2018 IEEE International Symposium on Performance Analysis of Systems and Software

SuperSim: Extensible Flit-Level Simulation of Large-Scale Interconnection Networks

Nic McDonald
Hewlett Packard Enterprise
nicmcd@hpe.com

Adriana Flores
Hewlett Packard Enterprise
adrianaf@hpe.com

Al Davis
Hewlett Packard Enterprise
ald@hpe.com

Mikhail Isaev
Georgia Institute of Technology
michael.v.isaev@gatech.edu

John Kim
Korea Advanced Institute of Science and Technology
jjk12@kaist.edu

Doug Gibson
Hewlett Packard Enterprise
doug.gibson@hpe.com

***Abstract*—The interconnection networks of modern large-scale computing systems are quickly increasing in size and complexity to keep up with the demand for computing capability. These systems rely heavily on complex router microarchitectures and intelligent adaptive routing algorithms structured for cost-optimized low-diameter networks. These technologies need to be properly modeled and evaluated during design space exploration and for performance characterization of the system.**

We present SuperSim, an open-source flit-level interconnection network simulator that enables focused evaluation of issues related to designing and deploying large-scale high-performance networks. SuperSim is a programmer-centric simulation framework explicitly designed to be flexibly extended and is supported by a number of tools making it easy to use and allowing users to model systems quickly. In this work we show the results for simulation case studies demonstrating the power of SuperSim to uncover otherwise overlooked details in large-scale interconnection networks.

***Keywords*-simulation, modeling, network, architecture**

I. Introduction

For high-performance computing systems the interconnection network is the critical piece of hardware that makes the system a “supercomputer”. In other realms, cloud and enterprise data centers and personal computers for example, one can find the same processors, memory, storage devices, and accelerators. It is the network that tightly couples these devices in such a way that it can be viewed and used as a single high-performance system. The network enables the programming runtimes (e.g., MPI, SHMEM, UPC, etc.) to be highly productive to programmers by supplying large amounts of bandwidth and low message latencies. Supercomputer architectures are seeing a large increase in size and complexity to achieve an exaflop of performance [32]. As the number and complexity of nodes continues to increase to achieve exascale, the importance and critical role of the network also increases.

This work has been supported in part by the U.S. Defense Advanced Research Projects Agency under Contract N66001-13-2-4003 and U.S. Department of Energy under LLNS Subcontract B621301.

It has been shown that with high router I/O bandwidth, the pairing of high-radix routers [21], [29] with low-diameter networks [3], [4], [17], [18], [20] most efficiently uses the router's bandwidth, minimizes the cost of the network, and increases application performance. These systems rely heavily on complex router microarchitectures and intelligent adaptive routing algorithms. These technologies need to be properly modeled and evaluated during design space exploration and for performance characterization. It has been our experience that current tools, whether they be high-level large system simulators or detail-oriented small system simulators, lack the appropriate features to properly model the important attributes of large-scale high-performance interconnection networks.

Simulators have long been used to quantify the performance of interconnection networks of many kinds. Various simulators operate at different granularities depending on what they are designed for, common granularities being *flows*, *packets*, and *flits*. Flow-level simulators model high-level interactions of streams of data across a network. These simulators are specifically useful for analyses of steady-state or long periods of time. Packet-level simulators model each individual packet which enables more accurate modeling of the effects of congestion and transient packet-based scheduling conflicts. Flit-level simulators model every flit of every packet. A *flit*, or **fl**ow control dig**it**, is the smallest unit of resource allocation in a router [11]. Flits are the units upon which routers manage buffering, data flow, and resource scheduling. As a result, flit-level simulation is required to gain thorough understanding of the behavior of a router microarchitecture.

In this work we present **SuperSim**, an open-source flit-level interconnection network simulator that properly models the architectural details of large-scale networks. Many of the drivers for the design of SuperSim align with key topics of interest in large-scale networks. These include:

- Large topologies with high channel latencies
- Architectural implications of congestion delays
- Deep pipelines and long processing latencies
- Complex hierarchical microarchitectures
- Multi-frequency designs
- The interaction of multiple workloads
- Heterogeneous endpoints

SuperSim is a flexible and extensible event-driven simulator written in C++. SuperSim models time abstractly allowing it to be used in cycle-accurate simulation, subcycle-accurate simulation, and higher-level simulation. It is a generic network simulator able to simulate large-scale networks, small-scale multiprocessor networks, and networks-on-chip (NoCs) [10]. It is used in both industrial and academic research projects and was chosen for the Interconnection Networks class at Stanford University. SuperSim places significant priority on programmer productivity by maintaining a well-defined hierarchical abstract interface enabling users to easily add their own component models. SuperSim provides effective *object factories* to enable users to integrate their code simply by dropping in new source files requiring zero changes to the existing code base.

SuperSim [13] is supported by many tools [23], [24], [25], [26] that make running simulations extremely easy. With these tools a user can write a simple Python script of only a few tens of lines which may result in an extensive simulation sweep containing tens of thousands of command line operations covering simulation runs, output parsing, statistics analysis, and plotting. These tools efficiently schedule the corresponding operations to run in the proper order and without resource conflicts on a single computer or across a large cluster of computers.

In summary, this work makes the following contributions:

- We describe the structure and capabilities of the open-source SuperSim interconnection network simulator.
- We describe the many open-source tools that support SuperSim and how these tools make it extremely easy to generate sweeps of simulations, analyses, and plots.
- Using the detailed modeling in SuperSim for large-scale router microarchitectures we perform three case studies showing the impact of common assumptions made in other simulators. Specifically we show the effects of latent congestion detection, realistic credit accounting, and common flow control techniques.

II. Related Work

Current simulators that model architectural details at the flit-level often focus modeling on networks-on-chip (NoCs) and point-to-point processor interconnects. In contrast, SuperSim was explicitly designed to be able to simulate the architectural details of large-scale networks. GEM5 [5] is a full system simulator that uses the Garnet [2] network simulator to model the network. Garnet structurally limits the size of simulation to 256 endpoints. BookSim [16] is a stand-alone network simulator designed to be flexible for the simulation of NoCs. While these tools are invaluable for the research of interconnection networks at small scale, their flexibility for expressing the architectures of large-scale networks lacks detail in key areas of importance.

Some tools are designed specifically for simulation of large systems, however, these tools compromise simulation accuracy to the flow or packet-level for a reduction in simulation execution time. SuperSim does not make this compromise as it is a flit-level simulator designed for large-scale networks. The NS [14], [27] and OMNeT++ [33] simulators are tools designed to model wired and wireless networks and network software stacks. The Structural Simulation ToolKit (SST) [28] and CODES [8] combined with TraceR [15] model the system at the packet-level and focus on how traffic is generated and injected into the network. For example, SST is able to run parallel program (e.g., MPI, SHMEM) *motifs* which are skeleton versions of the program which only express fixed compute times and the interactions with the parallel runtime. Similarly, CODES/TraceR replays MPI dependency-oriented trace logs to accurately inject traffic into the network. While these tools are invaluable for application-level exploration and high-level network bottleneck issues, their high-level packet-oriented focus makes them unusable for network microarchitecture design exploration and validation.

III. Simulation Framework

SuperSim[1] is written in standard C++ currently consuming over 30,000 lines of code with over 20% being explanatory comments. The code is implemented in a well-organized abstract class hierarchy utilizing over 350 source files.

Since no existing simulator focuses on detail-oriented large-scale networks, we designed SuperSim to fill this gap. Meticulous effort has been put into SuperSim to make it a flexible and extensible simulation framework instead of a ready-to-go simulator. This decision stems from an insight gained while discussing with a colleague about changes needed to be made in a simulator during a research project:

> *"If a simulator already does what you want it to do, there's a good chance you aren't asking the right questions."*
>
> — *Christos Kozyrakis [22]*

The insight of this statement is that if a simulator is already doing what you want it to do then someone else has likely already asked or answered those questions. The ramification of this insight is that design exploration research

[1]The project that spurred the first work of this simulation infrastructure was an architectural design space exploration for very high-radix routers. We called the design *SuperSwitch* thus the simulator became *SuperSim*.

generally requires significant extensions to existing simulators. If an industrial project requires detailed modeling in simulation it is likely that the proposed technology is not modeled in existing simulators. The #1 goal of SuperSim is to enable architects to quickly develop, instrument, and analyze new designs.

A. Discrete Event Simulation

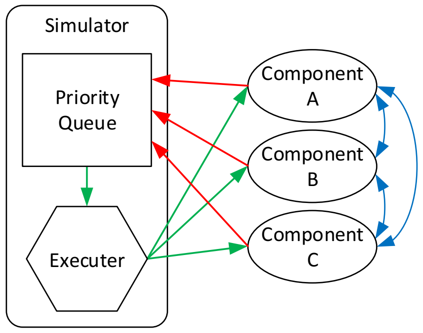


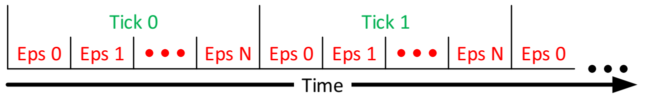
Figure 1. The discrete event simulation (DES) core of SuperSim

The simulation core of SuperSim is a discrete event simulation (DES) engine as shown in Figure 1. A simulation is natively built of *components* which are able to create *events*. An event is a simple object with a time value indicating when it is to be executed and a pointer to the component that will perform the execution. It may also contain component specific data. Each component links to the global simulator object and pushes its new events into the priority queue of the simulator. Components that interact during simulation call each other's functions which may in turn cause new events to be enqueued into the global event queue. The priority queue sorts the events such that the event with the earliest execution time is presented at the head of the queue. The engine's executer sequentially pulls events from the priority queue and executes them. The simulation is over when the event queue runs empty.

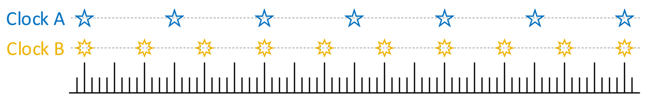
B. Time Representation

SuperSim flexibly represents time as a hierarchical value to enable architectures with custom needs. Time is composed of *ticks* and *epsilons*, as shown in Figure 2a. Ticks represent actual time and the user gets to decide the value of a tick. Examples of tick values could be 1 nanosecond, 457 picoseconds (an arbitrary number), or a specific clock cycle time. Using a real time value (e.g., picoseconds) yields performance results that are easy to understand by those who may not be familiar with digital logic design.

Epsilons are used to order operations performed within one time tick. Each tick has its own unique epsilons. Epsilons are **not** meant to represent real time and only serve to maintain order of operation within a given tick. The order at which the priority queue sorts events first looks at the tick value where a lower tick value is always higher priority regardless of the epsilons. If two events have equal tick values the epsilons are used to determine priority.



(a) Time divided into *ticks* and *epsilons*



(b) Multiple clock frequencies

Figure 2. Time representation in SuperSim

SuperSim allows the use of multiple clock frequencies in a design. Clock frequencies are specified by stating their cycle time in units of number of time ticks. Figure 2b shows an example of specifying two clock frequencies, Clock A having a 3 tick cycle time and Clock B having a 2 tick cycle time. Because SuperSim is modeling the digital logic of network architectures, having multiple clock frequencies is extremely useful in this framework. This is most commonly used to model switch frequency speedup where the switch core is run at a higher frequency than the links.

C. Configuration and Settings

The configuration of simulation infrastructures has long been a tedious and error-prone process. Instead of using a custom file format for configuration, SuperSim provides flexible configuration through the JSON open-standard file format that is described as "easy for humans to read and write" [12]. The natural hierarchy of JSON makes it a great API for configuration. For instance, the top level of any network simulation includes both a network configuration and a workload configuration. Using JSON these are specified in two separate blocks, *network* and *workload*. Besides having its own configuration variables, beneath the *network* block are more blocks such as *router* and *interface* specifying the configuration for the router and interface architectures, respectively. Similarly, the *router* block has its own configuration variables and more blocks such as *arbiter*.

When the simulator is building a particular configuration it simply looks at the current JSON hierarchy then is able to pass sub-blocks on to other constructors. For example, the simulator builds a Network object using the JSON's *network* block. A Network object contains Router objects but it does not care of what type they are. Instead of peeking down into the JSON hierarchy, the Network constructor passes the *router* block to the Router constructor for each Router object it builds.

SuperSim's JSON API implementation uses JSON at its core and also provides command line overrides, file inclusions, and object referencing. Listing 1 is a command line

```
$ supersim myconfig.json \
> network.router.architecture=string=my_arch \
> network.concentration=uint=16
```

Listing 1. An example usage of SuperSim command line arguments

example showing a user starting SuperSim with a settings file named “myconfig.json” and overriding two settings: the router architecture and the network concentration.

D. *Abstract APIs and Smart Factories*

Implementing new component models in SuperSim is made easy through the use of abstract C++ APIs and smart object factories implemented in the C++ preprocessor. The structure of SuperSim is both hierarchical and abstract. Each major type of component (e.g., Workload, Application, Network, Router, Allocator, Arbiter, etc.) within SuperSim is abstractly defined using an abstract base class. Specific implementations must derive from the base class to fit into the simulation infrastructure.

SuperSim provides a method for easily including new component models without modifying the existing code base. Traditionally, object factories are implemented by a function that takes a key, like a string description, then constructs and returns a new object. This function is commonly placed alongside the base class to define its generic interface. For example, the BookSim simulator uses a statically defined “`New()`” function for an object factory of each specific component type. However, this forces developers to alter the existing code base in order to be able to use their new component models. This makes working with the code base tedious particularly when multiple developers are contributing new component models to a design. This is especially problematic for scenarios where certain models will be kept proprietary.

SuperSim uses an extremely easy-to-use object factory generator. Developers simply put a call to the “`registerWithObjectFactory()`” macro in the source file of new simulation models. SuperSim’s factory generator automatically generates the factory function needed for all component types. When building the simulation components the simulator simply calls the factory function and an object is constructed corresponding to the name specified in the JSON settings. All this works in standard C++ without any additional code generation tools.

IV. Component Structure

The top level structure of SuperSim creates a distinct isolation between workload modeling and network modeling, as shown in Figure 3. This strict isolation ensures that network modeling has no baked in assumptions about the workload and the workload structurally works with any network model. Workloads can be customized to specific network configurations by passing the required network

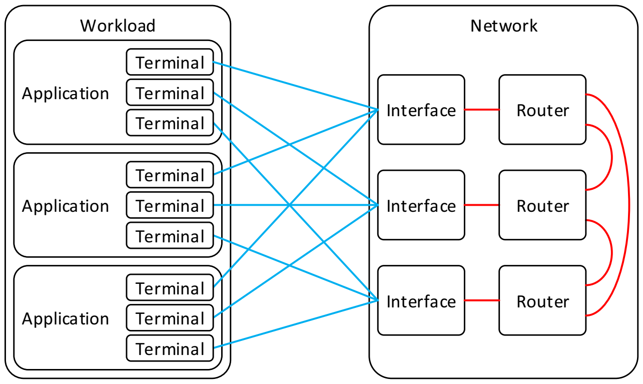


Figure 3. The isolation between workload modeling and network modeling

attributes, via JSON, to workload models that may require it. For example, the Tornado traffic pattern is an adversarial traffic pattern for a Torus topology, thus, when specifying this traffic pattern the user also gives it the configuration of the corresponding Torus topology.

A. *Workload*

SuperSim traffic generation and modeling is done hierarchically. SuperSim provides an API for multiple overlapping *Application* models to run concurrently. Each Application constructs one *Terminal* object per network endpoint. Each Terminal is responsible for generating the traffic for its specific Application on its specific endpoint.

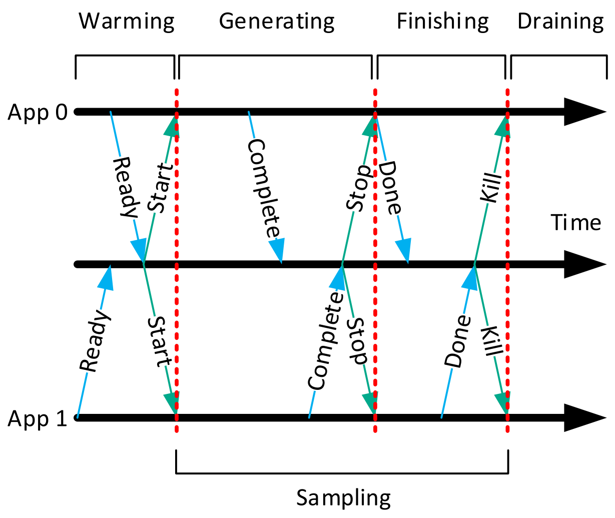


Figure 4. The handshake protocol between the Workload and multiple Applications to control the four phases of simulation

The *Workload* is defined as a state machine that monitors and controls the execution of all *Applications*. It aligns the Applications areas of interest with its sampling window where detailed information is gathered. As shown in Figure 4, the Workload uses a special handshake protocol to control the execution phases of all Applications. The workload defines four phases of execution:

1) **Warming:** This phase can be used by applications that require simulation time to prepare the network.
2) **Generating:** This phase is the primary time for applications to generate traffic to be sampled.
3) **Finishing:** This phase is used for any roll over traffic from the Generating phase that still needs to be sampled.
4) **Draining:** This phase simply drains the network as applications are not allowed to generate traffic.

This four phase protocol is derived from the common three phase approach of warming, sampling, and draining [11]. SuperSim divides sampling into generating and finishing to allow multiple applications to interoperate without explicitly being designed for each other. This allows users to design new applications and compose new workloads built from multiple applications.

Implicitly all applications start in the warming phase. When an application decides it is done warming it sends the *Ready* signal to the Workload. Applications may send network traffic to "warm up" the network if desired. Other applications might immediately send the Ready signal if they do not wish to perform any warming.

When all applications have reported Ready, the Workload simultaneously sends the *Start* command to all applications. This puts them in the generating phase. Each unique application generates its own traffic during this phase. When an application has determined that is has performed its necessary traffic generation, it sends the *Complete* signal to the Workload. At this point, some applications may have stopped sending traffic while others might continue sending traffic waiting for the next phase.

When all applications have reported Complete, the Workload simultaneously sends the *Stop* command to all applications. This puts them in the finishing phase where they are able to finish remaining traffic generation if they have not already done so. When finished the application sends the *Done* signal to the Workload.

When all applications have reported Done, the Workload simultaneously sends the *Kill* command to all applications. This puts them in the draining phase. After receiving a kill command, applications are not allowed to generate new traffic and all traffic in the network drains out. As a result, the event queue in the simulation core will run empty and the simulation will end.

A common multi-application experiment in SuperSim is a transient analysis of an adaptive routing algorithm. This uses the *Blast* application for steady state traffic and the *Pulse* application to generate a temporary disturbance, as shown in Figure 5 showing only traffic from Blast. While Blast warms up the network, Pulse remains idle waiting for the Start command from the workload. At that point, Blast continues sending traffic and begins sampling while Pulse starts sending traffic. Blast can immediately give the Complete signal because it does not care how long the sampling lasts. Pulse gives the Complete signal when it completes sending its traffic. After the Stop command is given the Blast application stops flagging traffic to be sampled but continues sending traffic at a constant injection rate. Once all traffic flagged for sampling has exited the network the draining phase safely begins.

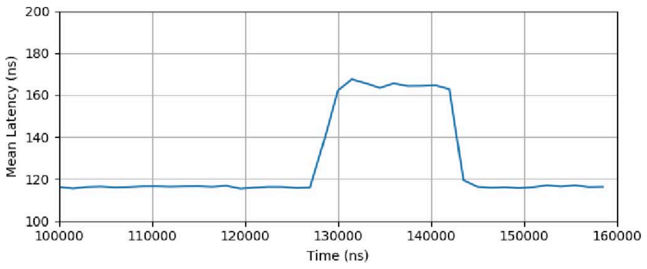


Figure 5. Blast application mean latency disrupted by the Pulse application.

B. Network

A SuperSim Network component model is responsible for defining the topology of the network and the routing algorithm being used in the topology. While the Network component does not define the architecture of the *Router* nor the *Interface*, it does instantiate these components and connects them together via *Channel* components. Similarly, the Interface and Router components are not made for a specific network topology or routing algorithm, but they must understand how to route packets through the network. When constructing a Network, the Network constructor provides access to its *RoutingAlgorithm* component factory for each Router object it creates. As the Router builds itself it uses this factory to construct RoutingAlgorithm components as needed. In this way, the router microarchitecture and the topology with its accompanying routing algorithm are modeled independently. The Network builds and configures Interface components in a similar fashion.

SuperSim comes with models of various topologies and routing algorithms. It is packaged with standard topologies such as Torus [9], Folded-Clos [7], HyperX [3] (which can make all configurations of HyperCube [30] and Flattened Butterfly [20]), and Dragonfly [18]. Across these topologies are implementations of both oblivious and adaptive routing.

SuperSim also has topologies that are used to stress test certain router architectural features. For instance, SuperSim contains a simple topology that creates the parking lot problem where age-based arbitration is known to fix the bandwidth unfairness of round-robin arbitration [1], [11].

C. Microarchitecture

SuperSim currently has three flexibly defined router microarchitectures. Each of these router architectures can be configured in various ways. The architectures are built using common components such as crossbars, virtual channel schedulers, crossbar schedulers, allocators, arbiters, and congestion sensors. All of these common components implement an abstract class interface and have many implementations available. Beyond these three architectures, users can piece together architectures of their own using the same building blocks or implement their own component models.

Output-Queued Architecture: The output-queued (OQ) architecture is an idealistic architecture designed to model a router in which there is zero head-of-line blocking and no scheduling conflicts. The size of the output queues can be infinite or finite. All input ports can simultaneously put a packet in any output queue (i.e., no scheduling conflicts). Another benefit of this architecture is a reduced simulation execution time. The modeling of the OQ router is simple and devoid of the details that commonly slow down simulation (e.g., virtual channel allocation, crossbar scheduling, etc.).

Input-Queued Architecture: The input-queued (IQ) architecture is modeled after the standard input-queued architecture in [11]. It has full crossbar input speedup and an optimized input-queue pipeline for processing flits of back-to-back packets. In this architecture, flits wait in the input queues until available downstream (i.e., next hop) credits are available.

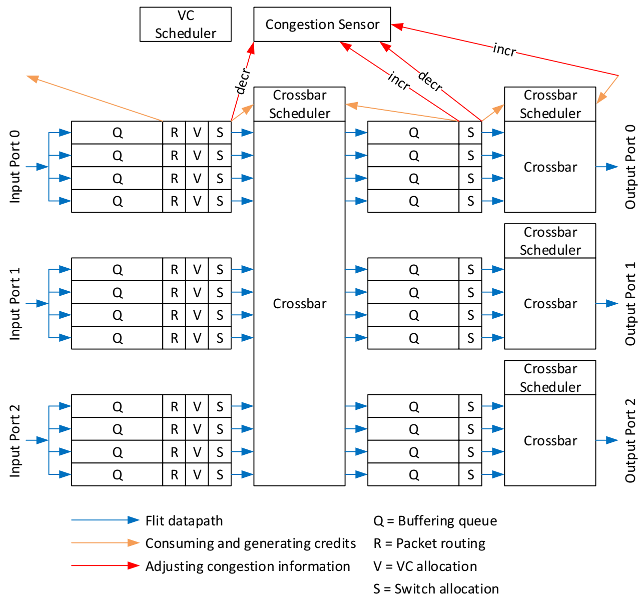


Figure 6. Input-output-queued (IOQ) microarchitecture in SuperSim

Input-Output-Queued Architecture: The input-output-queued (IOQ) architecture, shown in Figure 6, is also modeled after the standard input-queued architecture in [11] but has been extended as a combined input/output queued switch [6]. It has full crossbar input and output speedup and has similar pipeline optimizations in both the input and output queues. In this architecture, flits only wait in the input queues until credits are available for the output queues. After arriving in the output queues flits wait until downstream (i.e., next hop) credits are available.

D. *Error Detection*

Since SuperSim is a framework designed to be extended, users often design new component models. The worst case scenario is a bug in the code that goes unnoticed. The framework of SuperSim is designed to catch these bugs early on. A few examples are as follows. Every flit delivered to a destination is guaranteed to have arrived at the right destination and in the right order with respect to other flits in the packet. The outputs generated by routing algorithms are checked to not use VCs which have not been registered to that specific algorithm. Traffic that attempts to target an unused router output port is rejected. Buffers never silently overrun and credits never go negative.

V. Accompanying Tools

The common workflow for a simulation experiment follows these steps:

1) **Configure**: Create the simulation configurations needed for the experiment.
2) **Simulate**: Run the simulations using the configurations and generate raw data.
3) **Parse**: Parse the results of the simulation output into the format needed in the remaining steps.
4) **Analyze**: Analyze the parsed results from simulation to create desired statistics.
5) **Plot**: Generate plots of analysis data.
6) **View**: View the analyzed and plotted results.

We have created a multitude of tools accompanying SuperSim for each of these steps as well as tools to perform the whole flow autonomously. Having intelligent and easy-to-use tools gives power to users to uncover insights and problems where they might not have thought to look.

SSParse [23]: During the sampling window in a SuperSim simulation, network transaction information is logged to a verbose file format. The SSParse tool parses this file format and generates latency based information in various formats. It generates latency and hop count based information for packets, messages, and transactions. It generates both aggregate information, like latency distributions, as well as raw latency data used for plotting. SSParse has an easy-to-use filtering mechanism that allows users to view subsets of the data generated by SuperSim. For example, passing SSParse a filter of "`+app=0`" tells SSParse to only analyze traffic generated by application 0. A filter of "`+send=500-1000`" tells SSParse to only analyze traffic that was sent from time 500 to 1000. Multiple filters can be specified and many more filtering options are available.

SSPlot [24]: SSPlot is a plotting package that uses the Python-based Matplotlib package. SSPlot is both a Python

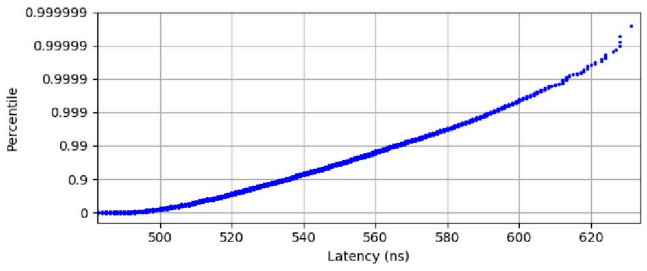


Figure 7. A percentile distribution plot generated by SSPlot

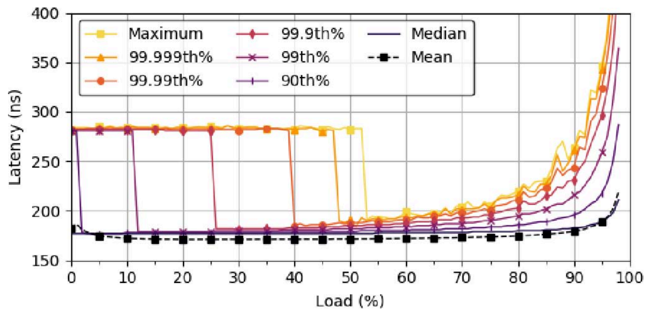


Figure 8. A sample load versus latency plot showing latency distributions

package and a set of command line executables and users can use whichever they prefer. Of critical importance to all the analysis tools is analyzing and viewing latency distributions, not just average latency. SSPlot generates average latency plots (shown in Figure 5), scatter plots, probability density function (PDF) plots, cumulative distribution function (CDF) plots, and percentile distribution plots (shown in Figure 7). The percentile distribution is particularly useful for understanding expected latencies in a network. As shown, the 99.9^{th} percentile latency is 592 ns which means that only 1 in 1000 packets experience latency greater than 592 ns. This is the expected latency for 1000-way parallelism.

Figure 8 is a load versus latency plot, the primary method used to describe network performance [11], showing multiple latency distributions across a sweep of simulations. The plot lines stop at the point where the network becomes saturated (98% in this case) because a saturated network yields infinite latency. This particular plot was generated from an adaptive routing experiment where the effects of phantom congestion [34] were present. In this simulation a non-minimal adaptive routing decision causes a packet to traverse an additional 50 ns channel latency and 50 ns router traversal. In the plot it is clearly shown that at low loads a significant amount of traffic is going non-minimal. Since the plot shows latency distributions we are able to specifically see that more than 1 in 10 packets (90^{th}%) are going non-minimal at zero-load. At 12% injection it drops to 1 in 100 packets (99^{th}%) and by 40% injection phantom congestion effects have eased to where less than 1 in 10,000 packets (99.99^{th}%) are going non-minimal. This detailed knowledge is in stark contrast to the prior art that only viewed the little bump at zero-load produced by the mean latency.

TaskRun [25]: TaskRun is an easy-to-use Python package for running tasks with dependencies, conditional execution, resource management, and much more. The process from running simulations, parsing the results, analyzing the data, and plotting the results entails many steps and each step has dependencies on previous steps. TaskRun makes this process easy and automated. A TaskRun script can elegantly run thousands of simulations and all required post-simulation tools. TaskRun is also able to interface with batch scheduling systems (e.g., Slurm, GridEngine, PBS, LSF, etc.). TaskRun is not specific to SuperSim but was created to fit its scalable and dependency-ordered execution needs.

```
latencies = [1, 2, 4, 8, 16, 32, 64]
def set_latency(latency, config):
    return "network.channel.latency=uint=" +
        str(latency)
sweeper.add_variable("ChannelLatency", "CL",
    latencies, set_latency)
```

Listing 2. An example definition of a simulation sweep variable in SSSweep

SSSweep [26]: SSSweep is a flexible Python package that automatically generates and executes simulations, parsing, analyzing, and plotting across one or more sweeping variables. Users write a short Python script that includes a few lines of code per simulation variable they desire to sweep. Each variable is accompanied with a user specified function that generates the corresponding SuperSim command line setting override. Listing 2 is a code snippet showing a small block of code declaring a sweep variable for simulating various channel latencies. SSSweep generates all the commands with corresponding dependencies of all permutations of variables and uses TaskRun to run all the tasks. SSSweep can literally turn a tiny amount of Python code into a complex, exhaustive, and autonomous simulation, analysis, and plotting sweep. SSSweep generates a web viewer using HTML, CSS, and Javascript. Simulation sweeps often result in many thousands of plots and the web viewer organizes all of this making it easy to find and share particular plots of interest.

VI. Case Studies

In this section we present the results of three simulation case studies that show the importance of realistic modeling of the router architectures of large-scale networks. These experiments exemplify some critical features of large scale network and router design that SuperSim is designed to support. SuperSim's natural focus on large scale system attributes enabled us to perform these experiments without altering any code. The critical simulation parameters for the experiments are in Table I.

Table I
PARAMETERS FOR THE THREE SIMULATION CASE STUDIES

Parameter	Latent Congestion Detection	Congestion Credit Accounting	Flow Control Techniques
Network topology	3-level folded-Clos, 4096 terminals	1D flattened butterfly, 32 routers, 1024 terminals	4D torus 8x8x8x8, 4096 terminals
Network channel latency	50 ns (i.e., 10 meter cables)	50 ns (i.e., 10 meter cables)	5 ns (i.e., 1 meter cables)
Routing algorithm	adaptive uprouting	UGAL	dimension order routing
Router radix	32 ports	63 ports	9 ports
Router architecture	output-queued (OQ)	input-output-queued (IOQ)	input-queued (IQ)
Frequency speedup	1x (i.e., none)	2x	1x (i.e., none)
Number of VCs	1 VC	2 VCs	2,4,8 VCs
Input buffer size	150 flits	128 flits	128 flits
Output buffer size	infinite and 64 flits	256 flits	n/a
Router core latency	50 ns queue-to-queue	50 ns main crossbar latency	25 ns main crossbar latency
Message size	1 flit	1 flit	1,2,4,8,16,32 flits
Traffic pattern	uniform random to root	uniform random, bit complement	uniform random

A. Latent Congestion Detection

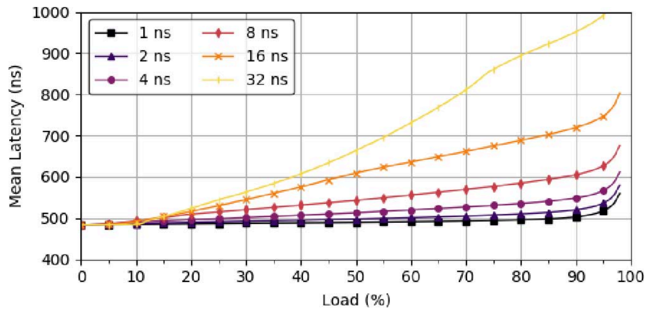
In this simulation experiment we analyze the negative effects of congestion detection latency within a high-radix router architecture. To the best of our knowledge all prior work has assumed that the propagation of congestion information from the point of calculation from within the microarchitecture to all the routing functions occurs in a single cycle. The delays through modern switch architectures are realistically in the range of 5-20 clock cycles.

In a high-radix router architecture each input port's routing engine operates independently. Kim et al. [19] showed that adaptive routing causes the packets resident in multiple input ports to bombard a seemly good output port resulting in excessive queuing delay. In our study, we show how latent congestion detection exacerbates this problem.

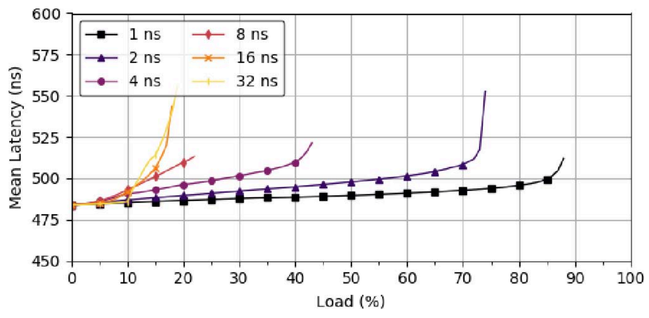
For this study we use a 4096-node 3-level folded-Clos topology and a random traffic pattern where all traffic reaches the root of the network. We use the same basic adaptive routing algorithm as Kim et al. where each packet chooses the least congested output port on its way up the network. We use the idealistic output-queued (OQ) router architecture to remove any microarchitecture-based bottlenecks. We test infinite output queues as well as finite output queues. We vary the propagation latency of sensed congestion information from 1 to 32 nanoseconds.

Figure 9 shows the results of this simulation sweep. The performance variation of using infinite output queues (shown in Figure 9a) shows that high latency congestion sensing will cause significantly higher message latencies. The throughput is not affected because the infinite queues can infinitely sink traffic. In contrast, the performance variation of using finite 64 flit output queues (shown in Figure 9b) shows that throughput performance is severely limited with high latency congestion sensing. With 1 ns of congestion sensing delay good throughput is achieved but with 2 ns of delay throughput drops by ∼15%, with 4 ns of delay throughput drops by ∼45%, and with 8 ns of delay throughput drops by ∼65%.

Our simulations of smaller systems yield less severe



(a) Infinite output queues



(b) 64 flit output queues

Figure 9. Comparison of various congestion sensing latencies using adaptive routing on a 3 level folded-Clos topology with a) infinite output queues, b) 64 flit output queues

penalties. For example, the same setup using radix-16 routers builds a system of 512 terminals. The achieved throughput of 1, 2, 4, and 8 ns of congestion latency delay is 90%, 90%, 75%, and 40%, respectively. More network levels and higher radix routers exacerbates the issue of multiple routing engines simultaneously choosing the same output port.

This experiment clearly outlines the importance of modeling router architectures accurately at the flit-level as packet-level modeling has simplified assumptions about buffer management that loses detail about the specific flow of flits and the reverse flow and timing of credits. Many high-level simulators yield performance characteristics similar to Figure 9a, however, realistic implementations yield performance more like Figure 9b when these effects are not properly handled. The difference can yield detrimental mispredictions of network performance.

B. Congestion Credit Accounting

In this simulation experiment we analyze the effects of various forms of credit accounting methodologies. Each of the three router architectures described in Section IV-C have a flexible design for the way in which credit information is given to the Congestion Sensor component. The supplied Congestion Sensor component in SuperSim uses current credit information to yield a congestion value for potential paths being considered by a routing algorithm implementation. Adaptive routing algorithms use this congestion information to make decisions about which path(s) yield best performance.

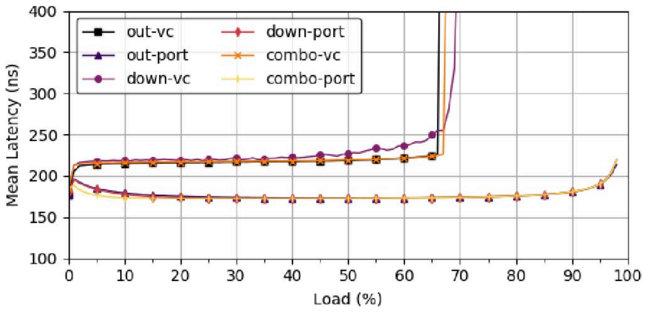
In the seminal work on source-based global adaptive routing, Singh developed the *Universal Global Adaptive Load-balancing (UGAL)* algorithm [31]. The simulations that proved its usefulness used an idealistic output-queued switch architecture, channels with zero latency, and congestion was strictly based on the number of flits resident in the output queues. In this experiment we use the input-output-queued (IOQ) router architecture (shown in Figure 6) to investigate the performance ramifications of various styles of congestion credit accounting on systems with realistic router and channel latencies. The IOQ architecture supports reporting congestion status on a per-VC basis or on a per-port basis. It also supports viewing only credits for the output queues, only credits for the downstream queues, or the combination of credits of both output and downstream queues.

This experiment is run using both load-balanced uniform random (UR) traffic and unbalanced bit complement (BC) traffic on a 1024-node 1D Flattened Butterfly topology.

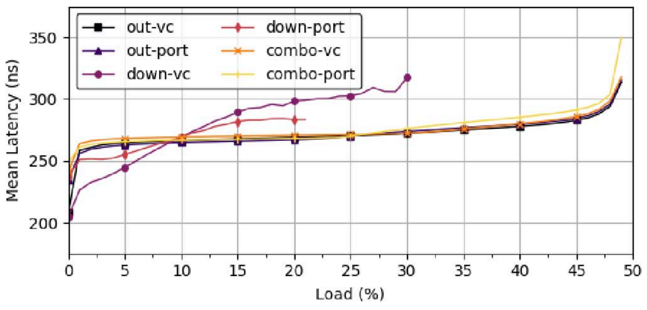
Figure 10a shows the performance achieved with UR traffic across the six different congestion credit accounting styles. As shown, the port-based accounting methodology provides higher throughput by an average of 31.6% and significantly less latency. Using output, downstream, or combined output and downstream credits does not seem to change the performance by any significant amount.

Figure 10b shows the performance achieved with BC traffic across the six different congestion credit accounting styles. In contrast to the above, the VC-based accounting methodology provides higher throughput, which it does by 3.33% on average. Apparent from these results is that using only downstream credits does not allow the adaptive routing algorithm to properly sense the congestion of BC traffic.

This type of experimentation is crucial for router architects as they need to decide how credit information is fed to the congestion sensing logic and they need to know the ramifications of such options. For example, port-based congestion sensing may have adverse effects when using traffic classes where some VCs are meant to be isolated from others. VC-based congestion sensing may have adverse effects when using technologies like age-based arbitration because being at the head of a queue does not yield any specific priority. SuperSim's flexible framework allows router architects to easily experiment with credit accounting styles in conjunction with other technologies to see if they fit well together.



(a) Uniform random traffic



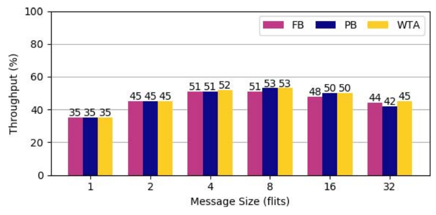
(b) Bit complement traffic

Figure 10. Comparison of various six credit accounting styles with a) uniform random traffic and, b) bit complement traffic

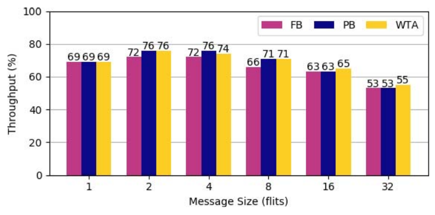
C. Flow Control Techniques

In this simulation experiment we analyze the effects of various flow control techniques on a 4096-node 4D torus topology using the input-queued (IQ) router architecture. In SuperSim, configuring different flow control techniques is easily done by giving the Crossbar Scheduler component various settings. In this experiment we evaluate the following three flow control techniques, described by Dally et al. [11]:

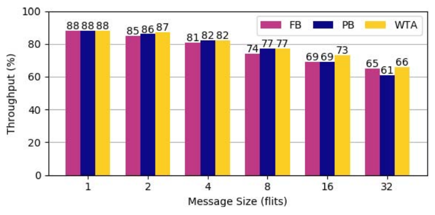
- **Flit-Buffer Flow Control (FB)**: This technique performs flit-by-flit scheduling of the crossbar. If two packets are in arbitration for the same output channel the flits of both packets interleave each taking 50% of the bandwidth. This is a fair bandwidth allocation policy.
- **Packet-Buffer Flow Control (PB)**: This technique performs packet-by-packet scheduling of the crossbar. A packet is only able to win arbitration if there is enough downstream space for the entire packet. Once a packet wins arbitration the decision is locked until



(a) 2 VCs



(b) 4 VCs



(c) 8 VCs

Figure 11. Throughput performance of various flow control techniques on a 4096-node 4D torus across various messages with a) 2 VCs, b) 4 VCs, and c) 8 VCs

the tail flit enters the crossbar. Because the scheduler ensures there is enough available space for the full packet, there will be no credit stalls once the packet starts streaming.

- **Winner-Take-All Flow Control (WTA)**: This technique is a hybrid between the prior two. It performs flit-by-flit scheduling of the crossbar and the scheduler locks the decision once made. Because this is a flit-level technique the scheduler does not wait for enough credits for the full packet before letting it start. Since credit stalls can occur with this technique, if a streaming packet encounters a lack of credits, the scheduling decision is unlocked and other packets with available credits are able to take over.

For this experiment we test systems with various numbers of virtual channels (i.e., 2, 4, 8) with various message sizes (i.e., 1, 2, 4, 8, 16, 32 flits). Figure 11 shows the results of 1,800 simulations generated by SSSweep using only 50 lines of Python. For the most part there is very little performance differences between the three flow control techniques. The explanation of this can be seen by looking only at the simulation results for single flit messages where the effects of the three flow control techniques have no variance because they all act the same. On small systems with small latencies, the length of packets and wormhole routing [11] causes packets to physically hold many resources concurrently potentially existing simultaneously in many routers. However, in large systems with high latencies packets generally are only able to exist within one device and the blocking issues related to long packets are significantly reduced. Thus, at large scales with small to medium sized packets, the unit of allocation (i.e., flit vs. packet) is negligible.

The latency results of this experiment are mostly similar across the three flow control techniques with few exceptions. The comparison shown in Figure 12 shows a load versus latency plot of the configuration of 8 VCs with 32 flit messages. The blocking effects of large 32 flit messages are severe but 8 VCs provides the potential for the flow control technique to find ways around blocked packets. As shown, the pure flit-buffer flow control technique provides the best resilience to blocking and results in the lowest latency. Packet-buffer flow control does the worst and winner-take-all flow control performs in the middle, which makes sense

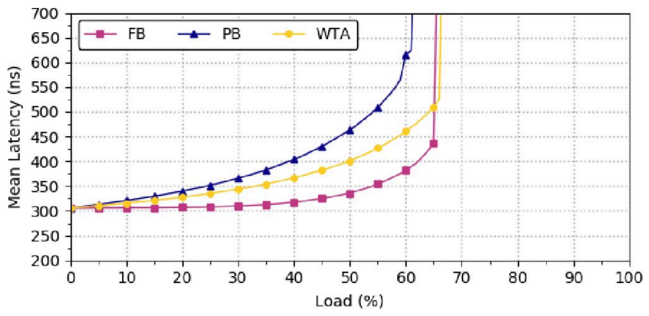


Figure 12. Latency performance of the flow control techniques with 8 VCs and 32 flit messages

because it is a hybrid of the two.

The results of this experiment oppose that of prior art for small-scale networks [11] which states that the flow control technique used for resource allocation plays an enormous part in overall performance. Our results yield the insight that if bandwidths continue to increase and packets remain relatively small, the number of flits per packet decreases and the flow control technique becomes less meaningful. A critical design outcome of this is that if packet-buffer flow control is easier to implement than flit-buffer flow control and is therefore desired, the maximum packet size should be kept as small as possible within the bounds of reasonable payload efficiency.

VII. Conclusion

In this work we have presented the SuperSim network simulation framework and its supporting tools. SuperSim's design allows users to easily model new topologies, routing algorithms, and microarchitectures and easily integrate them into the SuperSim codebase. SuperSim's supporting tools are able to autonomously generate and execute complex simulation sweeps, analyses, and plots with very minimal effort from the user. In this work we used SuperSim's ability to model large-scale systems to show that accurate prediction of system performance critically depends on modeling the microarchitecture accurately at the flit-level.

References

[1] D. Abts and D. Weisser, “Age-Based Packet Arbitration in Large-Radix k-ary n-cubes,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2007.

[2] N. Agarwal, T. Krishna, L.-S. Peh, and N. K. Jha, “GARNET: A Detailed On-Chip Network Model Inside a Full-System Simulator,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2009.

[3] J. H. Ahn, N. Binkert, A. Davis, M. McLaren, and R. S. Schreiber, “HyperX: Topology, Routing, and Packaging of Efficient Large-Scale Networks,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2009.

[4] M. Besta and T. Hoefler, “Slim Fly: A Cost Effective Low-Diameter Network Topology,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2014.

[5] N. Binkert, B. Beckmann, G. Black, S. K. Reinhardt, A. Saidi, A. Basu, J. Hestness, D. R. Hower, T. Krishna, S. Sardashti *et al.*, “The GEM5 Simulator,” *ACM SIGARCH Computer Architecture News*, vol. 39, no. 2, pp. 1–7, 2011.

[6] S.-T. Chuang, A. Goel, N. McKeown, and B. Prabhakar, “Matching Output Queueing with a Combined Input/Output-Queued Switch,” *IEEE Journal on Selected Areas in Communications*, vol. 17, no. 6, pp. 1030–1039, 1999.

[7] C. Clos, “A Study of Non-Blocking Switching Networks,” *Bell Labs Technical Journal*, vol. 32, no. 2, pp. 406–424, 1953.

[8] J. Cope, N. Liu, S. Lang, P. Carns, C. Carothers, and R. Ross, “CODES: Enabling Co-Design of Multilayer Exascale Storage Architectures,” in *Proceedings of the Workshop on Emerging Supercomputing Technologies*, 2011.

[9] W. J. Dally and C. L. Seitz, “The Torus Routing Chip,” *Distributed computing*, vol. 1, no. 4, pp. 187–196, 1986.

[10] W. J. Dally and B. Towles, “Route packets, not wires: On-chip interconnection networks,” in *Design Automation Conference (DAC)*. ACM/IEEE, 2001.

[11] W. J. Dally and B. P. Towles, *Principles and Practices of Interconnection Networks*. Elsevier, 2004.

[12] European Computer Manufacturers Association (ECMA). Ecma-404 the json data interchange standard. [Online]. Available: https://www.json.org/

[13] Hewlett Packard Enterprise. SuperSim Interconnection Network Simulator. [Online]. Available: https://github.com/HewlettPackard/supersim

[14] Information Sciences Institute. The network simulator - ns-2. [Online]. Available: https://www.isi.edu/nsnam/ns/

[15] N. Jain, A. Bhatele, S. White, T. Gamblin, and L. V. Kale, “Evaluating HPC Networks via Simulation of Parallel Workloads,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2016.

[16] N. Jiang, J. Balfour, D. U. Becker, B. Towles, W. J. Dally, G. Michelogiannakis, and J. Kim, “A Detailed and Flexible Cycle-Accurate Network-on-Chip Simulator,” in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*. IEEE, 2013.

[17] G. Kathareios, C. Minkenberg, B. Prisacari, G. Rodriguez, and T. Hoefler, “Cost-Effective Diameter-Two Topologies: Analysis and Evaluation,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2015.

[18] J. Kim, W. J. Dally, S. Scott, and D. Abts, “Technology-Driven, Highly-Scalable Dragonfly Topology,” in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2008.

[19] J. Kim, W. J. Dally, and D. Abts, “Adaptive Routing in High-Radix Clos Networks,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2006.

[20] J. Kim, W. J. Dally, and D. Abts, “Flattened Butterfly: A Cost-Efficient Topology for High-Radix Networks,” in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2007.

[21] J. Kim, W. J. Dally, B. Towles, and A. K. Gupta, “Microarchitecture of a High Radix Router,” in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2005.

[22] C. Kozyrakis, Stanford CS316, Advanced Multicore Systems, Sept. 2012.

[23] N. McDonald. SSParse - Parsing Engine for SuperSim. [Online]. Available: https://github.com/nicmcd/ssparse

[24] N. McDonald. SSPlot - Plotting Engine for Network Analysis. [Online]. Available: https://github.com/nicmcd/ssplot

[25] N. McDonald. TaskRun - Task Scheduling and Management in Python. [Online]. Available: https://github.com/nicmcd/taskrun

[26] N. McDonald and A. Flores. SSSweep - Simulation Sweep Generator. [Online]. Available: https://github.com/nicmcd/sssweep

[27] nsnam. ns-3. [Online]. Available: https://www.nsnam.org/

[28] A. F. Rodrigues, R. C. Murphy, P. Kogge, and K. D. Underwood, “The structural simulation toolkit: exploring novel architectures,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2006.

[29] S. Scott, D. Abts, J. Kim, and W. J. Dally, “The Blackwidow High-Radix Clos Network,” in *International Symposium on Computer Architecture (ISCA)*. IEEE, 2006.

[30] C. L. Seitz, “The Cosmic Cube,” *Communications of the ACM*, vol. 28, no. 1, pp. 22–33, 1985.

[31] A. Singh, “Load-Balanced Routing in Interconnection Networks,” Ph.D. dissertation, Stanford University, 2005.

[32] U.S. Department of Enery (DOE). Exascale computing project. [Online]. Available: https://exascaleproject.org/

[33] A. Varga and R. Hornig, “An Overview of the OMNeT++ Simulation Environment,” in *International Conference on Simulation Tools and Techniques for Communications, Networks and Systems*. ICST, 2008.

[34] J. Won, G. Kim, J. Kim, T. Jiang, M. Parker, and S. Scott, “Overcoming far-end congestion in large-scale networks,” in *International Conference for High Performance Computing Networking, Storage, and Analysis (SC)*. ACM/IEEE, 2015.