System-Level Analysis of Network Interfaces for Hierarchical MPSoCs

Johannes Ax*, Gregor Sievers*, Martin Flasskamp*, Wayne Kelly†, Thorsten Jungeblut*, and Mario Porrmann*

*Cognitronics and Sensor Systems Group, CITEC, Bielefeld University, Bielefeld, Germany
jax@cit-ec.uni-bielefeld.de

†Science and Engineering Faculty Queensland University of Technology Brisbane, Australia
w.kelly@qut.edu.au

ABSTRACT

Network Interfaces (NIs) are used in Multiprocessor System-on-Chips (MPSoCs) to connect CPUs to a packet switched Network-on-Chip. In this work we introduce a new NI architecture for our hierarchical CoreVA-MPSoC. The CoreVA-MPSoC targets streaming applications in embedded systems. The main contribution of this paper is a system-level analysis of different NI configurations, considering both software and hardware costs for NoC communication. Different configurations of the NI are compared using a benchmark suite of 10 streaming applications. The best performing NI configuration shows an average speedup of 20 for a CoreVA-MPSoC with 32 CPUs compared to a single CPU. Furthermore, we present physical implementation results using a 28 nm FD-SOI standard cell technology. A hierarchical MPSoC with 8 CPU clusters and 4 CPUs in each cluster running at 800 MHz requires an area of 4.56 mm^2.

Categories and Subject Descriptors

B.4.3 [**Interconnections (Subsystems)**]: Interfaces; C.1.4 [**Parallel Architectures**]

1. INTRODUCTION

The integration of multiple CPUs on a single chip is a common approach to satisfy the increasing performance and energy efficiency requirements of embedded systems. This requires an efficient on-chip communication infrastructure for MPSoCs with dozens to hundreds of CPU cores. To cope with limitations of the scaling of bus-based communication, state of the art multiprocessors introduce dedicated communication networks, i.e. Network-on-Chips (NoCs). NoCs usually (but not solely) are based on explicit communication. This not only affects the hardware but also requires additional work to be done in software to initiate end-to-end communication. The CoreVA-MPSoC used in this work features a hierarchical architecture and targets streaming applications in embedded and energy-limited systems. In [10] we have presented the basic architectural concept of this hierarchical system. We have shown that it is reasonable to tightly couple several CPUs in a cluster and to couple several clusters via a NoC.

A NoC is composed of three components: The (i) routers transport the data through the NoC in a packet-based manner. Routers are connected via (ii) network links. A (iii) network interface (NI) implements the interface between the routers and the CPUs. In this work we focus on the design-space-exploration of the NI of our NoC. It bridges between the read-write transaction interface used by the CPUs and the packet streaming interface used by the routers in the network. Therefore, NIs have a great impact on system performance.

The main contribution of this work is the system-level analysis of both software and hardware costs of different NI configurations. Particularly the CPU's software costs for communication using NoCs is rarely considered in related work. The NI presented in this work is based on a novel architecture, which supports sending requests from several CPUs in parallel. To reduce the CPU load the NI acts like a DMA controller with several independent transaction channels. The partitioning of applications to hundreds of CPUs is a challenging task, which cannot be done manually. Therefore, several approaches for the automatic partitioning of the applications have been introduced. For our work we use a compiler for streaming applications presented in [6]. To compare the performance of different configurations of the NI we use a set of streaming applications. In addition, physical implementation results are evaluated using a 28 nm FD-SOI standard cell technology.

2. RELATED WORK

State of the art NIs can generally be separated into two different architectural approaches. One approach is the usage of a packet–based streaming interface to the routers of the NoC. This approach is commonly used in classic network approaches and can be seen as the state of the art in NoC research. The NI bridges between the address-based interface used by the CPUs and the packet-streaming interface used by the NoC. Examples are the NIs of the AEthereal NoC [8] and the Spidergon-STNoC [9]. Packet headers and payload data are directly stored in FIFOs or ring-buffers within the NI. The NI typically segments the packets into atomic units,

NoCArc '15, December 05 2015, Waikiki, HI, USA

ISBN 978-1-4503-3963-6/15/12...$15.00

DOI: http://dx.doi.org/10.1145/2835512.2835513

called flits. Flits of the same packet can be identified by the NoC flow control at the receiver. Packet transfers increase the throughput of payload data via the NoC. For allowing CPUs random memory access on payload data, the data needs to be copied from the NI to their local memories and vice versa. This leads to CPU runtime costs in software or requires special hardware components like DMA Controllers. Analyses in [7] have shown, that the usage of hardware support by DMA controllers in combination with scratch-pad memories outperforms a cache-based mechanism in NoCs. The NI presented in [12] integrates a DMA controller by storing packet information (e.g. routing coordinates and write-, read-pointer, and size of the data) in a DMA table and supports several DMA channels. Another NI with two independent DMA channels is presented in the STM STHORM MPSoC [2]. This MPSoC has a hierarchical architecture where several CPUs are grouped in a CPU cluster and share one NI. An alternative approach for NI architectures employs a common global address-space for the whole MPSoC. Adapteva's Epiphany [1] is an example for this approach. On the one hand a global address-space allows CPUs to directly and randomly access all memories of the MPSoC—A memory access across the NoC can be realized with a single CPU memory operation via the NoC. On the other hand a DMA Controller can assist the CPUs with block transfers. The NI transfers a single flit that contains address and payload data to the NoC routers. In general this decreases the NoC's throughput of pure payload data.

Kalray's MPPA-256 [3] does not provide a global address-space on NoC level but uses a shared address-space within each CPU cluster. Thus thread communication between different CPUs is limited to a cluster. CPUs of different clusters communicate via inter-process communication across the NoC. A combination of both approaches is used by commercial NoC vendors like Arteris (FlexNoC), Sonics (SonicsGN) or NetSpeed (Orion) [4]. Incoming transactions (e.g., from a conventional AXI interface) are converted into packets, transfered via the NoC, and converted back to AXI. There is no detailed information available regarding the architecture of these commercial NoCs.

Our CoreVA-MPSoC employs a combination of both approaches as well. The NoC utilizes a packet transfer, while the NI supports a memory-based DMA functionality to minimize CPU load for communication. To allow for a scalable and efficient MPSoC the NI supports a flexible management of independent channels at runtime.

3. THE COREVA-MPSOC

This chapter presents the hardware and software architecture of the CoreVA-MPSoC (cf. Figure 1).

3.1 Architecture

The CPU contained in our MPSoC is named CoreVA [11] and features a configurable 32 bit VLIW architecture. It has separate instruction and data memories and six pipeline stages. The number of VLIW issue slots, arithmetic-logic-units (ALUs), multiply-accumulate (MAC), and load-store-units (LD/ST) can be adjusted at design time. To avoid CPU stalls due to bus congestion, a FIFO is used to decouple CPU bus writes. Within a cluster several CoreVA CPUs are tightly coupled via an interconnect fabric [10]. The cluster implements a NUMA (Non-Uniform Memory Access) architecture, where each CPU can access the L1 data memories

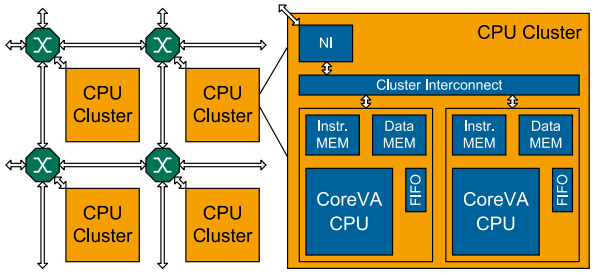


Figure 1: The hierarchical CoreVA-MPSoC.

of all other CPUs within a cluster. Bus standard (AMBA AXI4, Wishbone) and topology (shared bus, partial or full crossbar) are both configurable at design time. In this work the cluster interconnect is fixed to a full AXI crossbar and a data bus width of 64 bit.

For realizing MPSoCs with dozens or hundreds of CPU cores, a second interconnection hierarchy level, a Network on Chip (NoC), is introduced to the CoreVA-MPSoC. Our NoC implementation features packet switching and wormhole routing. Each packet is segmented into small flits, each containing a header for control information and 64 bit payload data. The NoC interconnect is built up of routers, each having a configurable number of ports. This flexibility permits the implementation of most common network topologies. In this work, a 2D-mesh topology is used (cf. Figure 1) and a router has a latency of two clock cycles. One port of each router is connected to a cluster via a network interface (NI, cf. Section 4). For very large scaled CoreVA-MPSoCs the NoC can be extended by a Globally-Asynchronous Locally-Synchronous (GALS)-based approach by using mesochronous links [5]. The NoCs considered in this work are small enough to not require GALS.

3.2 Communication Model

The CoreVA-MPSoC platform particularly targets streaming applications like signal and video processing. A typical streaming application is build up of many different tasks which are connected via a directed data flow graph. To allow communication between the tasks executed on different CPUs, an efficient communication model is required. Within the CoreVA-MPSoC we use a communication model with unidirectional communication channels. This approach is more scalable and efficient compared to shared memory concepts where the memory access can become the bottleneck [7].

In general a task will read from one or more input channels and write to one or more output channels. Each channel manages one or more read/write data buffers. The application can request a buffer to read from (getReadBuf) and a buffer to write to (getWriteBuf). A channel manages synchronization at the granularity of the buffer size. The operation of requesting a buffer from a channel blocks the CPU if sufficient data has not yet arrived or if there is no free buffer available to write to. However, once the application receives a buffer by the channel, it is free to read or write (as appropriate) any memory location within that buffer without need for any further synchronization. When the application has finished writing to or reading from a buffer, it must inform the channel so that the buffer can be read by the reader (setWriteBuf) or reused by the writer (setReadBuf).

There are three different types of channels: A (i) Memo-

ryChannel for communicating between tasks mapped to the same CPU. The communication between CPUs of the same cluster is handled by a (ii) ClusterChannel. A (iii) NoCChannel allows for communication between CPUs of different clusters via the NoC. Inter-CPU channels (ClusterChannel and NoCChannel) maintain at least two separate buffers so that latency can be hidden (double buffering).

In case of the ClusterChannel, the data buffers are allocated in the receiving CPU's memory to avoid the latency of a read over the bus. For synchronization, a mutex pair per buffer is used. One mutex is used for getWriteBuf and setWriteBuf while the other one is used for getReadBuf and setReadBuf. From a programmer's perspective the interface remains the same for a NoCChannel. In detail, a NoCChannel is implemented as a pair of ClusterChannels, one on the sending cluster and the other on the receiving cluster. The NI on the sending cluster acts as a consumer and the NI on the receiving cluster acts as a producer. For a NoCChannel, data buffers are allocated at the sending and at the receiving cluster.

3.3 Compiler Infrastructure

For programming a single CoreVA CPU, a C compiler tool chain based on the LLVM compiler infrastructure has been developed. Our compiler supports VLIW and SIMD vectorization. However, it is challenging for the programmer to make effective use of a complex MPSoC. Therefore, we have developed a compiler for streaming applications to assist in programming the CoreVA-MPSoC [6]. The applications need to be written in the StreamIt language [13]. An application is represented by a structured data flow graph of its tasks.

Our MPSoC compiler for streaming applications tries to maximize the throughput of the application. An approach based on simulated annealing is utilized to partition the tasks of a program onto particular cores of the MPSoC. Additionally, it decides if a task can be cloned to exploit data parallelism. Furthermore, the granularity of work done in each iteration can be increased to reduce the overhead of communication. Every partitioning is checked for hardware limits. These can be NoC limits like maximum packet size or CPU limits like maximum memory size.

4. THE NETWORK INTERFACE

The network interface (NI) of the CoreVA-MPSoC realizes the communication between two CPU cores of different clusters. Each CPU cluster can be addressed by its unique X and Y coordinate in the 2D-mesh topology of the NoC. Within a cluster a shared address space is used for all memories and components of the CPU cluster. From the programmer's point of view, the communication is based on the CoreVA-MPSoC communication model presented in Section 3.2. To support this model with the NoC, the NI bridges between the address-based communication of the cluster and the flow- and packet-based communication of the NoC. The synchronized buffers (packets) of a channel lead to an end-to-end flow control, which prevents NoC deadlocks. Packets are only send if the destination memory is writable.

The most important task of the NI is to provide an efficient flow control between CPU cores, while minimizing the CPU's software costs for this communication. To achieve this, packet data is directly stored to and read from each CPU's local data memory. Hence, the CPUs can benefit from the low access latency of its local memory. Additionally, the NI acts like a DMA controller by sending and receiving packets (buffers) concurrent to CPU processing.

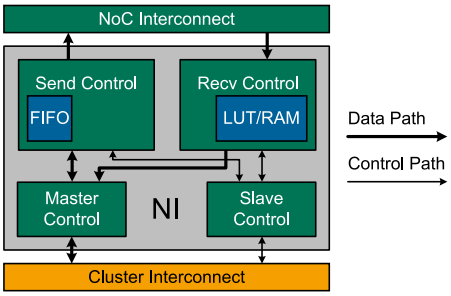


Figure 2: Network-Interface (NI).

Figure 2 gives an abstract overview of the NI architecture. The NI is connected to the cluster interconnect via an AXI master and an AXI slave port. Via Slave Control all CPUs can access and configure the NI. The Send Control handles the sending of buffers. When the source CPU has produced a data buffer, it stores a sending request in a FIFO within the Send Control. With the help of this FIFO the NI is able to handle concurrent sending requests from several CPUs of a cluster without any blocking of the CPUs. Requests within the FIFO are executed successively by the Send Control. To minimize the NI's area requirements, a sending request just consists of a pointer to the CPU's memory where additional channel information is stored. At first the NI starts reading additional channel information, e.g. XY-coordinates of the destination, or the size and pointer of the channel's data buffer. Afterwards the Send Control starts reading the data from the data memory of the source CPU, segments it into flits, and sends it to the NoC interconnect. A flit can be sent every clock cycle and contains a control-header (containing, e.g., flow ID) and 64-bit of payload data.

The Recv. Control handles incoming flits from the NoC interconnect. Receiving is more complex compared to sending, because flits of the same packet might not arrive successively. It is possible that flits of packets from other sources arrive at the NI in between. To identify flits of a packet (buffer), all flits get the same flow ID (Transaction ID), which needs to be unique within a receiving NI. With the unique flow ID the Recv. Control can acquire information about the corresponding channel from a look up table (LUT). The channel information is a pointer to the CPU's memory to store data or to set a mutex for synchronization. With this information the NI is able to store incoming flits directly in the data memory of the target CPU via the Master Control. A LUT entry must be configured before a buffer can be sent via the NoC. According to the number of entries the LUT can be implemented using registers or SRAM memory. The number of LUT entries is the maximum number of independent receiving channels which can be used simultaneously by the tasks executed on the CoreVA-MPSoC.

5. RESULTS

This section presents the analysis of the CPU software costs of NoC-based communication. In addition, we vary the number of independent NI channels and discuss its impact on the performance of streaming applications and hardware costs using a 28 nm FD-SOI standard cell technology.

5.1 Software Costs of NoC Transfers

System-level analysis of MPSoCs requires the consideration of both hardware and software costs and latencies. Message-based communication has to be initiated by the sending CPU and terminated by the receiving CPU. This results in software costs in terms of CPU cycles.

To determine these costs we have implemented a synthetic benchmark and analyze the software costs and hardware latencies for CPU-to-CPU communication. Two tasks are mapped to two CPUs of different clusters. One task is producing data and sends it periodically via a NoCChannel to the consuming task. Figure 3 presents the abstract sequence for sending and receiving one buffer of the channel. In addition, the software costs of the getWriteBuf, setWriteBuf, getReadBuf and setReadBuf functions is shown (cf. Section 3.2). These costs are independent from the buffer's payload size and include both the CPU's buffer management and the CPU's communication with the NI. While the CPU is executing these functions, it can not perform its actual work. Hardware communication latencies, which are illustrated by the data transfer arrows (cf. Figure 3), are typically hidden by using double- or multi-buffering.

There are two approaches to handle the limited number of physical NoCChannels. The first approach manages this limit and restricts the number of communication channels to the number of physical channels. LUT entries of the NI's receiving path (physical channels) are configured only once during an initialization phase. Afterwards a pre-configured LUT entry can be used periodically for the same channel. We call this approach semi-static.

The second, more dynamic approach does not limit the number of communication channels. In this case a sending task can reserve a physical channel for the time which is required to transmit a single data buffer. Afterwards the physical channel can be used by other channels. This approach requires a reconfiguration of the LUT in the receiving NI for every data buffer transfer. To ensure synchronization the reconfiguration is done by the sending task, which needs to know the address for data and mutex location at the receiving CPU. This requires global address management or a handshake at the application's start up.

To show the difference between both approaches we use a synthetic benchmark and analyze the latency for CPU-to-CPU communication. The results in Table 1 present the

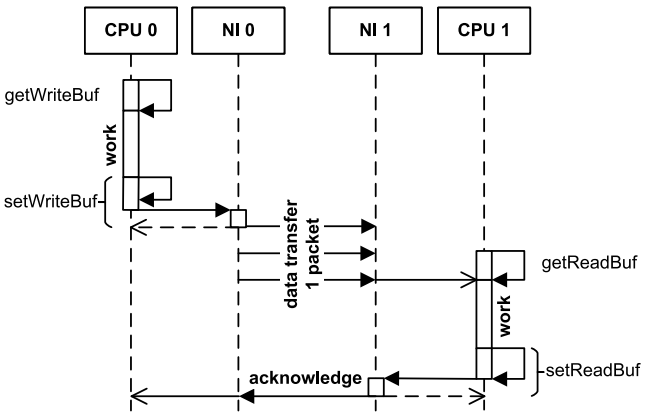


Figure 3: Sequence diagram of the NI.

Table 1: Best-case cycle count for ClusterChannel and semi-static and dynamic NoCChannels.

	semi-static	dynamic	cluster
getWriteBuf	15	17	15
setWriteBuf	28	126	19
getReadBuf	15	15	15
setReadBuf	19	19	19
data transfer (16 B)	26	148	20
data transfer (1 kB)	177	285	20

different software runtime costs for the transfer of one data buffer of a channel. In addition to both NoC approaches, costs for a CPU-to-CPU communication within a cluster are shown. Even a cluster communication needs to manage and synchronize the data buffers. The main difference between all approaches is reflected in the setWriteBuf function, which initiates the sending of the buffer. Due to the reconfiguration of the LUT entry on the receiving NI, the runtime for setWriteBuf increases by a factor of 4.5 for the dynamic approach compared to the semi-static approach.

In addition to software costs, the latency for a data transfer (payload size of 16 B and 1 kB) is shown. The latency is measured from the time the sender has produced the buffer (before setWriteBuf) until the receiver is able to consume it (after getReadBuf). Due to the block transfer of data (1 kB) via NI, the latency for semi-static NoC channels is 157 cycles longer compared to a cluster communication. During the task's work function, CPUs within the same cluster can store the data directly to the local memory of the other CPUs. For a NoC transfer the entire data buffer needs to be produced before the NI can start sending the buffer.

The results presented in this section represent best case scenarios. All buffers are available and there is no congestion with other traffic. If all physical channels of an NI are in use and the dynamic strategy is chosen, a requesting CPU is blocked until a physical channel becomes available. In addition the read required for this request can be delayed due to high network congestion. For this reason the semi-static strategy is more deterministic, which is important for real time applications.

5.2 Streaming Benchmarks

As shown in the previous section, the semi-static approach is more efficient in terms of CPU cycle costs. The drawback is that the partitioning of an application becomes more challenging due to the limited number of communication channels. Especially streaming applications can benefit from the semi-static strategy, because the processing of a continuous stream of data is a periodic task. Once a physical channel is configured it can be used periodically for the whole runtime of the application.

To determine the impact of the number of physical communication channels on application performance, we use 10 streaming applications. These applications are derived from the StreamIt benchmark suite [13]. Because our CPU lacks floating point support, we have ported some benchmarks to a fixed-point representation. We use our MPSoC compiler (cf. Section 3.3) to automatically partition the different tasks of the benchmarks to our CoreVA-MPSoC. Three different CoreVA-MPSoC configurations with 32 VLIW CPUs are considered: two hierarchical configurations 4x2x4 and 2x2x8

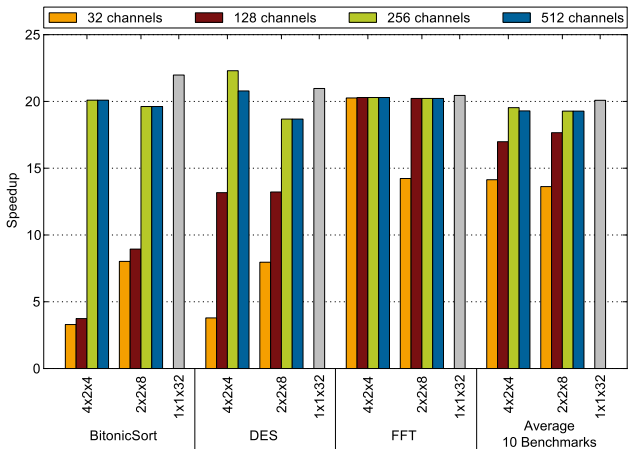


Figure 4: Speedup of MPSoC configurations with different number of receive channels per NI compared to a single CPU.

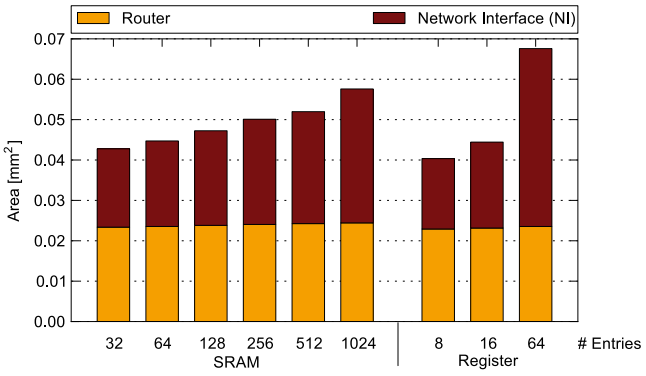


Figure 5: Area requirements of routers and different NI configurations.

(NoC_dimesion_1 x NoC_dimension_2 x #CPU_per_cluster) and a single CPU cluster (1x1x32).

Figure 4 shows the increase of throughput using selected benchmarks for the considered MPSoC configurations compared to a single CPU. The column *Average* shows the mean speedup of all 10 applications of our benchmark suite. Additionally, we vary the maximum number of physical channels from 32 to 512 per NI. A low number of physical channels constrains the number of valid partitionings that are considered by the optimization algorithm of our MPSoC compiler. Depending on the application this may effect the achievable speedup.

Considering all benchmarks, the 1x1x32 configuration shows the best speedup (20 in average). For this configuration no NoC communication is used and the number of communication channels is only limited by the data memories of the CPUs.

Considering hierarchical MPSoCs, BitonicSort shows the highest impact on a low number of physical NoC channels. For channel numbers of 32 and 128 the speedup does not exceed 3.8 for the 4x2x4 MPSoC and 8.9 for a 2x2x8 MPSoC. In these cases the compiler does not find a partitioning that benefits significantly from more than one CPU cluster. The speedup increases to 20.1 (4x2x4) and 19.6 (2x2x8) if 256 channels are used. DES shows a performance similar to BitonicSort except that 128 channels show a better speedup (13.2) compared to 32 channels with a speedup of 3.8 (4x2x4). The 2x2x8 configuration with 256 channels shows a higher speedup than 1x1x32 because the NI acts like a DMA controller and offloads the CPUs from copying. In addition the 512 channel configuration is slower due to a local minimum of the partitioning algorithm. For FFT most configurations show a speedup of about 20. The average of all 10 benchmarks shows that 256 physical NoC channels is a reasonable number. Performance increases by 27.6% or 13.1% compared to 32 or 128 NoC channels respectively. No benchmark benefits from more than 256 NoC channels.

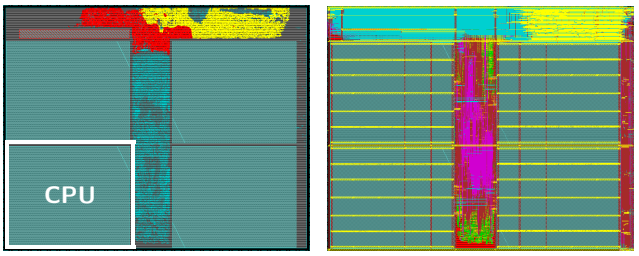
5.3 Physical Implementation Results

This section presents physical implementation results for a single CPU cluster including 4 CPUs, the AXI interconnect, the NI and one router. To connect several of these CPU clusters, four ports of the router are used as IO-interfaces. We use a highly automated standard-cell design-flow based on Cadence Encounter Digital Implementation System. The maximum frequency of the CoreVA CPU is 900 MHz in a 28 nm FD-SOI standard cell technology[1] with low- and regular-VT standard cells. Basic blocks of our analysis are hard macros of our CoreVA CPU using 2 VLIW slots and 16 kB L1 instruction and 16 kB L1 data memory. In this work, we use a slightly relaxed timing of 800 MHz. This results in area requirements of 0,116 mm^2 per CPU macro.

Synthesis results presented in Figure 5 show the area requirements of NI and router with a different number of independent receiving channels per NI (LUT entries). The router requires about 4.1% of the total area of a CPU cluster with 4 CPUs. Router area increases only slightly with the number of NI channels, because the channel ID (flow_ID) is part of each flit header. The size of the flit header is 24 bit for 8 physical channels and 31 bit for 1024 physical channels (for a 4x4 NoC). An NI with 32 channels using an SRAM LUT has an area of 0.019 mm^2 and takes just 3.45% of the total cluster area (0.563 mm^2). To support up to 1024 independent receive channels, larger SRAM memories are used, which increase the area of the NI to 0.033 mm^2 and the total cluster area to 0.578 mm^2. As expected, area requirements increase dramatically for a register-based implementation of the LUT and many channels. A register-based NI with 64 channels has an area of 0.044 mm^2 and takes 7.50% of the total cluster area. For NI configurations with 8 channels (0.017 mm^2) or less, a register implementation could be an option. Using registers reduces the latency of a flit in the receiving path by one cycle compared to an SRAM implementation. Total area requirements of MPSoC configurations with 32 CPU cores are 4.56 mm^2(4x2x4), 4.44 mm^2 (2x2x8) and 4.82 mm^2 (1x1x32). The NoC-based configurations feature an SRAM-based NI with 256 channels.

Place and route (P&R) results of a CPU cluster with 4 CPUs and an NI with a memory block for 128 channels are presented in Figure 6. Figure 6a shows the placed design with the NI highlighted in red and the router highlighted in yellow. Additionally the routed design shows routing wires to the four IO ports of the router (cf. Figure 6b). The area requirements (0.67 mm^2) increase by 18% compared

[1]STMicroelectronics, 10 metal layer, Worst Case Corner: 1.0 V, 125°C



(a) Placed design (NI is marked red, router is marked yellow).

(b) Routed design.

Figure 6: Physical implementation of a CPU cluster with 4 CPU macros and 4 times 32 kB local memory. Area requirements are 0.70 mm^2.

to synthesis results. Due to our hierarchical synthesis flow, multiple of these cluster macros can be placed next to each other to build up the NoC.

For comparison of area requirements and application performance, we use the ratio of speedup and area as a metric. Figure 7 shows the speedup-area ratio for the considered benchmarks and hardware configurations. On average, the 4x2x4 MPSoC with 256 NI channels shows the best speedup-area ratio of 4.45. Configurations with both 256 and 512 NI channels show good results and outperform the 1x1x32 full crossbar CPU cluster (4.16). In addition the NoC-based MPSoC configurations easily allow for large-scale MPSoCs with hundreds of CPU cores.

6. CONCLUSION

In this work a novel network interface (NI) architecture for our hierarchical CoreVA-MPSoC is presented. A system-level analysis has been performed, considering both software and hardware costs of different NI configurations. Benchmark results show that NoC communication with a semi-static channel approach outperforms a dynamic channel approach. In addition we have compared different NI configurations for the semi-static approach using a benchmark suite of 10 streaming applications. The best performing NI configuration shows an average speedup of 20 for a hierarchical CoreVA-MPSoC with 32 CPUs compared to a single CPU. Physical implementation results using a 28 nm FD-SOI standard cell technology show area requirements of 4.56 mm^2 for a 4x2x4 CoreVA-MPSoC at 800 MHz. Compared to a single CPU cluster with 32 CPUs the ratio of speedup and area increases by 7% for a 4x2x4 MPSoC with 256 NI channels. The proposed approach enables fast design space exploration to optimize MPSoC configurations for a given application scenario. In future work we will extend our MPSoC compiler to choose dynamic or semi-static channel approach at compile time. In addition, we will connect the NI to a tightly coupled shared L1 memory [11] of a CPU cluster. This minimizes the NI's write and read latencies.

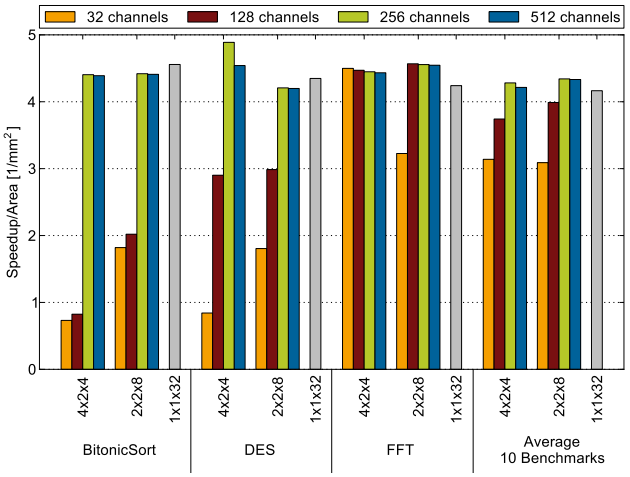


Figure 7: Speedup-area ratio of MPSoC configurations with different number of receive channels.

Acknowledgments

This work was funded as part of the DFG Cluster of Excellence Cognitive Interaction Technology 'CITEC' (EXC 277), Bielefeld University and the BMBF Leading-Edge Cluster "Intelligent Technical Systems OstWestfalenLippe" (it's OWL), managed by the Project Management Agency Karlsruhe (PTKA).

7. REFERENCES

[1] Adapteva. E64G401 Epiphany 64-Core Microprocessor Datasheet, 2014.

[2] L. Benini et al. P2012: Building an Ecosystem for a Scalable, Modular and High-Efficiency Embedded Computing Accelerator. In *DATE*. IEEE, 2012.

[3] B. Dinechin et al. A Clustered Manycore Processor Architecture for Embedded and Accelerated Applications. In *HPEC*. IEEE, 2013.

[4] T. R. Halfhill. Opportunity NoCs, NetSpeed Answers. *Microprocessor Report*, (December), 2014.

[5] T. Jungeblut et al. A TCMS-based Architecture for GALS NoCs. In *ISCAS*, pages 2721–2724. IEEE, 2012.

[6] W. Kelly et al. A Communication Model and Partitioning Algorithm for Streaming Applications for an Embedded MPSoC. In *SoC*. IEEE, 2014.

[7] T. Marescaux et al. The Impact of Higher Communication Layers on NoC Supported MP-SoCs. In *NOCS*. IEEE, 2007.

[8] A. Radulescu et al. An Efficient On-Chip NI Offering Guaranteed Services, Shared-Memory Abstraction, and Flexible Network Configuration. *IEEE Trans. on Computer-Aided Design*, 24(1):4–17, 2005.

[9] S. Saponara et al. Design of an NoC Interface Macrocell with Hardware Support of Advanced Networking Functionalities. *IEEE Trans. on Computers*, 63(3):609–621, 2014.

[10] G. Sievers et al. Evaluation of Interconnect Fabrics for an Embedded MPSoC in 28nm FD-SOI. In *ISCAS*. IEEE, 2015.

[11] G. Sievers et al. Comparison of Shared and Private L1 Data Memories for an Embedded MPSoC in 28nm FD-SOI. In *MCSoC*, 2015. In press.

[12] J. Sparso et al. An Area-efficient Network Interface for a TDM-based Network-on-Chip. In *DATE*. IEEE, 2013.

[13] W. Thies et al. StreamIt: A Language for Streaming Applications. In *Compiler Construction (CC)*. 2002.