Managing QoS Flows at Task Level in NoC-Based MPSoCs

Everton Carara, Ney Calazans, Fernando Moraes
Faculdade de Informática - Pontifícia Universidade Católica do Rio Grande do Sul, PUCRS- Porto Alegre, Brazil
{everton.carara, ney.calazans, fernando.moraes}@pucrs.br

Abstract—**The use of NoCs in complex MPSoCs is a reality in academic researches and industrial designs. A lot of research effort has been conducted in the last years in NoC and MPSoC designs, but few works address the gap between the NoC *infrastructure* and the MPSoC *software applications*. An important issue in MPSoC design is QoS, since applications running in such systems may have tight timing constraints, as video processing or fast communication protocols. This work bridges the hardware/software gap, exploring the integration of low-level NoC services into an application programming interface (API). Such API hides the interconnection complexity from programmer and provides efficient design space exploration to meet the QoS application requirements. Results shows that, even with the huge available bandwidth offered by NoCs, such interconnection architecture is not capable to meet QoS constraints when flows compete for common resources inside the NoC. Using the priority scheme developed in this work, applications executing in the MPSoC achieve the performance requirements. This work highlights the need to integrate NoC and MPSoC design efforts in a unified framework. (Abstract)**

Keywords-MPSoC; NoC; QoS; API (key words)

I. INTRODUCTION

Multiprocessor systems-on-chips (MPSoCs) provide a huge design space exploration for applications with high computational demands. MPSoCs are used in applications such as networking, signal processing, and multimedia. Increasing its programmability makes them more flexible, allowing its use in a wide range of digital systems. In this way, the MPSoC lifetime increases, reducing the price for the final consumer.

Since the platform computational power is distributed in several processing elements (PEs), its synchronization and message passing have a crucial role in the system performance. As the number of PEs trends to increase to dozens in a near future, an unscalable interconnection architecture, such as traditional busses, are not recommended to be used in such systems. Networks-on-chip (NoCs) support the communication requirements of modern MPSoCs, due to features as scalability, QoS support, parallel transactions, and high aggregated throughput.

Increasing MPSoCs flexibility also imply new applications added to the systems at run time. Therefore, design time approaches to support QoS, such as network dimensioning, reduce the NoC-based MPSoC flexibility, since the NoC resources cannot be managed. To support QoS in a dynamic environment, applications should have access to the NoC services. Thus, the programmer can manage the NoC resources to meet the application requirements at run-time.

This paper shows the need to expose NoC services at the task level in NoC-based MPSoCs. A complete system is presented, enabling the management of QoS flows through a dedicated API.

This paper is organized as follows. Section II presents related work in resource management in NoC-based MPSoCs. Section III gives an overview of the target NoC-based platform. Section IV describes the implemented support to QoS in the reference NoC. Section V describes the QoS integration from the NoC physical level up to the task level. Section VI presents evaluation results. Finally, Section VII presents conclusions and directions for future work.

II. RELATED WORK

With the shifting in the interconnection architecture from busses to NoCs, modern MPSoCs need to jointly manage computation and communication resources to ensure QoS to specific flows. The abstraction of the communication or the computation architectures to higher abstraction levels (e.g. through an API), hides the hardware complexity, allowing the system programmer to explore the design space in an efficient way.

The Tilera TILE64 [1] MPSoC consists of an 8x8 grid of tiles connected by five overlapped 2D mesh NoCs (iMesh). To take advantage of the whole bandwidth afforded by the on-chip integration of multiple mesh networks, Tilera provides a C-based user-level API library called iLib. There are two broad categories of communication in iLib: socket-like channels for streaming algorithms and a MPI-like message passing for ad hoc messaging. iLib provides several channel APIs, each optimized for a different communication needs such as low latency and high throughput. Through several communication primitives, it lets the programmer to use the best communication interface for the application being developed.

Winter and Fetteweis [2] present a global communication resource allocator working at the task level. The Guaranteed Traffic (GT) provided by the employed NoC is managed by a unity called NoC Manager (NoCM).

This unit finds, allocates and releases channels between two PEs, providing deterministic latency and bandwidth. NoCM has knowledge about all links in the MPSoC, and where they are available or not. As soon as a PE needs a GT path to another PE, it requests a path reservation to the NoCM. This, in turn, performs the resources allocation and notifies the PEs about the connection establishment. To speed up resources allocation, the NoCM has direct connections with all NoC routers, which is an *incompatible* design decision considering the distributed NoC paradigm.

Moreira et al. [3] present a similar resource allocation approach based on a global unit. For each set of tasks belonging to an application, resource budgets are computed offline (compilation time) such that the application meets the timing constraints. For hard real-time applications, an exhaustive temporal analysis is performed to determine these budgets. For soft real-time applications, a combination of temporal analysis and simulation may be used. When an application is requested to start, resources that meet the required resource budgets have to be found by the resource allocator. To meet the application constraints the resource allocator ensures: (*i*) *admission control* – an application is only allowed to start if the system can allocate upon request the resource budget it requires to meet its timing requirements; and (*ii*) *guaranteed resource provisions* - the access of a running application to its allocated resources cannot be denied by any other application.

Pastrnak et al. [4] describe a hierarchical QoS model for managing multimedia applications running on a MPSoC. The target application is a MPEG-4 shape-texture decoder that is fully object based, using arbitrary object shapes. The work considers a class of QoS systems that relies on predicting the execution times of the application at run-time, while also taking into account the data dependencies. The architecture of the proposed QoS concept is based on two negotiating managers, instead of a conventional single resource manager. A *Global QoS manager* controls the total system performance involving all applications and a *Local QoS manager* controls an individual application within the assigned resources.

The trend in MPSoCs design is to have dozens of PEs in a near future. Therefore, centralized approaches to ensure QoS to flows is a non scalable method, inappropriate to be used in large systems. An exception to centralized implementation is the Tilera MPSoC, which uses a distributed approach. However, the article describing the Tilera architecture does not present the iMesh QoS features and the integration of the NoC services at the task level. Several mechanisms must be used jointly to ensure QoS, as task mapping, resource reservation, traffic monitoring and task migration.

The *contribution* of this work is to expose NoC features, as a priority scheme to transmit packets, up to the task level, to ensure QoS to specific application flows. This paper is a starting effort on bridging the gap between NoC *services* and MPSoC *software tasks* aiming to increase the overall system programmability.

III. MPSoC Architecture Overview

Our target architecture is a homogeneous NoC-based MPSoC platform called HeMPS [8]. Figure 1 presents a HeMPS instance using a 2x3 mesh NoC. The main hardware components are the HERMES NoC [6] and the mostly-MIPS processor Plasma [9]. A PE, called Plasma-IP, wraps each Plasma processor, attaching it to the NoC. This IP also contains a private memory, a network interface, and a DMA module.

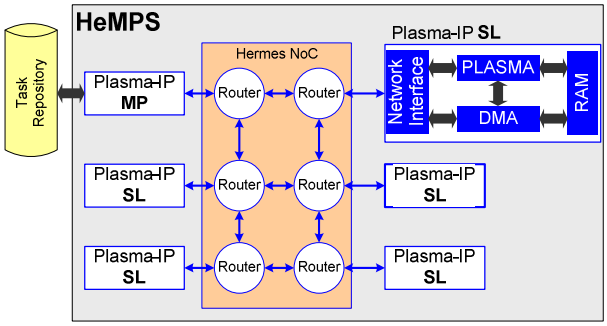

Figure 1 - HeMPS instance using a 2x3 mesh NoC.

Typical applications running in MPSoCs, such as multimedia and networking, often present dynamic workload needs. This implies a varying number of tasks running simultaneously, and their number or load often exceeds the available resources. To tackle this issue, HeMPS assumes: (*i*) applications are modeled using task graphs; (*ii*) only the kernels (the kernel is in effect a minimal operating system, one per Plasma) are initially loaded into the system. All application tasks are stored in an external memory, named *task repository*. Each application has at least one *initial task*, being the remaining tasks requested by the initial task, or other task already loaded into the system. Figure 2 shows a synthetic application, modeled using a task graph, with one initial task (task 0).

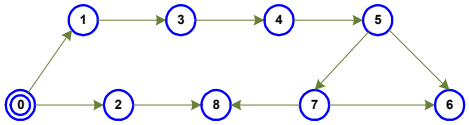

Figure 2 – Task graph modeling a synthetic application, being '0' the *initial task*.

The system contains a *master processor* (Plasma-IP MP), responsible for managing system resources. This is the only processor having access to the task repository. When HeMPS starts execution, the master processor allocates initial tasks to the *slave processors*. The user defines the mapping of the initial tasks. The remaining tasks are mapped at run time according to some mapping heuristic, targeting e.g. congestion minimization.

Each slave processor (Plasma-IP SL) runs a kernel, supporting *multitasking* and *task communication*. The kernel segments memory in pages, which it allocates for itself (first pages) and tasks (subsequent pages). The memory pages are protected and all communication among tasks occurs through *message passing*. During execution, tasks are

dynamically loaded from the task repository to the slave processors on demand. In addition, resources may become available when a given task finishes execution. Such dynamic behavior enables smaller systems, since only those tasks effectively required are loaded into the system at any given moment.

To achieve high performance, the Plasma-IP architecture separates communication from computation. The network interface and DMA modules are responsible for sending and receiving packets, while the Plasma processor performs task computation and wrapper management. The local RAM is a true dual port memory allowing simultaneous processor and DMA accesses, avoiding extra hardware for elements as *mutex* or cycle stealing techniques.

IV. QoS Support at the NoC Level

Most NoC implementations only provide support to best effort (BE) services [10], even those proposed by NoC companies like Arteris [11]. BE services guarantee delivery of all packets from a source to a target, but provide no bounds for throughput, jitter, or latency. This kind of service usually assigns the same priority to all packets, leading to unpredictable transmission delays. The term Quality of Service (QoS) refers to the capacity of a network to control traffic constraints to meet design requirements of an application or some of its specific modules. Thus, BE services are inadequate to satisfy QoS requirements for applications/modules with tight performance requirements, as in the case of multimedia streams. To meet performance requirements and thus guarantee QoS, the network needs to include specific characteristics at some level in its protocol stack. Accessing the relative priority and requirements of each flow enables an efficient assignment of resources to flows [12]. Current NoC designs employ at least one of three methods to provide QoS: (*i*) dimensioning the network to provide enough bandwidth to satisfy all IP requirements; (*ii*) providing support to circuit switching for all or selected PEs; (*iii*) making available priority scheduling for packet transmission.

The Hermes NoC employs a 2D mesh topology. Routers have input buffers, a control logic shared by all router ports, an internal crossbar and up to five bi-directional ports. The main modification in the original router to give support to some QoS level is the duplication of the physical channels (bi-directional ports), which connect neighbor routers. The physical channels were duplicated in all four directions (North, South, East and West), resulting in a router supporting up to nine bi-directional ports. Thus, priority mechanisms can be used to differentiate flows. Channel replication was preferred to virtual channels due to its as smaller area overhead, increased router bandwidth and reliability, and simpler implementation [7]. Besides, priority mechanisms based on virtual channels lack on QoS support when more than one high priority packet requires the same output port, since the packets share the link bandwidth. Figure 3 illustrates the QoS router architecture.

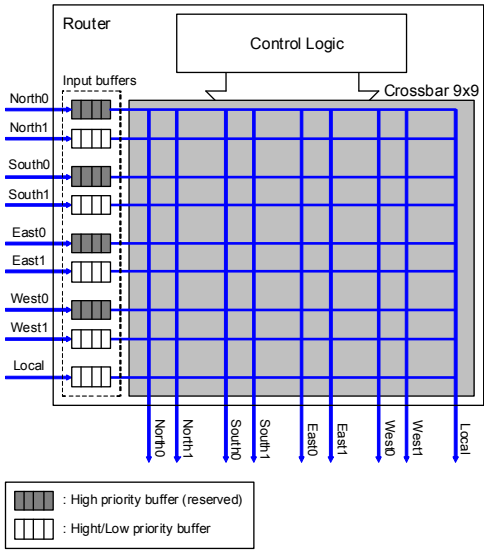

Figure 3 – QoS router architecture with duplicated physical channels.

In the present work, the implemented QoS support relies on *fixed priorities* mechanism. Two prioritized best effort (BE) traffic classes are distinguished inside the NoC: (*i*) *high* priority packets and (*ii*) *low* priority packets. One physical channel (*channel 0*) is *reserved* to transmit exclusively high priority packets, whereas *channel 1* may transmit both packet classes. Sharing one of the two physical channels provides higher support to high priority traffics, since two high priority flows can share common paths. The priorities mechanism provides a *soft guaranteed* service (latency and bandwidth) to high priority traffics through a *virtual resource reservation*.

The packets are injected into the NoC through the router Local port (not duplicated), which is *shared* by high and low priority packets. To differentiate incoming flows, besides the target address, the packet header flit has a *priority bit*. Each time a new packet enters the router, the *control logic* reads the priority bit and executes the routing algorithm and physical channel allocation. A high priority packet allocates the first free physical channel available in the direction selected by the routing algorithm, whereas a low priority one can allocates only the physical channel 0 in the selected direction.

In this architecture, when more than two flows compete for common paths inside the NoC, QoS guarantees are affected, reducing the application performance. In fact, NoCs employing priority mechanism to ensure some QoS level tend to perform like BE NoCs as the amount of higher priority traffic increases. Priority mechanisms are simple and low cost solutions to ensure QoS for traffics with no rigid time constraints.

A second modification in the original Hermes NoC, not related to QoS, is the routing algorithm. Packets are routed according to the *Hamiltonian* routing algorithm, which supports the *dual path multicast* [13] enabling the transmission of multicast and broadcast messages.

V. INTEGRATING QOS SUPPORT FROM NOC TO TASK LEVEL

Computational systems are layered in different abstraction levels in an effort to master its complexity. Each layer has a given functionality and communicates with adjacent levels (above/below it). Therefore, each layer uses the services of the lower layers and supplies them to the upper layer. Figure 4 shows the HeMPS system layers and the corresponding entities.

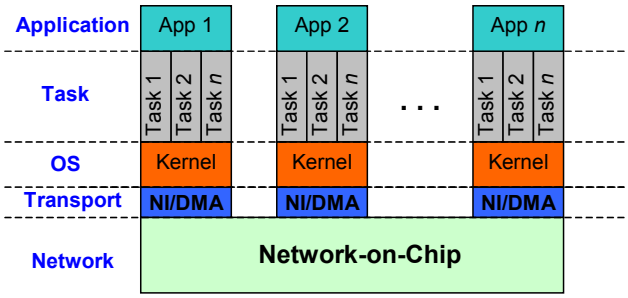

Figure 4 – HeMPS system layers.

Each layer is responsible for:

- *Application layer*: this level contains the description of each application to be mapped into the system. Each application is described as a task graph;
- *Task layer*: this level contains the description of each task, using the synchronization and communication primitives available in the HeMPS API, provided by the OS layer;
- *OS layer*: is a set of drivers responsible for task scheduling, task loading, memory management, DMA management, packet assembly/disassembly, and provides to the task layer the HeMPS API;
- *Transport layer*: execute packet injection/reception and flow control;
- *Network layer*: responsible for BE and QoS data transmission.

The C-based HeMPS API provides two MPI-like communication primitives to allow inter-tasks message passing: (*i*) *Send* and (*ii*) *Receive*. These two primitives implement the message passing between tasks, which can be located in the same processor or different ones. The local communication is performed through the kernel area memory while the remote communication uses the NoC. Each kernel has a *task-table* with the location of local and remote tasks. Tasks location is transparent to the programmer, and the kernel is responsible for setting the NoC packet address where the target task is located.

The *Send* primitive, available at the task level, has a dedicated parameter to set the message priority. The *Send* syntax is:

*Send(Message *msg, int target_task, int priority)*

where:

- *message *msg:* points a message structure in the task memory;
- *int target_task:* the message target task identifier;
- *int priority:* the message priority (0-HIGH or 1-LOW).

Each time the *Send* primitive is called, the kernel is scheduled to execute. The raw message is copied to the kernel area and tagged with a NoC header. To set the header with the target address, the kernel access its task table and searches the target task location using its identifier (*Send* 2nd parameter). Then, the NoC packet priority is set filling the *priority bit* in the header with the 3rd *Send* parameter. Therefore, a *memory image* of the NoC packet is created. This is the only function of the *Send* primitive: create a packet image in the kernel area. Once this image is created, the task resumes its execution. This approach allows the parallel task execution and message transmission.

A *Receive* primitive, executed by the consumer task, fires a packet transmission. The *Receive* primitive generates a *message request* packet to the producer task. An incoming message request interrupts the executing task and the kernel is scheduled. Then, the kernel configures the DMA module (*DMA_Send()*) to transmit the packet and the interrupted task is rescheduled. Since a true dual-port memory is used, packet transmission and task execution are carried out in parallel. All the implicit inter-kernel communication (*control messages, e.g. message request*) are transmitted in high priority packets. These messages are commonly short and do not disturb the high priority flows. Figure 5 illustrates the *Send* sequence diagram.

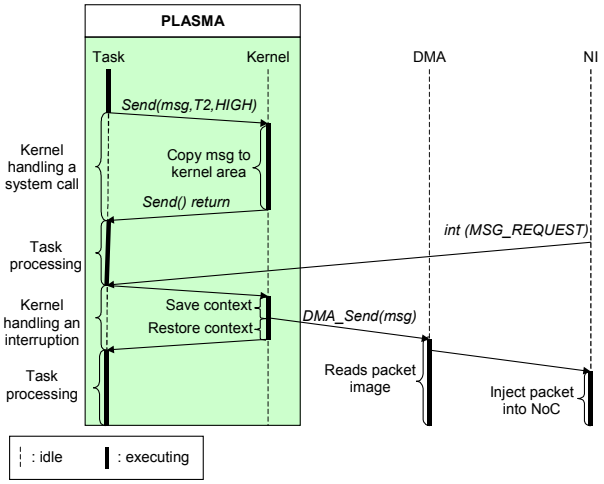

Figure 5 – *Send()* sequence diagram.

Figure 6 illustrates the Plasma-IP modules interaction during a *Send-Receive* processing. The illustrated steps are:

1. The Plasma processor copies the message (*msg*) from the task memory area to the kernel memory area. The packet image is created in the kernel area.
2. The DMA module is programmed to transmit the packet when the NI interrupts the Plasma-IP due to a *message request*.
3. DMA module reads the packet image from the kernel memory area and sends it through the NoC using the NI

module (*the processor is not disturbed in this step*). As the priority information is part of the packet header, there is no extra DMA pinout to inform the NI about the packet priority.

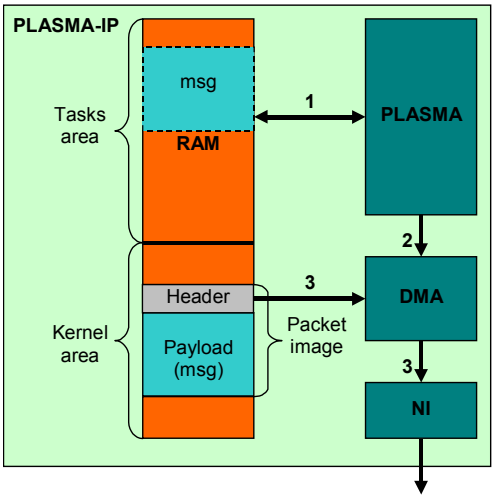

Figure 6 – Modules interaction.

This inter-task communication scheme, non-blocking writing (if there is available memory space in the kernel area), and blocking reading has the following advantages:

- A producer task may continues its execution, independently of the consumer status;
- Network traffic is reduced, since the consumer task enters in wait state after a *Receive* execution until the producer task has data to transmit (there is no polling);
- If a blocking *Send* and a non-blocking *Receice* was chosen, messages must be stored in the consumer side, or they may block the NoC if the consumer side has no available memory space. Also, if the consumer task is not yet allocated in the system, the producer task stalls for a large amount of time (until the consumer task be allocated).

VI. RESULTS

Two simulation scenarios are evaluated in a 4x4 HeMPS instance. In the first one all involved flows are generated by Plasma-IPs, simulating a homogeneous MPSoC. The second one simulates a heterogeneous MPSoC with two traffic generators flows disturbing the Plasma-IP QoS flows. To speed up simulations, Plasma processor and RAM memories are modeled in SystemC (cycle accurate simulation), whereas the remainder system is described in synthesizable VHDL RTL.

A. Homogeneous MPSoC QoS evaluation

The goal of the first simulation scenario is to show the NoC behavior as the number of QoS flows increases. Figure 7 presents the spatial distribution of the flows. Flows F1 and F2 (dotted lines) are QoS flows (higher priority) and the remainders (F3, F4, F5 and F6) are disturbing flows (lower priority). A flow is composed by burst packets (524 flits) interleaved by idle times. The injection rate of flows F1 and F2 is 30% of the link bandwidth, and the disturbing flows has an average injection rate of 18.5%. All flows are generated by the software executing in the Plasma processors.

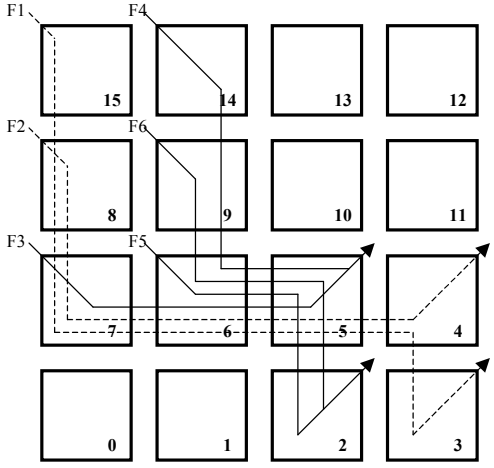

Figure 7 – Spatial distribution of flows. F1 and F2 are QoS flows. F3, F4, F5 and F6 are disturbing flows.

Figure 8 presents the throughput of each flow, as the number of QoS flows increases. Since the disturbing flows pairs F3-F4 and F5-F6 have the same target, the calculated pair throughput is the target throughput divided by 2. Initially only F1 and F2 are QoS flows and its throughputs are close to the injection rate. As the disturbing flows become QoS flows, the throughput of flows F1 and F2 reduces. When all disturbing flows became QoS flows, all six flows have the same priority and the NoC acts as a BE NoC. In this condition, F1 and F2 throughput decays from 29.91% and 28.83% to 21.95% and 20.91%, respectively. BE NoCs underutilize its resources, since there is no effective QoS management.

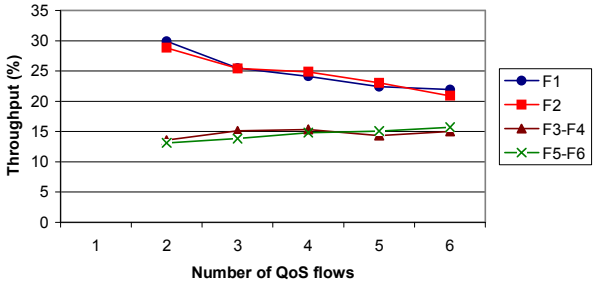

Figure 8 – QoS flows deterioration.

This experiment demonstrates that:

- When the number of QoS flows competing for resources do not exceed the designed QoS support, the *soft* QoS is guaranteed (in this experiment the NoC supports up to 2 competing QoS flows);
- The huge bandwidth provided by NoCs is not sufficient to ensure QoS when the number of concurrent QoS flows increases, even in an homogeneous scenario, with all flows being generated by processors (remember that each processor has the kernel overhead and sequential

instruction execution, which reduces the injection rate);

- Even do not exceeding the total NoC bandwidth, the flows may disturb each others;
- The HeMPS API ensures *soft* QoS to the flows from the task level. Using it, flows F1 and F2 achieved a throughput equal 29.91% and 28.83% respectively.

B. *Heterogenous MPSoC QoS evaluation*

The second experiment shows that the QoS support, based on priorities, can efficiently guarantee throughput to a flow even under disturbing flows with high injection rates. Figure 9 presents the spatial distribution of the flows. The QoS flow F1 (dotted line) is generated by a Plasma-IP while the disturbing flows F2 and F3 (low priority) are generated by traffic generators. F1 injection rate is 30% of the NoC link bandwidth, and flows F2 and F3 have an injection rate of 100%.

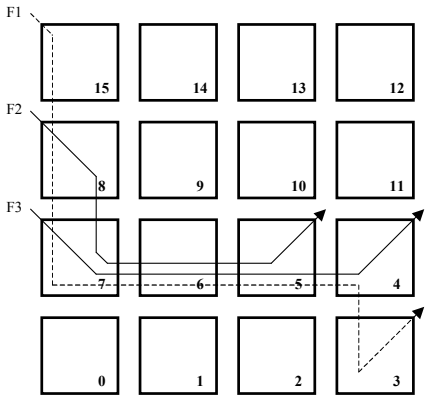

Figure 9 – Flows spatial distribution. F1 is QoS flow. F2 and F3 are disturbing flows.

In this scenario, considering a BE NoC, the F1 throughput is 10.5%. Considering a QoS NoC, QoS flow F1 achieves a throughput of 29.8%. This result is due to the virtual reservation, where QoS resources are available exclusively for high priority packets.

VII. CONCLUSIONS AND FUTURE WORK

A lot of research effort has been conducted in the last years in NoC and MPSoC designs, but few works address the gap between the NoC *services* and the MPSoC *software tasks*. This research work presented the design of a simple priority mechanism to ensure *soft* QoS for the MPSoC at the NoC level. This feature is integrated at the kernel level (HeMPS API) and it is available to application tasks executing in the MPSoC. This integration increases the overall system programmability and enables system programmers to manage QoS flows at the task level. Thus, design space exploration can be accomplished in an efficient way, due to the abstraction of interconnection architecture details.

Results have shown that even with duplicated physical channels (higher bandwidth), the concurrence for resource may degrade the performance of QoS flows. Therefore, having a massive amount of interconnect resources is not sufficient to provide QoS, if these cannot be effectively utilized. This highlights the need to integrate NoC and MPSoC design efforts in a unified framework.

Future work includes the addition of new services in the NoC, such as multicast/broadcast and circuit switching to provide hard QoS to flows with tight time constraints. As presented in this paper, these services will be exposed at task level, enriching the HeMPS API, and increasing the system programmability.

VIII. ACKNOWLEDGMENTS

This research was supported partially by CNPq (Brazilian Research Agency), projects 300774/2006-0 (PQ) and 471134/2007-4 (Universal).

IX. REFERENCES

[1] Wentzlaff, D.; et al "*On-Chip Interconnection Architecture of the Tile Processor*". IEEE Micro, 27(5), Sept.-Oct. 2007, pp. 15-31.

[2] Winter, M.; Fettweis, G. P. "*A Network-on-Chip Allocator for Run-Time Task Scheduling in Multi-Processor System-on-Chip*". In: Euromicro Conference on Digital System Design Architectures, Methods and Tools, 2008, pp. 133-140.

[3] Moreira, O.; Mol, J.; Bekooij, M. *"Online Resource Management in a Multiprocessor with a Network-on-Chip"*, In: Symposium on Applied Computing, 2007, pp.1557-1564.

[4] Pastrnak, M.; et al "Novel QoS Model for Mapping of MPEG-4 coding onto MP-NoC", In: International Symposium on Consumer Eletronics, 2005, pp. 93-98.

[5] Joven, J.; Carrabina, J.; et al *"xENOC – An eXperimental Network-on-Chip Environment for Parallel Distributed Computing on NoC-based MPSoC Architectures*". In: Euromicro Conference on Parallel, Distributed and Network-Based Processing, 2008, pp. 141-148.

[6] Moraes, F.; et al. "*HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip*". Integration the VLSI Journal, 38(1), Oct. 2004, pp. 69-93.

[7] Carara, E.; et al "*A New Router Architecture for High-performance Intrachip Networks*", Journal of Integrated Circuits and Systems, 3(1), Mar. 2008, pp. 23-31.

[8] Carara, E. et al "*HeMPS - A Framework for NoC-Based MPSoC Generation*", In: International Symposium on Circuits and Systems, 2009 (to be published)

[9] OpenCores, www.opencores.org.

[10] Rijpkema, E.; Goossens, K.; Rădulescu, A. "*Trade-offs in the Design of a Router with Both Guaranteed and Best-Effort Services for Networks on Chip*". In: Design, Automation and Test in Europe, 2003, pp. 350-355.

[11] Arteris. "*Arteris Network on Chip Company*". 2005. Available at http://www.arteris.net.

[12] Duato, J.; Yalamanchili, S.; Ni, L. "Interconnection Networks". Elsevier Science, 2002, 600 p.

[13] Lin, X.; McKinley, P. K.; Ni, L. M. "*Deadlock-free Multicast Wormhole Routing in 2-D Mesh Multicomputers*". IEEE Transactions on Parallel and Distributed Systems, 5(8), 1994, pp. 793-804.