Low-Cost Guaranteed-Throughput Communication Ring for Real-Time Streaming MPSoCs

Berend H.J. Dekens, Philip Wilmanns, Marco J.G. Bekooij, Gerard J.M. Smit
University of Twente, Department of EEMCS
Enschede, The Netherlands

***Abstract*—Connection-oriented guaranteed-throughput mesh-based networks on chip have been proposed as a replacement for buses in real-time embedded multiprocessor systems such as software defined radios. Even with attractive features like throughput and latency guarantees they are not always used because their hardware cost tends to be higher than buses.**

In this paper we present a communication ring that provides throughput and latency guarantees. This ring is an attractive communication network as replacement for buses for small to medium scale embedded multiprocessor systems for real-time stream processing because of its relatively low hardware cost.

We show that the data serialization of our ring makes it contention free and enables sharing of buffers which reduces the hardware cost. A further cost reduction is achieved by implementing end-to-end flow-control in software and by supporting only writes over the network. Data-flow analysis techniques are used to prove that throughput and latency guarantees can be given despite that the proposed communication ring is connectionless.

We evaluated the performance and hardware cost of our communication ring using a 16 core multiprocessor system and a real-time PAL video decoder application. This design was implemented on a Virtex 6 FPGA and the ring was found to use roughly 2% of the logic cells used for the complete MPSoC design. Such a low hardware cost can justify the use of the ring in systems with low bandwidth utilization, as is the case for our PAL video decoder application which uses only 3% of the available bandwidth.

I. Introduction

An Multiple Processor System on Chip (MPSoC) for real-time processing is usually used for Software Defined Radio (SDR) applications. The network used with these designs should be able to provide guarantees at the application level.

In this paper we distinguish between *connection oriented* networks and *connectionless* networks. We define connection oriented as having separate connections between masters and slaves where properties can be specified for each individual connection [1]. In a connectionless network, communications are not separate and as such can influence each other which makes it hard to provide real-time guarantees [2]. Connection oriented networks tend to have dedicated buffers per connection at the edges of the network while in connectionless networks buffers at the edge of the network can be shared.

We distinguish two switching policies that are used in Networks on Chip (NoCs): circuit switched or packet switched [3]. The first policy is commonly associated with for example buses and cross bars. Routers in the interconnect are set up to provide a dedicated channel for communication between end points [4], [5]. While suitable for use in a real-time system, without detailed knowledge of communication within applications at design time, contention can cause the set up of a communication channel to fail.

The second and more actively studied [3] switching policy is packet switching. In these networks packets move from router to router based on routing information embedded into the data packet. While this type of network might share links for packets to different recipients, this sharing can also result in contention. As with circuit switched networks, without detailed knowledge of communication within applications at design time, it is impossible to know how many buffers will be required. Since buffering requires expensive memories, the amount of hardware for these buffers can become a considerable part of the total cost.

In this paper we present a low-cost ring interconnect as a replacement for buses for small to medium scale real-time multiprocessor systems. Despite that the ring is connectionless we show with a Synchronous Data Flow (SDF) model that useful throughput guarantees can be given at the application level. Guarantees can be given by reserving slots for each master. Reserved but empty slots can be claimed by other masters which makes the network work conserving. Under the assumption that slaves always accept data we show that only one FIFO buffer in the Network Interface (NI) is needed because this buffer can be shared between streams. Because buffers can be shared, the ring supports all-to-all communication. Furthermore, there is no need for FIFO buffers in the routers because no contention or head-off-line blocking can occur inside the ring network.

The organization of this paper is as follows. We discuss various networks and their topologies and relate them to our ring network in section II. In section III we present an architecture for stream processing where our ring interconnect provides all-to-all communication. We evaluate the resulting system by means of an application and evaluate the hardware in section IV. We will present the conclusions of our paper in section V.

II. Related Work

Multi-layer buses or buses in general are commonly used in industry. Buses are a form of circuit switched interconnects [6], [7]. Compared to buses, our ring interconnect uses

This work is supported in part by the SenSafety project in the Dutch COMMIT program (commit-nl.nl)

a small two port multiplexer per connection instead of an M-port multiplexer per slave to select between M masters. To support all-to-all communication, each multiplexer in a bus needs as many input ports as there are slaves, resulting in high resource usage and high wire count.

When a lot of bandwidth between masters and slaves is required and buses are no longer viable, a possible solution is to resort to using cross bars which are often fully connected, circuit switch matrices [8]. This topology is usually constructed from a large number of multiplexers and therefore can also become quite expensive in terms of number of wires, power and area. This will result in a much higher resource usage compared to our ring network.

SoCBUS [4] is a solution which tries to combine the low and fixed latency of a bus with the flexibility of a mesh network. Along with relatively high hardware usage, SoCBUS locks a dedicated route through its mesh topology for each connection making it circuit switched. This means that connections can fail to complete when all channels on a critical router are in use and as such there is always contention for capacity. Our ring provides automatic serialization due to its topology and as a result there is no contention within the network.

Another mesh based network with virtual channels is presented by Wolkotte et al. [5]. This network can be used both in packet switched and circuit switched mode. While the circuit switched version lowers hardware requirements because less buffering in routers is required, all packet decoding and flit routing still requires a significant amount of logic and buffers. In contrast, our ring network uses packets consisting of a single word and destination address which makes routing trivial and requires no address decoding.

Æthereal [9] is a packet switched network which also employs a mesh topology where all buffering for Guaranteed Throughput (GT) traffic is done in the NIs. When used for Best Effort (BE) traffic, buffering is needed in all routers. The network uses a pre-calculated Time Division Multiplex (TDM) schedule to provide GT where both bandwidth and/or latency can be guaranteed. The amount of buffering required for each connection in the NIs results in more hardware compared to our ring.

Smaller and faster than Æthereal is its successor called dAElite [10]. In this version the support for BE traffic is removed which results in less buffers in the routers. The NIs still contain a buffer per connection. We will show that despite being connectionless, our packet switched ring network can also support GT traffic. Unlike dAElite, our arbitration policy for the ring interconnect not only provides guarantees but makes it work conserving as well. This means that we can use the interconnect whenever it has capacity available instead of just using a fixed transmission schedule.

The use of a ring network as interconnect in a multi-core system is not new; it has been used in the Cell processor from IBM [11] and more recently by Intel in the Nehalem processor architecture [12]. The difference between our ring and existing implementations are the guarantees that our ring interconnect provides while keeping hardware costs low.

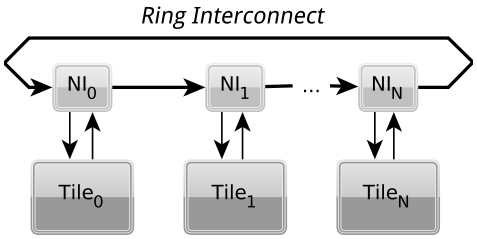


Fig. 1. Overview of our MPSoC architecture

However, while a small ring results in low latency communication, increasing the ring size also increases the maximum latency. Increasing the latency usually is less of an issue in stream processing applications as the use of pipelining can hide most if not all latency. In case of a large design, the number of nodes might become too large for a single ring network. A potentially interesting solution is to use a hierarchical topology to promote locality while being able to service a large number of peripherals [13].

In the next section we will present our ring in more detail.

III. Proposed Architecture

In this section we will present the architecture of our connectionless communication ring. It has a low hardware cost while still being able to provide the guarantees needed for real-time stream processing. In Figure 1 an overview of our architecture is depicted where multiple tiles are connected by the ring interconnect.

A. Ring Interconnect

For multiprocessor systems that are designed without precise knowledge of the applications which will run on these systems, it is often desirable that all masters can communicate with all slaves. However, the support of all-to-all communication can result in an expensive communication network if buffers at the edges cannot be shared between different data streams. In contrast, input and output buffers can be shared in our NIs, reducing the required amount of buffering and thereby the costs.

In order to reduce network complexity, we do not have support for back-pressure: whenever a packet enters the ring, it will travel one hop per cycle until it reaches its destination. As such we require that the receiving slave has to accept the packet in the same cycle it is delivered. This guaranteed acceptance also means we can determine the number of hops a packet will travel and therefore how much time is required to reach its destination. The concept of this guaranteed write acceptance is similar to what is required for many multi-layer buses.

Our ring interconnect is write-only: there is no support for remote reads. If remote data is required from a specific peripheral, this peripheral must act like a master and respond to a request by sending data.

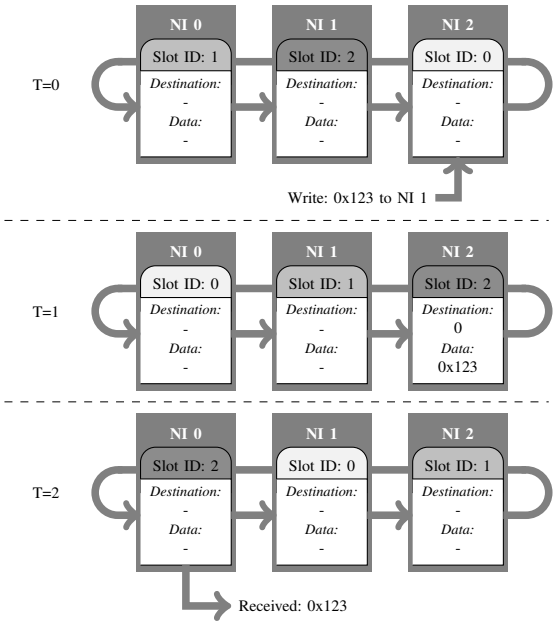


Fig. 2. Ring slot arbitration example for a word write

The ring is unidirectional which has various advantages over other alternatives. Routing decisions are trivial as packets on the ring cannot be stalled or deflected [13]. When a new packet arrives and the ring is available, a packet is injected into the ring. This is further explained in subsection III-B.

Unserialized multiple master to one slave communication usually requires memory port arbitration. As the topology of our ring interconnect provides automatic serialization, no memory port arbitration is required if a dedicated memory port is used such that data can be written at the same speed as the network can produce data.

B. Ring Slotting

To prevent deadlock and starvation we implement a bandwidth reservation algorithm. We call our algorithm "ring slotting": we consider the information contents of each NI a "slot". Each clock cycle, the content of all NIs is passed to their neighbors. This way, slots are cycled around the ring. By numbering these slots, we can uniquely identify each one.

By numbering the NIs themselves as well, we introduce the concept of "owned" slots: a slot which has an identifier which matches the NI it currently resides in, is owned by the NI. We now present the first rule of our arbitration:

Rule 1: *If a slot identifier matches the identifier of the NI it currently resides in, it is "owned" by that NI. NIs can always use their own slot to inject data onto the ring.*

Figure 2 shows an example of our slot based arbitration. In this example a write is stalled one cycle until its "own" slot comes up. The data is in transit the next cycle and delivered in the third cycle.

It is clear from Rule 1 that waiting time, i.e. the time between emitting the data to the input of the NI and the actual injection into the ring, is limited to $N-1$ where N is the number of peripherals on the ring. Similarly, as each N cycles one slot is available to the peripheral, the available bandwidth is exactly $\frac{1}{N}$.

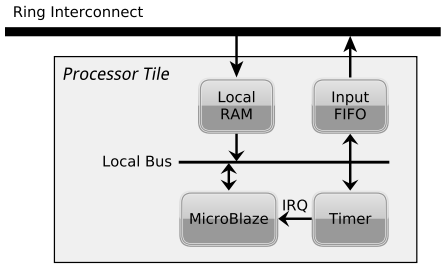


Fig. 3. Overview of a single processing tile

C. System Architecture

We will now introduce our system architecture which consists of multiple interconnected tiles.

Figure 3 shows an overview of a processing tile. We have multiple processing tiles connected to the ring. Each contains a RISC MicroBlaze CPU, timer for interrupts and local memory for instructions and data. The local data memory is dual ported and is connected to the output port of the NI in order to deliver data directly to it. All the memories are capable of single cycle reads or writes and as such adhere to the guaranteed acceptance required for the ring network. We expand the address to not only address data to a specific ring peripheral but also use some bits to denote a specific remote memory location at said peripheral.

See Figure 4 for a schematic overview of a single NI. Our ring network does not have distinct routers; by chaining a number of NIs together the ring topology is formed. The "Arbitration Control" block contains only combinatorial logic.

At the local input port of the network interface there is a small FIFO. This FIFO contains tuples consisting of a network address and a data word. The depth of the FIFO is configurable at design time. To prevent confusion with the distributed FIFO from subsection III-D we will refer to this hardware FIFO simply as "buffer" for the remainder of this paper. Whenever it is full, any further writes to the network will stall the CPU.

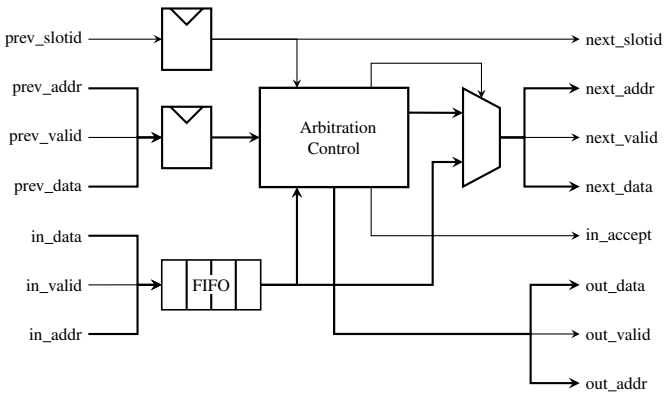


Fig. 4. Schematic overview of a single NI

Increasing the buffer depth will allow for larger write bursts without incurring stall cycles. However increasing the buffer depth will also increase the worst case waiting time as with a full buffer a single data word is dispatched every N cycles. Therefore, with a buffer of depth δ in the worst case a new word has to wait $N \cdot \delta$ cycles before it is dispatched.

The ring interconnect allows core to core communication and synchronization. We run applications from the local memories at each processing tile and stream data over the ring interconnect.

D. Distributed FIFO

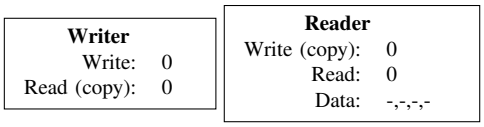
Despite only supporting remote writes our ring network can be used for a distributed FIFO implementation, based on C-HEAP [14]. This FIFO provides support for back-pressure at the application level using the ring interconnect. These FIFOs are used for the communication between parallel tasks in stream processing applications that are described by task graphs.

The FIFO uses data containers which are an arbitrary number of words in size and works by means of distributing read and write pointers. These pointers are used in a round-robin on the containers of the FIFO instance. At the FIFO input the local write pointer is compared with the read pointer to determine whether there is space to write data. When the pointers are the same, a special bit on both pointers, called a wrap flag [14], is used to distinguish between an empty and a full FIFO. This wrap flag is toggled every time a pointer rolls over: when the pointers and their wrap flags are identical the FIFO is empty, otherwise its full. The C-HEAP algorithm does not require locks, mutexes or atomic read-modify-write support.

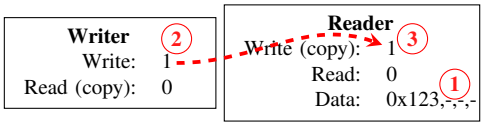
The read pointer at the FIFO *writer* is a copy of the read pointer from the output of the FIFO: it is only updated by the "remote" peripheral. See Figure 5b for an example for writing a container into a FIFO between two cores. Whenever data is written, it is sent to the allocated FIFO memory at the remote peripheral (1). After the data is written, the write pointer is updated (2) and the update is written to the remote peripheral as well (3). Note that posted writes to a remote memory are always guaranteed to be received in the same order; as such the pointers are always modified after the data is written. Our hardware supports streaming memory consistency [15] which guarantees that reads and writes from a specific processor arrive in the same order that they were issued.

At the *reader* of the FIFO the copy of the write pointer is compared to the read pointer to determine if data is available. If this is the case (the read and write pointer differ), the data is already stored in the local data memory of the peripheral. As such, reads from the FIFO are always from local memory which also means every read is performed in a single cycle.

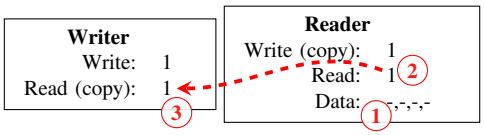
See Figure 5c for an example for reading a container from a FIFO. After a data container is consumed from the FIFO (1), the read pointer is updated to reflect this (2). The read pointer is then written to the memory of the remote peripheral (3) to update the state of the FIFO.



(a) Empty FIFO



(b) Write data to remote memory, update local write pointer and update remote write pointer



(c) Read from local memory, update local read pointer and update remote read pointer

Fig. 5. Examples for a distributed FIFO with capacity 3

E. Throughput Analysis Model

As explained we can use the write-only scheme from the interconnect for a software FIFO channel where the read and write pointers are placed in the memories of both the producer and consumer. This is similar to a credit based handshake system which can be modeled as an SDF graph [16].

An SDF model is a directed graph $G_S(E, V)$ consisting of actors $v \in V$ and directed edges $e \in E$. Each edge describes a directed connection between two actors: $e = (v_i, v_j)$. The edges represent an unbounded FIFO queue to store tokens.

Actors have a firing rule: when at least a specific number of tokens, called quanta, at the input queue or queues of an actor are available, the actor fires. This means that the required input tokens are consumed instantly and after the execution time of that actor has elapsed a specified number of tokens are instantly produced. When enough input tokens are available, an actor can fire multiple times simultaneously. To prevent parallel execution where this is not desired, we can use self edges with one token to prevent overlapping firings of actors.

We model our software FIFO with capacity α as an SDF graph as is depicted in Figure 6. This graph is an abstract model describing both hard- and software where an actor does not need to describe a single hardware or software component. In this model we have a producing actor P and consuming actor C with their corresponding execution times: ρ_P and ρ_C. The sharing of a small hardware input buffer, as depicted in Figure 4, with capacity δ by multiple tasks can be modeled by incorporating the waiting time into the execution time. We use the same δ for all NIs in the rest of this paper. Each write can cause a wait time β of at most $\beta = \delta \cdot N$ cycles if the hardware input buffer is full. Our operating system uses a TDM scheduler with time slice length ξ and as such, a single execution of a task will be preempted $\lceil \frac{\rho}{\xi} \rceil$ times. In the worst case the input buffer is full each time our task starts or resumes. Therefore, the total maximum wait time φ during

the write of a complete container is:

$$\varphi = \beta \cdot \lceil \frac{\rho}{\xi} \rceil = \delta \cdot N \cdot \lceil \frac{\rho}{\xi} \rceil \tag{1}$$

We now incorporate this wait time into the original execution time: $\rho + \varphi = \hat{\rho}$.

As φ directly influences the execution time of the producer and consumer in this model, it should be kept small. This can be done by increasing the length of time slice ξ or decreasing capacity δ. Reducing buffer capacity δ lowers φ but potentially increases the occurrence of processor stall cycles.

Latency from the network is based on the number of hops ω that a word travels from the master to the slave with a maximum of $N-1$. The total transit time of a single write is the sum of the waiting time β and the number of hops ω:

$$L = \beta + \omega = \delta N + \omega \tag{2}$$

The communication between actors P and C is rate limited to $\frac{1}{N}$ which is modeled by actor R_D. The number of hops needed to get from P to C is defined as K. We use two separate actors to model latency and rate between P and C where we need to subtract the execution time of the rate limiter from the execution time of the latency actor. The latency actor L_D from P to C has execution time:

$$L_D = (\delta N + \omega) - N = \delta N + K - N \tag{3}$$

After S tokens have been received and consumed by the consumer, a credit token can be sent back to the producer. The communication between C and P is again rate limited to $\frac{1}{N}$ which is modeled by actor R_C. As the number of hops between P and C was K hops, the credit token will travel $N-K$ hops from C to P as the total number of hops from P to C and back will always be N. Together with waiting time β we can model the total credit token latency as:

$$L_C = \beta + (N - K) - N = \delta N - K \tag{4}$$

We will now give the definition of the Cycle Mean (CM) of an Homogeneous Synchronous Data Flow (HSDF) graph [17]. Let G_H denote an HSDF graph and let C denote a cycle in G_H. The weight $w(C)$ of C is defined as the sum of all execution times of the actors on cycle C. The number of tokens on cycle C is called τ. The mean between the execution times of the actors on the cycle and the number of tokens on the edges of it is the CM:

$$\lambda(C) = \frac{w(C)}{\tau} \tag{5}$$

The CM of C gives a weighed average rate between the execution times of all actors in that cycle and the number of tokens on that cycle. This average also describes the throughput of that individual cycle. The Maximum Cycle Mean (MCM) of G_H is defined as:

$$\lambda^* = \max_{\forall C \in G} \{\lambda(C)\} \tag{6}$$

The throughput of an HSDF graph is the inverse of its MCM. In order to calculate the MCM for the SDF graph

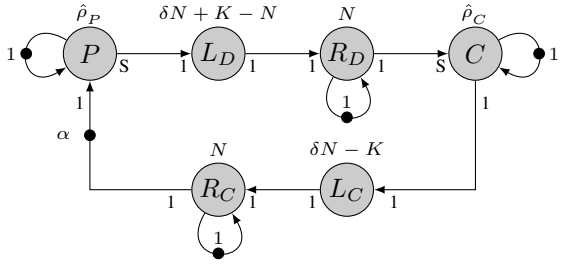


Fig. 6. SDF graph for a task with core-to-core communication

from Figure 6 we transform it into an HSDF graph [18], [19], [20]. In the transformation to an HSDF graph, the actors and edges between P and C are duplicated to S parallel paths with quanta 1. The self-cycle on v_{R_D} becomes a single cycle c_1 across all copies of v_{R_D} with a single token on it:

$$c_1 = ((v_{R_{D_1}}, v_{R_{D_2}}), ..., (v_{R_{D_S}}, v_{R_{D_1}})) \tag{7}$$

The CM of all cycles from the transformed SDF graph from Figure 6 can now be determined. We define the cycle c_2 as the sending of a complete container from actor P to C and the credit from C to P. This simple cycle is denoted by:

$$c_2 = ((v_P, v_{L_{D_1}}), (v_{L_{D_1}}, v_{R_{D_1}}), ..., (v_{R_C}, v_P)) \tag{8}$$

We denote the CM of cycle c_1 as CM(c_1):

$$\text{CM}(c_1) = \frac{SN}{1} = SN \tag{9}$$

Similarly we derive the CM of cycle c_2:

$$\begin{aligned} \text{CM}(c_2) &= \\ &\frac{\hat{\rho}_P + \delta N + K - N + SN + \hat{\rho}_C + \delta N - K + N}{\alpha} \\ &= \frac{\hat{\rho}_P + \hat{\rho}_C + 2\delta N + SN}{\alpha} \\ &= \frac{\rho_P + \varphi + \rho_C + \varphi + 2\delta N + SN}{\alpha} \\ &= \frac{\rho_P + \rho_C + 2\delta N \lceil \frac{\rho}{\xi} \rceil + 2\delta N + SN}{\alpha} \end{aligned} \tag{10}$$

We can now consider the throughput of the entire HSDF graph by deriving the MCM as is shown in Equation 11.

$$\lambda^* = \max_{\forall C \in G} (\text{CM}(c_1), \text{CM}(c_2), \hat{\rho}_P, \hat{\rho}_C, N) \tag{11}$$

From Equation 11 we can see that as long as CM(c_2) is the largest component in λ^*, increasing the FIFO capacity α directly lowers CM(c_2) and thus the MCM. We can also see that when α is sufficiently large, it is no longer the largest component in λ^* and as such no longer influences the MCM. At this point we can say that the network latency no longer influences application throughput as the execution time of actors P or C or the rate limiters R_D and R_C will limit throughput.

Note that increasing the capacity α of the FIFO placed at C increases the initial number of tokens on the edge between actor R_C and P but it does not influence the execution time of

any of the actors. This means that the latency resulting from the ring interconnect and the FIFO algorithm does not change by altering the FIFO instance capacity.

For most stream processing applications latency requirements are usually less tight than throughput and as such using buffers with sufficient capacity can often be used to compensate for latency.

F. Work Conserving

The use of unique slots on the ring interconnect provides strict latency and bandwidth guarantees required for real-time applications. However just like most GT networks it is not work conserving as unused slots remain empty which we will now change with a small modification to our arbitration policy.

Since we know when a slot reaches the rightful owner, who may or may not use the slot, we can use an empty slot to address peripherals up to and including the owner of the slot. This would allow the use of otherwise wasted bandwidth without interfering with the GT packets and thus without interfering with all guarantees. We now define the second rule in our arbitration policy:

Rule 2: *If a NI is ready to send data, the current slot is empty and the owner of the slot is not reached before the destination NI is reached, data can be injected into that slot.*

This small addition adds a little bit of hardware to the injection logic but makes the ring interconnect work conserving. By making more slots available we increase the upper bounds of the bandwidth based on the distance a packet has to travel on the ring.

We define the GT bandwidth as $\gamma = \frac{1}{N}$. We use M_{hops} to denote the number of hops a packet has to travel between peripherals. As we are not allowed to use any of the slots belonging to NIs between the source and destination peripheral, we can potentially use $N - M_{hops}$ slots out of N slots. We can now derive the upper bound on the available bandwidth, called $\hat{\gamma}$ for traffic between two NIs:

$$\hat{\gamma} \quad = \quad \gamma \; + \; \frac{N - M_{hops}}{N} \quad = \quad \frac{N - M_{hops} + 1}{N} \qquad (12)$$

We can see from Equation 12 that when the distance data has to travel on the ring increases, the upper bandwidth bound decreases proportionally.

G. External Memory

As long as the stream processing can exploit data locality the use of local memories suffices while containers are relatively small. When this is not the case, larger memory buffers might be required. An example of this is a video deinterlacer which requires the buffering of at least half a frame. For 32-bit color at VGA resolution this already requires more than 600 kB.

To this end, we share the external memory between all cores using a latency bounded tree shaped interconnect [21]. By adding instruction and data caches to the CPUs, we can lower the contention for the external memory. We have used this memory for the video deinterlacer task and the DVI controller frame buffer, as explained in section IV.

In theory this memory can be used to store data for our FIFO channels. However, experiments with our test application showed that this communication method can reduce throughput by a factor 2, as discussed in more detail in section IV.

In the next section we will describe how we mapped a streaming video decoder application onto a 16 core system with our ring interconnect in order to evaluate our design.

	Complete Design		Ring		Ratio	
# CPUs	Sl.Regs:	LUTs	Sl.Regs:	LUTs	Sl.Regs:	LUTs
2	5746	7905	115	119	2.0%	1.5%
4	11492	15810	237	241	2.1%	1.5%
8	22984	31620	481	452	2.1%	1.4%
16	45968	63240	977	1006	2.1%	1.6%
32	91936	126480	1985	1975	2.2%	1.6%

TABLE I
LOGIC USAGE ON A VIRTEX-6 FPGA SHOWING A LINEAR SCALE IN RESOURCE USAGE FOR OUR RING INTERCONNECT.

IV. EVALUATION

In this section we will describe the used hardware instance for evaluation and mapping of a video decoding application.

We will evaluate the hardware usage for the ring interconnect compared to the rest of the system and the suitability of the interconnect for the implemented video decoding application. We will also look into the utilization of the bandwidth provided by the ring. We end this section by comparing the ASIC synthesis results with another interconnect.

A. Hardware Costs

We implemented our ring interconnect for an MPSoC with 16 CPU cores on a Xilinx ML-605 prototyping board.

See Table I for more details on hardware usage. These results indicate that the hardware usage of our ring scales linearly with the number of CPUs. The minor increase in hardware usage is due to the increasing width of the address bus for addresses on the ring itself when the number of NIs increase. We can see that the ring interconnect accounts for only $\sim$2% of the total hardware costs which is small compared to other components in the design.

B. Case Study: PAL Video Decoder

In order to evaluate the performance of the proposed architecture we implemented a Phase Alternating Line (PAL) decoder in software on a 16 core design. The PAL standard describes a field interlaced, 25 frames per second color video signal, usually in combination with FM modulated audio [22]. The signal consists of a Amplitude Modulation (AM) luminance signal (Y) and a quadrature modulated color difference signal (R-Y and B-Y) at 4.43 MHz from the luminance carrier. During decoding, the color signal has to be removed from the original signal before the luminance can be extracted.

In our demonstrator we only process the luminance and ignore the color signal. The luminance signal contains 625 lines per frame, two fields per frame and 25 frames per second.

As such, the vertical resolution is fixed to the number of lines whereas the horizontal resolution is directly related to the sample rate. By sampling the signal at a lower speed and using the appropriate filters, we eliminated interference from the color signals. The horizontal resolution is not high enough at this sample rate to produce the native 4:3 aspect ratio from the PAL standard and as such has to be upscaled to obtain the correct aspect ratio.

Every line is separated by a sync pulse which uses a lower signal value than the rest of the inverted luminance signal, as shown in Figure 7. It is customary to normalize the input format so that the luminance ranges from 0.0 to 1.0 where 0.3 and lower are used for line syncs. Each frame consists of two interleaved fields. All fields are separated by a number of special synchronization pulses.

Our parallel decoding algorithm is depicted as an SDF graph in Figure 8. The actors in this SDF graph correspond with tasks where each task is using a dedicated CPU. After acquiring I/Q samples at baseband, the AM signal is reconstructed from the magnitude of the I/Q pairs. The resulting intensity of the luminance signal is recorded and used in the field synchronization detection. At the beginning of a field, the field type is detected: one field type is used for even lines, the other for odd lines. At this point, the signal is normalized to obtain a range suitable for displaying. The field alignment is used to calculate the start of each line and the deinterlacer generates complete 2D frames. When a frame is complete, a 1D scaler is used to scale the horizontal resolution to obtain the 4:3 display ratio. A resulting test picture is shown in Figure 9 which corresponds with the input signal from Figure 7.

Where the detection of field synchronization is required, a state machine is used to detect the edges in the luminance signal in combination with the signal levels. A similar state machine is used to detect the field type right after a field sync was found. These two tasks contain state machines and as such execute sequentially and cannot be duplicated to benefit from parallelism. Other, arithmetic intensive tasks like AM demodulation, Automatic Gain Control and Upscaling can work on small chunks of data individually and as such are duplicated to improve throughput. The bottleneck of the application are the sync and type detect tasks which each use a dedicated CPU.

This algorithm uses 16 cores at 100 MHz and processes PAL video real-time at 3 MS/s or 12 MB/s which corresponds to a horizontal resolution of 192 pixels[1].

[1]Commercial grade PAL decoders require a sample rate of at least 12 MS/s which would result in a horizontal resolution of 768 pixels

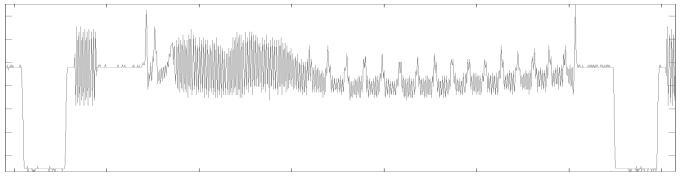


Fig. 7. Complete PAL line with low valued synchronization pulses

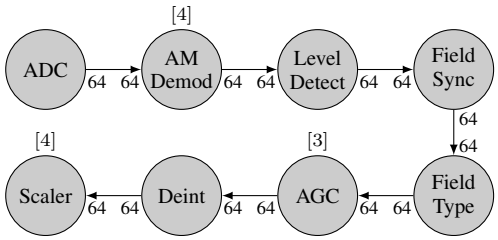


Fig. 8. Data flow graph for PAL decoding application. Duplicated tasks for data level parallelism are denoted by the numbers in brackets.

As we use four byte wide data words and a NI transfers a complete word per clock tick, the total bandwidth between two NIs is $4 \cdot 100 = 400$ MB/s. The structure from Figure 8 is a processing pipeline and by using this structure as a mapping to our platform, we can derive that each connection between two NIs on the ring will transfer 12 MB/s. This is a mere 3% of the total available bandwidth; commercial grade PAL would require 12% load per connection.

When we consider the longest distance data streams have to travel through the ring network, we see that data travels four hops down the ring. By using Equation 12 we find the best case bandwidth of $\frac{13}{16} = 81.25\%$ of 400 MB/s. Our minimum guaranteed bandwidth is $\frac{1}{16} = 6.25\%$, which is sufficient for our application. We conclude that the ring capacity could be halved for this particular application but this is of little value as the ring occupies only 2% of the complete design.

Our ring communication is handled by a library to abstract various system aspects. This library can also be used to perform FIFO communication using the external SDRAM memory. However, if all synchronization and data are placed in external memory, we would have 16 cores contending for access. Results showed that the PAL decoder performs more than a factor two slower when the ring network is not used.

C. Synthesis Results and Power Estimates for ASICs

As stated before, the structure of the ring results in low hardware costs. While we demonstrated this by presenting the hardware resource usage for a Virtex-6 FPGA, we also synthesized our ring interconnect to a standard cell implementation

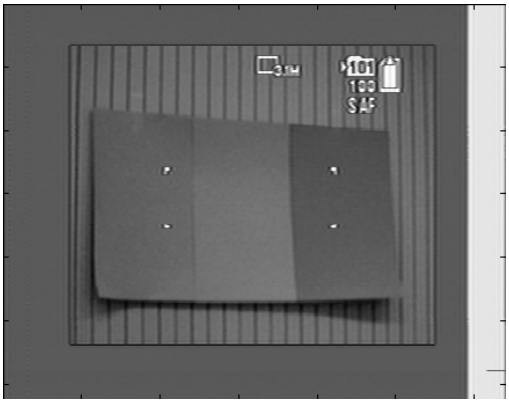


Fig. 9. Complete deinterlaced PAL video frame

Voltage	Area	Speed	3% Load	100% Load
V	μm^2	GHz	mW	mW
1.0 GP[3]	1100	2.0	1.24 (5%)	1.60 (6%)
1.2 LP[4]	1107	1.0	0.79 (0.09%)	1.00 (0.07%)

TABLE II
POST-SYNTHESIS RESULTS OF 65 NM ST IMPLEMENTATION IN ASIC LOGIC, LEAKAGE IS SPECIFIED AFTER POWER.

used to describe ASIC logic. We used the Synopsys synthesis tooling using the low power and low leakage 65 nm library from STmicroelectronics.

Synthesis results indicate that for every NI 1100 μm^2 of silicon is required, see Table II. While wiring and the small input buffer are not included, we can compare this rough estimate to specified area of routers of other interconnects. Compared to the latest version of Æthereal light, our NI[2] is 20 times smaller than their smallest GS router (0.07 mm^2) [9].

V. CONCLUSION

In this paper we presented a low cost communication ring for real-time multi-core stream processing architectures as used in the SDR domain. Our ring interconnect realizes low hardware cost by sharing buffers within a NI. This is possible because the ring provides automatic serialization and requires guaranteed acceptance at the slaves. Despite being connectionless, our ring provides bandwidth and latency guarantees by reserving slots for each master. The work conserving scheduling policy of the ring interconnect allows tasks to use the slack of other tasks which can improve the average throughput at the application level.

We demonstrated that the write-only property of the ring interconnect is sufficient to implement a FIFO with split administration. We use the guaranteed ordering of writes for our FIFO implementation to provide application level back-pressure.

We derived an SDF model for the communication channel between a master and slave which can be used to determine the required capacity in order to reduce the effects of network latency on application throughput.

We implemented our design of the ring interconnect in an MPSoC on an FPGA and in ASIC logic. On the FPGA we found that our interconnect scales linearly and occupies 2% of the resources used in the design.

We evaluated our design by means of a PAL video decoder application, using 16 cores in parallel. The low hardware cost of the ring can justify the use in systems with low bandwidth utilization, as is the case for our PAL video decoder application which uses only 3% of the available bandwidth. The presented results indicate that the described communication ring interconnect is an attractive candidate for medium scale multiprocessor systems for real-time stream processing applications.

[2]Including all logic and registers, without the optional small input buffer
[3]General Purpose Logic, Low Voltage: 1.00V, 25°C
[4]Low Power Logic, Nominal Voltage: 1.20V, 25°C

REFERENCES

[1] E. Rijpkema, K. Goossens, A. Radulescu, J. Dielissen, J. van Meerbergen, P. Wielage, and E. Waterlander, "Trade offs in the design of a router with both guaranteed and best-effort services for networks on chip," in *Design, Automation & Test in Europe*, no. c. IEEE Comput. Soc, 2003, pp. 350–355.

[2] M. Harmanci, N. Escudero, Y. Leblebici, and P. Ienne, "Providing QoS to connection-less packet-switched NoC by implementing diffserv functionalities," in *International Symposium on System-on-Chip*, 2004, pp. 37–40.

[3] E. Salminen, A. Kulmala, and T. D. Hamalainen, "Survey of Network-on-chip Proposals," *White Paper, OCP-IP*, no. Mar., pp. 1–13, 2008.

[4] D. Liu, D. Wiklund, and E. Svensson, "SoCBUS: The solution of high communication bandwidth on chip and short TTM," in *Real-Time and Embedded Computing Conference*, 2002.

[5] P. Wolkotte, G. Smit, G. Rauwerda, and L. Smit, "An Energy-Efficient Reconfigurable Circuit-Switched Network-on-Chip," in *IEEE International Parallel and Distributed Processing Symposium*, vol. 19, no. c. Denver, Colorado, USA: IEEE, 2005, pp. 155a–155a.

[6] IBM, *128-Bit Processor Local Bus Architecture Specifications - Version 4.7*, 2007.

[7] ARM, *AMBA AXI and ACE Protocol Specification*, 2011.

[8] A. G. C. Koppelaar, A. Burchard, and W. Tang, "Concurrent Viterbi decoding for dual-channel ITS communication on a SDR platform," in *WIC Symposium on Information Theory in the Benelux*, vol. 33, Boekelo, 2012, pp. 149–156.

[9] K. Goossens and A. Hansson, "The Aethereal network on chip after ten years: Goals, evolution, lessons, and future," in *Design Automation Conference*, Anaheim, California, USA, 2010, pp. 306–311.

[10] R. Stefan and A. Molnos, "A TDM NoC supporting QoS, multicast, and fast connection set-up," in *Design, Automation & Test in Europe*, Dresden, Germany, 2012, pp. 1283–1288.

[11] M. Kistler, M. Perrone, and F. Petrini, "Cell Multiprocessor Communication Network: Built for Speed," *IEEE Micro*, vol. 26, no. 3, pp. 10–23, May 2006.

[12] S. Kottapalli and J. Baxter, "Nehalem-EX CPU Architecture," *Hot chips*, pp. 1–19, 2009.

[13] C. Fallin, X. Yu, G. Nazario, and O. Mutlu, "A high-performance hierarchical ring on-chip interconnect with low-cost routers," Computer Architecture Lab (CALCM), Carnegie Mellon University, Tech. Rep. SAFARI No. 2011-007, 2011.

[14] A. Nieuwland, J. Kang, O. P. Gangwal, R. Sethuraman, N. Busá, K. Goossens, R. P. Llopis, and P. Lippens, "C-heap: A heterogeneous multi-processor architecture template and scalable and flexible protocol for the design of embedded signal processing systems," *Design Automation for Embedded Systems*, vol. 7, no. 3, pp. 233–270, 2002.

[15] J. W. Van den Brand and M. Bekooij, "Streaming consistency: a model for efficient mpsoc design," in *EUROMICRO Conference on Digital System Design*, vol. 10, 2007, pp. 27–34.

[16] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, "Enabling application-level performance guarantees in network-based systems on chip by applying dataflow analysis," *IET Computers & Digital Techniques*, vol. 3, no. 5, p. 398, 2009.

[17] A. Dasdan and R. Gupta, "Faster maximum and minimum mean cycle algorithms for system-performance analysis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 17, no. 10, pp. 889–899, 1998.

[18] E. Lee and D. Messerschmitt, "Synchronous data flow," *Proceedings of the IEEE*, vol. 75, no. 9, pp. 1235–1245, 1987.

[19] S. Sriram and S. S. Bhattacharyya, *Embedded Multiprocessors: Scheduling and Synchronization*. CRC Press, 2000, vol. 3.

[20] R. de Groote, J. Kuper, H. Broersma, and G. J. M. Smit, "Max-plus algebraic throughput analysis of synchronous dataflow graphs," in *EUROMICRO Conference on Software Engineering and Advanced Applications*, vol. 38, 2012, pp. 29–38.

[21] J. H. Rutgers, M. J. Bekooij, and G. J. Smit, "Evaluation of a Connectionless NoC for a Real-Time Distributed Shared Memory Many-Core System," in *EUROMICRO Conference on Digital System Design*, vol. 15. IEEE, Sep. 2012, pp. 727–730.

[22] ITU-R, "Rec. ITU-R BT.470-6: Conventional Television Systems," Tech. Rep., 1998.