Efficient Hardware Support for the Partitioned Global Address Space

Holger Fröning and Heiner Litz
Computer Architecture Group, Institute for Computer Engineering
University of Heidelberg
Mannheim, Germany
{holger.froening, heiner.litz}@ziti.uni-heidelberg.de

Abstract— **We present a novel architecture of a communication engine for non-coherent distributed shared memory systems. The shared memory is composed by a set of nodes exporting their memory. Remote memory access is possible by forwarding local load or store transactions to remote nodes. No software layers are involved in a remote access, neither on origin or target side: a user level process can directly access remote locations without any kind of software involvement. We have implemented the architecture as an FPGA-based prototype in order to demonstrate the functionality of the complete system. This prototype also allows real world measurements in order to show the performance potential of this architecture, in particular for fine grain memory accesses like they are typically used for synchronization tasks.**

Keywords- computer communications, high performance networking, distributed shared memory

I. Introduction

High Performance Computing (HPC) today relies on parallelization in order to increase the performance of a system. Large number of nodes organized as a cluster then work cooperatively on a single problem. For communication and synchronization purposes messages are used, which are exchanged among two or more nodes. However, message passing requires an active process to handle the message both on the sending side and on the receiving side.

Amdahl's law tells us that the maximum achievable speed-up of a program is limited [1]. Only the parallel fraction of time can achieve a speed-up by parallelization. In particular synchronization – but also communication - contributes significantly to the serial fraction, reducing the parallel fraction. Considering for instance a serial fraction of only 5%, according to Amdahl's law parallelization can only yield a speed-up of at most 20x.

Taking this into account, and additionally the continuously increasing degree of parallelization it is obvious that the serial portion of an application must be minimized. One approach is to avoid any unnecessary kind of software involvement on the target side by solely using one sided communication. In such a *Remote Direct Memory Access (RDMA)* the overhead for message and tag matching is avoided, instead the network interface directly handles remote requests for data. The overhead due to registration and the use of descriptors limit this approach to large packet sizes. In order to efficiently support small packet sizes – like they are typically used for synchronization – new approaches are necessary.

Such small packets are frequently used in the shared memory programming paradigm, which recently receives a resurgence of interest in form of the *Partitioned Global Address Space (PGAS)* [2]. It combines the programming convenience of a shared memory system with the scalability of a message passing based cluster. PGAS allows to maintain local coherency domains (typically one node consisting of several CPUs and memory controllers), while the view of the global shared memory is non-coherent. This allows leveraging the high performance caching structures found in modern CPUs, while avoiding the overhead found in global coherency schemes for large systems.

The main contribution of this work is a novel communication engine for shared memory accesses implemented completely in hardware. It relies on the HyperTransport protocol which is extended from the scope of one node to a large set of nodes. Local load/store transactions are forwarded over a custom high performance network to remote nodes.

Due to the scalability problems of coherency protocols there is no global coherency, but the local coherency domain within one node is maintained. This approach makes this architecture highly suitable for the PGAS programming model and also allows avoiding software communication overhead.

The remainder of this work is structured as follows: in the following section the overall architecture of the system is shown. In section 3 the communication engine is described in detail. Section 4 provides performance results from both real-world measurements and simulations. An overview on related work is given in section 5, while the last section concludes and presents an outlook on future work.

II. System Architecture

Goal of this work is to set up a Partitioned Global Address Space with local coherency domains but no global coherency. The system is based on commodity CPUs, memory and main boards for high cost effectiveness. PGAS applications can benefit from this architecture by directly accessing remote memory locations without any overhead due to software involvement.

While for bulk transfers likely an RDMA operation will be used, with this work here we concentrate on fine grain accesses. In particular for such small data transfers RDMA units are typically not very well suited. The associated overhead is significant and often even exceeds the time for data transfer.

A. *Address Space Layout*

The address space of one node is split up into local addresses and global addresses. From the architecture's point of view the decision which addresses are local and which global is arbitrary, in the example shown in Figure 1 all local addresses above 2^{40} are global. Modern CPUs typically have 48bits of physical address space, thus there is sufficient space left in order to set up a global address space.

While there is no memory directly behind a global address, local addresses point to local memory. This local memory is separated into a private partition and a shared partition. The set of shared partitions form the globally addressable memory.

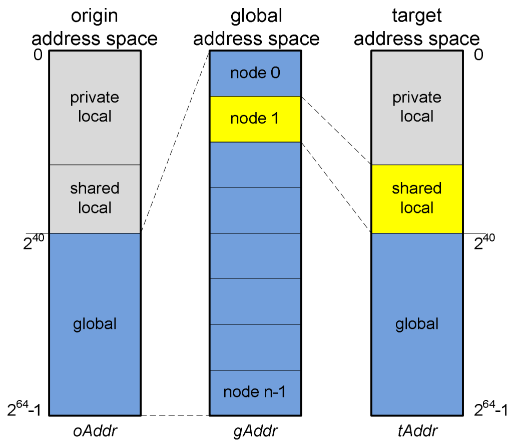

Figure 1. Address Space Layout

In order to minimize the communication overhead, we avoid logical to physical address translations by using physically contiguous and pinned memory region for the shared local partitions on the target side. Then the hardware unit can directly access the memory using physical addresses.

B. *Remote Memory Access*

To access remote memory the CPU performs a load or store instruction which results in a PIO read or write operation on a global address. A part of the global address determines the target node identifier, where the network packet is routed to. The operation is encapsulated in a network packet and transferred over the network to the target node. The other part of the global address points to a location within the shared local partition of this node. In the case of a store the operation is completed by writing the payload to the shared local memory address. In the case of a load the corresponding memory location is read and an appropriate response is sent back to the source node. This response is then forwarded to the CPU which is awaiting the response.

C. *Prototyping*

Latency is of paramount importance when accessing memory, in particular for remote memory locations. Due to this, *HyperTransport (HT) [3]* is used as the interface between the communication engine and the host system. HT allows a direct connection between communication engine and host CPUs and memory controllers, avoiding any kind of protocol conversion or intermediate bridges. In addition to the host interface we apply a custom high performance interconnection network which is optimized for low latencies in order to minimize the overall latency [4].

In order to allow fast prototyping of the complete system an existing HT infrastructure based on *Field Programmable Gate Arrays (FPGAs)* is used: it consists of an FPGA-based add-in card [5] with HTX connector, an HT core as interface to the host and the custom high performance interconnection network.

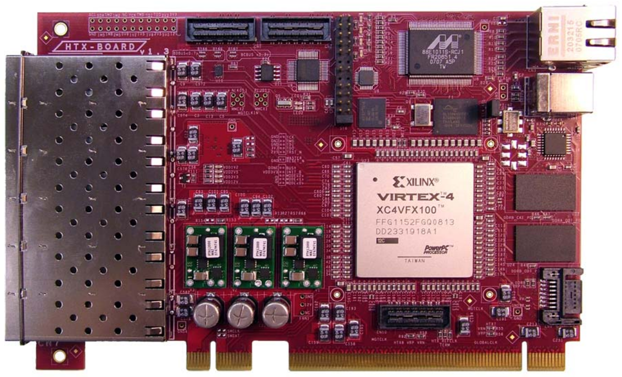

Figure 2. Add-in card implementing host interface, network interface and network switch

The add-in card is shown in Figure 2. It is based on a Xilinx FPGA, and basically provides six links to the network and an HTX connector towards the host. The six links allow building up direct networks, avoiding any kind of central switching units. Both the network interface and the network switches are implemented in the FPGA of the add-in card.

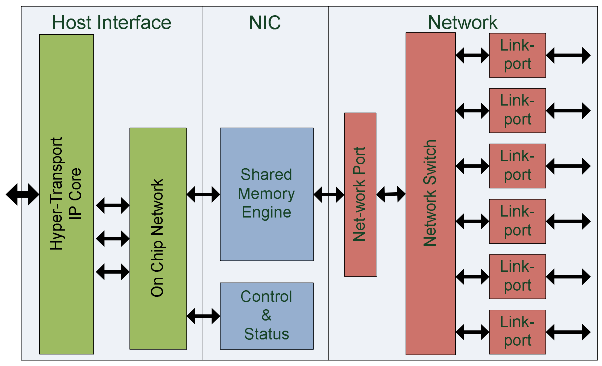

Figure 3. Top-Level diagram of add-in card

In Figure 3 the block diagram of the logic implemented in the FPGA of the add-in card is shown. On the left side (in green) the HT core connects to the host system and to the *On Chip Network (OCN)*. The on chip network connects to the shared memory communication engine and auxiliary units for control and status purposes (shown in blue). On the right side (in red) is the network switch with the six connections to the network.

III. Communication Engine

The major task of the shared memory communication engine is to forward HT transactions like *non-posted reads* and *posted writes* from origin node to target node. Non-posted reads are carried out as split phase transaction, whereas the read request is answered by an independent packet based response. The response carries a tag which enables the originator to match the response to the corresponding request. Posted writes are carried out by simply writing data to a specific address. Several sub-tasks can be identified based on this, which are target node determination, address translation and source tag management.

A. Address translation and target node determination

Three different address spaces exist in this architecture: one global address space and two local address spaces, one for the origin and one for target side. As a remote memory access is traversing all three address spaces, address translation must take place. To avoid unnecessary latencies for this task the translation scheme should be kept simple but efficient.

Considering Figure 1, the global address (*gAddr*) can be determined by subtracting the origin start address of the global partition (*oStartAddr*) from the origin address (*oAddr*):

$$gAddr = oAddr - oStartAddr \qquad (1)$$

This global address is also used to determine the target node identifier. A bit mask is applied to the address, and a centrifuge operation compacts the masked bits to a value which represents the target node identification (*tNodeID*):

$$tNodeID = \left(gAddr \,\&\, mask\right) >> shift\_count^{1} \qquad (2)$$

The operation is very similar to the one described by S. Scott in [12] and adds only minimal latency. Other approaches like table look-ups are also possible, but in favor of less latency a calculation is preferred here.

The inverted mask applied to the global address together with the start address of the target local shared partition (*tStartAddr*) is then used to calculate the target address (*tAddr*):

$$tAddr = \left(gAddr \,\&\, \sim mask\right) + tStartAddr \qquad (3)$$

This address translation scheme allows determining the appropriate address to access data on the target node by converting the origin local address to a global address and then to a target address. This calculation is completely implemented on the origin side, i.e. the network packet already contains the target address. The translation only takes place for requests. Responses do not contain an address; they are assigned to their corresponding requests by using source tags.

B. Source tag management

The need for a source tag management is introduced by the existence of multiple source tag domains. The existing uniqueness of source tags is limited to the local domain, if source tags from multiple domains are mixed the identification is lost. Considering the case that multiple origins are sending requests to one single target, the source tags cannot by uniquely identified.

Because unique source tags are essential for response matching a source tag translation takes place. Each transaction gets assigned a new source tag, which is unique within the domain of the target node. Response matching is handled within a target node; hence there is no need to make source tags unique over multiple target nodes.

The handling of incoming non-posted requests consists of the following steps:

1. Determine free source tag
2. Store origin source tag and origin node identifier in a table indexed by new source tag
3. Send out local non-posted request to access the memory location
4. Upon arrival of response, look-up in table using target source tag to determine origin source tag and origin node
5. Send network packet to origin node, containing non-posted response and origin source tag

For posted requests no source tag translation is necessary because no response is required in this case. Unlike to the address translation, the source tag management is implemented completely on the target side. Thus, a network packet always contains the origin source tag.

C. Egress and Ingress module

From a functional perspective, the communication engine can be divided into three parts: Requester, Responder and Completer. All three units are used for non-posted transactions, while for posted ones only Requester and Completer are required.

To simplify implementation – in particular the number of ports for the OCN, the communication engine is divided into two parts, which are the *Egress* and *Ingress* modules. The first acts as Requester and Responder (sending out packets), while the latter acts as Responder and Completer (handling incoming packets).

[1] This is a simplified calculation for a contiguous mask. Furthermore, the value of *shift_count* is dependant on the actual value of the mask.

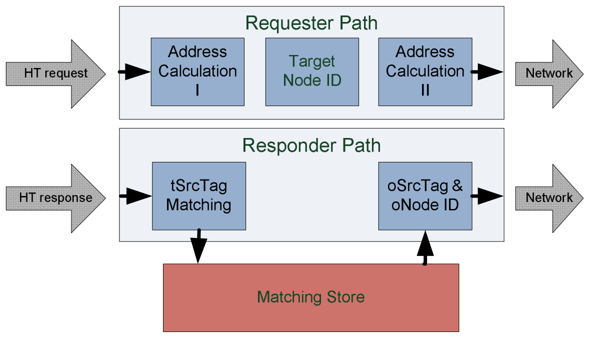

Figure 4. Egress Module

The *Egress* module is shown in Figure 4. When acting as a Requester, HT transactions to be forwarded are coming in from the left. The address is translated and the target identifier extracted from the global address. Then a network packet containing the transaction is sent out. In the second use case – acting as a Responder for non-posted transactions – the local HT response is matched against the source tag table (Matching Store). Using the origin source tag and origin node id from this table a network packet is generated and sent out to the origin node.

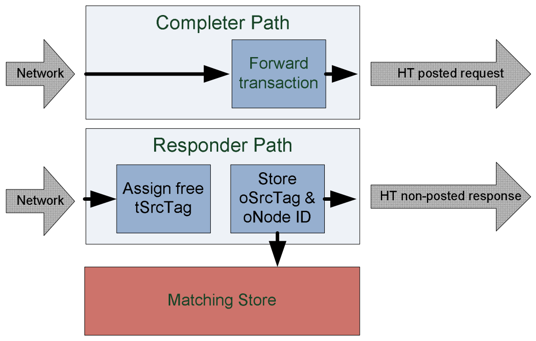

Figure 5. Ingress Module

The *Ingress* module shown in Figure 5 acts either as a Completer or as a Responder. In both cases it receives incoming transactions from the network. In the case of a non-posted transaction it acts as a Responder, i.e. it assigns a free target source tag to the transaction, and stores origin source tag and origin node identifier in the appropriate entry of the matching store. Because the OCN supports 1024 source tags the matching store also has 1024 entries. The transaction with the new (locally unique) source tag is then forwarded over the OCN to the host system. In the case of a posted transaction the Ingress module acts as a Completer and directly forwards the transaction without any source tag translation.

IV. Performance Results

The architecture itself is very lean, efficient and due to it's pipelined implementation performant. However, several problems exist which limit the overall performance of the system. They do not origin to the communication engine, but to the behavior of the used CPUs for the prototype (AMD Opteron K10[2] [11]).

A. *Limitations*

First, the communication engine as a device is only accessible using *Memory Mapped IO (MMIO)* space; hence the host CPU treats it like a peripheral device. Peripheral address space is typically limited in size – most systems do not support more than 256MB. This limits the amount of remotely accessible memory. This situation can be overcome by using a custom BIOS which supports larger peripheral address space.

Additionally, modern CPUs restrict PIO accesses to MMIO in at least two aspects: the number of outstanding load transactions (non-posted) is typically limited to one and the payload of such a transaction is at most 64 bit. However, these limitations do not apply for store transactions (posted).

These problems can be overcome by moving the communication engine from MMIO space to DRAM space. Then it is treated by the host CPU as a memory controller and the CPU does not restrict the accesses. However, moving it into DRAM space is a very complex step and requires the communication engine to participate in the cache coherency protocol.

B. *Performance numbers from measurements and simulations*

In this paper basic measurement results using MMIO space are presented, enriched with results from simulation in order to estimate the achievable performance when more outstanding non-posted transactions with an increased payload size are available. Additionally, the accuracy of the simulation model is validated by measurements, and performance predictions for an ASIC technology instead of an FPGA are provided.

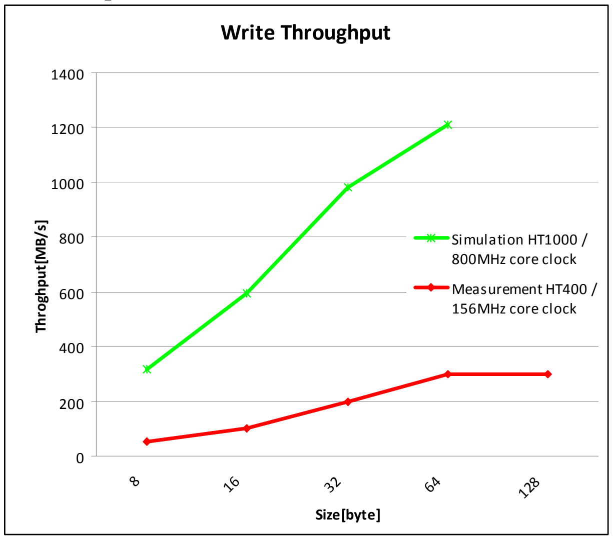

Figure 6. Write Throughput

[2] Similar restrictions do apply for comparable x86-64 architectures from other vendors.

In Figure 6 the write throughput is shown, including the results gained from real-world measurements as well as simulations of an ASIC technology. The FPGA used in the measurement cannot achieve more than 156MHz core clock and an HT link speed of 400 MHz *Double Data Rate (DDR)*. It can be seen that the performance-limited FPGA prototype achieves an excellent throughput, starting at about 50MB/s for 8 byte sized transactions and yielding up to 300MB/s. This is also the available throughput for a payload of one cache line. The simulation of an ASIC technology shows the potential of this architecture.

In order to put this performance data into context and allow a kind of comparison with message passing, in Figure 7 the transaction rate achieved with posted transactions is shown. For sizes below or equal 32 bytes we measured a rate exceeding 6 million transactions per second. One cache line can still be sent out nearly 5 million times per second.

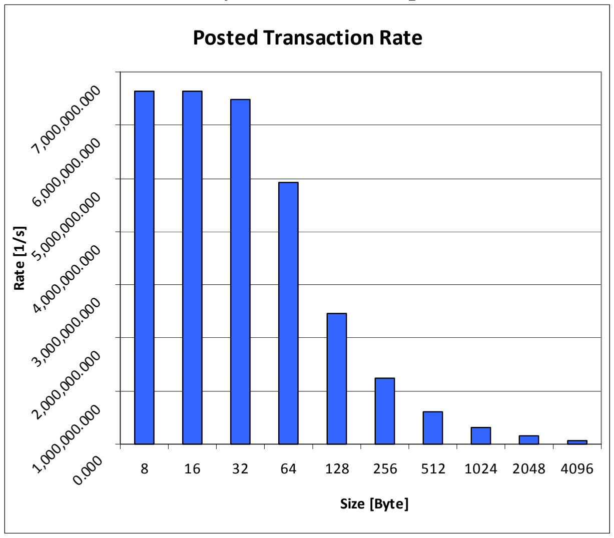

Figure 7. Posted Transaction Rate

The latency of a remote memory access is shown in Figure 8. Again, this figure reports results from the prototype measurement as well as from a simulation using an ASIC technology. Using the prototype, a data word of 8 bytes can be fetched in less than two microseconds. Using an ASIC technology this latency can be reduced to 1.27 microseconds. Note that this time is the full-round trip latency, in opposite to message passing latencies which are usually reported as half-round trip latencies.

The measurement results from Figure 8 are limited in two aspects: only one transaction can be outstanding and the maximum payload is 8 byte. As already pointed out in section three, this is a limitation of the CPU microarchitecture. This behavior could be either changed by modifying the microprocessor architecture or by implementing the shared memory unit as a coherent device which increases complexity but also performance significantly. Due to this, we here only perform simulations in order to show the potential.

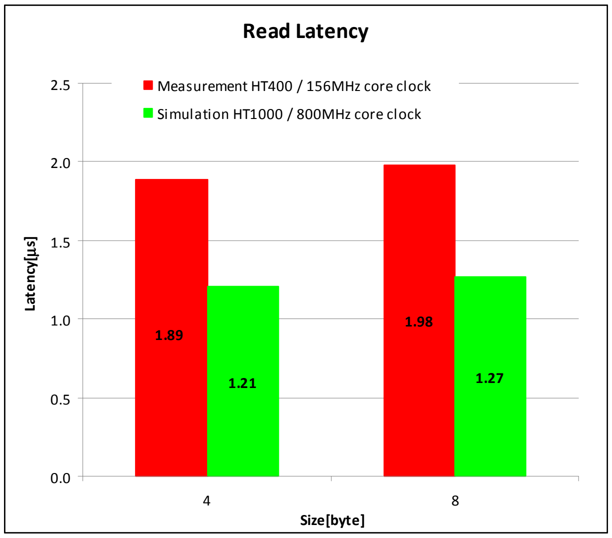

Figure 8. Read Latency

The simulations are based on the FPGA technology and use larger payload sizes and more outstanding transactions. The resulting throughput for non-posted transactions is shown together with results from measurements in Figure 9. It can be seen that the measurement and simulation results match closely at the 8 byte transaction size point. Additionally, the simulations show that the latency for a 64 byte non-posted transaction only increases by about 12% compared to one 8 byte transaction.

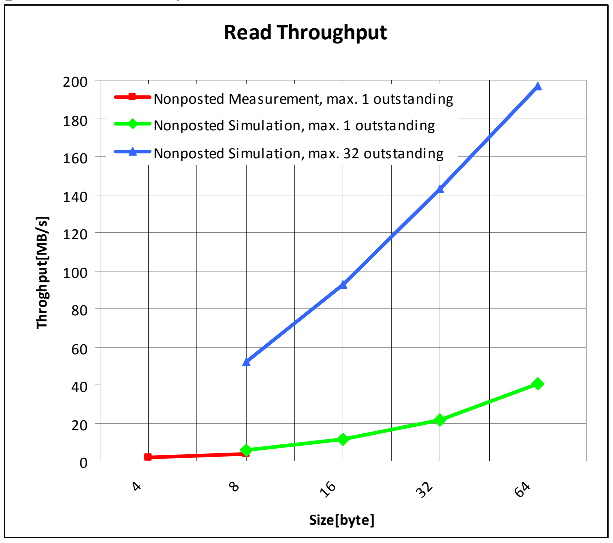

Figure 9. Read Throughput

V. RELATED WORK

Best to our knowledge the only other existing project in the area of non-coherent distributed shared memory architectures is described by Yalamanchili et al. in [7]. In [6] a specification is provided to extend HyperTransport to a large number or nodes, but this is only related to the technical part of HyperTransport.

Typically, shared memory architectures do not focus on the PGAS programming paradigm, hence they have to rely on the memory hierarchy in order to achieve peak performance. Considering coherent distributed shared memory systems there are a couple of systems. Examples for commercial projects include the SGI Origin [8] and SGI Altix [9]. An example for a research project is the Stanford DASH/FLASH [10], but this project was finished a long time ago.

Unlike these solutions we target a non-coherent distributed shared memory system, which explicitly supports PGAS languages. These languages receive an increasing amount of interest and meantime several different PGAS languages exist. Two widely-known examples are UPC [13] and Titanium [14].

Generally, PGAS operations are somewhat similar to RDMA operations like Put/Get. Both approaches allow the user to access remote memory without software involvement on the target side. However, RDMA operations still require a registration process for shared memory regions, while remote load/store operations do not require this. In particular for small payloads this overhead is significant. But RDMA operations will likely be used to complete the shared memory functionality. In other words, for fine grained operations – e.g. used for synchronization – the shared memory approach is highly efficient and viable. But for large bulk transfers RDMA has benefits, including higher throughput and CPU off-loading.

VI. Conclusion and Outlook

We present here a novel architecture of a communication engine for non-coherent distributed shared memory. It leverages HyperTransport to forward local transactions to remote nodes. Besides the development of the architecture we also implemented a prototype using an FPGA for real world experiments. Because the CPU has certain limitations on remote load operations when using MMIO space – in particular regarding the number of outstanding transactions and the payload size – we complete the measurement with results from simulations.

The results presented in this paper demonstrate the high efficiency and lean architecture of our approach. With our FPGA-based prototype we achieve remote load latencies of less than 2 microseconds for an 8 byte data transfer. Simulations show an ASIC technology operating at 800MHz and HT 1000 will lower this down to 1.27 microseconds[3]. For remote stores more than 6 million transactions per second are possible. These outstanding results, in particular, show the advantages of such an approach for fine grain communication and synchronization.

As the next step we will further investigate how the number of outstanding transactions and the payload size for remote loads can be increased, especially by exploiting a tighter coupled coherent shared memory unit. Regarding the architecture, we want to add support for remote atomic operations in order to make synchronization more efficient and to add a kind of pre-fetching with caching in order to decrease latencies for linear or strided accesses.

Last but not least, we are planning to complete this communication engine for fine grain accesses with an RMA unit for high-bandwidth bulk data transfers. Together with a port of UPC/GASNet [15] or another suitable PGAS language we can then show the benefits of this approach using benchmarks (e.g. [16]) and applications.

[3] Note that these times are reported as full-round trip latency, in opposite to message passing latencies which are usually reported as half-round trip latencies. For a comparison these times should therefore be divided by two.

References

[1] Amdahl, G. Validity of the Single Processor Approach to Achieving Large-Scale Computing Capabilities. In *AFIPS Conference Proceedings* (30): 483–485, 1967.

[2] Yelick, K. et al. Productivity and Performance using Partitioned Global Address Space Languages. In *Proceedings of the 2007 International Workshop on Parallel Symbolic Computation*, London, Ontario, Canada, July 27 - 28, 2007.

[3] Slogsnat, D., Giese A., Nüssle M., and Brüning U. An Open-Source HyperTransport Core. In *ACM Transactions on Reconfigurable Technology and Systems (TRETS)*, 1(3):1-21, 2008.

[4] Nüssle, N., Geib, B., Fröning, H., and Brüning, U. An FPGA-based custom high performance interconnection network. In *Proceedings of the 2009 International Conference on Reconfigurable Computing and FPGAs*, Cancun, Mexico, 2009.

[5] Fröning, H., Nüssle, N., Slogsnat, D., Litz, H. and Brüning, U. The HTX-Board: A Rapid Prototyping Station. In *Proceedings of the 3rd Annual FPGAworld Conference*, Stockholm, Sweden, 2006.

[6] Duato, J. et al. Extending HyperTransport Protocol for Improved Scalability. *HyperTransport Consortium Whitepaper*, 2009.

[7] Yalamanchili, S., Young, J., Duato, J., and Silla, F. A Dynamic, Partitioned Global Address Space Model for High Performance Clusters. In *CERCS Technical Reports*, Georgia Tech, 2008.

[8] Laudon, L., and Lenoski, D. The SGI Origin: A ccNUMA Highly Scalable Server. In *Proceedings of the 24th Annual International Symposium on Computer Architecture*, 1997.

[9] Saini, S., Jespersen, D. C., Talcott, D., Djomehri, J., and Sandstrom, T. Application-based early performance evaluation of SGI altix 4700 systems for SGI systems. In *Proceedings of the 5th Conference on Computing Frontiers*, Ischia, Italy, 2008.

[10] Kuskin, J et al. The Stanford FLASH multiprocessor. In *25 Years of the International Symposia on Computer Architecture (Selected Papers)*, Barcelona, Spain, 1998.

[11] P. Conway and B. Hughes. The AMD Opteron Northbridge Architecture. In *IEEE Micro 27(2):10-21*, 2007.

[12] Scott, S. L. Synchronization and communication in the T3E multiprocessor. In *Proceedings of the Seventh International Conference on Architectural Support For Programming Languages and Operating Systems*, Cambridge, Massachusetts, United States, 1996.

[13] UPC Consortium. UPC Language Specifications v1.2. In *Lawrence Berkeley National Lab Tech Report LBNL-59208*, 2005.

[14] Yelick, K. et al. Titanium: A High-Performance Java Dialect. In *Concurrency: Practice and Experience, Vol. 10, No. 11-13*, 1998.

[15] Bonachea, D. Proposal for Extending the UPC Memory Copy Library Functions and Supporting Extensions to GASNet, v2.0. In *Lawrence Berkeley National Lab Tech Report LBNL-56495 v2.0*, 2007.

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