Achieving Composability in NoC-Based MPSoCs Through QoS Management at Software Level

Everton Carara[1], Gabriel Marchesan Almeida[2], Gilles Sassatelli[2], Fernando Gehm Moraes[1]
[1]PUCRS, Porto Alegre, Brazil, {everton.carara, fernando.moraes}@pucrs.br
[2]LIRMM, Montpellier, France, {marchesan, sassatelli}@lirmm.fr

Abstract— **Multiprocessors systems on chip (MPSoCs) have become the de-facto standard in embedded systems. The use of Networks-on-chip (NoCs) provides to these platforms scalability and support for parallel transactions. The computational power of these architectures enables the simultaneous execution of several applications, with different time constraints. However, as the number of applications executing simultaneously increases, the performance of such applications may be affected due to resources sharing. To ensure applications requirements are met, mechanisms are necessary for ensuring proper isolation. Such a feature is referred to as composability. As the NoC is the main shared component in NoC-based MPSoCs, quality-of-service (QoS) mechanisms are mandatory to meet application requirements in term of communication. In this work, we propose a hardware/software approach to achieve applications composability by means of QoS management mechanisms at the software level. The conducted experiments show the efficiency of the proposed method in terms of throughput, latency and jitter for a real time application sharing communication resources with best-effort applications.** *(Abstract)*

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

I. INTRODUCTION

The growing interest for MPSoCs lies in their ability to combine high performance and software-oriented programming. The mapping of multiple concurrent and independent applications on such platforms requires special attention to avoid inter-application interferences. These interferences arise due to resources sharing among applications that may lead to significant performance degradation. The independence degree of an application, irrespective of the presence or absence of other applications in the platform, is called [1][2]. This property aims to guarantee application requirements are met in different scenarios. As applications with different requirements may be executed simultaneously, composability must be ensured for those with time constraints (soft or hard real-time).

Composability is frequently achieved through mechanisms integrated in the Operating System (OS), such as specific scheduling polices in the case of real-time operating systems. For example, in the aerospace and automotive industry, composability is frequently ensured by not sharing the resources between applications [2].

For a NoC-based MPSoC, the services offered at the network level may be advantageously exposed at the software level to ensure composability at run-time. Most of the related work ([2]-[4]) relies in hardware designed for a fixed set of applications, therefore precluding dynamic workload (insertion of new applications into the system at run-time), an important feature of present embedded systems.

This work proposes a combined hardware/software approach to both improve composability and facilitate its management at the software level. It relies on NoC hardware QoS mechanisms exposed at the software level, making possible to ensure composability at higher abstraction levels.

This paper is organized as follows. Section II presents related works in composability and QoS. Section III overviews the homogeneous message-passing NoC-based MPSoC used in this work. Sections IV and V present respectively the hardware and software support to achieve composability. Section VI presents throughput and latency results for a real application in the presence of disturbing traffic, as well as the area overhead induced by the proposed approach. Section VII draws conclusions and gives directions for future work.

II. RELATED WORK

Kumar et al. [1] propose a resource manager (RM) responsible to control inter-applications interferences. The RM runs on one of the processing tiles and controls the usage of resources in the system. During system execution, it monitors application performance through periodically received messages. If any application is found to be running below the desired throughput, the application which has the most slack is suspended. The application is then resumed if remaining applications are running above the desired throughput. The system composability was shown through a high-level simulation model. A case study with an H.263 and a JPEG decoder demonstrates that the RM ensures that both applications are able to meet their throughput requirements. It is important to mention that the employed high-level model does not take into account the contention on the communication infrastructure, which can be the main source of inter-applications interference.

CoMPSoC [2] is a composable and predictable NoC-based MPSoC that gives special attention to the system resources shared by the applications, such as the NoC and shared memories. In this platform, the processing elements are not shared between applications because the used processor (Silicon Hive) does not support preemptive multitasking. TDM (time division multiplexing) arbiters are used to achieve application predictability and composability. The system prevents any interference between applications through reservation of hardware resources at design time.

978-3-9810801-7-9/DATE11/©2011 EDAA

Molnos et al. [3] overcome the CoMPSoC processor sharing limitation through processor virtualization. TDM scheduling is used in three abstraction levels: application, task, and processor (MicroBlaze). Each application has a constant time slice allocated to each TDM period. The application execution order in a TDM period is static and in case a given application is not ready to execute, its time slice is wasted (idle slice). In an application slice, application tasks are executed in sub-slices. Each application may have a different scheduler, and the task order inside a slice does not have to be the same at each TDM period.

Hansson et al. [4] present a composable and predictable NoC, named aelite. This mesochronous NoC architecture is based on Aethereal [5] and only offers guaranteed services. Functional scalability is obtained by completely isolating applications, and by having a router architecture that does not limit the number of connections.

Few works address QoS support at run-time across the software stack, from the application down to packet level [6]. In [7] NoC guarantees are extended to application level through decoupling buffers at the NI. The work shows how to construct a Cyclo-Static Dataflow (CSDF) that conservatively models a NoC connection. This model enables the verification of application requirements and the sizing of the decoupling buffers to meet the QoS constraints.

Works presented in [6] and [8] expose the NoC QoS features at the task level through an application programming interface (API). Using such approaches, the application programmer may set some NoC QoS parameters, such as priorities, routing and connections. This is done explicitly in the application code, increasing the system programmability, design space exploration and the software support to composability.

All reviewed works targeting composability ([1]-[4]) employ the Aethereal NoC, which provides guaranteed services through TDM based connections. Aethereal is not freely available, which makes its access from the research community difficult. The work presented in this paper uses the open-source Hermes NoC [9]. QoS mechanisms have been included in the Hermes architecture and exposed at the task level through an implemented API. Contrarily to Aethereal, which has a configuration master [10] responsible for opening/closing connections between IPs (Intellectual Property), Hermes resources can be managed at run-time by the IPs in a distributed and scalable way.

An architecture is said to be composable with respect to a specified property if the system integration will not invalidate this property once the property has been established at the subsystem level [17]. The properties addressed in this work are performance figures like throughput and jitter. *Our goal is to define composability mechanisms to be used in applications with time constraints through the management of the QoS at the software* level. Applications with no constraints are managed in best-effort mode. This work aims to ensure the proper functionality of real-time applications even if the isolation between applications is not total. In this way, a good trade-off can be set between system flexibility and applications composability.

III. HS-SCALE ARCHITECTURE OVERVIEW

HS-Scale [11] is a NoC-based homogeneous MPSoC. It is built using a distributed memory architecture, and tasks communicate with each other by using a message passing protocol. Contrary to the MPI standard, tasks are not mapped to a given processor, but shall freely move in the system according to user-defined policies to optimize performance parameters, as execution time and power consumption. Also, for scalability reasons, there is no master in the system.

The architecture is a homogeneous array of Network Processing Units (NPUs) communicating through a Network-on-Chip (described in Section IV). Each NPU has multitasking capabilities, which enable time-sliced execution of multiple tasks. This is supported by a preemptive multitasking µKernel running on each NPU. This µKernel further provides usual operating systems services such as queues, threads, semaphores and mutexes. Figure 1 shows the platform architecture.

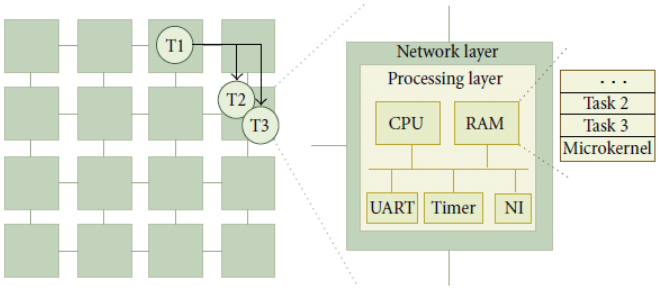

Figure 1. Structural platform view.

Incoming messages are read from the network interface (NI) and stored in the target task FIFOs (SW implemented). A target task has one FIFO for each source task it receives data from. Outgoing messages are written to the NI, which is responsible for packaging and injection in the NoC. Figure 2 presents the NPU functional view.

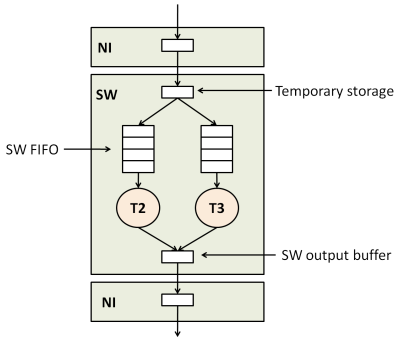

Figure 2. Functional NPU view.

The processing layer is based on a simple and compact RISC microprocessor (Plasma, instruction set compatible with MIPS-1), its local memory and a few peripherals (timer, interrupt controller and UART), as shown in Figure 1. To keep the processor as small as possible, the Plasma processor has 3 pipeline stages, no cache, no Memory Management Unit (MMU) and no memory protection support. The original version of the µKernel, as well as the RTL description of the Plasma processor, used as part of this work are freely available at open cores [12].

IV. THE HERMES NOC QOS SUPPORT

NoC designs can provide two kinds of services: best-effort (BE) and guaranteed services. 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 QoS refers to the capacity of a network to control traffic constraints to meet design requirements of an application. Thus, BE services are inadequate to satisfy QoS requirements for applications/ modules with tight time constraints, 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.

The original Hermes NoC employs a 2D mesh topology and wormhole packet switching. Routers have input buffers, a control logic shared by all router ports, an internal crossbar and up to five bi-directional ports. This NoC provides only BE services, so no bounds for any kind of performance figures can be ensured. In order to offer guaranteed services, two kinds of QoS mechanisms were implemented: (*i*) priorities, supporting soft QoS through two priorities levels (*high and low*) and (*ii*) connections, supporting hard QoS through circuit switching.

The main modification applied to the original router to enable soft QoS lies in the duplication of the physical channels (bi-directional ports). The resulting router supports up to ten bi-directional ports. As a result, priority mechanisms can be used to differentiate flows. Channel replication was preferred to virtual channels due to its smaller area overhead, increased router bandwidth, and simpler implementation [13]. Figure 3 illustrates the router architecture.

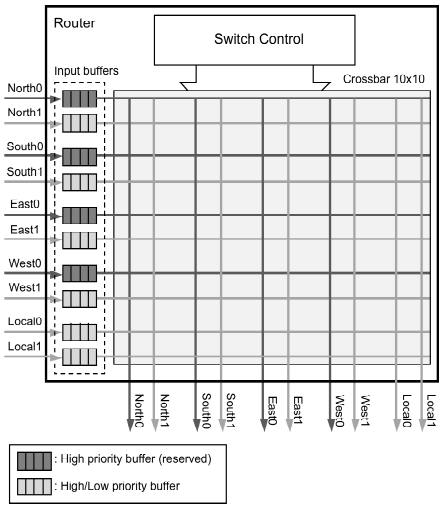

Figure 3. Router architecture with duplicated physical channels.

The soft QoS support relies on a *fixed priorities* mechanism. Two prioritized 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 the other one (*channel 1*) may transmit both packet classes. Sharing one of the two physical channels between the traffic classes increases the support to high priority traffics, because two high priority flows can be transmitted simultaneously in the same direction. The priorities mechanism provides a soft guaranteed service (latency and bandwidth) to high priority traffics through a *virtual resource reservation* (dark gray resources in Figure 3). However, when more than two high priority flows compete for common paths inside the NoC, the soft QoS guarantees may be affected. In fact, NoCs employing priority mechanism to ensure QoS tend to perform like BE NoCs as the amount of high priority traffic increases [14].

The hard QoS support is based on a *circuit switching* mechanism. A connection is established between a source/target pair and the NoC resources stay allocated throughout the entire duration of the communication. The connections are unidirectional and established/released by the source through *connection establishment/release packets*. Resources are reserved for worst case allowing the applications to achieve its maximum throughput without any interference from other communications. Bounds for performance parameters as latency, jitter and throughput are guaranteed. Since the worst-case allocation can be a drawback when applications throughput is low or paths are blocked for a long time, connections are *restricted only to channel 0*. As the NoC combines circuit and packet switching modes, one physical channel is always available for packet switching.

The NoC router differentiates incoming flows through specific fields in each packet header. When a new packet reaches the router, the *Switch Control* (Figure 3) reads the header to execute the routing algorithm and physical channel allocation or deallocation (in case of connection release packets). Figure 4 illustrates the packet structure. The first flit is the packet header, and the remaining ones are payload flits. The last packet flit is signaled by a side band signal named *eop* (end-of-packet). The header flit contains the following fields: *Service* (4 bits): connection establishment/release, switching mode; *P* (1 bit): packet priority bit ('0': high; '1': low); *Target* (8 bits): indicates the packet target IP address.

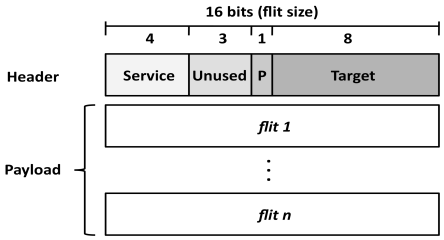

Figure 4. Packet structure.

V. ACHIEVING COMPOSABILITY AT THE TASK LEVEL

Both QoS approaches (priorities and circuit-switching) were integrated in the HS-Scale API allowing NoC resources management at the task level. The link between the API and the NoC QoS services is accomplished by the µKernel. In this way, the programmer is able to avoid interferences on real-time applications through the primitives provided by the API. The composability of a given application can be exploited from its source code. Listing 1 presents the HS-Scale API primitives to be used by programmers to control the QoS. The parameter target, present in the primitives, refers to the target task. Tasks location is transparent to the programmer, and the µKernel is responsible for resolving destination address based on mapping

tables. This approach grants a higher abstraction level to the programmer.

The non-blocking `MPISend()` primitive is used to send local/remote messages to another task (`int target`). Before transmission, the message (`void *data`) is split into multiple packets by the µKernel. Each packet has its header priority bit (Figure 4) set to the parameter `int priority`. This process effectively links the NoC soft QoS support to the task level. The priority values are constants defined as `HIGH`, `LOW` and `GT` (guaranteed throughput, used for connection). The parameter `int fifo` specifies the FIFO where the target task must store the received messages (Figure 2) and `int size` represents the message size in bytes. `MPIRcv()` is a blocking primitive used to read a message from a specific FIFO (`int fifo`).

LISTING 1 – HS-SCALE API PRIMITIVES.

```
/* Sends a message to a target task */
void MPISend(int target, int fifo, void *data,
             int size, int priority);

/* Reads a message from a FIFO */
void MPIRcv(int fifo, void *data, int size);

/* Establishes a connection */
void Connect(int target);

/* Releases a connection */
void Unconnect(int target);
```

The establishing/releasing of a connection between a source/target pair is done by the `Connect()`/ `Unconnect()` primitives, respectively. By means of these primitives, connection establishment/release packets are created and sent through the NoC. These packets perform the allocation/deallocation of NoC resources for a given connection as they are forwarding towards the target. Once the connection is established, messages can be sent using the `MPISend()` primitive setting the priority parameter as GT. In this case, the GT parameter is not used to set the packet header priority bit, but rather the service (Figure 4). The packet header sent through a GT connection is used only end-to-end and is handled by the routers as payload.

The GT connection is established at the task level (`Connect()`). As each router can handle one GT connection, only one task per-NPU can establish a connection at the time. This connection stays open until the call of the `Unconnect()` primitive, even if the task connection owner is not scheduled. If another task in the same NPU tries to establish a connection, this task will stay blocked until the connection has been released. This limitation enables, per-NPU, one task communicating with hard QoS guarantees, and the remaining tasks communicating with soft QoS guarantees.

MPSoCs execute several applications with different traffic patterns and QoS requirements. A *use-case* is a set of these applications executing simultaneously in the MPSoC, characterizing the system from the user perspective [1][15]. Assuming, for example, at most one real-time application being executed in all use-cases. Soft QoS can guarantee the composability of this application by setting all communication events as high priority, via `MPISend()` primitive. As the number of real time applications per use-case increases, soft QoS may not be enough to achieve composability, since the NoC tends to perform like a best-effort as the number of high priority flows increases. Therefore, a combination of soft and hard QoS can extend the guarantees to support several real time applications. This set of primitives allows tasks to manage the NoC resources in a distributed way, without the need of having a central resource manager.

When a given NPU is shared among several applications tasks, the operating system scheduling priorities can be used to avoid interferences from best-effort on real time applications. However, share the same NPU between real time applications tasks may harm the composability.

VI. RESULTS

In this section, the task level QoS management is applied in a scenario mixing real and synthetic applications. The real application is a synchronized audio/video decoder composed of 7 tasks (Figure 5) that requires specific timing constraints. The video pipeline decoding is executed by a MJPEG decoder split into 3 tasks (MJ1, MJ2 and MJ3) and the audio pipeline decoding is composed by an ADPCM decoder (AD) and a finite impulse response filter (FIR). An initial task called SPLIT performs the demultiplexing of the compressed audio/video streams and sends them to the respective decoder pipelines whereas the task JOIN synchronizes decoded streams. The required minimum throughput is set at 30 frames/s for video and 32,000 audio samples/s; with image-level audio/video synchronization. The synthetic applications (best-effort) are dummy tasks executing memory accesses or accessing some output device. Memory and output devices are emulated by software tasks.

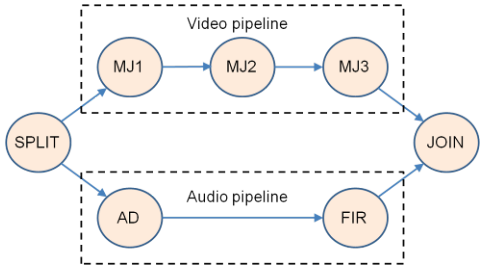

Figure 5. Application task graph.

Results were obtained on a 4x4 instance of the HS-Scale, through cycle accurate simulation. Both the NoC and the NI are described in synthesizable RTL VHDL, whereas the NPU processor employs a SystemC ISS. The first use-case contains the audio/video decoder mapped into the platform (optimal mapping, illustrated in Figure 6), and the measured throughput at the output of the JOIN task was 31.13 frames/s, value used as reference. Soft QoS is used in this use-case.

The mapping shown in Figure 6 is obtained during system initialization. In a dynamic system, new applications are frequently mapped and removed, resulting in a dispersion of the available resources (fragmentation). Therefore, an optimal mapping is rarely achievable at run-time, unless a remapping of currently running application is acceptable. The new mapped applications commonly share system resources used with

already mapped applications. Figure 7 shows a use-case that is representative of such a situation (non-optimal mapping incurring inter-application interferences). Four new applications were added, each one with two tasks: T1→MEM, T2→MEM, T3→OUT and T4→OUT. Dashed lines show the audio/video decoder data flows and solid lines show the new applications flows, according to the Hamiltonian routing algorithm [16] employed by the NoC.

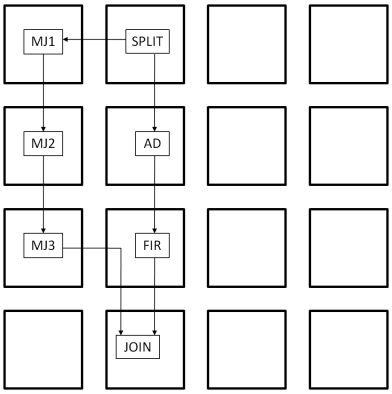

Figure 6. Optimal mapping for the audio/video decoder application.

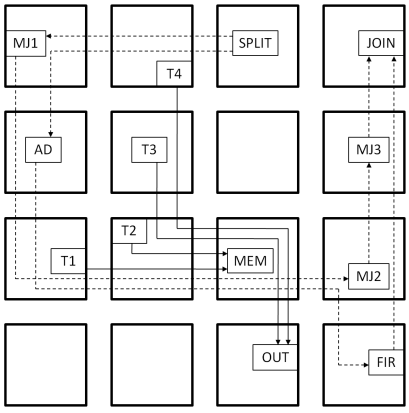

Figure 7. Audio/video decoder disturbed by the new applications.

Table 1 presents the measured audio/video decoder throughput varying the number of flows with high priority in the use-case presented in Figure 7, for six different scenarios. When only the audio/video decoder flows have high priority (scenario S1), the obtained throughput is 1.6% smaller compared to the reference one, but still guaranteeing the application requirement (30 frames/s). The QoS mechanisms isolate the real time application flows from the best-effort flows avoiding interferences. Scenarios S2 to S5 show the throughput degradation as the number of high priority flows increases. The throughput is reduced by 52% when all application flows have high priority (scenario S5). In this situation, the NoC works in best-effort mode, and the high router bandwidth is not enough to ensure the application requirements. Since the video decoder generates higher traffic rates than the audio one, it was chosen to use GT connections. This is shown in scenario S6, where the video decoder tasks communicate through GT whereas all other tasks use high priority. Note that the QoS guarantees were extended and the audio/video decoder throughput increased 3.5% with regard to the reference one. The results obtained in scenarios S1 and S6 show the effectiveness of the management of the QoS mechanisms at the task level to achieve composability in use-cases with multiple applications flows competing for common NoC links.

TABLE I. THROUGHPUT RESULTS FOR 6 DIFFERENT SCENARIOS CORRESPONDING TO THE USE-CASE OF FIGURE 7.

	Flows			Throughput
	Low Priority	High Priority	GT Connection	
S1	T1,T2,T3,T4	Audio, Video	-	30.62 frames/s
S2	T1,T2,T3	Audio, Video, T4	-	23.8 frames/s
S3	T1,T2	Audio,Video,T3,T4	-	16.76 frames/s
S4	T2	Audio, Video, T1,T3,T4	-	16.08 frames/s
S5	-	Audio, Video, T1,T2,T3,T4	-	14.81 frames/s
S6	-	Audio,T1,T2,T3,T4	Video	32.24 frames/s

The transmission of a given flow through the NoC may modify the original data rate, inducing variable latency values and resulting in missed deadlines at the target IP. Jitter is this instantaneous variation in latency and must be minimized in applications with QoS constraints. Figures 8 and 9 show the video decoder pipeline *jitter*. The X-axis represents the time interval between decoded blocks arriving at the JOIN task and the Y-axis represents the number of decoded blocks arriving within each time interval.

Figure 8 presents the jitter for the optimal mapping use-case (Figure 6); high priority for audio/video tasks (scenario S1); and GT connections for video tasks (scenario S6). For these three scenarios, most of the decoded blocks arrive at the JOIN task within the intervals (average and standard deviation values): 191 ± 55, 192 ± 59, 189 ± 57 kilo-clock cycles for optimal mapping, S1 and S6 respectively. Besides equivalent throughput values are achieved, the similarity between the three plots shows that proposed mechanisms efficiently decorrelates the target application, ensuring equivalent latency and jitter for different scenarios, even with disturbing traffic.

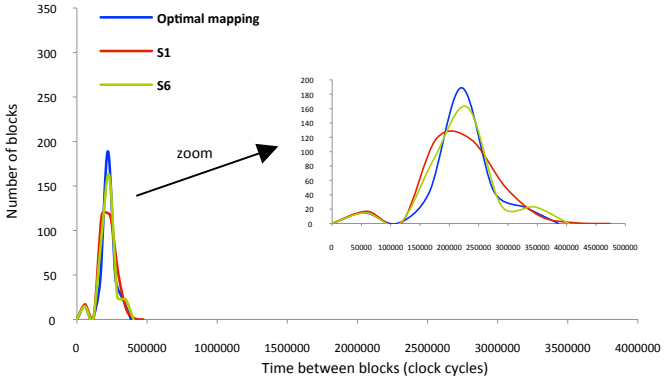

Figure 8. Jitter for scenarios guaranteeing the application requirement.

Figure 9 depicts the jitter for the scenarios S2-S5. The decoded blocks arrive at the JOIN task wit hin the interval: 254 ± 212, 416 ± 1042, 366 ± 1119, 382 ± 1132 kilo-clock cycles for each scenario. The disturbing traffic, flows T1 to T4, increase the average time between blocks, as well the jitter, being responsible of the throughput degradation observed in the Table 1.

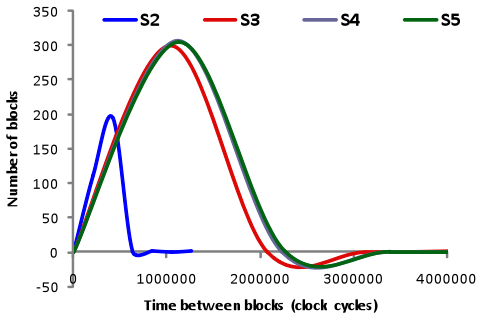

Figure 9. Jitter for scenarios not guaranteeing the application requirement.

Figure 10 shows the area, measured in LUTs and FFs, for the main components of the NPU: (*i*) PE, containing the Plasma Processor, the network interface and local memory; (*ii*) NoC Router. The original NPU design, mapped for a Xilinx Virtex-5 LX330 device, uses 4016/1997 LUTs/FFs and the NPU design with duplicated physical channels, priorities and circuit-switching uses 5652/2384 LUTs/FFs. The area overhead for the NPU design with support to soft and hard QoS is 40.74%/19.38% in LUTs/FFs. The overhead comes from the increased size of the NoC router, 165%, due to the physical channels duplication which doubles the router bandwidth. This area overhead, 40.74%, may be smaller when processors more complex than Plasma are used. For example, the CoMPSoC [2] reports an area of 57,882 LUTs for an MPSoC with 3 processors, SRAM, and audio/video I/O.

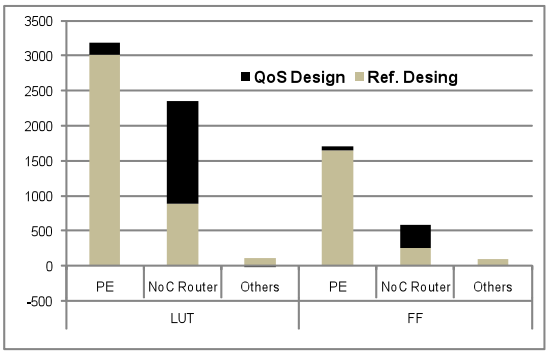

Figure 10. Number of LUTs and FF for the main NPU components. Black bars represent the overhead induced by the included QoS mechanisms. Target device: Virtex5 LX330

VII. Conclusions and Future Work

This work proposed an approach for achieving composability based on NoC QoS mechanisms, with the associated software API made available to application programmers. Performance results show the efficiency of the software controlled QoS mechanisms to isolate communication flows of applications with different time constraints and thus achieve composability. The obtained area results reveal a significant area overhead in the NoC design to support the different levels of QoS. Such an overhead can however be better amortized for systems based on complex processors which may include MMU, cache memories and multithreading support.

Exposing low-level hardware services in the API is a generic approach applicable to any platform. This increases the system programmability and enhances the control of the programmer over the platform at a high level of abstraction.

Future work lies in extending this approach to system adaptation; which promotes endowing the system with a certain degree of self-awareness. By continuously monitoring the achieved performance, the system software could adapt itself through selecting and tuning various QoS strategies, such as control over both communication and computation resources.

Acknowledgments

This research was supported partially by CNPq (Brazilian Research Agency), projects 301599/2009-2 and 142045/2008-0 and CAPES project 3503-09-7.

References

[1] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Ha, “Analyzing composability of applications on MPSoC platforms”. Journal of Systems Architecture: The EUROMICRO Journal, v.54(3-4), pp. 369-383, 2008.

[2] A. Hansson, K. Goossens, M. Bekooij, and J. Huisken, “CoMPSoC: A Template for Composable and Predictable Multi-Processor System on Chips”. ACM Transactions on Design Automation of Electronic Systems, v.14(1), pp. 1-24, 2009.

[3] A. Molnos, A. Milutinovic, D. She, and K. Goossens, “Composable Processor Virtualization for Embedded Systems”. In: Computer Architecture and Operating System Co-Design, 2010.

[4] Hansson, A.; Subburaman, M.; Goossens, K. “Aelite: A flit-synchronous Network on Chip with Composable and Predictable services”. In: Design, Automation and Test in Europe Conference, pp. 250-255, 2009.

[5] K. Goossens, J. Dielissen, A. Radulescu, "AEthereal Network on Chip: Concepts, Architectures, and Implementations". Design & Test of Computers, v.22(5), pp. 414- 421, 2005.

[6] J. Murillo, “HW-SW Components for Parallel Embedded Computing on NoC-Based MPSoCS”. PhD Thesis, Universitat Autònoma de Barcelona, 2010.

[7] A. Hansson, M. Wiggers, A. Moonen, K. Goossens, and M. Bekooij, “Enabling Application-Level Performance Guarantees in Network-Based Systems on Chip by Applying Dataflow Analysis”. Computers & Digital Techniques, v.3(5), pp.398-412, 2009.

[8] E. Carara, N. Calazans and F. Moraes, “Managing QoS Flows at Task Level in NoC-Based MPSoCs”. In: International Conference on VLSI and System-on-Chip, 2009.

[9] F. Moraes, N. Calazans, A. Mello, L. Möller, and L. Ost, “HERMES: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip”. Integration the VLSI Journal, v.38(1), pp. 69-93, 2004.

[10] A. Hansson and K. Goossens, “Trade-offs in the Configuration of a Network on Chip for Multiple Use-Cases”. In: International Symposium on Networks-on-Chip, pp. 233-242, 2007.

[11] G. M. Almeida, et al. “An Adaptive Message Passing MPSoC Framework”. International Journal of Reconfigurable Computing, vol. 2009, 20 pages, 2009.

[12] S. Rhoads, “Plasma - Most MIPS I(TM) opcodes”. Available at: http://www.opencores.org/project,plasma.

[13] E. Carara, N. Calazans, and F. Moraes, “A New Router Architecture for High-performance Intrachip Networks”, Journal of Integrated Circuits and Systems, v.3(1), pp. 23-31, 2008.

[14] A. Mello, L. Tedesco, N. Calazans, and F. Moraes, "Evaluation of Current QoS Mechanisms in Networks on Chip". In: International Symposium on System-on-Chip, pp.1-4, 2006.

[15] S. Gheorghita, T. Basten, and H. Corporaal, “Application Scenarios in Streaming-Oriented Embedded System Design”. Design & Test, v.25(6), pp. 581-589, 2008.

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

[17] H. Kopetz. “Real-Time Systems: Design Principles for Distributed Embedded Applications”, Kluwer Academic Publishers, 1997.