On-chip Network Interfaces supporting automatic burst write creation, posted writes and read prefetch

Radu Stefan, Jason de Windt
Computer Engineering
Delft University of Technology
Email: R.A.Stefan@tudelft.nl, J.C.deWindt@student.tudelft.nl

Kees Goossens
Electronic Systems
Eindhoven University of Technology
Email: K.G.W.Goossens@tue.nl

***Abstract*—Networks-on-Chip are seen as a scalable solution for facilitating the development of Systems-on-Chip with an increasing number of IP cores. Many studies already address the implementation details of such networks and a large effort has been invested in optimizing the routing strategy and the organization of the network, however by comparison the interface between the network and the IPs has been largely ignored. In this study, we explore optimizations that can be performed at the layer that connects the IPs to the services offered by the NoC. In our FPGA prototype, a MicroBlaze soft-core is connected to a remote memory via the Æthereal NoC. By employing our optimizations to the interface between the MicroBlaze and the NoC, we demonstrate an improvement in terms of speed above 880% in memory intensive tests and of up to 12% in real life applications with little use of communication.**

I. Introduction

The shrinking feature sizes in silicon technology allowed both an increase in the processing power and the number of IPs integrated on a single chip. At the same time consumer expectations adapt to the benefits provided by technology by both diversifying and demanding higher quality from already existing services. It is now customary for mobile phones for example to provide maps, route planning, gps and video capabilities.

Building complex systems to support these features, possibly integrating IPs from multiple vendors becomes then a difficult, time-consuming task. Employing networks-on-chip for connecting together the many IPs can significantly reduce the development effort [1], [2] and facilitate verification [3]. The reuse of interconnect elements like routers, network interfaces, arbiters, long communication lines, clock domain crossings is yet another benefit.

Networks on chip offer an optimized interconnect capable of running at higher frequencies and having better scalability properties than a standard bus design. The price that is often paid in exchange is an increase in latency in terms of clock cycles. However we find this increase in latency to be unavoidable as the size of the system itself increases.

Previous studies focused to a large extent on the routing of messages inside the network, while little effort has been devoted to optimizing the way the IPs are connected to the network. We focus in this work on optimizing the interface between IPs and the “network core”, the actual infrastructure transporting data between different points on the chip. This layer is in charge of performing protocol conversions between the bus standards used by IPs and the internal message format used inside the network.

We propose three main optimizations 1) write coalescing, or automatically combining writes to consecutive addresses into single burst transactions, 2) a mechanism for ensuring memory consistency while at the same time allowing posted writes and 3) a mechanism for offering prefetch functionality to IPs otherwise lacking it. The first two optimizations are transparent from the point of view of the application developer while the last one requires modifications at the software level.

To prove our concept and evaluate the performance of the proposed optimizations, we implement our design on FPGA, using MicroBlaze cores as processing elements. As benchmarks we use several memory intensive computation loops, and a real-life application, a JPEG decoder with a much higher computation-to-communication ratio. The total gain obtained by applying all three techniques ranges from up to 880% (or in other words a speedup of 9.8) in memory intensive applications, to a modest 12% in the case of the JPEG application, which is not communication but computation bounded. It shall be noted though that this speedup represents the speedup of the entire system, while the optimization targeted only one of its components.

The remainder of this paper is structured as follows: The Section II shows the relation of our study to previous work. Section III presents the architecture used in our experiments. Section IV explains the optimizations employed and expected benefits. Experimental results detailing the performance improvement can be found in Section V. The last section presents our conclusions.

II. Related Work

Many Network-on Chip implementations already exist, some providing mature flows with the possibility of generating a hardware description of the entire infrastructure [4], [5], [6], [7]. The authors often provide complete solutions covering all aspects of interconnect design including the interface to the IPs.

Many studies also exist covering the optimization of different aspects of the interconnect, like routing [8], [9] and network topology [10], [11], however we have found that little effort was dedicated to optimizing the interfaces between

traditional IPs, unaware of the existence of the NoC and the NoC itself.

Of the literature dedicated specifically to network interfaces we mention [12] which is a precursor of the architecture we use in our current research proposed separating the networking function of the NI from the actual interface to the IP. The implied benefit is the possibility of easily reusing part of the design when developing interfaces to other types of IPs. The same idea is also present in [13].

Options for connecting IPs to NoC are explored by [14]. The work considers solutions based on both software and hardware, with an approach focused on modifying the IP building wrappers around it. By comparison we choose to leave the IP unaltered and only interface with it based on its existing connections to standard buses.

Wrappers are also used by [15], with the advantage of both hiding the implementation details of the interconnect from the IP and avoiding modifications of the IPs and the network internals.

None of the works previously mentioned suggests performance optimizations at the level of the NI.

In the domain of high-performance cache-enabled processors, write coalescing is a function commonly performed by write buffers [16] or by the cache itself, as in the case of write-back caches. Prefetching has also been studied extensively [17]. Our work is focused on less costly solutions which do not assume the presence of caches and integrate the functionality of the write buffer into the communication infrastructure.

Despite the fact that the optimizations presented in this paper are generally well known, we believe to be the first to propose and analyze them in the context of Networks-on-Chip.

III. SoC ARCHITECTURE

One of the most important choices defining the behavior and performance of a multi-IP system is the memory and communication model. From the early multiprocessors, two models have emerged, the shared memory paradigm and the message passing paradigm for example as implemented by MPI [18]. The first encourages a global view of the system in which elements can access each other's resources as part of a global address space, while the second regards cores as performing individual, separate tasks and only exchanging data in blocks when the processing is finished or about to begin.

The system we envision has multiple IPs, each with a small amount of local memory (scratchpad) for fast access to temporary data and larger shared remote memories, possibly off-chip, available to all IPs. It may also be possible for IPs to access each other's memory directly, as it has been already demonstrated in [19].

In our prototype, the underlying communication network is provided by the Æthereal network-on-chip [4], [20], although our proposals are not tied to one particular implementation. Æthereal is a connection oriented, TDM based NoC, that can be built and optimized according to a set of user specifications in terms of desired bandwidth and latency.

The main components of the interconnect are described in Figure 1.

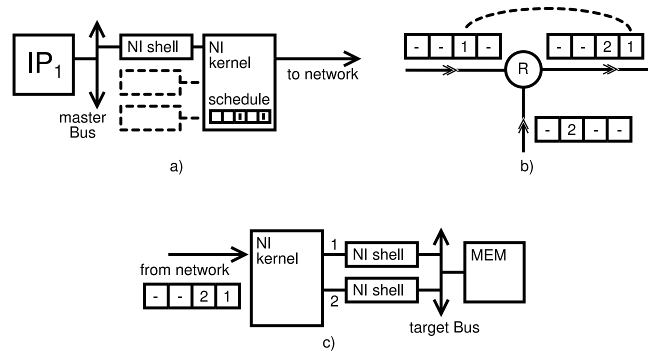

Fig. 1. System based on the Æthereal network, and example allocated timeslots.

As illustrated in figure 1a, requests from IPs acting as masters are translated by NI shells into the message format used inside the network. The connections between IPs and NI shells are either back-to-back or through a bus, using the native bus standard of the IP. A bus transaction with parallel signal groups (command, address, data) is serialized into a single signal group having the word width of the communication link.

A separate module, the NI kernel, is responsible for packetizing the request and sending it over the network. An NI kernel offers point-to-point connections, behaving like FIFOs between network endpoints, usually NI shells. An NI kernel may service multiple NI shells. Using the formalization of the OSI protocol stack [21], the NI shells provide the services of the session layer and the NI kernel the services of the transport layer. A more detailed discussion about the correspondence with the OSI layers can be found in [22].

At the remote side (Figure 1c) NI shells translate the encoded message into the protocol of the target bus, to be served by a slave IP, for example a memory. The internal message format is independent of the bus protocol and pairing "master" and "slave" NI shells using different bus protocols is possible. Each remote NI shell serves a single connection and several instances may be necessary for serving requests from multiple masters. Although not represented in Figure 1, connections are bidirectional and allow sending back confirmations and results of the read requests.

Inside the network (Figure 1b) a contention-free routing strategy [4] is employed. A global schedule in the form of a revolving slot table is used to ensure packets can be forwarded at each router without delay. The schedule is computed at design time and a number of timeslots is reserved for each connection to be used exclusively for that connection. This ensures that: 1) the communication behavior including bounds on bandwidth and latency can be accurately described (in other words predictability) and 2) separate applications within a system do not interfere with each other (a property we call composability) .

The assumptions of predictability and composability allow us to use a simpler test setup, as presented in Section V. Since several tasks running on multiple processors are guaranteed not to interfere with each other, it is not relevant whether those tasks and processors are present or not, as long as their share of network bandwidth is taken into account during the allocation phase.

IV. Proposed Optimizations

We propose three optimizations in two categories: one is transparent from the point of view of the IP, one requires the explicit intervention of the application developer.

The first optimization consists of using write coalescing, for a better use of bandwidth, the second consists of an efficient and safe, from the memory consistency point of view, mechanism for performing posted writes, and the third uses existing features of the IP to offer otherwise absent prefetching functionality.

Our prototype system uses a MicroBlaze soft-core and interfacing is performed on the PLB bus and with the FSL links. We implement our optimizations into NI shells for the PLB bus.

A. Write coalescing

Memory transactions, encoded as messages, traverse the network in a serialized fashion, with headers, addresses and data sharing the same bandwidth. Longer messages, with multiple words of data for a single header/address pair would thus have better payload efficiency.

Burst transactions on the bus side correspond to such messages inside the network, however, not all IPs have the capability of generating burst transactions. The instruction set of the MicroBlaze soft-core does not provide an instruction to write to memory more than one word at a time, and this is true for many other simple IPs also.

The solution we found was to automatically identify sequences of write operations to consecutive addresses and combine them into a single message for the purpose of traversing the network. At the destination NI shell these messages can then be split again into individual write operations or optionally they can be served directly to the destination bus if burst transactions are supported.

We perform this write coalescing as long as the addresses are consecutive, the burst length has not reached the maximum value, 32 for the message format we are currently using, and there is data in the outgoing network queue. This last condition is to ensure that we are not unnecessarily delaying messages.

A diagram of the NI shell we implemented is found in Figure 2. The shell uses several independent queues for storing the data, the address and the headers. Data is copied from the PLB bus to the send queue whenever a write transaction is accepted, new headers and address are copied to the header queue whenever a transaction cannot be merged with the previous transaction or the network is idle and the current transaction can be processed immediately.

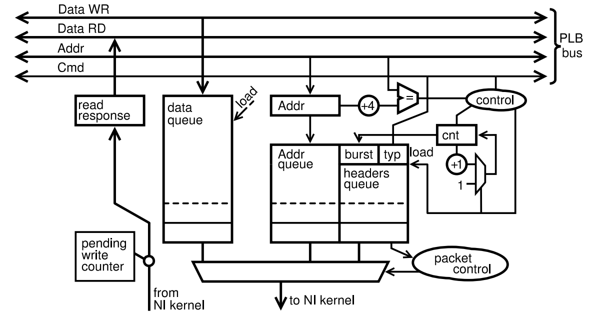

Fig. 2. NI shell supporting burst write

It would also be possible to implement the same functionality for read transactions, however in our case the processor will always stall until a read transaction is complete thus making this feature useless.

For comparison, the NI shell without burst or posted write support (Figure 3) consists of a simple serializer shifter. The entire request message is generated in a single cycle when accepting a request from the bus and is sent to the network word-by-word in the following cycles. Because all requests are blocking additional queues would not provide any benefit.

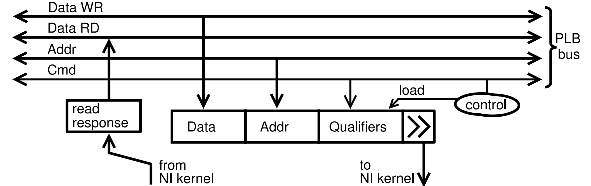

Fig. 3. NI shell without burst or posted write support

B. Posted writes and memory consistency

Memory consistency is a term used to describe the expected system behavior with regard to the order in which memory writes are visible to the different IPs in the system. In general a stricter memory model provides more guarantees regarding the order of memory operations thus making the programmer's job easier, but makes the hardware implementation more difficult and less efficient.

The strictest consistency model in use is the sequential consistency [23], which requires that the memory operations of each individual processor, as seen by the other processors in the system appear to execute in the order of specified by the processor's program. Any particular interleaving between the instructions of different programs is allowed, but the same interleaving shall be seen by all processors or processes. This is essentially the same result that can be expected from multiple threads running on a single processor, and studies in general agree [24], [25] that this is what programmers intuitively expect their machines to behave like.

We first present a basic hardware implementation that would provide sequential consistency, then show how the requirements can be relaxed to allow performance optimizations. We

assume from the beginning an architecture where memory requests, both read and writes can be pipelined, but the ordering of requests, even between reads and writes is preserved within the pipeline. It must be mentioned that allowing reads to bypass writes is sometimes accepted as an optimization [26], reads being considered more important, as the reader process was likely stalled waiting for data. Although we do not accept reordering of normal read operations with respect to write operations, we allow it for prefetch reads as it will be explained in the following section.

Despite the fact that our architecture does not employ caches, and the system seems to maintain ordering of requests, consistency problems may still arise. Consider the following scenario involving a transfer between a producer and a consumer of data, represented in Figure 4. The producer (node A) generates data items and places them in some memory location, for example in external memory. Upon completion, it signals to node B that the data is ready to be processed.

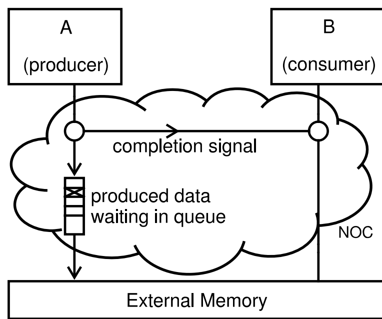

Fig. 4. Consistency issues raised by signals arriving at different destinations with different delays

Because the NoC allows messages with different destinations to travel independent of each other, it may be possible for the confirmation message from A would arrive while the data is still queued waiting to be written to the memory. An attempt by B to read the data through its own connection to the memory would return stale values. Not only it is easy to see that this produces an erroneous program behavior, but it does that by breaking the assumptions of sequentiality, which required that the confirmation message, for example a write operation to a specific flag in memory, would be seen by B strictly after the write operation to main memory was completed.

One possible solution would be that all write operations need to be confirmed (non-posted writes), preventing A from sending a message to B before the previous operation has been completed. Unfortunately, this would completely prevent pipelining, resulting in a severe performance penalty.

Our solution consists of performing posted writes (write operations which are acknowledged at the source without waiting for confirmation from the target), but keeping track in the NI shell of which write operations have been confirmed and which have not been. Future writes to different destinations will be stalled until all pending writes have been confirmed, in other words, consecutive writes to the same destination are pipelined like posted writes but writes to different destinations behave like non-posted ones.

Let us analyze why this ensures sequentiality. The sequential consistency model required that from each process's point of view, all memory accesses in the system seem to take place in the same order, with an arbitrary interleaving of accesses belonging to different processes, but maintaining the program order for accesses of each individual process. This order is not necessarily the order given by the physical time of issue of each access, but in our case it can be chosen as the order of physical time of completion of each access (the physical read or write to each memory). Because we did not allow the reordering of write operations in each pipeline and all messages between two nodes travel on the same path, writes to one memory from one specific processor will occur in program order. Because our system waits for completion confirmation when switching between different targets, writes to memory B that occur in the program after writes to memory A will also take place physically later in time. It is necessary for the same to happen for read requests so our mechanism also enforces this.

It is possible to further relax these restrictions to allow higher system performance. For example when one memory is not read by any other process, like it might be the case of a video frame-buffer, it is not necessary to order the accesses to that memory with respect to accesses to other memories. It is also possible to emulate the behavior of other consistency models by only partially connecting the command signals used to block some memory accesses until accesses to other modules have completed and mapping synchronization variables to specific memories, Figure 5. We can for example implement the weak consistency model [27] by mapping synchronization variables in one memory and enforcing sequentiality between that memory and each of the data memories, but not among data memories. A consistency model similar to the Release/Acquire model [28] could be implementing by splitting the synchronization memory in two separate memories and enforcing only one way ordering between accesses to these memories and the data memories, for example an access to Acquire must complete before an access to Data, but not the other way around.

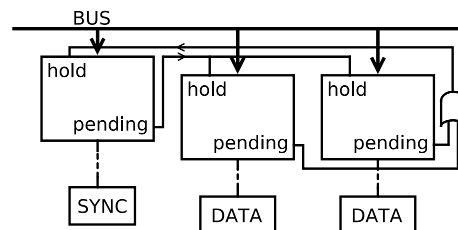

Fig. 5. Emulation of a weak consistency model

C. Software prefetch

The previous proposals are transparent from the point of view of the software developer. Except for the variation in performance, the system behaves no differently from a system with only local memory, even the introduction of the NoC between an IP and its memory is completely transparent. The following proposal however introduces a feature that needs to be explicitly used by the programmer, and in some cases requires a significant rewrite of the software.

Networks-on-chip and large scale interconnects in general have a higher latency than back-to-back connections. Given that, it is natural that we look for ways to cope with that latency. While posted writes provide an efficient method of dealing with write transactions, For read transactions we do not have a similar solution. Processors providing out-of-order execution mitigate the problem to some extent by allowing other instructions to continue while data is being fetched from memory. This type of approach though is expensive and not often predictable.

In our system we opt for introducing an explicit command to bring data from memory some time in advance before actually being needed (prefetch). Note that, unlike other prefetch implementations, the data is not used to update a local cache from which it can be later read using a normal read instruction, but instead is deposited in a queue from where it needs to be explicitly read by the program. The technique can be seen as a software split transaction, where the request for data is decoupled from the receiving of data. Example code is provided in Figure 6.

```
    // original loop
 1  for (i=0; i<n; i++)
 2  {
 3       v=a[i];
 4       // use value v
 5  }

    // modified loop
 6  prefetch(a[0]);
 7  for (i=0; i<n; i++)
 8  {
 9       if (i<n-1) prefetch(a[i+1]);
10       v=read_prefetch_fifo();
11       // use value v
12  }
```

Fig. 6. Example prefetch loop compared to original code

In the modified code, one or more data items are requested prior to entering the main processing loop (line 6). In each loop iteration, more data is requested in advance (line 9) with care being taken to not exceed the limits of the actual input. Data from the previous requests which should have already arrived in the buffers is then retrieved in line 10.

The hardware necessary to support the prefetch mechanism consists of an NI shell connected to the FSL bus of the MicroBlaze processor. The shell behaves as a processing FIFO, accepting at the input port memory read requests and delivering at the output port the retrieved data. The shell accepts additional commands for configuring the size of burst reads.

All prefetch operations are under explicit control of the program, which may also have to ensure ordering with respect to the normal read and write operations. Currently the code must be manually edited by the software developer, which is also what we did in our experiments. Although in principle it might be possible to offload this task to the compiler for example, this task is far from trivial and is complicated by consistency issues especially in multi-core systems [29].

All our proposals were implemented and tested in FPGA. The hardware cost of the implemented shells relative to the size of the entire system is presented in Figure 7. In our test system, the prefetch module was connected to a dedicated NI kernel, thus doubling the size of the interconnect, however in practice this would not be necessary. The optimized shell implements both the burst and posted writes.

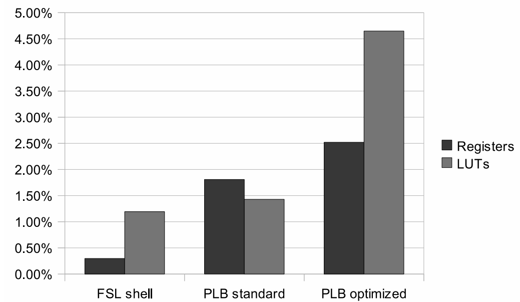

Fig. 7. Hardware cost of the shells in FPGA implementation

V. Experimental Results

We have performed our tests on three similar systems, all having the structure presented in Figure 8. For the main PLB target we have substituted three different NI shells, one performing only non-posted operations, one capable of performing posted writes with the described safety mechanism, and one performing both burst coalescing and posted writes. The FSL interface was always present, but since its use is always explicit, we specify through the MicroBlaze program whether it should be used or not.

Our tests involved only one processor, however we simulate the effects of having multiple processors by allocating only a fraction of the link bandwidth inside the network. For a fair comparison, in the tests where both the FSL and PLB link is used, we restrict the total bandwidth for both links to the bandwidth offered to the PLB alone in the non-prefetch scenario. The most efficient way of distributing the bandwidth between the PLB and FSL connection was found using exhaustive search with increments of 25%. We also perform an additional test, where two separate full-bandwidth links are provided, one for each link (marked B in the result graphs, Figures 9 - 14).

For the software we have chosen several benchmarks, ranging from synthetic to real applications. In short, these are: read

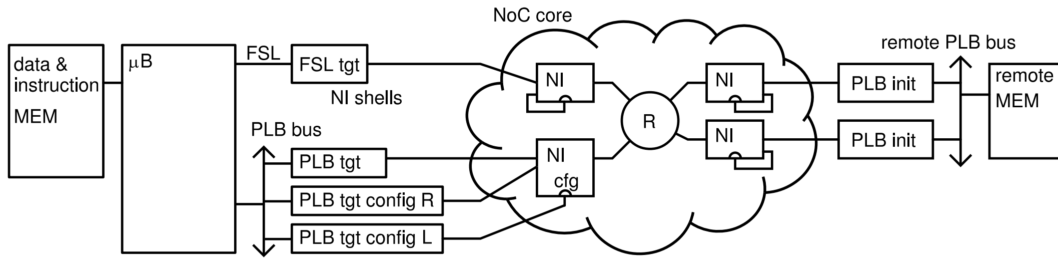

Fig. 8. Test setup: one MicroBlaze core is connected through two separate channels, one on the PLB bus and one on an FSL link to a remote memory.

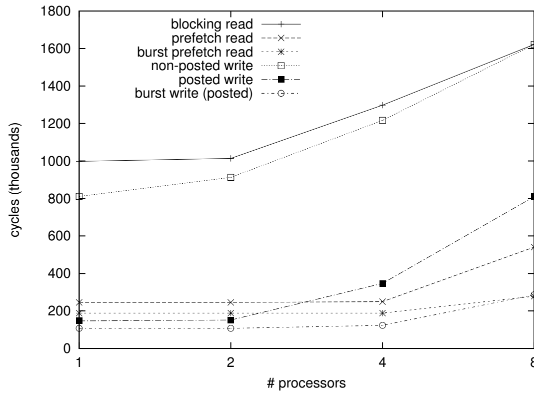

Fig. 9. Performance of Read and Write tests under different setups

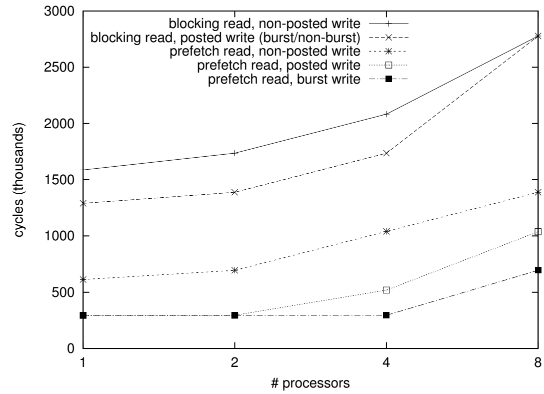

Fig. 10. LL kernel 1

and write loops, several kernels with a relatively high amount on memory activity chosen from the Livermore Loops [30], and a JPEG decoding application.

Our first set of tests consists of a read and a write loop for an array of 16 K words. Due to their intensive use of memory transactions they show the highest performance variations among our tests. The results are presented in Figure 9. As the number of processors in the system is increased, there are two factors causing performance degradation: an increase in latency and a reduction in the available bandwidth. The posted write and the prefetch read tests, which are largely immune to the increase in latency provide a good indication of the point when bandwidth becomes the limiting factor. The use of burst read and write operations alleviates the effect of the bandwidth reduction. Not using posted write and prefetch read incurs a large penalty even for the one processor configuration.

The Livermore Loops are small kernels that were used for testing the performance of supercomputers. They are representative for scientific applications but we have chosen to use them as they provide a variety of memory access patterns. The versions we employed were translated in C and were set to use only integer operands. The loops had to be manually modified to perform software prefetch, and thus we only used four of them in our tests, more specifically kernels 1 (hydro), 6 (linear recurrence), 12 (first difference), and 21 (matrix multiplication). The results of the Livermore Loops tests are shown in Figures 10-13. While the latency hiding techniques provide a significant advantage, the burst optimization does not always produce an improvement as it depends on the patterns

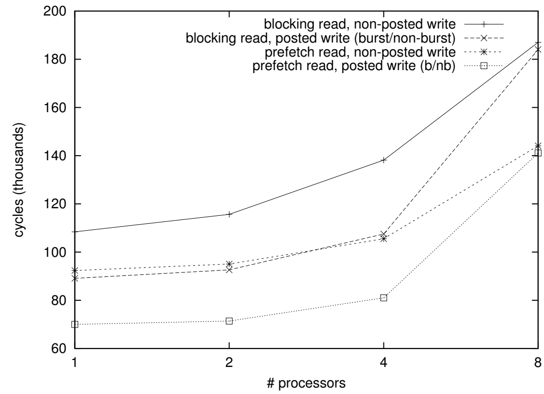

Fig. 11. LL kernel 6

of memory access. The numerical results for all tests can be found in Table I.

The JPEG application is a complex program with a high computation-to-communication ratio. It requests data from an external memory, performs calculations on the retrieved data and writes back the result to the external memory in small bursts which are essentially memory copy operations on the already decoded data. The read operations are almost always sequential with the exception of a few initial headers, while the write operations follow an access pattern characteristic for accessing a sub-matrix out of a larger matrix.

The results are presented in Figure 14. The difference is

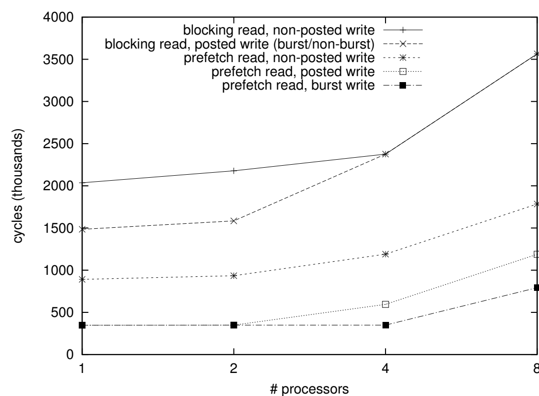

Fig. 12. LL kernel 12

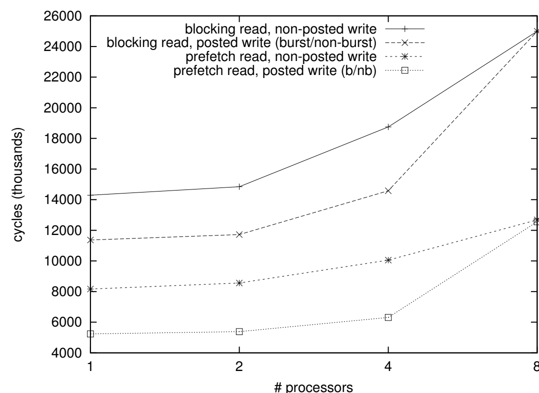

Fig. 13. LL kernel 21

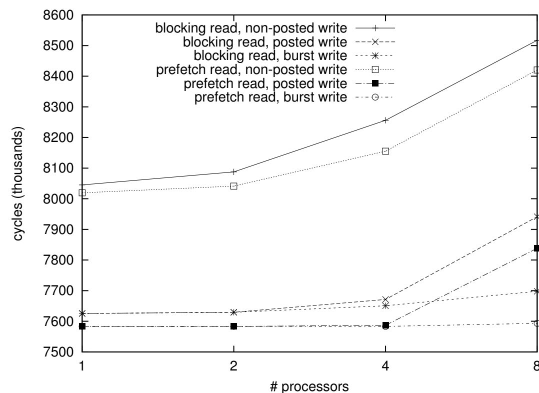

Fig. 14. JPEG

more pronounced when the available bandwidth is severely limited as it is the case with the 8 processors scenario. The improvement in the case of the JPEG application is a modest 6 to 12%, however this represents an increase in the performance of the entire application, while the optimization targeted a single component of the system.

Figure 15 shows the average performance increase over all bandwidths obtained in all applications. The performance increase due to particular optimizations is represented separately. The performance increase provided by the posted writes is presented in two separate scenarios, when prefetch read is not used and when it is. The performance of burst posted writes is presented in comparison to normal posted writes. The overall speedup is the speedup for the entire system with all optimizations enabled. Numerical results are presented in Table I.

test	pf	pw	pw w/pf	bw w/pf	overall
write-1p	-	451.08%	-	36.85%	654.17%
write-2p	-	500.70%	-	41.24%	748.43%
write-4p	-	250.46%	-	182.01%	888.33%
write-8p	-	100.28%	-	182.36%	465.52%
read-1p	429.66%	-	-	-	429.66%
read-2p	437.93%	-	-	-	437.93%
read-4p	588.53%	-	-	-	588.52%
read-8p	481.70%	-	-	-	481.71%
LL1-1p	158.95%	23.08%	108.07%	-	438.81%
LL1-2p	149.76%	25.00%	135.88%	0.02%	489.24%
LL1-4p	100.12%	20.00%	100.49%	76.17%	606.85%
LL1-8p	100.00%	0.01%	33.75%	49.38%	299.61%
LL6-1p	17.41%	21.69%	31.91%	-	55.01%
LL6-2p	21.72%	24.85%	33.10%	-	61.95%
LL6-4p	30.89%	28.48%	30.26%	-	70.39%
LL6-8p	29.78%	1.61%	2.12%	-	32.53%
LL12-1p	128.41%	37.14%	155.96%	-	484.64%
LL12-2p	132.96%	37.50%	168.33%	0.01%	525.20%
LL12-4p	99.81%	-	99.90%	70.68%	581.71%
LL12-8p	99.74%	-	50.12%	49.88%	349.41%
LL21-1p	75.00%	25.76%	55.88%	-	172.79%
LL21-2p	73.45%	26.66%	59.01%	-	175.79%
LL21-4p	86.46%	28.57%	59.31%	-	197.06%
LL21-8p	97.07%	-	0.98%	-	99.02%
jpeg-1p	0.32%	5.50%	5.75%	-	6.09%
jpeg-2p	0.58%	6.00%	6.04%	-	6.65%
jpeg-4p	1.24%	7.62%	7.49%	0.05%	8.87%
jpeg-8p	1.15%	7.25%	7.43%	3.22%	12.17%

TABLE I
IMPROVEMENT OVER BASE SYSTEM OBTAINED WITH DIFFERENT APPLICATIONS AND BANDWIDTHS

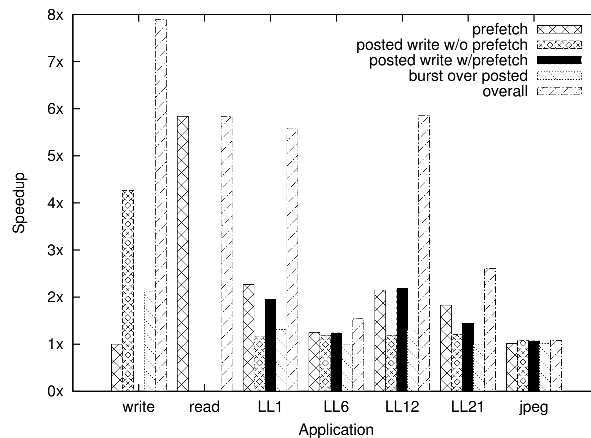

Fig. 15. Average speedup across all bandwidths obtained with the different techniques in all applications

VI. Conclusions

In this paper we have evaluated the benefits offered by optimizations at the level of protocol translations between the IPs and the network core in a NoC based system. We show how some optimizations can be performed transparently, hiding operation latency or improving bandwidth usage without breaking the view of the system from the IP side, and without requiring any intervention from the system designer or application writer. Other optimizations require explicit intervention from the programmer, but can achieve further gains.

We have tested our proposal on FPGA, using synthetic and real-life applications, and we observe in most cases significant speedups. Our implementation also demonstrates how inter-operation can be achieved transparently over the NoC between components using different bus standards, for example requests passed to the network through the FSL by MicroBlaze soft-core are served on the remote side by slaves on the PLB bus.

References

[1] M. Sgroi, M. Sheets, A. Mihal, K. Keutzer, S. Malik, J. Rabaey, and A. Sangiovanni-vincentelli, "Addressing the system-on-a-chip interconnect woes through communication-based design," in *DAC*, 2001.

[2] W. J. Dally and B. Towles, "Route packets, not wires: On-chip inteconnection networks," in *DAC*, 2001.

[3] B. Vermeulen and K. Goossens, "A network-on-chip monitoring infrastructure for communication-centric debug of embedded multi-processor socs," in *VLSI-DAT*, January 2009.

[4] K. Goossens, J. Dielissen, and A. Radulescu, "Æthereal network on chip: Concepts, architectures, and implementations," *IEEE Design & Test of Computers*, vol. 22, no. 5, 2005.

[5] E. Bolotin, I. Cidon, R. Ginosar, and A. Kolodny, "QNoC: QoS architecture and design process for network on chip," *Journal of Systems Architecture*, vol. 50, no. 2-3, pp. 105 – 128, 2004, special issue on Networks on Chip.

[6] D. Bertozzi and L. Benini, "Xpipes: a network-on-chip architecture for gigascale systems-on-chip," *Circuits and Systems Magazine*, vol. 4, no. 2, pp. 18–31, 2004.

[7] A. Pullini, F. Angiolini, P. Meloni, D. Atienza, S. Murali, L. Raffo, G. D. Micheli, and L. Benini, "NoC design and implementation in 65nm technology," in *NOCS*, 2007, pp. 273–282.

[8] J. Hu and R. Marculescu, "DyAD: Smart routing for networks-on-chip," in *DAC*. New York, NY, USA: ACM, 2004, pp. 260–263.

[9] J. Flich, A. Mejia, P. Lopez, and J. Duato, "Region-based routing: An efficient routing mechanism to tackle unreliable hardware in network on chips," *NOCS*, vol. 0, pp. 183–194, 2007.

[10] K. Srinivasan, K. S. Chatha, and G. Konjevod, "An automated technique for topology and route generation of application specific on-chip interconnection networks," in *ICCAD*, 2005, pp. 231–237.

[11] U. Y. Ogras and R. Marculescu, "Energy- and performance-driven NoC communication architecture synthesis using a decomposition approach," in *DATE*, 2005, pp. 352–357.

[12] A. Radulescu, J. Dielissen, S. G. Pestana, O. P. Gangwal, E. Rijpkema, P. Wielage, and K. Goossens, "An efficient on-chip network interface offering guaranteed services, shared-memory abstraction, and flexible network programming," *Transactions on CAD of Integrated Circuits and Systems*, vol. 24, no. 1, 2005.

[13] R. Holsmark, A. Johansson, and S. Kumar, "On connecting cores to packet switched on-chip networks: A case study with microblaze processor cores," in *DDECS Workshop*, April 2004, slovakia.

[14] P. Bhojwani and R. Mahapatra, "Interfacing cores with on-chip packet-switched networks," in *VLSID '03*, 2003, p. 382.

[15] S. P. Singh, S. Bhoj, D. Balasubramanian, T. Nagda, D. Bhatia, and P. T. Balsara, "Generic network interfaces for plug and play NoC based architecture." in *ARC*, vol. 3985. Springer, 2006, pp. 287–298.

[16] K. Skadron and D. W. Clark, "Design issues and tradeoffs for write buffers," in *HPCA*, 1997, pp. 144–155.

[17] T. fu Chen and J. loup Baer, "A performance study of software and hardware data prefetching schemes," in *ISCA*, 1994.

[18] T. LeBlanc and E. Markatos, "Shared memory vs. message passing in shared-memory multiprocessors," in *IPDPS*, Dec 1992, pp. 254–263.

[19] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, "CoMPSoC: A template for composable and predictable multi-processor system on chips," *ACM Trans. Des. Autom. El. Syst.*, vol. 14, no. 1, pp. 1–24, 2009.

[20] K. Goossens and A. Hansson, "The æthereal Network on Chip after ten years: Goals, Evolution, Lessons, and Future," in *DAC*, Jun. 2010.

[21] J. D. Day and H. Zimmermann, "The OSI reference model," pp. 38–44, 1995.

[22] A. Hansson and K. Goossens, "An on-chip interconnect and protocol stack for multiple communication paradigms and programming models," in *CODES-ISSS*, 2009.

[23] L. Lamport, "How to make a multiprocessor computer that correctly executes multiprocess progranm," *IEEE Trans. Comput.*, vol. 28, no. 9, pp. 690–691, 1979.

[24] S. V. Adve and K. Gharachorloo, "Shared memory consistency models: A tutorial," *Computer*, vol. 29, no. 12, pp. 66–76, 1996.

[25] M. D. Hill, "Multiprocessors should support simple Memory-Consistency models," *Computer*, vol. 31, no. 8, pp. 28–34, 1998.

[26] J. Goodman, "Cache consistency and sequential consistency," SCI Committee, Technical Report 61, Mar. 1989.

[27] M. Dubois, C. Scheurich, and F. Briggs, "Memory access buffering in multiprocessors," in *25 years of the international symposia on Computer architecture (selected papers)*. ACM, 1998, pp. 320–328.

[28] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy, "Memory consistency and event ordering in scalable shared-memory multiprocessors," in *Proceedings of the 17th annual international symposium on Computer Architecture*. ACM, 1990, pp. 15–26.

[29] S. W. Son, M. Kandemir, M. Karakoy, and D. Chakrabarti, "A compiler-directed data prefetching scheme for chip multiprocessors," in *Proceedings of the 14th ACM SIGPLAN symposium on Principles and practice of parallel programming*. Raleigh, NC, USA: ACM, 2009, pp. 209–218.

[30] F. H. McMahon, "The Livermore Fortran kernels: a computer test of the numerical performance range," 1986.