Abstract—This paper presents a system architecture, data communication scheme and application programming interface model or concept for a multiprocessor system based on a network-on-chip (NoC) platform. Each processing node connected to a mesh node has its own local (instruction and data) memory portion, and a global (shared) memory portion. The introduced communication scheme gives only a minimum overhead in order to offer direct memory-to-memory data transfer. Each processor can make direct message delivery to another processor (producer initiated), or make a request to copy memory blocks from a remote processor (consumer initiated). The complete data transmission is handled by the network interface and a special memory controller. The network interface managed by the specialized memory controller can directly access the shared memory portion. Thus the processing node can continue its normal operation and will not be blocked during the data transfer process.

Keywords—Network-on-Chip, Many Core Processors, Application Programming Interface, Network Interface

I. INTRODUCTION

Parallel multiprocessor systems with multiple cores are a state of the art of next computer generations. Further exploitation of the full performance of multiprocessor systems brings the challenge to overcome the common bottleneck, the shared memory in a bus-based platform. In a bus-based multiprocessor system only one processor can use the bus to read or write data from or to the memory at a time. In the meantime the other processors have to wait until they can perform their memory access. This idle waiting time wastes processing power of the system, thus the performance can not be fully exploited. The scaling issue can be encountered with the Network-on-Chip (NoC) paradigm [3].

Distributed Shared Memory (DSM) has been an interesting issue in all kinds of multiprocessor systems in the recent years. Memory access topologies and memory bandwidth are crucial points to gain the overall targeted system performance. In [6] a performance evaluation for the Cray X1 DSM architecture is presented. In X1 multistreaming processors (MSPs), memory access is performed via a cache, which is shared by four single stream processors (SSPs). Four MSPs share 16 memory banks, having 16 individual memory controllers. This allows local memory access in parallel to global data communication, accessing some of the 16 memory banks. Principals of DSM architectures have already been presented in [9], where structure, granularity and coherence issues are described. [5] gives a clear description and evaluation of producer-consumer mechanisms in shared memory multiprocessors. Comparing producer-initiated and consumer-initiated data communication schemes, producer-initiated mechanisms (as data forwarding and user-level message delivery) provide the highest efficiency, being comparatively insensitive to network parameters (latency, bandwidth) [5]. In [2] a dynamic approach for balancing memory access and avoiding access contention is presented, which applies memory page migration in consumer-initiated DSM systems. Interesting DSM reference architectures are the MIT Alewife Machine architecture [1] and the Stanford DASH (Directory Architecture for Shared Memory) multiprocessor [7]. Bhuyan et al. [4] present a multistage bus-based architecture for the realisation of a DSM system. In [8] a crossbar NoC architecture as a platform for a shared-memory architecture has been presented, where several processing elements, several shared memory units and a main memory controller are connected to a central crossbar. This approach also follows the NUMA paradigm.

II. CONTRIBUTION

In this paper, we present an efficient memory organization, which is implemented based on scalable mesh-based NoC architecture (XHiNoC). Conceptionally, the system architecture presented in this paper is a distributed memory multiprocessor system supporting a parallel programming model with functional-task-level parallelism. The presented approach is based on the following main goals:

- Slim architecture with reduced administration effort.
- Support of different DSM data exchange paradigms (producer-initiated message delivery and consumer-initiated programming models).
- Enhanced benefit from NoC multicast capability for advantageous producer-initiated data communication.
- Low requirements to application programming interfaces (APIs), which allows to integrate processors, but also dedicated hardware components with low wrapping effort.
- Applicability to heterogeneous NoC-based multiprocessor systems.

This paper presents also an efficient approach to develop functional-task-based programming by using instruction library (application programming interface, API) which have been developed to program the MIPS-based multiprocessor systems. Some existing concepts of the programming models for multiprocessor systems have been presented in [6], [9], [5], [2], [1], [7] and [4], which are mainly not dedicated for on-chip multiprocessor systems. The work in [8] presents the commonly used shared memory programming model for on-chip multiprocessor. However, the work in [8] cannot support producer-initiated message delivery programming mode and has not presented in detail so far how to create a simple computer
program to implement task applications which are commonly used in embedded application.

The ongoing part of this paper is organized as follows: first we introduce one possible instance of a NoC attached processing node (in this case applying a “Plasma” [10] core) and the general memory architecture of the XHiNoC based DSM system. Then we describe the system’s transaction handling in context with the NoC network interface, followed by a section which outlines the low level access to the transaction mechanism. Finally we present some simulation results of the VHDL-implemented system followed by conclusions and open issues.

III. SYSTEM ARCHITECTURE WITH DISTRIBUTED MEMORY

A. On-Chip Network

Our distributed memory multiprocessor system is designed on a NoC platform where each processing node is connected with a local port of a NoC mesh router through an on-chip network interface. The selected NoC, which is called XHiNoC (Extendable Hierarchical Network-on-Chip) [11], [12], [13] is designed based on modular approach to support further extensions in terms of network communication services and topologies for bandwidth requirements. The main characteristic of the XHiNoC is that, packets are switched with a wormhole technique, routed by a routing engine, which consists of routing hardware logic and identity-tag-based routing table unit, and scheduled in the NoC based on the local identity-tag management systems at runtime. The local identity-tag attached to every flit allows different flits of different packets to be mixed in the same queue or to share communication channels, because flits belonging to the same message will have the same local ID-tag in certain communication channels and are updated to support bandwidth share and scalability concepts.

B. MIPS-based Tile Processing Unit (TPU)

Fig. 1(a) shows a snapshot of one tile in the NoC. Each tile consists of a MIPS core, an UART (I/O component), a specialized memory controller (MemCtrl), a private data memory, an instruction memory, a shared data memory with a reserved block/segment for transaction handling (TsnBuffer), and an on-chip network interface (OCNI). The private data and instruction memories are actually integrated into the same memory component but are separated with different memory blocks/segments. The MemCtrl and UART units are memory-mapped components.

We have designed by ourself new blocks i.e., XHiNoC router, OCNI and MemCtrl and integrate them with the existing Plasma MIPS system [10]. Fig. 1(b) presents the circuit layout of one tile using 180-nm CMOS standard-cell technology from UMC. With 185 MHz targeted data frequency, the logic cell areas of the Router, MemCtrl and OCNI are about 0.1635 mm$^2$, 0.0629 mm$^2$ and 0.2460 mm$^2$, respectively. The memory controller is used to decouple the processors memory accesses from the OCNI. Due to the fact that the processor will try to access the memory in each clock cycle for either instruction or data fetch operations, a single memory block would be busy all the time. In order to grant memory write access to the OCNI promptly on incoming data without blocking the processor, the available memory has to be split up. The L2 memory is used exclusively by the processor, and in parallel, the OCNI can operate on the shared data memory. During this parallel memory operations, the MemCtrl forwards address and data to control memory read and write operation between the shared memory and OCNI.

The MemCtrl may grant the processor to access the global memory, as long as the OCNI has no incoming data to process. Incoming data is prioritized over outgoing data to avoid network congestion. A small block of the shared data memory is reserved for the transaction buffer (TsnBuffer). This ring buffer is managed by the MemCtrl, whereas the rest of the memory is managed by the processor, or the operating system.

1) Transaction Descriptor and Packet: In order to keep track of transaction, each transaction has a unique ID (Tsn ID). This ID is required to send short acknowledge messages or reject a transaction request. The used 4-bit Tsn ID field limits the amount of transactions.
### A. Consumer-Initiated Transaction Mechanism

By using consumer initiated communication scheme, the system may execute various actions during application execution time such as (1) requests a remote memory block, (2) answers the request by sending the data, or (3) rejects the request. Due to the parallelism in multicore systems, each node may have more than one pending transaction to be handled at a time. A transaction consists of an issue and an answer. Fig. 3 shows the transaction mechanism between Tile Processor Unit (TPU) at node A and at node B. Each step in the transaction mechanism is labeled numerically in the figure, and is explained in the following items.

1) As presented in Fig. 3(a), (1) the TPU at node A fetches instruction from instruction memory (imem) and needs a block of memory from node B. (2) The TPU issues a new transaction (request action) to its memory controller (memc). (3) The request is placed to the transaction buffer (Tsn-b) to handle. The memc points the address to which the data will be stored. (4) The on-chip network interface (OCNI) at node A fetches the topmost transaction from the buffer, and (5) assembles the Tsn-describer to be XHiNoC packet. (6) The OCNI at node B receives and disassembles the packet and interprets it as a request, then (7) the memc write the new transaction to the Tsn-b. (8) The Tsn-b points the address from which the data will be sent and the length of data. (9) The OCNI at node B fetches the topmost transaction from the Tsn – b. Until this step, the data are ready to send.

2) As presented in Fig. 3(b), (10) the OCNI at node B assembles the data to be XHiNoC packet. The requested data is taken from the shared data memory sdmem. (11) The OCNI at node A receives the packet, interprets it as receiving data and places the data directly to the sdmem. (12) After data from the sdmem at node B has been copied to the sdmem at node A, then the memc at node A marks the answered action as “finished transaction” and informs the CPU.

Therefore we have implemented some mechanisms in our system architecture to provide the needed abilities, i.e. (a) processor access to the queue to issue new transactions, (b) grant direct access to the transaction queue to the OCNI, (c) the OCNI may store incoming transactions to the queue on its own, (d) a uniform transaction descriptor format to store source and destination memory addresses, and (e) an additional queue for finished transactions with processor interruption. A crucial point is the concurrent access to the transaction queue by the processor and the network interface. Thus the transaction buffer managed by the memory controller has been implemented to support atomic access methods.

Furthermore the exceptional state of rejecting a new request has been considered. If the OCNI at node B cannot issue a new transaction, for instance the transaction buffer is full, it must directly reply the request with a reject message. The processor at node A receiving the reject can retry the request. 

### B. Producer-Initiated Message Delivery

Our system architecture support also a direct data delivery communication scheme where a data producer directly send data to the consumer, as also illustrated in the figure. The transaction mechanism is labeled numerically in the figure, and is explained in the following items.

1) As presented in Fig. 3(a), (1) the TPU at node A fetches instruction from instruction memory (imem) and needs a block of memory from node B. (2) The TPU issues a new transaction (request action) to its memory controller (memc). (3) The request is placed to the transaction buffer (Tsn-b) to handle. The memc points the address to which the data will be stored. (4) The on-chip network interface (OCNI) at node A fetches the topmost transaction from the buffer, and (5) assembles the Tsn-describer to be XHiNoC packet. (6) The OCNI at node B receives and disassembles the packet and interprets the context as a request, then (7) the memc write the new transaction to the Tsn-b. (8) The Tsn-b points the address from which the data will be sent and the length of data. (9) The OCNI at node B fetches the topmost transaction from the Tsn – b. Until this step, the data are ready to send.

2) As presented in Fig. 3(b), (10) the OCNI at node B assembles the data to be XHiNoC packet. The requested data is taken from the shared data memory sdmem, (11) The OCNI at node A receives the packet, interprets it as receiving data and places the data directly to the sdmem. (12) After data from the sdmem at node B has been copied to the sdmem at node A, then the memc at node A marks the answered action as “finished transaction” and informs the CPU.
data consumer node without a transaction request as presented in the consumer-initiated data communication scheme. Fig. 4 show the mechanism which consists of data/message delivery phase (Fig. 4(a)) and data acknowledge phase (Fig. 4(b)).

1) As presented in Fig. 4(a), (1) both the TPU at node A and B fetch instruction from instruction memory (inmem). (2) The TPU unit at node A and B issue an initial action to their memory controller (memc). (3) The initial actions are placed to the transaction buffer (Tsn-b) of both TPU units. The memc at TPU A points the address from which the data will be sent, while the memc at TPU B points the address to which the data will be stored. (4) The on-chip network interface (OCNI) at node A fetches the data from the shared memory buffer, and (5) assembles the Tsn-descriptor to be XHiNoC packet. (6) The OCNI at node B receives and disassembles the packet, and then (7) the memc write the new transaction to the Tsn-b which points the address to which the data will be stored.

2) As presented in Fig. 3(b), (8) the memory controller unit memc informs the CPU B that the data has been successfully stored in the memory. (9) The CPU at node B then initiates to send a data acknowledge. (10) The OCNI at node B assembles the data to be XHiNoC packet. (11) The OCNI at node A receives the data acknowledge packet and send it to the memc. (12) Now, the CPU at node A has known that the sent data has been stored in the shared memory portion of the TPU at node B.

This mechanism represents a functional-task-level programming model. The data sender and data receptor have known their communication partner during application execution. They have even known the burst size of the data that would be transferred. Therefore, prior to the message delivery phase, a data sender and a data receptor have provided a memory space to copy data from the memory of the data producer and to buffer the data in the remote memory of the data receptor. The programming model of this communication scheme is also suitable to program task applications in embedded MPSoCs, which are commonly described with a task application graph.

C. Application Programming Interface

The low level system programs (API) can be written in C or Assembler to design software model. At system start, an initialization routine must be executed at each node in order to provide free transaction slots managed further by the memory controller. The Tsn slots are placed within the shared segments of each memory. Each slot needs 5 words (4 bytes), which leads to 20 bytes per slot.

In order to perform the consumer-initiated data communication scheme, we introduce two important functions i.e., NoCRequest(TsnID, Net_source, Smem_add, Tmem_add, Burst_size) and DataSend(TsnID, Net_target, Tmem_add, Smem_add, Burst_size). For the NoCRequest command, Net_source is network address of the remote processor where the data source is located, Smem_add is memory address location of the data source in the remote processor, and Tmem_add is memory address where the data source will be saved in the local requesting processor.

In order to perform the producer-initiated data communication scheme, we introduce a single function i.e., NoCDeliver(TsnID, Net_target, Tmem_add, Smem_add, Burst_size). In the NoCDeliver command, Net_target is network address of the remote processor who requesting the data source, Smem_add is memory address location of the data source in the local processor, and Tmem_add is memory address where the data source will be saved in the remote requesting processor. The TsnID and Burst_size is the ID label of the transaction and the size of the requested data, respectively.

Due to latency issues one has to provide one spare slot for ongoing transactions. This is crucial, because sending a request may lead to receiving the reject before the request has been sent completely.

During normal operation one may launch a new transaction. First a free slot a to be fetched from the list. If no free slots are available, this results in zero (null-pointer). Next one can fill in the needed transaction informations and push the slot to the queued list. Optionally one may register a callback function to the used transaction ID for further issues.

Each transaction is processed by the dedicated hardware until it has been finished. Finished transactions are pushed to the finished list by the network interface. Additionally an interrupt is sent to the processor, if finished transactions may be processed. One may fetch a slots from the finished list. The result will be zero, if no finished transactions are available. Then the registered callback function may be called for instance. Finally the transaction slot must be pushed back to the free list in order to make the slot available for further usage.

V. FUNCTIONAL-TASK-LEVEL MODEL APPLICATION

This sub-section presents how to develop a programming model to perform a task-based application into the NoC-based multiprocessor system. In this experiment, we use the producer-initiated data communication scheme, where each data producer node will send data directly to the data consumer node. An example of the task application graph model called picture-in-picture (PIP) application is presented in Fig. 5(a). The mapping result of the application on the NoC platform is presented in Fig. 5(b).

Alg. 1 presents a pseudo code on how to make a program to perform the functional-task-based application presented in Fig. 5. The listing shows the list of each task described in a program subroutine, the instantiation of each task on every node address of NoC according to the task application mapping result presented in Fig. 5(b). For instance, we can see that Task1 is allocated in the Processor node 0x02.

Fig. 6 depicts the data communication in the application layer between Task1, Task2 and Task3. For the sake of the simplicity, we
Algorithm 1 Programming model for an example of task-level application

```
/* begin of communication graph */
Comm1.Source=1; Comm1.Target=2;
Comm1.BurstSize=100; /* unit e.g. in byte */
Comm2.Source=4; Comm2.Target=1;
Comm2.BurstSize=120;
... etc.
/* end of communication graph */
/* begin of functional-task mapping */
TaskMap(1)==0x02;
TaskMap(2)==0x10;
TaskMap(3)==0x12;
... etc.
/* end of functional-task mapping */
/* begin of task procedure list */
void Task1(SCore,SBurst,TCore,TBurst)
... etc.
void Task4(TCore1,TBurst1,TCore2,TBurst2)
void Task5(SCore1,SBurst1,SCore2,SBurst2,TCore,TBurst)
... etc.
/* end of task procedure list */
void AppPIP()
int Source=OWN_NODE_ADDR; /* defined before */
/* do not forget to use compiler directive */
/* not shown in this pseudo code */
if Source==0x02 then
    Task1(0x01,VolCom2,0x12,VolCom1)
... etc.
else if Source==0x21 then
    Task5(0x22,VolCom5,0x10,VolCom7,0x11,VolCom6)
... etc.
end if
End AppPIP
```

VI. CONCLUSIONS

A NoC-based multiprocessor system with distributed memory architecture has been presented in this paper. Application programming interface (API) has been developed allowing users to design application softwares. The users can use consumer-initiated and/or producer-initiated communication schemes to develop application softwares. The use of producer-initiated communication scheme to implement a task-level application has been presented in this paper. By using the API library, the application software can be easily implemented by allocating the burst sizes (volumes) of every considered communication edge in the shared memory portion in the microprocessor systems. The main drawback of the designed programs is, that the executable (binary) code of all tasks of the compiled program must be stored in the every instruction memory of the microprocessor system. Thus, the portion of tasks that will not be executed by a certain local processor must include the single executable code, which probably results in a larger code size especially if the computation complexity of every tasks is very high.

By using the current C/C++ compiler, the users can solve the problem by writing a specific C/C++ program of a task for a specific processor core that will run the task locally. In the future, a work colaboration with software compiler and parallel computing
community to design time-efficient functional task-level parallel computer programs for NoC-based multiprocessor systems would be an interesting inter-disciplinary research activity.

In our system architecture, a software program of an application mapping can be run at compile time. From a master core connected to a NoC router node, the concurrent executable codes can be distributed to the instruction memory of every considered MIPS core, where a task is allocated. In the future, we will integrate heterogeneous processing elements, i.e., various microprocessor systems and ASIC cores, to support hardware-software co-design. This hardware-software on-chip integration in a NoC platform is also a promising issue in the fields of embedded system-on-chip and embedded multiprocessor system-on-chip.

REFERENCES


