PONTIFÍCIA UNIVERSIDADE CATÓLICA DO RIO GRANDE DO SUL
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

RUNTIME ADAPTIVE QOS MANAGEMENT IN NOC-BASED MPSOCS

MARCELO RUARO

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science at Pontifícia Universidade Católica do Rio Grande do Sul.

Advisor: Prof. Fernando Gehm Moraes
Co-Advisor: Prof. Everton Alceu Carara

Porto Alegre
2014

Dados Internacionais de Catalogação na Publicação (CIP)

R894r Ruaro, Marcelo

Runtime adaptive QOS management in NOC-based MPSOCS /
Marcelo Ruaro. – Porto Alegre, 2014.
89 p.

Diss. (Mestrado) – Fac. de Informática, PUCRS.
Orientador: Prof. Fernando Gehm Moraes.
Co-orientador: Prof. Everton Alceu Carara.

1. Informática. 2. Arquitetura de Computador.
3. Multiprocessadores. I. Moraes, Fernando Gehm. II. Carara,
Everton Alceu. III. Título.

CDD 004.35

Ficha Catalográfica elaborada pelo
Setor de Tratamento da Informação da BC-PUCRS

Pontifícia Universidade Católica do Rio Grande do Sul
FACULDADE DE INFORMÁTICA
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

TERMO DE APRESENTAÇÃO DE DISSERTAÇÃO DE MESTRADO

Dissertação intitulada *"Runtime Adaptive QoS Management in NoC-Based MPSoCs"* apresentada por Marcelo Ruaro como parte dos requisitos para obtenção do grau de Mestre em Ciência da Computação, aprovada em 26/02/2014 pela Comissão Examinadora:

Prof. Dr. Fernando Gehm Moraes – Orientador — PPGCC/PUCRS

Prof. Dr. Everton Alceu Carara – Coorientador — UFSM

Prof. Dr. Ney Laert Calazans – — PPGCC/PUCRS

Prof. Dr. Rodolfo Jardim de Azevedo – — UNICAMP

Homologada em..24../..04../2014, conforme Ata No. 006.... pela Comissão Coordenadora.

Prof. Dr. Luiz Gustavo Leão Fernandes
Coordenador.

AGRADECIMENTOS

Agradeço a minha família (meu pai Silvio, minha mãe Marinês, minha irmã Thaís e especialmente ao meu tio Nelson) por tudo, antes, durante e depois desse momento.

Agradeço ao meu orientador Prof. Fernando Gehm Moraes por me aceitar no Mestrado como seu "desorientando". Por ter a paciência de me explicar os conceitos da área de sistemas embarcados. Pela pontualidade nas orientações e disciplina nas cobranças, atitudes que certamente já me baseio em minhas atividades docentes. Pelo tempo passado depurando trechos de código em *Assembly* na memória. Pelas revisões de todos os volumes do PPGCC e artigos escritos e publicados. Enfim, por ser orientador.

Agradeço especialmente a minha namorada e meu amor Fabiana que me apoia, me inspira, me acalma e que foi muito paciente em minhas tarefas do mestrado.

Agradeço aos meus colegas do GAPH (Grupo de Apoio de Pesquisa ao Hardware), pelo companheirismo, pela ajuda prestada, e pelas discussões, que com certeza contribuíram na escrita desse documento.

Agradeço aos professores do PPGCC pelos ensinamentos e conhecimento transmitido ao longo do Mestrado.

Agradeço aos membros da banca avaliadora, do Mestrado e dos volumes anteriores (professores Ney, Afonso e Amory) pelas revisões a sugestões.

Agradeço ao meu coorientador Prof. Everton Alceu Carara, pelas revisões de textos e esclarecimento de dúvidas técnicas.

Agradeço também ao CNPq pelo suporte financeiro.

E quero prestar um agradecimento a Deus, que de uma forma muito misteriosa, tenho certeza que me ajudou a completar com calma essa caminhada.

Por fim, quero enfatizar minha felicidade por concluir mais essa importante fase em minha vida, rodeado por pessoas que estiveram comigo e me apoiaram. Sei que os desafios não devem acabar, por que aí a vida fica sem graça, e por isso estou iniciando uma nova fase (Doutorado) que certamente irá me fornecer muitas experiências e conhecimentos.

GERENCIAMENTO ADAPTATIVO EM TEMPO DE EXECUÇÃO DE QOS PARA MPSOC BASEADOS EM NOC

RESUMO

Sistemas multiprocessados em chip (MPSoCs), baseados em redes em chip (NoCs), são resultados da contínua redução no tamanho dos transistores e na busca por um crescente poder computacional. Este aumento da capacidade de computação é alcançado através da replicação de componentes (processadores, memórias, roteadores, etc.), o que também fornece escalabilidade e simplifica o projeto. A estimativa de MPSoCs com elevado número de processadores acompanha a lei de Moore, e segundo o ITRS 2011 são previstos até 1000 elementos de processamento (PE) em um único chip até o ano de 2025. Esta estimativa é impulsionada principalmente pela mercado de telecomunicações e multimídia, que inclui dispositivos como *smarthphones* e computadores móveis. Este tipo de emprego de MPSoCs exige que tais sistemas sejam capazes de suportar uma variada gama de aplicações e com diversos requisitos de desempenho. Assim, o sistema deve ser capaz de fornecer qualidade de serviço (QoS) para as aplicações e ajustá-la em tempo de execução.

As propostas atuais da literatura visam fornecer adaptação de QoS em tempo de execução levando em consideração o uso de poucas ou de somente uma técnica de QoS. Observada a variedade de aplicações é claramente perceptível que certas aplicações possuem requisitos diferenciados de QoS, podendo requisitar mais recursos de computação, comunicação, ou ambos simultaneamente.

Portanto, este trabalho propõe explorar diferentes níveis de QoS através do uso de quatro diferentes técnicas adaptativas de QoS que são gerenciadas em tempo de execução de acordo com os requisitos das aplicações. O gerenciamento é distribuído através do particionamento do MPSoC em regiões, denominadas *clusters*, havendo um PE responsável pela gerência de cada *cluster*. Este gerenciamento adaptativo é realizado através de uma heurística que é executada em cada sistema operacional gerente de *cluster*. Uma infraestrutura de monitoramento híbrido fornece as informações necessárias para a heurística adaptativa. Esta infraestrutura é dividida em dois níveis hierárquicos, sendo escalável e com um nível de intrusão que corresponde no pior caso a 0,8% de utilização do enlace da NoC. O gerenciamento adaptativo de QoS em tempo de execução age na computação, comunicação ou em ambos e faz com que aplicações de tempo real flexíveis possam restaurar seu desempenho após detectado uma queda do mesmo pelo monitoramento.

Neste tralho é também proposta uma ferramenta de depuração para MPSoCs baseados em NoC, esta ferramenta fornece depuração no nível de protocolo de comunicação ente os PEs, e facilita o processo de implementação, validação e obtenção de resultados de novos protocolos para o sistema.

Palavras Chave: MPSoCs baseados em NoC, QoS, adaptação, monitoramento.

RUNTIME ADAPTIVE QOS MANAGEMENT IN DISTRIBUTED NOC-BASED MPSOCS

ABSTRACT

Multiprocessor systems on chip (MPSoCs), using networks on chip (NoC) as the communication infrastructure, result from the continuous reduction in the transistors size and the need for increasing computational power. This increased computing capacity is obtained through the reuse of components (processors, memories, routers, etc.), which also provides scalability, and simplifies the design process. MPSoCs with hundreds of processing elements (PEs) follows the Moore's law, and according to the ITRS 2011 it is predicted up to 1000 PEs in a single chip at the end of 2025. This estimation is driven mainly by the telecommunications and multimedia market, which includes devices such as smartphones and mobile computers. Such devices require systems able to execute a wide range of applications, with different performance requirements. Thus, the system must be able to provide quality of service (QoS) to applications, and adjust the resources usage at runtime.

Literature proposals provide runtime QoS adaptation taking into consideration the use of only one or two QoS techniques. Considering the application diversity that may execute in MPSoCs, applications may have different QoS requirements, requiring more computing resources, communication resources, or both simultaneously.

Therefore, this work aims to explore different QoS levels addressing four different adaptive QoS techniques managed at runtime according to the soft real-time applications' requirements. It is assumed that the MPSoC is partitioned in regions, named clusters, with one manager PE per cluster. This adaptive management is controlled through a heuristic that is executed by the OS of each cluster manager. A hybrid monitoring infrastructure provides the necessary information for the adaptive heuristic. This infrastructure is divided in two hierarchical levels, being scalable and with an intrusion level that corresponds, in the worst case, to 0.8% of the link utilization. The runtime adaptive QoS management acts in computing, communication, or both, enabling soft real time applications to restore their performance after detected a performance decrease by monitoring.

Furthermore, a debugging tool for NoC-based MPSoCs is proposed in this work. This tool provides a communication protocol level debugging, and helps the process of implementation, validation and extraction of results of new system protocols.

Keywords: NoC-based MPSoCs, QoS, adaptation, monitoring.

LIST OF FIGURES

LIST OF TABLES

LIST OF ACRONYMS

API	Application Programming Interface
ASIP	Application Specific Instruction Set Processors
CA	Cluster Agent
CM	Cluster Manager
CPU	Central Processing Unit
CS	Circuit Switching
DMA	Direct Memory Access
DSP	Digital Signal Processor
DTM	Dynamic Thermal Management
DTW	Dynamic Time Warping
DVFS	Dynamic Voltage and Frequency Scaling
FFT	Fast Fourier Transform
FPGA	Field Programmable Gate Array
GA	Global Agent
GPP	General Porpoise Processor
GPU	Graphical Processor Unit
GS	Guaranteed Service
GT	Guaranteed Throughput
GUI	Graphical User Interface
IP	Intellectual Property
ISA	Instruction Set Architecture
ITRS	International Technology Roadmap for Semiconductors
MEP	Monitor Executive Processor
MNoC	Monitor Network on Chip
MPI	Messaging Passing Interface
MPSoC	Multi-Processor System on Chip
NI	Network Interface
NoC	Network on Chip
OCP	Open Core Protocol
OS	Operating System
PE	Processing Element
PLB	Peripheral Local Bus
PMU	Probes Management Unit
PS	Packet Switching
QoS	Quality of Service
RT	Real Time
RTL	Register Transfer Level
SDM	Spatial Division Multiplexing
TA	Tile Agent
TCB	Task Control Block
VC	Virtual Channel
VHDL	VHSIC Hardware Description Language

TABLE OF CONTENTS

1 INTRODUCTION

The reduction of the transistor size allowed the design of increasingly complex systems, with billions of transistors in a single chip [KOR13] [TIL13]. Currently, the state of the art in the semiconductor industry is addressing 22 nm transistors scale [INT13]. Together with this growing complexity, the reuse of IPs (Intellectual Property modules) is a design technique to explore the available number of transistors that can be manufactured in a single die. NoC-based MPSoCs arise in this context, aggregating multiples processing units interconnected by a communication network to compose a system offering scalability and a high degree of parallel processing. NoC-based MPSoCs are a concept that emerged in the beginning of last decade [BEN02], and thenceforth has evolved through several contributions in academic and industrial works.

MPSoCs are adopted in several areas of the electronic industry, with emphasis on embedded systems. Embedded systems are specialized computing systems, with specific requirements to be fulfilled. Examples of embedded systems include: smartphones, tablets, game consoles, avionic devices, automotive devices, medical monitoring systems, biometric systems, among others. Therefore, it is noticeable the potential that embedded systems have, and the relevance of the research in this field, aiming to add more computing power to such systems.

1.1 MPSoC

An MPSoC [WOL08] is composed by several IPs, whose objective is to perform a specific function according to the embedded system application. MPSoCs can be classified into two large groups: heterogeneous and homogeneous.

- In *heterogeneous* MPSoCs, also known as asymmetric MPSoCs, different PE (Processing Element) architectures are used, as GPPs, GPUs, DSPs and ASIPs.
- In *homogeneous* MPSoCs, also known as symmetric MPSoCs, all PEs have the same architecture and are replicated in the system enabling the support of multitask parallel processing. A PE contains a processing unit, a DMA (Direct Memory Access) module, a NI (Network Interface) module, and usually a local memory. In this work, it is also included the router module in the PE definition.

Heterogeneous MPSoC target specific applications, because their IPs are specialized according to the application. In the other hand, homogeneous MPSoCs target general-purpose applications, such as multimedia and telecom applications, and use the replication of identical components (PEs) to increase the processing power. Homogeneous MPSoCs are a trend, due the possibility to achieve better load balancing and fault tolerance due to the redundancy of components. Currently, the industry already works with homogeneous MPSoCs, with e.g. 72 PEs per chip [TIL13]. According to the ITRS 2011, at the end of 2025 MPSoCs will have more than 1000 PEs (http://www.itrs.net/).

Examples of important research topics in the MPSoCs arena include energy consumption, temperature control, QoS, programmability, and fault-tolerance.

1.2 NoC

A NoC [BEN02] is a communication infrastructure to interconnect IPs, decoupling the computation from communication. NoC is composed, basically, by routers and wires. Routers have the function to define the path for each exchanged packet between a given source-target pair, and wires have the function to interconnect routers, and to connect each router with its respective IP.

The main features of a NoC are topology, routing algorithm, flow control, and switching mode [MOR04]. As for the topology, currently stand out the 2D-mesh topology, due its ease of implementation and scalability. As for the routing algorithm, the turn-based algorithms (XY, West-first, etc.) are mostly used, due to low area requirement, minimum path, and deadlock free features. Example of flow control method is the synchronous credit-based, which provides lower latency compared to handshake methods. The most used switching mode is the wormhole packet switching, since it requires the smaller amount of space in buffers and avoids the reservation of network resources for a large amount of time as circuit switching.

An important challenge in the NoC area is to integrate more than one NoC in an MPSoC [TIL13], aiming to allow certain services to use a dedicated communication infrastructure. Another reason to have more than one NoC in the MPSoC is the need of system monitoring support, which can induce excessive traffic on a single network, disturbing applications' communication.

1.3 Distributed Resource Management

In a general purpose MPSoC, with dozen of PEs and several applications starting and finishing over time, it becomes necessary to employ system management. One PE, named master PE, usually performs this management. The master usually does not execute applications' tasks, it is responsible for the system management. Examples of functions that can be assigned to the master PE include: task mapping, task migration, monitoring, execution of adaptive heuristics, among others.

Considering the large number of PEs in current MPSoCs [TIL13], having only one master PE responsible for the entire system management (Figure 1-b) represents a bottleneck, compromising scalability. This single master PE can become unable to handle all system requirements, and induce hotspots in the NoC links close to the master PE.

In distributed management [SHA11][CAS13a] (Figure 1–a), the objective is to divide an MPSoC in clusters, and associate one master PE to each cluster, distributing the management tasks, making the system scalable. In addition, there is a greater isolation between applications, because mapping heuristic will try to map all tasks of a given application in the same cluster. Such isolation between applications favors composability, an important feature for QoS. Clusters can also change their size at runtime – through reclustering techniques – if the number of application tasks is greater than the number of available cluster resources [CAS13b]. The main drawback of distributed management is the amount of resources reserved for the system management, but this is an acceptable cost due the offered scalability and composability provided by this management scheme.

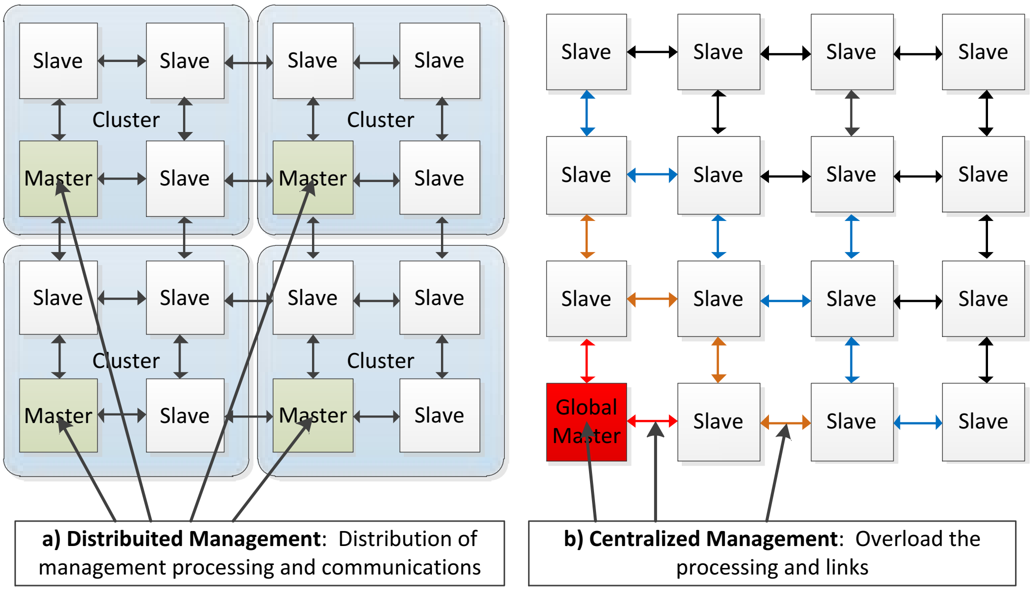

Slave
Master
Cluster
Global Master
a) Distribuited Management: Distribution of management processing and communications
b) Centralized Management: Overload the processing and links

Figure 1 – Distributed versus centralized MPSoC management.

1.4 Quality of Service

Quality of service (QoS) is a system capability to provide services that meet the user applications requirements. To achieve QoS the system can dedicate some of its resources exclusively to fulfill with the applications requirements. The availability of mechanisms to ensure QoS is an indispensable feature for systems with real-time applications support [CAR11].

There are two main classes of applications that a system with QoS support must consider: (*i*) Best-Effort (BE) applications, which do not have time requirements; (*ii*) real time (RT) applications, with temporal requirements, requiring certain levels of QoS to not suffer interference from other applications running on the system.

With the increased complexity of NoC-based MPSoCs in terms of processing elements (PEs), more applications can run simultaneously on such systems, requiring management techniques able to meet the RT applications constraints. In a general purpose MPSoCs, such those described in [HOW10] and [TIL13], applications may start their execution at any moment, characterizing a dynamic workload behavior. In addition, resource sharing among running applications may lead to performance degradation. Thus, a major challenge in the design of such systems is to ensure QoS without performance degradation after a certain period of execution, when several applications were inserted and removed from the system.

Composability is a key feature to achieve QoS in platforms with resource sharing. Composability has as its main objective minimize the inter-application interference and, consequently, allowing the RT applications to reach its constraints. As previously mentioned, the clustering management favors composability through application isolation.

In NoC-based MPSoCs context, two classes of techniques may help to provide composability: communication and computation. The main concern of communication techniques is to control the network resources avoiding traffic interferences. The main concern of computation techniques is to control the processor utilization avoiding interferences due to processor scheduling. To provide a complete QoS support, the platform must support these two classes of techniques.

Flow priority is a communication QoS technique. The basic assumption is to assign different priority levels to the packets injected into the NoC, and reserve network resources according to this priority. Packets with higher priority receive a differentiated treatment, with regard to the resource allocation and differentiated services (differentiated routing algorithms, larger buffer size, arbitration privileges, dedicated physical channels, among others). This approach enables to support soft real-time applications because it contributes to the isolation of the traffic on the network.

Circuit switching (CS) is also a QoS communication technique. CS is largely used to fulfill the constraints of hard real time applications. In CS communication mode, an end-to-end path reservation is established, and the communication takes advantage of the absence of interference from other flows to achieve maximum throughput and minimal jitter.

Besides these techniques, other QoS communication techniques may be employed such as: virtual channels [SAL11][FAN11], time division multiplexing [STE12], and clustering [CUI12][LIA11].

Another way to provide QoS is to act in the computation. In this context, most techniques avoid that RT tasks share the same processor with other RT tasks. Processor scheduling is an example of adopted technique [MOR12]. RT tasks can execute for larger time slices, running for longer periods without interference from other tasks. The scheduler must implement a priority policy enabling to increase the time slice value according to the task priority. The same task scheduler priority can be used together with network flow priority.

Task migration can also provide QoS, if the previous techniques do not achieve the required QoS level. Task migration moves a given task from one PE to another PE [MOR12]. The main goals of task migration in MPSoCs comprise workload distribution, temperature balancing and energy consumption optimization. Task migration may help to achieve the QoS level since the migrated task may be allocated in a free PE (computation technique), and can act in the communication, because the distance between communicating tasks can be reduced (communication technique).

1.5 Monitoring

Monitoring is the key mechanism to enable runtime system adaptability. The monitoring evaluates, at runtime, the system performance, and usually notifies through events an event manager [KOR13]. The event manager can choose the technique that best fits to restore the part of the system with a QoS requirement infringement. The monitoring also can be applied at task level, monitoring RT tasks performance and employing QoS techniques as necessary. Examples of system performance parameters that can be monitored include latency, throughput, jitter, CPU usage, temperature, voltage, among others. The main features to observe in the monitoring implementation are fast response and low intrusiveness, i.e. avoid that monitoring disturbs the application performance.

Monitoring systems can be implemented in software, hardware or be hybrid [KOR13]. Software implementation addresses monitoring at the system level, enabling to control the performance of applications or individual tasks. The main drawback is processing intrusion, because software monitoring generally uses the same processor used by tasks. Hardware implementation addresses monitoring at lower levels, enabling faster responses, and consequently support to real time applications. The hardware monitor can share the same regular data network or employ a specific hardware (dedicated network). Its main drawback is the area cost, and its advantage is low communication intrusion, if monitoring data do not interfere with application data. Finally, hybrid implementations mix software and hardware implementations, and enables the performance parameter monitoring at the system (e.g. tasks execution time, CPU usage) and network (throughput, latency, and jitter) levels. However, such hybrid schemes can penalize the CPU performance and the communication network. To cope with this, the monitoring infrastructure can be hierarchically organized. This feature allows the system to distribute the amount of data produced in multiple event managers, providing scalability.

1.6 Motivation

Everton Carara PhD Thesis [CAR11] addressed the exploration of differentiated communication services to provide QoS in NoC-based MPSoCs. These services are exposed to application's developers through API primitives that enable developers to configure, at application design time, different QoS levels to each application task. The main feature missing in that work is the runtime adaptation of the proposed QoS services.

The motivation to the present work was to integrate several QoS techniques together with a monitoring scheme. As can be observed in Chapter 2 (state of the art), current proposals are focused in providing some degree of adaptation, employing usually only one or two QoS techniques. The present work adopts a monitoring and QoS adaptation that can provide runtime adaptation with four techniques that maximize system resource utilization through heuristics that employ a gradual QoS adjustment according to the requirements of each communicating task pair.

1.7 Goals

This work has strategic and specific goals. The strategic goal is to propose an adaptive heuristic to be employed at runtime, with four different techniques to provide QoS for RT applications at the computation and communication levels for large MPSoCs. Each QoS technique is chosen according to the monitoring data extracted from the system and from application requirements.

The specific goals included:

- Hardware and software integration of two HeMPS platforms (Distributed-HeMPS [CAS13b] and QoS-HeMPS [CAR11]), to compose a new one called QD-HeMPS.
- Development of a software and hybrid distributed monitoring scheme;
- Evaluation of the intrusiveness of both monitoring schemes, choosing the one presenting smaller intrusiveness and higher scalability;
- Development of three adaptive QoS techniques: (*i*) flow priority; (*ii*) circuit switching; (*iii*) task migration;
- Integration of the processor scheduling priority mechanism proposed in [MAD13], to provide QoS (the fourth QoS technique);
- Development of an adaptive heuristic to employ at runtime all four QoS techniques;
- Evaluation of the QoS heuristic.

1.8 Contributions

The contributions of this work include:

- A distributed and QoS NoC-based MPSoC platform;
- A software and hybrid monitoring schemes, with the evaluation and comparison of both schemes;
- Three adaptive QoS techniques: (*i*) flow priority; (*ii*) circuit switching; (*iii*) task migration;
- A heuristic to employ the appropriate QoS technique at runtime.

Another important contribution was the development of a protocol-based debugging tool to aid in the implementation of the specific goals. This debugging tool became a powerful tool to implement and validate new protocols and to debug NoC-based MPSoCs.

1.9 Document Organization

This document is organized as follows. Chapter 2 presents the state of the art in two areas of NoC-based MPSoCs: monitoring and QoS. Chapter 3 presents the MPSoC platforms used as reference for the development of this work. Chapter 4 presents the debugging tool. Chapter 5 presents the monitoring infrastructure. Chapter 6 presents the adaptive QoS techniques. Chapter 7 presents the runtime adaptive QoS management of the techniques presented in Chapter 6. Finally, Chapter 8 presents conclusions and directions for future works.

2 STATE OF THE ART

This Chapter reviews related works in monitoring and QoS of NoC-based MPSoC systems. At the end of each session, a comparative table is presented positioning the present work with the state of the art.

2.1 Monitoring

Stan et al. [STA11] proposes monitoring at runtime to detect timelines violations on MPSoCs. The monitoring scheme is implemented in hardware, enabling run-time verification according to the loss of deadlines in the communication between tasks. This evaluation occurs at each PE in the PLB (Peripheral Local Bus), observing the transactions made at a circular buffer, mapped in the processor memory space. This circular buffer is used in the communication between the tasks. The monitor evaluates communication's time intervals defined in experiments (profiling step) with the communication's time intervals measured in the PLB, and generates events when a violation is detected. No monitoring results are presented.

Matos et al. [MAT10] proposed a monitoring scheme observing the NoC communication buffers. The monitor consists of one hardware block implemented inside the local buffer controller. The monitor counts the number of packets transmitted through the channel. If the number of monitored packets violates a threshold (configured at design time), the monitor sends the amount of monitored packets to a central controller. The central controller checks whether the amount of received packets achieved a threshold (set at design time), and then invokes a policy for buffer resizing. The obtained area overhead was 37,5% over the reference router. No intrusion results are explored.

Faruque et al. [FAR12] proposed a monitor to support QoS through buffer resizing. The implemented monitoring evaluates the occurrence of events. Events are situations when successful traffic propagation is prevented in the communication architecture. The monitoring component is event-based. Events are created when a performance reduction in a subsystem of an individual router is detected (*TTL-expire-event*, *No-route-found-event*, *No-buffer-event*, *Buffer-full-event*). Figure 2 presents the monitor overview. The monitor is implemented between the router and the NI, being a counter that evaluates the number of events up to a threshold. If the number of events is greater than the threshold, the monitor notifies the NI. When the NI receives a notification from the monitor (in the form of a monitoring packet), it performs adaptations to avoid new violations. As monitoring is based on events, there is no monitoring traffic when the network operates normally. The monitoring intrusion obtained by experiments was 0.7% of the total link capacity. The area overhead was 297 slices on an FPGA to each PE.

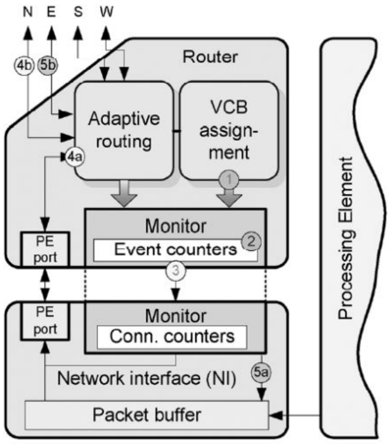


Figure 2 – Overview of the monitoring approach [FAR12].

Saind et al. [SAI10] proposed a technique to monitor tasks with temporal constraints. Only periodic tasks are considered. The descriptor of the task TCB (Task Control Block) was modified aiming to support variables that enable to control time thresholds for periodic tasks. The monitoring is based on watchdog timers, divided into two hierarchical levels: local and global watchdogs. Figure 3 presents the monitoring approach using the watchdogs. Local watchdogs control the execution at the task level, and the global watchdog control the execution at the system level. The global watchdog has a common hyper period for all tasks. A communication between a task and its watchdog is created when the task starts, initializing the watchdog with temporal parameters obtained in a profiling step. Local watchdogs communicate with the global watchdog to report the loss of deadlines through an event. The global watchdog stores these events. When the hyper period expires, the global watchdog analyzes the list of events and verifies if these affected the overall execution of the system. If there are any violations, the global watchdog searches where the source of violations is located, and triggers a measure to solve the problem. No monitoring results are presented.

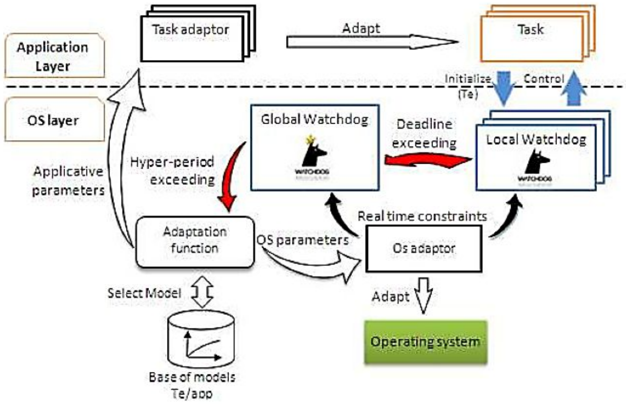


Figure 3 – Monitoring using watchdogs [SAI10].

Fiorin et al. [FIO10] proposes a monitoring architecture for NoCs. The architecture is based on probes that monitor the occurrence of events. The monitoring system is composed of three main components (Figure 4):

(i) *probes*, located inside NIs and monitor the cores' operations and events (reading the OCP/IP signals), as well as resources utilization and the communication events generation in routers and NIs;

(ii) a *subsystem* for data collection and storage;

(iii) a *central element* (Probes Management Unit - PMU), which is in charge of the configuration of the probes and the collected data handling.

The probes are generic and can be configured with different communication thresholds. In [FIO10] time windows are defined, and at the end of a window the events collected by each probe are sent to the PMU. The PMU has two basic tasks: program the probes' registers with thresholds, and process the collected data from probes. Both tasks are software implemented by the subsystem that runs in each PMU. The Authors report that the monitoring traffic presented an intrusion to 0.2% of the bandwidth of the NoC (equivalent to 2 GBps) and 5% of traffic generated by the PEs. The area overhead was 55% compared to the reference NoC and 3% over the entire PE.

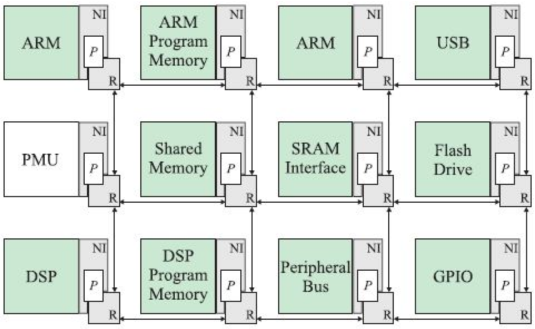


Figure 4 – Overview of NoC monitoring infrastructure proposed in [FIO10].

Wang et al. [WAN10] proposes a monitoring scheme based on sensors to monitor temperature and voltage. The sensors are distributed and are inserted at each PE (between the core and the NI). Each sensor sends its data using a dedicated NoC communication infrastructure. The data is sent to a central manager – CM, which can take actions to avoid violations of predefined thresholds. The sensors collect information at runtime for each PE. The monitoring information contains the address of the sensor that sent it, which allows the CM to identify the source of the information and apply the adaption procedure. The maximum intrusion was 0.8% compared to NoC without sensors. The Authors do not present area overhead results.

Zhao et al. [ZHA11] proposes a centralized monitoring infrastructure for temperature, voltage and error data monitoring. A dedicated monitoring NoC called MNoC (Monitor Network-on-Chip) interconnects the monitors. Figure 5 presents the MNoC overview. The MNoC also has an interface to an exclusive monitoring processor called MEP (Monitor Executive Processor). This processor has a software layer that implements the monitoring algorithms. The monitors are divided into two categories: regular monitors (used for periodic monitoring data, such as temperature), and irregular monitors (with reports abnormal events, as faults). The communication in this scheme occurs only between monitors and MEP and not between monitors.

The monitored data is classified in two classes. The regular monitors have low priority, and irregular monitors higher priority. Once the MEP receives monitoring data from the monitors, it uses this information to determine the type and location of the event and take the necessary actions by changing system parameters.

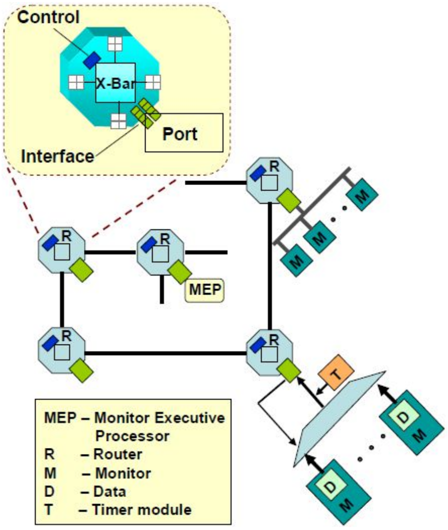


Figure 5 – MNoC Overview of [ZHA11].

An area overhead of 0.2% was obtained for the monitor interconnection when applied to an eight-core system based on the AMD Athlon processors. In network comparisons, the entire MNoC router is about 7% of the size of a reference NoC router.

Faruque et al. [FAR10] proposes software monitoring for Dynamic Thermal Management (DTM). The goal is to keep the temperature of each core below a threshold, avoiding hotspots. The monitoring is performed through agents implemented in a distributed manner. The scheme employs a hierarchical management with three agent classes, Figure 6. The Tile Agents (TAs) are attached to each tile, containing sensors that measure the tile temperature and communication volume. When a temperature threshold is reached, it is generated a notification from a TA to a CA (Cluster Agents), and this may initiate the DTM. CAs keep the temperature information of each cluster tile, and execute task remapping according to the tiles' temperature. They also communicate with other CAs, and store application details such as the application task graph. Global Agents (GAs) perform assignment of unmapped tiles to clusters, create and destroy CAs, and are called by CAs to remap applications to enable DTM across clusters.

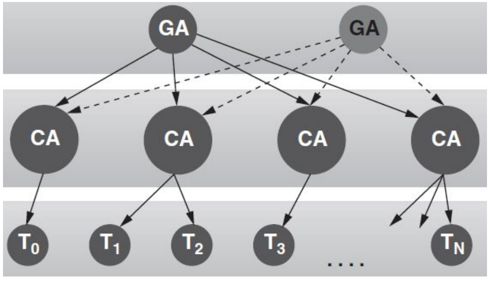


Figure 6 – Agents Hierarchies [FAR10].

The overhead is measured in terms of memory usage. Agents require 147.5 Kbytes, 494.1 Kbytes, and 19 bytes of on-chip memory for a GA, the CAs, and a TA, respectively, for a 64x64 architecture with 32 clusters. The required on-chip memory required was 0.65 Mbytes, which can be considered, according to the Authors, negligible in future technology nodes. No monitoring intrusion results were presented.

Guang et al. [GUA10] proposes a hierarchical monitor system for power management. The work evaluates latency and throughput parameters. If these parameters exceed a threshold, DVFS (Dynamic and Voltage Frequency Scale) is applied. The monitors are divided into four types of agents: cell agents (hardware implemented, corresponding to the lower hierarchical level); cluster agents (software and hardware implemented, second level of hierarchy); platform agents (software implemented, third level of hierarchy); and application agents (software implementation, with the higher hierarchy level). Each agent has a specific role. Application agents sends latency application's requirement to platform agents. Platform agents determine the number of processors for each application and map these processors on nodes of the network based on a tree algorithm. Platform agents also determine the maximum load of buffers for each cluster agent according to each application's requirements. Cluster agents apply DVFS for each application according to monitored cluster traffic reported by cell agents. The proposal overhead of the hardware agents was 0.003 mm² and 0.008 mm² to cluster and cell agents respectively. No overheads related to software agents are presented.

Fattah et al. [FAT11] proposes a model for a hierarchical management that mixes a distributed and centralized monitoring system. In the proposed scheme, there are various levels of managers/monitors. The first level corresponds to monitors allocated in PEs (cell monitors) that send information to the application manager, which sends its information to a global manager (Figure 7). According to system demands, each cell may contain different monitoring capabilities (e.g. temperature sensors, power monitors, fault detectors, etc.), along with several actuators such as voltage and frequency regulators, reconfiguration possibilities, and so on. Each cell has its own manager that generates events for its higher-level manager based in the monitored information. A mid-level manager groups and manages a set the cells. In the proposal, the size of the cluster is application oriented (a cluster consists of a group of PEs that runs tasks of an application), but different clustering policies may be employed. The mid-level manager, called application manager, manages each cluster. Each application manager is responsible for the distribution and management of tasks in order to supply the requirements of the application. A top-level manager, or System Manager, is responsible for managing the system, and coordinates the actions of lower-level managers (application managers). The System Manager runs at the operating system level, creating and removing application managers according to the characteristics of the system.

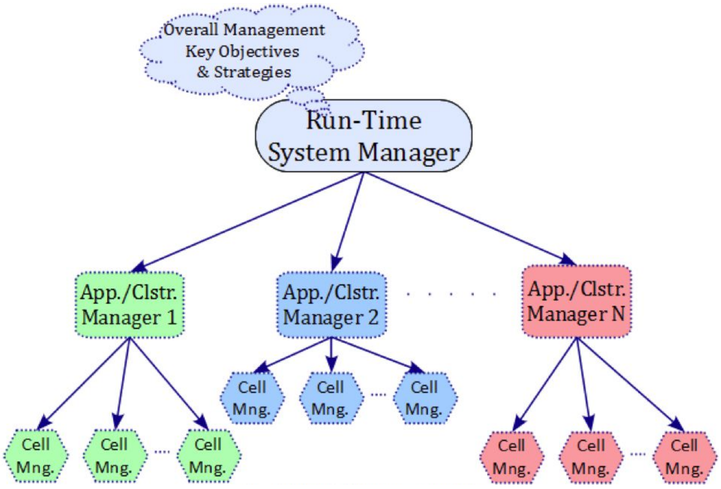


Figure 7 - MPSoC Management Hierarchy [FAT11].

Kornaros et al. [KOR12] proposes a real time hierarchical monitoring infrastructure. The implemented monitoring is based on local monitoring units of low granularity. These units are hardware implemented at the NI. The data collected at real time on each local unit is transferred to the monitoring manager. This manager, implemented in hardware, can analyze statistically the data and employ adaptive resource management techniques. The monitors implemented in the NI can also take local decisions, communicating with a buffer manager to execute a dynamic buffer resizing. Each monitor of the NI evaluates the packets' header, and through a filter mechanism, which can be programmed by software, can reduce the volume of data collected before sending to the monitoring manager. Local monitors also are able to differentiate packets from different clusters and applications. The monitoring scheme is hierarchically organized. In the first level (system level), the monitoring manager, execute software management routines. The manager is responsible for monitoring data from the probes of each NI. Furthermore, as a top hierarchical level, the work proposes a high performance global manager implemented in hardware. Each cluster monitoring manager or local monitors, can communicate with this central manager, and signalize the occurrence of events. The proposed monitoring scheme is generic and allows the designer to customize it at design time. It is also possible to determine how local monitors will communicate with the monitoring manager, being possible to use a dedicated link or the NoC links. The Authors mention that the monitoring process delay (capturing, selection and transmission of the identified event), takes in average 50 clock cycles. The obtained area overhead was 11,7% compared to a Hermes 2x2 NoC [MOR04].

Table 1 compares the reviewed works. The second column of the Table addresses the monitoring organization. Works [STA11][MAT10][FAR12][SAI10] adopt distributed monitoring. In this context, a monitor module is inserted at each node of the MPSoC or NoC, executing local adaptations. This scheme may be used with simple adaptation protocols, or with simple inter-monitor communication. However, it is not scalable because the protocols must be implemented at each node, increasing the overhead (redundancy) of the platform. In addition, a fully distributed approach does not contain a global or partial view of the system, leading to local decisions, which may compromise the performance of the whole system.

Works [FIO10][WAN10][ZHA11] employ a centralized monitoring scheme. The centralized organization avoids the drawback of the distributed monitoring, using a single global unit to control the system. This approach often adopts a hybrid implementation (hardware and software monitors), due to more abstract information handling by the control unit. As in the fully distributed approach, the centralized monitoring is also not scalable because concentrate all the monitored information in a single unit.

Works [KOR12][FAR10][FAT11][GUA10] use a hierarchical monitoring organization. These proposals divide the monitoring management in usually in two hierarchical levels. Each level executes specific monitoring actions, based on their granularity. This scheme is scalable, because it distributes the monitoring traffic and monitoring management in regions, named clusters. Inside each clusters the monitoring is centralized, but at the platform level, the monitoring is distributed. Such scheme also uses a hybrid implementation enabling generic monitoring, because the user can implement different adaptation techniques in the two management levels (cluster and global), and can configure the monitoring characteristics at design time.

Works [ZHA11][GUA10] use a dedicated network for monitoring. This approach eliminates the intrusion of the monitoring traffic in the regular traffic, but requires more area, energy, and increases the design complexity.

The present work adopts a hierarchical organization, with a hybrid implementation and a cluster management only, not adopting a global manager due the management bottleneck possibility in this manager level. The proposed monitoring can be used for general purposes because it is not linked with the runtime adaptation modules.

Table 1 - State-of-the-art comparing works targeting monitoring support in NoCs.

Proposal	Organization	Implementation	Monitoring Management	Application
Stan (2011)	Distributed	Hardware	Each PE	QoS
Matos (2010)	Distributed	Hardware	Router	Buffer resizing
Faruque (2012)	Distributed	Hardware	Router/NI	QoS and Buffer resizing
Said (2010)	Distributed	Software	Microkernel	Real time task deadline
Fiorin (2010)	Centralized	Hybrid	NoC Central Unit	Generic (resources availability)
Wang (2010)	Centralized	Hybrid	NoC Central Unit	Thermal and Voltage
Zhao (2011)	Centralized	Hybrid	NoC Central Unit	Temperature, voltage, error data
Faruque (2010)	Hierarchical	Hybrid	Cluster and Global Manager	Dynamic Thermal Management
Guang (2010)	Hierarchical	Hybrid	Cluster and Global Manager	DVFS
Fattah (2011)	Hierarchical	Hybrid	Cluster and Global Manager	General purpose
Kornaros (2012)	Hierarchical	Hybrid	Cluster and Global Manager	General purpose
This proposal	**Hierarchical**	**Hybrid**	**At each Cluster Manager**	**General purpose**

2.2 Quality of Service

Wang and Bagherzadeh [WAN12] propose a router able to meet QoS constraints through arbitration with dynamic congestion control and adaptive routing. The Authors adopt a NoC with 64-bit bi-directional links, where each router has 2 horizontal links, 4 vertical links, and 4 diagonal links. VCs are not used in the adopted NoC. For the dynamic control of QoS it is used an algorithm that evaluates the congestion at input and output ports. Packets coming from hotspots have high priority to be arbitrated, reducing congestion. However, this congestion control is restricted to the router, where local decisions are made, which may not necessarily eliminate congestion, but just transfer it to another point.

Wissem et al. [WIS11] presented the design of a *Switch Allocator* for a router to enable QoS support. The Switch Allocator contains five dynamic arbiters, and receives routing requests from each input port, assigning packets to the output port in accordance with the packet priority. The packet that has the highest priority grants access to the output port. When there is more than one packet with the same priority, the order is defined by a round-robin algorithm. According to the Authors, the proposed router supplies a bounded latency to real time packets.

Kakoee et al. [KAK11] propose a fault tolerant router and QoS support with flow priority and duplicated physical channels. The channels are divided in high and low priority. Each packet has a bit setting its priority. In the proposal, one channel is dedicated to high-priority traffic, and the second one for low-priority traffic, being possible to share it with high-priority packets.

Salah and Tourki [SAL11] propose a router architecture for real-time applications, using packet switching, QoS support and a priority-based flit scheduler for BE and GS flows. The scheduler evaluates the deadlines of the incoming flows, selecting VCs according to the flow class, BE or GS. According to the Authors, results show that their router achieves an optimal packets scheduling, increasing channel utilization and throughput, a reduction of network latency, and avoidance of resources conflicts.

Fan et al. [FAN11] propose a router combining VCs with duplicated physical channels. Each physical channel creates a *subnet*, with a dynamic number of VCs. The allocation of VCs is executed at runtime, by an arbiter, reserving VCs and dynamically allocating them according to the demand of each port. Each subnet uses a different routing algorithm: XY and YX. The NI chooses which subnet a given flow will use. Once on a subnet, a packet cannot change the subnet. The VCs for XY only accept packets routed according to the XY algorithm, and vice versa.

Lusala and Legat [LUS11] propose a NoC with simultaneous PS (packet switching) and CS (circuit switching). Two subnets are created: one for CS using SDM (Spatial Division Multiplexing), and the other one for PS. A “sub-router” controls each subnet. GT (guaranteed throughput) flows use the CS subnet, while BE flows use the PS subnet. The PS sub-router, besides transmitting BE traffic, is responsible for the configuration of the CS sub-router, receiving connection requests and reserving a channel of the CS sub-router.

Winter and Fettweis [WIN11] presented and evaluated different implementations of a central hardware unit, name *NoCManager*, which allocates at run-time guaranteed service VCs providing QoS in packet-switched NoCs. The Authors argue that the central *NoCManager* is superior to the distributed technique. Besides this conclusion, the Authors mention scalability issues, and point out a hierarchical method as future work.

Saponara et al. [SAP12] presents the design of a network interface with hardware support for error management, power management, ordering handling, security, QoS management, programmability, end-to-end protocol interoperability, remapping. The Authors adopts virtual channels and a bandwidth allocation scheme to ensure QoS for real-time applications. An API is used to manage the NI resources/services. The Authors evaluate the NI area, without performance results related to QoS. The main drawback of the hardware approaches to ensure QoS is the lack of flexibility. Software policies, as the ones presented in this work, may be easily modified according to the applications running in the MPSoC.

Joven and Marongiu [JOV10] integrate hardware and software to provide QoS control at the application level in an *xpipe*-based NoC. The QoS architecture, implemented at the hardware level consists of control blocks that are implemented on the router to handle different QoS services: GS (guaranteed service) and BE (best-effort). The Authors use communication priorities (up to 8 priority levels) and GS through allocation of connection channels. The implementation at the software level uses an *openMP* framework, adopting an API that interacts directly with the NI. The NI inserts the priority in the header of the packets and is responsible for the initiation and termination of connections. The kernel provides abstract functions to send and receive messages, and to establish and close connections. The Authors employ a layered design to expose the hardware QoS features to the architecture, similar to our proposal, but the services are statically controlled by the application, without a monitoring scheme to adapt the system on the fly when the applications requirements are violated.

Motakis et al. [MOT11] explores the management of the NoC services to adapt the hardware resources through techniques implemented in software, allowing the user to explore the different network services through an abstract API. The implementation is based on the *Spidergon* STNoC platform. Lower-level layers are explored, along with a library of functions named *libstnoc* accessible to the user. The work focuses on dynamic reconfiguration of the communication services through the *libstnoc*. The designer can access information services (energy management, routing, QoS and security), and also enable and change these parameters through memory-mapped registers. The API can also perform a diagnostic service of the NoC traffic, changing QoS parameters at run time based in constraints defined by the user. As Joven and Marongiu [JOV10], the hardware is exposed to the application level. The Authors conclude their work mentioning the need to include NoC traffic monitoring services to ensure QoS.

Cui et al. [CUI12] propose a decentralized heuristic for task mapping. The proposal adopts a cluster-based method, implemented using the Tera Scale platform NoC. The clusters do not have a fixed size. Clusters can be resized at runtime, according to the characteristics of each application, and they can contain more than one application. The local managers control the cluster resizing in a decentralized fashion. The clustered approach achieves QoS by isolating applications, ensuring composability.

Liao and Srikanthan [LIA11] explore QoS through a hierarchical structure, dividing the MPSoC in clusters. Each cluster contains one application and a cluster manager. The system also has a global manager responsible for high-level tasks. The goal of the clustering heuristic, implemented in software, is to ensure at runtime the isolation of the traffic between different applications, favoring composability, an important feature for QoS.

Table 2 compares the reviewed works. In most works, [SAL11][FAN11][WAN12][WIS11] [KAK11][LUS11], the QoS management (2nd column of the Table) is made at the router level. Such approach takes local decisions, which may be inefficient at the system level. A centralized approach, as the one proposed in [WIN11], has a global view of the system, but scalability is sacrificed. In [MOT11], a scheme of QoS adaptation is implemented at the PE level, as in the present work, where the user informs the constraints. However, that proposal does not solve the scalability issue, and the work is tightly dependent to the *Spidergon STNoC*. A trade-off is achieved with a hierarchical approach, with several managers distributed in the system [CUI12][LIA11]. Our work may be applied to centralized or hierarchical systems, and is able to provide QoS adaptability.

Table 2 - State-of-the-art comparing works targeting QoS support in NoCs.

Proposal	QoS Management	Physical channel	QoS Technique	Adaptive technique
Wang (2012)	Router level	2 horiz., 4 vertic., and 4 diagonal	Dynamic Arbitration and Adaptive Routing	Congestion Aware Routing Algorithm
Wissem (2011)	Router level	One per direction	Flow Priority	No
Kakoee et al. (2011)	Router level	Duplicated channels	Flow Priority	No
Salah (2011)	Router level	One per direction	Flow Priority Virtual Channels	Flit Scheduling
Fan (2011)	Router level	Duplicated channels	Virtual Channels	Virtual channel allocation
Lusala (2011)	Router level	Duplicate channels	PS+CS (SDM)	No
Winter (2011)	Router level / Centralized	One per direction	Virtual Channels / CS	Central NoC Manager (HW)
Saponara (2012)	Network interface level	One per direction	Virtual Channels / admission control	Management API
Joven (2010)	Router and PE levels	One per direction	Flow Priority / CS	Management API
Motakis (2011)	PE level	One per direction	Bandwidth allocation	Management API
Cui (2012)	Hierarchical	One per direction	Composability	Reclustering
Liao (2011)	Hierarchical	One per direction	Composability	Cluster Allocation
This Proposal	**Hierarchical**	**Duplicated channels**	**Flow Priority / CS / Task Migration / Scheduling Priority**	**Monitoring and runtime QoS adaptation according to constraints**

A common choice in NoC designs is to interconnect routers using single bi-directional links (3rd column). An alternative to such approach, leading to good results, in terms of communication performance and QoS, is to increase the number of physical channels [FAN11][WAN12][KAK11]

[LUS11]. Such approach replaces VCs, with a smaller silicon cost, and may result in disjoint networks, enabling the use of priorities or simultaneous switching modes [KAK11].

As can be observed in the fourth column of the Table, approaches with QoS management at the router level adopt three main techniques to meet QoS: flow priorities [FAN11][WIS11] [KAK11] [JOV10], virtual channels (VCs) [SAL11][FAN11][WIN11] and circuit switching (CS) [WIN11][JOV10]. The hierarchical management [CUI12][LIA11] meets QoS favoring composability, i.e., applications are “isolated” in clusters, without the interference of other flows. The drawback of hierarchical approaches is to find continuous regions to map applications, even if available resources exist, and avoid the application fragmentation in dynamic workload scenarios.

As can be observed in NoC research arena, designers focus their efforts to develop mechanisms to provide QoS at the network level. However, MPSoCs start to employ NoCs as the de-facto communication infrastructure, adding a software layer over the network. Hence, the designers start to work on APIs to expose at the software level the QoS mechanisms. Currently the main gap observed in the literature is the adaptive techniques managed at runtime to meet QoS (5th column of Table 1). Works [SAL11][FAN11][WIS11][KAK11][LUS11] ensure QoS at the NoC level using synthetic traffic. Works [MOT11][JOV10][SAP12] expose the NoC level to the application, leading to the application developer the responsibility to select the most appropriate adaptive technique. The goal of works [CUI12] and [LIA11] is to isolate applications in clusters, to minimize the interference of other flows, ensuring QoS.

The originality of the present work is to select at runtime a given adaptive technique by using monitoring to detect violations, and a QoS Manager to control the flows. The work herein proposed adopts flow priority, CS, task migration and scheduling priority, controlled by a QoS Manager.

3 REFERENCE PLATFORM

The reference platform, developed in this work, integrates two existing ones: Distributed-HeMPS and QoS-HeMPS. This Chapter describes such platforms, and the integration between them, corresponding to the first contribution of the present work.

3.1 Distributed-HeMPS

The Distributed-HeMPS platform [CAS13] implements in the PE microkernel (small operating system) the clustering concept to distribute management actions. The platform assigns to the PEs three different roles: slave, local master and global master.

Observing Figure 8, each cluster contains a set of slave PEs (for user applications processing), managed by a local master (manager PE). The local master is responsible to implement heuristics as task mapping, task migration, and reclustering. Reclustering is a technique that allows the local master to borrow resources from neighbor clusters if the cluster does not have enough resources to execute a given application assigned to it. A resource is a processor page at each slave processor memory, which can allocate one task (note that is assumed a paged memory organization in this work). The cluster size is variable, because reclustering can increase the number of slave PEs managed by the local master.

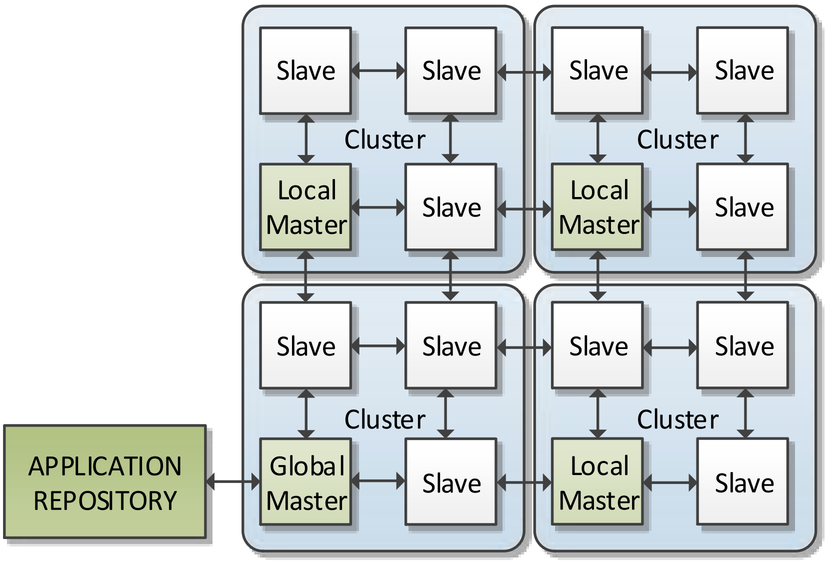


Figure 8 – Overview of Distributed-HeMPS platform. The cluster-based management is the main feature.

The platform has also one global master (global manager PE), which communicates with the local masters, characterizing a three level hierarchical management structure (global master → local masters → slaves). The global master implements the same functions as the local masters, and in addition accesses an external application repository to map applications to the clusters. The application repository is a memory that stores applications that can enter in system (workload). As soon as the global master receives an application request of a local master, it accesses the repository and transfers the object code of application (task object code) directly to the slave PE that will run the task.

The Distributed-HeMPS uses the Hermes NoC [MOR04], configured with 16-bit physical channels; XY routing algorithm; input buffering; credit-based flow control; round robin arbitration; and wormhole packet switching. The PE connected to each NoC router contains: a 32-bit Plasma processor (MIPS-like architecture); a local memory; a DMA (Direct Memory Access) module; and a NI (network interface).

3.2 QoS-HeMPS

Figure 9 presents overview of QoS-HeMPS [CAR11] platform. The QoS-HeMPS is a platform with centralized management, supporting QoS through differentiated communication services. The platform assigns to PEs two different roles: slaves and one global master. Slave PEs execute user applications, and their microkernel implements the software layer for QoS services. An API exposes to the application developer the control of differentiated communication services, implemented in NoC and NI. The single global master accesses the external repository, and implements protocols for task allocation and task termination in the slaves PEs.

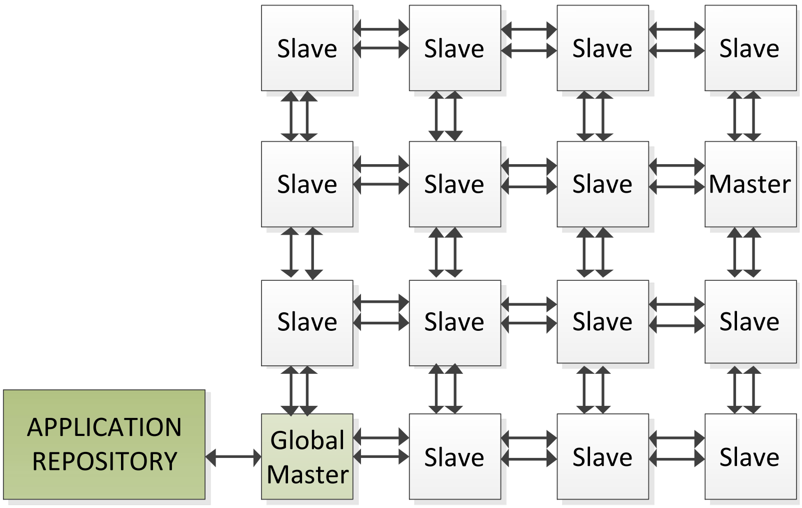


Figure 9 - Overview of QoS-HeMPS platform. The QoS support is the main feature of the platform.

The QoS-HeMPS NoC implements differentiated communication services to enable QoS support. The relevant features of this NoC are: (*i*) duplicated 16-bit physical channels, assigning high priority to channel 0 and low priority to channel 1 (high priority packets may use both channels); (*ii*) deterministic Hamiltonian routing [LIN94] in channel 1 and partially adaptive Hamiltonian routing in channel 0; (*iii*) simultaneous packet and circuit switching; (*iv*) unicast and multicast transmission; (*v*) input buffering; (*vi*) credit-based flow control; (*vii*) round robin arbitration.

Except for the NI, the hardware of the PE is practically the same as the Distributed-HeMPS. The difference comes from one mapped-memory register to signalize to the microkernel an incoming packet requisition made by CS. The NI implements a QoS treatment, in order to inject the packet into the appropriate physical channel.

3.3 Developed Platform: QD-HeMPS

A new platform was developed in the scope of this work, integrating the two previous platforms, named QD-HeMPS.

To accomplish such integration the main executed tasks were:

- Translation from VHDL to RTL-SystemC of the QoS-HeMPS NoC, and the corresponding validation through simulation;
- Translation from VHDL to RTL-SystemC of the QoS-HeMPS NI, and the corresponding validation through simulation;
- QoS software layer adaptation and validation in the slave, local and global Distributed-HeMPS microkernels;
- API adaptation to expose the QoS services to user.

The first two tasks were necessary because the Distributed-HeMPS was modeled in RTL-SystemC and the QoS-HeMPS was modeled in VHDL. SystemC was also chosen since it enables fast simulation, keeping the accuracy of the RTL VHDL model.

During this process, some errors in the Distributed-HeMPS were identified, being reported to the research group, and corrected. The backward compatibility was kept, and an application that runs in the Distributed-HeMPS executes in the new platform, without modifications. Figure 10 depicts the QD-HeMPS. The new QD-HeMPS includes the distributed management features of the Distributed-HeMPS and the duplicated physical channels, the main feature of QoS-HeMPS.

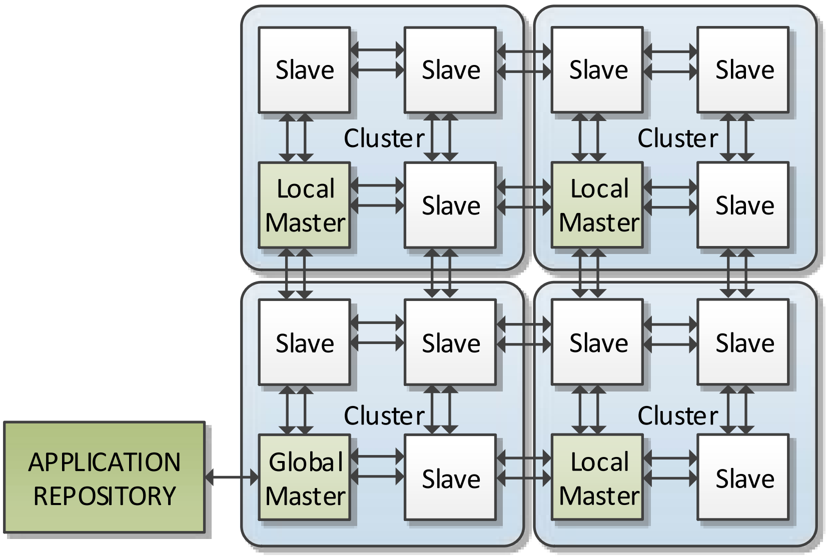
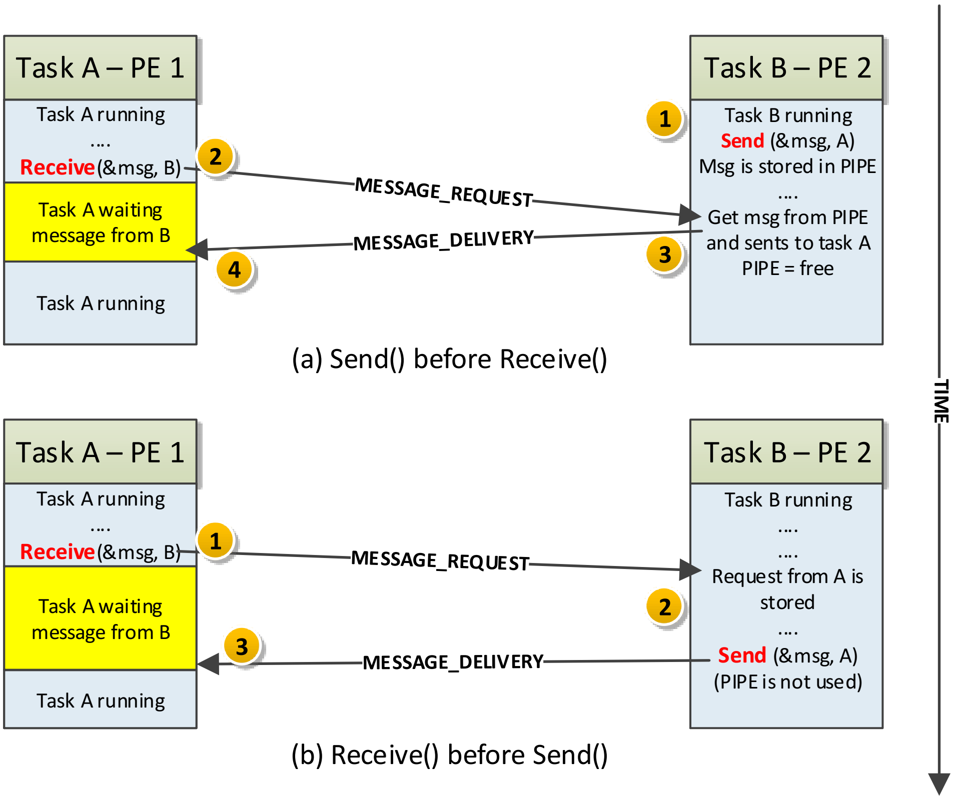


Figure 10 – QD-HeMPS Overview. QoS-HeMPS and Distributed HeMPS Integration.

3.3.1 Tasks Communication

The task communication model is the same in both platforms (Distributed and QoS HeMPS), and was kept in the QD-HeMPS. In this model (MPI-based), tasks communicate using *Send()* and *Receive()* primitives. The Send() primitive is called by the producer task, and the Receive() primitive is called by the consumer task.

The protocol is the *read-request*, i.e., a message is injected in the network only when the producer task has received a message requisition. This requisition corresponds to a control packet named MESSAGE_REQUEST. A MESSAGE_REQUEST packet is generated by a consumer task when it calls the *Receive()* primitive. The OS of the consumer task blocks the task, putting it in WAITING state, up to the reception of a MESSAGE_DELIVERY packet. The MESSAGE_DELIVERY is a data packet that is generated by the producer task in two situations: (*i*) *Send()* called before a Receive(); (*ii*) *Receive()* called before a *Send()*. Figure 11 describes the task communication model, explaining the possible two scenarios.



(a) Send() before Receive()

(b) Receive() before Send()

Figure 11 –Communication Model (MPI-like) adopted in the platforms.

Figure 11(a) presents a *Send()* called before a Receive(). It is supposed that task A is running in PE 1, and task B in PE 2. In step 1, task B executes a *Send()*, producing a message to task A. At this point, the microkernel searches for the respective MESSAGE_REQUEST, but no requisition is found because task A did not called the *Receive()* primitive yet. Hence, the message produced by task B is stored in a buffer named *pipe*, and task B keeps running normally, *characterizing a non-blocking send*. In step 2, task A executes a *Receive()*, generating a MESSAGE_REQUEST packet to task B. Task A is blocked, going to a WAITING state, *characterizing a blocking receive*. When PE 2 receives the requisition packet in step 3, the microkernel finds the message stored in the pipe, and sends it to task A through a MESSAGE_DELIVERY . Finally, in step 4, the microkernel of PE 1 handles the received MESSAGE_DELIVERY packet, transferring the packet to the memory space of task A, and re-schedules task A for execution (READY state).

Figure 11(b) presents a *Receive()* called before a *Send()*. In step 1, task A executes a *Receive()*, generating a MESSAGE_REQUEST packet that is sent to task B, and the task A switches to the WAITING state. In step 2, the packet is handled by the microkernel of PE 2, which searches for the respective MESSAGE_DELIVERY, but no message is found because task B did not called the *Send()* primitive yet. Thus, the microkernel stores this MESSAGE_REQUEST. In step 3, when task B calls the *Send()* primitive, the microkernel finds the stored MESSAGE_REQUEST, and the message produced by task B is directly forwarded to task A, without storing it in the pipe. When task A receives the MESSAGE_DELIVERY, the microkernel releases task A and restores its execution.

4 DEBUGGING TOOL FOR MPSoCs

This Chapter describes a tool to debug MPSoC platforms, corresponding to the second contribution of the present work.

Challenges in MPSoC software development include [GRA12]: inter-processor communication, protocol stacks definition, OS porting and analysis and exploration of better programming model facilities to address parallel programming.

Due to the inherent non-deterministic behavior of MPSoCs, they are prone to concurrency bugs. During the software development, the support of debugging tools is mandatory to find such bugs, deadlocks and bottlenecks. Given a composition of subsystems running different portions of an application, communicating with each other in a variety of ways, using a number of different ISA (Instruction Set Architecture) processors and running tasks written in multiple programming languages using multiple programming models, the task of understanding the system behavior to track down the root cause of a problem, poses a formidable challenge to concurrent debugging tools [MAR06]. Debugging tools for MPSoC may be grouped in the following classes:

- *simulators* allowing the use of waveforms, assertions, code coverage, etc. Such simulators enable accurate hardware verification, but have limited support to software debugging;
- *parallel software development environments* [MUR11][MUR12][PRA11][HED11], which enable application debug abstracting the hardware through virtual platforms;
- *NoC debuggers*, enabling to evaluate the NoC performance from synthetic traffic generators or application traces [MOL10].

Due to the huge design space to explore, each class has specific goals: *simulators* target hardware development; *software environments* evaluate the correct execution of parallel applications, allowing the detection of concurrency bugs (e.g. deadlocks and race conditions). NoC debuggers explore NoC architectural parameters. However, an important gap is observed concerning a complete NoC-based MPSoC: tools to inspect the traffic exchanged between processing elements (PEs) at a higher abstraction level, and not simply as raw data. The meaning of data exchanged in a multiprocessed system is crucial to enable the events sequencing during the debug process.

Therefore, the proposed debugging tool aims to define *a new class of debugging tools*, able to trace the messages exchanged between PEs, enabling the debug of the simulation results at the protocol level. The implemented environment of such debug class performs as *Wireshark*, a free and open-source packet analyzer used for network analysis, and development of software and communication protocols.

The motivation to propose this new debug class is to enable fast evaluation of communication APIs, mapping heuristics, monitoring schemes, QoS services, NoC routing algorithms. In summary, its goal is to make available to MPSoC designers a way to inspect the NoC traffic generated by the developed software at a higher level, keeping the confidence of the RTL simulation. The implemented environment presents the following features: (*i*) *generality*: it is possible to use it with different NoC-based MPSoCs, due its modular platform integration; (*ii*) *non-intrusiveness*: the added code to trace packets does not interfere with the applications and network performance, different, for example, of task to include log message generators in the source code; (*iii*) *simulator independent*: the traffic trace may be obtained from any simulator; (*iv*) *configurability*: the environment may be easily configured for different MPSoC parameters.

The proposed debugging method relies in two steps: (*i*) generation of a compact traffic trace with all packets transmitted through the NoC (Section 4.2); (*ii*) a set of tools to analyze the traffic trace (Section 4.3).

This debugging tool was used to implement, validate, and collect the results of all proposed techniques presented in the next Chapters.

4.1 MPSoC Architecture Assumptions in the context of the debug environment

An MPSoC contains of a set of PEs interconnected by a communication infrastructure. This work assumes NoCs as the communication infrastructure, due to their scalability and performance compared to buses. The following assumptions, commonly found in MPSoCs [GAR13], are adopted:

- each PE contains at least one processor with a private memory;
- a 2D-mesh NoC topology;
- applications modeled as task graphs;
- communication model is message passing;
- a multi-task operating system (OS) runs at each PE;
- a mapping function assigns tasks to PEs.

The NoC-based MPSoC is assumed modeled at the RTL-level (using e.g. SystemC, VHDL, Verilog), and should be simulated with a clock-cycle accuracy. Such requirement enables to accurately compute throughput, execution time of applications, and the traffic distribution in the NoC links.

The debug environment requires two files to describe an MPSoC (Figure 12), one with the platform configuration, and the other one with the set of available services.

Figure 12(a) presents one example of the platform configuration file. The *router_addressing* is related to how the routers are numbered, being possible to use a Hamiltonian sequence [LIN94], or XY coordinates. The subsequent parameters configure the MPSoC size, and if a clustering management is used, as in [SHA11][CAS13a], the parameters allow the cluster configuration. The clock period is used to compute the throughput. After the initial parameters, the set of tasks to be executed in the MPSoC are listed as a tuple [*task name, identifier*].

An MPSoC works by exchanging messages between PEs. Such messages correspond to specific protocols, such as communication between tasks (MPI-like behavior), or management functions, as task mapping or monitoring. Figure 12(b) presents the service description file, which contains all supported services by the MPSoC platform. The first lines of the file contain a tuple [*service name, identifier*]. The two last lines contain reserved words (starting with '$'), used to identify the services related to task mapping and end of task execution. With such information, it becomes possible to debug the traffic with the service name, instead of display the identifier stored in the packets.

```
router_addressing        hamiltonian
mpsoc_x                  6
mpsoc_y                  6
cluster_x                3
cluster_y                3
manager_position_x       0
manager_position_y       0
global_manager_cluster   0
clock_period_ns          10

BEGIN_task_name_relation
...
FIR                      774
iquant                   257
ivlc                     258
idct                     260
...
END_task_name_relation
```

(a) platform configuration file

```
MESSAGE_REQUEST              10
MESSAGE_DELIVERY             20
TASK_ALLOCATION              40
TASK_TERMINATED              70
TASK_DEALLOCATED             80
LOAN_PROCESSOR_RELEASE       90
DEBUG_MESSAGE                100
...
...
UPDATE_CS_CONTROL            280
QOS_REQUEST_SERVICE          290
MONITORING_PACKAGE           300
MIGRATION_CODE               320
MIGRATION_TCB_MEMORY         330

$TASK_ALLOCATION_SERVICE     40
$TASK_TERMINATED_SERVICE     70
```

(b) service description file

Figure 12 - Files used to describe the MPSoC platform.

Therefore, an important feature of the proposed environment is to be *generic*, being possible to use it with different MPSoCs types, by modifying the configuration files.

4.2 Packet Logger

A second important feature of the debugging environment is its non-intrusiveness. A packet logger, specific to each MPSoC implementation, is instantiated at each router, reporting in a compact way relevant data for all received packets. Each line of the generated log file contains the information about one packet, with the following parameters:

***tick* | *router*$_{cur}$ | *service* | *size* | *bandwidth* | *port*$_{cur}$ | *tgt* | *task*$_{ID}$**

where:

- *tick*: clock cycle number related to the reception of the packet header in a router port;
- *router*$_{cur}$: address of the router receiving the packet;
- *service*: function of the packet (integer);
- *size*: size of the packet in flits;

- *bandwidth*: number of clock cycles required to receive the packet, i.e., the amount of clock cycles the link is reserved for the packet;
- *port$_{cur}$*: port from which the packet was received in *router$_{cur}$* ;
- *tgt*: address of router to which the packet must be sent;
- *task$_{ID}$*: optional parameter, corresponding to the task identifier number, required when the packet contains the task allocation or task finish services.

The *service* parameter corresponds to the packet function. In MPSoCs there are data and management packets. Data packets correspond to the messages exchanged between tasks at the application level. Management packets are treated at the OS level, and correspond to messages such as start task mapping, task migration, processor initialization, among other services.

A given *service* is part of a given *protocol*. For example, when a new application starts, there is a set of packets exchanged between a manager PE and slave PEs, which will receive the application tasks.

The generated report makes possible to debug the MPSoC at the *protocol* level step by step. The simulation of a set of applications running simultaneously in an MPSoC requires millions of clock cycles, and dozens of signals to be monitored. Debugging with waveforms is unfeasible. With the proposed debugging environment, the designer can monitor the NoC traffic, inspecting for example only specific services corresponding to the protocol under development.

The result of the *packet logger* must be a unique file. In the present implementation, the *packet logger* is instantiated at each NoC router, corresponding to a SystemC module with less than 80 lines. The *packet logger* is an abstract code and is not synthesized. This logger is instantiated in each router, but each instance writes in a single common file. Thus, this file contains the log of all routers, and concentrates the information that will be read by debugging tool.

4.3 Debug Environment

Figure 13 presents the simulation and debugging flow. The MPSoC simulation requires an RTL description, and object code of the operating systems and tasks. The result of the simulation can be verified with waveforms (as mentioned, practically unfeasible for large MPSoCs), or in an efficient way using the proposed debug environment. The debug environment requires three inputs: (*i*) the *traffic log* file created by the *Packet Logger*; (*ii*) the *platform configuration* file; (*iii*) the *service description* file. Simulators as ModelSim (Mentor) and NCLaunch (Cadence) generate the *traffic log* only at the end of the simulation. The SystemC simulator generates the log file during the simulation, enabling to use the debugging tools concurrently with the simulator. In this way, the debugging tools acts as a GUI for the SystemC simulator.

The debug environment contains three main *views*: (*i*) throughput view; (*ii*) load view; (*iii*) mapping view, described in the next Sections

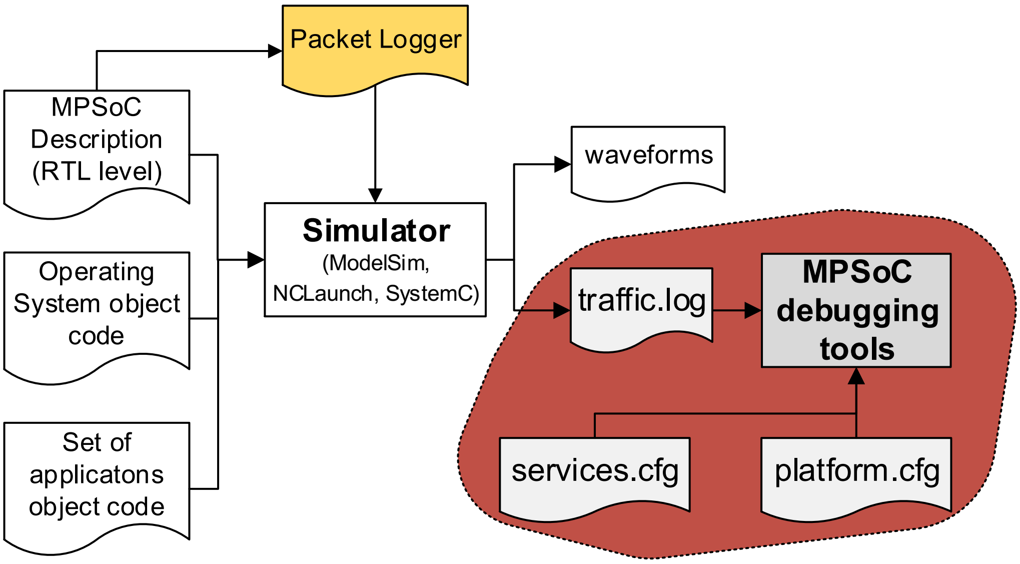
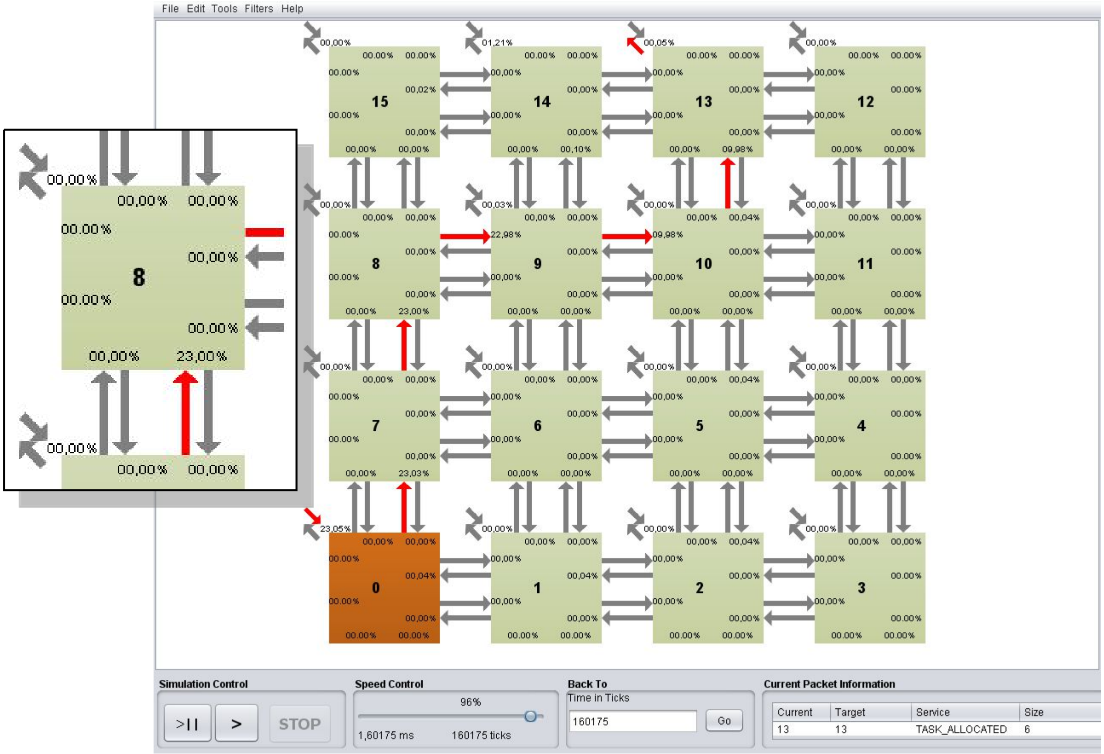

Packet Logger
MPSoC Description (RTL level)
Operating System object code
Set of applicatons object code
Simulator
(ModelSim, NCLaunch, SystemC)
waveforms
traffic.log
MPSoC debugging tools
services.cfg
platform.cfg

Figure 13 - Simulation and debugging flow.

4.4 Throughput View

Figure 14 illustrates the *throughput view* GUI, for a 4x4 MPSoC instance. The debugging environment assumes a 2D-mesh topology. Modifying the GUI can easily circumvent this restriction. The debug environment supports single and duplicated physical channels. There is no restriction regarding the number of virtual channels.



Simulation Control

Speed Control
96%
1,60175 ms 160175 ticks

Back To
Time in Ticks
160175

Current Packet Information

Current	Target	Service	Size
13	13	TASK_ALLOCATED	6

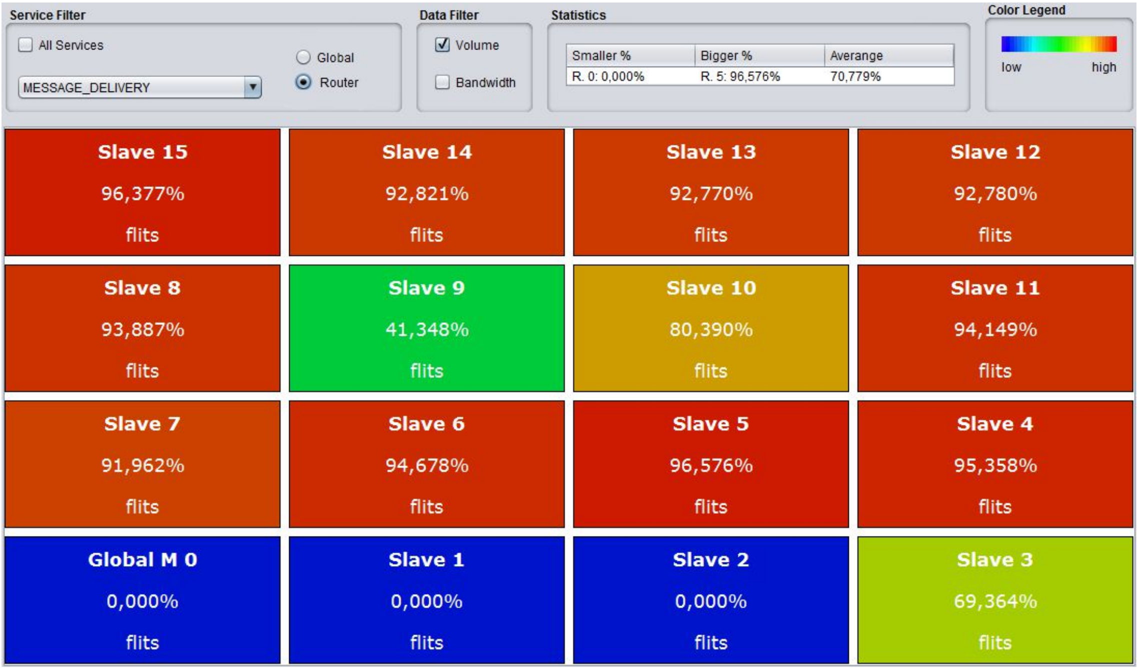
Figure 14 - *Throughput view* GUI, for a 4x4 MPSoC, with duplicated physical channels. PE0 is the manager PE. Router 8 is highlighted, with an average link use of 23% in one of the south ports.

This view enables the designer to select the speed to display the packets, and supports breakpoints at specific services. Each packet read from the traffic log file highlights the link being used (red arrows from PE 0 to PE 13). When the packet reaches its destination the path from the source to the target router returns to gray. Besides inspect packets from specific protocols with breakpoints, this view displays the throughput at each link, as highlighted in Figure 14. A parameterizable window interval counts the number of received flits at each input port (*bandwidth* field of the traffic log). A detailed report is created, with the throughput per link per time window.

Therefore, such view provides an overall traffic view, which helps the designer to validate protocols, to measure the load induced by specific services/protocols, to identify hotspot regions, and to validate routing algorithms (the path taken by packets is highlighted).

4.5 Load View

The goal of this view is twofold. First, it enables to evaluate *globally* the percentage of packets for a given service at each router. For example, it is expected that most packets related to task mapping have their source at the manager processor. Secondly, it enables the evaluation at the *router* level. Figure 15 illustrates the percentage of the reception of data packet at the router level. For example, 92.8% of the packets traversing PE 14 are data packets. It is possible to inspect in the "throughput view" the percentage of such packets addressed to the processor (local port).



☐ All Services
MESSAGE_DELIVERY
○ Global
● Router
☑ Volume
☐ Bandwidth

Smaller %	Bigger %	Averange
R. 0: 0,000%	R. 5: 96,576%	70,779%

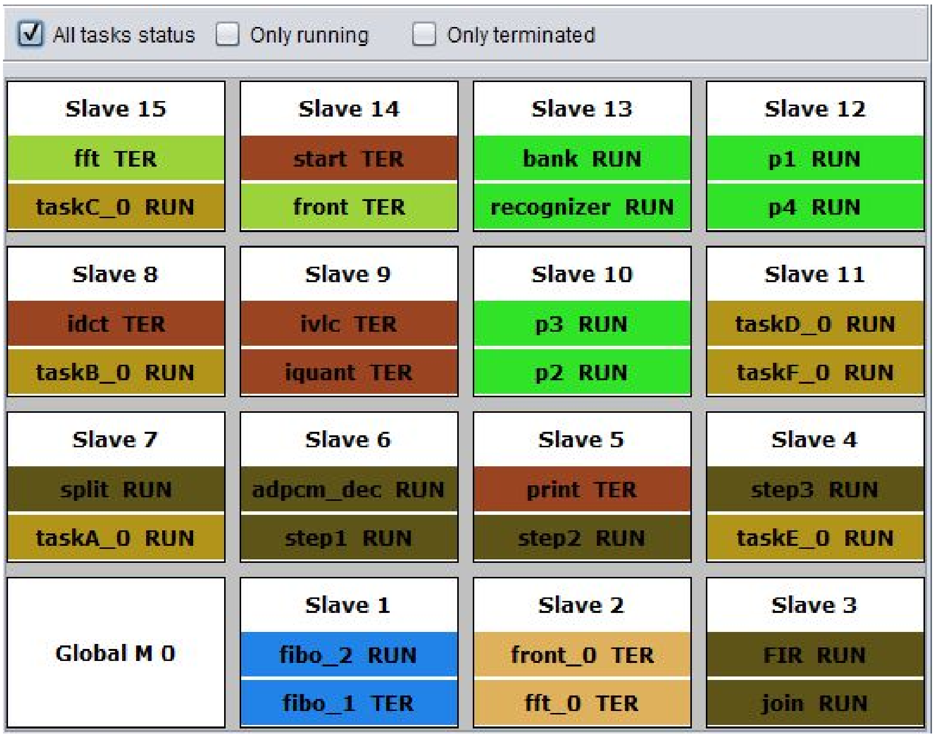
Slave 15 96,377% flits	Slave 14 92,821% flits	Slave 13 92,770% flits	Slave 12 92,780% flits
Slave 8 93,887% flits	Slave 9 41,348% flits	Slave 10 80,390% flits	Slave 11 94,149% flits
Slave 7 91,962% flits	Slave 6 94,678% flits	Slave 5 96,576% flits	Slave 4 95,358% flits
Global M 0 0,000% flits	Slave 1 0,000% flits	Slave 2 0,000% flits	Slave 3 69,364% flits

Figure 15 - Load view for the message delivery service, evaluating the percentage of received packets at each router corresponding to the reception of data packets.

Thus, this view enables to evaluate the impact of a given service or protocol in the traffic generated through the execution of applications. For example, if the MPSoC designer is implementing a given monitoring protocol to ensure QoS, it is possible to measure the impact of such protocol in overall traffic.

4.6 Mapping View

This view enables the designer to monitor how applications are dynamically mapped to the MPSoC. Figure 16 illustrates it. The DTW (dynamic time warping) application (in light green) uses PEs 10, 12 and 13. This view also shows the task status, i.e., running or terminated. Hence, this view is an important tool to evaluate mapping heuristics [SIN13]. Task mapping is a key research topic in MPSoCs, since its quality defines the performance and the power consumption in the NoC.



☑ All tasks status □ Only running □ Only terminated

Slave 15 fft TER taskC_0 RUN	Slave 14 start TER front TER	Slave 13 bank RUN recognizer RUN	Slave 12 p1 RUN p4 RUN
Slave 8 idct TER taskB_0 RUN	Slave 9 ivlc TER iquant TER	Slave 10 p3 RUN p2 RUN	Slave 11 taskD_0 RUN taskF_0 RUN
Slave 7 split RUN taskA_0 RUN	Slave 6 adpcm_dec RUN step1 RUN	Slave 5 print TER step2 RUN	Slave 4 step3 RUN taskE_0 RUN
Global M 0	Slave 1 fibo_2 RUN fibo_1 TER	Slave 2 front_0 TER fft_0 TER	Slave 3 FIR RUN join RUN

Figure 16 - Mapping view, for a scenario with 7 applications. The OS running in the slave PEs enable the execution of two simultaneous tasks. The Global PE manages the MPSoC.

4.7 Case Study using the Proposed Debug Environment

A simple case study is presented to show the use of the proposed debug environment. The protocol under evaluation is presented in Figure 17, corresponding to the migration of task t_i running in PE 8 to PE 4. The task migration protocol is particularly detailed in section 6.3. The manager PE sends a "*task migration request*" to PE 8, with the new t_i position. The OS running in PE 8 (OS_8) sends only the text area (instructions) to PE 4, without interrupting t_i – service "*migrate code*". When the OS_8 verifies that t_i may be effectively migrated, the data area is transferred to PE 4 – service "*migrate data*".

To debug this protocol, the user defines breakpoints for the 3 services used in the protocol. Once defined the breakpoints, the debug starts. When a given packet is read from the *log* file, the debug stops, and in main view (Figure 18 left) the user captures the time the packet was injected into the NoC.

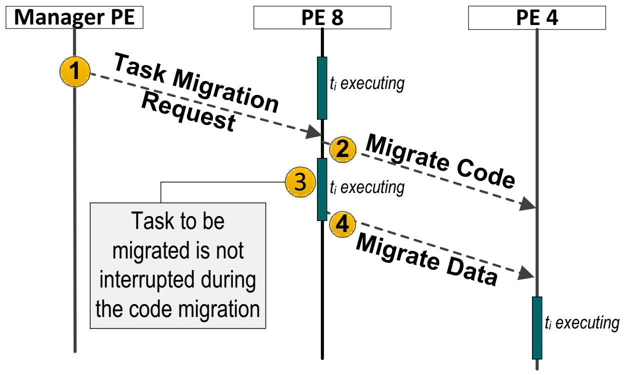
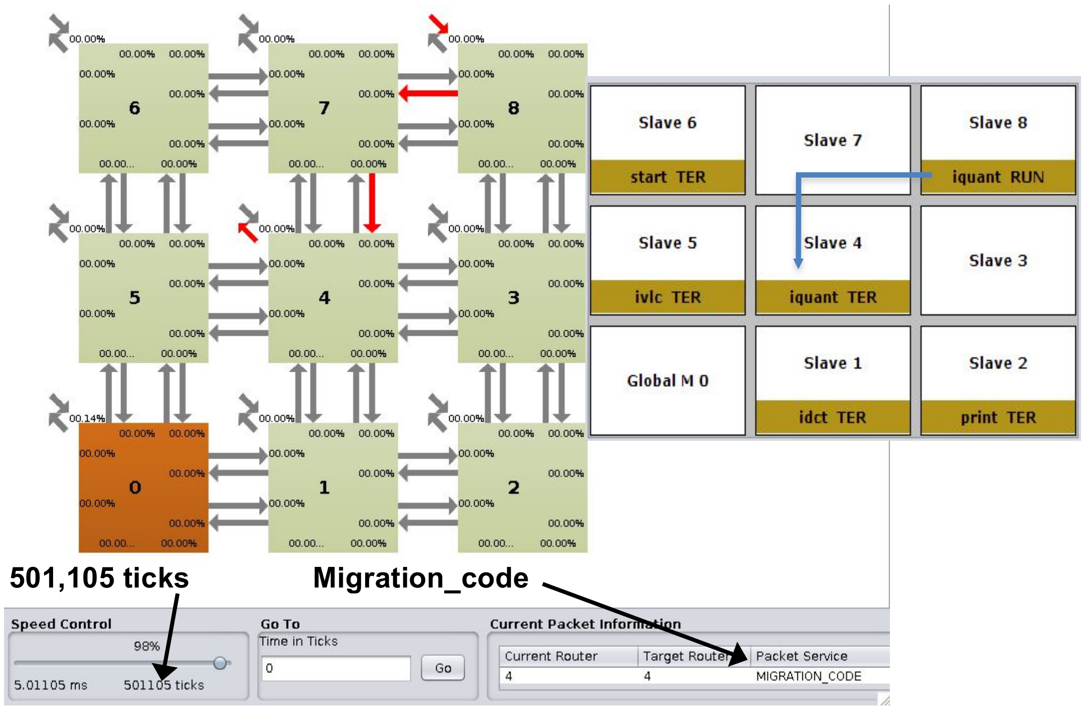


Figure 17 - Task migration protocol.



Slave 6 start TER	Slave 7	Slave 8 iquant RUN
Slave 5 ivlc TER	Slave 4 iquant TER	Slave 3
Global M 0	Slave 1 idct TER	Slave 2 print TER

Speed Control
98%
5.01105 ms 501105 ticks

Go To
Time in Ticks
0

Current Packet Information

Current Router	Target Router	Packet Service
4	4	MIGRATION_CODE

Figure 18 - Debug process for task migration protocol, with task *iquant* (MJPEG application) migrated from PE 8 to PE 4.

The delay of each service of Figure 17, in clock cycles (*cc*), for the *header flit*, corresponds to:

1. task migration request: 500,084 to 500,104 cc, 6-flits packet;
2. migrate code: 501,095 to 501,105 cc, 758-flits packet;
3. 16,518 cc – time the task execute during the code migration;
4. data migration: 517,613 to 517,623 cc, 1,066-flits packet.

Thus, the total time to execute the task migration is 17,539 *cc* plus 1,066 *cc* (corresponding to the size of the last packet). It is also possible to debug the path taken by the packets, the percentage of the links usage during migration, and to inspect the task mapping before and after migration (Figure 18 right), with t_i corresponding to task *iquant.*

It is important to mention that the protocol evaluation was carried out with no software instrumentalization, as most works in the state-of-the-art require.

5 MONITORING

This Section describes the monitoring implementation, used to guide the QoS support mechanisms. This work implements and evaluates two monitoring organizations: hybrid and software. The motivation to explore two monitoring strategies is to compare the features of each monitoring implementation, applying them in the adaptive QoS management. The hardware implementation was discarded, because the adaptive QoS management needs that the monitoring observes the platform at task level (software).

5.1 Hybrid Monitoring

The hybrid monitoring includes hardware and software modules. This scheme combines the fast response and low intrusiveness of hardware monitoring with the reuse and system view of software monitoring. The implemented monitoring allows the QoS manager to gather the amount of necessary data to execute the heuristics of QoS adaptation at runtime.

To enable monitoring, the first action is to obtain the application profile (section 5.1.1). This action enables the programmer to define monitoring thresholds, which are used to detect *violations* and generate *events*.

- *Violations* - corresponds to a “fine grain” treatment of the monitoring information. Violations are detected when threshold violation occurs. The monitors store the number of violations, and when a parameterizable number of violations is reached an event is created.
- *Events* - corresponds to a “coarse grain” treatment of the monitoring information. The QoS Manager handles the events, triggering QoS adaptations.

The action of accumulating in throughput and latency monitors a parameterizable number of violations before creating an event reduces the processing induced by the monitoring, since events are not created at each violation. Three violations are adopted as the default threshold number of violations to create an event. Experiments showed that this value safely estimates an event, because it can suppress random picks of latency in the network or processors, while keeping a high level of confidence in local monitors.

Figure 19 shows an overview of the proposed monitoring scheme, considering a 4x4 MPSoC instance, split in to four 2x2 clusters. The monitoring modules are implemented in slave and master PEs (cluster managers). Slaves PE contain the *Packet Monitor* (section 5.1.2). Master PE contains the *Throughput and Latency Monitors* (Section 5.1.3). The *QoS Manager* is also implemented in master PEs, receiving events from the Throughput and Latency Monitors. The QoS Manager is the event manager [KOR13] in this proposal, and is detailed in Chapter 7.

The dotted arrow in Figure 19 corresponds to the monitoring traffic. The Packet Monitor observes data packet coming from the router local port, extracts its latency and size, and sends this information in a small packet to the Throughput and Latency Monitors. Throughput and Latency Monitors handle packets received from Packet Monitors and generate events if necessary.

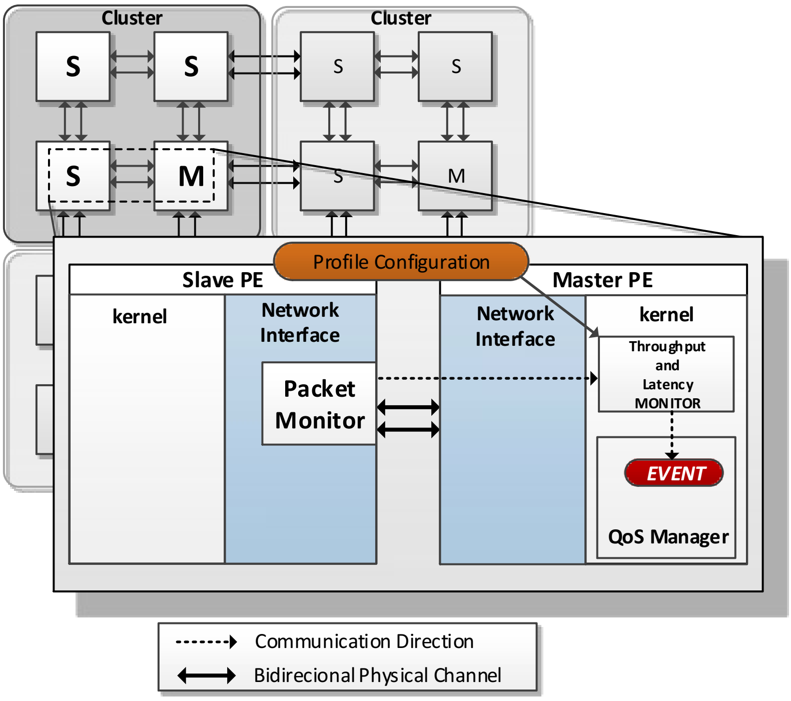


Figure 19 - Proposed monitoring system. 'S' corresponds to slave PEs, and 'M' to master PEs.

This monitoring architecture is classified as hierarchical, with two layers. The lower layer comprises slave PEs, which generate monitoring information to the second layer, the cluster master PE, which implements the Throughput and Latency Monitors and QoS Manager.

5.1.1 Profile Configuration and Monitoring API

In the profile configuration, the application developer executes the RT application without disturbing traffic to verify if the platform can deliver the expected performance to meet the application constraints. Throughput and latency values obtained during the profile step correspond to the best results that the application can achieve in the platform, and are the basis to obtain the monitor thresholds. Thus, the monitoring system can detect violations, notifying the QoS Manager.

The monitoring system was designed to support monitoring at the task level. To enable the monitoring the user must inform the communicating task pair that will have the threshold and latency monitored. Taking as example the task communication graph showed in Figure 20, the user can choose to monitor only the communications task pairs T1→JOIN and T2→JOIN, because these task pairs are critical to display (to the end-user), results of the application. If it is necessary a more detailed QoS control, the user can also choose to monitor T1→T2 and T2→T1 pairs. Thus, each communicating task pair can receive their own constraints.

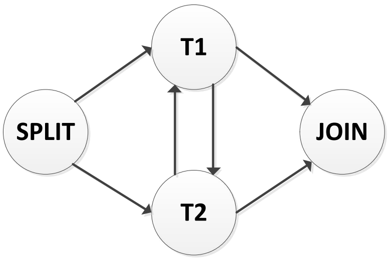
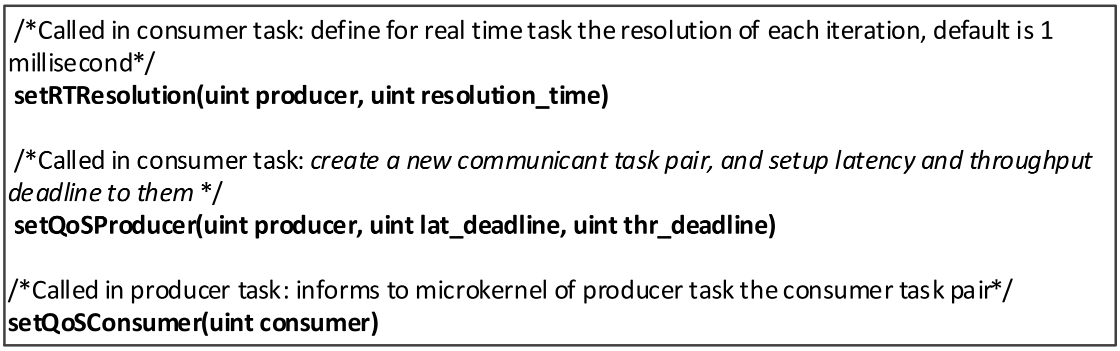


Figure 20 – Example of task graph.

The API of the slave PE was extended in the QD-HeMPS to include system calls used to enable and configure the monitors. Figure 21 shows the new system calls that are called by the monitored communicating task pair.



```
 /*Called in consumer task: define for real time task the resolution of each iteration, default is 1
millisecond*/
 setRTResolution(uint producer, uint resolution_time)

 /*Called in consumer task: create a new communicant task pair, and setup latency and throughput
deadline to them */
 setQoSProducer(uint producer, uint lat_deadline, uint thr_deadline)

/*Called in producer task: informs to microkernel of producer task the consumer task pair*/
setQoSConsumer(uint consumer)
```

Figure 21 – System call to configure task monitoring.

The new system calls are:

- **setRTResolution**: called by the consumer task, defines the resolution time (in clock cycles) of the monitoring window. This resolution time is the period for each round of the Throughput monitor. The latency monitor does not use this resolution time. The latency monitor computes the *network latency*, i.e., the amount of clock cycles necessary to transfer the packet header from the source to the target router. This system call is optional, and if not called, the monitors use the default resolution time of 500.000 clock cycles (5 ms@100MHz). The application developer should know the system frequency to define the correct value of the *resolution_time.*
- **setQoSProducer**: called by the consumer task, enables the communication monitoring between a *consumer* task (caller of this system call) and a *producer task*. This system call defines the latency (in clock cycles) and throughput (bits per resolution time) deadlines. When this system call is executed by the microkernel, one packet with these deadlines is sent to the cluster QoS Manager.
- **setQoSConsumer**: called by the producer task, informs to the microkernel of the producer task who is the consumer task. This system call is used to add at each packet transmitted to the *consumer* task a flag. This flag indicates that the packet must be monitored. In addition, it defines to the microkernel a communicating pair with QoS constraints.

5.1.2 Packet Monitor

The *Packet Monitor* is implemented in hardware, in the network interface (NI) of the slave PEs. The Packet Monitor observes all data packets (MESSAGE_DELIVERY packets) entering in the local port with a monitoring bit enabled in the packet header. This bit is set in the producer task when a new packet is sent to the consumer task. This procedure only occurs if the *setQoSConsumer* system call is called in the producer task. Thus, the microkernel of the producer task knows who is the consumer task to enable the monitoring bit.

The function of Packet Monitor is to extract from the data packet the *producer task id*, *consumer task id*, *size* (in flits), and *latency* (in clock cycles). The packet size and the tasks' ids (producer and consumer) are already in the packet. To obtain the packet latency, a timestamp flit was added in original data packet. This timestamp flit is generated during the packet creation in the producer task. According to Figure 22, the monitor reads the packet entering the local port, and extracts the required information. The *Packet Monitor* (in consumer task PE) computes the latency through the difference between the current timestamp and the timestamp stored in the packet.

After extracting the data, the monitor assembles a monitoring packet and sends it to it QoS Manager, in the cluster master PE. The monitoring packet contains a header (cluster master address), and a payload with 8 flits (4 32-bit words).

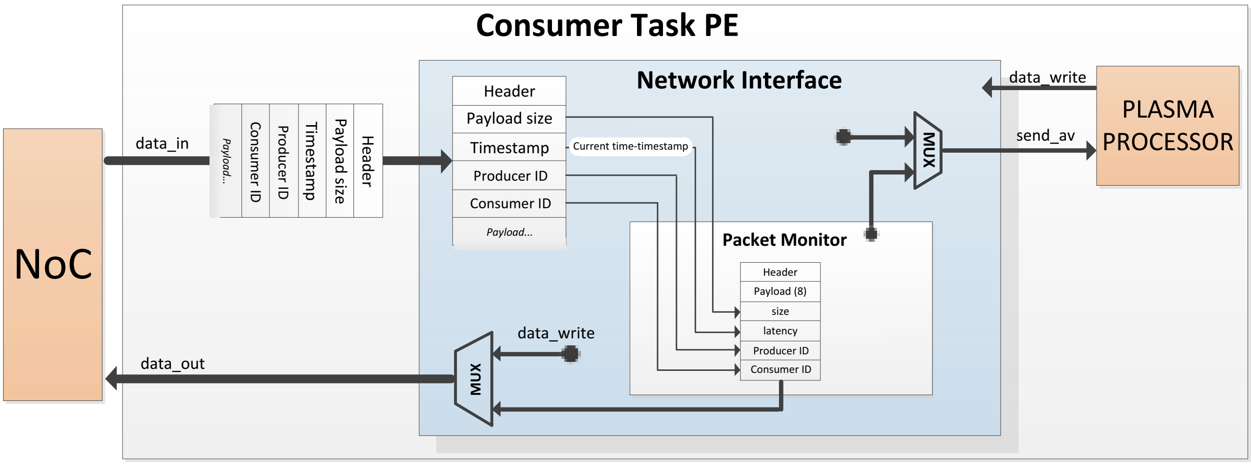


Figure 22 – Packet Monitor Overview.

The monitoring packet is transmitted using the NoC. Therefore, the *data_out* signal is multiplexed with the PE. When the PE is not injecting data, the packet monitor may transmit the monitoring packet to the master PE, forcing *send_av* to low, to avoid a data injection from the PE.

An important metric considered for the packet monitor implementation is the area overhead. Table 3 presents the area cost of Packet Monitor over the NI area and over the Plasma area (MB-Lite ISA, DMA and NI). The overhead is measured adding the packet monitor area to the reference design. The area overhead can be considered small due to two reasons: (*i*) the processor (Plasma) is very simple, using much less area compared to other embedded processors

[WAC11]; (*ii*) the memory area, dominant component of the PE area, is not considered in this result.

Table 3 – Packet Monitor area overhead

	FLOPS	LUTS
Reference NI:	121	386
Reference PE:	988	2887
Packet Monitor:	187	184
NI Overhead:	154,5%	47,6%
Plasma Overhead:	18,9%	6,3%

5.1.3 Throughput and Latency Monitors

The *Throughput and Latency Monitors* are software implemented in the master PE. The *Throughput Monitor* counts the number of received bits within the monitoring resolution time. When a monitoring packet coming from the *Packet Monitor* is received, the master microkernel identifies the communicating task pair and increments the throughput counter according to the packet size. When the resolution of the *Throughput Monitor* window expires, the monitor verifies if the throughput deadline was violated, i.e., a throughput smaller than the specified (each communicating task pair has a specific deadline, defined with the *setQoSProducer* system call). After a parameterizable number of violations, the monitor generates a throughput event to *QoS Manager*.

After receiving a monitoring packet, the master microkernel identifies the communicating task pair and calls the *Latency Monitor*. The *Latency Monitor* verifies if the latency carried in monitoring packet is higher than the latency deadline (defined in the *setQoSProducer* system call), generating a latency violation if it is true. After a parameterizable number of violations, the monitor generates a latency event to *QoS Manager*.

5.2 Software Monitoring

The software monitoring is similar to the hybrid monitoring implementation. As illustrated in Figure 23, the difference is where the *Throughput and Latency Monitors* are implemented, and the absence of the *Packet Monitor*. In this scheme, the *Throughput and Latency Monitors* are implemented in the slave PE microkernel, sharing CPU time with user tasks (higher intrusion compared to the hybrid implementation). In this approach, the function of the *Packet Monitor* is executed in the microkernel. The slave microkernel extracts the latency and the size of the data packet, increments the throughput counter, and calls the Latency Monitor. The Latency Monitor compares the extracted latency with the latency deadline. When the round of Throughput Monitor ends, the slave microkernel also calls the Throughput Monitor which compares the throughput counter with the throughput deadline. After three violations, both monitors generate one event to QoS Manager.

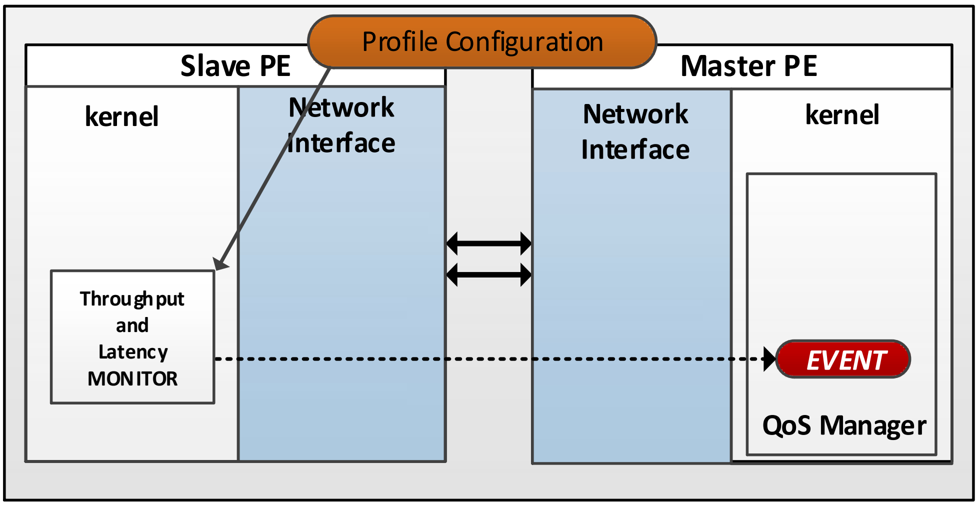


Figure 23 - Overview of software monitoring implementation.

In the previously hybrid scheme, the extracted information of the received data packets is sent to the QoS Manager into remote master PE, that increments the Throughput counter and calls the Latency Monitor. Therefore, the hybrid scheme generates more monitoring traffic than the pure software monitoring.

From one side the hybrid scheme is less *computation intrusive* but generates more monitoring traffic, therefore it is more *communication intrusive*. On the other side, the software monitoring is less *communication intrusive* because generates a smaller monitoring traffic, but its more *computation intrusive* due to CPU load required by software implementation. A monitoring evaluation of both implementations is presented in next subsection.

5.3 Monitoring Evaluation

The goal of the monitoring evaluation is to analyze the impact of monitoring implementations considering two performance parameters: application execution time and link utilization. The main performance parameter is the application execution time, but the link utilization is an important metric to identify possible communication bottlenecks. As the monitoring sends information packets (hybrid monitoring) and events (software monitoring) to the QoS Manager, the evaluated links are the input links of cluster masters (links that have the highest concentration of the monitoring traffic). All performed simulations use an RTL cycle accurate description.

Two scenarios were evaluated. The first scenario uses real applications and a full cluster occupation, aiming to evaluate a worst-case monitoring scenario using only real applications. All RT communicating task pairs have the monitoring enabled. This scenario uses a 6x6 MPSoC instance, with four 3x3 clusters, with two mapped tasks at each PE. The second scenario uses synthetic applications, aiming to stress at maximum the communication (with synthetic applications it is possible to obtain higher injection rates than real applications). This scenario uses a small 3x3 MPSoC instance, with 1 cluster, and with three tasks per PE.

5.3.1 Results for scenario with real applications

In this scenario, nine RT applications instances are mapped: 3 MJPEG, 1 audio_video, 3 DTW, and 2 FFT. These RT applications were mapped with synthetic applications without monitoring enabled. The system calls for latency and throughput to all RT communicating pairs were defined in such a way to generate violation for all transmitted packets. Figure 24 presents the execution time for the RT applications in three situations: *(i)* without monitoring; *(ii)* hybrid monitoring; *(iii)* software monitoring.

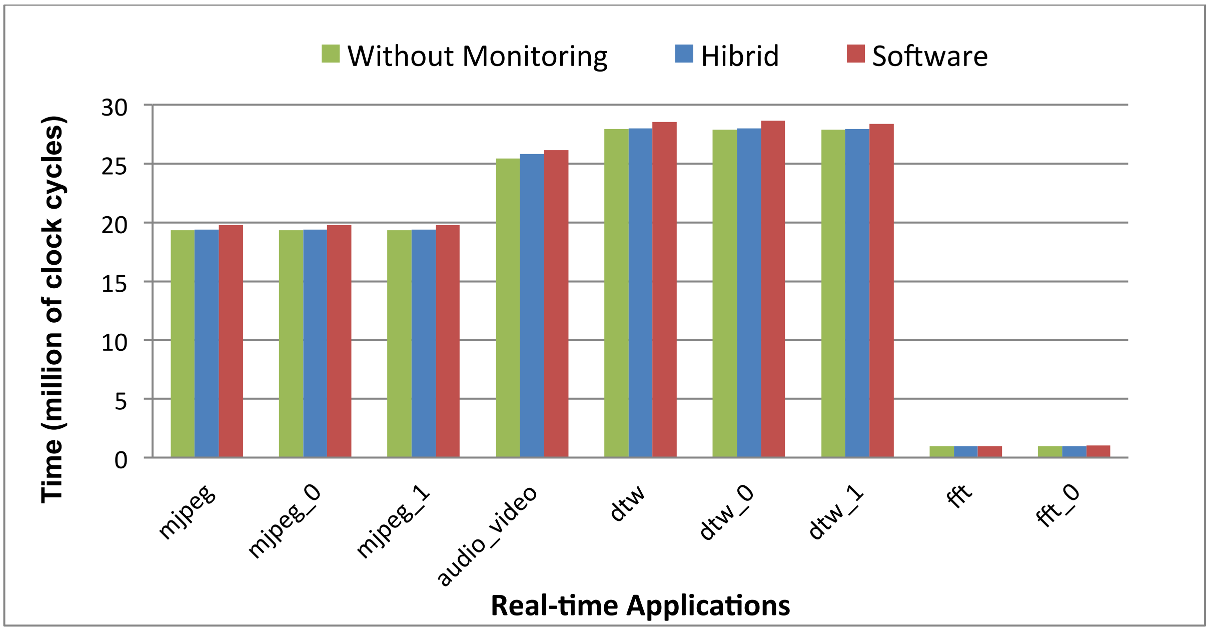


Figure 24 – Execution time comparison in real scenario.

The hybrid monitoring induces a small increase in the execution time. For example, the audio_video application presented an overhead of 1.5% compared to the system without monitoring. However, for all other applications, the runtime overhead is minimal, corresponding to a maximum of 0.6% compared to the execution without monitoring.

The software monitoring presented a higher penalty in the execution time compared to the execution without monitoring, 2.5% in average, due to the CPU sharing between tasks and monitors in slaves PEs.

Next, Figure 25 and Figure 26 present the link utilization. The observed links are the ones (East, West, North, South) of each cluster master (C. Master). Each curve corresponds to the total bandwidth usage (BW_{east}+ BW_{west}+ BW_{north}+ BW_{south}) for each cluster master. Figure 25 presents the link utilization with hybrid monitoring. An average of 0.086% was obtained, with picks of ~0.3% of utilization in each cluster master. Figure 26 presents the link utilization with software monitoring. As can be observed the link utilization is low, corresponding to an average of 0.036% and with picks of ~0.1% of utilization in each cluster master.

As expected, the hybrid monitoring has a smaller impact in the execution time compared to the software implementation (Figure 24), however increases the NoC usage.

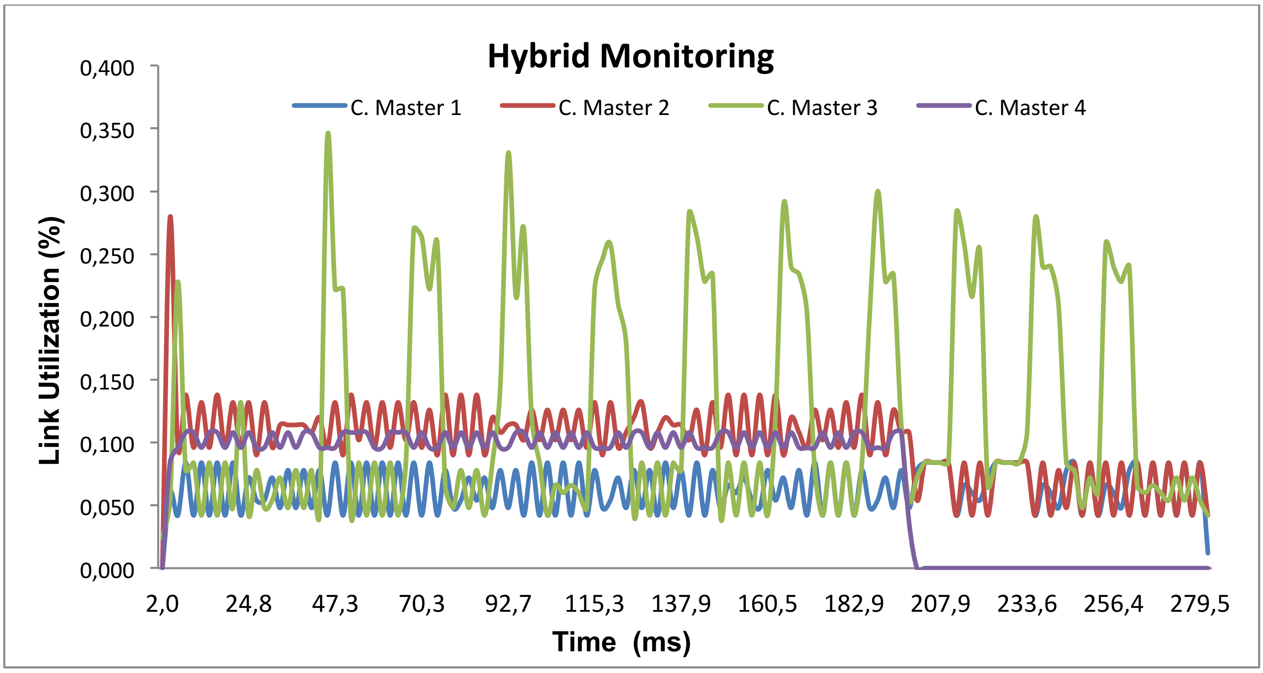


Figure 25 – Hybrid Monitoring link utilization.

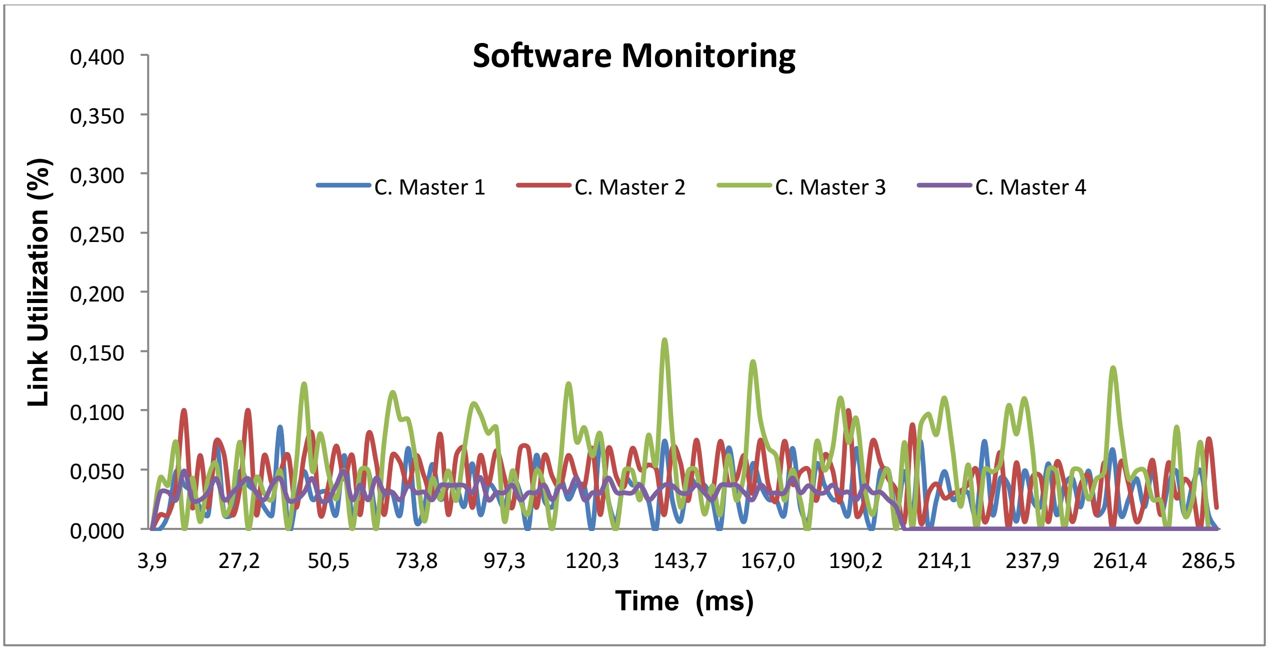


Figure 26 – Software Monitoring link utilization.

5.3.2 Results for scenario with synthetic applications

This scenario uses three synthetic applications, all containing six tasks. As in the previous scenario, the deadline system calls were defined to generate events for all transmitted packets.

Observing Figure 27 it is possible to note the execution time difference among software monitoring, hybrid monitoring, and the same test case without monitoring. The software implementation presented an important runtime overhead, in average 10.6% compared to the execution without monitoring. The reason explaining this result comes from the required time to execute the monitoring process in software, which is most frequent than the first scenario since synthetic applications have a much higher injection rate than real applications. On the other side, the hybrid implementation presents a lower intrusion, corresponding to an overhead smaller than 2% compared to the execution time without monitoring.

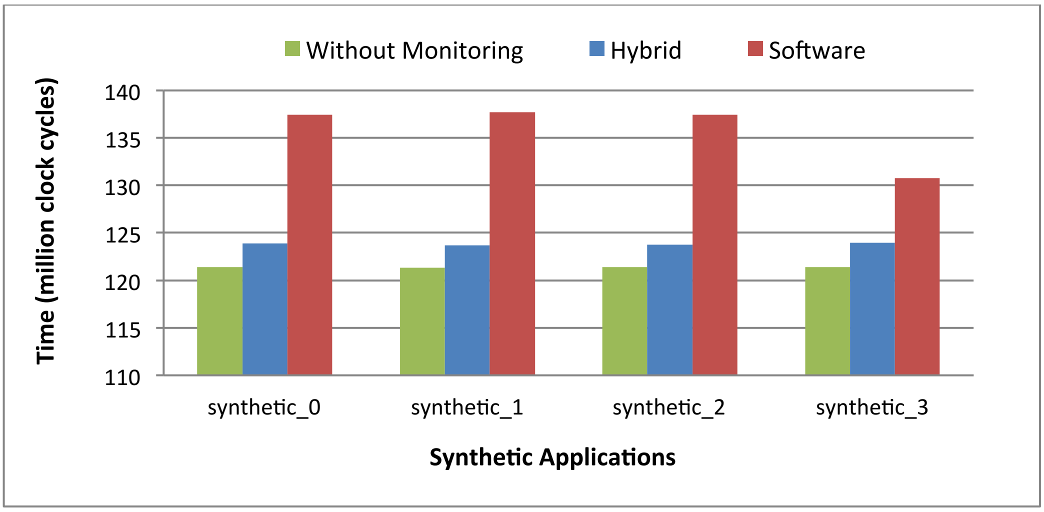


Figure 27 – Execution time comparison in the synthetic scenario.

Figure 28 presents the link utilization comparison. As expected the software monitoring presents a lower link utilization, ~0.2%, while the hybrid implementation the link utilization is in average 0.8%. Note that both monitoring implementations do not reach 1% of link utilization, which can be characterized as low intrusive methods.

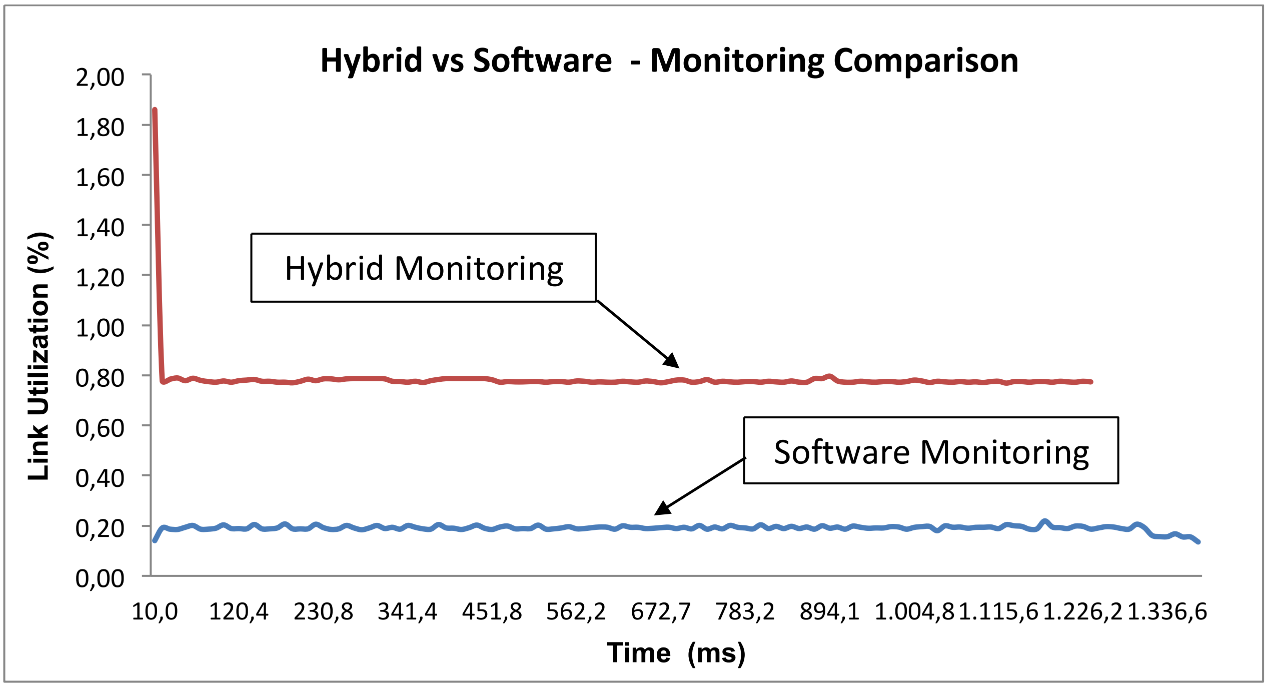


Figure 28 – Hybrid vs. Software monitoring comparison.

5.3.3 Conclusion

Although hybrid monitoring presents higher link utilization in both scenarios, the software monitoring penalizes the execution time. As the execution time is the main parameter to consider, and the link utilization obtained with the hybrid monitoring do not achieves 1% of link, even in a worst-case synthetic scenario, it is possible to conclude that the hybrid monitoring is the recommend choice.

However, this hybrid implementation is not completely non-intrusive, since the monitored data compete with data packets. Future work may addresses new monitoring models and architectures with the goal to eliminate this intrusion, by using for example a dedicated NoC for monitoring.

6 ADAPTIVE QOS TECHINIQUES

Due to the increasingly number of PEs in MPSoCs, programing a platform with support to QoS techniques is a challenge task. The application programmer must have a deep knowledge of the platform structure, to employ at design time the QoS techniques, to fulfill the real time requirements, and try to reduce the interference of other applications that may be running simultaneously in the platform. Therefore, runtime mechanisms are required, where QoS is adaptively managed on the fly, converging to a state in which inter-applications interference is minimized, and the system requirements can be achieved.

This Chapter describes the adaptive QoS techniques employed in this work. Four QoS techniques are addressed: *(i)* adaptive flow priority adaptation (coupled to adaptive routing); (*ii)* dynamic CS (circuit switching); *(iii)* task migration; *(iv)* task scheduling priority. All the adaptive QoS techniques were developed in the scope of this work, except the task scheduling priority technique, described in [MAD13]. Furthermore, a cluster defragmentation technique is exposed in the last section of this Chapter. This technique is an additional technique to the four adaptive QoS techniques. The cluster defragmentation is an original proposal of this work, and acts at runtime to defragment the system through task migrating tasks mapped in other clusters.

Table 4 summarizes the QoS techniques addressed in this work. The characteristics of these QoS techniques generate effects in order to improve both computation and communication. Task scheduling may improve the computation, while flow priority may improve the communication. CS affects the communication, reserving network resources to avoid disturbing from other flows, and task migration may influence in both computation and communication to restore the application performance.

Table 4 - Effect of QoS techniques in communication and computation.

	Effect of the QoS technique in:	
Technique	Computation	Communication
Flow priority	-	application traffic is prioritized – soft guarantee
Circuit Switching	-	exclusive reservation of network resources - hard guarantee
Task Scheduling	increase the CPU time, by reducing the processor sharing among several tasks	-
Task Migration	move a given task to a free PE (load balancing)	move a task to a non-congested NoC region

Comparing Figure 29 to Figure 19 (proposed monitoring system), an *Adaptation Module* was included in the slave PEs. This module implements the protocols to execute the QoS techniques after receiving an adaptation packet from the *QoS Manager*. The adaptation packet contains the request to execute one of the four QoS techniques, and the necessary information to execute it.

Note that all computation cost to select which QoS technique will be adopted is left to the *QoS Manager*. Therefore, the *Adaptation Module* receives the adaptation packet, extracts the information stored in the packet, and configures the microkernel to execute the required adaptation.

The next sections describe each one of the four adaptive QoS techniques, and the Cluster Defragmentation is described in the last section.

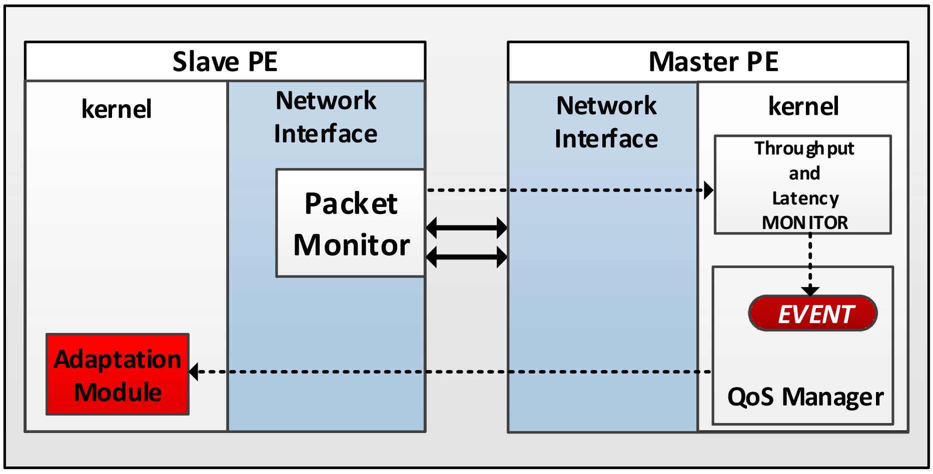


Figure 29 - Overview of QoS adaptation. The *QoS Manager* receives an event, chooses the appropriated QoS technique, and sends the adaptation order to the *Adaptation Module*.

6.1 Adaptive Flow Priority

Flow priority is a communication QoS technique. In the QoS-NoC [CAR11], each router has duplicated physical channels at each port. Such approach separates the traffic by creating two disjoint networks, dividing the flow in two classes, high and low priority through channels 0 and 1, respectively. Both channels use packet switching (PS). Channel 0 can also support CS.

The packets are injected in network with a bit in the header indicating its priority. This bit is configured in the kernel, and is used by the NI to insert the packet in the correct channel. During the routing, it is used by the routers to apply a differentiated routing algorithm according to the packet priority. A low-priority packet uses the minimal deterministic version of the Hamiltonian routing algorithm. A high-priority packet uses the non-minimal partially adaptive version of the Hamiltonian routing algorithm. Therefore, high priority packets may explore non-congested NoC regions, delivering the packets with a smaller latency. In the QoS-HeMPS the packet priority is defined at design time. The application programmer specify the priority of each message in the *Send()* primitive.

The present work adopts a different approach. Assuming a communicating task pair, the default flow priority used by the producer task is low. The priority is runtime adapted when a packet is received in the Adaptation Module from the QoS Manager to increase or decrease the priority.

In order to enable the runtime adaptation, the producer task must inform its consumer task. This information is defined by using the *setQoSConsumer* primitive. This primitive informs to the producer microkernel that the producer task forms a communication task pair with the consumer task. Thus, for each packet to be transmitted, the microkernel before injecting a packet into the NoC, verifies which is the flow priority configured by Adaptation Module and configures the next packets with this priority enabling or disabling the header priority bit.

6.2 Dynamic Circuit-Switching

Dynamic CS is another communication QoS technique. CS is established using channel 0. CS reserves a path between the producer and consumer PEs. Therefore, the CS flow can ensure maximum throughput, since it cannot be disturbed by other flows. In the *QoS-HeMPS* the CS is defined at design time. The application programmer must establish and release the CS manually, through specific system calls that inform the establishment and release of CS to a given flow.

The approach adopted in the present work is to establish and release CS at runtime. Differently of the priority adaptation, witch simply informs the new packet priority, the adaptive CS requires the synchronization of packets produced and consumed by the communicating task pair. Synchronization means that all packets produced in a given switching mode must be consumed before changing the switching mode. This synchronization is necessary because the packet structure changes according to the switching mode.

The changing of the switching mode adopts the protocol depicted in Figure 30.

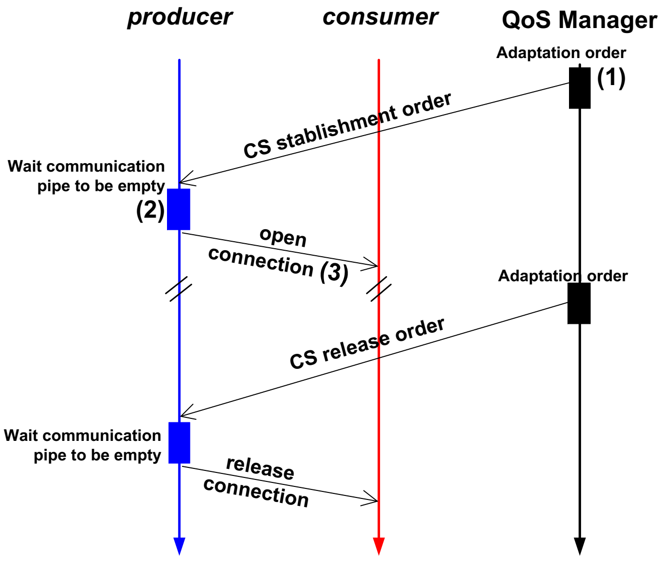


Figure 30 – Protocol to establishment and release of CS by *QoS Manager*.

The following sequence of events occurs to establish a CS connection. The same process is repeated to release a CS connection:

1. The *QoS Manager* sends a CS establishment order to the Adaptation Module of the producer task PE (event 1 in Figure 30).

2. The *Adaptation Module* in the producer task receives the adaptation order, and configures the microkernel to execute the packet synchronization. The synchronization is executed blocking the producer when calling the *Send()* primitive (normally non-blocking) until the consumer task read all packets stored in producer PIPE (event 2 in Figure 30).
3. After synchronization, the producer task opens a connection with the consumer task (event 3 in Figure 30). A control packet is sent in PS mode to the consumer task, allocating resources in the path. All subsequent data packets are transmitted throughout the established connection.

It is important to mention that all attempts to establish CS succeed. This is ensured by the QoS Manager, which keeps a data structure with all existing connections, verifying the feasibility of the new connection. This is an important difference to the state of the art, since most proposals try to establish the connection without ensuring its feasibility.

When a connection is established between a producer/consumer pair, the resources in the path stay allocated until a new adaptation order to release the path comes from the QoS manager. The connection ensures QoS, offering a maximum throughput and minimum jitter, since the communication uses a reserved path.

6.3 Task Migration

Task migration is a communication/computation QoS technique. As illustrated in Figure 20, applications are modeled using task graphs. During task mapping, some tasks may be mapped far from each other due to the unavailability of PEs resources. When resources become available, sometimes it is possible to move communicating tasks closer to each other. An important benefit of the task migration, besides performance improvement, is the energy reduction spent in the communication between tasks. In [MOR12] a task migration heuristic is detailed, with the following features: (*i*) complete task migration, including code, data and context; (*ii*) do not require migration checkpoints, i.e., the task may be migrated at any moment; (*iii*) in-order message delivery, i.e., tasks communicating with the migrated tasks will receive the messages in the order they were sent. The cost-function of this heuristic is to reduce the communication energy.

The present work optimized the task migration protocol proposed in [MOR12]. The proposal divides the amount of data to be migrated by memory sections, and migrates each memory section separately. Figure 31 details the new process. In event 1 of the Figure, the task migration process begins with the adaptation order that is received by the *Adaptation Module* coming from the *QoS Manager*. The adaptation order contains the identification of the task to be migrated and the target PE that will receive the task. The *QoS Manager* is responsible to select the best processor to migrate the task according to the adaptive QoS heuristics described in Chapter 7. When the processor receives the migration request, the *Adaptation Module* configures the kernel to immediately send the object code of task to a new processor (event 2). Next, it is verified if the remaining data memory segments (*bss*, *data* and *stack*) can be sent. If this verification returns false, the task remains running (event 3). To migrate the data memory segments the following condition must be satisfied:

- The state of the task must be READY, meaning that task is not *waiting* for a message delivery from another task, and that task is not *running*. In the READY state, the task context is safely stored in the TCB (task control block).

If the task is in WAITING state, this means that the task executed a *Receive()*, and is blocked waiting a data packet from some given task. Migrating the task in such state will incur in a message loss, since the producer will send the message to a PE without the corresponding task to consume it. We considered the cost to bypass the message very high and too susceptible to induce errors, so it was decided to migrate a task only when in READY state. If the task is in RUNNING state, the processor registers are in use. Therefore, the task must to be in the READY state to have it context stored in the TCB.

Therefore, if the task is not in the READY state, the remaining step 4 of the migration process is paused, and the task keeps running normally. The first preemption of the task to the READY state, after code migration, triggers the migration of the data memory segments (event 4 in Figure 31), concluding the task migration process.

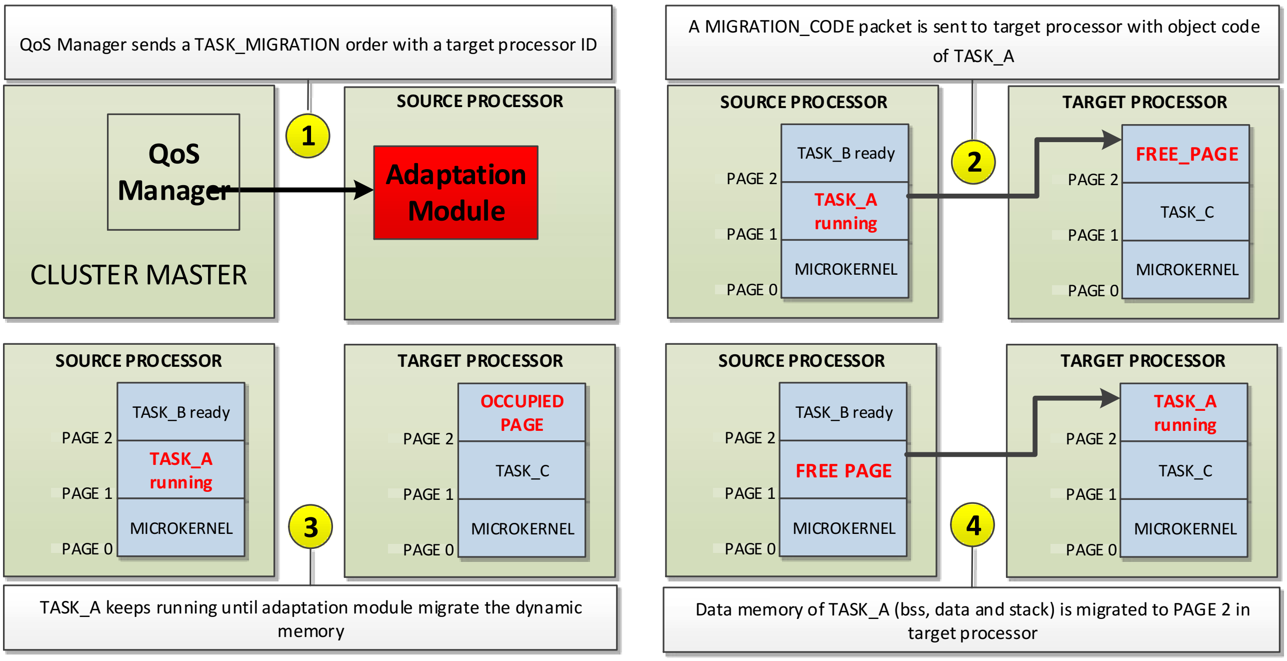


Figure 31 – Optimized task migration protocol.

An important issue in task migration is the synchronization of messages exchanged between communicating tasks. An error can occur when the message request or message delivery are not sent to the correct processor.

If the *consumer task is migrated*, all MESSAGE REQUEST packets arriving in the producer task update its task mapping table, with the new address of the consumer task. In such a way, all delivered messages are correctly transmitted to the new address.

If the *producer task is migrated*, two situations may arrive:

- If there are messages in the communication pipe in the original position of the producer, the consumer task will consume the messages from the old addresses, without update the producer address (such situation does not postpone the task migration);
- When the communication pipe in the original producer address becomes empty, a received MESSAGE REQUEST is forward to the new producer address. The migrated task (in a new address) delivers the requested message, with its new position, allowing the consumer task to update its task mapping table.

The advantage of the optimized task migration protocol is the smaller volume of information transmitted through the NoC. Comparing the optimized protocol to the original one [MOR12], with a page size equal to 16 Kbytes, the proposed migration protocol reduces the migration time, in average, by 69,5% (~960 clock cycles) against the reference protocol (~3150 clock cycles). In the reference protocol, the entire task page is migrated independent on the task code or data sizes hence, the advantage of the presented protocol may be even higher with larger pages.

6.4 Processor Scheduling

The OS scheduler supports multi-task execution. The HeMPS MPSoC adopts a round-robin police, with each task executing for a given period, named *time-slice*.

The QD-HeMPS adopts the Round-Robin Scheduling with Priority-based Preemption [LI03]. The processor is also allocated according to the round-robin policy. However, when a suspended task with higher priority than the current running one becomes ready to run, the former preempts the latter. In addition, the processor time slice for each task can be set at design time. Such approach adds a certain degree of priority since the time slice for each task can be set according to its processing requirements. A preempted task has its remaining time slice stored in its TCB, which is restored when the task is re-scheduled.

The TCB contains a variable to store the scheduling priority. All tasks are initialized with default and equal priority and time-slice. When the *Adaptation Module* receives the adaptation order to change the scheduling priority, it configures the TCB increasing simultaneously the priority and the time-slice (plus 60% [MOR12]).

6.5 Cluster Defragmentation

Castilhos [CAS13b] propose a reclustering heuristic based in the loan of resources between clusters. When a new application is requested to execute in the MPSoC, the Global Master searches a cluster with resources to execute the application. If no cluster has enough resources to allocate all tasks of the application, the selected cluster is the one with most available resources. The mapper (which runs mapping heuristic), executed in the Cluster Masters, can borrow resources from another cluster of the MPSoC, and allocate tasks in these borrowed resources.

When tasks executing outside of the original cluster finish their execution, the borrowed resources return to the original cluster.

The present work added a runtime verification of borrowed resources in the Cluster Masters, to defragment applications. Any task mapped in the cluster notifies the Cluster Master when its execution finishes. This corresponds to a free resource in the cluster. When receiving a free resource notification, the microkernel on the Cluster Master first verifies if there are RT tasks outside of the cluster, and then BE tasks. In an affirmative case, the task is migrated to the cluster, contributing to defragmenting the system.

The cluster defragmentation is a technique that is independent from the monitoring proposed in Chapter 5. The cluster defragmentation is an important feature added to the distributed system management.

7 RUNTIME ADAPTIVE QOS MANAGEMENT

This Chapter presents the heuristics of runtime adaptation, employed to choose the adaptive QoS technique by the QoS Manager. The *QoS Manager* module is implemented in the Cluster Master microkernel.

All heuristics are triggered with an event received by the QoS Manager, coming from the monitors. The monitoring generates latency and throughput events. The QoS Manager receives events from the monitors at runtime, selects an adaptive QoS technique, and notifies it to the Adaption Module. The QoS Manager assumes an application as a set of communicating tasks (at least two). Therefore, the management treats the application at the task level, analyzing the performance of each communicating task pair and applying QoS techniques when necessary.

The QoS Manager handles each received event individually:

- **Latency** events trigger flow adaptations: due the latency monitoring having a high sensibility of the communication behavior;
- **Throughput** events trigger computation adaptations: throughput events are more severe than latency events because real-time applications are typically throughput sensitive and variations are easily perceptible by the user.

The first set of adaptive QoS techniques implemented in this work corresponds to flow adaptation techniques: adaptive flow priority and dynamic CS. The main impact of these techniques is in the communication performance. Next, it is addressed the CPU adaptation techniques: processor scheduling priority and task migration. The main impact of these techniques is in the computation performance.

Figure 32 shows how the runtime QoS management works. When new events are received, the application goes from an optimal state to a degraded state, up to a critical state. When latency events are received, the first action to restore the application state is to increase the flow priority. If latency events are still received, the flow goes to CS. When throughput events are received,

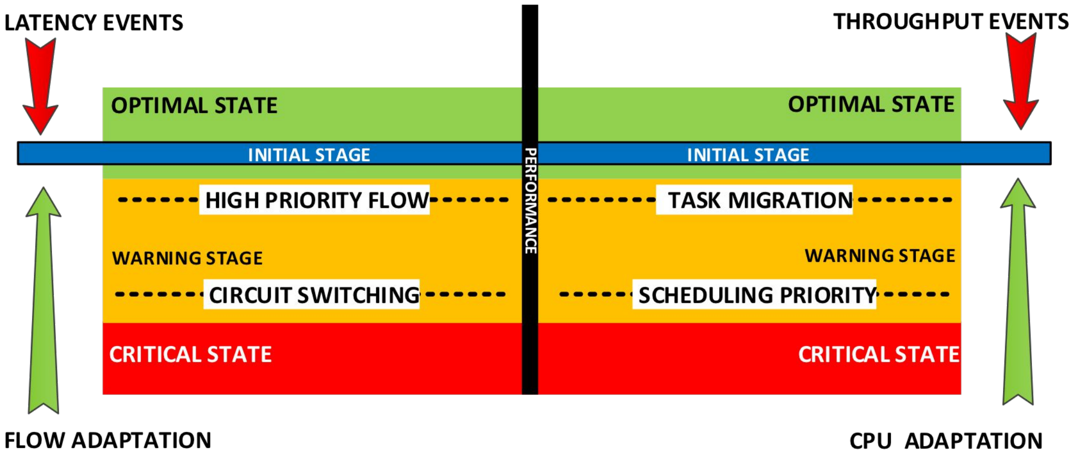


Figure 32 – Overview of the Runtime Adaptive QoS Management.

initially the task migration is tried and if it cannot be executed the processor scheduling priority is increased.

7.1 Flow Adaptation

The implemented flow adaptation described below employs runtime management using adaptive flow priority and dynamic CS.

When a latency event is received, the QoS Manager calls the *ChangeQoSFlow* function. This function is detailed in Figure 33. Its objective is to choose the appropriate flow adaptive technique to provide a communication QoS improvement. All flows are supposed to initiate using LOW priority (with deterministic routing). When a latency event is received the heuristic first increases the flow priority to HIGH (also changing to the partially adaptive routing), and if latency events are still received, the heuristic tries to change the switching mode to circuit switching (CS). If it is not possible change to CS, HIGH priority is kept. In this last case, no adaptation is made, because the flow is already in HIGH priority. Line 2 contains an instruction that will be explained jointly with explanation of next Figure (Figure 34).

```
1. ChangeQoSFlow(input: ctp) /*ctp is communicant task pair*/
2.   ctp_time_without_event <= 0
3.   IF ctp priority is LOW THEN
4.         Set ctp priority to HIGH
5.   ELSE IF Path_CS_Available(ctp) is true THEN
6.         Set ctp flow mode to CS
7.   END_IF
8. End ComputationAdapt
```

Figure 33 – Pseudocode for the flow priority and CS adaptation.

The function *Path_CS_Available (ctp)* (line 5) verifies if it is possible to establish a CS connection between tasks belonging to the communicating pair. The cluster master microkernel keeps a matrix with the state of all routers ports of the cluster. This state matrix is used as input for a software function that implements the partially adaptive Hamiltonian routing algorithm. The input of this function is the router addresses of the communicating tasks (ctp) that should use CS. If the algorithm finds a path, the state matrix is updated, and the function returns TRUE. Otherwise, the function returns FALSE. The complexity of this function is $\Theta(n)$, where n is the number of hops+1 (Manhattan distance between producer-consumer tasks + the NI input port). This process ensures that all attempts to establish a CS connection will succeed.

The increase of flow priority and, mainly, the CS establishment consumes more systems resources, and if are not carefully managed, it can lead to an overuse of resources. If all flows are switched to HIGH priority the mechanism becomes inefficient, since the behavior will be similar to a BE NoC (all packets with the same priority). The overuse also occurs with CS. When the path is established, the high priority channel (channel 0) of each router in the path is allocated, and no other CS flow may share the path used by this connection (note that the HIGH priority flows may use channel 1 if the channel 0 is in use).

To avoid resources overuse, the QoS Manager implements a *Time_Out_Monitor*. This function is described in Figure 34. The present work adopts watchdog timers for each communicating task pair, keeping the communication mode active for a given amount of time. The *Time_Out_Monitor* function is invoked periodically (1 ms is the default value of the WINDOW_ LENGTH constant – to put in perspective this value, it is in the same order of resolution than a real-time Linux). A given flow may remain in HIGH priority for *FCt* ms (Flow Counter timer, 15 ms being the default value). The flow returns to LOW priority when the timer expires (lines 7-9). The flow may also stays in CS mode for *CSt* ms (Circuit Switching timer, being 2**FCt* the default value). When *CSt* expires, the flow returns to HIGH priority and PS (lines 10-12).

```
1.  Time_Out_Monitor()
2.  FOR_EACH ctp DO /*ctp is communicant task pair*/
3.          ctp_time_without_event <= current_time – ctp_last_received_event_time
4.          IF ctp priority is LOW THEN
5.              NEXT FOR ITERATION
6.          END_IF
7.          IF ctp priority is HIGH && ctp_time_without_event > PSt THEN
8.              Set ctp priority to LOW
9.              ctp_time_without_event <= 0
10.         ELSE IF ctp flow mode is CS && ctp_time_without_event > CSt THEN
11.             Release CS Connection between ctp
12.             Set ctp priority to HIGH
13.             ctp_time_without_event <= 0
14.         ELSE
15.             ctp_time_without_event += TIME_OUT_MONITOR_WINDOW
16.         END_IF
17.   END_FOR
18.End Time_Out_Monitor
```

Figure 34 – *Time_Out_Monitor* pseudocode

Note that the *ChangeQoSFlow* function (Figure 33, line 2) resets the timer *ctp_time_without_event*, enabling the flow to stay in HIGH priority or CS for longer periods.

Two reasons justify the adoption of a watchdog timer to release CS: (*i*) CS reserves the entire link bandwidth, reducing the NoC resources for other flows; (*ii*) as CS does not suffer disturbing due to the exclusive link allocation, it is not possible to determine when the disturbing traffic finishes. Thus, when *CSt* expires the flow returns to HIGH, and if new events are received the flow may returns to CS. Note that CS and PS may coexist in the same link, due to the duplicated physical channels, and CS may only use one physical channel (channel 0).

For hard real-time applications, where violations are not acceptable, the *CSt* may receive a very large value, inducing the application to keep CS during the application lifetime.

7.1.1 Results

The experiments presented in this section evaluate the priority and CS adaptation, the computing adaptation are disabled in these evaluations. Results were obtained using synthetic and real applications, together with BE applications. In our experiments, BE applications are *disturbing* ones generating traffic that interferes with the evaluated applications, inducing deadline misses. All applications are described in C language.

Three performance scenarios were evaluated: *(i)* without disturbing; *(ii)* with disturbing (no adaptive techniques); *(iii)* with disturbing, but applying monitoring and the adaptive techniques. The scenario without disturbing was used to set the latency and throughput deadlines (profiling).

The MPSoC was modeled in VHDL (NoC, NI, DMA) and SystemC (processors and memory), using an RTL cycle accurate description, allowing accurate measurement of latency and throughput values. The MPSoC was simulated with ModelSim (Mentor Graphics).

7.1.1.1 Evaluation with synthetic applications

Two producer-consumer applications are adopted as synthetic applications with constraints: SR1 and SR2, with task S1/S2 generating data to tasks R1/R2, respectively. Four disturbing BE applications run concurrently to such tasks: two applications access a memory block (T1→MEM, T2→MEM); two applications send data to an IP that emulates an external access unit (T3→OUT, T4→OUT).

Figure 35 shows the mapping of these applications in a 5x4 MPSoC instance. PEs with no explicit task mapping are considered allocated to other applications and are not available. This mapping corresponds to a fragmented system, which is typical in MPSoCs with dynamic workload. The communicating task pair S1→R1 is disturbed by the flows generated by T1 and T2 to MEM, and the communicating task pair S2→R2 is disturbed by the flows generated by T3 and T4 to OUT, and T2 to MEM. In addition, the synthetic RT applications compete for resources in the link between PEs D→A. While the system stays fragmented, and constrained applications are disturbed by BE flows, the solution is to employ QoS techniques to ensure the communications requirements.

Figure 36 summarizes the communication behavior of applications SR1 and SR2 under scenario *(iii)* (with disturbing, but applying monitoring and the adaptive techniques). Initially, flows S1→R1 and S2→R2 start at low priority. The disturbing traffic starts at 5 ms (after the warm up of the target QoS applications). The monitor begins to generate events to QoS Manager. Therefore, the adaptive mechanism changes the flow priority to high (at 6.2 ms for SR1 and 8.3 ms for SR2). The deadlines of flow S1→R1 continue to be violated, inducing a changing in the switching mode to CS (at 7.8 ms). The same behavior happens to flow S2→R2 (at 10.8 ms). Such flow cannot establish CS using the original path, due to a competition in the south output port of router D (connection is allowed only in the high priority channel, which is in use by the CS connection S1→R1). As CS is coupled to adaptive routing, a new path is used, through routers 10-F-E-D-C-B-A (Hamiltonian path), enabling the two applications to use simultaneously CS. The square dotted rectangle in Figure 36 highlights the simultaneous CS transmissions. When the CSt expires, both flows return to high priority (at 40 ms for SR1 and 50.5 ms for SR2). At this time, the disturbing traffic is lower, and high priority is enough to ensure the communication requirements of SR1 and SR2. When the disturbing applications finish, both flows return to low priority (at 56.4 ms for SR1 and 80 ms for SR2). This runtime adaptability (*Time_Out_Monitor*) avoids the network overuse, using high priority and CS only when it is really necessary. The NoC resources are allocated on demand at runtime.

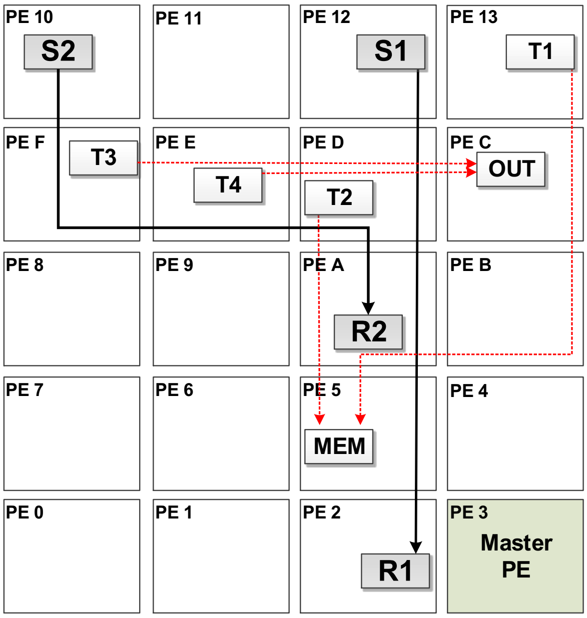


Figure 35 - Non-optimal mapping of the synthetic applications, with the disturbing applications. Empty PEs are supposed to be allocated to other applications.

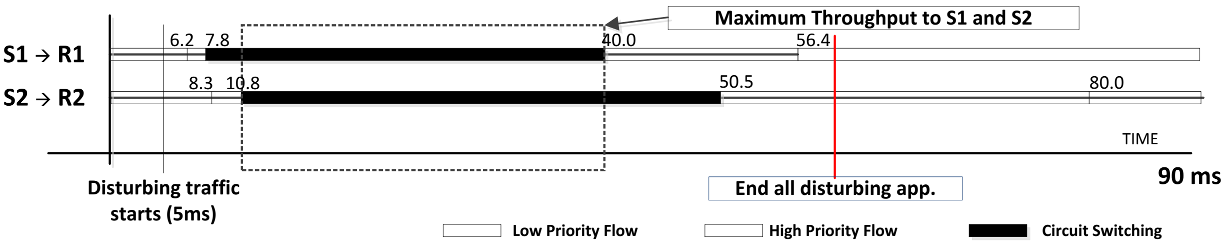
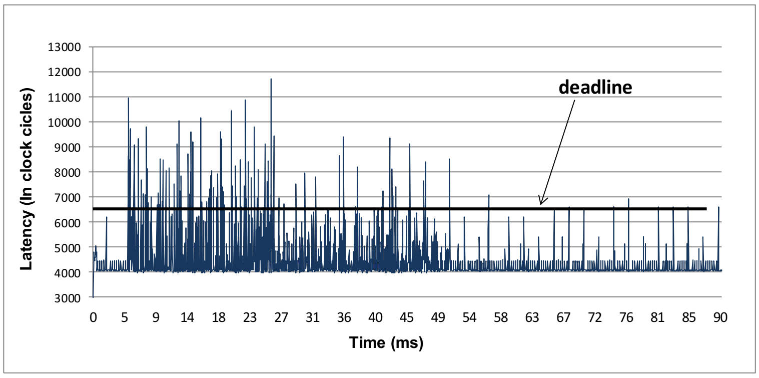


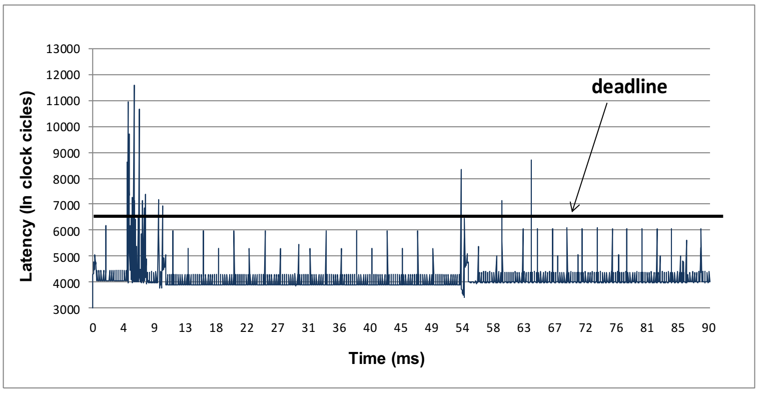
Figure 36 - Flow adaptation of the synthetic applications during execution. Each horizontal bar represents the flow state. The numbers above each bar indicate a change in the flow state (in ms)

The profile configuration set the latency deadline at 6,750 clock cycles. Figure 37 presents latency values with disturbing traffic for the SR2 application, without and with QoS adaptation (scenarios *ii* and *iii*). The X-axis corresponds to execution time, and the Y-axis the latency for each transmitted packet. The horizontal thicker line corresponds to the latency deadline. The first half of the execution time in Figure 37(a) has several latency violations due to disturbing traffic. As the disturbing applications finish its execution, the latency violations reduce. Figure 37(b) presents the results obtained when using adaptive techniques. When adaptation starts (8.3 ms) the latency reaches the constraint. In fact, only 5 violations arise after the adaptation starts, 2 during the CS establishment and 3 during the releasing. These violations are due the adopted protocol, which requires the consumption of all packets in the communication pipe. This condition stalls the application communication during the CS establishment/release, explaining the observed violations. After 50 ms the disturbing traffic finishes, and the flow returns to low priority. The three observed violations are related to the overhead of the connection release protocol.

Another performance parameter to evaluate is the jitter, i.e., the time interval between packets arrival. In Figure 37(a) 28% of packets have a jitter superior to 10% of the latency deadline. Applying the adaptive techniques, only 7.3% of the packets have a jitter superior to 10% of the latency deadline.



(a) Latency without adaptation (scenario ii)



(b) Latency with QoS adaptation (scenario iii)

Figure 37 – Latencies of the SR2 task with disturbing traffic. SR1 presents similar latency results.

Figure 38 plots the throughput for the SR2 application. With this adaptation, the throughput is implicitly improved. It is possible to observe at 5 ms the effect of the disturbing traffic reducing the throughput, and the reestablishment of the throughput at 8.3 ms, when the flow starts with high priority and then changes the switching mode to CS. When all disturbing applications finish its execution, the application throughput returns to a high throughput in both scenarios (with and without QoS adaptation). The throughput with QoS adaptation corresponds to the behavior presented in Figure 36.

This first set of experiments demonstrated the effectiveness of the proposed approach, enabling applications with communication constraints to run concurrently with disturbing applications. Jitter was also reduced, which is an important feature for applications with QoS constraints.

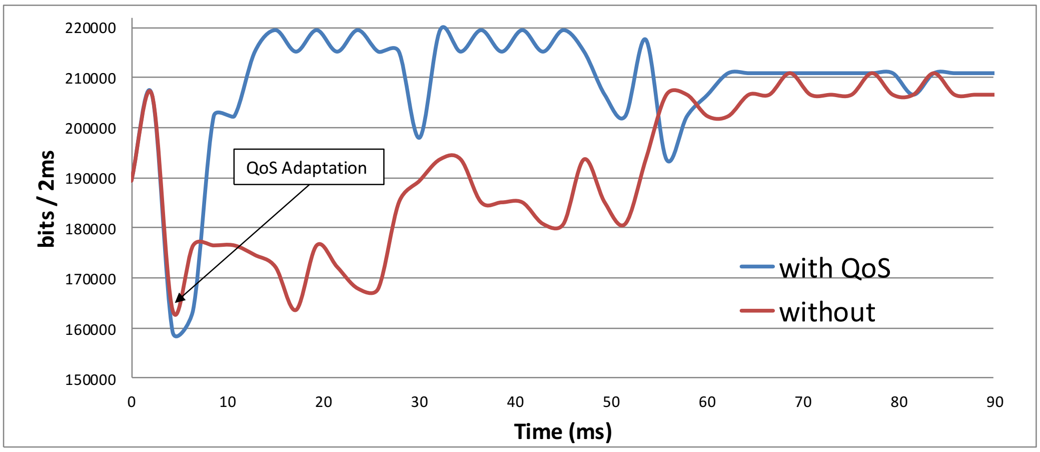


Figure 38 - Throughput for the SR2 application, without and with QoS adaptation.

7.1.1.2 *Evaluation with real applications*

Two real applications were used: MJPEG decoder and Dynamic Time Warping (DTW). The task graphs of such applications are presented in Figure 39. The MJPEG application has five tasks, two responsible to input and output processing (START and PRINT) respectively, and the remainder are responsible for image decoding tasks (IVLC, IQUANT and IDCT). The MJPEG application decodes 320 frames, where each frame corresponds to an array of 128 positions related to the compressed frame to be decoded. In the DTW application, the main flow occurs between the task BANK (bank of patterns), and tasks P1, P2, P3 and P4, which recognize the sample test with the patterns through the DTW algorithm. The DTW application recognize a test sample in a bank with 2000 patterns, where each task is responsible for processing 500 patterns

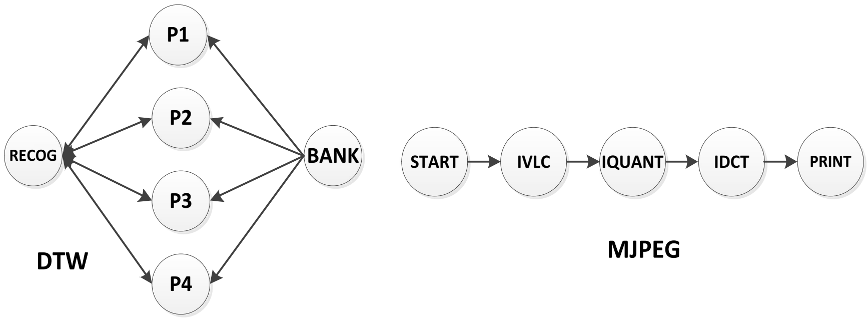


Figure 39 – Task graphs for applications DTW and MJPEG.

Figure 40 summarizes the communication behavior of the MJPEG and DTW applications, adopting the mapping presented in Figure 41, in a 5x5 MPSoC instance. As in the previous evaluation, PEs with no explicit task mapping are considered allocated to other applications and are not available. The same four disturbing applications run concurrently with the MJPEG/DTW applications. Some flows are not presented for sake of simplicity in the Figure. The disturbing applications start their execution at 10 ms, and stay running throughout the simulation. The dotted rectangles in Figure 40 highlights when the two applications use the maximum possible number of CS connections. The vertical lines signalize the end of the execution of the MJPEG and DTW applications. Note that the MJPEG application, due to disturbing traffic, stays most of the time communicating by CS, ensuring lower jitter and latency.

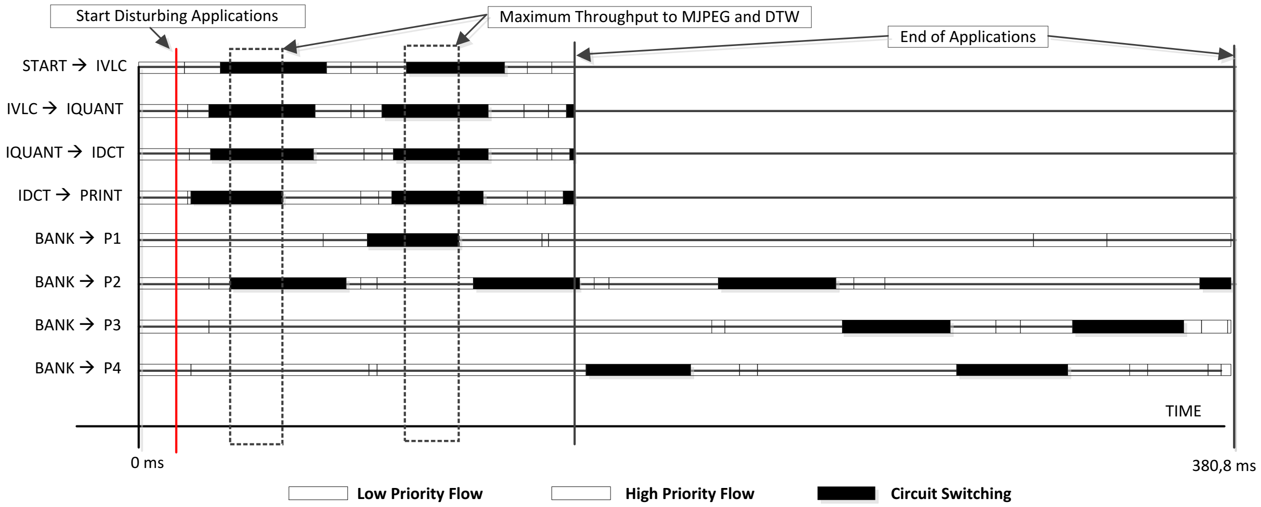


Figure 40 - Flow adaptation of MJPEG and DTW applications during execution.

Due to the competition in south port of router 15 (Figure 41), task BANK cannot establish simultaneous connections with tasks P2, P3 and P4. Throughout the execution, it is possible to observe a rotation of CS in tasks P2/P3/P4 due to this competition for the high priority channel. Observe in Figure 40 the first rectangle with five simultaneous CS connections. The BANK→P2 pair is communicating via CS. As IQUANT→IDCT pair blocks the minimum path between those tasks, the CS flow follows an adaptive route: 15-12-11-C-7-2-1 (router numbers highlighted in red).

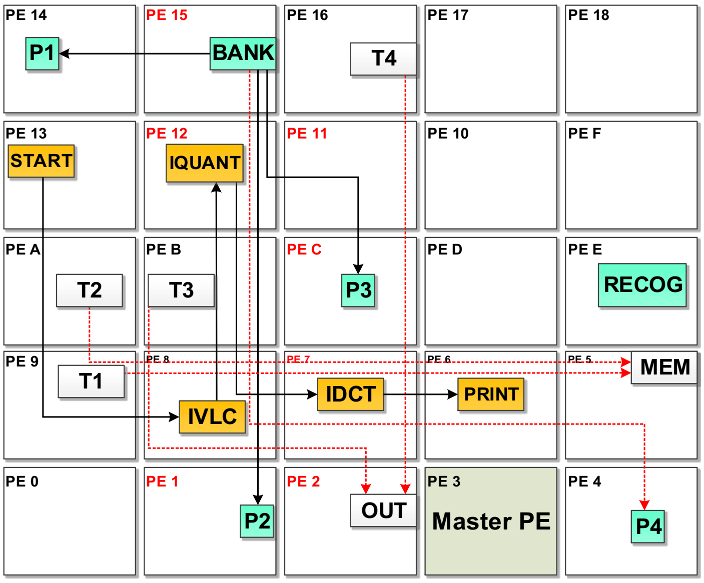
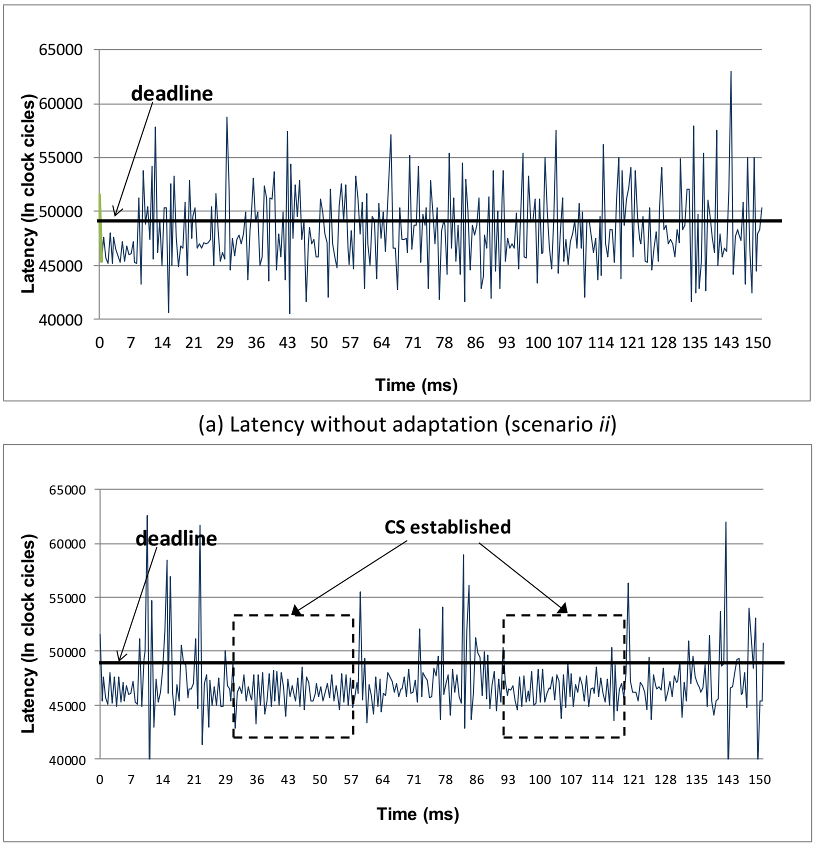


Figure 41 - Non-optimal mapping in a 5x5 MPSoC instance, with MJPEG and DTW applications running with disturbing applications. Empty PEs are supposed to be allocated to other applications.

The monitored task of the MJPEG application is PRINT. This task presents the frames decoded by the application. It runs on PE 6, where three other flows compete for west input ports along with the IDCT flow (being one flow belonging to the DTW application). The profile phase of the PRINT task set the latency deadline at 48,500 clock cycles. Figure 42 presents the latency values for the PRINT task. The dotted square rectangles in Figure 42(b) correspond to the periods where the communication IDCT→PRINT is through CS. These two CS periods can also be seen in Figure 33 (IDCT→PRINT).



(a) Latency without adaptation (scenario *ii*)

(b) Latency with QoS adaptation (scenario iii)

Figure 42 - Latencies of the PRINT task with disturbing traffic.

It is important to observe in the graphs of Figure 42:

- **Violations**. After 10 ms (moment when the disturbing traffic starts), the number of violations without QoS adaptation was 131 (Figure 42(a)). Using the QoS adaptation, such number decreases to 50 (Figure 42(b)), representing 61% of reduction.
- **Jitter**. Applying QoS adaption only 13 of the 50 violations have a latency superior to 10% of the latency deadline. This fact can be easily observed comparing both graphs.
- **CS**. Observe that the latency in the highlighted dotted square rectangles is inferior to the latency deadline. This demonstrates the effectiveness of the CS to real-time. If an application has hard-real time constraints, it is possible to increase CSt to a higher value, releasing the CS connection only at the end of the application. The effect of such action can be observed in Figure 43, where no violations are detected if CS is not released.

To avoid any violations in hard real-time applications, it is possible to set the applications to use CS from the beginning in all task's communications. In this case, monitoring and adaptation techniques are disabled. Due to the limitations on CS establishment (only using high priority channels), few hard real-time applications can run simultaneously.

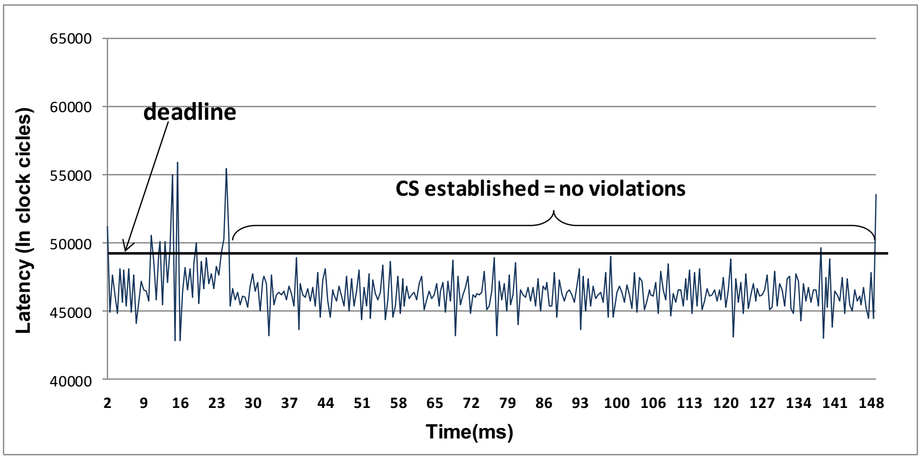


Figure 43 - Latencies of the PRINT task with disturbing traffic, with CSt set to a value equivalent to the application execution time.

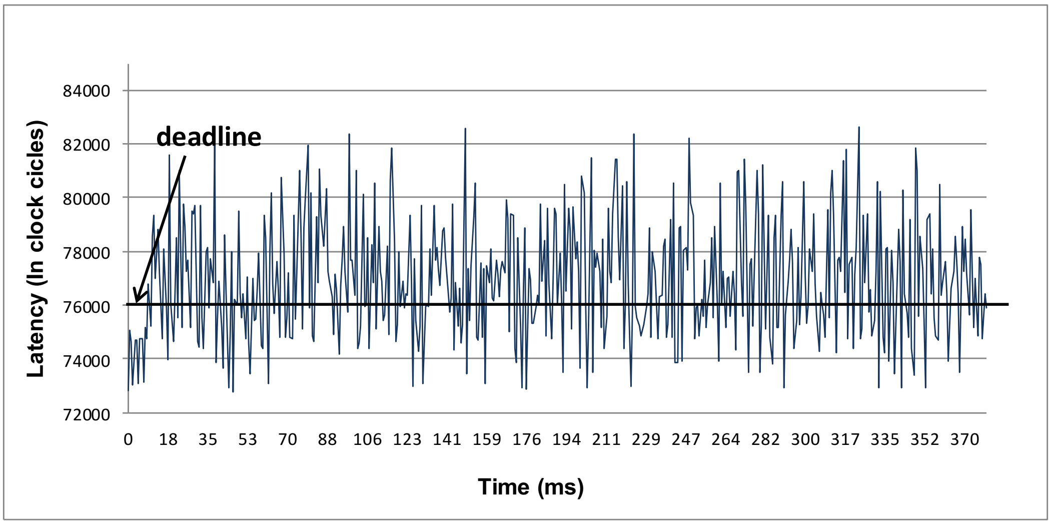
The monitored task of the DTW application is P4. All application tasks were previously monitored, and P4 was chosen due to the highest disturbing traffic, between tasks BANK and P4. The profile phase of the P4 task set the latency deadline at 76,000 clock cycles (value near to the one obtained during the profile step, to stress the adaptation techniques). Figure 44 presents the latency values for the P4 task. Without QoS adaptation, the number of latency violations presented in Figure 44(a) is 300. Using the QoS adaptation such number decreases to 126 (Figure 44(b)) representing 58% of reduction.

It is also possible to identify the periods where CS is used (highlighted by a dotted square rectangle), where only 8 latency violations were identified. Such CS periods can also be seen in Figure 33 (BANK→P4). It is important to observe in Figure 44(b) the execution interval 10ms-154ms, where most of the latency violations arise (79). The reason explaining this fact is that the flow BANK→P4 cannot use CS because it concurs with flows IQUANT→IDTC and IDTC→PRINT, which are already using CS. Such result demonstrates the limitation of the NoC to offer QoS by CS for concurrent applications with performance constraints. Other adaptive methods could be employed, as task migration, moving the task P4 to a position without disturbing traffic.

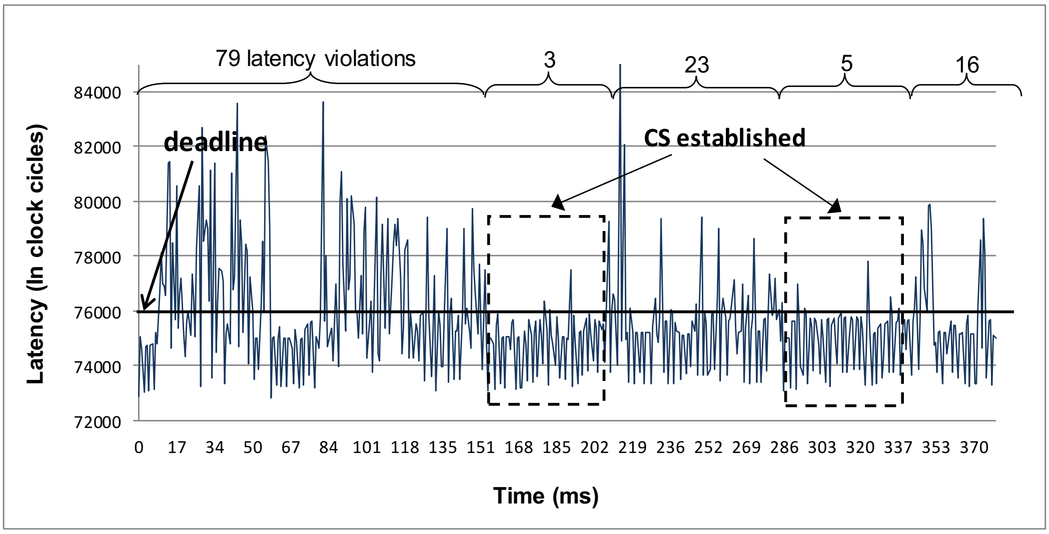
Throughput results were similar in all three scenarios. It was not observed a significant difference, since the tasks are computation intensive, i.e., the communication volume is small. Therefore, for such applications latency and jitter are the critical performance parameters to be monitored.

The adoption of the adaptive techniques did not affect negatively the total execution time. Without QoS adaptation, the disturbing applications increased the total execution time of the

MJPEG and DTW applications in 4.0% and 2.7%, respectively. Applying QoS adaptation, the execution time overhead was 1.2% and 1.0% for MJPEG and DTW applications, respectively. Therefore, the use of the adaptive techniques almost restored the baseline execution time (without disturbing traffic).



(b) Latency with QoS adaptation



(b) Latency with QoS adaptation

Figure 44 - Latencies of the P4 task with disturbing traffic.

The volume of the monitoring traffic was, in the worst case (with synthetic applications), 0.4% of the total traffic, corresponding to 0.06% of the link capacity. In applications as MJPEG and DTW, the volume of the monitoring traffic was even smaller (0.05%). Therefore, the monitoring traffic presents low intrusiveness (i.e., does not penalize the applications' performance), with smaller values than the ones reported in the literature: [FAR12] and [FIO10] employ 0.7% and 0.2% of the link capacity, respectively. The monitoring software overhead was in average 4% (this value is in fact overestimated, because monitoring is executed even when tasks are in IDLE state). This explains why the total execution time is not penalized.

7.2 Computing Adaptation

The implemented computing adaptation employs runtime management through processor scheduling priority and task migration.

When a throughput event is received, the QoS Manager calls the *ChangeQoSComputation* function. This function is detailed in Figure 45. Its objective is to choose the appropriate adaptive technique to provide a computation QoS improvement. As the QoS Manager analyzes a communicating task pair, the *ChangeQoSComputation* is employed for the producer and for the consumer tasks.

```
1.  ChangeQoSComputation ( input: RT_task)
2.    IF cluster has free resources THEN
3.        BE_task <= Search_BE_task(RT_task processor)
4.        IF  BE  != -1 THEN
5.            Migrate BE_task to the best processor without RT tasks in the cluster
6.            IF is not possible to migrate BE_task THEN
7.                Increase CPU scheduling priority for RT_task
8.            END_IF
9.         ELSE IF ∃ other RT task in the RT_task processor THEN
10.           Try migrate RT_task to best free processor into cluster
11.       END_IF
12.   ELSE IF  ∄ other RT tasks in the RT_task processor THEN
13.       Increase CPU scheduling priority for RT_task
14.   END_IF
15. End ComputationAdapt
```

Figure 45 – *ChangeQoSComputation* pseudocode.

The *ChangeQoSComputation* procedure receives as input an *RT_task* (producer or consumer), with QoS infringement in the communication task pair belonging to it. The first action (line 2) verifies if the cluster has free resources to enable the task migration. Next (line 3), the function *Search_BE_task* searches for BE tasks in the *RT_task* processor that can be migrated to an appropriated PE. It returns the task ID of the BE task or -1 if no BE task is found. The goal of all this process is:

- If exists a RT task, sharing CPU resources with BE tasks, the first action is try to migrate the BE tasks to another PE. The chosen PE is the one with the lowest load and without RT tasks running.

Lines 3-5 execute this action, and if it is not possible to migrate the BE task, the scheduling priority of RT task is increased (line 7).

However, if does not exist BE tasks running in *RT_task* processor but exists other RT task running in the *RT_task* processor, the procedure tries to migrate the RT task to a free PE (line 9 and 10). This action is made after trying to migrate the BE tasks because the migration process, although optimized, may penalize the RT task performance due to the stalling during the migration process.

If it is not possible to execute the migration (BE or RT tasks), the RT task has its scheduling priority increased (line 12).

The algorithm presented in Figure 45 tries to reserve one PE to each RT task, while grouping BE tasks in the same PE. This procedure contributes to defragment the system, and BE tasks can also present an execution time improvement since they will be moved to PEs without RT tasks.

7.2.1 Results

This section evaluates the gains obtained with scheduling priority and task migration. Results were obtained using RT applications mapped together with BE applications. All applications are described in C language and the simulation uses an RTL cycle accurate description

Three scenarios were evaluated: *(i) best scenario*: where each application is executed alone in the system, and each task is mapped in a free PE; *(ii) without adaptation*: without monitoring and adaptive techniques, with disturbing applications; *(iii) adaptation*: applying monitoring and adaptive techniques with disturbing applications. The best scenario is used as reference to set the latency and throughput deadlines (profiling step).

The MPSoC used in this test case contains 36 PEs (6x6), divided in four 3x3 clusters. Four RT applications are used: *MJPEG*, *DTW*, *audio_video* and *FFT*. The task graph of *MJPEG* and *DTW* applications were already presented in Figure 39. The task graphs of the *audio_video* and *FFT* applications are presented in Figure 46. The *audio_video* application is an audio/video decoder with seven tasks. The video pipeline is executed by a MJPEG decoder partitioned in three tasks (step1, step2 and step3), and the audio pipeline is composed by an ADPCM decoder (AD) and a FIR filter. The FFT application is a Fast Fourier Transform with a 128 input size, with two tasks, the FRONT task transmits the samples to the FFT task that performs the FFT algorithm (radix-2).

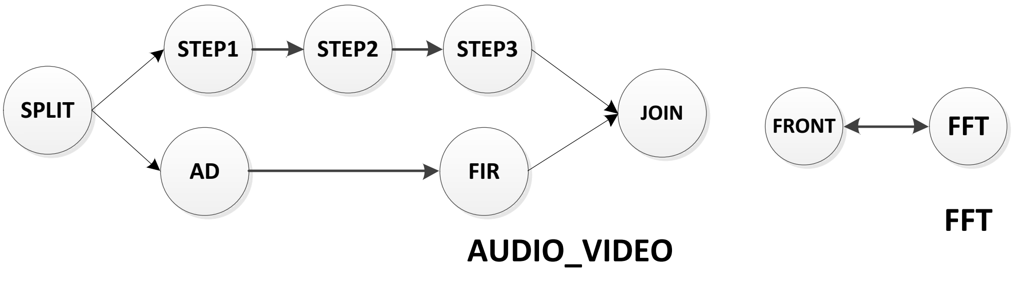


Figure 46 – Task graph of audio_video and FFT applications

Each cluster received one RT application, mapped together with synthetic applications that correspond to BE applications. The RT applications were mapped first on system, with the BE applications being initialized and finalized during the execution of the RT applications.

Figure 47 presents the execution time of the RT applications. It is possible to note that the adaptive techniques do not provide a significant improvement to *audio_video* and *FFT* applications, but for MJPEG and DTW applications the adaptive technique contributed to reduce the execution time compared to the scenario without adaptation. The MJPEG presented a final runtime just 0.8% higher than the best scenario runtime. The DTW presented a final runtime 16% higher than the best scenario, but 41% lowest that without adaptation scenario.

The audio_video and FFT applications were not benefited with the adaptive techniques due to the following reasons:

- The audio_video application was mapped in a cluster corner that contributed to minimize BE interference;
- Reduced communication rate between tasks (most of the time the tasks are executing, not communicating);
- The simulation time (<500ms) did not allow the adaptive techniques to provide a significant level of adaptability. It is quite infeasible to simulate large systems for longer periods, due to the modeling level: cycle-accurate RTL [CAS13a]. To obtain results with longer simulation times new abstract hardware descriptions must explored, as the Open Virtual Platform [REK13].

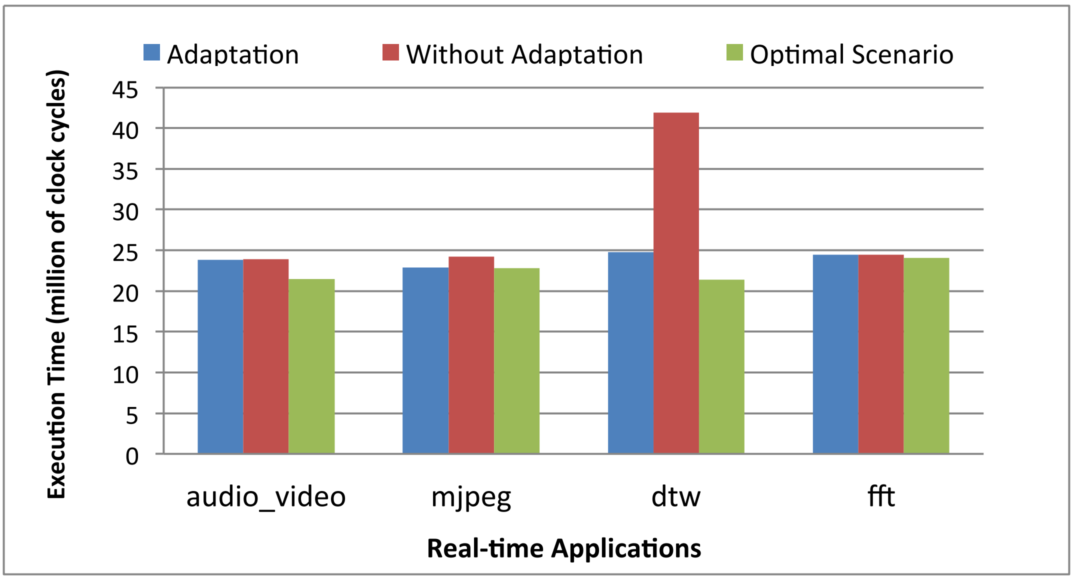


Figure 47 – Execution time of RT applications.

Figure 48 details the throughput obtained for the communicating task pair *iquant* → *idct*. These tasks belong to MJPEG application. The *iquant* task is the producer and *idct* is the consumer. As can be observed, the flow running in the best scenario presents a throughput of 24,435 bits/5ms. The configured throughput deadline is 24,200 (1% lower than best scenario) and the resolution time was configured to 5 ms. With the disturbing induced by the BE applications, the throughput decreased to 22,862 bits/5ms. Adding the adaptive techniques, the average throughput increased to 24,187, remaining above the deadline.

Figure 49 details the cluster task mapping, and the task migrations. The Figure shows only one cluster, the one that received the MJPEG application.

In step 1 the MJPEG application is mapped into the cluster, with PE 15 sharing *iquant* and *idct* tasks, and PE 8 sharing *ivlc* and *start* tasks. This CPU sharing reduces the application performance. Hence, the heuristic migrates some MJPEG tasks to free PEs (the migration is fired by monitoring events). Task *ivlc* is migrated to PE 17 and task *iquant* is migrated to PE 5. It is possible to observe in Figure 48(1) that this migration process did not affected the throughput of the migrated tasks, demonstrating the low cost of optimized task migration proposed in this work. After the end of migration process, each RT task of the MJPEG application runs individually at each PE, contributing to sustain the throughput according to the application requirements.

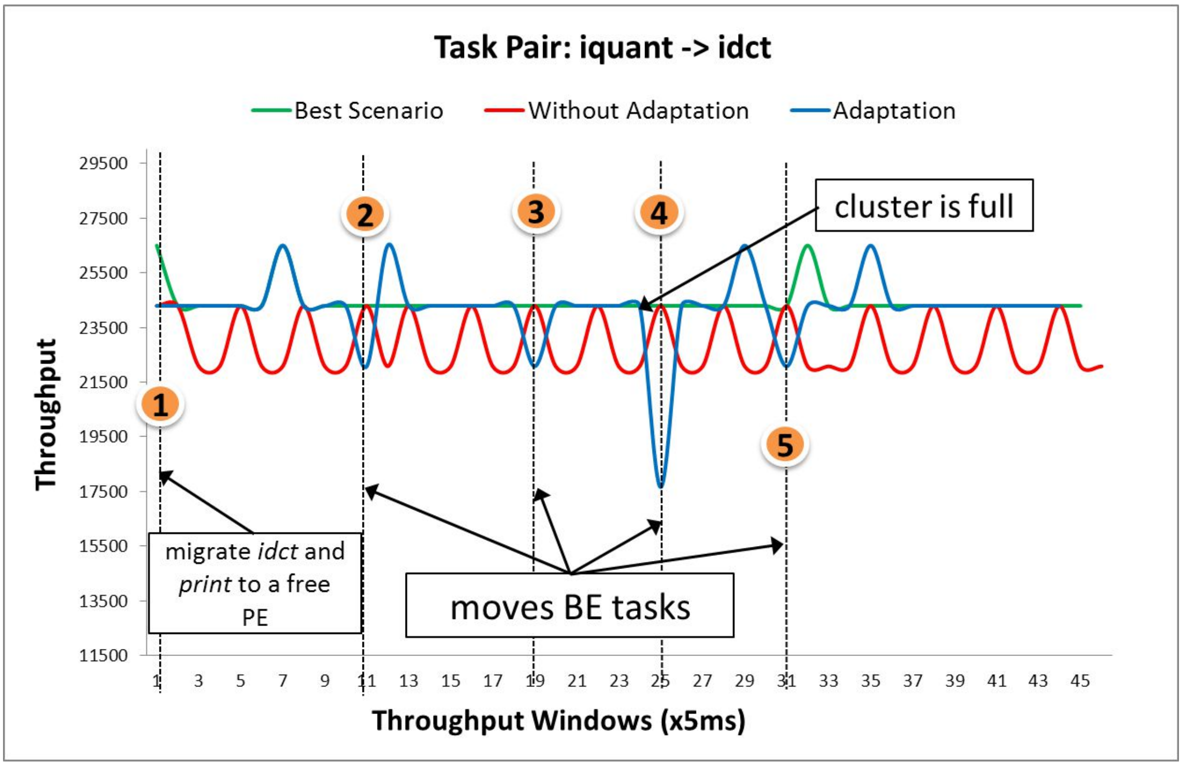


Figure 48 – Throughput results for the communicating task pair *iquant* → idct.

In step 2 the BE applications initially mapped in the cluster (in orange) finish their execution, and new BE applications (in green) are mapped into the cluster. The mapping heuristic does not considers RT tasks, and maps some BE tasks in PEs where MJPEG tasks are already running, affecting the *idct* and *start* performance, decreasing the application throughput, triggering the adaptation. In this case, the adaptation moves the BE tasks sharing CPU with RT tasks, to PE 6. At the end of this migration process, all tasks belonging to the MJPEG are running in a dedicated PE.

In step 3, the same process described in step 2 is repeated.

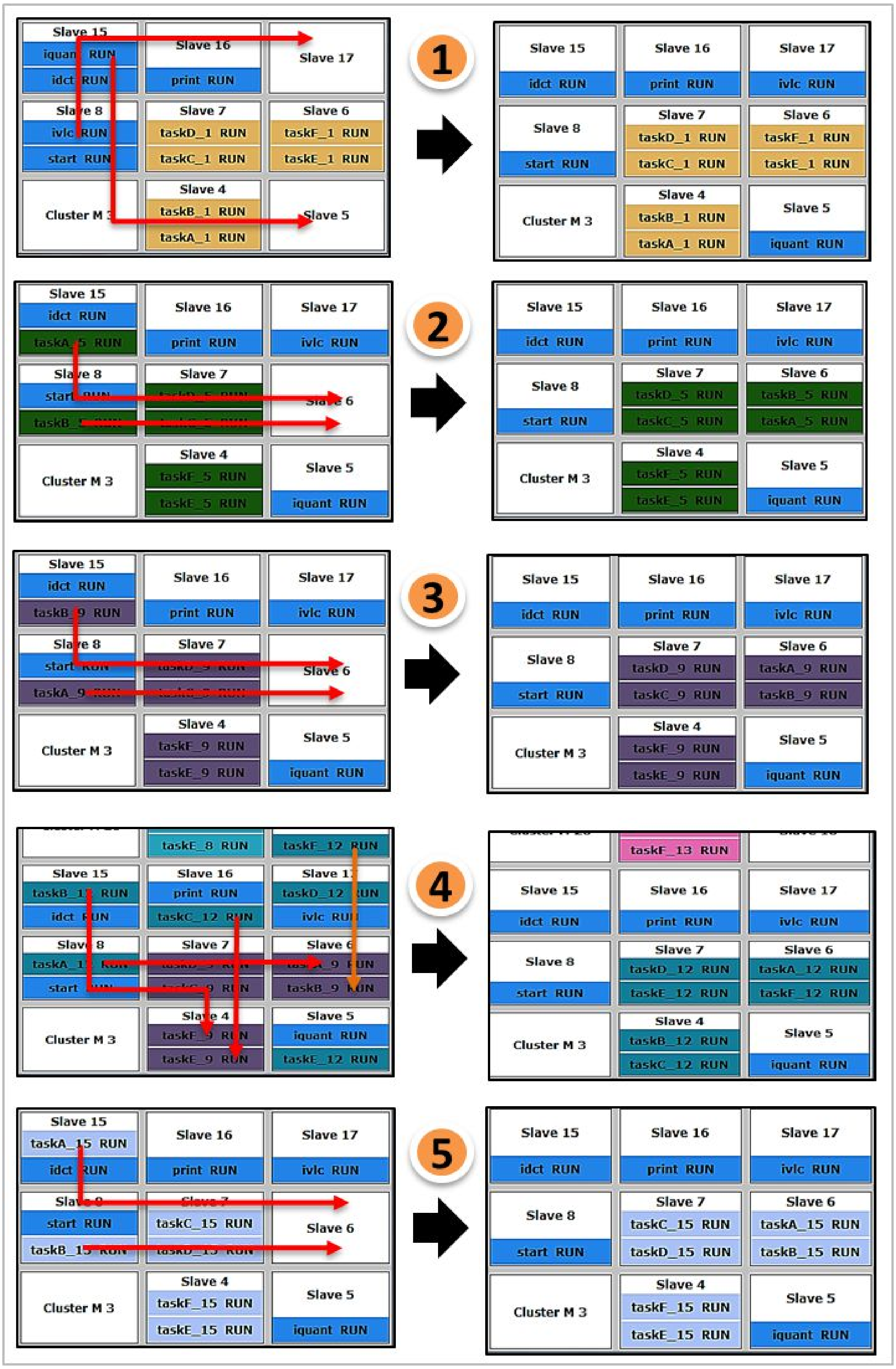


Figure 49 – Adaptation cluster steps for the MJPEG application.

In step 4 the BE application that was mapped in step 3 remains running, and another BE application is mapped in the cluster. The cluster becomes full and a BE task (taskF_12) is allocated by reclustering in other cluster. At this moment all MJPEG tasks are sharing CPU with BE tasks, and their respective throughput present an important reduction (Figure 48(4)). At this moment the previous BE application – that is still running since step 3 – finishes its execution. Promptly, the adaptation heuristic moves the remaining BE tasks to PEs where no RT tasks are running. Tasks in PE 15 and 16 go to PE 4, and tasks in PE 8 and 18 (this PE belongs to another cluster), go to PE 6, removing the CPU sharing with the RT tasks.

In step 5, all BE applications that were running in step 4 finish their executions, and a new BE application is mapped, as described in step 2. Some BE tasks are mapped in the same PEs of RT tasks, triggering a throughput event and the adaptation occurs migrating the BE task to another PEs.

This example, Figure 49, also illustrates the runtime reclustering management. In step 4 *taskF_12* was mapped in a neighbor cluster, using the reclustering heuristic proposed in [CAS13b]. When a given task finished in the cluster, the cluster master (in PE 3) is notified, and it verifies if there are tasks running in borrowed PEs from other clusters. The cluster master then migrates *taskF_12* to PE 6.

As can be observed in Figure 47, the DTW application has a significant improvement employing the adaptive techniques. Figure 50 presents the throughput of communicating task pair *bank→p2*. The best scenario generates a throughput of 28,512 bits/5ms. As this application is frequently used for pattern recognition (image, audio, speech recognition), the DTW tasks deadline must be the best the system can provide. For this reason the deadline was set to 28,500 bits/5ms.

The average throughput decreased to 14,476 bits/5ms when the DTW application runs with the BE applications, corresponding to a decrease in the throughput superior to 49%. With the adaptive techniques, the average throughput was 24,603 bits/5ms, presenting a throughput improvement of 70% in comparison with the scenario without adaptation.

Three moments are highlighted in Figure 50:

(1) the throughput is the same to the throughput without adaptation. The monitoring detects violation and generates events, the adaptation occurs and the heuristic migrates task *p2* to a free PE, restoring the throughput.

(2) a BE task is dynamically mapped with task *p2*, reducing the throughput. The adaptive technique migrates the BE task to another PE, restoring the throughput to the best scenario.

(3) other DTW tasks (*p3* and *recognizer*) are sharing resources with BE tasks, and the BE task migration is not possible due to the lack of free resources, thus, the throughput is

reduced to 24,192 (15 % lower than best scenario). During this period, two peaks are observed, due to an increase in the scheduling priority of tasks *bank* and *p2*.

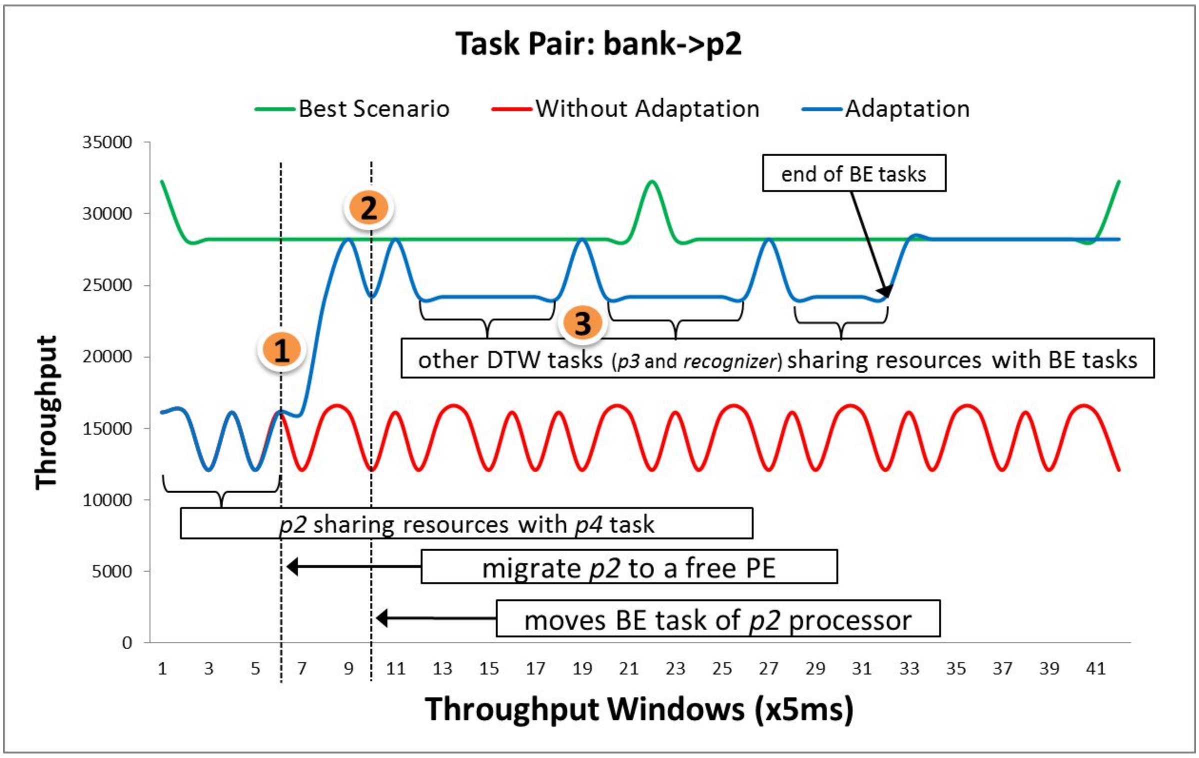


Figure 50 - Throughput results for the communicating task pair bank->p2.

The computing adaptation results presented until this moment mainly addressed task migration. To specifically observe the effect of the scheduling priority, Figure 51 presents the throughput of the communicating task pair *p1→recognizer* (DTW application). The flow with adaptation presents an initial decrease due the resource sharing of *p2* and *p4* with BE tasks. Next, in moment (1), the adaptive heurist tries to migrate task *recognizer* to free PE (task *p1* already runs alone), but there is no free PE. Therefore, the scheduler priority of the *recognizer* task is increased. This action provides a throughput improvement. This improvement not achieves the best scenario throughput, but provides a throughput increase of 39.7% over the scenario without adaptation. With the end of BE task, the *recognizer* task returns to run alone in the PE, and the flow achieves the maximum throughput.

This evaluation demonstrates the effectiveness of the computing adaptation heuristic. As can be seen, this heuristic is able of distribute the resources, by isolating (when possible) RT tasks in free processors and grouping BE tasks. This protocol contributes to the QoS performance improvement. The heuristics firstly migrates BE tasks, avoiding that the migration process (even optimized) disturb the RT task execution. If the processor with the RT task do not have BE tasks executing concurrently, but have others RT tasks, the heuristic try migrate the RT task with QoS violations to a free PE. This choice is made before increasing the processor scheduler priority due to the fact that migrate a RT task to a free PE avoids that the RT task shares resources with other task, besides distribute the system resource utilization (load balancing). Therefore, the increase of the processor scheduler priority is only employed when there are no free cluster resources (migration is impossible).

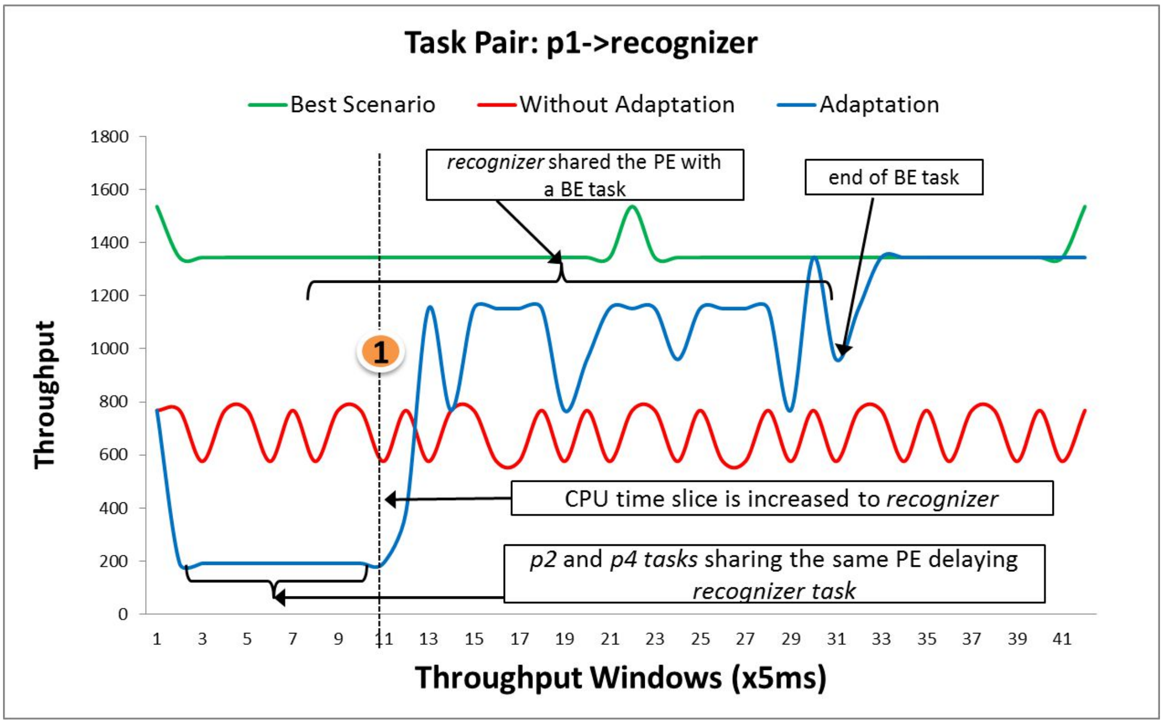


Figure 51 – Throughput results for the communicating task pair p1->recognizer.

8 CONCLUSIONS AND FUTURE WORKS

This work presented a runtime adaptive QoS management scheme, with four different adaptive QoS techniques. Each QoS technique has particular features, providing QoS improvement to RT applications, in both computation and communication. The results obtained demonstrated the effectiveness of the QoS management. The development of a cluster-based distributed management provides scalability, since each cluster has a QoS Manager responsible to select the most appropriate QoS technique for the monitored applications. The adaptation can be considered as fine grain, since monitoring/adaptation is executed at the task level.

To provide support to runtime adaptive QoS management, this work proposed and evaluated two monitoring techniques. The evaluation showed that hybrid implementation is the choice that provides the lowest degree of intrusiveness, with in the worst case a link usage equal to 0.8%, not penalizing the execution time of monitored applications. Both proposals are scalable, because they employ a two-level hierarchical monitoring scheme, distributing the monitoring data among each cluster.

The flow adaptation, differently from proposals that use probes and backtracking schemes to evaluate path availability for CS, adopted a procedure to evaluate the CS feasibility in software in the QoS Manager. This process ensures the management of several simultaneous CS connections. As observed in the results, a small number of violations remain after employing QoS adaptation. However, as described, increasing CS time enables the execution of hard real time applications, suppressing violations, ensuring no interferences from flows belonging to other applications.

The computing adaptations achieved a significant throughput improvement, increasing the scheduling priority and using task migration. An important feature of the method is to try to first migrate BE tasks interfering with RT tasks, and after that act over the RT application. This feature minimizes the performance penalty induced by the QoS techniques, since the RT tasks continue to run without interruptions.

As observed, the task mapper tries to use all available resource to enable new applications to run in the system. However, BE may be mapped together with RT tasks, penalizing the performance of RT applications. A suggestion for future works is to develop a QoS-aware task mapping heuristic, to minimize the number of task migrations.

This work also proposed a debugging tool. This tool provides a practical way to observe the behavior of the system at the protocol level. With this tool, the designer can follow the platform operation, the parallel communications, detect software and hardware errors, and validate new protocols. The tool also enables to monitor the NoC traffic volume, detailing this traffic according to each communication service. An important feature of this tool is that it can be used as a graphical interface with the SystemC simulation, and through the option to pausing the simulation prevents the user from having to wait until the end of the simulation to verify the behavior of the system.

8.1 Future works

Future works comprise to pursue the exploration of adaptive QoS techniques, as well as to develop methods to increase the MPSoC utilization (utilization wall, or dark silicon), in very large MPSoCs. Another research direction is to add DVFS (dynamic frequency and voltage scale) in the adaptive QoS techniques, resulting in green computing through energy economy, also extending the system lifetime.

Specific future works include:

- Abstract monitoring models;
- Abstract QoS management models;
- Dedicated monitoring and control network;
- QoS-aware task mapping heuristic;
- DFVS inclusion in runtime adaptive QoS management;
- A full QoS-aware operating system, designed to be modular and scalable;
- Including in the debugging tool the intra PE communication information (the current debug tool does not consider the communication volume exchanged between tasks in the same PE);
- An adaptive monitoring that adjusts itself to a monitoring frequency, according to the application profiling.

8.2 Publications

The current Dissertation had some of its results published in:

- MORAES, Fernando; CARARA, Everton; RUARO, Marcelo ; MADALOZZO, Guilherme.
  “**Evaluation of Adaptive Management Techniques in NoC-Based MPSoCs**”.
  In: ICECS, 2012, pp. 548-551.
- RUARO, Marcelo ; CARARA, Everton; MORAES, Fernando.
  “**Adaptive QoS Techniques for NoC-Based MPSoCs**”.
  In: SOC, 2013.
- RUARO, Marcelo ; CARARA, Everton; MORAES, Fernando.
  “**Tool-set for NoC-based MPSoC Debugging – A Protocol View Perspective**”.
  Accepted for publication in: ISCAS, 2014.

A submission to a journal is in minor review process:

- RUARO, Marcelo ; CARARA, Everton; MORAES, Fernando. “**Runtime Adaptive Circuit-Switching and Flow Priority in NoC-Based MPSoCs**”. IEEE Transactions on VLSI – In Minor Revision.

REFERENCES

[BEN02] Benini, L; Micheli, G. “Networks on chips: a new SoC paradigm”. Computer, v.35(1), 2002, pp. 70–78.

[CAR11] Carara, E. "Serviços de Comunicação Diferenciados em Sistemas Multiprocessados em Chip Baseados em Redes Intra-Chip". Tese de Doutorado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2011, 107p.

[CAS13a] Castilhos, G.; Mandelli, M.; Madalozzo, G.; Moraes, F., "Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes". In: ISVLSI, 2013, pp. 153-158.

[CAS13b] Castilhos, G. "Gerência Distribuída de Recursos em MPSoCs – Mapeamento e Migração de Tarefas". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2013, 66p.

[CUI12] Cui, Y.; Zhang, W.; Yu, H. “Decentralized Agent Based Re-Clustering for Task Mapping of Tera-Scale Network-on-Chip System”. In: ISCAS, 2012, pp. 2437-2440.

[FAN11] Fan, W.; Xiang, L.; Hui, S. “The QoS mechanism for NoC router by dynamic virtual channel allocation and dual-net infrastructure”. In: ICCP, 2011, pp. 1-5.

[FAR10] Faruque, M. Al; Jahn, J.; Ebi, T.; Henkel, J. “Runtime Thermal Management Using Software agents for Multi- and Many-Core Arquiteture”. IEEE Design & Test, v.27(6), 2010, pp. 58-68.

[FAR12] Faruque, M. Al; Ebi, T.; Henkel, J. “AdNoC: Runtime Adaptive Network-on-Chip Architecture”. IEEE Transactions on Very Large Scale Integration (VLSI) Systems, v.20(2), 2012, pp. 257 – 269.

[FAT11] Fattah, M.; Daneshtalab, M.; Liljeberg, P.; Plosila, J. “Exploration of MPSoC Monitoring and Management Systems”. In: ReCoSoC, 2011, pp. 1-3.

[FIO10] Fiorin, L.; Palermo, G.; Silvano, C. “A Monitoring System for NoCs”. In: NoCArc, 2010, pp. 25-30.

[GAR13] Garibotti, R., et al. “Simultaneous Multithreading Support in Embedded Distributed Memory MPSoCs”. In: DAC, 2013, 6p.

[GRA12] Gray, I.; Audsley, N. “Challenges in Software Development for Multicore System-on-Chip Development”. In: RSP, 2012, pp. 115-121.

[GUA10] Guang, L.; Bo Yang; Plosila, J.; Latif, K.; Tenhunen, H. “Hierarchical power monitoring on NoC - a case study for hierarchical agent monitoring design approach”. In: NORCHIP, 2010, pp. 1–6.

[HED11] Hedde, D.; Petrot, F. “A non intrusive simulation-based trace system to analyse Multiprocessor Systems-on-Chip software”. In: RSP, 2011, pp. 106-112.

[HOW10] Howard, J; Dighe, S.; Hoskote, Y. “A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS”. In: ISSCC, 2010, pp. 108-109.

[INT13] INTEL. “Intel® 22nm Technology”. Capturado em: http://www.intel.com/content/www/us/en/silicon-innovations/intel-22nm-technology.html, Novembro 2013.

[JOV10] Joven, J.; Marongiu, A.; Angiolini, F.; Benini, L.; Micheli, G. de. “Exploring Programming Model-driven QoS Support for NoC-based Platform”. In: CODES+ISSS, 2010, pp. 65-74.

[KAK11] Kakoee, M.; Bertacco, V.; Benini, L. “ReliNoC: A Reliable Network for Priority-based On-chip Communication”. In: DATE, 2011, pp. 1-6.

[KOR12] Kornaros, G.; Pnevmatikatos, D. “Real-Time Monitoring of Multicore SoCs Through Specialized Hardware Agents on NoC Network Interfaces”. In: IPDPSW, 2012, pp. 248-255.

[KOR13] Kornaros, G.; Pnevmatikatos, D. “A survey and taxonomy of on-chip monitoring of multicore systems-on-chip”. ACM Transactions on Design Automation Electronic Systems, v.18(2), 2013, pp. 38.

[LI03] Li, J; Yao, C. “Real-Time Concepts for Embedded Systems”. CPM Books, 2003, 294p.

[LIA11] Liao, X.; Srikanthan, T. “A Scalable Strategy for Runtime Resource Management on NoC Based Manycore Systems”. In: ISIC, 2011, pp. 297-300.

[LIN94] Lin, X.; McKinley, P.; Ni, L. “Deadlock-free Multicast Wormhole Routing in 2-D Mesh Multicomputers”. IEEE Transactions on Parallel and Distributed Systems, v.5(8), 1994, pp. 793-804.

[LUS11] Lusala, A.; Legat, J-D. “Combining SDM-BASED Circuit Switching with Packet Switching in a NoC for Real-Time Application”. In: ISCAS, 2011, pp. 2505-2508.

[MAD13] Madalozzo, G. "Controle Adaptativo para Atendimento a Requisitos de Aplicações em MPSoCs". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2013, 68p.

[MAR06] Martin, G.; “Overview of the MPSoC Design Challenge”. In: DAC, 2006, 6p.

[MAT10] Matos, D.; Concatto, C.; Kologeski, A.; Carro, L.; Kastensmidt, F.; Susin, A.; Kreutz, M. “Monitor-Adapter Coupling for NoC Performance Tuning”. In: SAMOS, 2010, pp. 193 – 199.

[MOL10] Möller, L.; et al. “Graphical interface for debugging RTL Networks-on-Chip”. In: BEC, 2010, pp. 181-184.

[MOR04] Moraes, F.; Calazans, N.; Mello, A.; Möller, L.; Ost, L. "Hermes: an Infrastructure for Low Area Overhead Packet-switching Networks on Chip". Integration, the VLSI Journal, Vol.38(1), 2004, pp.69-93.

[MOR12] Moraes, F; Madalozzo, G; Castilhos, G.; Carara, E. “Proposal and Evaluation of a Task Migration Protocol for NoC-based MPSoCs”. In: ISCAS, 2012, pp. 644-647.

[MOT11] Motakis, A.; Kornaros, G.; Coppola, Marcello. “Dynamic Resource Management in Modern Multicore SoCs by Exposing NoC Services”. In: ReCoSoC, 2011, pp. 1-7.

[MUR11] Murillo, L.; et al. “Debugging Concurrent MPSoC Software with Bug Pattern Descriptions”. In: S4D, 2011, 4p.

[MUR12] Murillo, L.; et al. “Scalable and retargetable debugger architecture for heterogeneous MPSoCs”. In: S4D, 2012, 6p.

[PRA11] Prada-Rojas, C.; et al. “A Generic Component-Based Approach to MPSoC Observation”. In EUC, 2011 , pp. 261-267.

[REK13] Rekik, W.; Ben Said, M.; Ben Amor, N.; Abid, M. "Virtual prototyping of multiprocessor architectures using the open virtual platform". In: International Conference on Computer Applications Technology, 2013 , pp.1-6

[TIL13] Tilera Corporation, “Tile-GX Processor Family”. Capturado em: http://www.tilera.com/products/processors/TILE-Gx_Family, Novembro 2013.

[SAI10] Said, M. B.; Loukil, K.; Ben Amor, N.; Abid, M.; Diguet, J.P. “A timing constraints Control technique for embedded real time systems”. In: DTIS, 2010, pp. 1 – 6.

[SAL11] Salah, Y.; Tourki, R. “Design and FPGA Implementation of a QoS Router for NoC”. In: International Conference on Next Generation Networks and Services, 2011, pp. 84-89.

[SAP12] Saponara, S.; Bacchillone, T.; Petri, E.; Fanucci, L.; Locatelli, R.; Coppola, M. “Design of a NoC Interface Macrocell with Hardware Support of Advanced Networking Functionalities”. IEEE Transactions on Computers, 2012, preprint.

[SHA11] Shabbir, A.; Kumar, A.; Mesman, B.; Corporaal, H. “Distributed Resource Management for Concurrent Execution of Multimedia Applications on MPSoC Platforms”. In: SAMOS, 2011, pp. 132-139.

[SIN13] Singh, A.; Shafique, M.; Kumar, A.; Henkel, J. “Mapping on Multi/Many-Core Systems: Survey of Current and Emerging Trends”. In: DAC, 2013, 10p.

[STA11] Stan, A.; Valachi, A.; Barleanu, A. “The design of a run-time monitoring structure for aMPSoC”. In: ICSTCC, 2011, pp. 1-4.

[STE12] Stefan, R.; Molnos, A.; Ambrose, A.; Goossens, K. “A TDM NoC supporting QoS, multicast, and fast connection set-up”. In: DATE, 2012, pp. 1283 – 1288.

[WAC11] Wächter, E. "Integração de Novos Processadores em Arquiteturas MPSOC: um Estudo de Caso". Dissertação de Mestrado, Programa de Pós-Graduação em Ciência da Computação, PUCRS, 2011, 92p.

[WAN10] Wang, Y.; Wang, Yu; Xu, J.; Yang, H. “Performance Evaluation of On-Chip Sensor Network (SENoC) in MPSoC”. In: ICGCS, 2010, pp. 323 – 327.

[WAN12] Wang, C.; Bagherzadeh, N. “Design and Evaluation of a High Throughput QoS-Aware and Congestion-Aware Router Architecture for Network-on-Chip”. In: Euromicro, 2012, pp. 457-464.

[WIN11] Winter, M.; Fettweis, G. “Guaranteed Service Virtual Channel Allocation in NoCs for Run-Time Task Scheduling”. In: DATE, 2011, pp. 1-6.

[WIS11] Wissem, C.; Attia, B.; Noureddine, A.; Zitouni, A.; Tourki, R. “A Quality of Service Network on Chip based on a New Priority Arbitration Mechanism”. In: ICM, 2011, pp. 1-6.

[WOL08] Wolf, W.; Jerraya, A.; Martin, G. “Multiprocessor System-on-Chip (MPSoC) Technology”. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, v.27(10), 2008, pp. 1701–1713.

[ZHA11] Zhao, J.; Madduri, S.; Vadlamani, R.; Burleson, W.; Tessier, R. “A Dedicated Monitoring Infrastructure for Multicore Processors”, IEEE Transactions on Very Large Scale Integration (VLSI) Systems, vol. 19-6, 2011, pp. 1011 – 1022.