Simulating Red Storm: Challenges and Successes in Building a System Simulation

Keith D. Underwood, Michael Levenhagen, and Arun Rodrigues
Sandia National Laboratories
P.O. Box 5800, MS-1110
Albuquerque, NM 87185-1110
{kdunder, mjleven, afrodri}@sandia.gov

Abstract

Supercomputers are increasingly complex systems merging conventional microprocessors with system on a chip level designs that provide the network interface and router. At Sandia National Labs, we are developing a simulator to explore the complex interactions that occur at the system level. This paper presents an overview of the simulation framework with a focus on the enhancements needed to transform traditional simulation tools into a simulator capable of modeling system level hardware interactions and running native software. Initial validation results demonstrate simulated performance that matches the Cray Red Storm system installed at Sandia. In addition, we include a "what if" study of performance implications on the Red Storm network interface.

1 Introduction

Modern supercomputers, such as the recent Red Storm machine (the first Cray XT3) and the IBM BlueGene/L machine, are complex systems combining microprocessors with custom network interfaces and routers. In general, the *system level* design of such machines is driven by intuition and supported by *component level* simulations. The system performance is vulnerable to interactions between components that are hard to predict. Furthermore, the performance impact of minor changes on MPI performance is seldom obvious to the designers until a system has been built.

To address such issues, Sandia National Labs is developing an open source, system level simulator to explore the many complex interactions that occur at the system level. The long term goal of the effort is to build a scalable parallel simulation capability that can be configured to provide varying levels of fidelity for each of the system components. We anticipate simulations that range from near cycle accuracy for every system component to simulations that model one component in detail while substituting very high level models for all of the other components.

We present an overview of the initial, serial version of the simulation framework. It is derived from the Structural Simulation Toolkit (SST) [19], which is a hybrid discrete-event/synchronous simulator that incorporates processor simulation models from the SimpleScalar processor simulation suite. Our first step toward system level simulation is to enable high fidelity simulation of supercomputer network interfaces. As observed in [17], traditional simulators are poorly matched to system level simulation. Specifically, accurately modeling network transactions and delays between the host processor and a connected network interface were particular challenges. Thus, we extended SST to accommodate a model of the Red Storm network interface (the Seastar 2.1). That model has been used as part of a two node system simulation running native Red Storm system software. We provide a discussion of the challenges of building a high fidelity network model with a focus on the changes to the general simulation framework that were needed.

The simulated system performance has been validated to match Red Storm within 5% over most ranges of operation. We were also able to explore several "what if" scenarios. For example, what would happen to message throughput if the clock rate of the network interface were doubled? If the latency of the HyperTransport interface was reduced, would it help performance? These are system level questions that would be difficult to simulate in a Verilog model; however, by using system level simulations we can rapidly explore such system level questions.

Sandia 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.

2 Background and Related Work

A variety of simulators and simulation strategies are used in computer architecture, providing a range of features and functionality. At the lowest level, architectural simulators explore design issues on the processor or system level. These simulators represent programs by execution-based, trace-based, or stochastic mechanisms and vary in level of detail, configurability, and focus.

2.1 Simulators

SimpleScalar [6] is a commonly used architectural simulation toolkit. It includes execution-based simulators, ranging from simple execution to cache simulation to full simulation of an out-of-order processor and memory hierarchy (`sim-outorder`). SST leverages SimpleScalar to provide a the processor simulation. Similarly, other simulators are derived from SimpleScalar and have extended its functionality. For example, the SImulator for Multithreaded Computer Architecture [9] (SIMCA) was developed to explore multi-threaded architectures.

Simics[13] and the Wisconsin Multifacet GEMS simulator[14] that extends it provide a flexible environment that addresses the needs of CMP and SMP simulation and memory hierarchy design. GEMS also decouples the functional portion from the timing and microarchitectural simulation to simplify timing design. In contrast, SST targets issues in CMP, SMP, and MPP systems where a node is a heterogeneous collection of processors, memories, NICs, and routers. As noted in [17], it is extremely challenging to simulate detailed system level details in Simics.

Some simulators have been developed to enable specific functionality. For example, SimOS [8] and ML-RSIM [20] support the execution of an OS. Simulators such as `simg4` [16], were developed to model a specific processor (the PowerPC 7400) in detail.

The ASIM [7] performance model framework is composed of a set of modules which can be composed to form different architectures. A novel feature of ASIM is the partial separation of the performance model for system components from the program execution. Other modular simulation efforts include the Liberty Simulation Environment [23], which has developed a number of modules in its own LSS language, and Microlib which provides a number of modules in SystemC [22].

The message PAssing computeR SIMulator, PARSIM [21] was developed to explore algorithms and network topologies for parallel computers. It models program execution as a generalized algorithm divided into computation and communication. Processor speed and network characteristics can be parameterized, but the internals of the processor are not modeled.

In general, most simulation efforts focus on a given piece of the system with varying degrees of accuracy and parameterizability. As noted in [17], readily available simulators tend to lack the ability to pull together high fidelity simulations of *all* system components into a single framework. Our experience extending SST highlights the challenges that are faced when using available tools to model a supercomputer accurately.

2.2 Models of Computation

An important characteristic of a simulator is the underlying model of computation [12], which defines how time is advanced and how components interact. SST leverages both a *synchronous* and a *discrete event* model. Synchronous, or time-stepped, models discretize time into fixed increments (cycles in the case of architectural simulators) at which all components are evaluated (e.g. SimpleScalar and PARSIM). SST provides a synchronous model to enable efficient implementation for those components that need near cycle accuracy. A discrete event, or event-driven, model generates *events* for each transition. Generally, event driven models are more efficient for components where events are relatively infrequent when compared to the clock rate. SST provides a discrete event model for the system level components that communicate less frequently than every cycle. For example, a bus interface that has a 100 ns latency and that is not used every cycle does not need to be evaluated at a 2 GHz rate.

3 Framework

The Structural Simulation Toolkit (SST) is an architectural simulator implemented in about 45,000 lines of C++. It is composed of four primary elements (see Figure 1): the **Front Ends**, which model the execution of a program; the **Back Ends**, which model architectural components of the system; the **Processor/Thread Interface**, which allows the front and back ends to interact; and **`Enkidu`** [18], a component-based discrete event and synchronous simulation framework that coordinates communication between back end components and models the passage of time. To provide modularity and reconfigurability, it is possible to select a front end and choose a variety of back end components at run time. This allows the user to explore a variety of hardware configurations while using the execution model best suited to the available toolset.

3.1 Front End

The front end generates instructions and threads to be processed by the back end. The front end also defines how

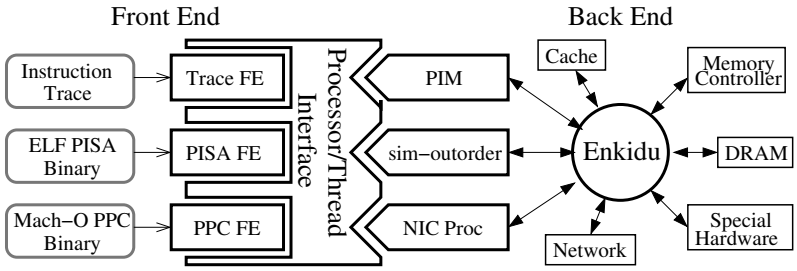


Figure 1. Simulator Framework and Component

the execution of these instructions changes the programmer visible state. In short, the front end simulates how the program executes from a software perspective — it ignores hardware and timing details and just looks at how the state of its registers and memory is modified by the `instructions`.

The front end provides a loader to load the program into the simulated memory. Once simulation begins, the front end provides instruction and thread objects to the back end (see Section 3.3). Most importantly, it determines how the program state (usually memory and registers) are modified by the instruction's execution. Much like SimpleScalar's [6] `ss.def` or `powerpc.def` files, or SPIM's [11] `run.c`, this usually involves a lookup of the instruction in a "big case statement" to ascertain how state should be updated. Currently, three front ends exist and can be selected at runtime: PPC, PISA, and Trace.

Only the PPC front end is used in these experiments. It is an execution-based front end which uses the PowerPC ISA [15] and the MachO [2] executable format. It includes a small subset[1] of the AltiVec vector extensions. The MachO format is the standard format for MacOS X executables and allows the use of binaries created by a number of modern compilers. It has been tested with a variety of compilers.

3.2 Back End

The back end models the hardware of the system. It consumes instructions and threads generated by the front end and determines how long it would take for the hardware to execute them. It ignores the specifics of what values are written where and focuses on the timing details of which components are accessed, how long memory transactions take, and other microarchitectural details. The back end is composed of many different `Enkidu` components that represent physical components such as processors, networks, memory controllers, and DRAMs. Components can communicate through `Enkidu`'s discrete event system.

[1] `lvx`, `stvx`, `vspltw`, `cmpequh[.]`, and `vand` instructions

Numerous back end components have been developed, but the subset used to model Red Storm includes:

- **Conventional Processor**: an out-of-order, multi-issue processor based on SimpleScalar's `sim-outorder`. It can use SimpleScalar's memory model or can connect to a memory controller component.
- **Memory Controller**: a memory controller model which simulates bus contention, bandwidth constraints, latency, and DRAM interleaving.
- **DRAM**: a model of a DRAM chip with a configurable number of DRAM banks. The DRAM bank size and width can be configured. Open page latency and contention effects are modeled, and the number of open rows per bank can be adjusted.
- **Simple Network**: a simple network model connecting NICs on different systems. Latency and bandwidth effects are modeled, but not topology.

3.3 Interface

The `Processor/Thread` interface is the key bridge between the front end and back end. This interface defines three abstract classes: `processor`, `thread`, and `instruction`. A `processor` is a back end component which can execute `instructions` belonging to one or more `threads`. Each front end defines a `thread` class and `instruction` class.

3.4 Hybrid simulation

Modern processors often have dozens of instructions in various stages of execution during each processor clock cycle. Each of these instructions (potentially) moves to a new stage on each cycle. As a result, several transition events can be expected to occur each cycle. This tends to make synchronous simulation more efficient.

In contrast, parallel supercomputers add network interfaces and routers. Communications between chips as well

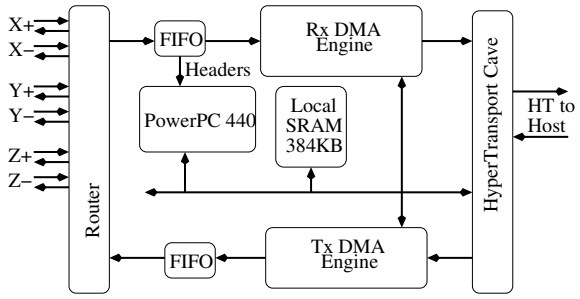


Figure 2. Block diagram of Seastar NIC

as communications within a network interface or router involve the movement of large amounts of data per unit of logic. This would be extremely expensive to model in a synchronous fashion. For example, a router only needs to know when a packet transfer starts and when it stops, rather than every transition of every flit. Simply tracking start and end times has proven to be sufficiently accurate.

To offer both efficient processor simulation and efficient system simulation, the Structural Simulation Toolkit (SST) is built around `Enkidu`, a hybrid simulation framework. In `Enkidu`, `component` objects represent each physical component of the system. Each of these `components` is evaluated every clock cycle, allowing it to advance its internal state. In addition, `components` can communicate by passing event messages to each other in an asynchronous manner.

4 Modeling Red Storm

Modeling a real system comes with many complications. To facilitate a discussion of the issues, Figure 2 shows a block diagram of the Seastar network interface [1]. There are five major components of the Seastar that are connected by a single bus. These include a processor, a local SRAM, transmit and receive DMA engines, and a HyperTransport interface to the host processor. The entire NIC runs at 500 MHz and the HyperTransport link to the host runs at 800 MHz and is 16 bits wide in each direction.

The PowerPC processor runs a firmware image that implements the Portals 3.3 API [4, 5]. Portals is an API that encapsulates MPI matching semantics and, thus, offloads most of the MPI work to the network interface. A detailed description and analysis of the Seastar hardware and software environment are presented in [3]. The most important note from that work is that the Opteron and PowerPC communicate through the SRAM on the Seastar.

4.1 Building the Hardware Model

As with most simulations, there were various components to be modeled. While most of the hardware needed new models, we could leverage the PowerPC model from SimpleScalar for the processor implementation on both the host and the NIC. Similarly, the DRAM component on the host was a standard part of SST as was the SRAM component, but both memory models had to be updated somewhat as discussed in Section 4.3. Finally, for these experiments, the network was modeled as a simple point-to-point network using components from SST.

4.1.1 Bus Components

At the core of the Seastar chip is a bus connecting all of the other components. It models both latency and contention effects for accesses from the PowerPC and from the Opteron (host). The processor back end uses an event mechanism to request the timing of accesses to devices on the Seastar. The event is routed to the bus timing component, which models latency and contention, and is then routed to the appropriate component for additional delays before returning back through the bus model to the processor back end.

The HyperTransport (HT) connection was modeled as two components: one to model latency and one to model bandwidth and contention. The `HTLink` (latency component) introduces latency for access across HT based on whether the access is a read or write. Each side of the HT connection has an `HTLink` component that models time in native cycles. An `HTLink_bw` connects the host processor to the Seastar with a model of bandwidth and contention. Since the combination of the DMA accesses and processor accesses can request much more than the HyperTransport bandwidth, the `HTLink_bw` tracks the backlog of requests. The backlog of requests is incremented by arriving events and decremented on each clock cycle based on the *bytes per clock* that the HT can service. This is an example of where discrete event simulation and synchronous simulation intersect for a more efficient and more accurate simulation model.

Because the HT component models contention, it would be possible for the backlog to grow arbitrarily long. Since this would be an impractical scenario (even for simulation) and the impacts on a broader network simulation would be unrealistic, the HT component maintains flow-control with requesters by having a finite depth request queue. As with real hardware, this queue depth is set to cover round-trip times such that full bandwidth can be sustained.

4.1.2 DMA Engines

The Seastar network interface also includes a robust DMA engine that is controlled by the PowerPC. While the detailed

complexities of the DMA engines are beyond the scope of this paper, the implementation was designed to respond to the same set of commands (e.g. bit patterns in a control word) and to present the same interfaces as real hardware. Thus, the Red Storm firmware could run unmodified in the simulator and interact correctly with these DMA engines.

Beyond the control interfaces, the DMA engines are also required to move data in the form of packets. On the transmit side, the DMA engine processes a command and makes 64 byte requests over a flow-controlled interface to HyperTransport with finite request buffer space. These requests return timing information about when the request should complete. Like the real hardware, all of these operations are packet oriented.

On the receive side, the Seastar has a FIFO of arriving packets. The DMA engine consumes up to 8 bytes per cycle from the FIFO and allows the network to deliver up to 8 bytes per cycle. That rate can be throttled by the HT interface because the receive DMA engine competes for bandwidth with both the processor accesses and the transmit DMA engine's HT read requests[2]. Tracking this contention and flow control is critical as it creates measurable, real system effects.

One of the novelties of real system hardware is that reads and writes have side-effects. For example, a read from a register may pop a value from a queue. The DMA engines have numerous such side-effects that must be tracked with correct timing. And, correspondingly, there must be a timing component that provides timing information back to the bus component for processor accesses.

4.2 The Software Stack

Our goal in modeling Red Storm was to run the software stack with as little modification as possible. On the host processor, Sandia runs a lightweight kernel that provides relatively bare access to the hardware [10]. Using the "accelerated" mode of operation [3], the network stack is split between a library in user space and firmware on the network interface; thus, we lose very little in fidelity when working without an OS. The small number of OS functions that are typically needed in the production software stack (e.g. protection) are not needed in the simulated environment.

In the simulated environment, the production software is used with minimal modification. The small portions of the host side code that normally run in protected mode are pulled into the library. On the firmware side, TLB setup instructions were eliminated and cache manipulation instructions (cache invalidates) were removed[3].

[2]HyperTransport connections are directional. Thus, a small read request goes over the same path as write data. Read data returns over a different path.

[3]The firmware can work without them because the processor model always pulls data from memory rather than storing the data in cache.

4.3 Modifying the Simulator

SST relies heavily on SimpleScalar to provide an accurate model of a microprocessor. The infrastructure was built around the desire to add a variety of system level aspects to SimpleScalar; however, it took for granted that the SimpleScalar model was sufficient to model the processor. In attempting to model Red Storm, we have found that there are a number of system level issues that could not be addressed in the current framework.

4.3.1 Memory and Addressing

One of the biggest challenges associated with using the existing simulation framework was associated with the memory and addressing models. Rather than a single SRAM, real systems tend to have numerous memory regions. For example, the Opteron model can see both its own cacheable DRAM and an uncacheable space of SRAM that is on the Seastar and used to communicate between the Opteron and PowerPC. On the Seastar, the PowerPC can access local SRAM, the Opterons DRAM (uncacheable), and numerous physical registers through its address space. Not only do these various regions have different timing properties, but they can also have side effects. For example, writing to a DMA register starts a data transfer.

SimpleScalar uses a split memory model that handles data access independently from access timing. Thus, independent paths need to be implemented to support the model. This adds some complexity to the implementation. In addition, the instruction is "executed" in the dispatch stage (early in the instructions life). This means that the memory resident state is changed long before the commit, which is the part dependent on the timing parameters. This has implications for correctness.

Supporting the various address ranges required handling of both timing and data redirection. Timing redirection was accomplished by adding a bus component that was the processor's timing point of contact. A registration function for the bus allowed other components (such as the SRAM or DMA engine) to indicate which ranges of memory they were responsible for. The bus can then contact the appropriate units for timing information when a memory access is made. Data redirection was accomplished by changing the memory component that backs the actual data in the conventional simulator. By allowing the components to also register with the memory object, the memory object is able to forward the data accesses to the appropriate component.

When multiple components interact, it becomes important to implement correct memory timings and semantics. One major issue is the need to separate cacheable and uncacheable address regions. In the real Seastar firmware, accesses to register space must be uncached for correctness.

Register accesses have side effects and registers change independently of the PowerPC. Thus, a flag was added to read timing requests to indicate to the processor back end whether the access could be cached (and the line was not inserted into the cache if the space was uncacheable).

In addition to the issues of cached versus uncached space, the split memory model introduces an issue of timing correctness. The host CPU and the Seastar CPU communicate using polling of various memory regions. Because the processor model directly changes memory early in an instruction's life and then commits at a time based on the timing model, the other processor could "see" changes before they should have actually happened. Rather than invasively modify the processor model, we chose to modify the memory component to delay writes based on timing. This issue also appears in high latency reads to shared space[4], but it has not been seen to cause a significant problem in practice.

Finally, because we desire minimal modifications from production software, we wanted a virtual memory capability comparable to what exists with the real host processor. This allows us to map hardwired addresses appropriately into the applications address space *in the same place* that they would otherwise land. The modification occurs in the simulators front-end, which remaps virtual addresses based on address ranges.

4.3.2 Simulator Infrastructure

There are two issues associated with the simulator infrastructure that needed to be addressed as well. The original SST assumed a homogeneous load of one binary onto some number of homogeneous processors. In a model of a Red Storm node, we need to execute 2 unique binaries using different processor models (PowerPC vs. Opteron), different memory layout, and different clock rate. The SST binary loader was modified to create multiple memory objects, each of which can be loaded with unique binaries. The loader was also modified such that the location of text, stack and heap could be specified on a per binary basis. The SST configuration scheme was modified such that the same component could be configured differently. This allows for example different clock rates and memory layout for the conventional processor component which is used to model both the Opteron and PowerPC.

SST also made the assumption that everything operated on a single clock. It was possible to run a component at a different (synchronous) clock but it had to be coded into each component. In contrast, a Red Storm node has components in three major clock domains: an Opteron at 2GHz, a PowerPC at 500MHz, and HyperTransport at 800MHz. Multiple clock support was generalized by modifying the base simulation component, `Enkidu`, such that it calls the work functions for each component at a frequency derived from the primary simulation clock rate. We plan further generalization to allow arbitrarily aligned clocks.

[4] A read could get "old" data

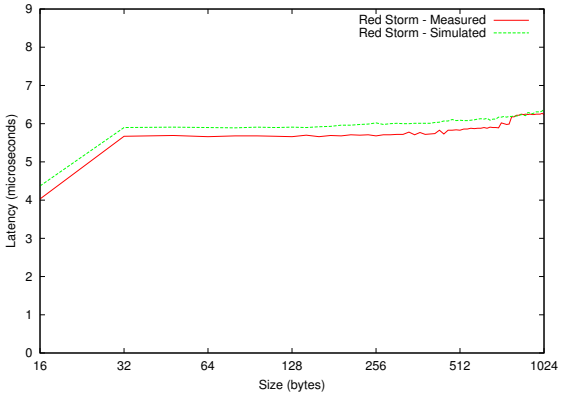


Figure 3. MPI ping-pong latency validation

5 Validation Against Red Storm

To validate the simulation, we have compared results from the simulator to measurements taken from Red Storm hardware. These measurements come from both high level network benchmarks and low level measurements from the firmware. The overall result indicates that the simulation very accurately models the real system.

At the high level, we used two benchmarks to validate the model of the Red Storm system. We began with measurements of MPI ping-pong latency. Ping-pong latency is a common network benchmark that helped to insure that basic latencies are correct. As Figure 3 illustrates, the MPI ping-pong latency matches within 5% between the simulated and real systems.

While an MPI ping-pong latency measurement demonstrates that numerous system timings have been modeled correctly, it can hide numerous other inaccuracies in a model of a network interface. Thus, we also used the streaming bandwidth benchmark developed by the Ohio State University, which is another standard benchmark for supercomputer networks. This benchmark posts 64 receives on a target node and then streams 64 messages from the source node. The goal is to measure peak streaming bandwidth at various message sizes; thus, a key part of this measurement is the rate at which new messages are handled.

For this test to match between the simulation and the real system, most of the model has to be correct. The bandwidth between the host processor and the network interface has to be accurate, or the peak bandwidth will be incorrect. The processors involved (both the host and the one on the NIC) have to be relatively accurate, or the messages

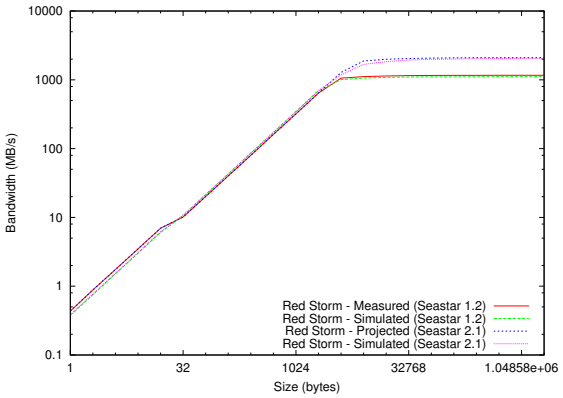


Figure 4. MPI OSU bandwidth validation

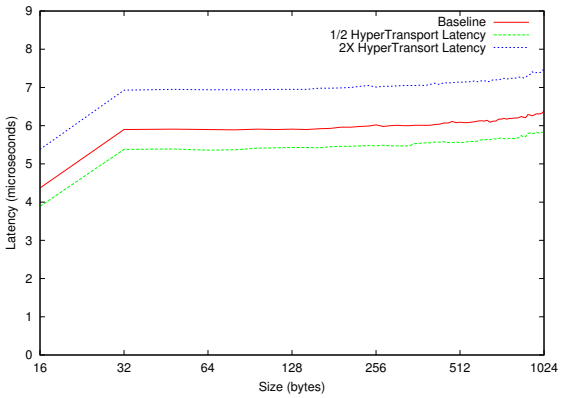


Figure 5. Impact of changing HyperTransport latency on MPI latency

Table 1. Individual routine timings

Routine	Simulated	Actual
handle_command PUT	0.486 us	0.592 us
tx_complete() USER message	0.196 us	0.154 us
rx_message() - ACK	0.959 us	1.002 us
rx_complete() - ACK	0.127 us	0.242 us
handle_command POST	0.477 us	0.442 us
rx_message() - USER message	1.936 us	1.686 us
tx_complete() - ACK	0.114 us	0.118 us
rx_complete() - USER message	0.230 us	0.378 us

will be handled too quickly. Similarly, most timings on the NIC have to be correct, or the messages will be handled too quickly or too slowly. Figure 4 indicates that the message processing rate and peak bandwidths match within 5% over a large range of message sizes. The graph includes measurements from a Red Storm system with Seastar 1.2 network interface chips along with simulations of the Seastar 1.2. It also includes a prediction of the performance of Seastar 2.1 chips (being installed now) based on low level measurements from the Seastar

The only major discrepancy in Figure 4 is for messages of 16 bytes or less. In this range, the simulated performance deviates from the real performance by approximately 12%. Messages of this size are handled slightly differently in the Red Storm system; thus, they exercise slightly different paths in the simulator. To further investigate where any discrepancies arise, we instrumented the firmware on both production hardware and simulated hardware and show the results in Table 1. The first four lines are send side operations while the last four lines are receive side operations. The biggest discrepancy between real and simulated hardware is the `handle_command` PUT. This arises from a known issue in the simulation: the SimpleScalar PowerPC instruction set implementation does not include cache line invalidate instructions. Thus, a region of the memory on the NIC that is used to communicate with the host is treated as always cached rather than being cached but invalidated when a new command arrives. The message rate limiting point appears to be the extra time in `rx_message`. We are still investigating the cause of this discrepancy, but the accuracy of the remaining routines gives us sufficient confidence in the model accuracy to move forward with evaluations.

6 Results

One of the surprising factors experienced with the Seastar was a much higher effective HyperTransport (HT) latency than expected. To explore the impacts of HT latency on the MPI latency, we ran simulations with the HT latency set to $\frac{1}{2}$ and $2\times$ the measured value. In Figure 5, we can see that MPI latency is linearly related to the change in HT latency. The one notable point is that the change in MPI latency is $4\times$ the change in HT latency; however, this is not surprising, since two HT writes and one HT read are involved in an end to end transaction (push a command to the Seastar, a DMA read of the data, and pushing a result to the host — the Seastar does not use a programmed-I/O mechanism for short messages). Also of note is that the change in HT latency had no impact at all on streaming bandwidth (not shown).

Like HT latency, the HT bandwidth was lower than initially hoped. In Figure 6, we present the results from varying HT bandwidth from $\frac{1}{2}$ to $2\times$ the measured value. Since HT bandwidth did not impact MPI latency, we only show the results from streaming bandwidth. The change in the HT bandwidth yields the expected change in *peak* streaming bandwidth, but does not change bandwidth at smaller message sizes at all.

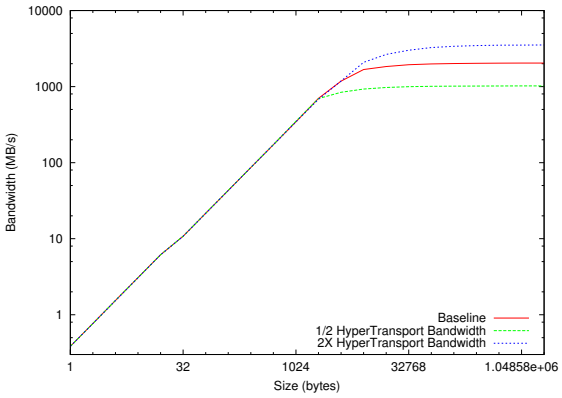


Figure 6. Impact of changing HyperTransport bandwidth on streaming bandwidth

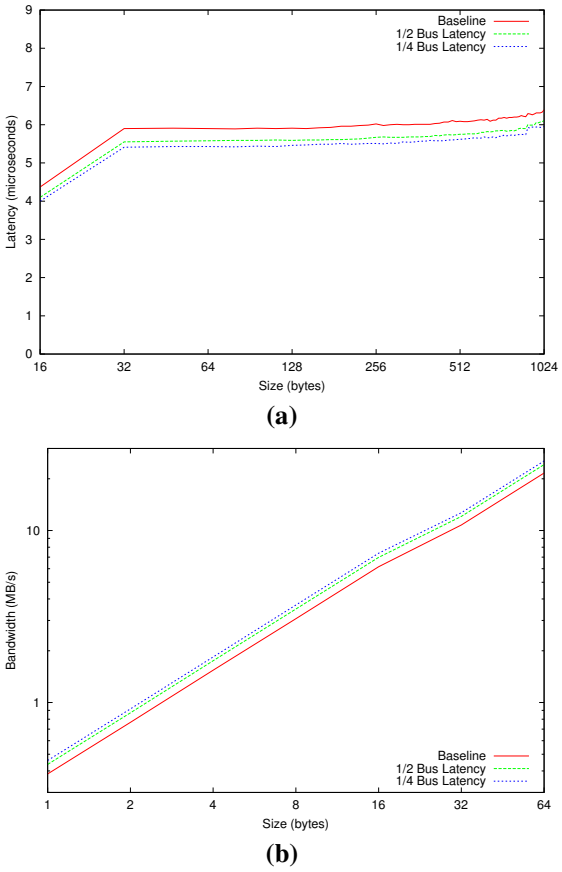


Figure 7. Impact of changing NIC bus latency on (a) latency and (b) streaming bandwidth

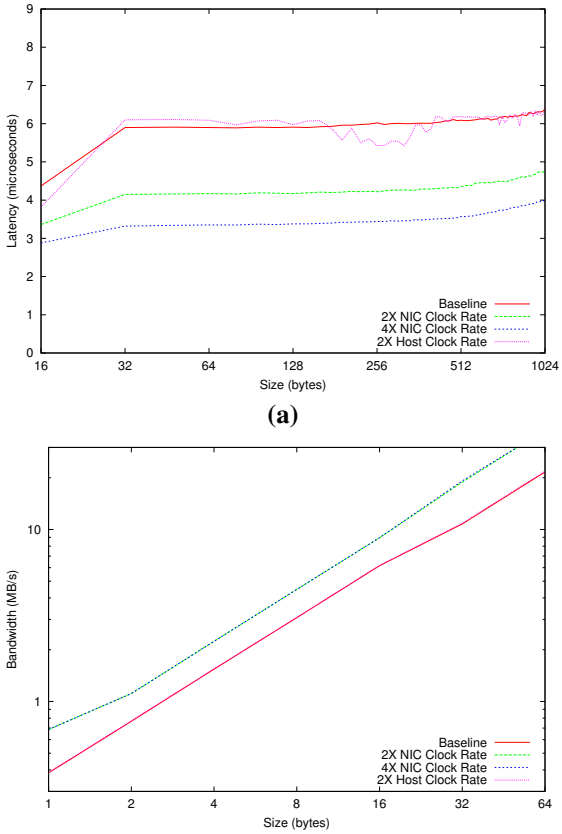


Figure 8. Impact of changing clock frequencies on (a) latency and (b) streaming bandwidth

The latency of accesses on the bus in the Seastar chip were also somewhat higher than originally predicted. The bus latency affects the performance of every processing operation performed by the PowerPC. It takes longer to read status registers, longer to access the local SRAM, and longer to write commands. Figure 7(a) indicates that an 8% latency reduction can be achieved by halving the bus latency; however, further improvements in bus latency have minimal impacts as other points become the bottleneck.

Halving the Seastar bus latency increased small message throughput, which increased streaming bandwidth by approximately 15% as seen in Figure 7(b). Only small messages are shown so that the effect can be seen, but the advantages persist all the way to 2 KB messages. Beyond that, bandwidth effects dominate the time. As with the latency, the improvements from further improving bus latency are minimal.

Another potential impact on performance comes from the clock frequency of the Seastar chip. Because the Seastar chip (NIC) is a standard cell design, it only runs at 500

MHz; however, it is clear that it is possible to do designs on a 130 nm process at 1 GHz or even 2 GHz. Figure 8 compares the impacts of changing various clock rates in the system. We get large advantages (30%) in latency for a 1GHz clock on the Seastar NIC and an additional 20% for increasing that to 2 GHz. Streaming bandwidth, however, is a different story. While we see an enormous 45% improvement in streaming bandwidth for a 1 GHz NIC clock, the increase to a 2 GHz NIC clock yields almost nothing as the host side processing becomes the bottleneck.

Unlike changing the NIC clock, changing just the host clock has virtually no impact. This is because most of the processing happens on the NIC and the NIC is a significant bottleneck with a 500 MHz clock (as seen by the impacts of changing the NIC clock). While the latency results for changing the host clock in Figure 8(a) look slightly odd, they correlate to our experience on the real system. There is a quantization of time changes caused by the main polling loop in the firmware. Thus, making some parts of the system slightly faster can make the MPI latency result worse.

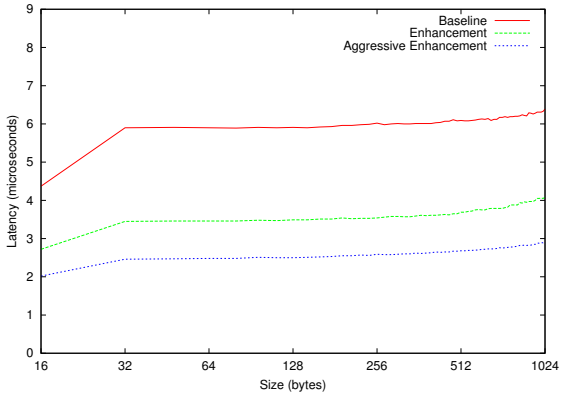
The final comparison, shown in Figure 9, considers two combinations of enhancements. The host processor performance is held constant. For the "Enhancement" line, the HT latency is reduced in half, the Seastar bus latency is reduced in half, and the Seastar clock rate is increased by a factor of two. With a slightly more aggressive design point, each of these should be achievable. This combination of improvements would yield a 45% improvement in latency and a comparable improvement in streaming bandwidth.

The "Aggressive Enhancement" line doubles HT bandwidth, quarters HT latency, quarters the bus latency, and increases the Seastar clock rate by a factor of four. While it yields another 30% improvement in MPI latency, it is an extremely aggressive design point. Streaming bandwidth is still constrained at smaller message sizes by MPI processing. After 2 KB, it starts to see an advantage from the extra HT bandwidth provided.

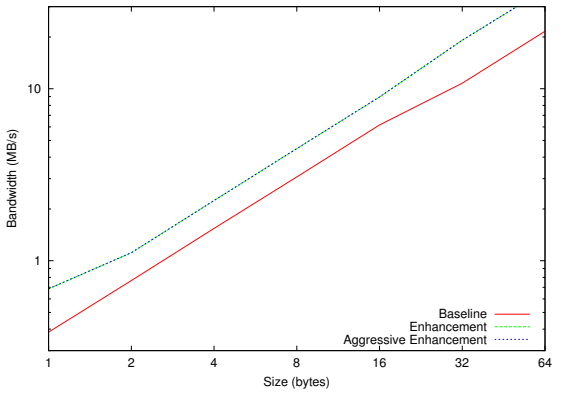
7 Conclusions

This paper has presented an overview of an initial version of a system simulation framework. The framework had to undergo numerous adaptations to allow us to accurately simulate a pair of Red Storm nodes. These changes highlight the challenges of accurately simulating production hardware in traditional simulation environments. However, they also illustrate that it is possible to build an environment that is capable of running virtually unmodified production software in a simulated system. This will become an important capability in the long run as systems become increasingly more complex.

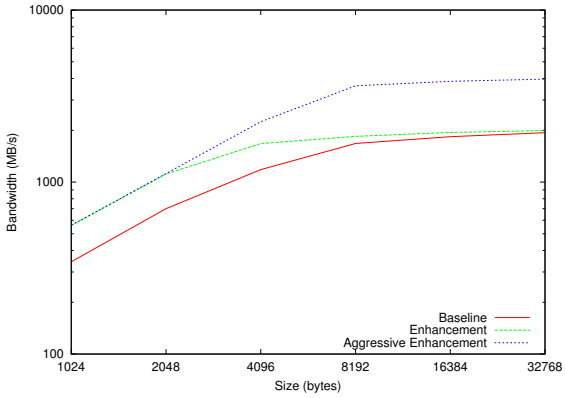
We also present a study using the modified simulator. Beginning with a validated model of the Red Storm node,



(a)



(b)



(c)

Figure 9. Impact of enhanced network interface designs on (a) latency, (b) small message streaming bandwidth, and (c) large message streaming bandwidth

we demonstrate that relatively minor changes to the performance of the network interface (Seastar) could yield significant improvements in network performance. Most notably, a small reduction in bus latency could have a 15% impact on streaming bandwidth. At the extreme end, increasing the Seastar clock rate by $2\times$ could yield an impressive reduction in latency by 40% and increase in streaming bandwidth by 45

8 Future Work

The changes to SST to facilitate modeling of Red Storm were performed in a branch of the tree to quickly demonstrate feasibility before implementing the concepts in the primary tree. As such, now that the model is "close enough" we have begun porting the changes into the primary tree — incorporating things we have learned along the way. There are still some aspects of the simulation environment that add inaccuracies to the system model that we intend to fix during the port. Primary among these is the lack of cache manipulation instructions in the PowerPC model. A lesser issue is the inability to model clocks that are not multiples of the primary simulation clock. In addition, the timing of the data access on memory reads in the SimpleScalar model raises concerns, although it has not been a problem in practice. We intend to address all of these issues as the changes are implemented in the primary version of the tree. Finally, we are in the process of releasing the simulator with an open source license.

References

[1] R. Alverson. Red Storm. In *Invited Talk, Hot Chips 15*, August 2003.

[2] Apple Computer, Inc. *Mach-O Runtime Architecture*, August 2003.

[3] R. Brightwell, T. Hudson, K. T. Pedretti, and K. D. Underwood. SeaStar interconnect: Balanced bandwidth for scalable performance. *IEEE Micro*, 26(3), May/June 2006.

[4] R. Brightwell, T. B. Hudson, A. B. Maccabe, and R. E. Riesen. The Portals 3.0 message passing interface revision 2.0. Technical Report SAND2006-0420, Sandia National Laboratories, January 2006.

[5] R. Brightwell, W. Lawry, A. B. Maccabe, and R. Riesen. Portals 3.0: Protocol building blocks for low overhead communication. In *Proceedings of the 2002 Workshop on Communication Architecture for Clusters*, April 2002.

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

[7] J. Emer, P. Ahuja, E. Borch, A. Klauser, C.-K. Luk, S. Manne, S. S. Mukherjee, H. Patil, S. Wallace, N. Binkert, R. Espasa, and T. Juan. Asim: A performance model framework. *Computer*, 35(2):68–76, 2002.

[8] S. Herrod, M. Rosenblum, E. Bugnion, S. Devine, R. Bosch, J. Chapin, K. Govil, D. Teodosiu, E. Witchel, and B. Verghese. *The SimOS Simulation Enviornment*. Stanford University, 1998.

[9] J. Huang. *The SImulator for Multithreaded Computer Architecture*. University of Minnesota, Minneapolis, 1.2 edition, June 2000.

[10] S. M. Kelly and R. Brightwell. Software architecture of the light weight kernel, Catamount. In *Proceedings of the 2005 Cray User Group Annual Technical Conference*, May 2005.

[11] J. R. Larus. *Computer Organization and Design: The Hardware/Software Interface*, chapter Appendix A. Morgan Kaufmann, third edition, August 2004.

[12] E. A. Lee. Overview of the ptolemy project. Technical Report M03/25, University of California, Berkeley, July 2003.

[13] P. S. Magnusson, M. Christensson, J. Eskilson, D. Forsgren, G. Hallberg, J. Hogberg, F. Larsson, A. Moestedt, and B. Werner. Simics: A full system simulation platform. *IEEE Computer*, 35(2), February 2002.

[14] M. M. Martin, D. J. Sorin, B. M. Beckmann, M. R. Marty, M. Xu, A. R. Alameldeen, K. E. Moore, M. D. Hill, and D. A. Wood. Multifacet's General Execution-driven Multiprocessor Simulator Toolset. *Computer Architecture News (CAN)*, 2005 (TBD).

[15] Motorola Inc. *MPC7400 RISC Microprocessor User's Manual*, March 2000.

[16] Motorola System Performance Modeling and Simulation Group. *Sim_G4 v1.4.1 User's Guide*, 1998. Available as part of Apple Computer's CHUD tool suite.

[17] A. Ortiz, J. Ortega, A. F. Daz, and A. Prieto. Protocol offload evaluation using simics. In *Proceedings of the 2006 IEEE International Conference on Cluster Computing*, September 2006.

[18] A. Rodrigues. Enkidu discrete event simulation framework. Technical Report TR04-14, University of Notre Dame, 2004.

[19] A. Rodrigues. *Programming Future Architectures: Dusty Decks, Memory Walls, and the Speed of Light*, chapter 3, pages 56–81. University of Notre Dame, 2006.

[20] L. Schaelicke and M. Parker. Ml-rsim reference manual. Technical Report TR04-10, University of Notre Dame, Notre Dame, Ind., 2002.

[21] A. Symons and V. L. Narasimhan. The design and application of PARSIM - a message passing computer simulator, 1997.

[22] *SystemC User's Guide*, 2.0 edition.

[23] M. Vachharajani, N. Vachharajani, D. A. Penry, J. Blome, and D. I. August. The liberty simulation enviornment, version 1.0. *Performance Evaluation Review: Special Issue on Tools for Architecture Research*, 31(4), March 2004.