Instruction-Level Simulation of a Cluster at Scale

Conference Paper · January 2009

DOI: 10.1145/1654059.1654063 · Source: DBLP

4 authors:

Edgar A. Leon
Lawrence Livermore National Laboratory
31 PUBLICATIONS **66** CITATIONS
SEE PROFILE

Rolf Riesen
Intel
94 PUBLICATIONS **1,148** CITATIONS
SEE PROFILE

Arthur B. Maccabe
Oak Ridge National Laboratory
112 PUBLICATIONS **1,667** CITATIONS
SEE PROFILE

Patrick G. Bridges
University of New Mexico
93 PUBLICATIONS **1,049** CITATIONS
SEE PROFILE

Instruction-Level Simulation of a Cluster at Scale

Edgar A. León*
University of New Mexico
leon@cs.unm.edu

Rolf Riesen†
Sandia National Laboratories
rolf@sandia.gov

Arthur B. Maccabe
Oak Ridge National Laboratory
maccabeab@ornl.gov

Patrick G. Bridges
University of New Mexico
bridges@cs.unm.edu

ABSTRACT

Instruction-level simulation is necessary to evaluate new architectures. However, single-node simulation cannot predict the behavior of a parallel application on a supercomputer. We present a scalable simulator that couples a cycle-accurate node simulator with a supercomputer network model. Our simulator executes individual instances of IBM's Mambo PowerPC simulator on hundreds of cores. We integrated a NIC emulator into Mambo and model the network instead of fully simulating it. This decouples the individual node simulators and makes our design scalable.

Our simulator runs unmodified parallel message-passing applications on hundreds of nodes. We can change network and detailed node parameters, inject network traffic directly into caches, and use different policies to decide when that is an advantage.

This paper describes our simulator in detail, evaluates it, and demonstrates its scalability. We show its suitability for architecture research by evaluating the impact of cache injection on parallel application performance.

1. INTRODUCTION

Future clusters and supercomputers require architectural changes in order to grow to the enormous sizes and speeds required by high-end parallel applications. Proposed changes include techniques for injecting incoming messages directly into processor caches [4, 18, 21]. Unfortunately, the impact of such changes on application performance is difficult to predict analytically due to the complex interactions between the architecture, operating system, system libraries, and the applications.

Architectural simulators that examine the impact of low-level system changes on application performance have not historically scaled well [11, 12]. For example, coarse-grained simulators skew dramatically when these changes are scaled up over tens or hundreds of systems. On the other hand, cycle-accurate simulators, which can accurately simulate every event to nanosecond accuracy in a single system, scale up poorly. Their running time increases dramatically even for a small number of processors. This limits designers in their ability to study how architectural changes affect scientific application performance as a cluster grows in scale.

To address this problem, we present an MPI-based cluster simulator designed to enable studies of architecture, operating system, and application interactions for current and future architectures. Unlike previous work, our cluster simulator is a parallel program that uses existing clusters to simulate future clusters. It does that by coupling a cycle-accurate node simulator with a network model.

The contribution of this paper is the design and implementation of a scalable cluster simulator that can be used to analyze the impact of novel architectural features on parallel application performance. We describe the node simulator and the design of our network and NIC model in Section 2 and evaluate the resulting cluster simulator in Sections 3 and 4. We analyze three cache injection policies to demonstrate the capabilities of our simulator in Section 5.

2. CLUSTER SIMULATOR

To study the performance impact of future architectures on scientific applications at scale, we built a flexible apparatus to simulate an entire cluster. The apparatus is based on an existing cycle-accurate simulator and leverages the parallel computation capabilities of in-house clusters to simulate future parallel architectures; or existing systems with different performance characteristics.

Our cluster simulation infrastructure consists of the following components: a multiprocessor simulator on each node of the host cluster, an OS and communication libraries for the simulator, a modeled NIC, a modeled network, a shim layer to connect the NIC model to the node simulator, and a runtime environment to launch applications inside the simulated (target) cluster. Figure **??** shows all of the components.

We will use *host node* when we refer to a physical host within the real cluster, and *target node* when we refer to a simulated node. *NIC* refers to a modeled NIC that connects

*This work was partially supported by an Intel fellowship and an IBM grant under subcontract from IBM on DARPA contract NBCH30390004.
†Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed Martin Company, for the United States Department of Energy under contract DE-AC04-94AL85000.

SC09 November 14-20, 2009, Portland, Oregon, USA

target nodes within the simulated cluster. In a similar vein, *simulation* or *simulated* time refers to the time frame an application experiences when it runs inside the simulator. This is the time frame an application uses to report performance results. *Wall-clock* time is the time observed outside the simulation; i.e., the actual time it takes to run a simulation experiment.

Our simulated cluster is launched on a host cluster by executing one instance of a target node per available core. The target nodes communicate with each other over a modeled network through a modeled NIC. The NIC serves as a bridge between a simulated target node and a host node. The NIC model uses the existing MPI implementation on the host cluster to communicate with NICs running on other host nodes or cores. We chose MPI instead of TCP/IP as the transport layer for two reasons. MPI is available on all clusters and on massively parallel systems, such as the Cray XT-3 Red Storm running the Catamount compute node OS, where TCP/IP is not available, or does not scale. Additionally, the MIAMI API described below maps particularly well onto the MPI API. The TCP/IP API has no message matching capability and would be harder to use.

Figure 1 illustrates the architecture of our apparatus. On each host node of a cluster we launch one or more instances of IBM's Mambo simulator. Each Mambo instance runs the K42 operating system. The application under test uses the MPICH [15] implementation of MPI to communicate with the simulated target system. This version of MPICH uses a device layer we call MIAMI to interact with its peers on other nodes. It does that by interacting with the local NIC model which, in turn, uses the host system's OpenMPI [13] layer to exchange messages with other nodes on behalf of the simulated application.

Using a full-featured, cycle-accurate simulator to simulate a multiprocessor target node provides us with a platform to study architectural features and configurations not yet available. Our goal in developing this cluster simulation infrastructure is to leverage existing single-node simulators in a cluster setting.

2.1 Mambo

Our cluster simulator uses an augmented version of IBM's Mambo full-system simulator [22] that provides cache injection of incoming network messages [4]. Mambo is called a full-system simulator because it simulates all the components of a node: CPU, caches, memory, and the buses that connect them. Each simulated target node is a multi-core, cache-coherent, distributed shared memory system [27]. We run the K42 research operating system [1] on Mambo.

We launch one instance of Mambo per available core on our host cluster. Simulation of systems larger than that is possible, as long as enough physical memory is available. Of course, the time to run an experiment increases with each additional instance of Mambo that we run on a core.

One of the key features of a cycle-accurate simulator is that many of its configuration parameters can be changed. Mambo is no exception and we chose the values in Table 1 for the experiments presented in this paper.

2.1.1 Fast-forward mode

Cycle-accurate simulation is expensive in terms of time. Slowdown factors of several orders of magnitude are common. Mambo spends a significant portion of its time simulating the operation of caches. Cache simulation can be turned off in Mambo to make it run faster. The simulated target node then behaves as if no caches were present and it accesses main memory at L1 cache speeds. For many experiments that is a useful feature that does not impact the end result. For example, in studies that focus on network characteristics, it may not be important how fast the application executes an inner compute kernel.

Table 1: Simulated system configuration.

Feature	Configuration
Simulator	Mambo PowerPC full-system simulator
Architecture	Power5 with cache injection to L2/L3
Processor	1.65 *GHz* frequency
L1 I/D cache	64 *kB*/32 *kB*, 2-way/4-way
L2 cache	1.875 *MB*, 3-slice, 10-way, 10 cycle latency
L3 cache	36 *MB*, 3-slice, 12-way, 80 cycle latency
Cache line	128 *B*
Main memory	1,024 *MB*, 230 cycle latency
OS	K42
Comm. Lib.	MPICH-MIAMI w/OS-bypass & 0-copy
Network	Cray XT-3 Red Storm

However, for accurate simulations, cache simulation needs to be on. We have added the ability to fast-forward in our cluster simulation infrastructure. During application startup, for example, we would like the simulation to proceed as quickly as possible. When execution reaches the inner kernel, we activate the cache simulation. In Section 4.3 we will demonstrate this feature. We run through the application setup phase with cache simulation disabled and turn it on before we reach the computational kernel we are interested in evaluating. We turn on cache simulation early enough to let the caches warm up.

Fast-forward can also be used while an application restarts from an earlier checkpoint. While reading the restart data and initializing, cache simulation is off. Once the calculation resumes, we enable cache simulation. This feature enables us to simulate longer application runs in less time.

For many of the experiments in Section 4 we ran with cache simulation turned off, since for those experiments we were not interested in the actual simulation results from Mambo. The experiments in Section 5 were done with cache simulation turned on. We have implemented fast-forward mode for the AMG application and use it for all AMG experiments in this paper.

2.1.2 Interfacing Mambo with the NIC

A target node and its associated NIC interact through a *shim layer*. This layer provides a bidirectional path between the simulated target node and the NIC. On one side of the shim layer, a target node communicates with its NIC through memory mapped registers. Using this mechanism, a user-level process can interact with the NIC directly, bypassing the OS. Access to these registers is controlled by the OS. On the other side of the shim layer, a NIC communicates with its target node through a well-defined *shim interface*. The operations provided by the shim interface allow reads and writes from the NIC to main memory and the L2 and L3 caches. The interface also allows the NIC to raise interrupts, delay cycles on the host, and launch processes on the

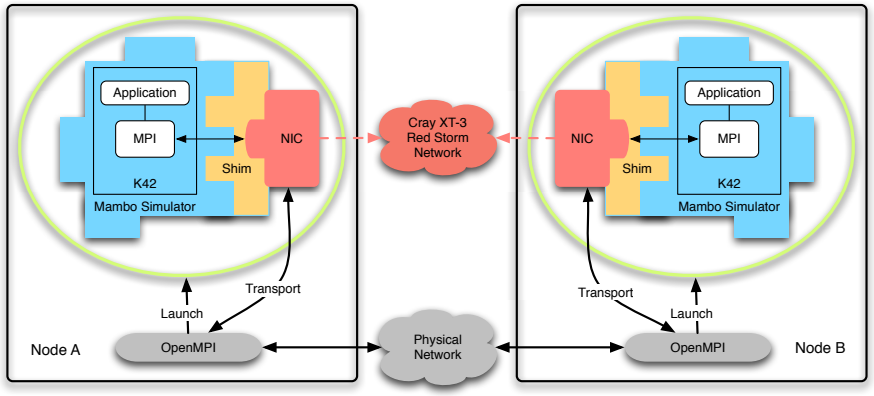

Figure 1: Two simulated target nodes communicating with each other through the host cluster's network.

host. In addition, the shim layer provides functions to load and unload NICs and other devices at run time.

2.2 The network and NIC model

A NIC connects a local target node with the rest of the target nodes running on the host cluster. It acts as a bridge between its target node and the host node on which it is running. The NIC uses the host node's transport layer (Open-MPI in our case) to communicate with NICs running on other nodes or cores of the host system.

Each message sent between simulated target nodes is augmented by the NIC with a time stamp and delay information about the modeled network. The NIC then sends that information with the actual data using the transport layer provided by the host cluster. The receiving NIC waits to deliver the message to the simulated target node until its simulation clock reaches the message's time stamp plus the modeled network delay.

This is possible because the network of the host cluster appears lightning fast in comparison to the slow-running simulated target nodes. The NICs can deliver messages at practically any latency and bandwidth in simulated time. This allows us to present the simulated target nodes with any type of network (physically possible or not) of our choice.

2.2.1 Synchronization

The clocks on the nodes of a typical cluster are not synchronized and drift over time. Furthermore, node performance in a cluster varies and impacts the performance of the simulator. Therefore, we need to synchronize the simulated target nodes occasionally so that none of them get too far ahead of the others in the system.

We accomplish this synchronization by executing a barrier operation at specific intervals of the simulation clock. For example, if the synchronization interval is set at 50,000, then the simulated target nodes will execute the barrier every time they have executed another 50,000 cycles. While the target nodes wait for each other in the barrier, the local simulation clocks are stopped.

The delay caused by the barrier is not visible to the simulated application. However, the barrier has externally visible effects. If one of the target node simulators is slow in reaching the barrier, because it writes a lengthy trace file to an external disk for example, then all the simulated target nodes are delayed. In other words, the overall execution time of our cluster simulator represents the slowest path through all instances of the individual target node simulators. We will evaluate the impact of synchronization in Section 4.

2.2.2 Network model

We use the network model that is part of Seshat [24] and has the characteristics of a Cray XT-3 Red Storm network to decide when to deliver messages. Seshat is an execution-driven discrete-event simulator to study application behavior under varying network characteristics.

The current network model does not take topology into consideration. In essence, we are modeling a fully connected network graph. Therefore it is not capable of emulating congestion. Two configuration parameters allow us to vary the bandwidth and latency of the model which enables us to simulate faster or slower networks.

A slightly more sophisticated model would take the bisection bandwidth of the network into consideration and delay messages that are injected into a "full" network. The next level up would be a model that incorporates knowledge about the network topology. Up to that level of detail, the network model could still be distributed and run sufficiently fast for our simulator to scale. A fully simulated network would require more frequent synchronization and would slow down simulator performance by orders of magnitude. The tradeoffs between accuracy and performance of network models have been studied in [5].

2.2.3 NIC model

To run message-passing applications on our simulated cluster, we developed a minimal API to support MPI: *MIAMI* – Minimal Interface for An MPI Implementation. The asynchronous MIAMI API is shown in Table 2. It supports OS-bypass and allows for zero-copy MPI transfers.

We implemented a driver in our modeled NIC for the eleven-function MIAMI API. We also created a corresponding device layer in the MPICH implementation of MPI that runs inside the simulator. Figure 2 shows how a simulated application interacts with the local NIC. The application makes calls into the MPICH library which uses the MIAMI API to interact with the NIC model. Most MIAMI functions in the NIC are implemented as straight forward calls into the host system's transport layer. In our case, that layer

Table 2: MIAMI API

Function	Description
int **init**(void);	initialize
int **finalize**(void);	clean up
int **size**(void);	number of processes in job
int **rank**(void);	my rank in job
double **clock**(void);	time in seconds
int **tx_start**(void **buf*, int *len*, int *cntxt*, int *tag*, int *dst*, int *lsrc*);	start a send
int **stx_start**(void **buf*, int *len*, int *cntxt*, int *tag*, int *dst*, int *lsrc*);	synchronous send
int **tx_done**(int *handle*);	check send completion
int **rx_start**(void **buf*, int *len*, int *cntxt*, int *tag*, int *src*);	post a receive
int **rx_done**(int *handle*, int **len*, int **tag*, int **src*);	check receive completion
int **rx_probe**(int **flag*, int **len*, int *cntxt*, int **tag*, int **src*);	probe for message arrival

is OpenMPI running on the host cluster.

Several factors complicate the implementation of MIAMI in our NIC. Our simulation infrastructure is in itself a complete parallel application and its behavior must conform to the MPI standard. For example, the infrastructure might be in the middle of a point-to-point transfer on behalf of the simulated application when Mambo's clock reaches a synchronization point and forces a barrier. Other nodes may still be sending or waiting for messages before they reach their barriers. It is important not to cause deadlock or violate the rules of the MPI standard in such situations. For that reason, we implemented our synchronization barrier as a series of point-to-point messages and make sure that synchronizations continue until all nodes have reached `MPI_Finalize()`.

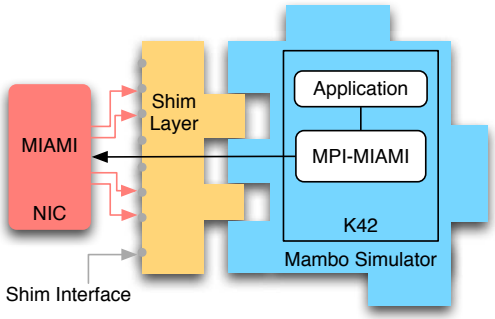

Figure 2: An application on a simulated target node interacting with the local NIC.

Another complicating factor is that we have to send envelope information along with each application message. We use information inside that envelope to decide when, in simulated time, to deliver a particular message. To examine the envelope information, we have to receive the message into a buffer. This is difficult when messages arrive unexpectedly and the simulated application does not post a receive until much later. The NIC has to perform careful buffer management and message matching while maintaining good performance and scalability.

3. EXPERIMENTAL SETUP

We ran our experiments on two large clusters at Sandia National Laboratories. In this section we briefly describe the characteristics of these machines, and list the benchmarks and applications we used to evaluate our infrastructure.

3.1 Test platforms

Thunderbird is comprised of 4,480 compute nodes. They are dual 3.6 *GHz* Intel EM64T processors with 6 *GB* of RAM. The network is an Infiniband with a two level Clos topology. The nodes run Red Hat Enterprise Linux with a 2.6.9 kernel and use Lustre as the parallel file system. We use the version 1.2.7 OpenMPI library and the version 1.3.1 OFED library to connect to the Infiniband fabric.

Spirit has 512 nodes and each has dual 3.4 *GHz* Intel EM64T processors with 2 *GB* of RAM and it uses a Myrinet network. It runs Red Hat Enterprise Linux with a 2.6.9 kernel and OpenMPI version 1.2.2, which uses version 1.2.7 of the MX library for the Myrinet.

Both of these systems have dual 64-bit CPU nodes. We run the 32-bit versions of the above mentioned libraries so we can link them with the 32-bit executable of the Mambo simulator. We do not have access to the Mambo source code.

To achieve the best performance, we run one instance of Mambo per core of the host machine. For smaller problem sizes that do not require too much memory, it would be possible to run multiple instances of Mambo on each core, thereby simulating a much larger cluster than the size of our host machines. However, wall-clock time would increase due to context switch overheads and the additional workload on each core.

3.2 Benchmarks and applications

We chose five different benchmarks and applications to test and evaluate our simulator.

3.2.1 IS from the NAS parallel benchmark suite

IS is the well-known integer sort benchmark from the NAS parallel benchmark suite [8]. We used version 2.4 for our experiments and chose IS because it has a very short run time. This makes IS very suitable for cycle-accurate simulations which take many hundreds to thousands of times longer to finish when being simulated.

The second reason we chose IS is that it is a C code. We currently do not have a Fortran cross compiler available for our test environment. This prevents us from running the remaining NAS parallel benchmarks which are written in Fortran. IS is a strong-scaling benchmark; i.e., the aggregate

work load is fixed, independent of the number of nodes used to solve the problem.

3.2.2 AMG from the Sequoia acceptance suite

AMG is an "algebraic multigrid solver for linear systems arising from problems on unstructured grids" [20]. It is one of several benchmarks used by Lawrence Livermore National Laboratory (LLNL) in its request for proposals and acceptance of the Sequoia supercomputer.

We chose AMG because it is a communication intensive application which can, for large problem sizes, spend 90% of its execution time inside MPI. AMG uses mostly MPI collective operations. A small percentage of communications are point-to-point messages of relatively small size (2 – 10 kB) [20].

AMG contains different solvers which can be selected from the command line. The data presented in this paper is from solver 0 (the default) and solver 1 runs. We chose these two solvers because they seem to place more demand on the memory subsystem than the other solvers. Cache injection should benefit these solvers more than the others available in AMG.

AMG operates in three distinct phases. The solver runs in the third phase, while the first two phases are used for problem setup. We augmented AMG so it runs in fast forward mode during most of its setup and initialization and turn on cache simulation before we enter the solve phase. Turning cache simulation on a little early is necessary to let the caches warm up before we enter the solve phase. We ran AMG in weak-scaling mode where the problem size increases with additional nodes used; i.e. the work per node remains constant.

3.2.3 LAMMPS from the Sequoia acceptance suite

LAMMPS [23] is a classical molecular dynamics code developed at Sandia. For our experiments we use the embedded atom method (EAM) metallic solid input script which is used by the Sequoia benchmark suite. The LAMMPS code and input scripts are provided on the LAMMPS web site [25]. For each experiment we mention whether we ran LAMMPS in weak or strong-scaling mode.

3.2.4 FFTW

FFTW (Fastest Fourier Transform in the West) is a C library for computing the discrete Fourier transform (DFT) in one or more dimensions [10]. We use the MPI parallel FFTW version 2.1.5 (MPI transforms are available only in this version). FFTW allows the computation of different types of transforms, including normal and transpose order, and with and without work space. The output data computed by these transforms maintain the same ordering as the input data for normal, and transpose order for the transpose transform. The "work" parameter uses `MPI_Alltoall` communication at the expense of extra storage space, while "no work" uses point-to-point communication.

We use the speed test provided by the parallel FFTW to benchmark complex multi-dimensional transforms on several processors. The results reported in this paper use normal and transpose order without work space. We ran FFTW in weak-scaling mode.

3.2.5 HPCCG

HPCCG is one of the micro-applications of the Mantevo project [26]. Micro-applications are "small, self-contained programs that embody essential performance characteristics of key applications." HPCCG is intended to be the "best approximation to an unstructured implicit finite element or finite volume application in 800 lines or fewer." These characteristics make HPCCG ideal for our purposes. HPCCG is weak-scaling.

4. CLUSTER SIMULATOR EVALUATION

In this section we evaluate several characteristics of our cluster simulation infrastructure. In the next section we demonstrate, using this infrastructure, the successful evaluation of a novel architectural feature. Here we evaluate the simulator itself.

4.1 Repeatability

Since cycle accurate simulation is time consuming, it is not always possible to repeat a single experiment many times to detect measurement errors. It is therefore important to know how repeatable individual results are and how often outliers occur.

Table 3 shows the execution times reported by several of our application. Since we are running on production clusters, an important consideration is whether running on the same set of nodes produces different results than running on whichever nodes the batch system allocates to us. We ran two different experiments. In "batch" mode we repeatedly submit the job and let the system determine which nodes are available for each subsequent run. In "same nodes" mode we submit a single batch script that runs the same benchmark several times. Each of the runs uses the same set of nodes allocated for the duration of the job.

For each application and mode we calculate the minimum, median, average, maximum, and standard deviation of the running time each benchmark reports. The minimum and maximum are expressed as a percentage of the median.

There is no significant difference between runs on the same set of nodes and letting the batch system allocate nodes for us. However, there is a difference in the behavior of the benchmarks. While IS and LAMMPS show almost no variation from one run to the next, HPCCG and AMG fluctuate more. Figures 3 and 4 show the difference between running AMG and LAMMPS several times in a row. AMG is much less deterministic. Indeed, in many of our experiments we noticed AMG producing outliers, requiring more runs to observe trends in a given experiment. The second row in Table 3 shows the results of 19 native runs using the same parameters as the simulated AMG runs. The reported times are different because we are running on a faster system natively than the one we are simulating, but the run time fluctuations are similar between simulated and native runs. Figure 3 shows this graphically.

4.2 Impact of synchronization interval

In Section 2.2.1 we explained how our simulation infrastructure keeps the individual target nodes synchronized. The choice of synchronization interval has an impact on how long a simulation takes and the accuracy of the simulation. Figures 5, 6, and 7 show results for LAMMPS simulating 4,000 atoms (strong scaling), HPCCG with a problem size of 10×10×10, and AMG with solver 0 and problem size r1. Since we are only interested in the synchronization impact, we turned cache simulation off for LAMMPS and HPCCG.

Table 3: Variations in reported simulation time.

Program	Minimum	Median	Average	Maximum	SD	Runs
AMG, 8 nodes, same nodes	-0.92%	33.2 *ms*	35.2 *ms*	+18.17%	3.4 *ms*	7
AMG, 8 nodes, same nodes, native	-25.0%	12.7 *ms*	12.3 *ms*	+15.26%	1.4 *ms*	19
AMG, 8 nodes, batch	-7.70%	36.0 *ms*	36.5 *ms*	+14.24%	2.9 *ms*	19
HPCCG, 64 nodes, same nodes	-0.66%	80.6 *ms*	81.2 *ms*	+4.06%	1.3 *ms*	2×7
HPCCG, 64 nodes, batch	-1.34%	80.9 *ms*	81.2 *ms*	+3.72%	1.1 *ms*	58
LAMMPS, 16 nodes, same nodes	-0.78%	480.2 *ms*	480.1 *ms*	+0.63%	2.7 *ms*	7
LAMMPS, 16 nodes, batch	-0.85%	480.4 *ms*	480.2 *ms*	+0.75%	2.4 *ms*	24
IS, 64 nodes, same nodes	-0.00%	750.0 *ms*	750.0 *ms*	+0.00%	0.0 *ms*	6×7
IS, 64 nodes, batch	-0.00%	750.0 *ms*	750.0 *ms*	+0.00%	0.0 *ms*	9

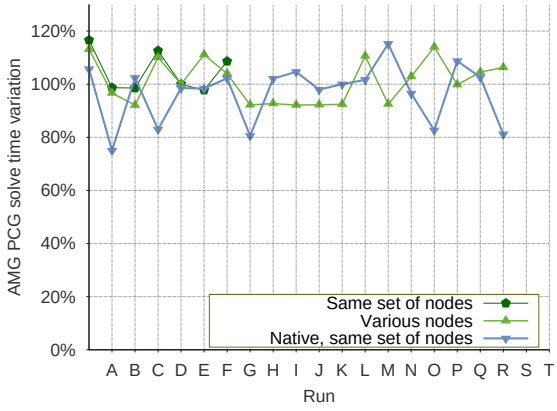

Figure 3: Variability of AMG solve time between runs on 8 nodes.

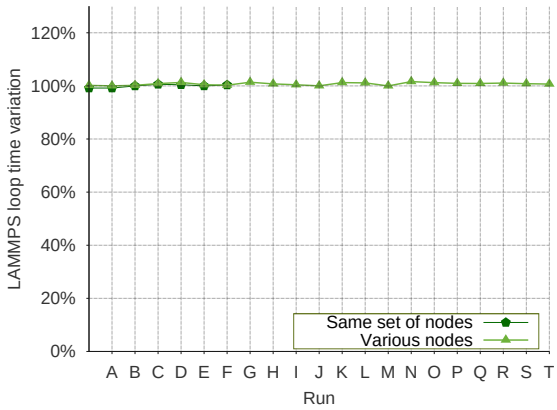

Figure 4: Variability of LAMMPS time between runs on 16 nodes.

We ran AMG in fast-forward mode (see Section 2.1.1).

In Figure 5, we increase the synchronization interval along the x-axis and plot the reported simulation time. The graph shows that if we increase the interval beyond about 10^6 clock cycles, the simulation results start to deviate from the results obtained when the nodes are more tightly synchronized.

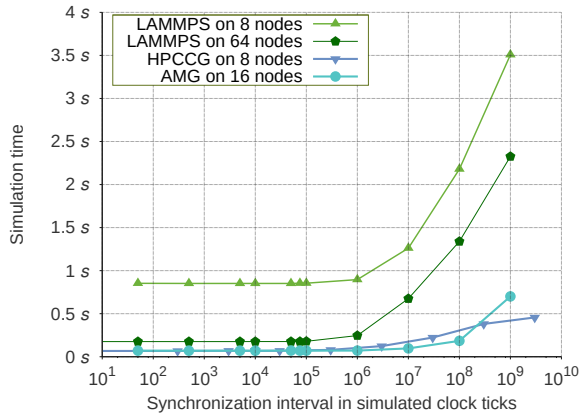

Figure 5: Frequency of synchronization impacts reported simulation time.

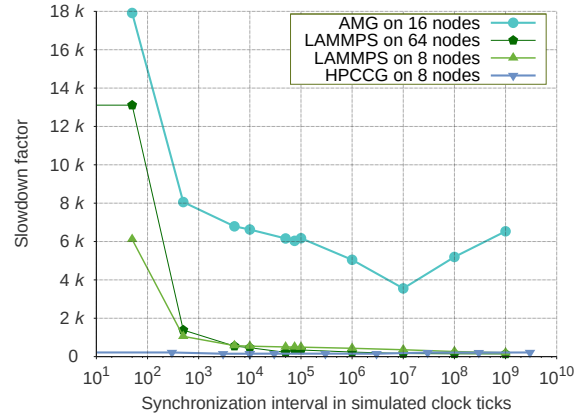

Figure 6: Synchronization frequency impact on wall-clock time.

We calculate slowdown by dividing the simulation time between `MPI_Init` and `MPI_Finalize` into the wall-clock time

between these two functions. Figure 6 shows that increasing the interval between node synchronizations lowers the slowdown factor because the wall-clock time required to run the simulation decreases. Our simulation infrastructure is in itself a parallel application. By increasing the synchronization interval, we reduce the synchronization overhead and, therefore, increase performance.

Since we should not increase the synchronization interval beyond 10^6 clock cycles, and want to keep it as high as possible for performance reasons, a range of 10^5 to 10^6 clock cycles seems appropriate. We chose 50,000 for all the results reported in this paper. Given the target configuration from Table 1 this corresponds to a simulated time of $\approx 30\mu s$. If we assume a simulation slowdown of 6,000 for cache-enabled simulations, this corresponds to about $200ms$ of wall-clock time between synchronizations of the host nodes. The slowdown factor, and whence the time between host node synchronizations, depends on the application and the cache simulation mode (see next section).

Before each synchronization we read the local wall-clocks. After the synchronization we find the lowest time value and subtract it from the highest time value. This measures how much these two extreme nodes have drifted from each other during the previous interval. We keep a running total and report the average drift at the end of the simulation. Figure 7 shows the impact the synchronization interval has on drift. Some applications are more prone to drift than others, with drift becoming a problem at an interval of about 10^6 clock cycles.

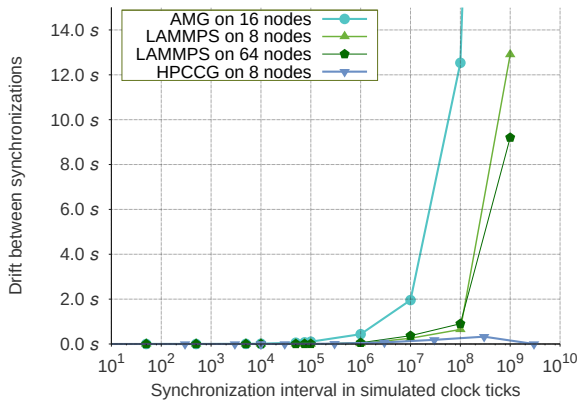

Figure 7: Synchronization interval impacts drift.

Synchronizations occur at specific simulation time intervals. If the nodes drift too much from each other, fast nodes will have to wait longer for others to reach their synchronization points. The simulation clock is stopped during these waits.

Drift is caused by the environment outside our simulation. Different nodes run at slightly different speeds, but a bigger factor is OS noise and I/O when writing logging information, for example. While a node is busy writing to an external file system, or doing OS housekeeping tasks, the simulation clock is not advancing. When that node joins the others in the next synchronization it will show up late, and we can measure that delay.

When the synchronization intervals are high (greater than 10^6 clock cycles), faster nodes can get too far ahead. That means our network model may deliver some messages late in simulation time. This leads to delays in simulation time at the receiving node. That is why drift in wall-clock time has an impact on simulation time. By choosing a good synchronization interval we can keep drift to a minimum and prevent artificial delays in simulation time. Another way to look at this is that our simulator gang schedules the simulated nodes. It can only do that if we keep the synchronization interval sufficiently small; i.e., below 10^6 clock cycles.

4.3 Fast-forward evaluation

In Section 2.1.1 we described the ability of our infrastructure to disable cache simulation and let the simulation proceed faster. At a strategic point in the application, we turn cache simulation back on to properly evaluate an inner kernel or continue a calculation after restarting from a checkpoint.

When turning cache simulation on, the caches will be empty and need some time to warm up. To avoid inaccurate results, it is necessary to turn cache simulation on a little before the program section of interest. That is accomplished with a statement inserted into the source code of the application. We ran AMG on 8 nodes with solver 0 and different workloads with cache simulation turned off, always on, and only on during the solve phase. We obtained the results shown in Table 4. For fast-forward mode we turned cache simulation on during the setup phase to allow the caches to warm up before entering the solve phase.

Both wall-clock and simulation time are measured between `MPI_Init()` and `MPI_Finalize()`. The slowdown factor is the wall-clock time divided by the simulation time. The results in Table 4 are the average of three runs each (problem size r3 is six runs for each simulation).

The solver-only column is the time AMG reports being in the solve phase. During that phase, cache simulation is turned on in the always-on and fast-forward mode. Therefore, the solver times reported for those two modes should be the same. The difference for each problem size in Table 4 is less than 1%, which is well within the running time variations of AMG.

Using fast-forward mode to advance a simulation to the point of interest should help us get the same simulation results as running the entire simulation with cache simulation turned on. The saving column shows how much faster the AMG simulation runs when cache simulation is turned on only during the solve phase.

4.4 Scaling

We have mentioned before that our infrastructure is itself a parallel application. Due to the tight synchronization of the simulated nodes, the parallel performance of the infrastructure is directly tied to the speedup and parallel efficiency of the simulated application.

LAMMPS scales very well as can be seen in Figure 8. The graph plots the reported LAMMPS time against the number of nodes. In weak-scaling mode we increase the overall problem size with the number of nodes available for the computation. This keeps the problem size on each node constant. The flatness of the curve is due to the fact that LAMMPS has very little communication overhead as more nodes are added. We ran with cache simulation turned off for this experiment.

In Figure 9 we show the wall-clock time of our simulation

Table 4: Cost of cache simulation for 8-node AMG

Problem size	Cache simulation	Wall-clock	Saving	Simulation	Solver only	Slowdown factor
1 1 1	Always off	0:00:19		123.465 *ms*	29.209 *ms*	158
	Always on	0:14:32	}30%	137.560 *ms*	33.258 *ms*	6,340
	On during solve	0:10:13		137.880 *ms*	33.626 *ms*	4,447
3 3 3	Always off	0:02:48		535.781 *ms*	233.962 *ms*	313
	Always on	1:40:32	}15%	786.404 *ms*	264.944 *ms*	7,671
	On during solve	1:25:43		753.849 *ms*	264.116 *ms*	6,822
4 4 4	Always off	0:06:15		1.118 *s*	522.955 *ms*	336
	Always on	3:58:37	}12%	1.854 *s*	614.203 *ms*	7,721
	On during solve	3:31:09		1.780 *s*	610.965 *ms*	7,118

infrastructure for the LAMMPS runs from Figure 8 (from `MPI_Init` to `MPI_Finalize`). Our simulator has some overhead due to the frequent synchronizations that occur every 50,000 clock cycles. Nevertheless, it scales very well and is suitable for simulating well-scaling applications, such as LAMMPS, to several hundreds and thousands of nodes.

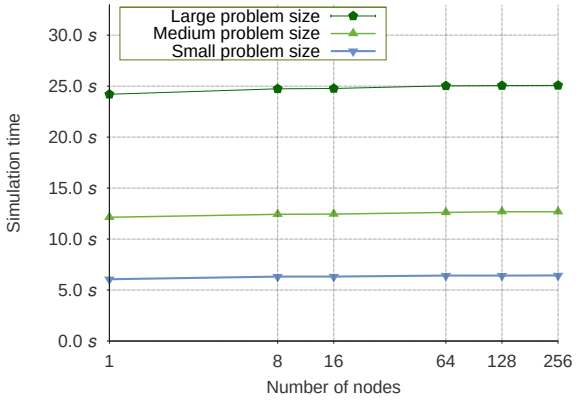

Figure 8: LAMMPS simulation time.

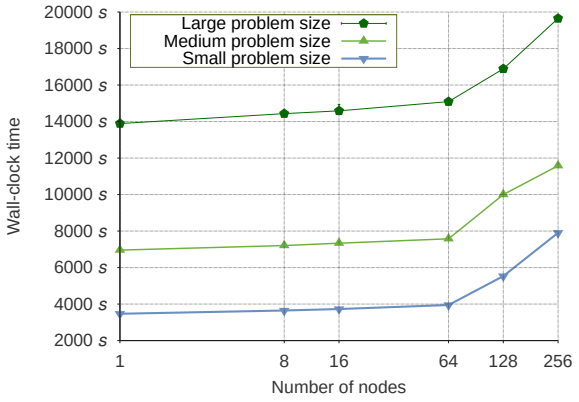

Figure 9: LAMMPS wall-clock time.

Figures 10 and 11 show the running time of the simulated HPCCG micro-application and the behavior of our simulation infrastructure in wall-clock time. Each plot point represents a single run of HPCCG with cache simulation turned off. The simulated HPCCG does not scale quite as well as LAMMPS due to the small problem size used for HPCCG. Comparing Figures 10 and 11, we can see that the inefficiencies of our simulation infrastructure are masked by HPCCG's parallel performance characteristics. Despite this, our infrastructure scales well enough to run HPCCG on several hundred nodes. The relatively poor scaling of HPCCG in our experiments is due to the very small problem size we were able to run within a reasonable time interval when doing cycle-accurate simulations.

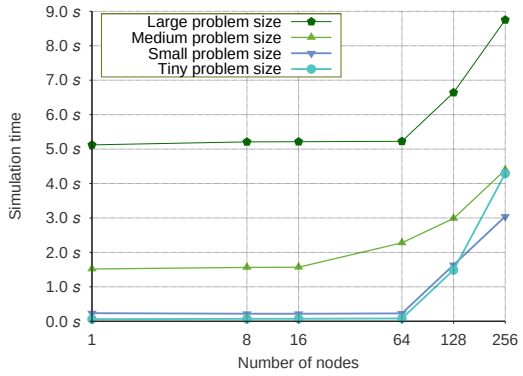

Figure 10: HPCCG simulation time.

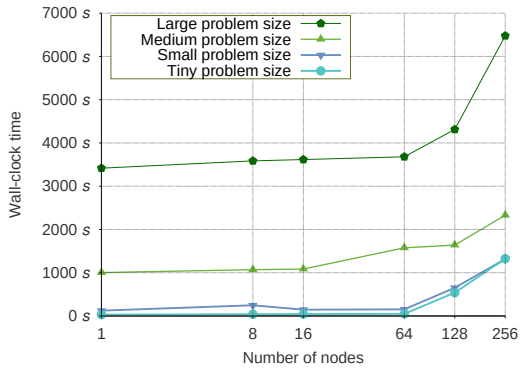

Figure 11: HPCCG wall-clock time.

5. CACHE INJECTION

We created our simulation infrastructure to conduct experiments where cycle-accurate node simulation is important when evaluating message-passing parallel applications. To demonstrate the capabilities of our simulator, we investigate cache injection and its impact on parallel applications performance.

Our version of Mambo has the capability of letting the NIC inject data directly into the L2 or L3 caches. Writing to memory is performed by issuing write-invalidate bus transactions. Writing to a cache is performed in chunks of one cache block and the state of the resulting block is set to clean exclusive [28]. Writes of less than one block are handled by a write with flush operation (flush the cache line first and then write the data into memory). Writes to a cache require the physical address of the destination to be block-aligned. Thus, writing incoming network data to a cache may involve writing the first few words using write with flush until the destination address is cache aligned, then writing full blocks to the cache. Currently, all writes to a cache also update main memory.

If the NIC injects network data before it is needed, it will displace current data, forcing a reload of that data plus a reload of the network data later on. Therefore, determining which network data to inject is an interesting question. We can inject entire messages with the risk of displacing too much data of the current working set. We can inject only MPI envelope information from the message header; such as source, tag, and length information about the message; or we can inject both the payload and the header.

We will look at four different injection policies. "None" is no cache injection at all, which is the base case to which we will make comparisons. "Hl2" injects the message headers into the L2 cache. "Payload" injects the body of the message into the L3 cache as long as the payload is at least 128 bytes (a cache line) but not more than half the L3 cache size. The fourth policy, "Hl2p", combines header injection into the L2 cache with payload injection into the L3 cache.

The version of the NIC we used for these experiments injects data at the time of a successful return from a user-level call to `rx_done` (see Section 2.2.3). That is, the data has arrived at the destination, and the network model has determined that the (simulated) time for delivery has arrived. The next time after that, when the application asks whether a particular message has arrived (using `rx_done`), the NIC injects the data and returns success to the application query. This approach should increase the likelihood that injected data will be used right away, since the application has just asked whether it was available.

Cache injection differs from pre-fetching because cache injection allows the NIC to deposit data directly into the cache (and simultaneously memory). The network data traverses the memory bus once. In pre-fetching the NIC deposit the data and then the CPU pre-fetches it. This requires the network data to traverse the memory bus twice.

Pre-fetching network data may incur additional overhead because the CPU cannot know when new data has arrived and a pre-fetch makes sense. Pre-fetching network data too early will result in old data traversing the memory bus until the NIC deposits the new data from the network in memory. Pre-fetching network data compared to cache injection increases memory bus use and latency.

Although cache injection is not available yet, it is a good example of an architectural innovation that high-end future parallel machines may incorporate into their designs. Before a manufacturer commits to such a design, its utility must be evaluated with simulators like the one we describe here.

5.1 Memory pressure

One of the goals for cache injection is to reduce the number of times data has to travel across the memory bus between arrival at the NIC and consumption by the CPU. In other words, we hope that cache injection relieves pressure on memory by serving more of the data directly from cache. We can measure memory pressure by counting the number of read requests to the memory unit. Only reads that cannot be satisfied from one of the cache levels result in a countable event. In Figures 12 and 13 we show the number of reads issued for the four cache injection policies on increasing number of nodes.

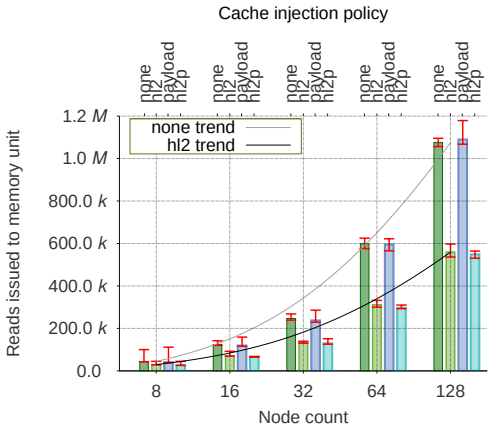

Figure 12: Memory pressure for AMG solve phase.

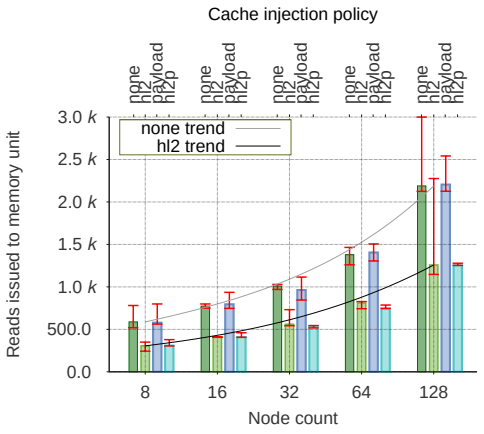

Figure 13: Memory pressure for FFTW transpose phase.

We ran both AMG (solver 0, problem size r1) and FFTW in weak-scaling mode where the problem size per node is kept constant. The number of reads we report are for the transpose portion of FFTW and the solve phase of AMG. Each bar in Figure 12 is the result of at least nine runs. Figure 13 shows five or more runs. We show the median as a colored box and the minimum and maximum as error bars.

One of the outliers we mentioned in Section 4.1 can be seen in Figure 13 on 128 nodes using no cache injection (none).

Analyzing Figure 12 we observe that, without cache injection, the number of reads to main memory increases exponentially as the node count goes up. The upper trend line plots a smooth curve between the no-cache injection data points. The lower trend line follows the hl2 data points. It is clear that injecting header information into the L2 cache greatly reduces memory pressure as the node count goes up. The reason the payload-only injection policy shows no benefit is due to the small problem size we use here, most of the messages are smaller than 128 bytes and will not be injected. The hl2p policy shows the same benefit as hl2 because the headers are injected as in hl2, but the payload is not, since it is too small. Figure 13 shows a similar benefit for FFTW.

5.2 Performance

Reduced memory pressure should lead to better application performance. As Figures 14 and 15 show, this is not really the case for our experimental runs. Median AMG and FFTW run times improved by less than 2%; again due to the small problem size. The memory subsystem has plenty of capacity to support the additional reads that become necessary as we increase the node count. Other work currently in progress shows that the reduced memory pressure enabled by cache injection does reduce run time for larger problem sizes and other applications.

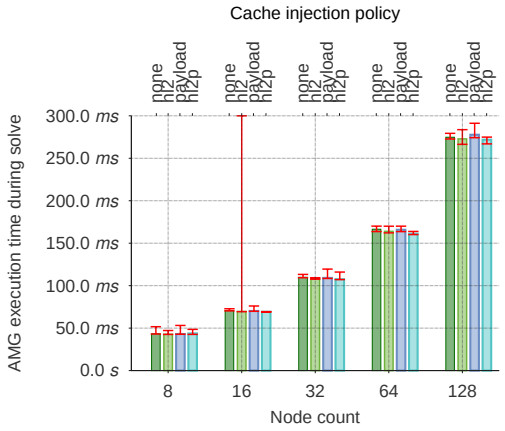

Figure 14: Performance of AMG solve phase.

5.3 Validation

Since we are simulating a system that does not exist – a Cray XT-3 network with PowerPC processor nodes and cache injection – direct comparison is impossible. Mambo on its own has been verified by IBM [22] and we have verified our network model, using bandwidth and latency benchmarks, against our Cray XT-3 [24]. This leaves the question of whether the combination of these two components yields an accurate simulator.

One way to evaluate whether our simulator accurately simulates a scalable machine, is to look at an application and see whether it continues to scale when it is run inside our environment. Figure 8 shows LAMMPS scaling when run inside the simulator. Calculating the parallel efficiency for the data contained in that plot we get 96.6% for 64 nodes and up. This compares favorably to the above 90% parallel

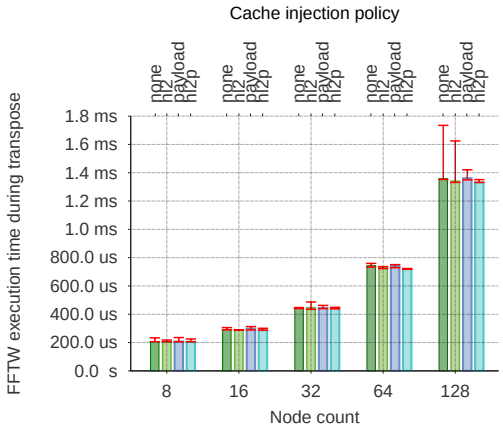

Figure 15: Performance of FFTW transpose phase.

efficiency reported on the LAMMPS web site [25] for the scaled-size metallic solid EAM problem run in our tests. We get better parallel efficiency because we run a smaller problem size and our bandwidth to flops ratio is different than the systems used for the native LAMMPS runs on the web page.

Another approach to validation is the one taken by Falcon et al. [9]. There, the tightly synchronized simulation is taken as the base. As they increase the synchronization interval, the times reported by the benchmark start to diverge. We see the same behavior in our simulator in Figure 5. As long as our synchronization interval does not exceed 10^5 clock ticks, the applications perform deterministically. These, and other experiments, indicate that our simulator is accurate. Nonetheless, until we can simulate an existing system, direct validation cannot be done.

6. RELATED WORK

Our cluster simulator combines a discrete event multiprocessor full-system simulator with a NIC and a network model implementation. Discrete event simulation has been a topic of study for many years. We refer the reader to the chapter on systems simulation in [3]. *Parallel* discrete event simulation has also been explored extensively and many techniques are in use to limit interactions between distant parts of a system. These interactions are necessary to coordinate the distributed event queues, which are needed for a simulator to scale. This, however, increases the need for additional synchronization. Fujimoto [11] provides a very nice description of the problems involved in parallel discrete event simulation.

Asynchronous distributed simulation [6, 14] is one way to address the problem of synchronizing distributed parts of the same simulator. Work on simulating large-scale systems is summarized in [12]. However, the problem of scalability remains. Some researchers turn to modeling instead [17, 16, 19]. While this is more efficient, it is also less accurate and less likely to predict future systems' performance precisely when compared to detailed discrete event simulation.

The work most closely related to ours is COTSon [9, 2]. These papers point out that synchronization is a critical aspect of parallel discrete event simulation. If synchronization is too frequent, the simulation will slow down and cannot scale. If it is too infrequent, messages may arrive in the

past. Compared to their approach, we prevent stragglers by synchronizing at a fixed interval. Additionally, our simulator scales to hundreds of nodes and can run on distributed memory machines. Our NIC and network model is distributed and does not become a centralized bottleneck.

In [5], Burger and Wood evaluate the cost of simulating increasingly accurate network models versus the accuracy they provide. Their target is a 32-node shared-memory machine with three different cache coherency protocols. They find that for message intense applications only the more accurate network models provide enough accuracy. Of course, the cost for more accuracy is more frequent synchronization which limits scalability and performance. We are simulating a distributed-memory target which allows us to use larger synchronization intervals. Because our network model is distributed we also have the luxury of more compute time for future, more accurate network models than the simple one presented in this paper. Nevertheless, the fundamental link between accuracy and performance shown in [5] is true for our simulator as well.

Other projects are attempting to solve the problem of simulating a large-scale HPC system. The MARS simulation framework [7] focuses on the network. It uses Mambo to generate traces that are fed into a network simulator (instead of a model). That approach scales well, but has the disadvantage that detailed analysis of network traffic impact on node architecture, such as cache injection, cannot be done. BigSim [29] uses optimistic synchronization which requires a rollback mechanism when causality errors occur. This approach is useful for fine-grained tasks to exploit parallelism more efficiently. Mambo, as an individual parallel task, is coarse grained. Therefore, optimistic synchronization would not be a benefit for our cluster simulator. Just like MARS, BigSim is not capable of detailed node simulation.

7. SUMMARY AND FUTURE WORK

In this paper we describe an infrastructure that allows systems researchers to study the impact of architectural changes on scientific parallel application performance. This infrastructure is designed to:

- leverage current single-node simulators;
- enable simulation of recent and future cluster architectures, including techniques to improve application performance and scalability;
- allow system designers to better understand the interactions between the OS, parallel applications and the NIC, as well as between nodes; and, finally,
- accurately simulate a cluster at scale.

The results presented in this paper indicate that our infrastructure can indeed meet these goals. We will continue making improvements to the simulation infrastructure itself. For example, we would like to replace K42 with Linux.

Many opportunities exist for future work with the simulator described in this paper. We have only scratched the surface of exploring the potential benefits of cache injection, and plan to explore cache injection policies for multi-core architectures. We also intend to assess the impact of different network characteristics, such as different latencies and bandwidths, on applications and their interactions with the memory system.

Mambo is able to simulate a multi-processor multi-core machine. This kind of architecture will become prevalent in the next-generation supercomputers and will significantly change the flops to network bandwidth ratio. We want to evaluate the impact of such an architecture on parallel application performance.

8. REFERENCES

[1] J. Appavoo, M. Auslander, M. Burtico, D. D. Silva, O. Krieger, M. Mergen, M. Ostrowski, B. Rosenburg, R. W. Wisniewski, and J. Xenidis. K42: an open-source Linux-compatible scalable operating system kernel. *IBM Systems Journal*, 44(2):427–440, 2005.

[2] E. Argollo, A. Falcón, P. Faraboschi, M. Monchiero, and D. Ortega. Cotson: infrastructure for full system simulation. *SIGOPS Oper. Syst. Rev.*, 43(1):52–61, 2009.

[3] J. Banks, J. S. C. II, B. L. Nelson, and D. Nicol. *Discrete-Event System Simulation*. Prentice-Hall, Inc., 3rd edition, 2000.

[4] P. Bohrer, R. Rajamony, and H. Shafi. Method and apparatus for accelerating Input/Output processing using cache injections, Mar. 2004. US Patent No. US 6,711,650 B1.

[5] D. Burger and D. A. Wood. Accuracy vs. performance in parallel simulation of interconnection networks. In *IPPS '95: Proceedings of the 9th International Symposium on Parallel Processing*, pages 22–31, Washington, DC, USA, 1995. IEEE Computer Society.

[6] K. M. Chandy and J. Misra. Asynchronous distributed simulation via a sequence of parallel computations. *Commun. ACM*, 24(4):198–206, 1981.

[7] W. E. Denzel, J. Li, P. Walker, and Y. Jin. A framework for end-to-end simulation of high-performance computing systems. In *Simutools '08: Proceedings of the 1st international conference on Simulation tools and techniques for communications, networks and systems & workshops*, pages 1–10, 2008.

[8] R. F. V. der Wijngaart. NAS parallel benchmarks version 2.4. NAS Technical Report NAS-02-007, Computer Science Corporation, NASA Advanced Supercomputing(NAS) Division, NASA Ames Research Center, 2002.

[9] A. Falcon, P. Faraboschi, and D. Ortega. An adaptive synchronization technique for parallel simulation of networked clusters. *Performance Analysis of Systems and software, 2008. ISPASS 2008. IEEE International Symposium on*, pages 22–31, April 2008.

[10] M. Frigo and S. G. Johnson. The design and implementation of FFTW3. *Proceedings of the IEEE*, 93(2):216–231, 2005.

[11] R. M. Fujimoto. Parallel discrete event simulation. *Commun. ACM*, 33(10):30–53, 1990.

[12] R. M. Fujimoto, K. Perumalla, A. Park, H. Wu, M. H. Ammar, and G. F. Riley. Large-scale network simulation: How big? How fast? In *11th IEEE/ACM International Symposium on Modeling, Analysis and Simulation of Computer Telecommunications Systems (MASCOTS)*, pages 116–123, Oct. 2003.

[13] R. L. Graham, T. S. Woodall, and J. M. Squyres. Open MPI: A flexible high performance MPI. In *Proceedings of the 6th Annual International Conference on Parallel Processing and Applied Mathematics*, September 2005.

[14] A. G. Greenberg, B. D. Lubachevsky, and I. Mitrani. Superfast parallel discrete event simulations. *ACM Trans. Model. Comput. Simul.*, 6(2):107–136, 1996.

[15] W. Gropp, E. Lusk, N. Doss, and A. Skjellum. A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789–828, Sept. 1996.

[16] A. Hoisie, G. Johnson, D. J. Kerbyson, M. Lang, and S. Pakin. A Performance Compariosn through Benchmarking and Modeling of Three Leading Supercomputers: Blue Gene/L, Red Storm, and Purple. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, Nov. 2006.

[17] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme. A general predictive performance model for wavefront algorithms on clusters of SMPs. In *ICPP '00: Proceedings of the International Conference on Parallel Processing*, pages 219–228, 2000.

[18] R. Huggahalli, R. Iyer, and S. Tetrick. Direct cache access for high bandwidth network I/O. In *32nd Annual International Symposium on Computer Architecture (ISCA'05)*, pages 50–59, June 2005.

[19] D. J. Kerbyson, H. J. Alme, A. Hoisie, F. Petrini, H. J. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of the ACM/IEEE conference on Supercomputing*, pages 37–48, Nov. 2001.

[20] Lawrence Livermore National Laboratory. ASC Sequoia benchmark codes. `https://asc.llnl.gov/sequoia/benchmarks`, Apr. 22 2008.

[21] E. A. León, K. B. Ferreira, and A. B. Maccabe. Reducing the impact of the memory wall for I/O using cache injection. In *15th IEEE Symposium on High-Performance Interconnects (HOTI'07)*, Aug. 2007.

[22] J. L. Peterson, P. J. Bohrer, L. Chen, E. N. Elnozahy, A. Gheith, R. H. Jewell, M. D. Kistler, T. R. Maeurer, S. A. Malone, D. B. Murrell, N. Needel, K. Rajamani, M. A. Rinaldi, R. O. Simpson, K. Sudeep, and L. Zhang. Application of full-system simulation in exploratory system design and development. *IBM Journal of Research and Development*, 50(2/3), Mar. 2006.

[23] S. J. Plimpton. Fast parallel algorithms for short-range molecular dynamics. *J Comp Phys*, 117(1):1–19, 1995.

[24] R. Riesen. A hybrid MPI simulator. In *IEEE International Conference on Cluster Computing (CLUSTER'06)*, 2006.

[25] Sandia National Laboratory. LAMMPS molecular dynamics simulator. `http://lammps.sandia.gov`, Nov. 6 2008.

[26] Sandia National Laboratory. Mantevo project home page. `https://software.sandia.gov/mantevo`, Nov. 6 2008.

[27] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. POWER5 system microarchitecture. *IBM Journal of Research and Development*, 49(4/5), 2005.

[28] J. M. Tendler, S. Dodson, S. Fields, H. Le, and B. Sinharoy. POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–26, Jan. 2002.

[29] G. Zheng, T. Wilmarth, P. Jagadishprasad, and L. V. Kalé. Simulation-based performance prediction for large parallel machines. *Int. J. Parallel Program.*, 33(2):183–207, 2005.