCOM.1981.1095021.

[61] P. Merlin, P. Schweitzer, Deadlock avoidance in store-and-forward networks–I: store-and-forward deadlock, IEEE Trans. Commun. 28 (3) (1980) 345–354, https://doi.org/10.1109/TCOM.1980.1094666.

[62] A. Singh, Load-balanced routing in interconnection networks, Ph.D. thesis, Stanford University, 2005.

[63] R.K. Jain, D.-M.W. Chiu, W.R. Hawe, et al., A quantitative measure of fairness and discrimination for resource allocation in shared computer systems, Eastern Research

Laboratory, Digital Equipment Corporation, Hudson, MA 21, 1984, https://arxiv.org/abs/cs/9809099.

[64] A. Auten, M. Tomei, R. Kumar, Hardware acceleration of graph neural networks, in: 2020 57th ACM/IEEE Design Automation Conference (DAC), 2020, pp. 1–6.

[65] B. Klenk, N. Jiang, G. Thorson, L. Dennison, An in-network architecture for accelerating shared-memory multiprocessor collectives, in: 2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA), 2020, pp. 996–1009.

[66] N. Jain, A. Bhatele, S. White, T. Gamblin, L.V. Kale, Evaluating hpc networks via simulation of parallel workloads, in: SC '16: Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, 2016, pp. 154–165.

[67] N. McKeown, The iSLIP scheduling algorithm for input-queued switches, IEEE/ACM Trans. Netw. 7 (2) (1999) 188–201, https://doi.org/10.1109/90.769767.