— Author's copy —

Communication Performance Characterization for Reconfigurable Accelerator Design on the XD1000

Tobias Schumacher, Tim Süß, Christian Plessl and Marco Platzner
University of Paderborn, Germany
Email: tobe@uni-paderborn.de

***Abstract*—Providing customized memory architectures is key for achieving high-performance with reconfigurable accelerators. Since reconfigurable computers provide limited possibilities for customizing the organization of external memory, a specific challenge is to make use of the existing memory layout in a flexible, yet efficient way.**

In this paper we build on IMORC, our architectural template and on-chip network for creating reconfigurable accelerators, and discuss its infrastructure for accessing memory. We characterize the IMORC communication bandwidth on the XtremeData XD1000 reconfigurable computer. Based on this characterization, we present a z-buffer compositing accelerator which is able to double the frame-rate of a parallel renderer.

I. Introduction

Reconfigurable accelerators achieve performance gains over CPUs by turning application hot spots into customized hardware cores and providing customized memory architectures to deliver the required high data bandwidth. Typical reconfigurable platforms for high-performance computing come with certain fixed memory architecture with no or limited possibility to change the size and organization of external memory on an per-application basis. A specific challenge is to be able to quickly develop accelerators and make use of the existing memory layout in a flexible, but most efficient way.

For supporting reconfigurable accelerator design, we have created the IMORC Infrastructure for Performance Monitoring and Optimization of Reconfigurable Computers [1], [2]. IMORC consists of an architectural template and an on-chip network. An application splits into an arbitrary number of cores that run at full speed in their own clock domains and communicate asynchronously via FIFO-buffered links. IMORC inserts bitwidth conversion modules into the links which speeds up accelerator design and facilitates core re-use. The IMORC infrastructure also includes memory controllers and host interfaces providing the cores with a unified and transparent way of accessing different kinds of memory, e. g., on-chip memory, off-chip memory or host memory.

Related work on architectural templates includes SIMPPL [3], which also connects different cores in an FPGA using asynchronous FIFOs. Examples for performance modeling and profiling approaches can be found in [4] which provides a model for determining an application's suitability for FPGA acceleration, in the reconfigurable computing amenability test (RAT) [5], and in [6] where profiling based on manually inserted load sensors is proposed. IMORC differs from these approaches in (i) its more flexible interconnect and (ii) extended infrastructure support. The IMORC interconnect employs a multibus architecture with slave-slide arbitration which allows for a multitude of topologies. Infrastructure cores such as bitwidth converters, farming cores, and load sensors [2] greatly ease accelerator design and performance analysis.

The contribution of this paper is the communication performance characterization for IMORC cores on the XtremeData XD1000 system and, based on that, the development of an accelerator for parallel rendering. In Section II, we discuss the memory layout on the XD1000 reconfigurable computer and the IMORC infrastructure for accessing memory on that machine. We then experimentally characterize the maximally achievable communication performance. In Section III, we use these data to develop an IMORC accelerator for a z-buffer compositing kernel, a hot spot in our parallel rendering application. The compositing kernel is a data-centric streaming kernel with almost no computation and thus ideal for evaluating the efficiency of the memory architecture. Measurements presented in Section IV show that the IMORC accelerator can double the frame rate achieved with the parallel rendering application. Section V concludes the paper.

II. Communication Performance Characterization

In this section, we quantitatively characterize the performance achievable for a reconfigurable core accessing memory on the XtremeData XD1000 reconfigurable computer. The XD1000 features a 2.2GHz AMD Opteron CPU with 4GB of host memory, and a module equipped with an Altera Stratix II EP2S180-3 FPGA and 4GB of external memory fitted into a second Opteron socket. The FPGA connects to the Opteron processor via a 16-bit wide HyperTransport link running at 800MT/s. Figure 1 displays the XD1000 architecture with the three types of memory a reconfigurable core can access:

Published in: *Proc. Int. Conf. on ReConFigurable Computing and FPGAs (ReConFig)*
© IEEE 2009, DOI: 10.1109/ReConFig.2009.32

- *Host memory:* For accessing the host memory, we implement an IMORC interface to HyperTransport which is based on the HT cave described in [7]. The cave maps three distinct address regions into the address space of the CPU and provides incoming HyperTransport packets to our interface. The interface decodes the requests and converts packets that hit one of the first two address regions into equivalent IMORC packets which are posted on two separate IMORC links. Packets targeting the third address region directly access an embedded block of memory used as page mapping table. The first two links are used for accessing control registers and for large data transfers, respectively. The page mapping table is responsible for a mapping in the other direction: host memory is mapped into the address space of a third IMORC link. For this purpose, the page mapping table needs to be filled with physical page addresses of the target host memory by the user application. When cores send requests over this link, the upper bits form an index into the page mapping table and the lower bits are the offset into the page. Using this address mapping, the IMORC packet is converted into an equivalent HyperTransport packet and posted to the HyperTransport cave.
- *External memory:* For off-chip memory access, IMORC wraps the Altera DDR SDRAM controller core which can access memory in blocks of configurable burst sizes. DDR SDRAM writes always cover a complete burst, i.e., 2, 4 or 8 clock cycles, depending on the controller configuration. The XD1000 system provides a 128-bit wide memory. Hence, depending on the burst size, 256 bit, 512 bit or 1024 bit are written to memory during a transfer. Since the XD1000 does not provide data mask pins for the memory to mask out bytes not to be written during a burst, IMORC implements a read-modify-write cycle for writes that do not match one of the burst sizes. The read-modify-write cycle incurs overhead but at the same time increases flexibility and potential for core re-use, as the core designer is not necessarily bound to pre-determined burst sizes and link widths.
- *Internal memory:* IMORC also provides a versatile interface for accessing on-chip memory which is functionally equivalent to the interface for accessing off-chip memory. Typical FPGA on-chip memories provide byte-enables which makes the implementation of the interface for internal memory less complex than for external memory. Moreover, on-chip memory can be easily adapted to an application's needs since it is customizable to a high degree regarding parameters such as width and depth. While the achievable bandwidth for accessing internal memory can be huge, the capacity of internal memory is rather limited.

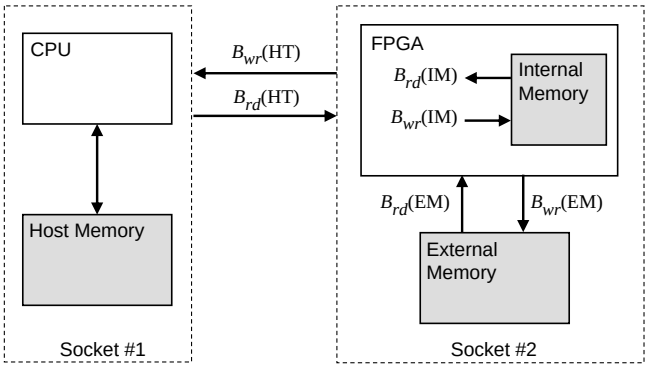

Figure 1. Memory architecture of the XD1000 architecture

Memory	Capacity	Parameter	Bandwidth
Host	4 GB	$B_{rd}(HT)$	1.6 GB/s
		$B_{wr}(HT)$	1.6 GB/s
External	4 GB	$B_{rd}(EM)$	5.4 GB/s
		$B_{wr}(EM)$	5.4 GB/s
Internal	1 MB	$B_{rd}(IM)$	$\gg$
		$B_{wr}(IM)$	$\gg$

Table I
CAPACITIES AND THEORETICAL BANDWIDTHS FOR THE XD1000

Table I summarizes the capacities and theoretical maximum bandwidths for all memories in the XD1000 system, as taken from specifications and data sheets.

The bandwidth figures in Table I can be considered upper bounds. Any concrete implementation such as IMORC will possibly not be able to meet these theoretical figures due to controller and protocol overheads. Typically, such overheads make the bandwidth dependent on the request size. Furthermore, the bandwidth achieved in an implemented accelerator can be reduced by contention, when several cores compete for accessing the host or external memory.

In order to obtain more detailed bandwidth figures, also for varying request sizes, we have implemented a micro benchmark. The micro benchmark essentially is an IMORC core consisting of a request generator, a data source, and a data sink. The core provides one IMORC link per memory to be tested and can be configured in the test type (read/write), the overall size of data transferred and the request size per transfer. The link to the HyperTransport interface is 64-bit wide, the link to the memory controller for external memory 256-bit. The core gathers the number of clock cycles required for completing a request and sends this data to the host application.

Figures 2 and 3 present the measured performance for accessing host and external memory with varying request sizes. External memory on the FPGA module is tested using different controller configurations with 2-, 4- and 8-cycle bursts. As expected, for writes to external memory Figure 2 clearly shows that the read-modify-write cycle produces a significant overhead. Memory-bandwidth bound

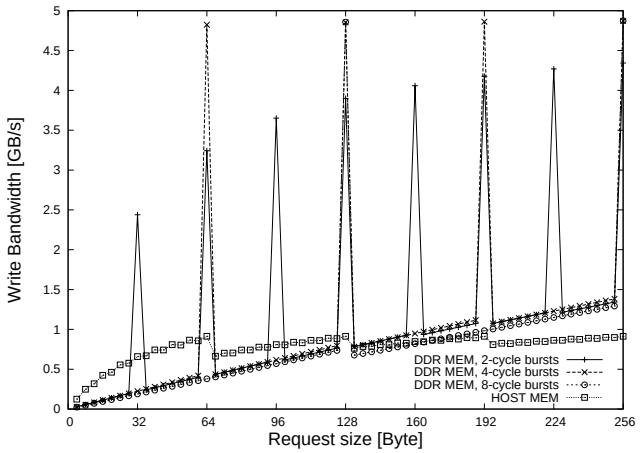

Figure 2. Measured write bandwidth $(B_{wr}(EM), B_{wr}(HT))$ for an IMORC core on the XD1000

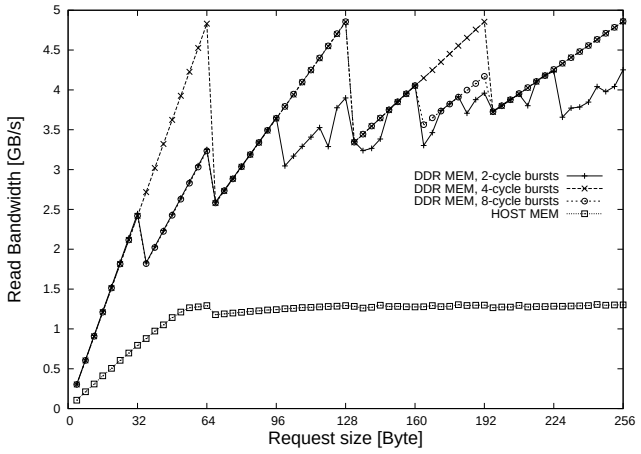

Figure 3. Measured read bandwidth $(B_{rd}(EM), B_{rd}(HT))$ for an IMORC core on the XD1000

applications should thus prefer write block sizes matching the controller's burst size. For request sizes that do not match the controller's burst size, all configurations achieve roughly the same bandwidth.

For full-burst writes the controller configurations differ strongly. Configurations with 4- and 8-cycle bursts achieve a write bandwidth of about 4.8GB/s to 4.9GB/s, while the 2-cycle burst configuration is inferior. The 2-cycle burst configuration delivers a write bandwidth of only some 2.5GB/s for small request sizes that comprise one burst, and improves to about 4.3GB/s for requests of 256 bytes.

For write accesses to host memory over the HyperTransport interface, the bandwidth increases with the block size to about 1GB/s for requests of 64 bytes, which is the maximum packet size of HyperTransport. For larger request sizes, the bandwidth drops to about 0.6GB/s and again increases linearly to peak at about 1GB/s for multiples of HyperTransport's maximum packet size.

Figure 3 presents the measurements for read requests. The bandwidth for reads from external memory increases linearly with the request size, with drops whenever the configured burst size is exceeded. Again, the bandwidths are maximal for requests which are multiples of the burst size. Interestingly, for some non-matching request sizes 4-cycle bursts are superior to 8-cycle bursts which tend to match the lower 2-cycle performance. The reason for this behavior is yet unclear.

The achievable bandwidth for read requests to host memory increases linearly for request sizes up to 64 bytes, resulting in a bandwidth of about 1.3GB/s. Further increasing request sizes lead to a drop to 1.2GB/s, before the bandwidth levels out at 1.3GB/s without any further significant deviations.

III. An Compositing Accelerator for a Parallel Rendering Framework

A. Parallel Rendering

Modern *3D Computer Aided Design* applications such as production planning and optimization and mechanical component design require substantial computation for rendering 3D scenes. Using highly detailed object models originating from CAD tools or 3D scanners, such simulation and visualization applications generate complex 3D scenes with huge numbers of polygons to be rendered. Parallel rendering approaches can meet the stringent performance requirements, especially for interactive modes of operation. Molnar et al. [8] introduce three approaches for parallel rendering: sort-first, sort-middle and sort-last.

Our case study focuses on an in-house parallel renderer using the sort-last approach. A master node divides a scene (frame) into $N-1$ subscenes and distributes the workload to $N-1$ rendering nodes. The subscenes have roughly the same number of geometry primitives (polygons) but the assignment of polygons to rendering nodes is arbitrary. Each rendering node runs a rendering pipeline that computes a set of display primitives (pixels) from the received geometry primitives. A rendering node computes two buffers for its subscene, the frame buffer containing the color information and the z-buffer containing the depth information for each pixel. The resulting $2 \cdot (N-1)$ buffers are transferred back to the master node for compositing. Compositing performs the sorting step by comparing the distances of the $N-1$ candidates for each pixel to the view plane. Only the nearest display primitive is visible.

The parallel renderer is part of a visualization application that can be run in batch-mode or interactively. The master node stores or displays the composited picture and distributes the next subscenes to the renderer nodes. The application is implemented with MPI using double buffering for frames to overlap computation and network communication. To analyze potential bottlenecks in the parallel rendering application, it suffices to look at the following parameters:

```
void compose(int* pic_a,int* pic_b,int size) {
  int *z_a=pic_a+size;
  int *z_b=pic_b+size;
  for(int i=0;i<size;i++) {
    if(z_b[i]<z_a[i]) {
      pic_a[i]=pic_b[i];
      z_a[i]=z_b[i];
  }
}
```

Listing 1. Code for the compositing function

- H and W are the height and the width of a subscene (frame) in pixels. Each rendering node processes a frame buffer and a z-buffer for each subscene, resulting in $P = W \times H \times 8$ bytes of data.
- T_R [s/frame] is the time required for one rendering node to compute its subscene. T_R depends on the size of the subscene, i.e., on P and the number of polygons. Since the workload is evenly distributed, we can work with an average value over all rendering nodes.
- T_C [s/frame] measures the computation time for the master node and comprises the compositing time and the time needed to display or store the resulting picture and redistribute the next workload. The compositing time is dominating and depends on P and the number of rendering nodes, $N - 1$.
- B_{net} [byte/s] is the bandwidth of the link over which the master node connects to the computer network.

The aggregate data bandwidth generated by the rendering nodes is $(N - 1) \times P/T_R$ [byte/s]. Depending on P, the complexity of scenes, and parameters of the compute cluster, a reasonably designed and configured system will try to set the number of rendering nodes such that the network or the master node is not saturated. Bottlenecks occur if the aggregate renderer bandwidth exceeds B_{net} or if the master node's computation time T_C limits the throughput. The latter will be more likely in practice which makes compositing an interesting target for acceleration.

B. Compositing Accelerator Performance Estimation

Listing 1 shows the pseudocode for the compositing function. The function is computationally very simple, only comprising a regular loop with comparisons and assignments which can be parallelized in a straight-forward way. However, the main challenge for accelerating this code is to establish a continuous stream of data through the computing core. The main design decisions for the FPGA accelerator are (i) where to store the pictures and (ii) the number of parallel comparisons to implement. Referring to the communication characterization for the XD1000 system (Section II), we conclude that internal memory is not available in sufficient capacity for storing realistically large pictures.

To support the streaming nature of the application, both

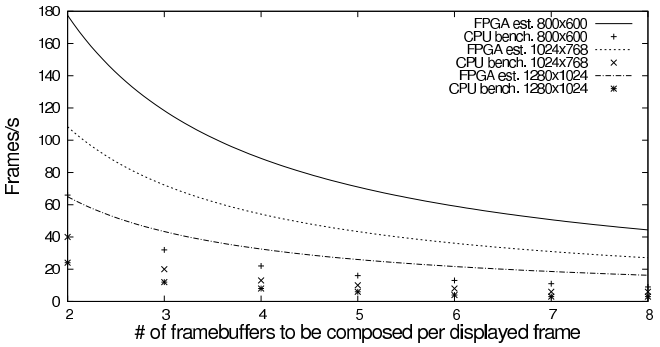

Figure 4. Comparison of CPU performance and estimated FPGA performance for the compositing function

the CPU and FPGA implementation of the compositing function need to store the picture (frame and z buffers) from the first rendering node in memory. Then, the stored picture is compared to each newly arriving picture and updated if necessary.

Overall, the accelerator has to transfer $(N-1) \times P$ bytes of data from the host to the FPGA module. Since the used OpenMPI implementation [9] is unable to perform MPI receives directly into the FPGA's address space, we have to explicitly transfer the frames to the FPGA.

We transfer the first frame to the external memory on the accelerator module. Since $B_{rd}(HT)$ is lower than $B_{rd}(EM)$, the time needed for this first phase of the accelerator is determined by the HyperTransport performance. The following $N-2$ frames of size P stream from the host to the FPGA accelerator and, at the same time, the stored frame streams from external memory to the FPGA, and the resulting frame streams back to external memory. The time required for this second phase of computation is dominated by memory accesses and given as $\frac{P}{B_{rd}(EM)} + \frac{P}{B_{wr}(EM)}$ for the external memory and $\frac{P}{B_{rd}(HT)}$ for the host memory read over the HyperTransport link. Consulting the bandwidth measurements of Section II, i.e., Figure 2 and 3, we conclude that for reasonably chosen request sizes the host memory access will limit the execution time. This holds only if external memory is accessed with request sizes that are multiples of the controller's burst size. The actually chosen burst size of the memory controller influences the access time for external memory, but has no effect on the overall compositing application.

For the last frame, we read P bytes from each host memory and external memory but write only $P/2$ bytes back to host memory since the resulting z-buffer is not needed for displaying the picture. Despite the fact that the write bandwidth to host memory is much lower than the read bandwidth, the execution time of this last accelerator phase will be determined by reading host memory since writing involves only half the data size. Using this performance estimation we compare the execution times for the FPGA

compositing accelerator with measured CPU execution times for the same task. Figure 4 shows the results of this performance comparison. The FPGA accelerator achieves higher frame rates than the CPU. When composing the results from four rendering nodes operating in parallel, the estimated speedup ranges from $4\times$ to $4.5\times$. For eight parallel rendering nodes, the speedup is between $5\times$ and $5.7\times$. Naturally, the achievable frame rate decreases with an increasing number of subpictures that need to be composed for generating the final picture. However, Figure 4 presents only the compositing time. The overall performance of the parallel rendering application will scale with the number of rendering nodes, up to the point where we hit a bottleneck.

IV. Implementation and Measurements

A. Accelerator Design

Based on the analysis of the compositing function and the bandwidth estimations, we implement a compositing accelerator using our IMORC infrastructure. The accelerator architecture is shown in Figure 5 and comprises a host interface core encapsulating the HyperTransport interface, a memory controller for accessing external memory, and a number of cores implementing the actual compositing function.

As shown in Listing 1, for each iteration we need to read two values of the z-buffer and two values of the frame buffer, and write one z-buffer and one frame buffer value. We always write back one z-buffer and one frame buffer value in order to keep regularity in the data flow. Figure 5 shows that we employ a separate stream buffer core for each of these six memory accesses. The memory controller is configured to 8-cycle bursts and the request size is set to 128 bytes. A seventh stream buffer handles frame buffer writes in phase three of compositing, when the resulting picture is streamed back to the host.

The composer core of the accelerator implements the actual comparison of z-buffer values and runs at 200MHz synchronously to the HyperTransport core. To fully utilize the HyperTransport link to host memory, which is the limiting factor in the design, we unroll the loop in Listing 1 and compute two iterations in parallel. Hence, all links from composer cores to the HyperTransport interface core are 64-bit wide.

The remaining two cores are the request core and the compositing controller. The request core is responsible for issuing read/write requests to the stream buffer cores. The compositing controller includes control registers for parameters such as the width and height of a picture and the number of rendering nodes, as well as logic needed for exchanging state information between the host and the FPGA accelerator.

B. Performance Evaluation

In order to evaluate the overall performance of the parallel renderer, we implement a test system comprising an Intel Clovertown machine connected via Infiniband to the XtremeData XD1000. The Intel Clovertown contains two quad-core processors running at 2.66GHz and 8GB of main memory and implements 1..8 rendering nodes. The XD1000 implements the master node including the compositing function. Theoretically, the Infiniband interconnect provides a peak bandwidth of 10GBit/s. Measurements with the Intel IMB benchmark show that our test setup reaches a sustained Infiniband bandwidth of 700MB/s.

Figure 6 compares the overall performance, measured in frames/s, of the purely CPU-based parallel renderer with the FPGA-accelerated parallel renderer. Additionally, the figure shows the frame rates that could be achieved if the Infiniband interconnect was fully utilized. Figure 6 covers all three system states of the parallel renderer application. In the following, we discuss these states for a resolution of 1280×1024 pixel: For a small number of rendering nodes (up to three) the performance increases linearly, and since neither the network nor the master node is saturated, there is no benefit from using FPGA acceleration. Increasing the number of rendering nodes further (four to five nodes) makes the master a bottleneck. The CPU-based system achieves its maximum frame rate for four rendering nodes. In this system state FPGA acceleration is highly useful as the improved compositing performance allows us to achieve a higher peak frame rate. For example, for four rendering nodes the performance gain is $1.25\times$. At a certain point (six nodes and more), the aggregate bandwidth from the rendering nodes saturates the network. Figure 6 clearly shows that the performance of the FPGA-accelerated system is limited by the Infiniband bandwidth. Obviously, also in this state FPGA acceleration is beneficial and delivers an improvement in the frame rate of $2.1\times$ for eight rendering nodes.

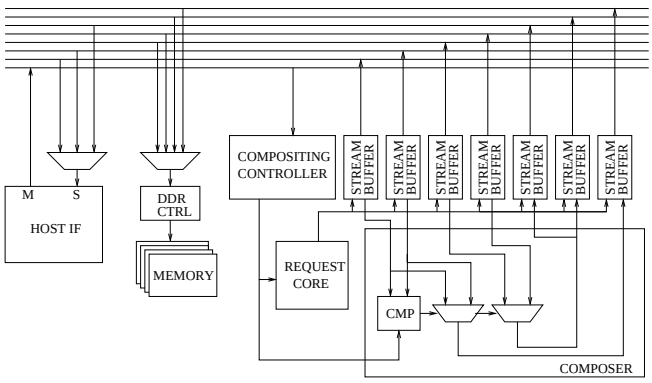

Figure 5. Architecture of the compositing accelerator

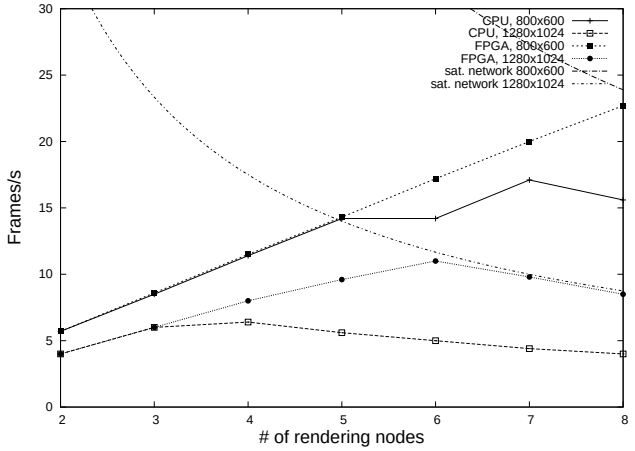

Figure 6. Performance values of the complete rendering application

V. Conclusion

One specific challenge in mapping applications to high-performance reconfigurable computing platforms is to optimally utilize the available memory architecture and layout. Leveraging our IMORC infrastructure and architectural template for creating reconfigurable accelerators, we experimentally characterize the communication performance for cores mapped to the XtremeData XD1000 reconfigurable computer. The achievable bandwidth is strongly varying with request sizes and controller configurations. An optimal design point cannot be chosen locally, but requires us to consider the complete accelerator architecture and the overall application.

We elaborate on accelerating the compositing function of a parallel rendering application as a case study. Computationally, compositing is a rather simple function. Any performance improvements will have to come from an optimized memory architecture that supports the streaming nature of the application. Using our IMORC infrastructure and the XD1000 communication performance characterization, we devise an accelerator which is an estimated factor of $5.7\times$ faster than the CPU, although the HyperTransport interface in the XD1000 forms a bottleneck.

Experiments with the overall parallel rendering application show that the FPGA accelerator is useful when the aggregate bandwidth of the distributed rendering nodes drives the compositing node into saturation. In this situation, the accelerator is able to double the achievable frame rate of the overall parallel renderer. This is significant since one FPGA module does not only boost the performance of a single CPU node, but increases the usability and thus the practical value of a visualization compute cluster which constitutes a considerable investment.

In this paper, we report on measurements conducted on a setup with a rather small number of rendering nodes and limited Infiniband bandwidth. A practical compute cluster will employ a higher number of rendering nodes connected through a faster network. This will increase the pressure on the master node and allow the FPGA accelerator to achieve even higher speedups (see the network bandwidth limitation for the accelerator's performance in Figure 6). An FPGA module with one or several network interfaces would eliminate the HyperTransport bottleneck and allow us to further increase the compositing performance.

Obviously, alternative approaches to parallel rendering such as sort-first and sort-middle or parallel compositing will show other bottlenecks and, perhaps, remove the need for FPGA acceleration. A comprehensive study of different parallel rendering approaches and their bottlenecks is, however, beyond the scope of this work.

Future work will include porting the IMORC infrastructure and the compositing accelerator to the latest generation of reconfigurable computers such as Nallatech's in-socket FSB modules. The parallel rendering application will be set up and tested on a larger cluster with a fast Infiniband interconnect.

References

[1] T. Schumacher, C. Plessl, and M. Platzner, "IMORC: Application Mapping, Monitoring and Optimization for High-Performance Reconfigurable Computing," in *Proc. Symp. on Field-Programmable Custom Computing Machines (FCCM)*.

[2] ——, "An Accelerator for k-th Nearest Neighbor Thinning based on the IMORC Infrastructure," in *Proc. Int. Conf. on Field Programmable Logic and Applications (FPL)*, 2009.

[3] L. Shannon and P. Chow, "Simplifying the Integration of Processing Elements in Computing Systems Using a Programmable Controller," in *Proc. Symp. on Field-Programmable Custom Computing Machines (FCCM)*. Washington, DC: IEEE Computer Society, 2005, pp. 63–72.

[4] C. Steffen, "Parametrization of algorithms and FPGA accelerators to predict performance," in *Proc. Reconfigurable System Summer Institute (RSSI)*, 2007, pp. 17–20.

[5] B. Holland, K. Nagarajan, C. Conger, A. Jacobs, and A. D. George, "RAT: A Methodology for Predicting Performance in Application Design Migration to FPGAs," in *Proc. High-Performance Reconfigurable Computing Technologies and Applications Workshop (HPRTCA)*, 2007.

[6] S. Koehler, J. Curreri, and A. D. George, "Performance analysis challenges and framework for high-performance reconfigurable computing," *Parallel Computing*, vol. 34, no. 4-5, pp. 217–230, 2008.

[7] D. Slogsnat, A. Giese, and U. Brüning, "A versatile, low latency HyperTransport core," in *Proc. Int. Symp. on Field Programmable Gate Arrays (FPGA)*. New York, NY, USA: ACM, 2007, pp. 45–52.

[8] S. Molnar, M. Cox, D. Ellsworth, and H. Fuchs, "A sorting classification of parallel rendering," Tech. Rep. TR94-023, 8, 1994.

[9] "OpenMPI homepage," http://www.open-mpi.org/.