On Achieving High Message Rates

Holger Fröning Mondrian Nüssle Heiner Litz Christian Leber Ulrich Brüning
Institute of Computer Engineering
University of Heidelberg
Mannheim, Germany
{holger.froening, mondrian.nuessle, heiner.litz, christian.leber, ulrich.bruening}@ziti.uni-heidelberg.de

***Abstract*— Computer systems continue to increase in parallelism in all areas. Stagnating single thread performance as well as power constraints prevent a reversal of this trend; on the contrary, current projections show that the trend towards parallelism will accelerate. In cluster computing, scalability, and therefore the degree of parallelism, is limited by the network interconnect and more specifically by the message rate it provides. We designed an interconnection network specifically for high message rates. Among other things, it reduces the burden on the software stack by relying on communication engines that perform a large fraction of the send and receive functionality in hardware. It also supports multi-core environments very efficiently through hardware-level virtualization of the communication engines. We provide details on the overall architecture, the thin software stack, performance results for a set of MPI-based benchmarks, and an in-depth analysis of how application performance depends on the message rate. We vary the message rate by software and hardware techniques, and measure the application-level impact of different message rates. We are also using this analysis to extrapolate performance for technologies with wider data paths and higher line rates.**

Keywords- computer communications, high performance networking, performance analysis, performance prediction

I. INTRODUCTION

Over the past years, there has been an increasing demand for more powerful computing systems. The TOP500 list [1] reveals that the vast majority of high performance computing systems are based on clusters, certainly due to their excellent price/performance ratio. Such clusters rely on commodity parts for computing, memory and enclosure, however, for highest performance they use specialized interconnection networks like Infiniband. Unlike clusters, *Massively Parallel Processors (MPPs)* make more use of specialized parts, which significantly increase performance but also cost.

The overarching goal of the EXTOLL project is to fill the gap between clusters and MPPs, specifically with a system that combines the performance of an MPP with the cost-effectiveness of a cluster. Workload analysis has determined that the message rate is one key characteristic that needs to be optimized to achieve this goal. In this paper, we will present a set of techniques utilized within the EXTOLL design, which enable high message rates, and show how applications can benefit from this improvement.

The message rate is defined as the number of messages that can be injected into a network from a host per second. Thus, it describes the achievable bandwidth for small message sizes. Latency is of paramount importance for round-trip communication patterns; however, for unidirectional transfers its impact is negligible. For such push-style communication patterns, the message rate is much more significant. Not only do MPI applications benefit from high message rates, PGAS-style applications also express fine-grained communication patterns, which benefit a lot from high message rates [2].

Taking into account that the increasing degree of parallelism [3] [4] also leads to an increased number of communication partners, communication pattern characteristics will shift to higher message counts with smaller payloads. Thus, the peak bandwidth is not the only metric that is crucial for the overall performance; instead, an increasing amount of attention must be paid to the performance of smaller messages. To conclude, the performance of small transfers should not only be characterized using the start-up latency, but also using the message rate.

The message rate performance particularly depends on the network interface controller, which needs to be optimized for high efficiency in order to yield high message rates. The theoretical upper bound of the message rate is the link's peak bandwidth divided by message size; a more practical upper bound is the link's sustained bandwidth that takes into account the network protocol overhead. While we have presented details of the communication units for small [5] and large data transfers [6] in previous work, this work extends previous publications by providing the following contributions:

1. An analysis of the impact of different message rates to communication-centric applications and workloads.
2. Details of the optimized and lean interface between EXTOLL's communication units and the corresponding software layers.
3. The first comprehensive disclosure of performance results based on prototype hardware in a standard cluster environment.
4. A methodology to characterize an application's dependency on message rate, also allowing predicting application-level performance for future technologies.

The remainder of this work is structured as follows: in the next section, we provide an analysis of sustained message rates. In the following two sections, we will introduce the

hardware and software architecture in detail. In section 5, performance results for the current prototype are reported. Section 6 is dedicated to the performance analysis for other technologies like ASICs. In section 7, we present related work, while the last section concludes.

II. On Achieving High Message Rates

Theoretically, the effective bandwidth of a network can be translated into a *message rate (MR)* for a given packet size. Practically, this is only true for large packets where the overhead of packet passing is marginal compared to the packet size. For small payloads, more effects become visible and limit the message rate. The following components contribute to the sustained message rate:

1. Network protocol overhead including framing, headers, CRC, etc
2. Message passing protocol overhead including tags, source identification, etc
3. Packet-to-packet gaps caused by network interface
4. Packet-to-packet gaps caused by switching units
5. Software overhead for sending and receiving

The following Table I provides some numbers for two popular interconnection networks (10 Gigabit Ethernet and Infiniband QDR) and for EXTOLL, together with the sustained message rates achieved in our experiments. All message rates are reported in millions of messages per second.

TABLE I. Analysis of Sustained Message Rate

Network	10GE[1]	IB-QDR[2]	EXTOLL[3]
Net Speed	10 Gbps	32 Gbps	5 Gpbs
Theoretical peak message rate (8B payload)	156.3	500.0	78.0
Network protocol overhead	82 B	38 B	32 B
MPI protocol overhead	24 B	10 B	16 B
Packet-to-Packet gap of switching units	NA	NA	8 B
Packet-to-Packet gap of network interface	NA	NA	0 B
Overhead total (as appropriate)	114 B (w/o gaps)	56 B (w/o gaps)	64 B (total)
Sustained Message Rate	0.66 (0.42%)	6.67 (1.33%)	9.73 (12.4%)
Calculated overhead derived from sustained MR	416.67 B	599.70 B	64.14 B

For theoretical peak message rate the assumed minimum payload is 8 bytes. Sustained message rate is the peak measure rate achieved with multiple communication pairs, also reported as percentage of theoretical peak.

[1] Intel 82598EB 10GE controller, no switch. 2x AMD Opteron 2380 (4 cores, 2.5GHz) per node, Open-MPI 1.4.2

[2] Mellanox MT26428 with MTS3600 switch, 1x Intel Xeon E5645 (6 cores, 2.4GHz) per node, MVAPICH 1.2.0

[3] EXTOLL R1, 2x AMD Opteron 2380 (4 cores, 2.5GHz) per node, Open-MPI 1.4.2

We report detailed numbers in Table I for our EXTOLL network prototype, but best to our knowledge such numbers are not published from typical network vendors. Note that the table does not include numbers for software overhead, as due to overlap between software and hardware processing these numbers cannot be determined with sufficient accuracy. While the overheads for network and MPI protocol are similar for all three network types, the ratios of sustained vs. theoretical message rates differ significantly. The last row presents the corresponding calculated overhead in bytes derived from the sustained message rate. This overhead attributes to several potential sources: protocols, network interface, switching units and software overhead.

EXTOLL's calculated overhead matches closely the total overhead from the network interface and switching units (64.14B vs. 64B), validating our calculation, but also showing that our software overhead is minimal. On the other hand, the two other networks show a huge difference between the calculated overhead and analyzed overhead. Although we had to omit numbers for gaps caused by network interfaces and switching units, the difference is too big to attribute it only to these hardware units. We assume that these networks suffer from huge software overheads. Thus, our network design is very competitive compared to existing solutions and therefore allows analyzing the impact of message rate at application level.

III. An Architecture for High Message Rates

In brief, EXTOLL's main characteristics are support for multi-core environments by hardware-level virtualization, communication engines for very low overhead and a minimized memory footprint. Figure 1 shows a top-level block diagram of the EXTOLL architecture, integrating the host interface, network interface and switching units. The host interface is based on *HyperTransport (HT)*, but could be replaced by PCIe logic (which in fact in another design is being done). The on-chip network HTAX closely matches the HT protocol, but overcomes some shortcomings with regard to the limited amount of source tags and the addressing scheme.

The second large block implements the different modules needed for message handling, i.e. to inject messages into the network and receive messages from the network. The two major communication units are the Virtualized Engine for Low Overhead (VELO), supporting programmed I/O (PIO) for small transfers, and the Remote Memory Access (RMA) unit that, uses DMA to handle large messages. The two supporting units are the Address Translation Unit (ATU) and the control & status register file.

The last block implements a complete network switch. It includes a crossbar-based switch, six ports towards the network side and three ports towards the message handling modules on the host side, allowing handling requests, responses and completions independently.

A. Switching Resources Integrated into Network Interface

The EXTOLL prototype can run any direct topology with a maximum node degree of six. The routing hardware is not limited to a certain strategy, like dimension order routing,

nor to a specific topology. For smaller networks, for instance different variants from fully interconnected to hypercube and tori are available. Larger configuration will most probably use a 3D torus configuration, though, based on the available number of links.

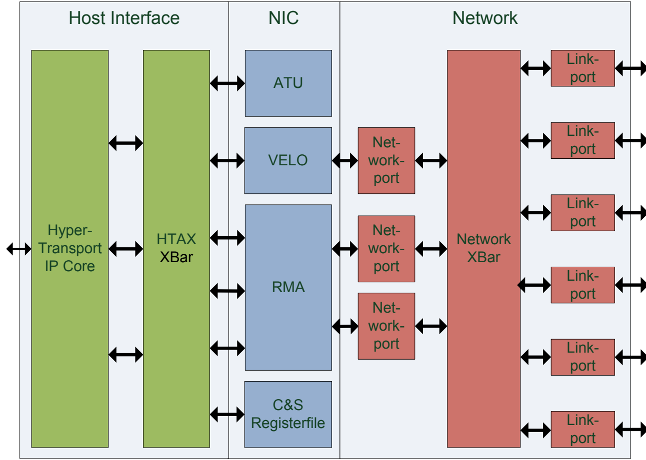

Figure 1. Top-level architecture

The integrated switch implements a variant of *Virtual Output Queuing (VOQ)* on the switch level to reduce *Head-of-line (HOL)* blocking and employs cut-through switching. Multiple virtual channels are used for deadlock avoidance. In particular in-order delivery of packets and reliable packet transmission significantly simplifies the protocol design in MPI layers, allowing for very low software overhead as can be seen in Table I.

B. Support for Small Data Transfers

As the network is designed particularly for ultra-low latencies and high message rates, EXTOLL includes a special hardware unit named VELO that provides optimized transmission for small messages. It offers a highly efficient hardware and software interface to minimize the overhead for sending and receiving such messages. Using VELO, messages are injected into the network using PIO to reduce the injection latency as much as possible. Details of this scheme, as well as for the general architecture of VELO can also be found in [5]. Here, we extend this work with details of the software part and its impact on application-level performance.

```
velo_ret_t velo_send(velo_connection_t* handle,
          uint32_t len, uint64_t* buf, uint8_t tag)
{
  uint64_t* dest= calc_address (handle, tag, len);
  for ( i=0; i < ( len>>3 ); i ++ )
    *dest++ = *buf++;
  return VELO_RET_SUCCESS;
}
```

For VELO, a single write operation into the *memory-mapped I/O (MMIO)* space is sufficient to send a message. For additional optimization, some of the message header information is encoded in the address to access VELO. This implementation saves space in the data section of the I/O transaction. The code snippet above helps to illustrate the software part of issuing a message to VELO.

As one can see, there are only two simple steps required to inject the message: First, calculate the target address of the store instruction, and second, copy the message to the device. A simple loop can be used here. The write-combining feature of modern CPUs is employed to aggregate the data of one message into a single host-to-interface transaction. An exception mechanism exists in hardware to address the case in which a transaction is split by the CPU, for example caused by an interrupt that occurs in the middle of a transaction. Access to VELO is done directly from user-space. Each process has distinct mappings, so hardware can immediately determine if the process is actually authorized to send messages.

The VELO hardware unit is a completely pipelined structure controlled by a number of finite state machines. So no (relatively) slow microcode or even embedded processing resource is involved in sending or receiving data. Another fact that has important consequences on performance is the amount and the location of context or state information for the hardware. VELO is stateless in the sense that each transaction is performed as a whole and no state must be saved for one message to proceed. As a corollary, there is no context information stored in main memory and no caching of such information is necessary. Thus, VELO is able to provide high performance independent of the actual access pattern by different processes, a very important fact in today's multi- or many-core systems.

On the receive side, messages are written directly to main memory using a single ring-buffer per receiving process. Each process allocates its own receive buffer, and any source can store messages to this ring-buffer. User-level processes waiting for messages can poll certain memory locations within this ring buffer for new arrivals. This can be done in a coherent way, so polling is done on cache copies. Updates in the ring buffers invalidate the now outdated copies, enforcing the subsequent access to fetch the most recent value from main memory.

The simplest flavor of a receive function is shown in the code snippet below. After determining the address to poll, the function waits for a new message to arrive. It subsequently copies the data to the application buffer, increments the internal read pointer and resets the header word to zero to prepare for the next time this slot will be used. The first quad word of a message in the VELO receive buffer is guaranteed to non-zero, identifying a valid packet.

```
velo_ret_t velo_recv_wait(velo_port_t* handle,
                       uint32_t len, uint64_t* buf)
{
  volatile uint64_t* s = handle->header;
  while (*(s) == 0l ) { /* busy wait*/ };
  memcpy ( buf, ( void* ) handle->msg, len );
  _velo_recv_inc_rp ( handle );
  handle->header = 0l;
  return VELO_RET_SUCCESS;
}
```

For the VELO transport, the order is maintained by utilizing the hardware's underlying flow-control principles, i.e. EXTOLL's flow-control in the network and

HyperTransport's flow-control for the host side. In extreme cases, this can lead to stalling CPU cores due to missing credits, which are needed to inject messages. To solve this problem a programmable watchdog is provided which prevents system crashes in such a case. On the software side, the problem can be avoided by using a higher-level flow-control. The actual hardware implementation has been improved from the implementation described in [5] in such a way. Also, we increased both frequency and data path width.

C. *Support for Bulk Data Transfers*

Larger transfers are efficiently supported using the RMA unit. The RMA unit offers Put/Get based primitives. There is a hardware-based *Address Translation Unit (ATU)*, which secures memory accesses from user-space. Registration and deregistration of pages is very fast and only limited by the time for a system call and the lookup of the virtual to physical translation by the kernel itself [6].

RMA allows for complete CPU-offloading, which is crucial for high overlap between computation and communication. Processors only have to post RMA command descriptors to the hardware using PIO. The complete data transmission, however, is executed by integrated DMA engines.

An interesting feature that has been used for the MPI implementation (see below) is the notification framework; allowing notifying processes at the sending, responding or completing side of a Put resp. Get operations. Obviously notifications for responses are only applicable to Get operations. Each process has exactly one notification queue, in which the hardware stores all incoming notifications for this process regardless of their type. These queues share many similarities with VELO receive queues.

IV. Software Stack

The software stack of the EXTOLL prototype is divided into the user-space part and the Linux kernel part. The actual process of sending and receiving messages is completely done in user-space. There are two low-level API libraries, one for VELO and one for RMA. Functions for a user-application to request hardware access, set-up memory-mappings and allocate receive queues are provided. For VELO, there are different flavors of *send()* and *receive()* functions available. A VELO context provides a single receive ring-buffer to the application, to which all messages destined to this application are written to by the hardware. For efficiency reasons, and to avoid stalling of the sending CPU-cores, a credit-based flow-control scheme has been implemented [7]. Measurements show that the performance is not significantly impacted by this additional software protocol.

On the RMA side, there are functions to register and de-register memory regions. One set of functions allows the user application to post arbitrary commands like put or get requests to the hardware. Another set of functions manages the notification queue, into which the hardware writes notifications to inform software about completed operations.

A. *Supporting MPI*

There exist several popular open source choices for the MPI implementations. We chose OpenMPI, mainly because the component-oriented architecture is well understood and formed a clean interface to the EXTOLL lower-level software. Within OpenMPI, there are again several interfaces available that can be used to implement support for a transport method. We chose the *Message Transfer Layer (MTL)* interface for EXTOLL. This interface is actually quite simple and encompasses only a small number of functions, for instance initialization and finalization functions, blocking and non-blocking send functions, and a callback-oriented progress function. A number of other functions for example to cancel or test requests are also available. Compared to other interfaces, in particular the *Bit Transfer Layer (BTL)*, the downside of the MTL component is that MPI matching semantics have to be implemented. On the other hand, this allows optimizing matching semantics for our network design.

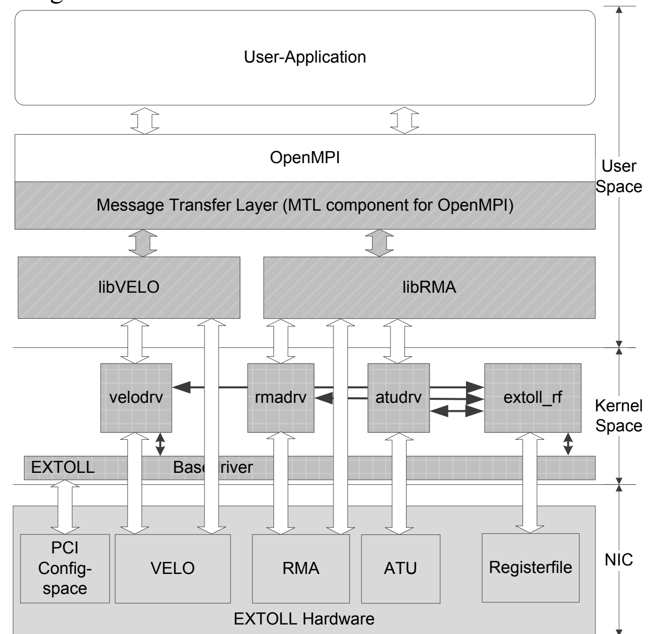

Figure 2. Software stack

Two protocols are implemented to transport messages using the EXTOLL hardware, with a tunable run-time parameter. There is an eager protocol for small messages and a rendezvous protocol for large messages. For the experiments in this work, the threshold was set to 2 kB.

B. *Eager protocol*

The eager protocol relies on the VELO engine. First, a MPI header is built mainly consisting of source process, tag, communicator and size information. This MPI header along with the actual data is then sent to the destination process using VELO send functions. On the receiver side, the VELO receive queue is checked by the progress function. Every incoming entry is software-matched against the receive queue using the MPI header which can be found at the beginning of the message. If no match is found, the message

is added to the unexpected queue. If a matching receive is posted later, this receive can immediately be completed via the unexpected queue. A hardware VELO message has a maximum size of 64 bytes in the EXTOLL prototype. Larger messages are assembled from multiple VELO messages. VELO messages can carry a small tag that is used to distinguish different message types on this protocol layer, for example if this is the first fragment of a larger MPI message. The software also leverages the fact that VELO messages arrive in the same order as they were originally sent.

C. Rendezvous protocol

The rendezvous protocol is built upon both VELO and RMA engines. First, a request message is built and sent to the receiver using VELO. Once this request is matched at the receiver, a sequence of RMA Get operations is used to fetch the data from the sender into the receiver's application buffer. The memory at the sender side is registered right before the request is sent; the receiver registers its buffer when starting the Get operation. For the rendezvous protocol, notifications are used to signal completion both on sender and receiver sides. After these notifications have been received, the respective buffers are de-registered.

V. Prototype Performance Results

The architecture described in the last two sections is currently implemented as a prototype using reconfigurable hardware, i.e. Field Programmable Gate Arrays (FPGAs). We developed a custom add-in card [8] combining the FPGA with an HTX connector as host interface and six serial links towards the network side. We use standard optical transceivers and fibers to connect these add-in cards. The implemented architecture is running at a frequency of 156 MHz with a data path width of 32 bits for the core logic, i.e. the NIC and network block, and 200 MHz for the HT Core. An 8 node cluster is equipped with these custom add-in cards and a 3D torus is set up. Each node includes two 4-core AMD Opteron 2380 processors running at 2.5 GHz, and 8 GB of DDR2-800 main memory. Linux version 2.6.31.6 is installed on these machines.

Our prototype is suffering from some performance limitations due to the used technology. Compared to ASICs, FPGAs are limited in terms of capacity and frequency, as well as flexibility due to integrated hard IP blocks. In particular, the link bandwidth is limited to only 6.24 Gbps and the hop latency is about 300ns, mainly due to the hard IP modules used for serialization.

A. Message rate

First, we report message rate characteristics based on the popular *OSU Message Rate Test* [9]. It reports the sustained message rate, i.e. how many send operations can be issued per second. For this test, multiple communication pairs are set up between two nodes. No message aggregation or other techniques are used to optimize this experiment.

Figure 3 shows the results of this test. We achieve a peak non-coalesced message rate of more than 9.5 million messages per second; yielding a peak bandwidth of 480 MB/s. For comparison, we have also included the results from our IB-QDR experiment in this figure, which is the same as in Table I. The message rate achieved with 10GE is not competitive and only included for reference.

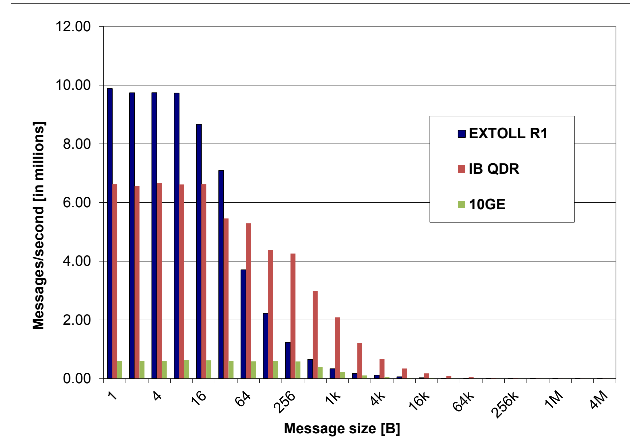

Figure 3. OSU Message Rate Test – Performance over Message Size

The maximum message rate for a varying numbers of communication pairs is shown in Figure 4. EXTOLL requires four simultaneous pairs to saturate, likely due to MPI layer overhead. However, while IB-QDR also requires four pairs to reach its peak, it does not maintain this performance level but instead performance starts dropping significantly.

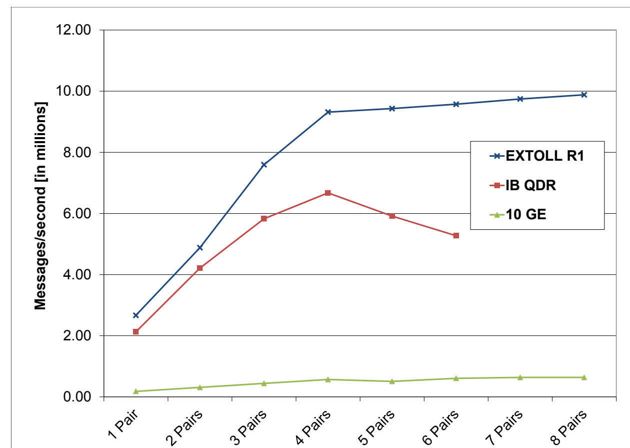

Figure 4. OSU Message Rate - Performance over Pair Count

VI. Performance Analysis and Extrapolation

As our prototype is based on FPGAs instead of much faster ASICs, we present here a methodology allowing estimating application-level performance for faster technologies. In addition, we show that two complex application-level benchmarks (HPCC RandomAccess and Weather Research and Forecasting Model) heavily depend on the message rate. To estimate the relative performance of future versions we rely on simulation- and calculation-based analyses, which are summarized in Table II. As for future versions not only the frequency is varied but also the data

path width increases, we assume that an increase to 64 bit at 300 MHz translates to a relative performance of 600 MHz at 32 bit (or 384%), respectively a 800 MHz at 128 bit translates to 2400 MHz at 32 bit (or 1538%).

TABLE II. PERFORMANCE INCREASE FOR DIFFERENT TECHNOLOGIES

Technology	Core speed	Data path width	Relative frequency	Relative message rate
Virtex-4 FPGA	156 MHz	32 bit	100%	100%
Virtex-6 FPGA	300 MHz	64 bit	384%	400%
65nm ASIC	800 MHz	128 bit	1538%	1000%

A. *Methodology*

In order to predict the performance of future EXTOLL implementations we apply the following methodology. First, we reduce the performance of the FPGA prototype in two manners:

1. The message rate is reduced by including delays after the send function in the API. As this is done after the send function there should be little influence on latency.
2. The core speed of the FPGA is reduced, which influences the message rate, latency and bandwidth.

Then, we rerun the micro-benchmarks to determine the correlation to the message rate, latency and bandwidth. In addition, we use complex benchmarks to determine the impact on application-level performance like GUPS and GFLOP/s. We apply a best fit to the different measurement points and extrapolate the resulting function to determine the impact of faster ASIC and FPGA technologies. As long as the system performance is not limited by third-order effects like the processor or main memory, this extrapolation should provide reasonable results.

1) *Message Rate Variation*

This is achieved by including delays in the API blocking and non-blocking send call after actually triggering the send operation. The delay is dependent on the message size, thus the longer a message is, the longer is the delay is. We use command line parameters to choose between several delay sets, yielding different message rates. Obviously, this also has an impact on bandwidth; however, this satisfies the definition of the message rate. For messages smaller than 128 bytes, the latency is not affected. For larger messages, the delay required to achieve a certain message rate also affects latency, but the latency impact for these messages sizes is not crucial.

In this way, we have selected sets of delay parameter to configure the network for certain message rates, varying between 100% and 70% of peak message rate.

2) *Frequency Variation*

Although some hard IP blocks prevent us from choosing arbitrary core frequencies we have found a set of four frequencies, shown in Table III together with relative performance, averaged over all message sizes.

TABLE III. IMPACT OF FREQUENCY VARIATION

Frequency	Relative frequency	Message rate	Band-width	Latency
156 MHz	100.00%	100.00%	100.00%	100.00%
140 MHz	89.74%	88.62%	92.64%	93.16%
125 MHz	80.13%	79.21%	85.23%	87.18%
117 MHz	75.00%	74.61%	80.89%	83.79%

B. *Experiments*

We employ the *HPCC RandomAccess* benchmark [10] and the *Weather Research and Forecasting (WRF)* model [11] on 8 nodes with 64 processes to characterize the application-level impact of varying message rates.

1) *HPCC RandomAccess*

First, we vary the message rate. We expect a high influence on the HPCC RandomAccess results, as this benchmark relies on a massive exchange of small messages.

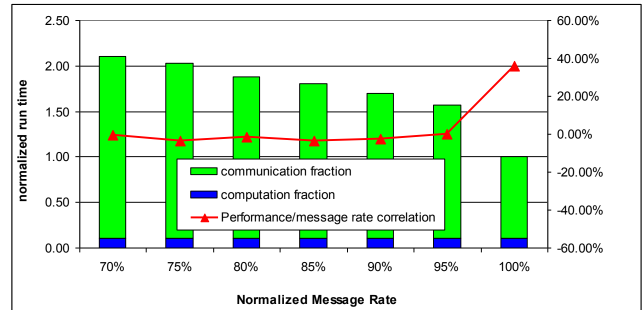

Figure 5. Message rate sensitivity for HPCC RandomAccess

Figure 5 shows the measurement results, reported in normalized run time components for communication and computation, and the performance-message rate correlation. We assume that our instrumentation is only influencing the communication time and not the computation time. Negative values in the correlation indicate that the performance decrease is bigger than the message rate decrease, and vice versa. The configurations for 95%-70% of message rate show the expected high correlation. However, the sharp performance drop from 100% to 95% is unexpected. A possible explanation for this super-linear slow-down might be the effect of the delay after message sending which, as a side effect, also sacrifices compute time. Thus, we remove this setting from our calculations.

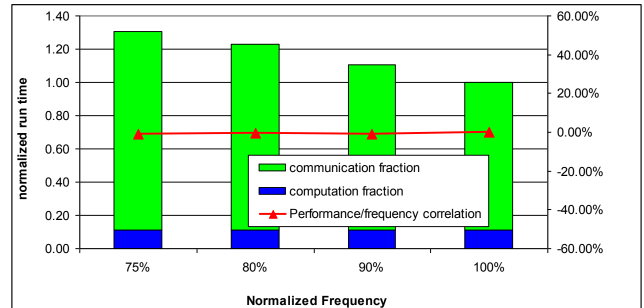

Figure 6. Frequency sensitivity for HPCC RandomAccess

Figure 6 shows the performance results when varying frequency. In this case, all measurement points show a high correlation and can be used for extrapolation.

Based on these results, we use an exponential best fit on the communication time, measured using mpiP [12]. For 64 processes on 8 nodes, this benchmark spends 89% of its execution time within MPI layers. The computing time fraction is kept constant as the amount of work (i.e. processing updates) is not varied. Figure 7 shows the resulting performance in terms of GUPS, saturating at more than 1.8 GUPS. We achieve a high match between the two used variations, validating our approach.

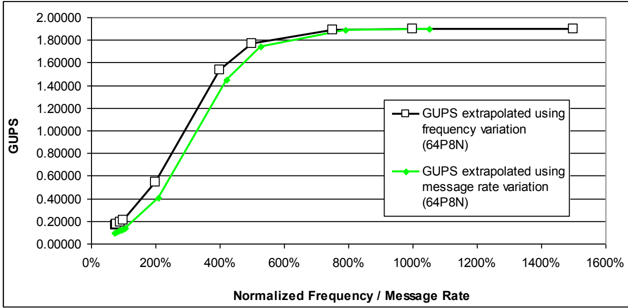

Figure 7. HPCC RandomAccess performance extrapolation

2) WRF– Message rate variation

The same methods are applied to WRF. Figure 8 shows the resulting performance, based on an MPI time of 45.00% for WRF on 8 nodes and 64 processes.

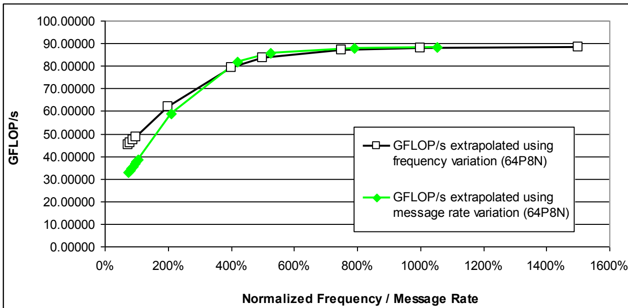

Figure 8. WRF performance extrapolation

The two variations show a very nice fit, in particular for the 400% setting and above. The difference for smaller input values can be explained by the more complex nature of this benchmark and the higher influence of frequency variation with regard to bandwidth.

3) Summary

The outcome of the performance analysis and extrapolation is summarized in Table IV. Although one might claim that for both experiments our ASIC-based version is located in the saturation of the performance graph, we would like to note that this only applies for these two benchmarks and that this is only an 8-node cluster. For larger installations, saturation will likely be reached much later.

TABLE IV. SUMMARY OF PERFORMANCE EXTRAPOLATION

EXTOLL version	Method	FPGA 156 MHz, 32bit	FPGA 300 MHz, 64bit	ASIC 800 MHz, 128bit
HPCC Random Access	Message rate variation	0.20853 GUPS	1.452 GUPS	1.895 GUPS
	Frequency variation		1.538 GUPS	1.896 GUPS
WRF	Message rate variation	48.50262 GFLOP/s	81.833 GFLOP/s	88.161 GFLOP/s
	Frequency variation		79.319 GFLOP/s	88.180 GFLOP/s

VII. RELATED WORK

A large amount of work has been published on interconnection networks and their performance impact on HPC systems. Due to its popularity, most of the research targets the Infiniband architecture, in particular the *ConnectX* technology provided by Mellanox [13]. Sur et al. provide a detailed performance evaluation of the *ConnectX* architecture in [14]. Another established network protocol is Ethernet, and more recently, 10G Ethernet. Hurwitz has studied the performance of 10G Ethernet on commodity systems [15].

Furthermore, several vendors have developed proprietary network technologies. This includes Cray's *Seastar* family, which is extensively studied by Brightwell in [16]. *Seastar* is similar to EXTOLL, however, lacks efficient user-level communication and virtualization support. The successor of Seastar is Cray's *Gemini* [17]. Fujitsu has proposed its Tofu network [18], which supports 6-dimensional Torus topologies with high fault tolerance using multidimensional routing paths. IBM has developed a series of proprietary interconnects, including its *Blue-Gene* family [19] and *PERCS* [20]. The Shaw research institute claims a 162 ns end-to-end communication latency with *Anton* [21], an ASIC that integrates both interconnect and highly specialized processor for executing molecular dynamics simulations. However, it only supports a small set of applications and is not compatible with general-purpose software.

The *HPCC RandomAccess* benchmark has been studied in detail [10] [22] [23], but not with regard to varying message rate. Similar applies to the *Weather Research and Forecasting Model* [11] [24].

VIII. CONCLUSION

We present a network architecture specifically designed for high message rates, including hardware modules and software layers. In particular, we describe the properties and characteristics that enable our design to offer high message rates, and put our performance results in context with other popular interconnection networks.

For in-depth analysis and evaluation, we have designed an entire hardware and software stack, rather than a simulation model. This allows us to test our architecture on an FPGA-based prototype cluster consisting of 8 computing nodes. As opposed to experiments based solely on

simulations, we can now perform comprehensive experiments, covering all aspects of an HPC system.

The FPGA implementation achieves a message rate exceeding 9 million messages per second, which is very competitive considering the performance-limiting FPGA technology. Apart from such micro-benchmark results, we report measurements of applications including the *Weather Research & Forecasting Model* and the *HPCC RandomAccess* benchmark.

For a detailed analysis of the impact of message rate on application-level performance, we modify our design in two ways to provide different message rates to the application. First, we delay send operations to manually create gaps between subsequent messages. Second, we vary the clock frequency, which obviously also impacts message rate.

The outcome of this analysis is two-fold: first, we show that the performance of applications like the *HPCC RandomAccess* and the *Weather Research & Forecasting Model* highly depends on the provided message rate. Second, this analysis allows us to extrapolate performance for other technologies. This includes more recent FPGAs but also ASICs, which allow for both higher clock frequencies and wider data paths.

IX. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive comments. Also, we would like to thank all the people that contributed to the continuing success of the EXTOLL project. In no particular order, we want to especially thank Frank Lemke, Alexander Giese, Sven Kapferer, Benjamin Geib, Niels Burkhardt, Sven Schenk, Benjamin Kalisch, Myles Watson and Richard Leys for all their support and ongoing work, without EXTOLL would never have been possible.

X. REFERENCES

[1] TOP500 list: http://www.top500.org

[2] Underwood, K. D., Levenhagen, M. J., and Brightwell, R. 2007. Evaluating NIC hardware requirements to achieve high message rate PGAS support on multi-core processors. In *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC '07)*. ACM, New York, NY, USA.

[3] Asanovic, K., et al. 2009. A view of the parallel computing landscape. *Communications of the ACM*, 52(10), 56-67.

[4] Kogge P. (Ed.), et al. 2008. *ExaScale Computing Study: Technology Challenges in Achieving Exascale Systems*. US Department of Energy, Office of Science, Advanced Scientific Computing Research, Washington, DC. Available at http://www.er.doe.gov/ascr.

[5] Litz, H., Fröning, H., Nüssle, M., and Brüning, U. 2008. VELO: A novel communication engine for ultra-low latency message transfers. In *Proceedings of 37th International Conference on Parallel Processing (ICPP-2008)*, Sept. 08-12, 2008, Portland, Oregon, USA.

[6] Nüssle, M., Scherer, M., and Brüning, U. 2009. A resource optimized remote-memory-access architecture for low-latency communication. In *Proceedings of 38th International Conference on Parallel Processing (ICPP-2009)*, Sept. 22-25, Vienna, Austria.

[7] Prades, J., Silla, F., Duato, J., Fröning, H., Nüssle, M. 2012. A New End-to-End Flow-Control Mechanism for High Performance Computing Clusters. In *Proceedings of IEEE International Conference on Cluster Computing (CLUSTER2012)*, Sept. 24-28, 2012, Beijing, China

[8] Fröning, H., Nüssle, M., Slogsnat, D., Litz, H., and Brüning, U. 2005. The HTX-Board: A Rapid Prototyping Station. In *Proceedings of 3rd annual FPGAworld Conference*, Nov. 16, 2006, Stockholm, Sweden.

[9] OSU Micro-Benchmarks 3.3: http://mvapich.cse.ohio-state.edu/benchmarks, last accessed Apr. 2011.

[10] Aggarwal, V., Sabharwal, Y., Garg, R., and Heidelberger, P. 2009. HPCC RandomAccess benchmark for next generation supercomputers. *In Proceedings of the 2009 IEEE International Symposium on Parallel & Distributed Processing (IPDPS '09)*. IEEE Computer Society, Washington, DC, USA, 1-11.

[11] Michalakes, et al. Development of a next generation regional weather research and forecast model. In *Proceedings of the Ninth ECMWF Workshop on the Use of High Performance Computing in Meteorology*. Eds. Walter Zwieflhofer and Norbert Kreitz. World Scientific, Singapore. pp. 269-276.

[12] Vetter, J., and Chambreau, C. mpiP: Lightweight, Scalable MPI Profiling: http://mpip.sourceforge.net, last accessed Apr. 2011.

[13] Mellanox Technologies: http://www.mellanox.com.

[14] Sur, S., Koop, M. J., Chai, L., and Panda, D.K. 2007. Performance analysis and evaluation of Mellanox ConnectX InfiniBand architecture with multi-core platforms. In *15th IEEE Symposium on High Performance Interconnects (HOTI)*, Aug. 22-24, 2007, Stanford, CA.

[15] Hurwitz, J., and Feng, W.C. 2004. End-to-end performance of 10-gigabit Ethernet on commodity systems. *IEEE Micro* 24(1):10-22.

[16] Brightwell, R., Pedretti, K., and Underwood, K. D. 2005. Initial performance evaluation of the Cray SeaStar interconnect. In *Proceedings of the 13th Symposium on High Performance Interconnects (HOTI '05)*. IEEE Computer Society, Washington, DC, USA.

[17] Alverson, R., Roweth, D., and Kaplan, L. 2010. The Gemini system interconnect. In *18th IEEE Symposium on High Performance Interconnects (HOTI)*, IEEE, 2010, p. 83–87.

[18] Ajima, Y., Sumimoto, S., and Shimizu, T. 2009. Tofu: A 6D Mesh/Torus interconnect for Exascale computers. *IEEE Computer* 42(11):36-40, Nov. 2009.

[19] Liang, Y., et al. 2006. BlueGene/L failure analysis and prediction models. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN '06)*. IEEE Computer Society, Washington, DC, USA, 425-434.

[20] Arimilli, B. Arimilli, R. Chung, V. Clark, S. Denzel, W. Drerup, B. Hoefler, T. Joyner, J. Lewis, J., and Li, J. 2010. The PERCS high-performance interconnect. In *Proceedings of 18th Symposium on High Performance Interconnects (HOTI)*. IEEE Computer Society, 2010, p. 75–82.

[21] Dror, R. O., et al. 2010. Exploiting 162-nanosecond end-to-end communication latency on Anton. In *Proceedings of the 2010 ACM/IEEE International Conference for High Performance Computing, Networking, Storage and Analysis (SC '10)*. IEEE Computer Society, Washington, DC, USA, 1-12.

[22] Dongarra, J., Luszczek, P. 2005. Introduction to the HPCChallenge Benchmark Suite. *ICL Technical Report*, ICL-UT-05-01. Available from http://icl.cs.utk.edu/hpcc, last accessed Nov. 2012.

[23] Garg, R., and Sabharwal, Y. 2006. Software routing and aggregation of messages to optimize the performance of HPCC Randomaccess benchmark. In *Proceedings of the 2006 ACM/IEEE Conference on Supercomputing (SC)*. ACM, 2006, p. 109.

[24] HPC Advisory Council. 2009. Weather Research and Forecasting (WRF) Model – Performance and Profiling Analysis on Advanced Multi-Core HPC Clusters, March 2009: http://www.hpcadvisorycouncil.com, last accessed Nov. 2012.