Runtime QoS Support for MPSoC: a Processor Centric Approach

Marcelo Ruaro[1], Everton A. Carara[2], Fernando G. Moraes[1]

[1] PUCRS – FACIN – Av. Ipiranga 6681 – Porto Alegre – 90619-900 – Brazil
[2] UFSM – DELC – Av. Roraima 1000 – Santa Maria – 97105-900 – Brazil

marcelo.ruaro@acad.pucrs.br, carara@ufsm.br, fernando.moraes@pucrs.br

ABSTRACT

The MPSoC literature related to runtime support of Quality of Service (QoS) presents proposals related to the management of the interconnection infrastructure and the processing elements (PEs). The QoS management of computation resources is essential to fulfill real-time (RT) applications, as in multimedia systems, where dynamic workload and CPU sharing are commonplace. However, few works concerning QoS at the processor level for RT applications are found in the literature. The proposed work provides a runtime support for QoS acting in the PEs, coupled to a monitoring scheme at the task level. The main goal of the present work is to employ a low overhead task migration combined with task scheduling priority, to increase the computation resources for RT applications. An important feature of the proposal is to act indirectly over the QoS application, by minimizing the interference of the best effort (BE) tasks in the RT application performance. If the monitoring infrastructure still detects deadline misses, the system management then tries to optimize the RT application acting at the task level, migrating the affected RT task or modifying the scheduling policy. The NoC-based MPSoC was modeled and validated using an RTL description, with real applications. Results use throughput as the reference performance parameter. The proposed technique restored the RT applications performance after the introduction of disturbing applications, with a small reaction time.

Categories and Subject Descriptors

B.7.1 [**Integrated Circuits**]: Types and Design Styles – advanced technologies, VLSI (very large scale integration).

General Terms

Management, Design, Performance.

Keywords

MPSoC, QoS, Monitoring, Adaptation, Runtime

1. INTRODUCTION

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 higher computational power. This higher computing capacity is obtained through the reuse of components (processors, memories, routers), 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 it is predicted up to 1,000 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 use of resources at runtime [1][2].

The MPSoC literature related to runtime support for QoS presents proposals with the main goal to control the interconnection infrastructure [3][4], with a lack of works exploring a processor centric approach to provide QoS for RT applications. QoS at the processor level is related to reserve hardware resources to real-time (RT) applications [5]. Provide QoS adaptation is a key feature in MPSoCs, because several applications may execute simultaneously, with some of these with QoS constraints. Enabling runtime QoS adaptation allows to the system to decide itself to dedicate more resources for RT applications.

In the literature, related proposals optimize at design-time the workload distribution, not offering a *runtime management* of processing resources for RT QoS support. The main issues addressed in the literature provide workload balancing targeting fault tolerance and thermal management. However, the runtime management of computation resources is essential to fulfilling the RT constraints, as in multimedia systems, where dynamic workload and CPU sharing are commonplace [5][6].

The proposed work provides runtime support targeting QoS for RT applications, acting in the PEs, coupled to a monitoring scheme at the task level. The runtime adaptation employs low cost task migration, combined with scheduling priority to increase the computation resources for RT applications. An important feature of the proposal is to act indirectly over RT applications, by minimizing the effect of the BE tasks that disturb the performance of the RT application. The main idea can be summarized as follows. After detecting deadlines misses by throughput monitoring, the runtime management tries to move all disturbing BE tasks sharing resource with RT tasks. With this action, RT tasks are not penalized with the migration overhead. If the monitoring still detects deadline misses, the system management then tries to migrate RT tasks to a free PE. If the migration is not possible, the last action is to increase the OS scheduling priority for the affected RT task.

This work is organized as follows. Section 2 presents related works to QoS management. Section 3 details the monitoring model, used to detect deadline misses. Section 4 presents the QoS techniques, used by the method for QoS adaptation (section 5). Section 6 presents the obtained results, and Section 7 concludes this paper.

2. RELATED WORKS

Table 1 summarizes the state-of-the-art related to QoS management.

Monitoring is essential for runtime management support (second column of Table 1). Software monitoring [6][7][8][9] is used for workload management because it can capture information related to the applications' performance (as deadlines). However, software monitoring must be carefully employed, due to its higher intrusion compared with other monitoring architectures [1]. Commonly, the software monitoring implementation uses the same processing resources of applications, penalizing the performance of the monitored application. This work adopts a hybrid monitoring, with hardware monitors in PEs, and software monitoring in PEs dedicated to management, named *manager PEs*.

The software monitoring proposed in [6][7][8][9] does not detail how the constraints are transmitted to the system. In addition, the monitoring proposals are not generic, and tightly coupled to the adaptation proposal. Differently from these works, our monitoring scheme is generic, with an API responsible for transmitting to the OS the QoS constraints, obtained through application profile step (section 3.1).

Kornaros [10] proposes a hybrid monitoring infrastructure, with software support to configure the hardware monitoring components. This proposal adopts DVS to manage the system workload. As in our proposal, the monitoring information is filtered to reduce the amount of monitored data. Despite the similar runtime monitoring scheme, the goal of our proposal is runtime QoS adaptation.

The third column of the Table analyses the adaptation organization. Two organizations are adopted: distributed or centralized. Scalability is not a problem in distributed organizations [5][7][8][11] because each node takes local decisions. For example, if a given PE detects an event, as workload higher than a fixed threshold, it sends messages to its neighbors searching for PEs able to share the current workload. The partial system view of distributed organization may lead to inefficient adaptations [12]. On the other hand, centralized approaches [6][9][10][13] have a full system view, being not scalable because the number of PEs to manage increases with the system size. Our work adopts a trade-off between such approaches, characterizing a hierarchical organization with two levels. Hardware monitors compute in a distributed way the flows' throughput and latency (first level), and manager PEs take adaptation decisions with a systemic view (second level).

The fourth and fifth columns present the goals and how the adaptation is executed for each proposal. As can be observed, most works adopt workload balancing at runtime. The workload balancing is employed for fault tolerance, performance improvement and thermal management. However, an explicit QoS support is not addressed in these works, except by [5] and [11] were the user defines at design time the adaptation points to be used at runtime, i.e., not being completely adaptive. Abas et al. [6] also uses an indirectly workload management employing a runtime parallelization selection to improve the resources utilization in a virtualized platform.

Note that most works use task migration as the adaptation method [7][8][9][11][12][13][14], demonstrating the effectiveness of the approach to managing the workload. However, a common feature in all these works is the migration of the task whose performance or other parameters exceed a pre-defined threshold. Therefore, the task must stop, its context saved and migrated to another PE. All this process incurs in a performance penalty during the migration process. In general, the proposals adopting task migration search new ways to reduce the migration cost, and consequently the adaptation overhead.

Table 1 – State-of-the-art comparing works related to QoS management.

Proposal	Monitoring	Adaptation Organization	Goal of the adaptation	How adaptation is executed
Saint et al. [7] (2008)	Software – Using FIFO task constraints	Distributed at each PE OS	Workload balancing	Migrating affected task to a PE with smaller FIFO utilization
Zangh et al. [14] (2010)	No	Centralized (bus)	Fault Tolerance, workload balancing	Task computation scheduling algorithm balance the workload between each PE
Filho et al. [8] (2012)	Software – Task constraints (*ticks*)	Distributed at each PE OS	Dynamic workload balancing	Uses task migration to distribute the workload
Holmbacka et al. [11] (2013)	No – The user must define migration checkpoints	Distributed at each PE OS	Power efficiency, performance improvement	Task migration in a shared memory and MMU MPSoC: changing memory references
Quan et al. [9] (2013)	Software – Task computation and communication cost	Centralized	Energy Saving, performance improvement	Run time task mapping algorithm, combining static with dynamic mapping (task migration)
Joven et al. [5] (2013)	No - The user must define the QoS level manually	Distributed at each PE OS	QoS reconfiguration to workload management	oCMPI library manage the workload using traffic priority and CS
Kornaros et al. [10] (2011)	Hybrid – Tasks workload violations	Centralized	Power Management (DVS), workload management	A manager unit is responsible for apply thermal policies according to the monitors information
Abbas et al. [6] (2014)	Software – Applications constraints	Centralized	Resource optimization	Two parallelization (defined at design time) are selected at each application iteration
Canella et al. [13] (2012)	No	Centralized	System adaptivity	Use task migration based in Polyhedral Process Networks
This Proposal	**Hybrid – hw and sw monitors**	**Hierarchical with manager PEs controlling the PEs executing applications**	**QoS of processing resources**	**According with throughput violation a heuristics employ task migration and scheduling processor**

The present work also employs task migration but acting *indirectly* over the RT tasks. A QoS manager tries to migrate the BE (best effort) tasks interfering with the affected RT task. Only if it is not possible to move the interference to another place the heuristic migrates the affected RT task. This procedure frequently eliminates the adaptation overhead. Salami et al. [15] describe a possible ping-pong effect, where a given task migrates from one core to another repeatedly. Any migrated task in our proposal is migrated to PEs without RT tasks, avoiding in this way the ping-pong effect.

3. MPSOC AND MONITORING MODELS

The communication infrastructure adopts a baseline 2D-mesh NoC, with the following features:

- duplicated 16-bit physical channels, assigning high priority to channel 0 and low priority to channel 1 (high priority packets may use both channels);
- deterministic Hamiltonian routing [16] in channel 1 and partially adaptive Hamiltonian routing in channel 0;
- input buffering;
- credit-based flow control;
- simultaneous PS and CS.

The processing element (PE) connected to each NoC router contains the following modules: 32-bit Plasma processor (MIPS-like architecture), local memory, DMA (Direct Memory Access), NI (network interface).

The MPSoC contains *manager* PEs (*MPs*) and *slave* PEs (*SPs*). *MPs* execute heuristics to control the MPSoC, as task mapping [17], monitoring, and task migration. *SPs* run an OS (operating system), responsible for task communication, multi-task execution, and to execute user applications.

The monitoring includes hardware and software modules, characterizing a hybrid scheme. This scheme combines the fast response and low intrusiveness of hardware monitoring, with a system view of the software monitoring.

Figure 1 shows an overview of the proposed monitoring scheme, considering a 4x4 MPSoC instance, split into four 2x2 clusters. The monitoring modules are implemented in *SPs* and *MPs*. *SPs* contain the packet monitor (3.2). *MPs* contain the throughput and latency monitors (3.3). The QoS manager is also implemented in *MPs*, receiving events from the monitors. The QoS manager is the event manager [1], detailed in Section 4. The QoS manager receives the monitored data, and executes the QoS heuristics.

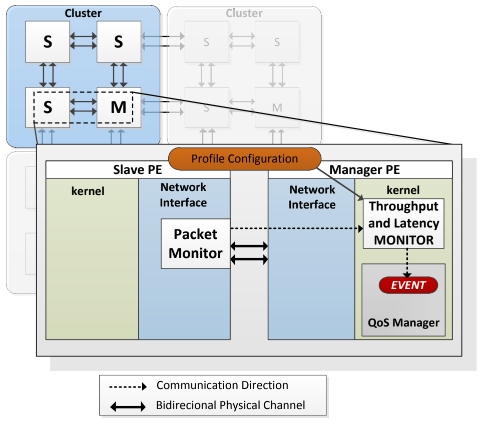

Figure 1 - Proposed monitoring system. 'S' corresponds to slave PEs, and 'M' to manager PEs.

To enable monitoring, the first action is to obtain the application profile (section 3.1). This action enables the programmer to define monitoring thresholds at design time, which are used to detect violations and generate events at runtime. The dotted arrow in Figure 1 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 monitor*. This monitor handles the received packets and if necessary generate *events*.

This monitoring architecture is classified as hierarchical, with two layers. The lower layer comprises *SPs*, which generate monitoring information to the second layer, the *MPs*, which implement the monitors and the QoS Manager.

3.1 Profile 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* (***ctp***) that will have the threshold and latency monitored. System calls were created to configure the monitors:

- *setRTResolution*: defines the resolution of the throughput monitor;
- *setQoSProducer*: called by the consumer task, enabling the communication monitoring between a consumer task and a producer task, also defining throughput and latency deadlines;
- *setQoSConsumer*: called by the producer task, informs to the OS 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 to the packet monitor to inspect the received packets.

3.2 Packet Monitor

The packet monitor is implemented in hardware, in the network interface of the *SPs*. The packet monitor observes all data packets entering in the local port with a monitoring bit enabled in the packet header (set by the *setQoSConsumer* system call).

The function of Packet Monitor is to extract from the data packet the *producer task id*, *consumer task id, packet size*, and *latency*. 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 the original data packet. This timestamp flit is generated during the packet creation in the producer task.

3.3 Throughput and Latency Monitor

The throughput monitor is software implemented, in the *MP*. The throughput monitor counts the number of received bits within the monitoring resolution time (defined by set *setRTResolution* syscall). When a monitoring packet coming from the packet monitor is received, the OS identifies the *ctp* 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 *ctp* 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.

The latency monitor is executed when the *MP* receives a monitoring packet. The *MP* identifies the *ctp* and calls the latency monitor. The latency monitor verifies if the latency carried in the monitoring packet is higher than the latency deadline (also 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 the QoS manager.

4. QOS TECHNIQUES

This section describes the two QoS techniques that act in computation performance of RT tasks.

4.1 Task Migration

In [18] 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 [18]. The proposal divides the amount of data to be migrated by memory sections, and migrates each memory section separately. Figure 2 details the new process. In event 1, the task migration process begins with the adaptation order sent by the QoS manager to the OS of the current task. 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 for selecting the processor to receive the task. When the processor receives the migration request, the OS immediately send the object code of the task to the 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 to execute" (READY), meaning that task is not waiting for a message from another task, and that task is not running. In this state, the task context is safely stored in the OS.*

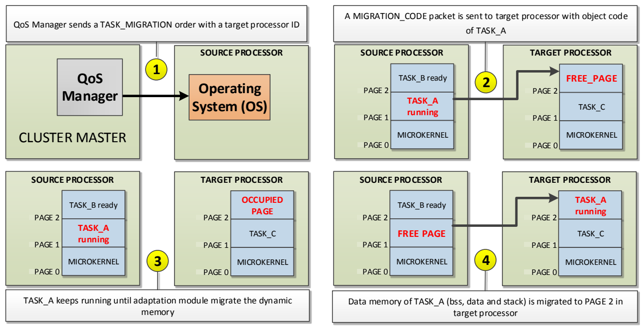
QoS Manager sends a TASK_MIGRATION order with a target processor ID
QoS Manager
CLUSTER MASTER
1
SOURCE PROCESSOR
Operating System (OS)
A MIGRATION_CODE packet is sent to target processor with object code of TASK_A
SOURCE PROCESSOR
TASK_B ready
TASK_A running
MICROKERNEL
PAGE 2
PAGE 1
PAGE 0
2
TARGET PROCESSOR
FREE_PAGE
TASK_C
MICROKERNEL
SOURCE PROCESSOR
TARGET PROCESSOR
OCCUPIED PAGE
3
TASK_A keeps running until adaptation module migrate the dynamic memory
FREE PAGE
4
Data memory of TASK_A (bss, data and stack) is migrated to PAGE 2 in target processor

Figure 2 - Optimized task migration protocol.

If the task is waiting for a message, this means that the task is blocked waiting a data packet from some given task (synchronous receive MPI protocol). Migrating the task in such state will incur in a message loss since the producer will send the message to an SP without the corresponding task to consume it. If the task is executing, the processor registers are in use, and the CPU registers are not safely stored by the OS.

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 2), concluding the task migration process.

4.2 Processor Scheduling Priority

The OS scheduler supports multi-task execution. The proposal adopts the Round-Robin Scheduling with Priority-based Preemption [19]. The processor is 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 the OS, which is restored when the task is re-scheduled.

All tasks are initialized with default and equal priority and time-slice. When the OS receives the adaptation order to change the scheduling priority of a given task, it increases its scheduling priority and time-slice.

5. COMPUTING QOS ADAPTATION

When a throughput event is received, the QoS manager calls the *ChangeQoSComputation* function (Figure 3). Its objective is to choose the appropriate adaptive technique to provide a computation QoS improvement. As the QoS manager analyzes a *ctp*, both producer and consumer tasks use the *ChangeQoSComputation* function. Definitions:

- ***resource***: page in the *SP* memory, used by the processor to execute one task (assumed paged memory organization). The number of *resources* for each SP is defined at design time.
- ***migrateBE** function*: tries to migrate a BE task to a processor without RT tasks executing. The selected processor is the one with the lowest communication cost between the migrated BE task and its communicating tasks.
- ***migrateRT** function*: if there are free PEs this function migrates an RT task to a free PE (without any task), selecting the one with the smaller communication cost. Otherwise, no action is taken.

The *ChangeQoSComputation* function receives as input an RT_{task} ID (producer or consumer), with QoS violations in the *ctp*. The first action (line 2) is to verify if the cluster has free resources to enable the task migration. If this condition is satisfied, the function *searchBEtask* searches for BE tasks in the same processor executing the RT_{task} (line 3). This function returns the task ID of the BE task (BE_{task}), or -1 if no BE task is found. If a BE task is found in the same processor executing the RT_{task}, the next action is to try to migrate the BE_{task} using the *migrateBE* function (line 5). The goal of this process is: "*if exists an RT task, sharing CPU resources with BE tasks, the first action is to try to migrate the BE tasks to another PE, improving the performance of the RT task by removing the disturbing tasks*". If it is not possible to migrate the BE_{task}, due to the lack of resources without RT tasks, the next step is to increase the scheduler priority of the RT_{task} (line 6).

If does not exist BE tasks running simultaneously with the $\mathrm{RT}_{\mathrm{task}}$, but other RT tasks share the processor with the $\mathrm{RT}_{\mathrm{task}}$, the procedure tries to migrate the RT_{task} to a free PE through the *migrateRT* function (line 8 and 9). This action is executed after trying to migrate the BE tasks because the migration process, although optimized, may penalize the RT_{task} performance due to the task stalling during the migration process.

```
1.   ChangeQoSComputation (input: $RT_{task}$)
2.     IF cluster has free resources THEN
3.        $BE_{task}$ ← searchBEtask($RT\_task$ processor)
4.        IF  $BE_{task}$ != -1 THEN
5.           IF migrateBE($BE_{task}$) == -1 THEN
6.               Increase CPU scheduling priority for $RT_{task}$
7.           END_IF
8.        ELSIF ∃ other RT task in the $RT_{task}$ processor THEN
9.             migrateRT($RT_{task}$)
10.       END_IF
11.    ELSIF ∄ other RT tasks in the $RT_{task}$ processor THEN
12.          Increase CPU scheduling priority for $RT_{task}$
13.    END_IF
14.  End ChangeQoSComputation
```

Figure 3 – Pseudocode of the computing QoS Adaptation heuristic.

In the absence of free resources, and if the CPU is not sharing the processor with other RT tasks, the RT_{task} has its scheduling priority increased (line 11 and 12).

The algorithm presented in Figure 3 tries to reserve one PE for 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.

6. RESULTS

This Section presents the results obtained with scheduling priority and task migration for QoS adaptation. 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 of the platform (SystemC).

Three scenarios were evaluated: *(i) best*: where each application is executed alone in the system, and each task is mapped to 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 the reference to set the latency and throughput deadlines (profiling step).

The MPSoC used in this test case contains 36 PEs (6x6), with four 3x3 clusters. Four RT applications are used (tasks graphs presented in Figure 4): *MJPEG*, *DTW* (Dynamic Time Warping), *audio_video* and *FFT*. Each cluster received one RT application, mapped together with synthetic applications that correspond to BE applications. The BE applications are executed simultaneously with the RT applications.

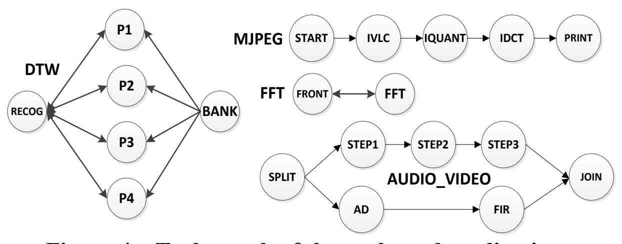

Figure 4 – Task graph of the evaluated applications.

Figure 5 presents the execution time of the RT applications. It is possible to observe that the adaptive techniques do not provide a significant improvement for the *audio_video* and *FFT* applications, but for *MJPEG* and *DTW* applications the adaptive technique reduced the execution time compared to a 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% lower than the *adaptation* scenario. The adaptive techniques do not reduce the execution time of the *audio_video* and *FFT* applications due to the small communication rate between tasks (most of the time the tasks are executing, not communicating). The major benefit of the adaptive techniques is to sustaining the throughput (i.e. QoS) as presented next.

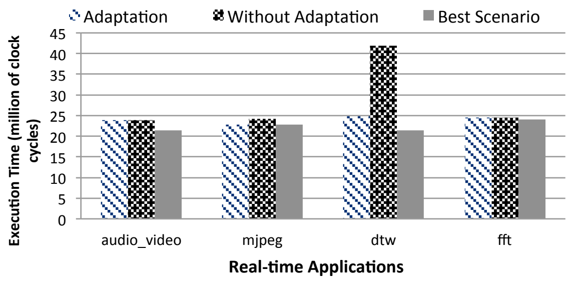

Figure 5 - Execution time of RT applications.

Figure 6 details the throughput for the *ctp iquant→idct* (MJPEG application), with the numbers corresponding to the moments where the adaption techniques are applied. The flow running in the *best* scenario presents a throughput of 24,435 bits/5ms. The throughput deadline is set to 24,190 (1% lower than best scenario), and the resolution time configured to 5 ms. The average throughput decreased to 22,862 bits/5ms in the scenario without adaptation. Adding the adaptive techniques, the average throughput increased to 24,192, returning to meet the specified constraint.

Another import point exposed by Figure 6 is the task migration performance. The advantage of the optimized task migration protocol is the smaller volume of information transmitted through the NoC and the absence of migration points defined by the programmer. Comparing the optimized protocol to the original one [17], 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 (~3,150 clock cycles) to the *ivlc* task of MJPEG application. The advantage of the presented protocol may be even higher with larger pages because in the reference protocol the entire task page is migrated, independently of the task code or data size.

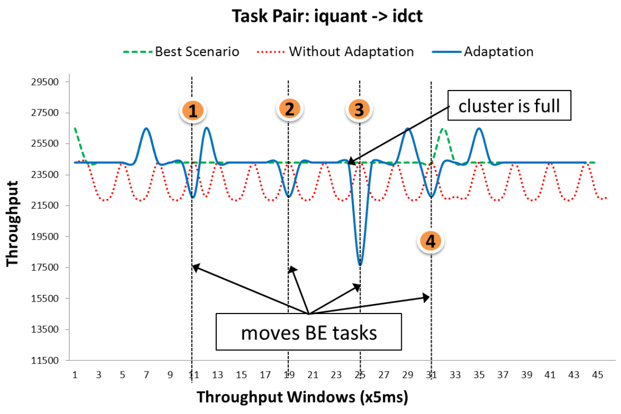

Figure 6 - Throughput results for the communicating task pair *iquant* → *idct*.

To put in perspective those values, [11] and [7] report a migration time of 100 ms and 131.35 ms, respectively (corresponding roughly to 1,000 times the time obtained by our method). A similar result to ours is presented by [13], with a migration time of 15,000 clock cycles for a task of the MJPEG application.

Figure 7 details the cluster task mapping, and the task migrations. The Figure shows only one cluster, the one that received the MJPEG application. Initially, the MJPEG application is mapped into a given cluster, with each RT task of the MJPEG application running individually at each PE, contributing to sustain the throughput according to the application requirements.

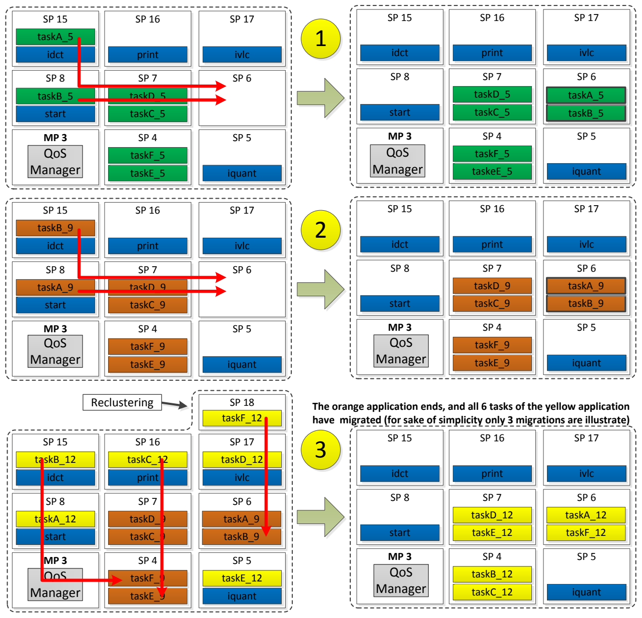

Figure 7 - Adaptation cluster steps for the MJPEG application.

In steps 1 and 2 of Figure 7, BE applications (in green and orange, respectively) are mapped into the cluster. The mapping heuristic searches free resources into the MPSoCs and may map BE tasks in PEs executing RT tasks. In step 1 the performance of the *idct* and *start* tasks are penalized due to BE tasks, decreasing the application throughput, and then triggering the adaptation (Figure 6 labels 1 and 2). In both cases, 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, and the throughput restored, as can be observed in Figure 6.

In step 3 the BE application that was mapped in step 2 remains running, and another BE application is mapped into the cluster (yellow). The cluster becomes full and a BE task (taskF_12) is allocated by reclustering in another cluster. At this moment, all MJPEG tasks are sharing CPU with BE tasks, and their respective throughput present an important reduction (Figure 6(3)). At this moment, the previous BE application – that is still running since step 2 – 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.

This example, Figure 7(3), also illustrates the runtime reclustering management. The reclustering procedure verifies if exists tasks executing in neighbor clusters. When a resource becomes available in the cluster, the *MP* (PE 3) verifies if there are tasks running in PEs outside of the cluster. If the condition is satisfied, the *MP* migrates the task back to the cluster. In this example, taskF_12 is migrated to PE6. The reclustering management restores the original cluster size at runtime, grouping communicating tasks near to each other, reducing the communication energy.

Figure 8 details the throughput for the *ctp bank*→*p2* (DTW application). This second example corresponds to the application having the highest execution time reduction applying the adaptive techniques (Figure 5). The flow running in the best scenario presents a throughput of 28,512 bits/5ms, with the deadline 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 reached 24,603 bits/5ms, presenting a throughput improvement of 70% compared to the scenario without adaptation.

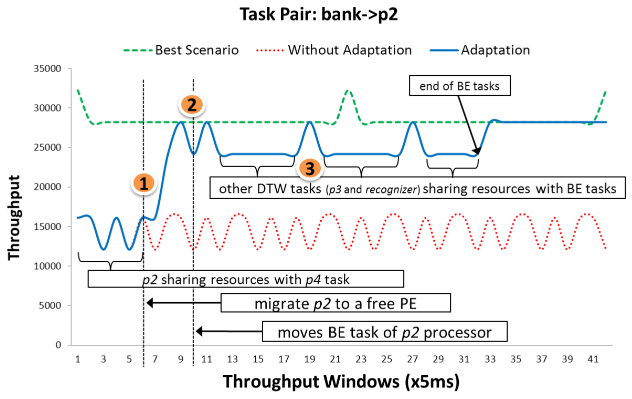

Figure 8 - Throughput results for the communicating task pair bank→p2.

Three moments are highlighted in Figure 8:

(1) The monitoring detects throughput violations generating 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 throughput peaks are observed, due to an increase in the scheduling priority of tasks *bank* and *p2*.

We define the *reaction time* as the time between the start of the BE interference and the triggering of the adaptation. The reaction time is proportional to the configured RT resolution for each *ctp* (*setRTResolution* system call). Both *ctp*$_s$ of MJPEG and DTW applications had the resolution time configured at 5 ms. For the MJPEG and DTW applications, the average reaction time was 4,73 ms and 2,17 ms, respectively (to put in perspective this value, it is in the same order of resolution than a real-time Linux). An important parameter affecting the reaction time is the communication volume of a given *ctp*. The DTW is more communication intensive than MJPEG, explaining the observed differences.

7. CONCLUSION

This work presented a runtime adaptive QoS management technique, with two different adaptive computing QoS techniques. To provide support to runtime adaptive QoS management, this work proposed a hybrid monitoring implementation that provided a small degree of intrusiveness, with a worst-case link usage equal to 0.8%, not penalizing the execution time of monitored applications. This proposal is scalable, because it employs a two-level hierarchical scheme, distributing the management and monitoring data among clusters.

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 migrate first BE tasks interfering with RT tasks, and then 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.

Suggestion for future works include: (*i*) QoS-aware task mapping, to minimize the BE mappings with RT tasks, reducing in such a way the number of migrated tasks; (*ii*) adaptive monitoring, to adjust the monitoring window according to the number of events; (*iii*) add another adaptive techniques, as DVFS.

8. ACKNOWLEDGEMENTS

The author Fernando Moraes acknowledges the support granted by CNPQ, processes 472126/2013-0 and 302625/2012-7; and CAPES process 708/11.

9. REFERENCES

[1] 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.

[2] Carara, E.; Calazans, N. Moraes, F. *Differentiated Communication Services for NoC-Based MPSoCs*. IEEE Transactions on Computers, v.63(3), 2014, pp 595-608.

[3] 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.

[4] 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, 6 p.

[5] Joven, J.; et al. *QoS-Driven Reconfigurable Parallel Computing for NoC-Based Clustered MPSoCs*. IEEE Transactions on Industrial Informatics, v.9(3), 2013, pp.1613-1624.

[6] Abbas, N.; Ma, Z. *Run-time parallelization switching for resource optimization on an MPSoC platform*. Design Automation for Embedded Systems, March, 2014.

[7] Saint-Jean, N.; Benoit, P.; Sassatelli, G.; Torres, L.; Robert, M. *MPI-Based Adaptive Task Migration Support on the HS-Scale System*. In: ISVLSI, 2008, pp.105-110.

[8] Filho, S.J.; Aguiar, A.; de Magalhães, F.G.; Longhi, O.; Hessel, F., *Task model suitable for dynamic load balancing of real-time applications in NoC-based MPSoCs*. In: ICCD, 2012, pp. 49-54.

[9] Quan, W.; Pimentel, A.D. *A scenario-based run-time task mapping algorithm for MPSoCs*. In: DAC, 2013, pp.1-6.

[10] Kornaros, G; Pnevmatikatos, D. *Hardware-assisted dynamic power and thermal management in multi-core SoCs*. In: GLSVLSI, 2011, pp. 115-120.

[11] Holmbacka, S.; Lund, W.; Lafond, S.; Lilius, J. *Task Migration for Dynamic Power and Performance Characteristics on Many-Core Distributed Operating Systems*. In: PDP, 2013, pp.310-317.

[12] Lee, C.; Kim, H.; Park, H.; Kim, S.; Oh, H.; Ha, S. *A task remapping technique for reliable multi-core embedded systems*. In: CODES+ISSS, 2010, pp. 307-316.

[13] Cannella, E.; Derin, O.; Meloni, P.; Tuveri, G.; Stefanov, T. *Adaptivity support for MPSoCs based on process migration in polyhedral process networks*. VLSI Design, 2012, Article 2.

[14] Zhang, Y.; Hao, Z.; Xu, X.; Zhao, W.; Wang, Z. *Workload-balancing schedule with adaptive architecture of MPSoCs for fault tolerance*. In: BMEI, vol.7, 2010, pp.2775-2779.

[15] Salami, B.; Baharani, M.; Noori, H. *Proactive task migration with a self-adjusting migration threshold for dynamic thermal management of multi-core processors*. The Journal of Supercomputing, March, 2014.

[16] 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.

[17] Mandelli, M.; Ost, L.; Carara, E.; Guindani, G.; Rosa,T.; Medeiros, G.; Moraes, F. *Energy-Aware Dynamic Task Mapping for NoC-based MPSoCs*. In: ISCAS, 2011, pp. 1676-1679.

[18] 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.

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