Exploiting Multicast Messages in Cache-Coherence Protocols for NoC-based MPSoCs

Tales M. Chaves, Everton A. Carara, Fernando G. Moraes
PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 – Brazil
tales.chaves@acad.pucrs.br, everton.carara@pucrs.br, fernando.moraes@pucrs.br

Abstract— **MPSoCs are largely used in embedded systems, allowing the design of complex systems within short time-to-market. The shift in the communication infrastructure, from buses to networks-on-chip (NoCs), adds new design challenges. Standard directory-based cache coherence protocols represent a performance bottleneck due to number of transactions in the network, reducing performance and increasing the energy consumption. State-of-the-art works investigate new protocols, at abstract levels (e.g. TLM), to optimize the performance of the memory organization. Differently from previous works, we investigate the benefits NoCs can bring to directory-based cache coherence protocols using RTL modeling. The main functionality NoCs may provide for the protocols is the way messages are sent through the network. Most NoCs support multicast as a set of unicast messages. Such method is not suitable for cache coherence protocols, because transactions as block invalidate and block update are naturally multicast. This work proposes the use of multicast messages to reduce the number of transactions to improve the performance of cache coherence protocols in NoC-based MPSoCs. Results show that performance of some transactions is improved up to 32% when using multicast messages.**

***Keywords-component:** embedded systems; memory organization, MPSoC, NoC, cache-coherence protocol (key words)*

I. INTRODUCTION

Multiprocessors Systems-on-Chip (MPSoCs) integrate multiple simple processing elements (PEs) on a single chip. As the number of PEs increases, there is a predominant adoption of Networks-on-Chip (NoCs) as the interconnection architecture due to the high level of parallelism provided through several communication channels [1].

The increasing number of functionalities included on current embedded systems raises the demand for hardware modules, resulting also in a growth of the amount of memory used in these devices. According to [2], the most critical component that determines the success of an MPSoC architecture is its memory architecture. Silva et. al [3] show that communication-dominated applications suffer from the latency of the interconnection infrastructure. These applications may benefit from having a shared memory that stores global data and allows for data interchange between PEs. From one side, shared-memory model facilitates interchange of data between processors and according to [4] facilitates the programming model. From the other side, shared memories are not scalable.

The exploration of caches reduces the required memory bandwidth, since most memory accesses are local, and reduce energy consumption, since the amount of transactions in the communication infrastructure is reduced. However, the use of caches might give raise to the problem of cache incoherence when multiple PEs can cache common data from a shared resource. To avoid this problem, cache coherence protocols must be implemented.

In NoC-based MPSoCs, directory-based protocols are mostly applied as they do not require global visibility of every memory access, as snoopy-based protocols. Directory-based protocols may be optimized to reduce the amount of traffic generated on the NoC. In this paper we demonstrate that the number of transactions can be reduced through the exploration of physical services provided by the NoC, such as multicast messages.

This paper is organized as follows. Section II presents the state of the art. Section III presents the reference MPSoC architecture adopted by this work. Section IV presents the proposed memory organization, along with the cache coherence protocol. Section V presents preliminary results. Finally, Section VI presents conclusions and directions for future works.

II. RELATED WORKS

Cache coherence protocols have been proposed in NoC-based MPSoCs. Jarger et al. [5] proposes a novel solution for cache coherence in multicore architectures, named Virtual Tree Coherence (VTC). It is based on a virtual ordered interconnection tree, which keeps a history of nodes sharing a common region of memory. For each region, a virtual tree of the nodes that shared that region is created. Every time a given region is accessed by one of the nodes, a request is sent to the root of the tree, which in turn, requests the data to the node holding the most updated copy of it. This request is done through a multicast message traversing the tree. The implementation of the multicast according to the tree topology decreases latency when compared to unicast-based implementations. The Authors compare their implementation to a directory-based implementation and to another implementation based on a greedy algorithm. Results show a performance improvement of 25% compared to a directory-based implementation, and 11% over the greedy scheme.

Chtioui et al. [6] present the development of a dynamic hybrid cache coherence protocol for shared-memory NoC-based MPSoCs. According to the Authors, existing protocols

such as invalidation and update, do not take into account the patterns of data accesses performed by processors. In response to that, this paper proposes a protocol based on the traditional directory-based protocol that adapts itself to the way in which the data is used, alternating between an update to invalidate protocol and vice-versa. It is considered dynamic because it can be changed during the execution of the application. Results show that the most significant gain of the hybrid protocol is the reduction on energy consumption when compared to only invalidate and only update protocols.

Petrot et al. [7] present solutions for the problems of cache coherency and memory consistency in NoC-based shared-memory MPSoCS. According to the Authors, software-oriented solution where shared data are uncached and the local, non-shared data, cached, provides a good tradeoff between complexity and performance. The method allows the design to benefit from the cache efficiency with no hardware cost and at moderate software cost for the system integrator. Another software solution for a cache coherence protocol in MPSoCs is presented in [8].

Bolotin et al. [9] explores the use of priority to optimize a distributed directory-based cache coherence protocol. The priorities are used to differentiate messages, such as control and long data messages to achieve a major reduction in cache access delay. Control messages, such as read requests have a higher priority than long messages, such as a L1 miss response of the L2 cache containing a copy of the block. To optimize invalidate messages, a replication-based broadcast is implemented on a store-and-forward 2D mesh NoC. Experiments show a delay reduction of 26% and 24% for read and read exclusive operations, respectively, in a heavily loaded NoC. Also, a 10% delay reduction is obtained, for both read and read exclusive transactions, in a lightly loaded NoC.

Yuang et al. [10] proposes a hierarchical cluster based cache coherence protocol for large-scale NoC-based shared memory architectures. The nodes of the NoC are grouped into units named clusters. Each cluster consists of a group of L1 cache banks and a L2 cache, which is named HEAD and contains a local directory. The shared memory is located off-chip and maintains the global directory. The cache coherence is enforced hierarchically. Similarly to the directory-based cache coherence protocol, the global directory is a flat, full-map directory, which stores the status of each block. The difference is that instead of keeping which caches have copies of the blocks, the global directory keeps which clusters have a copy of each block. Intra-cluster, the local directory stores which nodes of the cluster (L1 caches) contain a copy of each block. The advantages of using such protocol for enforcing coherence on the platform is the reduction on the number of hops traversed by messages to enforce the coherence protocol (Longest Manhattan Distance and Long distance travel). Another advantage is the reduction on the space required for storing the directory. A disadvantage of the clustered approach is that each application must be mapped onto a single cluster, which is not always possible, what can degrade performance significantly.

One common feature of previous works is the abstract modeling. Abstract models allows faster simulation, but does not capture precisely the low-level communication behavior, including phenomena such as congestion, burstiness and jitter. This work uses the HeMPS MPSoC [11], modeled in synthesizable RTL VHDL, enabling an accurate clock cycle performance evaluation. Also, the literature does not explore the advantages NoCs can bring to cache-coherence protocols, using such communication infrastructure only to send and receive messages.

This work adds a two-level memory hierarchy to the HeMPS MPSoC. The first level is composed by a shared memory bank attached to the NoC, and accessed by all PEs. The second level consists of the processor memory, with two distinct address spaces: one for caching global data and the other for storing private data and instructions. Cache coherence between shared memory and caches is ensured by a hybrid implementation of a MSI protocol. *Our main goal* is to evaluate performance gains when adopting multicast messages to reduce the number of transactions of the cache coherence protocol.

III. MPSoC Reference Architecture

HeMPS [11] is a homogeneous MPSoC, described in synthesizable VHDL, in which PEs are connected to the Hermes NoC. Each PE, named Plasma-IP, contains a MIPS-like processor (Plasma), a local memory (RAM), a DMA controller and a Network Interface (NI). A general view of a 2x2 instance of the HeMPS architecture is illustrated in Figure 1. Tasks communicate through message passing, and there is no shared memory.

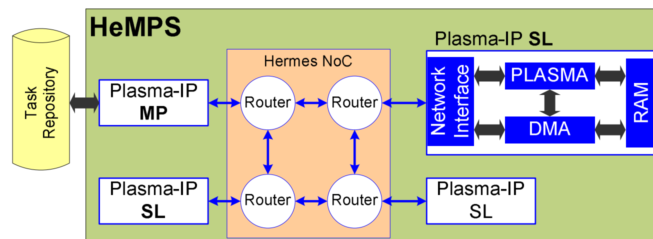

Figure 1 – Block diagram of the HeMPS Platform.

Two types of Plasma-IPs are used: slaves (SL) and master (MP). Plasma-IP SLs are responsible to execute application tasks. Plasma-IP MP is responsible to manage task mapping and system debug. The external memory, named *task repository*, contain all application tasks. According to some mapping heuristic, the Plasma-IP MP maps the tasks into the Plasma-IP SLs. The Plasma-IP MP can also receive debug messages from Plasma-IP SLs, transmitting them to an external host through an Ethernet interface (not shown in Figure 1).

Each Plasma-IP runs a tiny operating system (named *microkernel* whose memory footprint is around 20 KB), responsible to manage and support task execution and task communication. This microkernel is a fully preemptive operating system where each task uses the CPU for a pre-defined period of time called *timeslice*. This microkernel supports multitasking and software interrupts (*traps*).

The network interface and DMA modules are responsible for sending and receiving packets, while the Plasma processor performs task computation and management. The local RAM is a true dual port memory allowing simultaneous processor and DMA accesses. This local memory is organized in a parameterizable number of pages. The *microkernel* uses the first pages, and the application tasks the remaining ones.

This work adopts a modified version of HeMPS, which contains mechanisms to guarantee Quality of Service (QoS) at the software and hardware levels, named HeMPS-Q. The NoC used in HeMPS-Q has the following features to support QoS: packet and circuit switching, priorities, duplicated physical channels, and multicast. Each packet transmitted in the NoC is broken into *flits,* which corresponds to the number of bits that can be sent through the NoC links. In this work, each link between the routers of the NoC is 16-bit wide.

The most important feature of HeMPS-Q used in the present work is the multicast support. The dual-path multicast algorithm is used to transmit multicast messages [12]. In the dual-path algorithm two packets are transmitted. The first packet is transmitted to all PEs with addresses higher than the PE originating the multicast message. The second packet is transmitted to the other set of PEs. Both packets are routed according to the Hamiltonian routing algorithm. Unicast messages are also transmitted using this routing algorithm.

Multicast messages can be used by cache coherence protocols when invalidation messages must be sent to all processors that are caching a given block, for instance. Without multicast, an invalidation message would have to be sent individually to all processors in the system, increasing the number of transactions in the network, as well as energy consumption (due to enhanced switching activity on routers) and congestion. Another example of a NoC that provides multicast service is presented in [13].

IV. MEMORY ORGANIZATION

Our work adopts a two-level memory hierarchy, as illustrated in Figure 2. The first level is the shared memory (SM), accessible by all PEs. The SM contains: (*i*) a memory controller, responsible for executing read/write operations according to the cache-coherence protocol; (*ii*) a directory memory, which stores for each block its status (*shared* or *modified*) and the addresses of PEs holding a copy of it; (*iii*) a network interface (NI) connecting the SM to the NoC; (*iv*) the memory bank, logically divided into blocks of 128 32-bit words. Note that more than one SM can be used in the system, resulting in a distributed shared memory organization. In the scope of this work, only one SM is used.

The second level contains the processor memory, with two distinct address spaces: one for the *cache* memory and the other one for the *local* memory. The local memory stores private data and instructions, and can be seen as a scratchpad memory, storing the microkernel*,* application tasks and local data. The cache memory stores temporary copies of blocks from the SM, and it is managed by the cache controller, which contains the tag memory. All necessary instructions to execute a given task are stored in the local memory. The SM is exclusively used for storing shared data. Therefore, there is no traffic in the network due to fetching instructions from SM to PEs. The advantage of this memory hierarchy is the smaller traffic inside the NoC. In addition, the native HeMPS communication mechanism, message passing, is extended to shared memory communication.

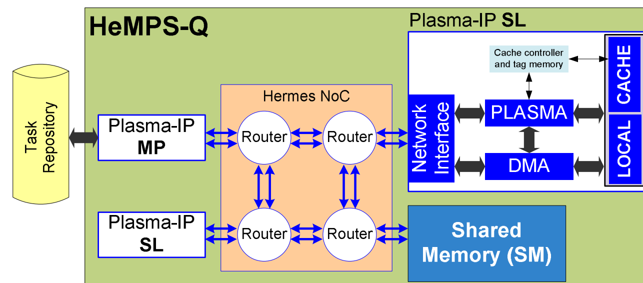

Figure 2 - HeMPS-Q with a two level memory hierarchy – shared memory and caches (only Plasma-IPs SL may contain caches).

The cache adopts the *direct mapping* scheme, due to the minimum hardware support required to implement it. The tag memory stores for each block: (*i*) *tag*, the most significant bits of the SM block address stored in the block; (*ii*) *dirty bit*, the block currently mapped is different from the block stored in SM; (*iii*) *valid bit*, block is valid and shared; (*iv*) *modified bit*, the PE modified its exclusive copy of the block.

The directory-based MSI protocol ensures the coherence between the SM and the cache memories. In the MSI protocol any block can be in one of 3 states: *modified* – block has been modified and is different from SM; *shared* – block is identical to the one stored in the SM but copies may exist in other caches; *invalid* – block data is not valid. Our work adopts a hybrid implementation of the protocol, being part of it implemented in hardware (in the cache controller) and part in software (in the *microkernel*).

The cache controller is responsible for: (*i*) detecting and signaling hit/miss when the address accessing the cache changes; (*ii*) updating the tag memory; (*iii*) executing read and write operations.

The microkernel is responsible for: (*i*) exchanging messages with the shared memory; (*ii*) replacing blocks when necessary; (*iii*) configuring the DMA; (*iv*) handling write-back operations. Next, memory operations are described, respectively, in cache and shared memory perspective.

A. Cache Memory

A read operation of a given block of the cache address space may result in:

- read hit: data read is returned to the processor;
- read miss in an invalid block: the *microkernel* sends a read request to the SM;
- read miss in a dirty block: the microkernel first executes the write-back operation and then sends a read request to the SM.

A write operation on a block of the cache address space may result in:

- write in a modified block (write hit): data is written into the cache;
- write in a shared block: microkernel sends a exclusivity request for the SM and then writes data into the cache.
- if tags are different (miss):
    - write in a invalid or shared block: *microkernel* sends a *read with exclusivity* request to the SM, receives the new block and writes data into the cache;
    - write in a dirty block: *microkernel* sends a write-back of the modified block and proceed as the previous case.

Except the read/write hit situations, the task that started the read/write operation remains blocked until the end of the operation.

B. Shared Memory

The SM may serve 4 requests: (*i*) read block; (*ii*) read block with exclusivity; (*iii*) write block; (*iv*) block exclusivity.

As the SM is shared among all PEs, it might receive simultaneous requests, which are serialized. To serialize operations, a FIFO-ordered buffer is implemented in the NI to store incoming flits that compose the request packets. After receiving a request, the NI notifies the SM controller.

In case of a read request, the status of the block in the directory might be in *shared* or *modified* state:

- Shared state: the block is read from the SM and a packet is sent to the requesting PE with a copy of the block. This packet is 260-flit wide.
- Modified state: the controller sends a write-back request to the PE holding exclusivity on the given block. The address of this PE is obtained by reading the directory. The PE holding exclusivity receives the write-back request and then assembles a packet with the block contents, which is sent to the SM. Only after updating the block at the SM (*write-back*), the memory controller can send the block to the requesting PE. These operations are illustrated in Figure 3(a). The performance of this operation can be optimized if multicast is employed. If the PE holding the block in modified state can send simultaneously this block to the SM and to the PE requesting it, the traffic in the network is reduced (one 260-flits instead two), and the SM controller does not spent time re-reading the data to retransmit the block. This new situation is illustrated in Figure 3(b).

When a read with exclusivity is requested, the read process is the same as before, being the block set to *modified* at the end of the reading process. The read in a modified block can also be optimized, through direct transmitting the block from the PE holding exclusivity to new PE.

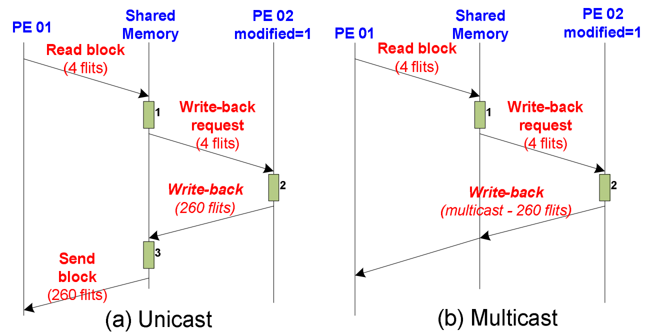

Figure 3 – Sequence diagram for a read operation of an exclusive block.

All writes in the SM are issued from write-back operations. However, PEs must acquire exclusive access on blocks, through a block exclusivity request to the SM, to write new data in the cache. An exclusivity request may access a block in *shared* or *modified* state:

- Block is only shared with the requesting PE: the modified bit of the block is asserted and a grant packet is sent to the requesting PE.
- Shared with more than one PE: the memory controller sends an invalidation message to all processors currently sharing the block. In unicast-only NoCs, a unicast packet must be sent for each PE. Figure 4(a) shows a situation where PE 01 requests exclusivity of a given block that is shared in the SM. Two other PEs are currently holding a copy of this block. Therefore, an invalidation message is sent to PE 02 and PE 03. The traffic generated on the NoC increases according to the number of PEs sharing the block. Figure 4 shows a scenario where multicast is exploited. In this case, several processors receive the invalidation message and the SM only needs to multicast one packet.
- Modified state: write-back from the PE holding the exclusivity to the PE requesting the block.

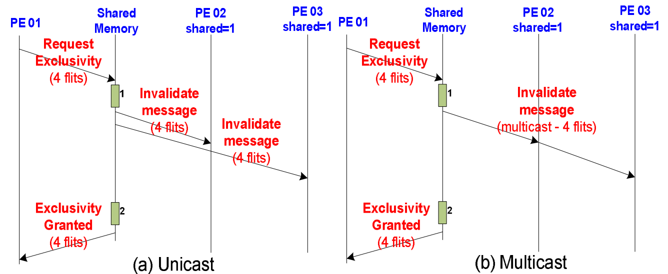

Figure 4 – Sequence diagram for a request of exclusivity on a shared block.

V. RESULTS

To evaluate the benefits of multicast messages, two different implementations of the HeMPS MPSoC were simulated in RTL-level using the ModelSim simulator. One implementation uses only unicast messages to implement the cache-coherence protocol, whilst the other one uses multicast messages. The platform used as a case study is configured as

shown in Figure 5: 3x2 NoC mesh topology, containing 5 Plasma-IPs (1 master and 4 slaves) and one SM.

A. *Read Miss in a modified block*

In this case study, PE03 is executing a given user application, which tries to read block 0 from the SM. This block is not loaded in its local cache (cache miss). Therefore, a read request is sent to the SM (arrows 1 and 2 - Figure 5). As block 0 is in modified state, a write-back request is sent to PE00, which holds exclusivity on block 0. The next actions vary according to the implementation.

In the unicast version, PE00 first sends a copy of block 0 to SM (arrow 4), which in turn, updates the block in the memory bank, updates the directory, and then sends a copy of the updated block to PE03 (arrows 5 and 6). This operation consumes 1027 clock cycles to be executed. In the multicast version, PE00 sends a multicast message to the SM and PE03 (arrows 4, 5 and 6), decreasing the number of clock cycles to execute the complete operation to 703. It represents a reduction of 32% in the number of clock cycles required to execute the complete operation. Note that a small NoC (3x2) was used in this scenario. Increased gains are expected in larger NoCs, due to the network latency, which is a function of the number of hops.

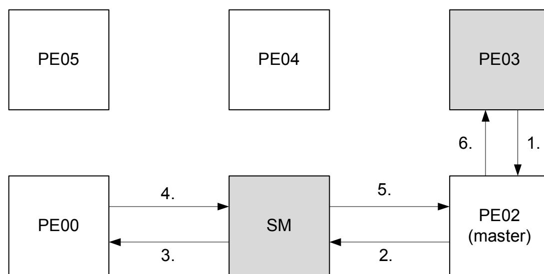

Figure 5 – Read miss in a modified block.

B. *Write Miss in a shared block*

In this case study, PE03 tries to write in block 0 of the shared memory, which is shared with PE00, PE04 and PE05. Therefore after reading the block from the SM, PE03 must acquire exclusivity on the block. To grant exclusivity of block 0 to PE03, the SM first sends an invalidation message to PEs sharing the block. The next actions vary according to the implementation.

In the unicast implementation one invalidation message is sent to each PE sharing the block, consuming 719 clock cycles to finish the overall operation. Each invalidation packet has 4 flits, consuming 15 clocks cycles to be transmitted between the SM and the PE when the distance is one hop. Each additional hop adds 5 clock cycles in the latency. Therefore, as the number of PEs increases, the latency of this operation also increases. Using multicast, this operation takes 691 clock cycles, which represents an improvement of 4%. Besides a small improvement observed in this experiment, larger NoCs and/or a higher sharing degree may results in higher gains.

C. *Summary of results*

Table 1 summarizes the total number of clock cycles required to execute a given memory operation according to the block state in the SM and the used message type.

Table 1 – Number of clock cycles per operation.

Operation	Unicast	Multicast	Gain
Read HIT / Write HIT	3	3	-
Read MISS (shared block)	638	638	-
Read MISS (modified block)	1027	703	32%
Write MISS (shared block)	719	691	4%
Write MISS (modified block)	1108	756	32%

A read/write hit takes 3 clock cycles, after putting the address in the memory address bus. The read miss operation in a shared block (3rd Table line), consumes the same number of clock cycles, regardless the message type. In this situation only a read request is sent to the SM, which sends back a message containing a copy of the block to be read. The remaining operations (4th to 6th Table lines) are optimized when multicast routing is available.

VI. CONCLUSIONS

The increased number of PEs and functionalities implemented in MPSoCs, result on a growth of the required memory for these devices. As the memory organization is one of the most crucial components of the MPSoC architecture, optimized cache-coherence protocols are required.

Results revealed the effectiveness to employ low-level NoC services, as multicast, to optimize the cache coherence protocol. The next natural step is to evaluate the performance of such optimizations using benchmarks as SPLASH-2. The observed gains, up to 32% in some memory operations, were obtained in a small MPSoC (3*x*2). For larger system configurations, the decrease in latency is sensitive to the task mapping and to the distance of the PE where a given task is mapped to the SM.

It is also important to highlight the use of the RTL modeling. The RTL modeling enables to get accurate results in terms of clock cycles due to the low-level NoC and processor modeling, and therefore the traffic transmitted through the network becomes subject to congestion and burstiness effects.

Future work includes: (*i*) evaluation of performance gains using a system with at least 25 PEs; (*ii*) use of parallel benchmarks to characterize performance; (*iii*) evaluate power/energy reduction; (*iv*) extend the number of memory-IPs in the memory hierarchy, resulting in a distributed SMs.

ACKNOWLEDGMENTS

The Authors acknowledge the support of CNPq, projects 301599/2009-2 and 133526/2010-0, and FAPERGS project 10/0814-9.

REFERENCES

[1] Millberg, M.; Nilsson, E.; Thid, R.; Kumar, S.; Jantsch, A. *The Nostrum backbone - a communication protocol stack for Networks on Chip*. In: VLSI Design, 2004, pp. 693 – 696.

[2] Kandemir, M.; Dutt, N. *Memory Systems and Compiler Support for MPSoC Architectures*. Multiprocessor Systems-on-Chips, Kluwer Academic Publishers, pp. 251–281, 2005.

[3] Silva, G. G. B.; Barcelos D.; Wagner, F. R. *Performance and Energy Evolution of Memory Hierarchies in NoC-based MPSoCs under Latency*. In: IFIP VLSI-SoC, 2009.

[4] Tota, S.V.; Casu, M.R.; Roch, M.R.; Rostagno, L.; Zamboni, M. *MEDEA: a hybrid shared-memory/message-passing multiprocessor NoC-based architecture*. In: DATE, 2010, pp.45-50.

[5] Jerger, E.; Peh, L., Lipasti, M. H. *Virtual tree coherence: Leveraging regions and in-network multicast trees for scalable cache coherence*. In: MICRO, 2008, pp. 35-46.

[6] Chtioui, H.; Atitallah, R. B.; Niar, S.; Dekeyser, J.; and Abid, M. *A Dynamic Hybrid Cache Coherency Protocol for Shared-Memory MPSoC*. In: EUROMICRO, 2009, pp. 3-10.

[7] Petrot, F., Greiner, A., and Gomez, P. *On Cache Coherency and Memory Consistency Issues in NoC Based Shared Memory Multiprocessor SoC Architectures*. In: EUROMICRO, 2006, pp. 53-60.

[8] Ophelders, F., Bekooij, M. J., Corporaal, H. *A tuneable software cache coherence protocol for heterogeneous MPSoCs*. In: CODES+ISSS, 2009, pp. 383-392.

[9] Bolotin, E.; Guz, Z.; Cidon, I.; Ginosar, R.; Kolodny, A. *The Power of Priority: NoC Based Distributed Cache Coherency*. In: NOCS, 2007. pp. 117-126.

[10] Yuang, Z.; Lu, Z., Jantsch, A.; Li, L.; Gao, M. *Towards Hierarchical Cluster based Cache Coherence for Large-Scale Network-on-Chip*. In: DTIS, 2009, pp. 119-122.

[11] Carara, E., Oliveira, R., Calazans, N., Moraes, F. *HeMPS - a Framework for NoC-based MPSoC Generation*. In: ISCAS, 2009, pp.1345-1348.

[12] Carara, E.; Moraes, F. *Deadlock-Free Multicast Routing Algorithm for Wormhole-Switched Networks-on-Chip*. In: ISVLSI, 2008, pp. 341-346.

[13] Samman, F.; Hollstein, T.; Glesnet, M. *Multicast Parallel Pipeline Router Architecture for Network-on-chip*. In: DATE, 2008, pp. 1396-1401.