Improvements to the Structural Simulation Toolkit

Arun Rodrigues
Sandia National Labs
afrodri@sandia.gov

Keren Bergman
Columbia University
kb2028@columbia.edu

David Bunde
Knox College
dbunde@knox.edu

Elliot Cooper-Balis
U. Maryland
ecc17@umd.edu

Kurt Ferreira
Sandia National Labs
kbferre@sandia.gov

K. Scott Hemmert
Sandia National Labs
kshemme@sandia.gov

ABSTRACT

Designing supercomputer architectures and applications is becoming more difficult because of their increased size and complexity, because of new technologies, and due to new constraints such as power and thermal limits.

The Structural Simulation Toolkit (SST) is an architectural simulation framework designed to assist in the design, evaluation, and optimization of High Performance Computing (HPC) architectures and applications. Its initial release included a parallel simulation core with a number of system component models.

The SST has been expanded and improved in a number of ways. New memory, network, and processor models have been added, as well as new high-level system simulation capabilities. Also, scalability results are presented.

Categories and Subject Descriptors

B.6.3 [**Simulation**]: Logic Design

Keywords

Simulation, Architecture

1. INTRODUCTION

The complexity of supercomputer architectures and applications continues to grow as their performance improves. New technologies such as GPGPUs, silicon photonics, and advanced packaging, and new constraints, such as power and thermal limits, also complicate design.

The Structural Simulation Toolkit (SST), first introduced in [21], is an open, modular, framework designed to assist in the design, evaluation, and optimization of HPC architectures and applications. It consists of a parallel simulation core with a number of network, memory, and processor models, capable of evaluating systems at different levels of resolution.

The SST has been expanded and improved in a number of ways. New memory, network, and processor models have been added, as well as new high-level system simulation capabilities for modeling scheduling and application resilience. The improvements covered in this paper include new network components (Sections 4.1, 4.2, and 4.3), A new memory model (Section 4.4), new processor and GPGPU models (Section 4.5 and 4.6) and high-level system models of scheduleing (Section 4.7) and node allocation (Section 4.8). Each of these models have been modified to share a common view of simulated time, and can pass each other events. Preliminary scalability results are also presented (Section 3).

Simutools 2012, March 19-23, Desenzano del Garda, Italy

DOI 10.4108/icst.simutools.2012.247848

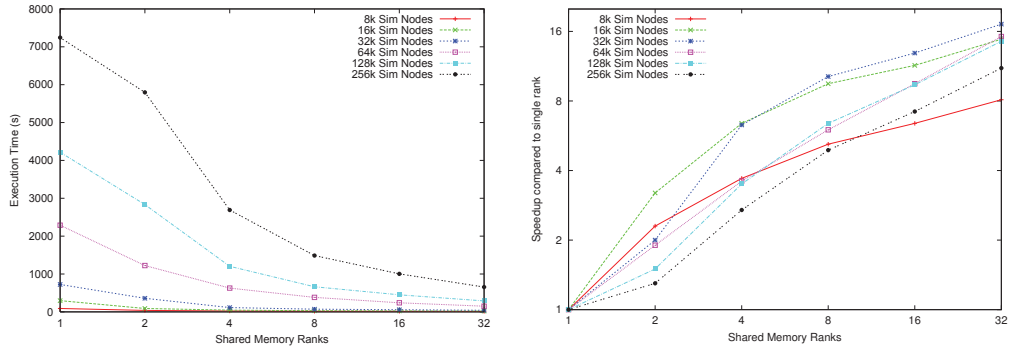
2. RELATED WORK

The SST simulation framework builds on a long history of architectural simulators. Giving full justice to this long tradition is beyond the scope of this paper, however some key simulators which influenced SST include Omnet++, SimpleScalar[3], NS-3[10], and A-Sim[19]. SST's parallel infrastructure builds on previous work[9] in parallel discrete event simulation.

3. SCALABILITY

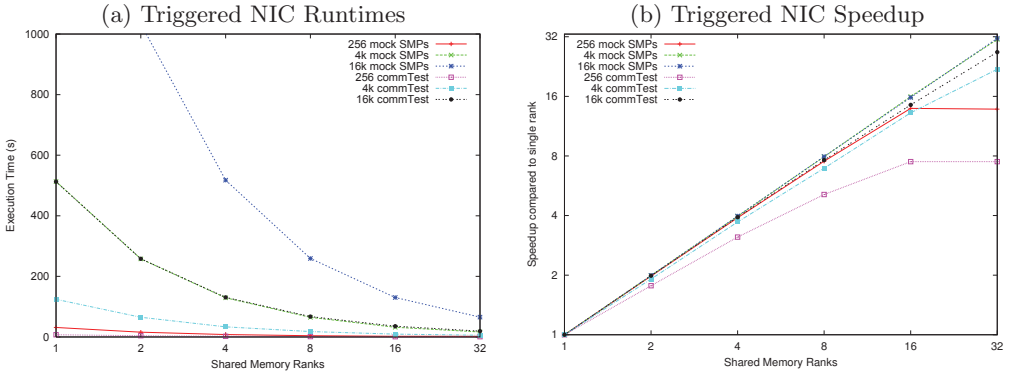
Early shared memory MPI scalability testing of the SST was performed on a 4-socket 32-core 2 GHz machine running Redhat Enterprise Linux 5 and OpenMPI 1.4.3. Two component sets were tested. The first is a Portals NIC-based simulation (Section 4.1) simulation which studies the advantages of NIC-based offload[27]. The second simulation uses a synthetic test component called `simpleComponent`.

The Portals simulation consists of three components: a cycle-approximate model of the Cray SeaStar router, a model of a Portals network API[20] offload network interface, and a state machine model of the driver application (in this case, the Allreduce algorithm). This setup stresses the scaling performance of the SST core because it is relatively memory intensive due to the high component count and highly communication intensive because there is little computation between synchronization points due to the high level of abstraction of the models. Between 8,192 and 262,144 simulated nodes were strong scaled on 1 to 32 shared memory MPI ranks on the host platform. Figure 1(a) shows the simulation runtimes, while Figure 1(b) shows the speedup compared to a single rank run. As can be seen, 32K simulated nodes provided the best scaling, achieving a speed up of 17.2 on 32 MPI ranks. All sizes saw better scaling with lower numbers of ranks, most likely due to limited memory bandwidth when using all shared memory cores. However,



(a) Triggered NIC Runtimes

(b) Triggered NIC Speedup



(c) SimpleComponent Runtimes

(d) SimpleComponent Speedup

1: Scaling Results

worst case scaling showed greater than an 11x speedup using 32 ranks. Interestingly, many of the runs showed superlinear scaling from 2 to 8 ranks, most likely due to cache and memory system effects.

The second simulation setup used a synthetic component named `simpleComponent`. This component emulates the computation and message characteristics of a more complex component. Each simulated cycle, `simpleComponent` will perform some work (simulated by a 6 instruction loop iterated over a configurable number of times) and may send a message to a random neighbor. Two configurations of `simpleComponent` were tested. The first was a "Mock SMP", representing a medium level processor core simulation, node which did a relatively large amount of work (4000 iterations per simulated cycle) and had a 1 in 50 chance of sending a 100 byte message per cycle. The second was a communications test which did relatively little work per cycle (100 iterations) and had a 1 in 25 chance of sending a 100 byte message. Each configuration was tested with 256, 4,096, and 16,384 components.

With only 256 components, both `simpleComponent` configurations failed to scale much beyond 16 ranks. However, with more components, they were able to scale effectively. With 16K components the "Mock SMP" was able to achieve a speedup of 15.79 on 16 ranks and 31.41 on 32 ranks (98% parallel efficiency) and even the more communication intensive test achieved speedups of 14.47 on 16 ranks and 26.75 on 32 ranks (84% parallel efficiency).

Future scalability studies will focus on distributed memory clusters and MPPs. Initial work shows that these system have higher overhead, but scale well for detailed simulations where there is a larger amount of computation to be performed between synchronizations.

4. IMPROVEMENTS

4.1 State machine driven Portals 4 NIC

The Portals 4 NIC component models a network interface which offloads the progress engine for the Portals 4 API[20]. The model is written at a medium level of detail, which is accurate at the block level, including computational units and buffering. The NIC supports most of the Portals 4 API, but is missing support for some of the less used options. The Portals 4 NIC is currently designed to work with the SeaStar router model, which is cycle approximate, and a state machine model of the driver application, which primarily accounts for the time required to communicate across the host interface to the NIC. Future work will expand the NIC model to work with the gem5 SST component (see Section 4.6), which provides a cycle approximate CPU simulation model and currently uses a simpler Portals NIC model. The SeaStar router has been validated in previous work[26] to within 7% of hardware performance.

4.2 IRIS

The IRIS Network simulator provides a pipelined, cycle-accurate router model capable of modeling a variety of Network-on-Chip (NoC) and inter-node interconnection architectures. The key IRIS component is a generic router, based on a generic input buffered router described in [6]. IRIS has been coupled with the Portals NIC model (Section 4.1), and will be used to study network topologies, routing methods, and network protocol tradeoffs.

The model is parameterized and can be configured to support different buffer sizes, virtual channels, routing and arbitration algorithms. The router is comprised of several subcomponents: buffers, routing units, and arbiters:

- **Buffer**: The basic unit of storage in a buffer is a flit.

There is one buffer per port of the router. The buffer depth and number of virtual channels are configurable.

- **Route Computation (RC) Unit**: There is one RC unit per input port. The RC unit currently reads the destination from the head flit and specifies the output port and output virtual channel. It can also support adaptive routing schemes.Currently, mesh and tori are supported.
- **Arbiters**: Both virtual channel arbiters and switch arbiters are modeled on the request-grant methodology. In a given cycle all active messages in the router populate the requester matrix.The arbiter then issues grants, in a round robin fasion, depending on the number of output channel and ports availability. Requests that fail must re-request in the next cycle.

Additionally, flit level flow control is managed by the router outside of all the sub-components. Future work will support additional topologies, more complex request granting mechanisms, and more detailed modeling of the router crossbar. Current work is focusing on validation of the router model.

4.3 PhoenixSim

Over the past decade, silicon photonic interconnect networks have been shown in simulation to outperform electronic networks, providing higher bandwidth at lower power via wavelength division multiplexing (WDM) while also alleviating IO pin constraints with high bandwidth density silicon photonic channels. Designing such integrated photonic systems will require careful exploration of the network design space through simulation. PhoenixSim[4], developed at Columbia University, allows us to simulate electronic and photonic networks at both a physical and abstract level and evaluate our network designs based on performance, energy, and thermal metrics.PhoenixSim was originally built on top of the discrete event network simulator OMNeT++[28], but has been ported to the SST to take advantage of SST's scalability and interact with other SST components.

PhoenixSim provides models of the waveguides, modulators, detectors, filters, switches, lateral couplers, and lasers used in silicon photonic communication. The components models are highly parameterized and can be updated to reflect the rapidly evolving capabilities of photonic devices.

The modeled electronic models include: wires, multiplexors, routers, serializers/deserializers, virtual channel logic, and IO pads. Also included is an abstract application model which can test and evaluate network functionality by producing regular, stochastic, or traced network traffic.

To integrate PhoenixSim with SST, we have interfaced OMNeT++ with SST. This interface was achieved by removing the event queue from OMNeT++, and forwarding scheduled events to the SST event queue. This gives us:

1. The ability to design network architectures using the well-established OMNeT++ framework, namely OMNeT++'s network description language, the OMNeT++ parameter configuration system, and the OMNeT++ IDE.
2. The ability to interface existing PhoenixSim models and any other OMNeT++ project with SST.
3. A stable interface between PhoenixSim and SST

Integrating PhoenixSim with the SST environment allows us to expand our design space to include the application, processor, and memory models available through SST. The more detailed processor and memory models in SST, such as BOBSim (Section 3.5), gem5 (Section 3.7), or the Portals NIC (Section 3.2) can drive network traffic and orchestrate any higher-level network protocols, while leaving the modeling of the interconnection network and any other photonic components to PhoenixSim. With PhoenixSim integrated into SST, we can closely analyze the interaction between processors, a photonic interconnection network, and photonic links to memory as they operate in a cohesive system.

Because silicon photonic network devices currently exist as small scale laboratory demonstrators, full scale validation is difficult to perform. However, individual device models have been calibrated to match the measured parameters of fabricated devices.

4.4 BOBSim

Due to the signaling characteristics of JEDEC standard double-data rate (DDR) DRAM DIMMs, current memory systems are limited in speed and capabity [8]. To circumvent these limitations, the buffer-on-board (BOB) memory system has been introduced in HPC and server systems. Similar to FB-DIMM, the BOB memory system uses a small buffer or controller that grants both an increase in clock rate and increased signal integrity. The buffer is responsible for controlling the ranks of DDR DIMMs and communicating with the CPU over a fast and narrow bus. Because there is only one buffer chip per channel, the BOB approach alleviates many of the problems (i.e. high power dissipation, cost, and latency) that plagued the FB-DIMM architecture.

BOBSim is a hardware verified BOB memory system simulator developed at the University of Maryland and integrated with the SST. BOBSim is a cycle-based simulator that encapsulates the main BOB controller and each link bus, simple controller, and DRAM channel that make up the memory system. All of the major logical portions of the design have a corresponding software object and associated parameters that give total control over every aspects of the system's configuration and behavior. Device timing parameters and power modeling are based on Micron data-sheets for DDR3-1333 and DDR3-1600 DRAM parts. The BOB simulator has been hardware verified at the DRAM level similar to DRAMSim2 [22]. Micron Technology, Inc. publicly provides Verilog HDL models for all DRAM devices it manufactures. These HDL models are used in conjunction with ModelSIM to ensure correct timing of commands and data on each bus. Like DRAMSim2, BOBSim provides an easy to use and accurate memory model for address stream or system performance analysis while still being able to model the most cutting edge memory architectures.

BOBSim has been tested with the SST's `genericProc` processor model, and also with the SST's port of gem5 (Section 4.6). To integrate with gem5, a small wrapper component was built around the BOBSim interface to provide backing store for the data in DRAM, as by default BOBSim only provides timing information.

4.5 MacSim GPGPU Simulator

The MacSim simulator provides a model of GPU/CPU cores or heterogeneous computing nodes. MacSim is a trace driven heterogeneous architecture simulator that can simulate x86 and PTX traces (CUDA source code). It can simulate one of the ISAs (homogeneous node) or both (hetero-

geneous node). Unlike MacSim, GPGPU-Sim [1] simulates only PTX traces. Recently released Multi2Sim [25] also simulates heterogeneous architectures (X86 and OpenCL).

Macsim can be driven by PTX or X86 traces. PTX Traces are generated by Ocelot [5] and x86 traces by Pin [16]. PTX traces include branch divergence information as bit masks and all thread memory access addresses. X86 traces include x86 instructions with decoded information. Both X86 and PTX traces are translated into MacSim micro-ops, which are RISC style ops. For the PTX case, almost all ops are directly translated into micro-ops. For x86 traces,typically 1 to 3 micro-ops are generated per X86 instruction. MacSim models in-order and out-of-order pipelines, SMT features, caches, and includes a simple DRAM model.

When integrated with SST, MacSim can represent (a) a heterogeneous node, (b) CPU/GPU processors (c) one core in a GPU or CPU. For the heterogeneous node case, SST only sends timing events and application starting events. For a CPU/GPU model, MacSim sends memory packets to the SST. In this case, memory and interconnection components are simulated using other components in SST, such as BOBSim (Section 4.4) or IRIS (Section 4.2). Finally, when MacSim models only one GPU/CPU core, MacSim sends all the memory traffic, including cache accesses, to other SST components.

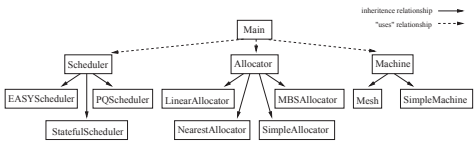
MacSim can be used to explore the performance of GPGPU style computation. Integrated with SST, it can be used to examine the effects of different memory arrangements, and the tradeoffs in integrating a GPGPU with a more conventional processor or using a GPGPU as a standalone compute node. Leveraging SST's network components will allow the data movement issues different systems to be evaluated.

4.6 gem5

Like SST, the gem5 simulator[2] is a modular framework for architectural research. It includes a number of simulation structures for processors, caches, busses, and TCP/IP network components. Unlike SST, its focus is on smaller scale systems using commodity networks – not HPC. However, its detailed processor models with support for multiple ISAs and full system simulation are a valuable addition to the SST's set of components.

Integrating gem5 into SST required several changes. First, gem5's Python-based initialization was replaced with an XML-based configuration to better support SST's two phase initialization process (where components are partitioned across the parallel job before being instantiated). Secondly, gem5 was encapsulated as an SST component, and its internal event queue modified to be driven by SST's. To allow communication with SST, translator `SimObject`s were created which pass gem5 events to SST components through the normal SST inter-component `link` objects. Lastly, changes were made to the gem5 loader to avoid the use of synchronous (untimed out-of-band) messages. This was required because gem5 used "backdoors" between `SimObject`s in the same address space. In the distributed memory SST these "backdoors" cannot be used.

Once this basic integration was complete, gem5 was tested with DRAMSim and BOBSim (Section 4.4) as its main memory model. Additionally, gem5 has been integrated with a Portals NIC (similar to that described in section 4.1), to allow larger scale simulations of parallel distributed memory machines. This has been used to simulate up to 512



2: Simplified class diagram for standalone scheduling/allocation simulator

instances of gem5/SST all running a parallel MPI program.

Previous work[23] has validated gem5's processor model to within 15% of actual hardware performance for complex benchmarks, mainly due to inaccuracies in its relativelty simple internal memory model. It is expected that the more detailed DRAMSim/BOBSim memory models in SST will correct much of this error and validation is ongoing.

4.7 Scheduling Simulation

The *scheduler* component simulates the placement and scheduling of jobs onto a simulated system. This component is based on an existing standalone HPC simulator[14, 24] which allows exploration of when jobs run (scheduling), and which processors they are assigned to (allocation) [29]. Porting this simulator into the SST framework provides realistic workloads for low-level simulations and will benefit research on scheduling and allocation with low-level details on how jobs in a system interact.

The simulator parses a trace derived from the Parallel Workloads Archive [7] for characteristics of the jobs to run, and manages an stream of job arrivals and completions. It uses objects from the abstract classes `Scheduler`, `Allocator`, and `Machine` to store most of the simulation state and to make decisions with derived classes implementing different versions of the necessary functionality (See Figure 2). Common schedulers and allocators have been implemented for use as comparison baselines. These include include EASY [13] (`EASYScheduler`), a generalization of Conservative [18] (`StatefulScheduler`), and a priority-based scheduler (`PQScheduler`). The latter uses a set of comparators to select between FCFS, Shortest Job First, Widest Job First, and similar priority-based schemes. Most of the allocators are meant for mesh systems, with classes derived from `Allocator` implementing allocators based on linear orderings [15, 12, 29] (`LinearAllocator`), center-based allocators [17, 11] (`NearestAllocator`), and buddy systems [15, 29] (`MBSAllocator`). In addition, to speed up simulations where scheduling is the focus and allocation is unnecessary, the system provides a "bag of processors" with no notion of locality (`SimpleMachine`) along with the appropriate allocator (`SimpleAllocator`).

To integrate with the SST, a new class was added to represent the scheduling components and to interface with the other SST components. Starting jobs and learning of their completion now involves interacting with *node* components, one for each node in the system. The Allocator classes notify the interfacing class, which sends the appropriate nodes an SST message to begin processing. As each node completes its part of a job, it sends an SST event message back to the interfacing class, which gathers these until all nodes assigned to the job have completed, at which point the rest of the scheduler is told of these nodes' availability. Returning the nodes to availability all together mimics the behavior of real systems and, fortunately, minimized the changes necessary to the existing scheduler.

Future work will focus on the node component. Currently, the node simulates job duration as a fixed static interval, neglecting intra- and inter-job interactions. We are now starting to add communication behavior to the nodes, which will use networking components such as IRIS or PhoenixSim. Eventually, processor and memory models such as BOBSim, MacSim, and gem5 will be added. This will allow exploration of topology aware allocation algorithms where network interference effects can be better understood.

4.8 Reliability Simulation

As high-end computing machines continue to grow in size, failure rates have begun to limit application scalability. Current techniques to ensure progress across faults, like coordinated checkpoint-restart (CCR), are increasingly problematic at these scales due to high overheads.Many alternative resilience algorithms and fault-tolerance mechanisms have been proposed, however, few of these techniques have been accurately evaluated at large scale.With the reliability simulation components, we extend the SST framework for simulating the resilience mechanisms of a large-scale system, giving us the ability to run relative performance comparisons of more than a million cores while adjusting a wide range of system parameters.

The reliability simulation components include a **Router**, a **Storage** component, and an **End-point** component.

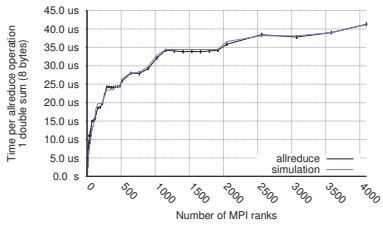
The **Router** component can be connected into 1-D, 2-D, or 3-D meshes or tori. They use source-based routing and wormhole-based routes. This is typical of supercomputer networks. Message latency is calculated simply based on a configurable router latency, port contention, and the length of the message. We use the same basic router model for the system-level interconnect and the NoC within each socket of our simulation.

Evaluating resilience algorithms requires simulation of storage access. For our work two types of **Storage** are be simulated. In each node there is some amount of NVRAM that is accessible to the local cores and nodes can access remote storage. Each storage component accepts data write requests and queues them. Based on a configurable write speed parameter, this queue is processed and write acknowledgements are sent back to the requester. There is a similar queue for read requests and read data events are sent back to requesters based on the read speed of the device and the number and size of pending requests.

Due to the cost, we cannot afford to run a detailed execution-based processor model at each **end-point**. So, we generate application events with pattern generators. For example, a five-point stencil computation using ghost (halo) cells to exchange data with neighboring ranks has a simple loop structure: Send data to neighbors, wait for data from neighbors, perform computation, repeat. A collective operation, such as an allreduce, is occasionally inserted to determine convergence. Many similar patterns exist in current applications. Our simulator is capable of producing any communication pattern as long as the pattern is not based on data organization.

These communication patterns are represented by state machines. However, describing computation as state machines can be difficult due to the state explosion that occurs when we combine a communication pattern generator, a resilience algorithm, collective operations, and the handling of asynchronous I/O events and faults. The solution we have chosen is that of a gate keeper. This is a C++ SST component from which all communication patterns inherit. This gate keeper ensures all events arrive in proper order and queues events for later retrieval which arrive too early.

Validation of this work has only begun, but, early results, such as the one in Figure 3 look very promising. That figure shows the results of a hand-coded allreduce operation using a one-double sum on up to 4,096 cores of a Cray XE-6 system. We simulated the same algorithm and configured SST to mimic the XE system and obtained the results in Figure 3.



3: One double sum allreduce on a Cray XE-6 system

5. SUMMARY

The SST provides a parallel, scalable, modular, and open framework for architectural simulation of HPC systems. Recent improvements have enhanced its scalability and added functionality in network, memory, and processor simulation.

The SST will continue to expand and improve. A number of enhancements are planned, in particular, improving component interoperability will be emphasized. Component and inter-component validation will also be a major task, with particular emphasis on the error analysis of mixed resolution simulation. Though the scaling results presented in this paper are for a small shared memory machine, optimization for distributed memory clusters will be a priority. The long-term goal of the SST is to target advanced architectures aimed at exascale-class supercomputers.

Acknowledgment

Sandia National Laboratories is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy's National Nuclear Security Administration under contract DE-AC04-94AL85000.

6. ADDITIONAL AUTHORS

Additional authors: Brian Barrett (Sandian National Labs, bwbarre@sandia.gov), Cassandra Versaggi (Knox College, cversagg@knox.edu), Robert Hendry (Columbia University, rh2519@columbia.edu), Bruce Jacob (University of Maryland, blj@umd.edu), Hyesoon Kim (Georgia Tech, hyesoon@cc.gatech.edu), Vitus Leung (Sandia National Labs, vjleung@sandia.gov), Michael Levenhagen (Sandia National Labs, mjleven@sandia.gov), Mitchelle Rasquinha (Georgia Tech, mitchelle.rasquinha@gatech.edu), Rolf Riesen (IBM, rolf.riesen@ie.ibm.com), Paul Rosenfeld (U. Maryland, prosenf1@umd.edu), Maria del Carmen Ruiz Varela (University of Delaware, maria.ruiz.varela@gmail.com), Sudhakar Yalamanchili (Georgia Tech, sudha@ece.gatech.edu)

7. REFERENCES

[1] A. Bakhoda, G. Yuan, W. Fung, H. Wong, and T. Aamodt. Analyzing cuda workloads using a detailed gpu simulator. In *Performance Analysis of Systems and Software, 2009. ISPASS 2009. IEEE International Symposium on*, pages 163 –174, april 2009.

[2] N. L. Binkert, R. G. Dreslinski, L. R. Hsu, K. T. Lim, A. G. Saidi, and S. K. Reinhardt. The m5 simulator: Modeling networked systems. *IEEE Micro*, 26:52–60, 2006.

[3] D. Burger and T. Austin. *The SimpleScalar Tool Set, Version 2.0*. SimpleScalar LLC.

[4] J. Chan, G. Hendry, A. Biberman, K. Bergman, and L. P. Carloni. Phoenixsim: a simulator for physical-layer analysis of chip-scale photonic interconnection networks. In *Proc. of the Conference on Design, Automation and Test in Europe*, DATE '10, pages 691–696, 2010.

[5] G. Diamos, A. Kerr, S. Yalamanchili, and N. Clark. Ocelot: A dynamic compiler for bulk-synchronous applications in heterogeneo us systems. In *PACT-19*, pages 353–364, New York, NY, USA, 2010. ACM.

[6] J. Duato, S. Yalamanchili, and N. Lionel. *Interconnection Networks: An Engineering Approach*. Morgan Kaufmann Publishers Inc., San Franciso, CA, USA, 2002.

[7] D. Feitelson. The parallel workloads archive. http://www.cs.huji.ac.il/labs/parallel/workload/index.html.

[8] H. Fredriksson and C. Svensson. Improvement potential and equalization example for multidrop dram memory buses. *IEEE Transaction On Advanced Packaging*, 32(3):675–682, 2009.

[9] R. M. Fujimoto. Parallel discrete event simulation. In *Proceedings of the 21st conference on Winter simulation*, WSC '89, pages 19–28, New York, NY, USA, 1989. ACM.

[10] T. Henderson, , T. R. Henderson, and S. Roy. ns-3 project goals.

[11] S. Krumke, M. Marathe, H. Noltemeier, V. Radhakrishnan, S. Ravi, and D. Rosenkrantz. Compact location problems. *Theoretical Computer Science*, 181(2):379–404, 1997.

[12] V. Leung, E. Arkin, M. Bender, D. Bunde, J. Johnston, A. Lal, J. Mitchell, C. Phillips, and S. Seiden. Processor allocation on Cplant: Achieving general processor locality using one-dimensional allocation strategies. In *Proc. 4th IEEE Intern. Conf. on Cluster Computing*, pages 296–304, 2002.

[13] D. Lifka. The ANL/IBM SP scheduling system. In *Proc. 1st Workshop Job Scheduling Strategies for Parallel Processing*, number 949 in LNCS, pages 295–303, 1995.

[14] A. Lindsay, M. Galloway-Carson, C. Johnson, D. Bunde, and V. Leung. Backfilling with guarantees granted upon job submission. In *Proc. 17th Intern. Euro-Par Conf. Parallel Processing*, number 6852 in LNCS, pages 142–153, 2011.

[15] V. Lo, K. Windisch, W. Liu, and B. Nitzberg. Non-contiguous processor allocation algorithms for mesh-connected multicomputers. *IEEE Trans. Parallel and Distributed Systems*, 8(7):712–726, 1997.

[16] C.-K. Luk, R. Cohn, R. Muth, H. Patil, A. Klauser, G. Lowney, S. Wallace, V. J. Reddi, and K. Hazelwood. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[17] J. Mache, V. Lo, and K. Windisch. Minimizing message-passing contention in fragmentation-free processor allocation. In *Proc. 10th IASTED Intern. Conf. Parallel and Distributed Computing and Systems*, pages 120–124, 1997.

[18] A. W. Mu'alem and D. G. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the IBM SP2 with backfilling. *IEEE Trans. Parallel and Distributed Syst.*, 12(6):529–543, 2001.

[19] D. Nellans, V. K. Kadaru, and E. Brunv. Asim- an asynchronous architectural level simulator abstract.

[20] R. E. Riesen, K. T. Pedretti, R. Brightwell, B. W. Barrett, K. D. Underwood, T. B. Hudson, and A. B. Maccabe. The Portals 4.0 message passing interface. Technical Report SAND2008-2639, Sandia National Laboratories, April 2008.

[21] A. F. Rodrigues, K. S. Hemmert, B. W. Barrett, C. Kersey, R. Oldfield, M. Weston, R. Risen, J. Cook, P. Rosenfeld, E. CooperBalls, and B. Jacob. The structural simulation toolkit. *SIGMETRICS Perform. Eval. Rev.*, 38:37–42, March 2011.

[22] P. Rosenfeld, E. Cooper-Balis, and B. Jacob. Dramsim2: A cycle accurate memory system simulator. *IEEE Computer Architecture Letters,*, 99(RapidPosts), 2011.

[23] A. G. Saidi, N. L. Binkert, L. R. Hsu, and S. K. Reinhardt. Performance validation of network-intensive workloads on a full-system simulator. In *First Ann. Workshop on Iteraction between Operating System and Computer Architecture (IOSCA)*, Oct. 2005.

[24] O. Thebe, D. Bunde, and V. Leung. Scheduling restartable jobs with short test runs. In *Proc. 14th Workshop Job Scheduling Strategies for Parallel Processing*, 2009.

[25] R. Ubal, J. Sahuquillo, S. Petit, and P. López. Multi2Sim: A Simulation Framework to Evaluate Multicore-Multithreaded Processors. In *Proc. of the 19th Int'l Symposium on Computer Architecture and High Performance Computing*, Oct. 2007.

[26] K. Underwood, M. Levenhagen, and A. Rodrigues. Simulating red storm: Challenges and successes in building a system simulation. In *IEEE International Parallel and Distributed Processing Symposium*, Long Beach, CA, 2007. IEEE.

[27] K. D. Underwood, J. Coffman, R. Larsen, K. S. Hemmert, B. W. Barrett, R. Brightwell, and M. Levenhagen. Enabling flexible collective communication offload with triggered operations. In *Proceedings of 19th Annual Symposium on High-Performance Interconnects (HotI)*, August 2011.

[28] A. Varga. Omnet++ discrete event simulation system. http://www.omnetpp.org., 2011.

[29] P. Walker, D. Bunde, and V. Leung. Faster high-quality processor allocation. In *Proc. 11th LCI Intern. Conf. High-Performance Clustered Computing*, 2010.