Self-Adaptive QoS Management of Computation and Communication Resources in Many-Core SoCs

MARCELO RUARO, Pontifical Catholic University of Rio Grande do Sul, Brazil
AXEL JANTSCH, TU Wien, Austria
FERNANDO GEHM MORAES, Pontifical Catholic University of Rio Grande do Sul, Brazil

Providing quality of service (QoS) for many-core systems with dynamic application admission is challenging due to the high amount of resources to manage and the unpredictability of computation and communication events. Related works propose a self-adaptive QoS mechanism concerned either in communication or computation resources, lacking, however, a comprehensive QoS management of both. Assuming a many-core system with QoS monitoring, runtime circuit-switching establishment, task migration, and a soft real-time task scheduler, this work fills this gap by proposing a novel self-adaptive QoS management. The contribution of this proposal comes with the following features in the QoS management: (*i*) comprehensiveness, by covering communication and computation resources; (*ii*) online, adopting the ODA (Observe, Decide, Act) runtime closed-loop adaptation; and (*iii*) reactive and proactive decisions, by using a dynamic application profile extraction technique, which enables the QoS management to be aware of the profile of running applications, allowing it to take proactive decisions based on a prediction analysis. The proposed QoS management adopts a decentralized organization by partitioning the system in clusters, each one managed by a dedicated processor, making the proposal scalable. Results show that the proactive feature accurately extracts the applications' profile, and can prevent future QoS violations. The synergy of reactive and proactive decisions was able to sustain QoS, reducing the deadline miss rate by 99.5% with a severe disturbance in communication and computation levels, and avoiding deadline misses up to 70% of system utilization.

CCS Concepts: • **Computer systems organization → Real-time system architecture**; **System on a chip**;

Additional Key Words and Phrases: Quality of service, many-core, self-adaptation, prediction

ACM Reference format:
Marcelo Ruaro, Axel Jantsch, and Fernando Gehm Moraes. 2019. Self-Adaptive QoS Management of Computation and Communication Resources in Many-Core SoCs. *ACM Trans. Embed. Comput. Syst.* 18, 4, Article 37 (June 2019), 21 pages.
https://doi.org/10.1145/3328755

Author Fernando Gehm Moraes is supported by FAPERGS (17/2551-0001196-1) and CNPq (302531/2016-5), Brazilian funding agencies. Author Marcelo Ruaro is supported by FAPERGS (18/2551-000501-0).
Authors' addresses: M. Ruaro, Pontifical Catholic University of Rio Grande do Sul, Av. Ipiranga, 6681, Porto Alegre, RS, 90619-900, Brazil; email: marcelo.ruaro@acad.pucrs.br; A. Jantsch, TU Wien, Gußhausstraße 27-29, Vienna, Austria; email: axel.jantsch@tuwien.ac.at; F. G. Moraes (corresponding author), Pontifical Catholic University of Rio Grande do Sul, Av. Ipiranga, 6681, Porto Alegre, RS, 90619-900, Brazil; email: fernando.moraes@pucrs.br.
Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
© 2019 Association for Computing Machinery.
1539-9087/2019/06-ART37 $15.00
https://doi.org/10.1145/3328755

1 INTRODUCTION

Advances like the IBM's 5nm silicon transistor [13] keep up Moore's law, paving the way to create even more densely integrated chips. Many-core systems with a dozen to hundreds of cores are a reality and are the state-of-the-art regarding computing capacity in a single chip [3]. Many-cores provide outstanding processing power, but also pose challenges for temperature management (dark silicon), energy consumption, security, and quality of service (QoS).

Due to the number of resources to manage and the unpredictability that many-core systems have, self-adaptive management becomes fundamental to address such challenges [6]. While in an adaptive system the adaptation process may be triggered from the outside, e.g., by an application management layer or a human operator, a self-adaptive system itself identifies the triggering condition and initiates the adaptation process [9, 12]. Dutt et al. [6] present a self-adaptive hierarchical management, with the goal to ensure scalability in large many-core systems. At the lower level, monitors/sensors collect the system's status (resources, applications, constraints) and send it to a high-level management unit, which can make decisions according to the specified requirements.

Many-core resources can be classified into two main groups: computation and communication. Communication resources are concerned with transmitting data. Features as scalability and parallelism on the communication flows make Networks-on-Chip (NoCs) the communication infrastructure adopted in many-cores. Flow priority, Time Division Multiplexing (TDM), Spatial Division Multiplexing (SDM), and Circuit-switching (CS) are examples of techniques to achieve QoS at the communication level [2, 10, 14, 15]. Computation resources are related to data processing, including processors and memory components. Many-cores adopt different memory schemes, from distributed shared memories to local scratchpad memories. QoS techniques for computation resources are centered on real-time task schedulers and memory schedulers in a fine-grain level [17], and task mapping and task migration in a coarse-grain level [11].

The design of a many-core system with QoS support requires comprehensive and self-adaptive features, including actuation at both computation and communication resources. An efficient dynamic task mapping that considers computation and communication [25] is the first step toward QoS support in a many-core. Alone, task mapping cannot handle unpredictable events related to variable workload and interference in the communication infrastructure. Related works [2, 10, 11, 14, 17, 20–23, 26] on self-adaptive QoS mechanisms for many-cores focus on techniques addressing either communication or computation resources. However, the literature lacks a comprehensive framework that manages in an integrated way the resources of many-cores to meet QoS.

The *goal* of this work is to fulfill this gap by proposing a comprehensive self-adaptive QoS management for soft real-time applications on many-core platforms.

The self-adaptive QoS management proposal is the *original contribution*, with the following properties:

(a) *comprehensiveness*: the self-adaptive QoS management addresses communication and computation resources;
(b) *online*: not requiring the knowledge of the applications set at design-time;
(c) *proactive*: besides reactive action, the QoS management also employs a dynamic application profile extraction (DAPE) technique to profile tasks on-the-fly to direct resources according to such profile.

To achieve the proposed goal, this works adopts the ODA (Observe-Decide-Act) closed-loop management [8], a systematic and modular self-adaptation method, dividing the system components in roles for Observation (monitoring), Decision heuristics, and Actuation. The

Table 1. Related Works on Self-Adaptive QoS for Many-Cores

Work	QoS focus	Method	QoS technique
[5]-2013	Computation	Online Task Scheduler	Task Mapping
[11]-2014	Computation	Dynamic Specification Behavior and Dynamic Mapping	Task Remapping
[17]-2014	Computation	Hierarchical Scheduler	Task Migration
[18]-2015	Computation	Cluster Scheduler	Task Mapping/Task Migration
[20]-2016	Computation	Dynamic Mapping Based on Prediction	Task Migration
[23]-2016	Computation	Dynamic Task Scheduler	Task Migration/Task Scheduler
[14]-2010	Communication	Bandwidth Self-Adaptation	Flow Priority
[10]-2013	Communication	Expose NoC QoS Services	Flow Priority/CS
[2]-2013	Communication	Proactive CS Establishment	CS
[21]-2015	Communication	Self-Adaptive Management	Flow Priority/CS
[22]-2018	Communication	Self-Adaptive Management	CS

self-adaptation management receives QoS fulfillment monitoring data (deadline miss, latency miss) and applications QoS feedback (notifications about runtime workload changes), enabling the QoS management to act reactively according to the severity of the events. Additionally, due to the DAPE technique, the QoS management continuously observes the tasks' profile, aiming to early improve the resource allocation and helping to avoid future QoS violations.

The scope of this work is to present the high-level self-adaptive QoS management. The low-level adaptive QoS techniques, responsible for reconfiguring at runtime the system, use techniques previously developed to meet QoS at a specific level: (*i*) communication QoS: circuit-switching establishment and release [22]; (*ii*) computation QoS: real-time task scheduler with task migration [23].

This article is organized as follows. Section 2 reviews related work, and Section 3 presents the many-core architectural features and the application model. Section 4 introduces DAPE, and Section 5 details the proposed self-adaptive management technique. Section 6 presents the experimental results, and Section 7 concludes the article.

2 RELATED WORK

This section discusses related works according to the main goal of this work: self-adaptation for QoS. Several proposals provide runtime QoS mechanisms for many-core systems, with self-adaptive techniques, targeting resource management (dynamic mapping, task migration, task scheduling, flow priority, and CS). Table 1 presents the self-adaptive QoS proposals most related to this work. The second column in the table presents the QoS focus, i.e., if the proposed technique is applied at the communication or computation levels. Next, the third column details the specific method to meet the QoS constraints. The last column, QoS technique, refers to the mechanisms and policies adopted to meet the QoS goals.

Several works [5, 11, 17, 18, 20, 23] adopt dynamic task remapping/migration to answer to the workload changes or real-time violations, addressing QoS at the computation level. Some of these works [11, 18, 20] adopt a hybrid task remapping heuristic, assuming that application characteristics are known at design-time. In contrast, we assume the application set is unknown at design-time. The proposed DAPE observes the running applications, creating an online profile. In Ref. [11], applications can tune the workload at runtime by using an API. This feature provides

high flexibility to an application to change its workload. Our work enables the workload reconfiguration by an API, where each task can change its real-time constraints at runtime. Most works assume a hierarchical management organization [17, 18, 20, 23], which distributes the management load by adopting a cluster-based organization, with a set of cores managed by a cluster manager. This work also adopts a hierarchical organization, with slave cores running the user's applications and sending monitored/feedback data to a cluster manager, which executes the QoS management.

The aforementioned works, while satisfying the computation constraints, also try to reduce the communication cost by mapping tasks closer to each other [16]. However, these approaches address communication QoS indirectly and are not able to handle unpredictable events that can disturb the traffic in the network, e.g., a high-priority flow crossing the communication path between two RT tasks. To mitigate this disturbance, works focusing only on computation QoS have to migrate the affected tasks to other processors, instead to act directly at the communication level.

The works in Refs [14] and [21] address QoS at the communication level. Several works develop techniques to implement a QoS-driven infrastructure considering only the NoC (e.g., see Refs [2] and [26]). Joven et al. [10] expose the communication QoS support to the software layer enabling the developer to define the QoS constraints. Abousamra et al. [2] observe the message requests to set proactive CS, which is used for future message deliveries. Authors in Ref. [14] propose a self-adaptive mechanism that exposes the hardware through a set of registers, allowing to program the QoS constraints for a bus-based SoC. The work in Ref. [21] proposes a self-adaptive flow priority management and CS establishment based on latency and throughput constraints. The authors of Ref. [22] propose a runtime CS based on a Software-Defined Networking (SDN) paradigm, enabling to establish CS paths that remain during the whole application lifetime.

As also can be observed in related works, task scheduling/migration, and CS stand out as techniques to provide QoS at computation and communication levels, respectively. The novelty of our work is a unified self-adaptive QoS management addressing QoS of computation (task scheduling/migration) and communication (CS) for soft real-time applications.

An important feature to leverage a fully autonomous system is the ability to extract the application's profile. References [11] and [20] obtain the applications' task graph mixing design-time and runtime steps. A dynamic mechanism has access to the application graph, using it to optimize runtime decisions, such as application remapping. This approach simplifies the runtime techniques because it provides a detailed application profile. However, it increases the complexity of the application development because the developer is in charge to provide an accurate application profile at design-time. Ganeshpure et al. [7] propose a runtime technique that extracts the communication task graph of the applications. The operating system implements this extraction by observing the execution phases for each task.

Our work adopts a more flexible approach. As in Ref. [7], the operating system extracts the tasks' behavior. However, Ref. [7] uses less than 200 iterations to profile the application while our technique continuously extracts the application profile. This feature enables to support applications with dynamic behaviors, i.e., the workload changes at runtime (common in multimedia applications). Additionally, we are concerned with scalability since DAPE adopts a hierarchical organization with monitors at each core, sending the task level information to a high-level manager.

3 BASELINE PLATFORM AND ASSUMPTIONS

3.1 Hardware and Software Platform

The baseline many-core architecture adopted in this work is an open-source project, available online [19] and illustrated in Figure 1. The processing elements (PE) are interconnected through a

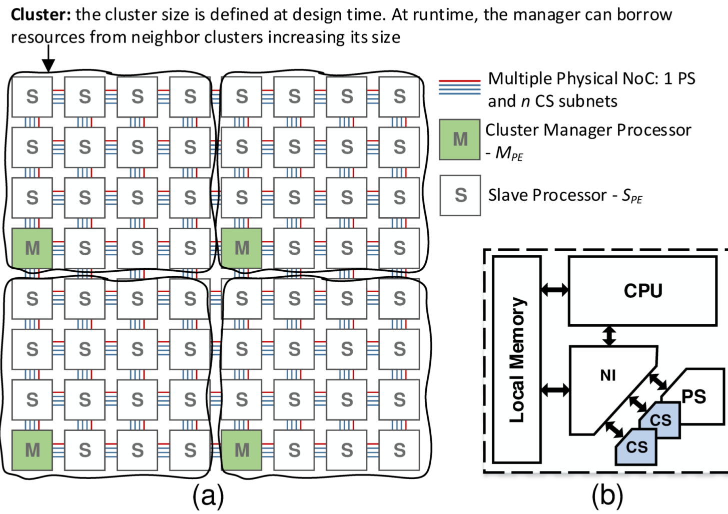


Fig. 1. (a) Overview of the baseline many-core system (four 4x4 clusters). (b) Overview of the PE architecture.

2D-mesh NoC network. Each PE contains a 32-bit processor (MIPS architecture), a local scratchpad memory, and a Network Interface (NI) connected to the packet-switching (PS) and circuit-switching (CS) routers.

This work adopts a memory hierarchy, with scratchpad memories storing code and data. The reasons for adopting such memory hierarchy are twofold. First, this model enables an easy understanding and reproduction of the QoS proposals. Second, it enables to capture accurate results, at the clock-cycle level, due to the usage of an RTL platform model. Architecture and QoS mechanisms are *orthogonal* concepts. Architectural features, like caches, out-of-order execution, or different hardware affect the communication at the processor level and the communication at the NoC level. The proposed QoS mechanisms monitor the computation and communication loads, firing a given action according to the processor load and NoC traffic.

At the software level, PEs assume two roles: slave PEs—S_{PE}, and manager PEs—M_{PE}. S_{PE}s execute an in-house μkernel (small operating system) and user's tasks. The μkernel executes task scheduling, inter-task communication (message passing), multitasking, NoC interruptions, and system calls. This μkernel can be easily customized according to specific goals (e.g., QoS, security, fault-tolerance). M_{PE}s execute system management algorithms, as application admission, task mapping, reclustering, and the proposed self-adaptive algorithms. This architecture can also run Linux-based operating systems. Abich et al. [1] use an untimed model of the platform describing it with Open Virtual Platform (OVP), replacing the μkernel by a Free Real-Time Operating System (RTOS) Linux distribution.

Figure 1 presents the many-core organized with four clusters, each one having 1 M_{PE} and 15 S_{PE}s. The reason to adopt this hierarchical organization is to ensure scalability by distributing the QoS management actions at different M_{PE}s. The cluster size is defined at design time. At execution time, when the cluster has all its resources in use, an M_{PE} may borrow resources from neighboring clusters, in a process named *reclustering* [4]. According to Ref. [4], a cluster size with 18 (6×3) or 16 (4×4) S_{PE}s represents a good tradeoff between execution time optimization and resources reserved for management.

The many-core supports Best-Effort (BE) and soft Real-Time (RT) applications. Task graphs describe the applications (Figure 2(a)), with nodes corresponding to tasks and edges denoting the

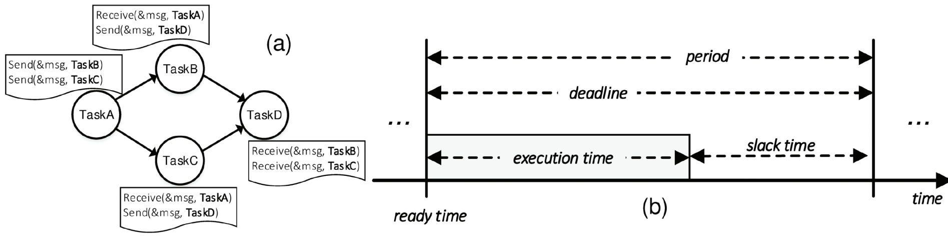


Fig. 2. (a) Application model. (b) RT task model.

communication between tasks. Each edge is a Communicating Task Pair (CTP) with a producer task and a consumer task. Each RT task has timing constraints (Figure 2(b)) modeled as *period*, *deadline*, *execution time*. BE tasks do not have timing bounds and explore the slack time of RT tasks. All tasks communicate based on Message Passing Interface (MPI) with *Send* and *Receive* primitives. Receive primitives block the consumer task execution and fire a request to a producer task, which handles the request and delivers the data when ready, unblocking the consumer task.

3.2 Adaptive Quality of Service Techniques

This section reviews the adaptive QoS techniques used in the proposed high-level self-adaptive QoS management. Those low-level techniques do not correspond to a new contribution since they were previously proposed to ensure QoS at the communication or computation level.

3.2.1 Dynamic Circuit-Switching (CS)—[22]. The CS support employs a Multiple Physical Network (MPN) architecture. The MPN contains one packet-switching (PS) sub-net (red wires in Figure 1) and CS sub-nets (blue wires in Figure 1). The PS sub-net has a conventional PS router, supporting XY routing and credit-based flow control. The CS sub-nets contain simple and programmable CS routers managed by software (Software-Defined Networking (SDN)). One CS router corresponds to 20% of one PS router area. The SDN paradigm implies that the complex logic for runtime CS establishment is removed from the routers' level (hardware), assigned to a CS-Controller that is software implemented. The CS-Controller is allocated as a real-time task in an S_{PE}. Its role is to handle CS requests from M_{PE}s, to search a path using a shortest-path algorithm, and to define a path by physically configuring the CS routers. The CS routers have the role of forwarding packets from a given input port to an output port according to a look-up table configured by the CS-Controller at runtime. The PS sub-net is used for best-effort flows and management traffic, and CS sub-nets are used for real-time flows.

3.2.2 Dynamic Real-Time Task Scheduler—[23]. The dynamic task scheduler supports best-effort and soft real-time tasks. It is hierarchically organized in two levels. At the lower level, a task scheduler algorithm executes at each S_{PE}. Its role is to schedule the Central Processing Unit (CPU) time according the least slack time (LST) algorithm, which assigns a higher priority to tasks with the least slack time to execute. Best-effort tasks explore the slack time left by the real-time task execution. At the higher level, a cluster scheduler on the M_{PE} executes task mapping when a given application enters the system and handles task migration. The task migration is triggered according to the decisions made by the algorithms proposed in Section 5. The scheduler has an important feature that enables a task to reconfigure its real-time constraints at runtime by calling an API that allows tasks to change its period, execution time, and deadline.

3.2.3 Task Migration—[24]. The task migration protocol is a lightweight process optimized for many-core systems with a distributed memory hierarchy. Relevant features of the protocol include:

there is no need to replicate the code of the tasks; it is not necessary to modify the source code neither to add checkpoints; support to simultaneous migrations; and inter-task synchronization without migrating produced messages. The task migration is an operating system (OS) service, triggered by the algorithms presented in Section 5. The OS handles this order by configuring the NI to transfer first the text section to the target PE through a MIGRATION_CODE message. The task continues its execution up to reach a safe state (ready to be scheduled, without waiting for incoming messages). Reaching this state, the task stops, and the OS saves the task context, transferring the CPU registers and the data memory sections to the target PE by a DATA_MIGRATION message. When the target PE receives the task data, its OS restores the task context, and the task may be scheduled. The overhead of the task migration protocol comes mainly from the second part of the process, where the task data is transferred. This task migration protocol presents a low latency compared to the state-of-the-art [24].

3.2.4 QoS Fulfillment Monitoring—[21, 23]. QoS monitoring is supported by observing the communication and computation parameters for real-time tasks. At the computation level, the monitoring consists of deadline misses that are captured by the local scheduler at the S_{PE} level, and sent to the M_{PE}. At the communication level, the monitoring consists of a latency monitor implemented in the μkernel. The latency monitor compares the current latency with a threshold value. The threshold may be either configurable or computed by the monitor itself based on previous observations. If the monitored latency is higher than the reference value, the monitor generates a latency miss message to the M_{PE}.

4 DYNAMIC APPLICATION PROFILE EXTRACTION (DAPE)

This section details the dynamic application profile extraction (DAPE) technique. DAPE also takes advantage of the hierarchical system organization. The lower level, implemented in the S_{PE}s, and the higher level, implemented in the M_{PE}s. The DAPE process comprises data extraction (within S_{PE}s) and data analysis (within M_{PE}s). The QoS management uses the results of this analysis to perform its proactive decisions (described in Algorithm 3).

At each S_{PE}, the μkernel monitors the tasks' profile at runtime observing the following parameters for each real-time task t: (*i*) computation, T_p, the portion of time where t is using the CPU; (*ii*) communication, T_m, the portion of time where t is waiting for a requested message from a producer task; (*iii*) idle time, T_i.

The profile monitoring extracts the relative amount (percentage) of T_m, T_p, and T_i, for each task periodically, over non-overlapping windows, where $T_m + T_p + T_i = 1$. The task scheduler computes T_p and T_i. The communication API computes T_m. T_m is computed from the perspective of the consumer task, evaluating the time spent between the requisition of a message until its reception. T_m is a function of three factors: (*i*) the time spent by the producer task to generate the requested message; (*ii*) the message size; (*iii*) the NoC congestion. Note that the communication percentage is computed only for the received messages because the μkernel adopts a non-blocking send operation (produced but not consumed messages are locally stored in a buffer). In scenarios with congestion in the NoC, the observed communication time tends to increase when the consumer task spends more time waiting for messages. This behavior helps to mitigate network congestion because the management will pay more attention to affected communications since the task will have a higher communication profile.

Each S_{PE} sends the monitored profile periodically to its M_{PE}, which implements the DAPE data analysis. The M_{PE} handles the received profile applying an accumulated mean of the received profile with the past profiles. The resulting value is used by the QoS management to estimate the profile of each task.

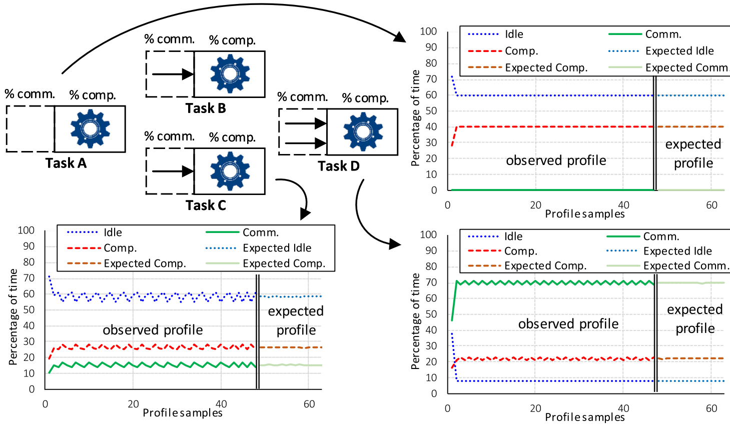


Fig. 3. Overview of the dynamic application profile extraction (DAPE) method.

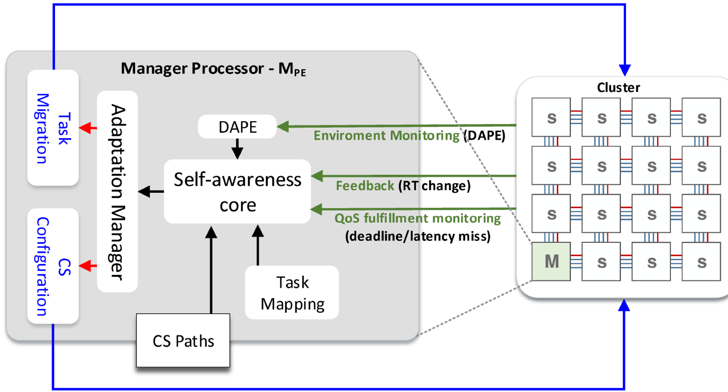


Fig. 4. Organization of the self-adaptive QoS management. A comprehensive dataset feeds the self-awareness core: applications' feedback, QoS fulfillment monitoring, DAPE, task mapping, and CS paths (provided by the CS-Controller). In the figure: green arrows represent observation messages, red arrows correspond to decision actions, and blue arrows correspond to the actuation messages.

Consider as an example the task graph of Figure 2(a) and assume that each task executes the same computation load periodically. Figure 3 presents the profile graphs according to the DAPE method. Task A does not receive packets from other tasks, thus $T_m = 0$. Tasks B and C receive packets from task A, resulting in a mixed profile with $T_p = 26\%$ and $T_m = 15\%$ (tasks B and C have similar graphs). Finally, task D has two communication flows, receiving packets from tasks B and C. In task D, the communication is higher, as depicted in the graph of Figure 3.

5 SELF-ADAPTIVE QOS MANAGEMENT

This section details mainly the contribution of this work. The self-adaptive QoS management is implemented at each cluster's M_{PE}. The QoS management adopts the Observe, Decide, Act (ODA) method [8]. The ODA methodology describes a closed-loop that is constantly aware of the system status and divides the role of each component into observation, decision, and actuation. It is generic and can be adapted to different many-core architectures. Figure 4 presents an overview of the self-adaptive QoS management implemented according to the ODA method.

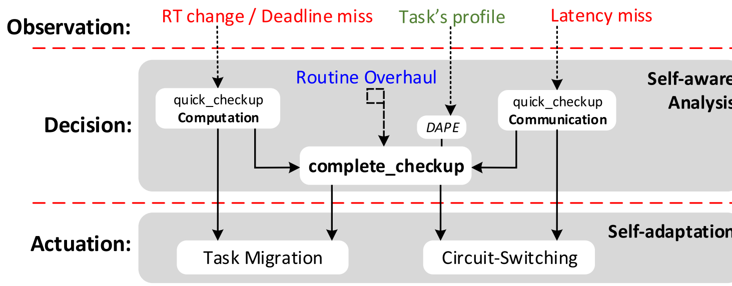


Fig. 5. Self-adaptation QoS management flow, executed by the manager processors (M_{PE}).

Observation: There are three message classes sent by S_{PE}'s to the M_{PE}:

- *Feedback* messages: provide performance figures related to the tasks. Feedback messages comprise changes on RT constraints (period, deadline, task execution time). Tasks are allowed to change their RT constraints at runtime through an API, enabling flexible workloads [11, 23]. When this RT change occurs, a message is sent to the M_{PE} with the new RT constraints.
- *Environment monitoring* messages: allow the manager to gather periodic information about the status of the resources and applications. These messages comprise the DAPE data with the extracted profile of each task.
- *QoS fulfillment monitoring* messages: warn violations related to the QoS fulfillment. These messages comprise packet latency miss (communication) and deadline miss (computation) violations.

Decision: This work proposes a *self-awareness core*, achieving this awareness by accessing a wide range of high-level information at runtime:

(1) observation messages;
(2) tasks' location (provided by the task mapping heuristic);
(3) CS paths (provided by the SDN-based CS-Controller).

The *self-awareness core* acts as a trigger deciding which system component is the target of the adaptation and when the adaptation occurs. The *self-awareness core* decides reactively as well as proactively according to the algorithms presented in Sections 5.1 and 5.2.

Actuation: This work proposes an *Adaptation Manager* (AM) to work together with the self-awareness core. Whereas the self-awareness core takes decisions, the AM is in charge to take the actions based on the decisions. Thus, the AM manages the reconfiguration of the system resources controlling the task migration protocol and CS configuration (establishment/releases) protocols, according to decisions made by the self-awareness core. These protocols change the resources of computation and communication physically. The AM also ensures that the resources will be correctly updated after the adaptation to be used by the self-awareness core in future decisions.

Figure 5 presents the self-adaptation flow. The method is similar to a health checkup, with *quick* and *complete* checkup.

- *Quick-checkup*: the quick checkup acts when a symptom appears, such as a feedback message notifying a real-time constraint modification or a deadline/latency miss, leading to *reactive* actions. According to Figure 5, there are two quick checkup functions, one to deal with computation events (RT change and deadline miss) and the other one to deal with communication events (latency miss).

— *Complete-checkup*: the complete checkup enables to evaluate an application status in detail, evaluating all application's tasks. The complete-checkup is triggered in two situations: (*i*) when the quick-checkup was not able to solve the violation responsible for invoking it; (*ii*) periodically, by the *routine overhaul* trigger, which may lead to a *proactive* action.

The next sections detail the quick and complete checkup algorithms. They adopt the following design-time parameters:

(1) ***cpu_TH:*** maximum allowed CPU utilization per processor.
(2) ***comp_profile_TH:*** threshold defining a computation intensive task. Higher values reduce the proactive actions at the computation level, and vice-versa.
(3) ***comm_profile_TH:*** threshold defining a communication intensive task. Higher values reduce the proactive actions at the communication level, and vice-versa.
(4) ***comp_profile_sum_TH:*** threshold used to fire a proactive adaptation at the computation level when two or more tasks share the same PE, corresponding to the total computation load of the tasks on a given processor. It corresponds to a tradeoff between time to react (higher values delay proactive actions), and the number of proactive actions (lower values trigger proactive actions too early).
(5) ***deadline_TH:*** maximum number of deadline misses in a sampling period.
(6) ***latency_TH:*** maximum number of latency misses in a sampling period for a given communicating task pair.

5.1 QUICK-CHECKUP Algorithms

The *feedback* and *QoS fulfillment monitoring* messages fire the QUICK-CHECKUP algorithms, presented in Algorithms 1 and 2.

ALGORITHM 1: QUICK-CHECKUP-COMPUTATION

```
  Input: Target task: task, and target task's CPU address: task_cpu
  Output: Void, the algorithm decide or not to call another function
1 cpu_util ← get_cpu_utilization(task_cpu);
2 deadline_miss_rate ← get_deadline_miss_rate(task);
3 if cpu_util > cpu_TH or deadline_miss_rate > deadline_TH then
4 |   if task_migration(task) = FALSE then
5 |   |   COMPLETE-CHECKUP(task's application)
6 |   end
7 end
```

The goal of the QUICK-CHECKUP-COMPUTATION algorithm (Algorithm 1) is to evaluate reactively when it is necessary to migrate a task due to a computation change caused by an RT constraint change or by a deadline miss on the target task. The algorithm inputs are the task identification (*task*) and the CPU address executing the task (*task_cpu*). Line 1 computes the CPU utilization where the task is running, and line 2 calculates the deadline miss rate (percentage) for the task. The deadline miss rate is the relationship between the number of missed deadlines divided by the number of tasks' periods since the last application adaptation. These two parameters (*cpu_util* and *deadline_miss_rate*) are used to verify the status of the task (line 3). If the *cpu_util* or the *deadline_miss_rate* is higher than the predefined thresholds (*cpu_TH* and *deadline_TH*), the decision is to migrate the task. If the migration fails (no available processor in the whole system with enough CPU utilization to receive the task), the COMPLETE-CHECKUP (line 5) is called with the goal to try to migrate another critical task of the same application. If there is more than one

ALGORITHM 2: QUICK-CHECKUP-COMMUNICATION

```
  Input: An CTP: prod_task, cons_task
  Output: Void, the algorithm decide or not to call another function
1 ctp_latency_count ← get_ctp_latency_number(prod_task, cons_task);
2 if ctp_latency_count >2*latency_TH then
3 |   COMPLETE-CHECKUP(tasks' application);
4 else
5 |   if ctp_latency_count > latency_TH then
6 |   |   CS_configuration(prod_task, cons_task);
7 |   end
8 end
```

available PE, the closest processor from the original position of the task is selected. Note that the task migration of the most critical task may fail. In this case, it is not possible to recover the application from the deadlines misses. This situation only occurs in systems with full utilization, i.e., all processors executing tasks with an important load. The migration protocol does not consider the NoC state since there is the communication level management that can establish CS if necessary.

The goal of the QUICK-CHECKUP-COMMUNICATION algorithm (Algorithm 2) is to evaluate reactively when it is necessary to establish CS connection for a given *CTP* due to a communication interference inducing a latency miss. The algorithm receives as input a *CTP*, with a producer (*prod_task*) and consumer (*cons_task*) task identifiers. Line 1 obtains the total number of latency misses since the last application adaptation. When the latency misses exceed 2×*latency_TH* (line 2), the COMPLETE-CHECKUP is invoked. If the latency misses exceed *latency_TH* (line 5), the algorithm tries to establish a CS path between *prod_task* and *cons_task*. Note that while the CS is not established (no available path), the *CTP* may continue generating latency misses, increasing the number of latency misses, which can result in the fulfillment of condition at line 2 and the invocation of the COMPLETE-CHECKUP algorithm. Thus, the COMPLETE-CHECKUP may act in the penalized *CTP* by migrating the most critical task (line 5 of Algorithm 3).

5.2 COMPLETE-CHECKUP Algorithm

The main goal of the proposed self-adaptive QoS management is to reduce the reactive actions, acting proactively when possible to avoid future QoS violations. Algorithm 3 presents the COMPLETE-CHECKUP algorithm, which receives as input an *application* identifier. This algorithm has two operating modes, reactive and proactive.

The activation of the reactive mode occurs when a QUICK-CHECKUP algorithm fails and calls the COMPLETE-CHECKUP from it, comprising the code lines 2 to 5. Line 2 ranks the application tasks according to the QoS violation severity, using Equation (1).

$$r_T = d_m + l_m + 10.(u_{CPU} > cpu_TH?1 : 0), \tag{1}$$

where: r_T is the task rank, d_m is the number of deadline misses, l_m is the number of latency misses, and u_{CPU} is the real-time CPU utilization where the task is executing.

According to Equation (1), the rank of a given task is higher when it is running on a CPU with a utilization higher than *cpu_TH*. Line 4 selects the most critical task, (i.e., the one with the highest rank), and the task migration is fired to *selected_task* (line 5). The decision to migrate the most critical task is due to the fact that the QUICK-CHECKUP algorithms failed to define a CS connection or to migrate a task. Note that the reactive mode acts only on one task of the application. As one single QoS adaptation can impact in the whole application performance, gradual steps are preferable to simultaneous adaptations.

ALGORITHM 3: COMPLETE-CHECKUP

```
   Input: application
   Output: Void, the algorithm decide or not to call another function
 1 task_migration_list ← Ø;
 2 task_rank[] ← computes_tasks_score(application);
 3 if task_rank[] ≠ EMPTY then
 4 |   selected_task ← get_high_task_score(task_rank[]);
 5 |   task_migration(selected_task);
 6 else
 7 |   for t_i ∈ application do
 8 |   |   comp_task_num ← get_num_comp_tasks(t_i_cpu);
 9 |   |   if comp_task_num > 1 and get_comp_sum(t_i_cpu) ≥ comp_profile_sum_TH then
10 |   |   |   task_migration(t_i);
11 |   |   |   task_migration_list ← t_i;
12 |   |   end
13 |   end
14 |   for t_i ∈ application and t_i ∉ task_migration_list do
15 |   |   if get_comm_profile(t_i) ≥ comm_profile_TH or get_comm_profile(t_i) ≥ get_comp_profile
   |   |   (t_i) then
16 |   |   |   for ctp_i ∈ C which t_i is consumer do
17 |   |   |   |   prod_task ← get_producer(ctp_i);
18 |   |   |   |   if prod_task ∉ task_migration_list and ctp_i = PS then
19 |   |   |   |   |   CS_configuration(prod_task, t_i);
20 |   |   |   |   end
21 |   |   |   end
22 |   |   end
23 |   end
24 end
```

The activation of the proactive mode occurs periodically (lines 7–23). The trigger to activate the COMPLETE-CHECKUP algorithm in this mode is the *overhaul routine*, which calls the COMPLETE-CHECKUP at the end of 10 hyper-periods of the application. This number is a trade-off, a higher value reduces the COMPLETE-CHECKUP calls, delaying the time to take proactive actions; and smaller values increase the CPU usage of M_{PE}. Since the COMPLETE-CHECKUP function is the same for both modes, the proactive mode starts as in the reactive mode, ranking the tasks according to Equation (1). As the COMPLETE-CHECKUP was invoked by the *overhaul* routine and not by QUICK-CHECKUP algorithms, it is expected that the *task_rank[]* set be empty, i.e., all application tasks are fulfilling their constraints, and the algorithms jump to line 7. The proactive mode acts first on the computation (lines 7–13) and then on the communication (lines 14–23).

At the computation level, for each task t_i of the *application*, the algorithm verifies if there are more than one high computation task in t_i's core (tasks exceeding *comp_profile_TH*), and if the sum of the computation profile of all tasks sharing t_i's core exceeds *comp_profile_sum_TH* (line 9). If true, t_i is migrated proactively to an available processor (line 10), and the task identifier is added to the set *task_migration_list*. The goal is to proactively reduce the CPU sharing between high computation tasks. Also, as tasks may change their RT constraints dynamically, this action can prevent deadline misses when a given task increases the CPU utilization.

At the communication level, for each task t_i of the *application* that is not in the *task_migration_list*, the algorithm verifies if t_i's communication profile exceeds *comm_profile_TH*

or is higher than the computation profile (line 15). If true, t_i is a candidate to have its communication mode changed to CS. The loop in lines 16–21 sets CS for each CTP that has t_i as a consumer task. Line 18 verifies if the producer task is not in *task_migration_list* and if CTP's communication is assigned to the PS network. If this condition is true, a CS is established proactively for the CTP.

In summary, the proactive QoS actions try to reduce the CPU sharing between high computation tasks and to establish CS on tasks with a high communication profile.

6 EXPERIMENTAL RESULTS

The many-core used in the experiments has 64 PEs, with four 4×4 clusters. The MPN has one PS and a set of four CS sub-nets. RTL-level descriptions (VHDL (VHSIC (Very High Speed Integrated Circuit) Hardware Description Language) and SystemC) model the many-core system. Descriptions in C language model the software components (μkernel and applications). The RT applications include the following benchmarks: Video Object Plan Decoder (VOPD), matrix multiplication (MATRIX_MULTI), Multi-Window Display (MWD), Advanced Encryption Standard (AES), Dijkstra's Shortest Path First algorithm (DIJKSTRA), Dynamic Time Warping (DTW), video decoding (MPEG2), video decoding (MPEG4). Synthetic RT tasks run in parallel on the system with the purpose to induce computation and communication disturbances.

The experiments adopt the following parameters: $cpu_TH = 99\%$, $comp_profile_TH = 50\%$, $comp_profile_sum_TH = 75\%$, $comm_profile_TH = 10\%$, $deadline_TH = 1$, $latency_TH = 2$. The selection of the threshold values is made according to the expected applications' behavior. The chosen parameters enable a high CPU utilization for soft RT ($cpu_TH = 99\%$), with a high computation profile per S_{PE} ($comp_profile_sum_TH = 75\%$). The $deadline_TH = 1$ implies a very low tolerance to deadline misses. The value of $latency_TH = 2$ safely estimates a latency deadline because it can suppress random picks of latency while it keeps a high level of confidence in QoS at communication level [21]. We selected the value of $comm_profile_TH = 10\%$ empirically, achieving a good tradeoff between proactive actions and CS path diversity.

This Results section has four subsections. Section 6.1 evaluates single-objective QoS managers presented in the literature, highlighting the motivation for a comprehensive QoS method. As DAPE is a key feature to enable proactive actions, Section 6.2 evaluates its overhead. Next, Section 6.3 evaluates the contribution of this work presenting its benefits in the execution time and the reduction of deadline misses using a set of benchmarks. Finally, Section 6.4 presents the tradeoff system utilization versus deadline and latency misses.

6.1 Single Objective QoS Managers

The available proposals in the literature adopt different architectures, targeting only computation or communication self-adaptive techniques. This section provides a comparison at the computation and communication levels using values presented in related works, which are put into perspective with our results.

At the computation level, Petrucci et al. [18] propose a cluster scheduler that performs dynamic resource allocation (dynamic task mapping). The scheduler uses a QoS monitor that collects performance data at runtime. As our work, they assume thresholds to trigger adaptations during job scheduling, accepting up to of 5% of QoS violations (we assume 1% of deadline misses). Results show that the proposed QoS management meets 99.8% and 91% of the deadlines for memcached and web-search benchmarks, respectively (we meet 99.5% of deadlines for all benchmarks with severe interference). Delimitrou et al. [5] propose an online scheduler for large-scale data centers, which can also be employed for many-cores. Like our work, they also assume an unknown application workload. The main goal of the scheduler is to achieve application composability maximizing system utilization. The proposal degrades performance by only 4% and guarantees QoS for

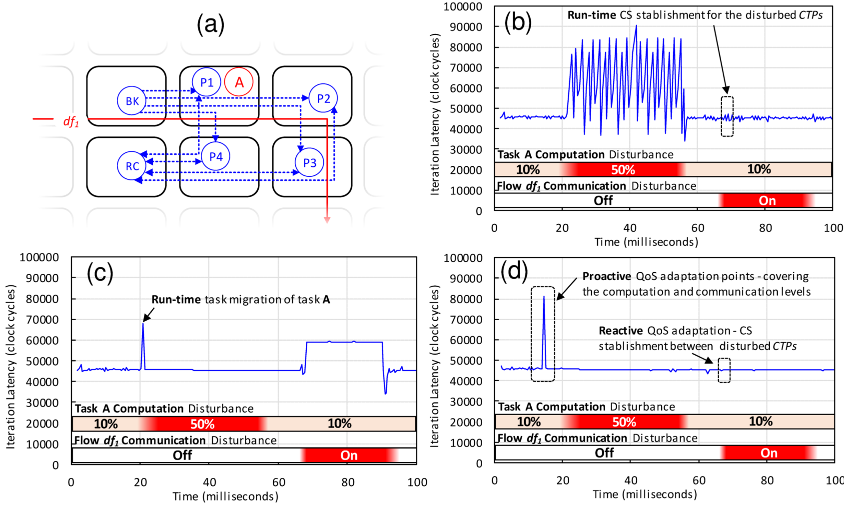


Fig. 6. Comparison with Refs [21] and [23]: (a) scenario setup; (b) iteration latency of Ref. [21] with focus on communication adaptation; (c) iteration latency of Ref. [23] with focus on computation adaptation; (d) iteration latency of the proposed work.

61% of workloads for high load. Our work does not present QoS degradation (see Figure 10) up to 70% of system utilization.

At the communication level, Abousamra et al. [2] propose a proactive CS allocation. The work focus is on the evaluation of L2 cache hit latency, which directly impacts in the execution time of the applications. The results show an execution time speedup of 12% in a scenario with disturbing traffic when compared to an optimal scenario. Our work also evaluates the execution time using scenarios with and without (optimal) interferences. Our experiments show that the execution time increases, compared to the optimal scenario, by 13.8% using only reactive actions and 2.4% assuming reactive and proactive actions. Mangano et al. [14] propose an adaptive QoS hardware that is supported by a performance monitoring to establish CS at runtime. This work presents results where the proposed techniques successfully fulfill all bandwidth requirements for two RT flows considering a reduction in the NoC clock frequency of 25%. The authors do not consider congested scenarios making it impossible to observe how bandwidth requirements are met in experiments with congested traffic. In a previous work [22], we showed that the success rate of CS establishment can reach values equal or near to 100% in experiments assuming large system sizes (up to 16×16 PEs), with each S_{PE} supporting and executing two tasks (a stressed scenario, with full system occupation), and assuming an MPN with four, six, and eight subnets. Joven et al. [10] propose a hardware/software infrastructure with runtime QoS support by establishing CS based on the packet header information. The proposed QoS policy was able to speed up the MPEG application execution in a scenario with interference by 83%–87%. In a similar experiment, the MPEG application of our work has an execution time speedup of 54% against the scenario with interference, presenting an execution time of only 0.4% above the baseline execution time.

These comparisons show that related works consider QoS fulfillment only partially. The experiment presented in Figure 6 compares the effect of communication QoS management [21], computation QoS management [23], and of comprehensive QoS management of this work. It shows the importance to cover both computation and communication, and also, how proactive actions can avoid future violations of QoS targets.

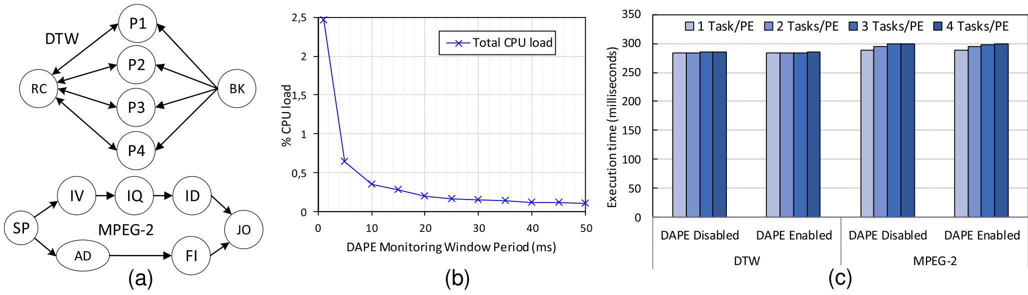


Fig. 7. (a) DTW and MPEG-2 application task graphs. (b) Rate of MPE CPU load with respect to the DAPE monitoring window period. (c) Overhead evaluation of the dynamic application profile extraction.

The experiment presented in this section uses the DTW application, a pattern recognition algorithm with its task graph described in Figure 7(a). Figure 6(a) presents the DTW mapping, with the hop distance between communicating tasks minimized. To evaluate the performance of computation and communication, we inserted a disturbing RT task A mapped at the same PE of task *P1*, and a disturbing communication flow called df_1, which crosses some flows of DTW's tasks. The interference of computation occurs from 20ms to 55ms, where task A increases its CPU utilization from 10% to 50%. The interference of communication occurs from 66ms to 90ms, which is the time required for df_1 flow to transmit its data stream. The y-axis of Figure 6(b)–(d) presents the latency of each DTW's iteration and the x-axis shows the simulation time. Bars below the x-axis represent when computation and communication disturbances occur.

Figure 6(b) presents the results obtained by using a single objective QoS manager [21], which targets self-adaptive QoS at the communication level by setting up CS connections. The work establishes CS connections at runtime for the disturbed $CTPs$, but it cannot mitigate the interference induced by the execution of task A.

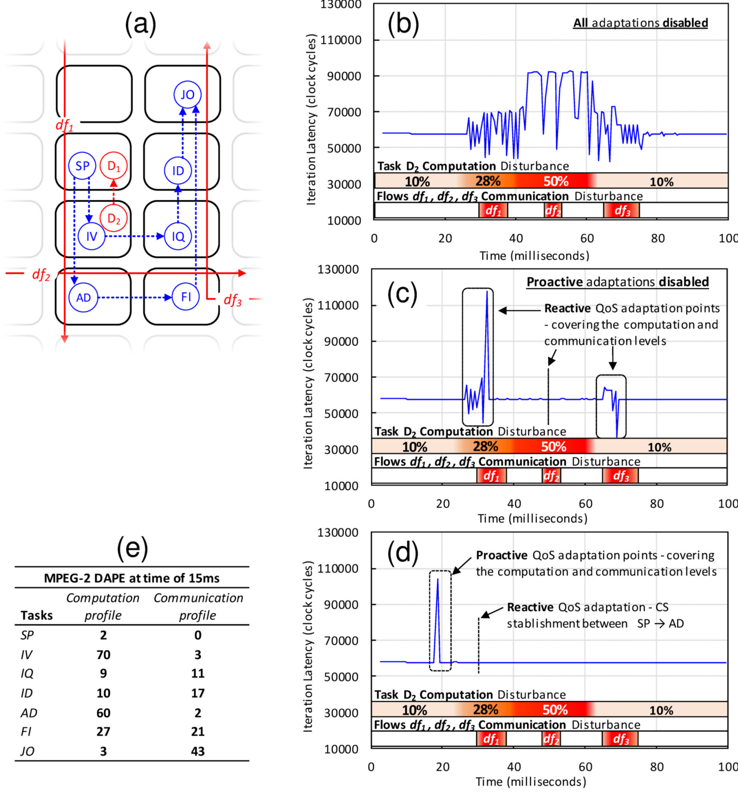
Figure 6(c) presents the results obtained by using a single objective QoS manager [23], which targets self-adaptive QoS at the computation level by using task migration. The manager counteracts the computation disturbance by migrating task A to another processor, but the communication interference remains.

Figure 6(d) shows the results of the proposed work, covering a comprehensive QoS support. The QoS management mitigates both computation and communication interference using proactive and reactive actions. The proactive action is triggered before the computation disturbance because there is enough time for DAPE to extract the application profile and to the QoS management decide the best proactive action. These results highlight that a QoS manager must act in both computation and communication together, in such a way to mitigate disturbances that may occur at runtime and are not possible to predict at design-time or to mitigate only using an effective task–mapping algorithm.

6.2 Dynamic Application Profile Extraction (DAPE) Overhead

The DAPE implementation part, within S_{PE}s, can penalize the tasks' execution time due to the monitoring process added on the task scheduler and the communication API. The μkernel adopts two actions to minimize this overhead: (*i*) the monitored profile transmission occurs preferably in idle periods of the S_{PE}; (*ii*) each S_{PE} uses a different counter to trigger the sending of the information, thus distributing the monitoring load.

The experiment presented in Figure 7 aims to evaluate the DAPE overhead. It uses two applications with its communicating graph detailed in Figure 7(a): DTW (recognizing 2,500 patterns), and MPEG-2 (decoding 500 frames/audio arrays). The functionality of the applications is not relevant



MPEG-2 DAPE at time of 15ms

Tasks	Computation profile	Communication profile
SP	2	0
IV	70	3
IQ	9	11
ID	10	17
AD	60	2
FI	27	21
JO	3	43

Fig. 8. Evaluation of the self-adaptive QoS management over the MPEG-2. (a) Application mapping. (b) No adaptation, deadline miss = 20.3%. (c) Only reactive adaptations, deadline miss = 2.3%. (d) Proactive and reactive adaptations, deadline miss = 0.5%. (e) DAPE for MPEG-2 at 15ms of simulation.

in this case, only the number of tasks per PE. More tasks per PE increase the interference due to the application profile extraction since there are more data to be collected and sent to the M_{PE}. Another important factor is the frequency that DAPE messages are transmitted from S_{PE}s to M_{PE}. We define a period of 10ms, which corresponds to a good tradeoff between communication volume and profiling update (i.e., the M_{PE} load). Figure 7(b) displays the M_{PE} CPU load (MIPS-CPU@100MHz), according to the frequency of the DAPE monitoring period. As shown in the figure, the M_{PE} reaches a total CPU load lower than 0.5% with a 10ms period, which does not impact the M_{PE} performance.

Figure 7(c) compares the overhead of the DAPE on application's execution time with a different number of tasks running at the same PE. All tasks in the "DAPE Enabled" scenarios have the DAPE monitoring enabled, with an expected overhead at each core increasing according to the number of tasks per PE. However, the results show that the impact on the application execution time is negligible. The worst-case overhead is achieved in the MPEG-2 scenario, with three tasks/PE, corresponding to an increase of 0.0024% in the application execution time.

6.3 Self-Adaptive QoS Evaluation

This section evaluates the proposed self-adaptive QoS management. Figure 8(a) shows the MPEG-2 task mapping, represented by blue circles. Tasks D_1 and D_2 share the CPU with tasks *SP* and *IV*, respectively. They belong to another RT application, exemplifying a disturbance at the

computation level. The red arrows denote disturbances of communication, other messages that interfere with the application flows. On the y-axis of Figure 8(b)–(d), the latency of one iteration of the MPEG-2 application is shown, as measured at the task $\mathcal{JO}$ (joint). The QoS constraint is 58,000 clock cycles, corresponding to the time to decode one audio/image frame of 576 bytes. Bars below the x-axis represent the CPU utilization of task D_2 (computation disturbance) and when the disturbing flows occur (communication disturbance).

The first evaluated scenario has all QoS adaptations disabled—Figure 8(b). When the CPU utilization of task D_2 increases, at 25 and 40ms, the latency increases due to the CPU sharing. In the same way, disturbing flows affect the latency. Note that when flow df_3 is active, the task D_2 presents a low CPU utilization and the latency also increases. This result shows that the communication disturbance also impacts the QoS constraint significantly.

The second evaluated scenario (Figure 8(c)) activates only reactive adaptations. When D_2 increases its CPU utilization from 10% to 28%, the total S_{PE} utilization reaches 98% (28% + 70% from task *IV*). The task migration is not immediately triggered because the CPU utilization remains below *cpu_TH*. Thus, task *IV* starts to generate deadline misses, and the QoS management decides to migrate task *IV* to a free processor at 31.4ms. Also, flow df_1 induces latency misses in the flows $SP{\rightarrow}IV$ and $SP{\rightarrow}AD$, making the QoS management to decide to establish CS for these flows at 30.3 and 32ms. There is no impact on the latency when D_2 increases its utilization to 50% because task *IV* was previously migrated. Flow df_2 induces latency misses in the flow $AD{\rightarrow}FI$, resulting in a new CS establishment at 49.2ms. As the CS establishment for one CTP affects in average 150 clock cycles of the application's latency, its effect is not perceptible in the graph. Finally, flow df_3 starts, disturbing three MPEG-2 flows: $IQ{\rightarrow}ID$, $FI{\rightarrow}JO$, and $ID{\rightarrow}JO$. The consequence is several latency misses, and CSs are reactively established for all penalized pairs at 66.5, 67.4, and 68.5ms.

The third evaluated scenario activates proactive and reactive adaptations—Figure 8(d). The first call to the COMPLETE-CHECKUP occurs at 15ms due to the *overhaul routine*, with the extracted profile presented in Figure 8(e). The COMPLETE-CHECKUP decides to migrate proactively task *IV* because it is sharing the CPU with D_2 and the sum of its computation profile is higher than *comp_profile_sum_TH* (at 17.6ms). Additionally, according to the obtained profile, tasks ID, FI, JO, and IQ have a high communication profile (higher than *comm_profile_TH*). Therefore, the QoS management proactively establishes CS for the flows: $IQ{\rightarrow}ID$ (18.5ms), $AD{\rightarrow}FI$ (18.6ms), $ID{\rightarrow}JO$ (19.1ms), $FI{\rightarrow}JO$ (19.7ms), and $IV{\rightarrow}IQ$ (23ms). This scenario also has a reactive QoS adaptation, which is a CS establishment between $SP{\rightarrow}AD$ at 32ms due to the disturbance caused by df_1. The QoS management did not establish CS's previously since the DAPE revealed that *AD* has a communication profile smaller than the *comm_profile_TH*.

Comparing deadline miss rates in all three scenarios, we observe a miss rate of 20.3% when QoS is disabled, 2.3% when only reactive adaption is used, and 0.5% with both proactive and reactive adaptions.

While Figure 8 shows the MPEG-2 case in detail, Figure 9 summarizes results for eight benchmarks. It compares execution time (a), and deadline miss rate (b), for: *(i)* baseline scenario (best performance); *(ii) DIST*—disturbances and no QoS mechanism; *(iii) REACT*—disturbances and only reactive QoS enabled; *(iv) P+R*—disturbances and both QoS mechanisms enabled (proactive + reactive). The disturbance setup consist of two tasks providing computation disturbance (randomly mapped within PEs running benchmark's tasks), and three disturbing communication flows. Initially, the experiment simulated the P+R scenario varying the disturbance setup for each benchmark, i.e., varying randomly the mapping of disturbance tasks and the position of the disturbance flows. The disturbance setup of each benchmark resulting in the largest number of deadline misses is the one selected for comparison purposes. Next, the selected disturbance setup for each benchmark was used to obtain the results of the baseline, DIST, and REACT scenarios, enabling

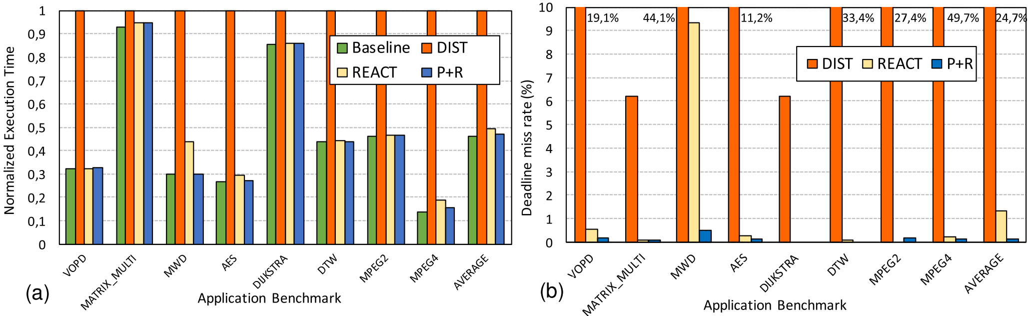


Fig. 9. Benchmark evaluation: (a) Execution Time; (b) Deadline miss rate. Number of tasks of each benchmark: VOPD—13; MATRIX_MULTI—14; MWD—12; AES—11 tasks; DIJKSTRA—7; DTW—6; MPEG2—7 tasks; MPEG4—12 tasks.

the comparison against the P+R scenario using the same interference. The mean of all scenarios corresponds to the last columns of the graphs (Figure 9), showing the benefits of P+R over REACT.

Compared to the baseline execution time, the *DIST* increases the execution time, on average, by 224.7% (severe disturbance). Applying QoS management, the execution time is restored close to the baseline: 13.8% for *REACT* and 2.4% for *P+R* above the baseline. Note that *REACT* exhibits a higher average execution time than *P+R*, highlighting the benefit of proactive actions. As Figure 9(b) shows, the deadline misses were reduced, on average, by 98% for *REACT*, and 99.5% for *P+R*, with a rate below 0.6% for all *P+R* benchmarks.

The adaptive actions require task migrations and CS establishment, inducing an overhead over the application iteration latency of 150 clock cycles for one CS establishment and 600 clock cycles for one task migration. The P+R scenario can present a slight increase in the execution time over the REACT in some benchmarks because the P+R scenario can require more adaptations than REACT, explaining the behavior observed in the execution time of VOPD and MPEG2 benchmarks, where the REACT execution time is lower than or equal to P+R. Despite the higher execution time for these two benchmarks, the number of deadline misses for P+R is lower in VOPD and slightly higher for MPEG2. Such phenomena occur because a deadline miss can be masked during the adaptation process, mainly for task migration, since the task needs to stop in one processor and restart on another one.

Those experiments demonstrate the synergy between proactive and reactive actions. The proactive adaptation reconfigures the system according to the application profile extraction, preventing/minimizing future deadline misses due to interference. Reactive adaptation deals with unpredictable events to restore the applications' performance.

6.4 Self-Adaptive QoS Tradeoff

Section 6.1 showed that the QoS manager must act in both computation and communication conjointly. Section 6.3 presented the effectiveness of the proposed method to provide QoS for a set of benchmarks in the presence of disturbing events. This last evaluation stresses the proposed method, with all PEs executing tasks with QoS constraints, and increasing the CPU utilization gradually. This evaluation adopts as target three synthetic RT applications, with different profiles: (*i*) *COMP*—computation intensive (66% comp., 10% comm.); (*ii*) *COMM*—communication intensive (32% comp., 61% comm.); (*iii*) *HYB*—hybrid, a mix of computation and communication profile (45% comp., 30% comm.).

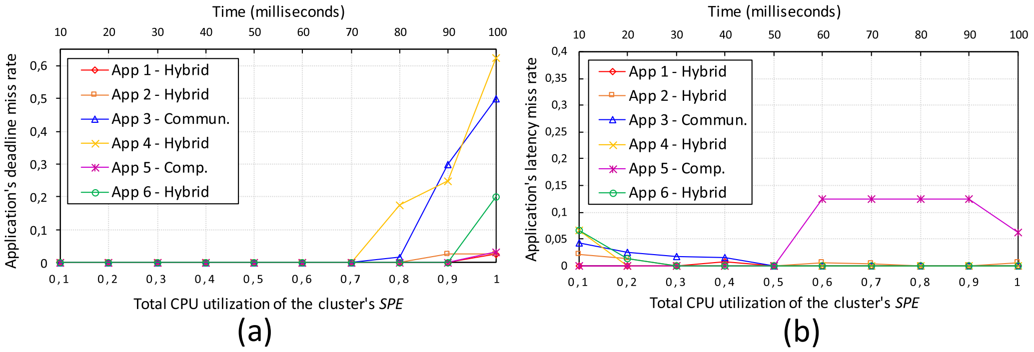


Fig. 10. QoS provisioning tradeoff: (a) Deadline miss rate; (b) Latency miss rate.

The simulated workload corresponds to six applications executing simultaneously: 1 *COMM*, 1 *COMP*, and 4 *HYB*. The applications were randomly mapped in a cluster with 16 PEs (1 M_{PE} and 15 S_{PE}s). The S_{PE}s were configured to run two tasks concurrently. To use all S_{PE}s of the cluster, all S_{PE} received two tasks. With such configuration, task migration is disabled due to the full system usage. All applications start their execution at the beginning of the simulation, with a warm-up period of 10ms. After the warm-up period, each S_{PE} has 10% of CPU utilization (5% from each mapped task). As the simulation advances, all tasks increase their CPU utilization steadily. The utilization increases by 5% for each task at each 10ms, resulting in a total CPU utilization increase of 10% at each 10ms (two tasks per CPU).

Figure 10(a) presents the deadline miss rate for this experiment (*y*-axis: percentage of deadline misses, *x*-axis: CPU utilization per S_{PE}). Results correspond to the average of several runs for each CPU utilization. The deadline miss rate remains 0 up to 70% of CPU utilization. After 70% of CPU utilization, all applications start to miss deadlines. Due to the systems unpredictable behavior with a higher CPU utilization, some applications miss more deadlines than others (*Apps 3, 4, 6*). As task migration is not possible due the full system occupation, this result showed the effectiveness of the task scheduler to ensure QoS at the computation level.

Figure 10(b) presents the latency miss rate. All applications start their execution communicating through the PS NoC. As the simulation advances, the QoS management aided by DAPE identifies the applications' profile and sets CS to the *COMM* and *HYB* applications, reducing the latency miss to less than to 0.5%. The exception is the *COMP* application (*App 5*), which continues using the PS NoC because it does not satisfy the requirements for proactive CS. With the increase in the CPU utilization, this application receives one latency miss at each 10ms, resulting in a latency miss rate of 12.5%. As *App 5* is computation intensive, this latency miss does not impact the deadline misses, as can be observed in Figure 10(a), not justifying the establishment of CS.

This experiment enabled to observe the QoS provisioning tradeoff at the computation and communication levels in a stressed scenario. The computation QoS starts to be affected after 70% of CPU utilization. The method ensures communication QoS for all communication sensitive applications by establishing CS at runtime. The adoption of an MPN provides sufficient CS paths even in a cluster with all S_{PE}s running the allowed number of tasks.

7 CONCLUSION

This work proposes dynamic profiling and self-adaptive QoS management for soft real-time applications. A runtime application learning profiling (DAPE) technique allows the QoS management system to take proactive actions, and when necessary, react to cope with the interference induced

by the dynamic workload. The low overhead of DAPE demonstrates that runtime techniques can be used to characterize applications. Our results lead to two conclusions. First, mechanisms for managing both communication and computing are essential for overall QoS management. Second, proactive techniques can avoid future deadline and latency misses.

While we have demonstrated the benefits of the proposed QoS management in a specific setting, there is little in the NoC, the PEs, and the middleware that we require for our techniques to work. Hence, we expect similar benefits on very different platforms, which, however, have to be demonstrated in future work.

REFERENCES

[1] Gean Abich, Marcelo Mandelli, Felipe R. Rosa, Fernando G. Moraes, Luciano Ost, and Ricardo Reis. 2016. Extending freeRTOS to support dynamic and distributed mapping in multiprocessor systems. In *ICECS*. IEEE, 712–715. DOI: https://doi.org/10.1109/ICECS.2016.7841301

[2] Ahmed Abousamra, Alex K. Jones, and Rami G. Melhem. 2013. Proactive circuit allocation in multiplane NoCs. In *DAC*. ACM, 35:1–35:10. DOI: https://doi.org/10.1145/2463209.2488778

[3] Brent Bohnenstiehl, Aaron Stillmaker, Jon Pimentel, Timothy Andreas, Bin Liu, Anh Tran, Emmanuel Adeagbo, and Bevan Baas. 2017. KiloCore: A 32-nm 1000-processor computational array. *J. Solid-State Circuits* 52, 4 (2017), 891–902. DOI: https://doi.org/10.1109/JSSC.2016.2638459

[4] Guilherme Castilhos, Marcelo Mandelli, Guilherme Madalozzo, and Fernando G. Moraes. 2013. Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes. In *ISVLSI*. IEEE, 153–158. DOI: https://doi.org/10.1109/ISVLSI.2013.6654651

[5] Christina Delimitrou and Christos Kozyrakis. 2013. QoS-aware scheduling in heterogeneous datacenters with paragon. *ACM Trans. Comput. Syst.* 31, 4 (Dec. 2013), 12:1–12:34. DOI: https://doi.org/10.1145/2556583

[6] Nikil D. Dutt, Fadi J. Kurdahi, Rolf Ernst, and Andreas Herkersdorf. 2016. Conquering MPSoC complexity with principles of a self-aware information processing factory. In *CODES+ISSS*. ACM, 37:1–37:4. DOI: https://doi.org/10.1145/2968456.2973275

[7] Kunal P. Ganeshpure and Sandip Kundu. 2013. On runtime task graph extraction in MPSoC. In *ISVLSI*. IEEE, 171–176. DOI: https://doi.org/10.1109/ISVLSI.2013.6654654

[8] Henry Hoffmann, Martina Maggio, Marco D. Santambrogio, Alberto Leva, and Anant Agarwal. 2013. A generalized software framework for accurate and efficient management of performance goals. In *EMSOFT*. IEEE, 19:1–19:10. DOI: https://doi.org/10.1109/EMSOFT.2013.6658597

[9] Axel Jantsch, Nikil D. Dutt, and Amir M. Rahmani. 2017. Self-awareness in systems on chip—A survey. *IEEE Design & Test* 34, 6 (2017), 8–26. DOI: https://doi.org/10.1109/MDAT.2017.2757143

[10] Jaume Joven, Andrea Marongiu, Federico Angiolini, Luca Benini, and Giovanni De Micheli. 2013. An integrated, programming model-driven framework for NoC-QoS support in cluster-based embedded many-cores. *Parallel Comput.* 39, 10 (2013), 549–566. DOI: https://doi.org/10.1016/j.parco.2013.06.002

[11] Hanwoong Jung, Chanhee Lee, Shin-Haeng Kang, Sungchan Kim, Hyunok Oh, and Soonhoi Ha. 2014. Dynamic behavior specification and dynamic mapping for real-time embedded systems: HOPE approach. *ACM Trans. Embedded Comput. Syst.* 13, 4s (2014), 135:1–135:26. DOI: https://doi.org/10.1145/2584658

[12] Peter R. Lewis, Marco Platzner, Bernhard Rinner, Jim Trresen, and Xin Yao. 2016. *Self-aware Computing Systems: An Engineering Approach* (1st ed.). Springer.

[13] N. Loubet et al. 2017. Stacked nanosheet gate-all-around transistor to enable scaling beyond FinFET. In *Symposium on VLSI Technology*. IEEE, 230–231. DOI: https://doi.org/10.23919/VLSIT.2017.7998183

[14] Daniele Mangano and Giovanni Strano. 2010. Enabling dynamic and programmable QoS in SoCs. In *NoCArc*. ACM, 17–22. DOI: https://doi.org/10.1145/1921249.1921255

[15] Aline Mello, Leonel Tedesco, Ney Calazans, and Fernando Moraes. 2006. Evaluation of current QoS mechanisms in networks on chip. In *SOC*. IEEE, 1–4. DOI: https://doi.org/10.1109/ISSOC.2006.321981

[16] Luciano Ost et al. 2013. Power-aware dynamic mapping heuristics for NoC-based MPSoCs using a unified model-based approach. *ACM Trans. Embedded Comput. Syst.* 12, 3 (2013), 75:1–75:22. DOI: https://doi.org/10.1145/2442116.2442125

[17] Sangsoo Park. 2014. Task-I/O Co-scheduling for pfair real-time scheduler in embedded multi-core systems. In *EUC*. IEEE, 46–51. DOI: https://doi.org/10.1109/EUC.2014.16

[18] Vinicius Petrucci, Michael A. Laurenzano, John Doherty, Yunqi Zhang, Daniel Mossé, Jason Mars, and Lingjia Tang. 2015. Octopus-Man: QoS-driven task management for heterogeneous multicores in warehouse-scale computers. In *HPCA*. IEEE, 246–258. DOI: https://doi.org/10.1109/HPCA.2015.7056037

[19] GAPH PUCRS. 2018. Hermes Multiprocessor System-on-Chip. Retrieved from http://www.inf.pucrs.br/hemps/index.html.

[20] Wei Quan and Andy D. Pimentel. 2016. A hierarchical run-time adaptive resource allocation framework for large-scale MPSoC systems. *Design Autom. Emb. Sys.* 20, 4 (2016), 311–339. DOI: https://doi.org/10.1007/s10617-016-9179-z

[21] Marcelo Ruaro, Everton Carara, and Fernando G. Moraes. 2015. Runtime adaptive circuit switching and flow priority in NoC-based MPSoCs. *IEEE Trans. VLSI Syst.* 23, 6 (2015), 1077–1088. DOI: https://doi.org/10.1109/TVLSI.2014.2331135

[22] Marcelo Ruaro, Henrique Medina, Alexandre Amory, and Fernando G. Moraes. 2018. Software-defined networking architecture for NoC-based many-cores. In *ISCAS*. IEEE, 385–390. DOI: https://doi.org/10.1145/2881025.2889474

[23] Marcelo Ruaro and Fernando G. Moraes. 2016. Dynamic real-time scheduler for large-scale MPSoCs. In *GLSVLSI*. ACM, 341–346. DOI: https://doi.org/10.1145/2902961.2903027

[24] Marcelo Ruaro and Fernando G. Moraes. 2017. Demystifying the cost of task migration in distributed memory many-core systems. In *ISCAS*. IEEE, 1–4. DOI: https://doi.org/10.1109/ISCAS.2017.8050257

[25] Amit Kumar Singh, Piotr Dziurzanski, Hashan Roshantha Mendis, and Leandro Soares Indrusiak. 2017. A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems. *ACM Comput. Surv.* 50, 2 (April 2017), 24:1–24:40. DOI: https://doi.org/10.1145/3057267

[26] Markus Winter and Gerhard P. Fettweis. 2011. Guaranteed service virtual channel allocation in NoCs for run-time task scheduling. In *DATE*. IEEE, 419–424. DOI: https://doi.org/10.1109/DATE.2011.5763073

Received December 2017; revised February 2019; accepted April 2019