# Interconnect Emulator for Aiding Performance Analysis of Distributed Memory Applications

Qi Wang[1,2], Ludmila Cherkasova[1], Jun Li[1], Haris Volos[1]

[1]Hewlett Packard Labs [2]The George Washington University

interwq@gwu.edu, lucy.cherkasova@hpe.com, jun.li@hpe.com, haris.volos@hpe.com

## ABSTRACT

Many modern large graph and Big Data processing applications operate on datasets that do not fit into DRAM of a single machine. This leads to a design of scale-out applications, where the application dataset is partitioned and processed by a cluster of machines. Typically, these applications rely on high speed interconnects which employ Remote Direct Memory Access (RDMA) technology to provide fast and high bandwidth communications. Distributed memory applications exhibit complex behavior: they tend to interleave computations and communications, use bursty transfers, and utilize global synchronization primitives. This makes it difficult to analyze the impact of communication layer on the application performance and answer the questions: how interconnect latency or bandwidth characteristics may change the application performance? will the application performance scale when processed by a larger system? In this work,[1] we introduce a novel emulation framework, called *InterSense*, which is implemented on top of existing high-speed interconnect, such as InfiniBand, and which provides two performance knobs for changing the (today's) interconnect bandwidth and latency. This approach offers an easy-to-use framework for a sensitivity analysis of complex distributed applications to communication layer performance instead of creating customized and time-consuming application models to answer the same questions. We evaluate the emulator accuracy with popular OSU MPI benchmark suite and two clusters with different generation InfiniBand interconnects (DDR and FDR): *InterSense* emulates the specified *bandwidth and latency* values with less than 2% error between the expected and measured values. *InterSense* supports an efficient emulation of a wide range of interconnect latencies and bandwidth characteristics for enabling performance and scalability analysis of Big Data applications, deriving and interpolating their requirements for performance characteristics of the underlying communication layer. To demonstrate the *InterSense*'s ease of use, we present a case study, where we apply *InterSense* for sensitivity analysis of four applications and benchmarks for getting non-trivial insights.

**Keywords**: Performance emulation; InfiniBand; MPI; distributed shared memory; benchmarking; profiling

## 1. INTRODUCTION

Exponential increase in online data and a corresponding proliferation of data-centric applications (Big Data analytics) forces system architects to revisit assumptions and requirements of the future system design, and at the same time, it challenges the application designers to tune and optimize their applications' implementation to efficiently utilize underlying hardware and its performance characteristics.

As a working set size of modern applications grows, to hold the entire dataset in main memory requires more than a single machine. This leads to a scale-out, distributed application implementation on a cluster of machines, where each server handles a portion of the complete dataset, and needs to communicate with each other to synchronize the main processing phases. Message passing interface (MPI) is a popular and widely used programming paradigm for scale-out, distributed memory applications. Performance of distributed memory applications inherently depends on performance of communication layer in the cluster. One can execute MPI programs using a traditional TCP/IP based network. However, over last decade, traditional networking often gets replaced by high-speed interconnects with Remote Direct Memory Access (RDMA) technology for optimizing performance of distributed memory applications. During last couple years, many Big Data applications, such as Hadoop, Spark, Memcached, etc., were re-written to take advantage of high-performance RDMA-capable interconnects [22, 23, 14, 13] which provide fast and high-bandwidth communications. The application analysis of potential performance improvements due to faster and higher bandwidth interconnects is a challenging task. Does the existing application implementation take a full advantage of the underlying interconnect or not? Will the application performance get worse if the interconnect has X% increased latency or Y% lower bandwidth?

With MPI, a distributed memory application runs on multiple machines as separate computation processes. These processes are also responsible for handling the communications, which can be a significant portion of the execution. In a new, popular, large graph processing benchmark Graph 500 [2], MPI communications could easily consume more than 50% of the program execution time (for larger graphs and smaller cluster sizes). Moreover, depending on the performance characteristics of the underlying interconnect, dif-

[1]This work was originated and largely completed during Qi Wang' summer internship at Hewlett Packard Labs.


*ICPE 2016,* March 12-18, 2016, Delft, Netherlands.
 ISBN 978-1-4503-4080-9/16/03 ...$15.00
DOI: http://dx.doi.org/10.1145//2851553.2851574.

ferent implementation decisions on communication style and size of transfers are made as a part of program optimization [12, 11, 18]. Many other scale-out applications [7, 13, 4] are also reported to be sensitive to either communication latency or communication bandwidth.

Complex MPI-based programs might interleave communication portions with computational ones in different patterns which makes it difficult to perform an accurate analysis of a communication layer impact on application performance and predict scaling properties of these programs. Building an accurate application model for predicting the application performance as a function of bandwidth and latency of underlying interconnect could be a very challenging and time-consuming task. Typically, such customized application model requires deep understanding of application functionality and its implementation, and could additionally necessitate detailed application profiling. Currently, it is practically impossible to analyze the application sensitivity to performance characteristics of the underlying interconnect and to answer the question: what impact the changed interconnect latency or/and bandwidth may have on performance of these applications?

To enable the analysis of an application dependency on performance characteristics of the underlying interconnect, we aim to offer an emulation framework, called *InterSense*, with *two performance knobs* for changing the interconnect perceived *bandwidth* and *latency*. Our earlier short paper [21] sketches the initial design of the interconnect performance emulator and provides preliminary evidence of its effectiveness. Here, we provide a complete description of our implementation along with discussion of different design alternatives and implementation challenges related to these choices. In the paper, we discuss technical subtleties of implementing the low-overhead emulation for high-speed interconnects and for decoupling latency and bandwidth emulations. We evaluate the emulator accuracy and the imposed overhead with popular OSU MPI benchmark suite [6] and two cluster-testbeds deployed with different generation InfiniBand interconnects (DDR and FDR): *InterSense* emulates the specified *bandwidth and latency* values with less than 2% error between the expected and measured values. To demonstrate the *InterSense*'s ease of use, we present a case study, where we apply *InterSense* for sensitivity analysis of modern applications and popular benchmarks, such as *Memcached* application [13], *NAS Parallel Benchmarks suite* [7], *RandomAccess memory benchmark (GUPS)* [4], and *Graph 500* benchmark [2].

Forward-looking projects like Firebox [9] and HP's The Machine [3] envision future scale-out computing architectures with enormous amount of non-volatile memories (NVMs) and with nodes connected via a high-speed interconnect. While the slowed-down interconnect appears to emulate a slower communication layer, *InterSense* can be used as a tool to predict the performance of future higher-performance systems. Intuitively, these future systems will have a higher computation-to-communication ratio than today's systems. Our tool can assist system designers in useful assessment of computation-to-communication ratio ranges that could support expected (or desired) performance of popular applications. We believe that the designed emulator can be used for conducting application sensitivity analysis, interpolating the application scalability, and projecting its performance on future interconnects with different performance characteristics.

The remainder of the paper is organized as follows. Section 2 provides background on MPI programming, outlines our approach to the emulator design, and discusses implementation challenges. Section 3 introduces our approach to bandwidth emulation and subtleties of its implementation. Section 4 outlines the latency control in our emulator, possible implementation choices, and our solution. Section 5 presents the evaluation study: assessment of emulator accuracy and its ease of use. Section 6 describes a review of related work. Finally, Section 7 provides summary and future work directions.

## 2. INTERCONNECT PERFORMANCE EMULATOR: ITS DESIGN AND IMPLEMENTATION CHALLENGES

Message Passing Interface (MPI) offers a language-independent communications protocol for implementing parallel and distributed memory applications. The MPI paradigm is attractive due to its wide portability: it can be used for communication by distributed-memory and shared-memory multiprocessors, as well as by clusters of servers. The MPI framework is applicable in different settings, it is independent of network speed or memory architecture. **Figure 1** shows the underlying communication methods available and used in MPI libraries over different media (that utilize different protocols). We use this diagram in order to explain the communication layer addressed by our interconnect emulator.

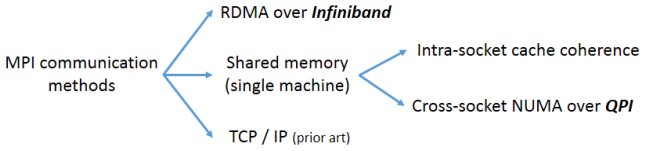



**Figure 1: MPI Communication Methods**

One can execute MPI programs using a *traditional TCP/IP based network* (the *bottom branch*). This communication style is based on a traditional (slow) networking, and therefore, one can apply some existing software-based networking emulators (e.g., *ModelNet, Netbed*, or *netem* [20, 8, 24]) that were designed and actively exploited for controlling network performance characteristics in the analysis of their impact on the application performance.

There are two branches (top and middle ones) that rely on a much faster and efficient media. The *middle branch* supports MPI program implementation on *shared memory* machines. There are two different sub-cases:

- different MPI processes are assigned and executed on the same socket (e.g., executed by different cores of the same processor). This communication style and its implementation relies on *intra-socket cache coherence* protocol for sending messages across MPI processes;
- different MPI processes are assigned and executed by different sockets (e.g., executed by different cores of different processors). This communication style and its implementation relies on *inter-socket or cross-socket (NUMA) cache coherence* protocol and QPI (Quick Path Interconnect between the processors) for sending messages across MPI processes.

The *top branch* is related to a cluster of machines connected via InfiniBand. This configuration allows implementing large distributed memory (combined across all the machines) and commu-

nications between the machines (i.e., between MPI processes residing on different machines) is done by using Remote Direct Memory Access over InfiniBand (*RDMA over InfiniBand*). The MPI communications performed over InfiniBand is the target of our work.

**InfiniBand** is the state of the art approach for high-speed interconnect between multiple machines. Different from QPI interconnect which requires CPU involvement in transmission, InfiniBand adapter is RDMA enabled and accepts requests actively. This enables the transmission to be asynchronous and without consuming CPU processing power. Based on the performance/features of InfiniBand, we choose it to study performance of future *RDMA-like devices*.

The large-scale Internet environment and high-speed interconnects have substantially different characteristics. For example, InfiniBand achieves bandwidth above 100 Gb/s and latency lower than 1 microsecond. These performance characteristics differences prevent the use of existing networking emulators for the interconnect emulation due to their high overhead. The high-speed interconnect emulation poses a number of challenges:

- Because of the high-speed nature of interconnect, the emulation overhead needs to be minimized. Imposing an additional software layer (e.g., TCP/IP protocol) or extra hardware (e.g., *ModelNet core node*) is not acceptable for interconnect emulation because of the overhead.
- Latency and bandwidth characteristics of the interconnect emulation need to be orthogonal to each other, i.e., the emulation mechanism of one characteristic should not interfere with the other. Therefore, separate latency and bandwidth emulation mechanisms are required.
- There is no bandwidth or latency control support in existing hardware, e.g., there are no hardware knobs in InfiniBand adapters/switches that can be used for emulation of different performance characteristics. By the way, the QPI situation is similar.

To implement the low-overhead emulation and to decouple latency and bandwidth emulations, we designed the following software techniques for a bandwidth and latency control as summarized in Figure 2:

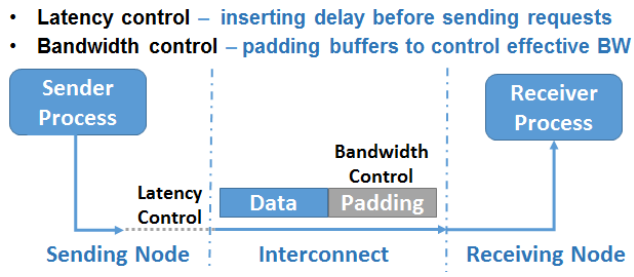



**Figure 2: Emulation Mechanism Overview.**

- *For bandwidth emulation*, we add padding packets to reduce an effective bandwidth for applications.
- *For latency emulation*, we insert a software emulated delay before sending the application's messages.

One additional interesting point to note here is that the latency emulation approach described later in Section 4 can be also applied to control the latency via shared memory (middle branch shown in Figure 1).

## 3. BANDWIDTH EMULATION

For emulating interconnect with different bandwidth characteristics, we impact effective bandwidth of the interconnect by sending extra padding packets. The ratio between padding packets and data packets determines the effective bandwidth for applications. It is defined in a software emulation layer so that we can achieve a fine-grained control over effective bandwidth. There are multiple layers in software, where bandwidth can be impacted:

1. an application,
2. the communication library, i.e., MPI library, and
3. a device driver.

Figure 3 summarizes advantages and disadvantages of each layer for bandwidth throttling.

| Padding Layer | Pluses and Minuses |
|---|---|
| Application | - : Application specific<br>- : Modification complexity |
| MPI library | +: Portable for MPI applications<br>+: MPI library support – MPI information available<br>- : High complexity of MPI implementations |
| IB driver library | +: Accuracy – padding right before sending<br>- : Complexity – missing high-level library support<br>- : Non-portable – device specific |

**Figure 3: Bandwidth Control Layer.**

If we implement an interconnect bandwidth emulation in the *application layer* this requires only modification to the application itself. However, this may cause many lines of changes to the application code (and in many cases, the application source code might not be available). Also, for some MPI operations that perform both computation and communication, e.g., *MPI_Allreduce*, it is not possible to impact the interconnect bandwidth accurately because the communication pattern, implemented in the underlying MPI library, is transparent to the application.

On the other end, in the *device driver layer* (e.g., driver library of InfiniBand adapters), we would communicate with devices directly. However, at this layer, we are losing a higher level operation view, and have no information from MPI library. In addition, the driver library is *device specific*, and the process of setting up and sending of padding packets are two rather complex operations. This means that if the bandwidth control is done in the driver layer, migrating (re-implementing) the emulation platform to a different hardware would be difficult.

We believe the **MPI library layer** offers the *best trade-off* among portability, accuracy and complexity: it works for all MPI-based applications, it is close to hardware (right above the driver layer), and it provides additional flexibility of seeing the MPI operators used by the program. This information can be used in the

emulator for exploiting different bandwidth values across the interconnect links to mimic special interconnect topologies or resources available to the program.

To emulate correct bandwidth impact on packet latencies, a padding packet is sent before the corresponding data packet. Bandwidth does have a natural impact on the latency of larger packets. By sending the padding packets before the actual data packets we aim to correctly capture the bandwidth characteristic (i.e., latency of the large packets will increase with lower bandwidth). To avoid impacting the latency of small packets, their padding packets are batched (combined together) until the total size reaches a pre-defined threshold (64 KB in our experiments). The threshold parameter should be large enough to amortize software overhead, while not too large that bandwidth is not controlled in time. We found that for InfiniBand FDR interconnect with 56 Gbits/sec, 64 KB threshold is sufficient to ensure both no latency interference for small packets and the accurate bandwidth control.

We implement bandwidth control in a widely-used MPI implementation: MVAPICH2 [5]. To send padding packets, memory buffer needs to be allocated as for sending and receiving padding packets.

There are a few padding buffer allocation options:

- *Naive approach*: allocate a buffer with a size equal to the largest application message multiplied by a padding ratio. However, this may result in a very high memory requirement for padding buffers: the maximum application message size could be very large and its size is unknown to the MPI library.
- *Buffer-reusable approach*: allocate a buffer with a size equal to the largest MPI packet segmentation size. Since MPI library does packet segmentation internally (with a fixed maximum segmentation size: 4MB in our case), the size of packets at the interconnect level will not be greater than the segmentation size. Based on this, we can point all the padding packets to the same memory region. This requires the minimal amount of memory for padding buffer. We use this approach in our emulator.

Moreover, inside the MPI library, there are a few implementation choices, where adding of padding packets can be done: high-level interface layer (closer to MPI's user API) or low-level device API layer (closer to hardware, e.g., InfiniBand Verbs API). We prototyped and compared these two implementation alternatives for the interconnect bandwidth control:

- *High-level MPI interface layer* approach – it requires no deep modification of MPI library. A new *padding_send* operation is attached to each *send*-based MPI operation to consume desired bandwidth. For example, *MPI_Send()* will send a separate padding packet along with the data packet. To avoid unnecessary receiving calls for padding packets, which add overhead and prevent batching for small packets, one-sided communication is used to send padding packets.

  However, the considered MPI's user API level imposes the following implementation requirement: the described "padding" modification should be introduced for **all** API functions. Unfortunately, there are MPI functions with a high complexity (e.g., *MPI_Allreduce*) that cannot be easily modified to impact bandwidth accurately. This makes the high-level API approach less promising for achieving an accurate interconnect bandwidth throttling outcome.

- *Low-level MPI Verbs-based layer* – it needs modifications at a deeper level inside the MPI library, such as InfiniBand Verbs layer. For InfiniBand, the *padding_send* operation is attached to each *ibv_post_send*() invocation. The operating level here is closer to hardware, i.e., right before communicating with hardware driver. In such a way, this low-level approach is much more compact and provides a *unified method* for handling bandwidth characteristics across **all** high-level MPI operations.

Therefore, for implementing the interconnect bandwidth control, we add padding packets at a low-level Verbs layer of MPI library. For emulating the correct bandwidth impact on large packet latencies, the padding packet is sent before the corresponding data packet.

# 4. LATENCY EMULATION

Since high-speed interconnects have an ultra low native latency (e.g., 1 $\mu$s for InfiniBand), the emulation overhead needs to be very low as well. This means that expensive operations like context switches cannot be involved. Our approach for emulating the interconnect latency is to generate an additional delay by *spinning*, i.e., by introducing an extra *idle time* for desired latency. A spinning approach has the following advantages:

1. a high accuracy (close to 10ns),
2. a low implementation complexity, and
3. a low overhead (no other operations are needed).

The only disadvantage is that a spinning process results in additional consumption of CPU computation cycles [2]. However, we believe that the computation cycles spent on spin is acceptable because the desirable emulation targets of the interconnect latency emulation are at nanosecond or microsecond level (depending on the interconnect).

Similar to the bandwidth control implementation, multiple layers in the software stack can be used for impacting the interconnect latency. Figure 4 summarizes advantages and disadvantages of each layer.

For latency control of InfiniBand, because of the spin implementation simplicity, we choose the **driver library layer**. In addition, this approach aims to maximize the emulation accuracy (despite the driver's portability limitation). When migrating the emulation to a different hardware platform, the effort of re-implementing the spin-based latency control in a different driver library is minimal – it is approximately 20 lines of code in our case.

When a driver specific modification is not preferred, the latency control can also be done in a *low-level MPI library layer* (i.e., before calling *ib_verb* API) to achieve an MPI-portable implementation. However, there is a *limitation* of this approach: the delay inserted in the MPI library layer could be invalid when *multi-threading is enabled* in the MPI library because multiple threads could be contending for locks in a driver after inserting their delays.

For latency control over shared memory (a middle branch in

[2] One can use dedicated resources for spinning (e.g., spinning cores) to alleviate competing with application for the same CPU cycles. However, it has a different complication: in order to inform the spin cores about the packet, message passing is required, e.g., using lock-free ring buffers. A message between cores itself has a rather high overhead (approx. 100ns per message within a socket), and it goes up dramatically when there is a contention. So, this approach will reduce the spin cycle consumption (minus the message overhead), but will cause an extra cache coherency traffic.

| Delay Insertion Layer | Pluses and Minuses |
|---|---|
| Application | +: Trivial to implement<br>- : Application specific<br>- : Inaccurate for asynchronous requests |
| MPI library | +: Portable for MPI applications<br>- : High complexity of MPI implementations<br>- : Not close to hardware (Infiniband) |
| IB driver library | +: Accuracy – delay right before sending<br>- : Non-portable – device specific |

**Figure 4: Latency Control Layer.**

Figure 1) a similar approach can be applied. The communication latency over shared memory and/or QPI can be controlled in the MPI library layer: right before writing to shared memory (since no driver is needed for QPI communication due to direct memory operations using a shared memory protocol). When using this approach one needs to remember a granularity of introduced additional delay relative to a basic latency.

# 5. EVALUATION

In this section, we evaluate the emulator accuracy and demonstrate its ease of use for application sensitivity analysis by presenting a case study with four distributed-memory applications and benchmarks.

## 5.1 Experimental Testbeds and Workloads

We evaluate the effectiveness and accuracy of our emulator by using a popular OSU MPI benchmark suite [6] and two clusters with different generation interconnects: DDR InfiniBand (20 Gbits/s) and FDR InfiniBand (56 Gbits/s):

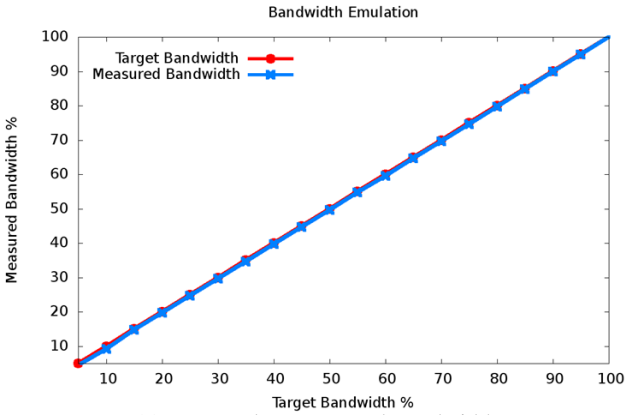
- *Cluster_1* with 4 nodes, where each node is based on HP DL380 servers (two sockets Xeon E5-2697 with 12 cores per socket), and connected with FDR InfiniBand (56 Gbits/s);
- *Cluster_2* with 8 nodes, where each node is based on HP Proliant BL460c servers (two sockets Xeon E5345 with 12 cores per socket), and connected with DDR InfiniBand (20 Gbits/s).

The Ohio MPI Microbenchmark suite [6] is a collection of independent MPI message passing performance microbenchmarks developed and provided as an open source by the Network-Based Computing Laboratory of the Ohio State University. In particular, it includes some simple benchmarks for performance measurements of latency and bandwidth for basic MPI communications.

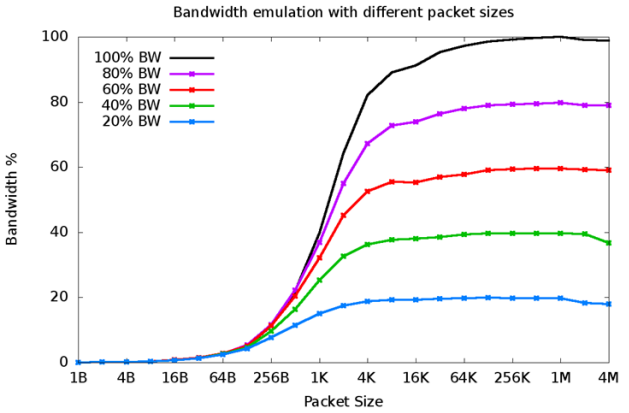
## 5.2 Emulator Accuracy

**OSU MPI bandwidth test** is implemented by having a sender sending out a fixed number (equal to a window size) of back-to-back messages to a receiver and then waiting for a reply from the receiver. The receiver sends the reply only after receiving all the messages. This process is repeated for several iterations and the bandwidth is calculated based on the elapsed time (from the time sender sends the first message until the time it receives the reply back from the receiver) and the number of bytes sent by the sender. The objective of this bandwidth test is to determine the maximum sustained data rate that can be achieved at the network level. Thus, non-blocking version of MPI functions (*MPI_Isend* and *MPI_Irecv*) are used in the test.

**Figure 5 (a)** shows the bandwidth emulation results measured with OSU MPI benchmark executed on the FDR InfiniBand-based Cluster_1 (with 56 Gbits/s links). The X-axis shows the target, emulated bandwidth (aimed between the sender and receiver), while Y-axis reports on the measured bandwidth. We have executed benchmark runs with default parameters of the bandwidth test program: each configuration is run for $20 \times 64$ iterations. From the




(a) Expected vs Measured Bandwidth.




(b) Bandwidth Control for Packets with Different Sizes.

**Figure 5: OSU benchmark: Evaluating Accuracy of Interconnect Bandwidth Emulation.**

experimental results, we observe that the interconnect *bandwidth emulation* is supported with a high accuracy: less than 2% error between the expected, emulated interconnect bandwidth and the measured interconnect bandwidth in the experiments (packet size = 1 MB).

Moreover, the bandwidth control works correctly for packets with different sizes as shown in **Figure 5 (b)**: large packets are impacted by bandwidth control, while small packets are not impacted because they are not limited by bandwidth.

- The large packets (i.e., packets $\geq$ 128 KB) are capable of fully utilizing the available interconnect bandwidth. We can

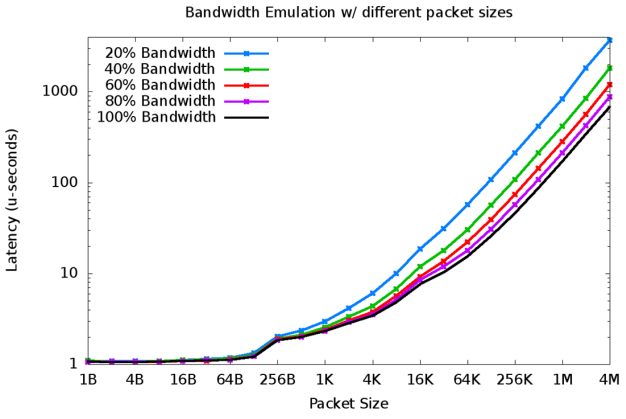



**Figure 6: Bandwidth Impact on Packet Latency.**

see that when 100% of bandwidth is available, the large packets are able to saturate and utilize 100% of interconnect bandwidth (top line). If only 80% of bandwidth is available, the large packets are utilizing 80% of bandwidth, etc. Therefore transfers of large packets are impacted correctly by bandwidth control.

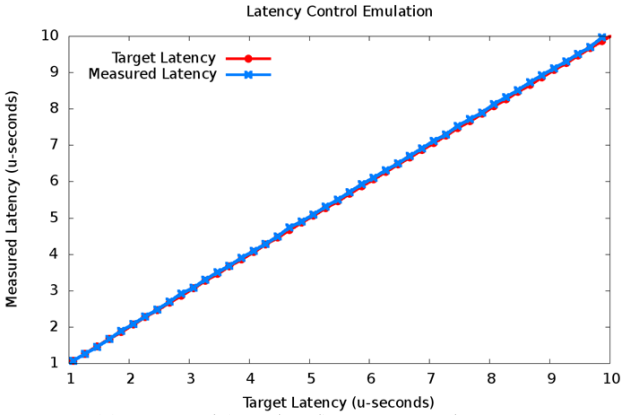
- The small size packet transfers (less than 256 bytes) are not impacted by bandwidth emulation. Their transfers are not limited by the interconnect bandwidth (small packets are not capable of utilizing the available bandwidth). Small size transfers are limited by the interconnect message rate. Therefore, the achievable bandwidth for small size packets is practically the same under different emulation values of interconnect bandwidth as shown in the left side of **Figure 5 (b)**.

We also measure the latency of different size packets under different emulated interconnect bandwidth. We find that these packets are controlled accurately in the emulator, and the bandwidth emulation is orthogonal to latency emulation. **Figure 6** shows that small packets (not limited by bandwidth) have no latency change under different emulated interconnect bandwidth control, while the latency of large packets (limited by bandwidth) is impacted correctly.

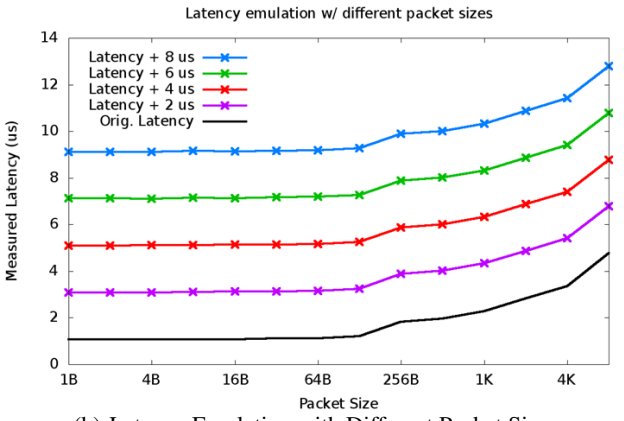
**OSU MPI latency test** is carried out in a ping-pong fashion. A sender sends a message with a certain data size to a receiver and waits for a reply from the receiver. The receiver receives the message from the sender and sends back a reply with the same data size. Many iterations of this ping-pong test are carried out and average one-way latency numbers are obtained. Synchronous version of MPI functions (*MPI_Send* and *MPI_Recv*) are used in the tests.

**Figure 7** shows the latency emulation results measured with OSU MPI benchmark executed on the FDR InfiniBand-based Cluster_1 (with 56 Gbits/s links).

The *latency emulation* results, measured with OSU MPI benchmark, again show high accuracy: $\leq 2\%$ error between expected (emulated) latency and measured one as shown in **Figure 7 (a)**. Moreover, the designed mechanism works effectively for packets with different sizes as demonstrated in **Figure 7 (b)**. The bottom line on this graph shows the measured latency of different size packets. As we can see, when we emulate the increased interconnect latency (i.e., *latency*$+2\mu s$, *latency*$+4\mu s$, etc.) the measured packet latency closely follows the original latency pattern. It means that the designed latency emulation mechanism does not impact the interconnect bandwidth, and therefore the emulation mechanisms for latency and bandwidth do not interfere with each other.




(a) Expected (Emulated) vs Measured Latency.




(b) Latency Emulation with Different Packet Sizes.

**Figure 7: OSU benchmark: Evaluating Accuracy of Interconnect Latency Emulation.**

The results obtained in the DDR InfiniBand-based Cluster_2 (with 20 Gbits/s links) show similar accuracy results. We omit them due to a paper space limitation.

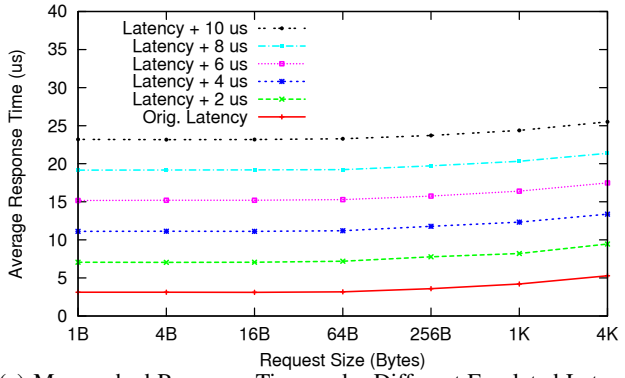
## 5.3 Application Sensitivity Analysis with InterSense

Complex MPI-based programs might interleave communication portions with computational ones in different patterns which makes it difficult to analyze the communication layer impact on application performance and predict scaling properties of the program. Currently, it is extremely challenging to analyze the application sensitivity to performance characteristics of the underlying interconnect and to answer the question: what impact the changed interconnect latency or/and bandwidth may have on performance of these applications?

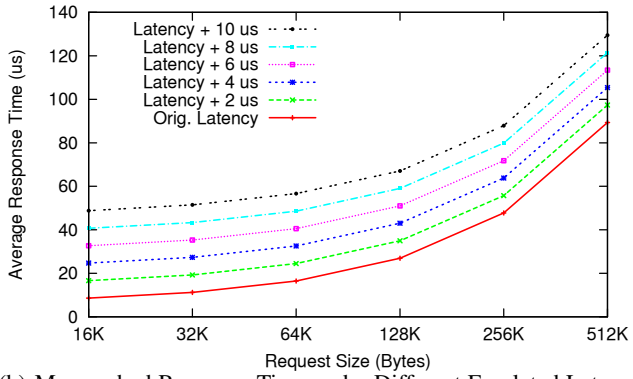
To demonstrate the *InterSense*'s ease of use, we present a case study, where we apply *InterSense* for a sensitivity analysis of modern applications and popular benchmarks, such as *Memcached* ap-

plication [13], *RandomAccess memory benchmark (GUPS)* [4], *NAS Parallel Benchmarks suite* [7], and *Graph 500* benchmark [2]. Note, that the only other way to get the results below is to build customized performance models of these benchmarks and applications as a function of interconnect latency and bandwidth, which is a difficult and challenging task even for a skilled performance analysts.

**Figure 8** shows performance of *Memcached* application [13] as a function of emulated (increased) interconnect latency. *Memcached* is a key-value distributed memory application used in the data-center environment for caching results of database calls, API calls, etc. This application is redesigned for RDMA capable networks instead of traditional BSD Sockets implementation. It shows an extremely low response time for small requests, and therefore, it is very sensitive to any increase in the interconnect latency, which directly impacts the application performance as shown in Figure 8 (a).




(a) Memcached Response Time under Different Emulated Latency and Small Request Sizes.




(b) Memcached Response Time under Different Emulated Latency and Large Request Sizes.

**Figure 8: Memcached Response Time under Different Emulated Latency and Larger Request Sizes.**

For example, an additional latency of $2\mu$s via interconnect increases the response time of small requests ($\leq$ 4KB) almost twice. For larger requests the relative impact of increased latency diminishes as shown in Figure 8 (b).

Another interesting point is that for small requests ($\leq$ 4 KB), when interconnect latency is increased by $N$ $\mu$s, the response time increases by $2 \times N$ $\mu$s as we can see in Figure 8 (a) because of a single round-trip communication that is required to serve this request. While for large requests ($\geq$ 16 KB) shown in Figure 8 (b), the response time increases by $4 \times N$ $\mu$s, because two round-trip communications are required for large requests: first one for setting up the memory buffer and the second one for actual data transfer. For small requests the data is embedded in the first packet. These differences are not obvious without understanding application implementation. However, our tool can discover this easily without having a deep knowledge of the application.

**Figure 9** shows a sensitivity of *RandomAccess memory benchmark (GUPS)* to the interconnect latency. *GUPS* is a new benchmark proposed by IBM Research [19] a few years ago for measuring how frequently a computer can issue updates to randomly generated RAM locations. Giga-updates per second (*GUPS*) is a measure of random memory access capability of multicores platforms. *GUPS* is latency sensitive, but in a very special way. Its sensitivity is defined by the number of outstanding concurrent requests. If the number of outstanding requests is limited ($\leq 1K$) then *GUPS* performance is 30-100% worse with increased interconnect latency. However, for outstanding requests $\geq 4K$ there is no difference in performance: the pipeline of processed requests is constantly full, and it hides the increased interconnect latency.

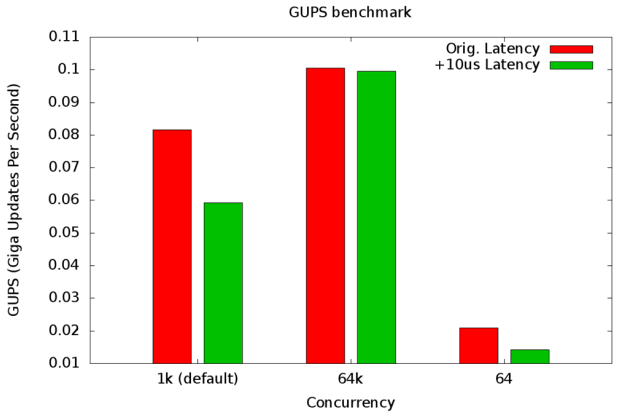



**Figure 9: GUPS: Emulated Latency.**

Both *Memcached* and *GUPS* are not bandwidth sensitive in our experiments: they operate with very small size requests and cannot utilize the available interconnect bandwidth in our *Cluster_1* testbed. Small size transfers are limited by the interconnect message rate.

**Figure 10** shows performance of a popular *NAS Parallel Benchmarks suite* [7] as a function of emulated bandwidth. *NAS Parallel Benchmarks* are used for the evaluation and comparison of parallel supercomputers. This suite includes seven diverse applications with different computation and communication patterns. The bandwidth sensitivity among benchmarks is very different: *LU, MG, SP,* and *BT* show 20-40% increase in the execution time at 2.8 Gb/s available bandwidth (i.e., at 5% of the original interconnect bandwidth). However, *CG* and *FT* react to diminished bandwidth in a more extreme way: their execution time increases by more than 300%. These applications are extremely bandwidth sensitive for delivering good performance and this dependency impacts the applications' scalability on a larger cluster. Finally, *EP* application

(*Embarrassingly Parallel*) is not sensitive to bandwidth at all. None of the NAS studied benchmarks is sensitive to the increased interconnect latency (up to 10x latency in our experiments) because they use extensively the asynchronous communications which hide the impact of communication latency.

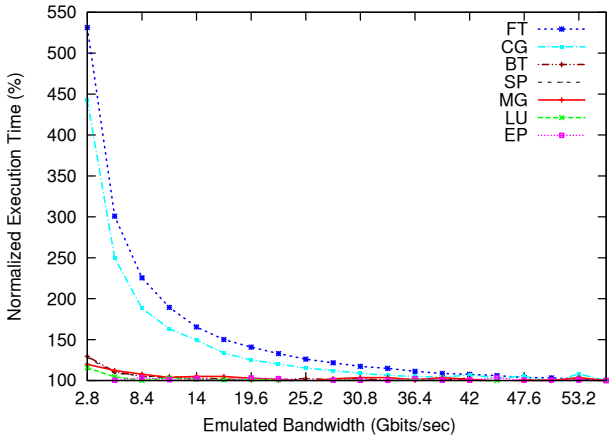



**Figure 10: NAS parallel Benchmarks: Emulated Bandwidth.**

**Figure 11** shows performance of *Graph 500* benchmark [2] with emulated interconnect bandwidth. *Graph 500* is a benchmark implementing a *Breadth First Search* algorithm on large graphs and used for assessing system performance based on processing efficiency of their memory and interconnect. Producing winning results is a goal for many companies with leading hardware and system design.

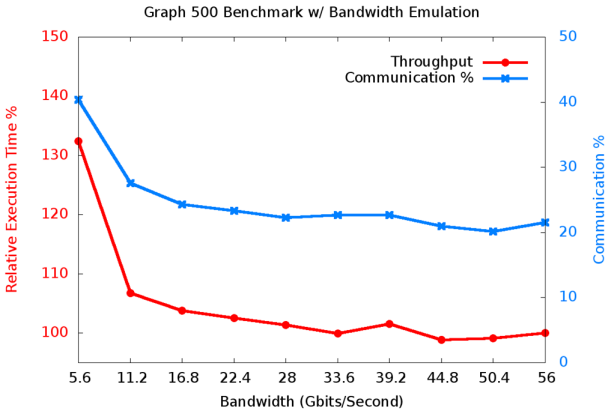



**Figure 11: Graph500: Emulated Bandwidth.**

Figure 11 shows that only 30% of FDR InfiniBand bandwidth is effectively used (for selected graph and cluster sizes). With 10% of interconnect bandwidth, the execution time (red line) is increased by 30%, and a fraction of communication time (blue line) is doubled. This type of sensitivity analysis is very useful for understanding the cost/performance trade-offs. *InterSense* can help answering capacity planning questions which require careful estimates of the available "remaining" interconnect capacity for supporting a higher load/volume service without compromising its performance.

*Graph 500* was not sensitive to a higher (10x) interconnect latency in our experiments (due to using asynchronous communication and high messaging volume).

This concludes our case study with *InterSense*. The goal of this study was to show the ease of tool use and the appeal of the designed interconnect emulator for performing the sensitivity analysis of emerging benchmarks and modern applications instead of creating customized and time-consuming application models to answer the same questions.

# 6. RELATED WORK

Many previous efforts explored emulation environments for evaluating the networking impact on their applications [10, 15, 17, 25]. However, the initial attempts were targeting static and relatively small scale systems. Later, more advanced efforts [20, 24, 16, 8] offered a variety of flexible TCP/IP-based emulation approaches to support a broad range of research efforts for evaluating the Internet and data center environments.

*ModelNet* [20] is a large-scale network emulator that allows users to evaluate distributed networked systems and analyze performance of their applications in the Internet-like environments. It employs virtualization and routes packets through control nodes (*ModelNet* core) to emulate desired delay, bandwidth and loss rate. It further performs full hop-by-hop network emulation, allowing it to capture the effects of contention and bursts in the middle of the network.

Another closely related to this approach is *Netbed* [24] (a descendant of *Emulab* [1]). This tool allows users to configure and access integrated network resources composed of emulated, simulated and wide-area nodes and links for distributed systems and networking experiments.

However, large-scale Internet environment and high-speed interconnect have substantially different characteristics: for Internet infrastructure, bandwidth and latency performance is hundred times worse than the InfiniBand ones. In addition, the scale, security, reliability issues are often of concern as well. These performance characteristics differences prevent the *ModelNet* approach from being applied for high-speed interconnect emulation. For example, the high overhead coming from virtualization and re-routing is not suitable for the interconnect emulation.

Another effort [16] applies the emulation for evaluating a variety of effects in wide-area network on web server performance. The authors advocate emulating network performance characteristics at end hosts rather than in the network core for improved and simplified scalability. While this approach requires appropriate emulation software on the edge nodes and must share each host CPU between the emulation and the target application, this approach is attractive due to its flexibility and simplicity. Our *InterSense* emulator follows a similar approach that offers performance knobs for controlling the interconnect latency and bandwidth at the end points.

*netem* [8] is an open source network emulation tool that is enabled in the Linux kernel. However, it also focuses on wide area networks like Internet, which makes it unfit for the interconnect emulation. Other related tools are available for QoS management over TCP/IP based network. But they all suffer from high protocol overhead, when running on a high-speed interconnect, such as IP over IB (IPoIB).

The network-related emulation tools are traditionally designed around TCP/IP protocol, which functionality is significantly dif-

ferent compared to RDMA over InfiniBand. The designed *InterSense* emulator offers unique capabilities for analysis of scale-out distributed memory applications.

# 7. CONCLUSION AND FUTURE WORK

In this work, we introduce novel bandwidth and latency control mechanisms for performance emulation of the high-speed interconnects. We built a prototype of a new emulator and carefully evaluated its performance, efficiency, and accuracy. *InterSense* can assist researchers and engineers in emulating a variety of performance characteristics of future large-scale interconnects and conducting the application sensitivity and scalability analysis dependent on these characteristics.

We are working on augmenting the proposed approach with additional profiling, modeling, and prediction technique. By performing the emulation in small deployments with increased interconnect latency and decreased bandwidth we aim to derive the predictive models for application performance when processing larger data amounts in large-scale distributed environments. We believe that the *InterSense* ability to accurately indicate the *needed* interconnect bandwidth for achieving the user-defined application performance objectives and to reflect the application sensitivity to the increased interconnect latency will help in applications' optimization and redesign.

# 8. REFERENCES


[1] Emulab - Network Emulation Testbed, `http://www.emulab.net/` .

[2] Graph 500 Benchmark. `www.graph500.org/` .

[3] HP Labs. The Machine: A new kind of computer. `http://www.hpl.hp.com/research/systems-research/` .

[4] HPCC RandomAccess (GUPS) Benchmark. `http://icl.cs.utk.edu/projectsfiles/hpcc/RandomAccess/` .

[5] MVAPICH: MPI over InfiniBand, 10GigE/iWARP and RoCE. `http://mvapich.cse.ohio-state.edu/`.

[6] MVAPICH Ohio State University Micro benchmark. `http://mvapich.cse.ohio-state.edu/benchmarks/`.

[7] NAS Parallel Benchmarks. `http://www.nas.nasa.gov/publications/npb.html` .

[8] netem, `http://www.linuxfoundation.org/collaborate/workgroups/networking/netem` .

[9] K. Asanovic. FireBox: A Hardware Building Block for 2020 Warehouse-Scale Computers. In *Proc. of FAST*, 2014.

[10] G. Banga, J. C. Mogul, and P. Druschel. A scalable and Explicit Event Delivery Mechanism for UNIX. In *Proc. of the USENIX Annual Technical Conference*, 1999.

[11] F. Checconi and F. Petrini. Traversing Trillions of Edges in Real Time: Graph Exploration on Large-Scale Parallel Machines. In *Proc. of Intl. Parallel and Distributed Processing Symposium, IPDPS'14*, 2014.

[12] F. Checconi, F. Petrini, J. Willcock, A. Lumsdaine, A. R. Choudhury, and Y. Sabharwal. Breaking the Speed and Scalability Barriers for Graph Exploration on Distributed-Memory Machines. In *Proc. of Conference on High Performance Computing Networking, Storage and Analysis, SC'12,*, 2012.

[13] J. Jose, H. Subramoni, M. Luo, M. Zhang, J. Huang, M. Wasi-ur Rahman, N. S. Islam, X. Ouyang, H. Wang, S. Sur, and D. K. Panda. Memcached Design on High Performance RDMA Capable Interconnects. In *Proc. of the 2011 International Conference on Parallel Processing*, ICPP '11, 2011.

[14] X. Lu, M. Wasi-ur Rahman, N. S. Islam, D. Shankar, , and D. K. D. Panda. Accelerating Spark with RDMA for Big Data Processing: Early Experiences. In *Proc. of Hot Interconnects*, 2014.

[15] R. P. Martin, A. M. Vahdat, D. E. Culler, and T. E. Anderson. Effects of Communication Latency, Overhead, and Bandwidth in a Cluster Architecture. In *Proc. of the 24th Annual International Symposium on Computer Architecture*, ISCA '97, 1997.

[16] E. M. Nahum, M.-C. Rosu, S. Seshan, and J. Almeida. The Effects of Wide-area Conditions on WWW Server Performance. In *Proc. of the 2001 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '01, 2001.

[17] B. D. Noble, M. Satyanarayanan, G. T. Nguyen, and R. H. Katz. Trace-Based Mobile Network Emulation. In *Proc. of SIGCOMM*, 1997.

[18] X. Que, F. Checconi, and F. Petrini. Performance Analysis of Graph Algorithms on P7IH. In *Proc. of the 29th Intl. Conference on Supercomputing, ISC'14*, 2014.

[19] V. Saxena, Y. Sabharwal, and P. Bhatotia. Performance evaluation and optimization of random memory access on multicores with high productivity. In *Proc. of Intl. Conference on High Performance Computing (HiPC)*, 2010.

[20] A. Vahdat, K. Yocum, K. Walsh, P. Mahadevan, D. Kostić, J. Chase, and D. Becker. Scalability and accuracy in a large-scale network emulator. *SIGOPS Oper. Syst. Rev.*, 36(SI), Dec. 2002.

[21] Q. Wang, L. Cherkasova, J. Li, and H. Volos. InterSense: Interconnect Performance Emulator for Future Scale-out Distributed Memory Applications. In *Intl. Symposium on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, 2015.

[22] M. Wasi-ur Rahman, N. S. Islam, X. Lu, J. Jose, H. Subramoni, H. Wang, and D. K. D. Panda. High-Performance RDMA-based Design of Hadoop MapReduce over InfiniBand. In *Proc. of the 2013 IEEE 27th International Symposium on Parallel and Distributed Processing Workshops and PhD Forum*, IPDPSW '13, 2013.

[23] M. Wasi-ur-Rahman, X. Lu, N. S. Islam, R. Rajachandrasekar, and D. K. Panda. MapReduce over Lustre: Can RDMA-Based Approach Benefit? In *Proc. of the 20th International Conference EuroPar*, 2014.

[24] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar. An Integrated Experimental Environment for Distributed Systems and Networks. *SIGOPS Oper. Syst. Rev.*, 36(SI), Dec. 2002.

[25] H. Yu and A. Vahdat. The Costs and Limits of Availability for Replicated Services. In *Proc. of the 18th ACM Symposium on Operating Systems Principles (SOSP)*, 2001.