Modular and Distributed Management of Many-Core SoCs

MARCELO RUARO and ANDERSON SANT'ANA, Pontifical Catholic University of Rio Grande do Sul
AXEL JANTSCH, TU Wien
FERNANDO GEHM MORAES, Pontifical Catholic University of Rio Grande do Sul

Many-Core Systems-on-Chip increasingly require Dynamic Multi-objective Management (DMOM) of resources. DMOM uses different management components for objectives and resources to implement comprehensive and self-adaptive system resource management. DMOMs are challenging because they require a scalable and well-organized framework to make each component modular, allowing it to be instantiated or redesigned with a limited impact on other components.

This work evaluates two state-of-the-art distributed management paradigms and, motivated by their drawbacks, proposes a new one called *Management Application (MA)*, along with a DMOM framework based on MA. MA is a distributed application, specific for management, where each task implements a management role. This paradigm favors scalability and modularity because the management design assumes different and parallel modules, decoupled from the OS.

An experiment with a task mapping case study shows that MA reduces the overhead of management resources (−61.5%), latency (−66%), and communication volume (−96%) compared to state-of-the-art per-application management. Compared to cluster-based management (CBM) implemented directly as part of the OS, MA is similar in resources and communication volume, increasing only the mapping latency (+16%). Results targeting a complete DMOM control loop addressing up to three different objectives show the scalability regarding system size and adaptation frequency compared to CBM, presenting an overall management latency reduction of 17.2% and an overall monitoring messages' latency reduction of 90.2%.

CCS Concepts: • **Computer systems organization → Multicore architectures**; • **Hardware → On-chip resource management;**

Additional Key Words and Phrases: Many-core, distributed resource management, System-on-Chip (SoC)

ACM Reference format:
Marcelo Ruaro, Anderson Sant'Ana, Axel Jantsch, and Fernando Gehm Moraes. 2021. Modular and Distributed Management of Many-Core SoCs. *ACM Trans. Comput. Syst.* 38, 1-2, Article 1 (July 2021), 16 pages.
https://doi.org/10.1145/3458511

This study was financed in part by the Coordenacao de Aperfeicoamento de Pessoal de Nivel Superior – Brasil (CAPES) – Finance Code 001. Fernando Gehm Moraes is supported by FAPERGS (17/2551-0001196-1) and CNPq (309605/2020-2), Brazilian funding agencies.

Authors' addresses: M. Ruaro, A. Sant'Ana, and F. Moraes, School of Technology - Pontifical Catholic University of Rio Grande do Sul (PUCRS), Av. Ipiranga, 6681, 90619-900, Porto Alegre, Brazil; emails: marcelo.ruaro@acad.pucrs.br, anderson.santana@edu.pucrs.br, fernando.moraes@pucrs.br; A. Jantsch, Institute of Computer Technology (ICT), TU Wien, Gusshausstrasse 27–29 / 384, 1040 Vienna, Austria; email: axel.jantsch@tuwien.ac.at.

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.
© 2021 Association for Computing Machinery.

0734-2071/2021/07-ART1 $15.00
https://doi.org/10.1145/3458511

1 INTRODUCTION

Technology advances allow increasing semiconductor integration levels enabling systems-on-chip (SoCs) with dozens to hundreds of cores, herein named Many-core System-on-Chip (MCSoC). MCSoCs are state-of-the-art regarding processing power because they reach a high computational parallelism level in a small silicon area. Such processing power is fundamental to leverage real-time applications, which can process a huge amount of data in parallel, such as artificial intelligence, autonomous systems, health imaging processing, IoT, and cyber-physical systems.

For small systems with a homogeneous and known application set, the system's management can be performed statically, at design-time. This static management corresponds to select the cores where applications will be mapped, establishing the appropriate communication level bandwidth among tasks, and controlling the chip voltage and frequency. Larger systems, as MCSoCs, designed to support applications with different profiles, workloads, and constraints, dynamic management is essential.

Therefore, MCSoCs are evolving from fixed objective resource management to dynamic multi-objective resource management (**DMOM**) [11]. Simultaneous and self-adaptive management of conflicting objectives such as power, performance, and user experience [23] characterizes a DMOM. While this is a considerable challenge, there are recent works in the literature addressing DMOM [8, 10, 20, 23, 26]. The DMOM system can be seen and implemented as an Observe-Decide-Act (ODA) control loop, requiring different management modules, each in charge of one specific role and addressing different system resource categories. Naturally, any DMOM should adopt a scalable and modular architecture.

Resources in an MCSoC comprise three levels: computation (CPU, memory), communication (NoC, network interface), and physical (as, voltage and frequency). A comprehensive DMOM implements the ODA phases addressing all three categories. These categories received considerable attention from researchers, and the literature proposes solutions for *individual* self-adaptive management. Table 1 summarizes contributions for each level.

The design of a DMOM system becomes more challenging with the increase in the MCSoCs' size [3]. The increase in the number of cores brings undesired features, as higher degrees of unpredictability, process variation, and the utilization wall [21]. As a starting point, it is mandatory to consider a scalable and flexible system organization where each component has its role well-defined and decoupled from other components to promote scalability and modularity.

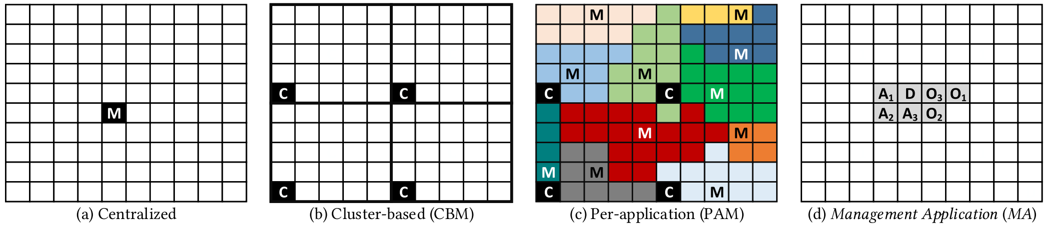
Related works addressing *distributed* DMOM systems are divided into Cluster-Based Management (**CBM**), with one manager per cluster of cores, and Per-Application Management (**PAM**), with one manager dynamically created for each running application. This work has two main *goals*. First, evaluate existing DMOM approaches, identifying their drawbacks related to parallelism, communication volume, management flexibility, and management latency. The second one is to propose a new DMOM approach that fulfills the weakness of existing methods. We call the proposed management paradigm *Management Application* (**MA**). Its essence consists of exploiting the parallel and flexible design environment used for user's applications to implement system resource management services at the user-level, decoupled from the Operating System (OS), where each MA task is organized according to the ODA control loop. Additionally, we present a detailed framework structure, describing the DMOM implementation based on the MA paradigm.

The *main contributions* of this work are as follows:

(i) Proposition of the MA paradigm for MCSoC management;
(ii) DMOM framework based on the MA paradigm called *DMOM-MA*.

Table 1. Self-adaptive techniques state-of-the-art

Level	Resources	Observation	Decision/Actuation
Computation [7, 24]	CPU, memory	Task deadline Application heartbeats	Task migration and task scheduling
Communication [17, 19]	NoC, NI	Packet latency Flow throughput	Flow priority, virtual-channel circuit-switching
Physical [12, 29]	Voltage, frequency	Power Energy	Dynamic DVFS adaptation, power/clock gating



(a) Centralized (b) Cluster-based (CBM) (c) Per-application (PAM) (d) *Management Application* (*MA*)

Fig. 1. Resource management organizations. **M** - Manager core, **C** - Cluster manager core, $\mathbf{O}_{1,2,3}$ - Observation tasks, $\mathbf{A}_{1,2,3}$ - Adaptation tasks, **D** - Decision task.

The management decision quality is a topic orthogonal to the management organization scheme and thus outside the scope of this work. We focus on the DMOM organization and the interaction between its components. Thus, ODA tasks are synthetically implemented, allowing a fair comparison among management techniques.

The rest of this paper is organized as follows. Section 2 presents related work on resource management organization and the motivations to the MA proposition. Section 3 details the MA paradigm. Section 4 presents DMOM-MA framework. Section 5 evaluates DMOM-MA in comparison with other distributed state-of-the-art paradigms. Section 6 concludes this work.

2 RESOURCE MANAGEMENT ORGANIZATIONS

On-chip dynamic resource management has been studied extensively, as witnessed by recent comprehensive surveys of this area [11], but we focus more narrowly on system-level resource management schemes.

Figure 1(a-c) details existing resource management organizations: (*i*) centralized; (*ii*) CBM (Cluster-based Management); (*iii*) PAM (Per-application Management). Centralized approaches are feasible for a small number of cores but not suitable for large-scale systems with dozens of cores since the manager core easily becomes a bottleneck.

CBM [1, 5, 6, 10, 13, 27, 28] stands out as the most widespread organization in the literature. CBM divides the management into clusters, each one managed by a cluster manager (*C*) [1, 13], implemented in a dedicated core. Authors in [5, 27, 28] assume two hierarchical management levels with slave cores that execute user's tasks and a system core assigned to the management of slave clusters. Works [1, 10, 13] propose three hierarchical levels, with slave cores, cluster manager, and one global manager. Such approaches are scalable since they divide the management load among clusters. However, we identify the following drawbacks related to flexibility and resource utilization: (*i*) the managers' position and cluster bounds are defined at design-time, decreasing the management flexibility since management services may require different cluster sizes; (*ii*) as the management core is only used for management functions, its idle time is not used and could

be exploited to run low priority tasks; (*iii*) all services implemented in a manager core are tightly coupled, making maintenance hard, and the addition of new services can easily interfere with existing ones.

PAM [9, 25] assigns one manager core (*M*) per application. The manager is dynamically created when an application is loaded. The *M* core is responsible for supervising the application execution and distributing its workload at runtime. The works assume a cluster manager core (*C*), which is design-time defined and keeps the resource usage records of a cluster of cores. The works also assume that each application task runs on one single core. Overall results show that the proposed scheme improves the management execution time from 17.5% up to 30% compared to related works [2, 9].

PAM is fully distributed and conceptually more flexible than CBM since each manager is in charge of one application. We identify the following drawbacks of this approach: (*i*) it is most suitable for systems that assume applications with many tasks [25], which keeps the management overhead per task low (i.e., applications having a small number of tasks will lead to an excessive amount of resources dedicated to management); (*ii*) the dynamic creation of new managers imposes complex synchronization protocols to take systemic actions since managers are more focused on the applications' goals than system goals.

Figure 1(d) presents the MA approach proposed in this work. Each gray tile in the Figure is a management task, with a specific role. *Observation* tasks (tasks $O_{1,2,3}$) gather and abstract raw monitoring data. They also support an interface for users' commands. *Decision* task(s) (D) make adaptation decisions based on the observed scenario using heuristics or learning techniques. *Adaptation* tasks ($A_{1,2,3}$), manage adaptation protocols, such as DVFS (dynamic voltage and frequency scaling) and task migration. Figure 1(d) shows how the system designer can split the DMOM roles into several tasks. These tasks can be mapped at different positions of the system, and the number of tasks can also be defined at runtime according to the workload requirements.

Limitations of MA include: (*i*) its tasks have the maximum scheduling priority over other user's tasks and, consequently, cannot share a CPU with real-time (RT) tasks; (*ii*) the system designer needs to carefully choose when two or more MA tasks can share the same CPU since conflicting tasks can reduce the overall management performance.

Advantages of the MA paradigm that motivate this work include:

- MA's tasks do not require its execution on a dedicated core, as in CBM or PAM;
- MA's tasks have no fixed core allocation. If a core exceeds its TDP or becomes faulty, an MA task can migrate to another core with the same flexibility as user tasks;
- MA's tasks can be created or killed at runtime according to the management demand;
- Improved programmability since the management algorithms become decoupled from the OS level, with access to low-level resource abstracted through APIs. This also improves the developing process since one design team can work at the management level by implementing the MA's tasks. Another team can work on the architectural level by implementing APIs for MA tasks and their respective OS support.

3 MANAGEMENT APPLICATION PARADIGM

This Section details the MA paradigm. As Figure 2 illustrates, the MCSoC organization contains three levels: hardware, OS, and applications. Based on this hierarchy, we introduce the *Management Application* as a new type of application class focused on resource management.

Hardware: is orthogonal to our proposal. Figure 2 depicts a typical core architecture, with a scratchpad paged memory to store the OS and tasks, a CPU, a Network Interface (NI), and multiple physical NoCs, with one packet switching (PS) router for Best-effort and management flows, and

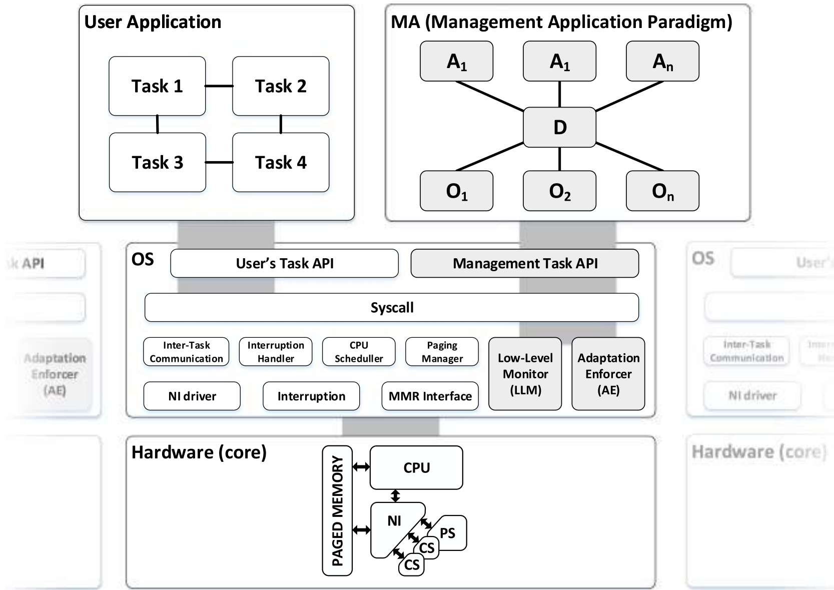


Fig. 2. Overview of the MA paradigm. An MCSoC architecture has three levels: hardware, OS, and application. The MA is implemented at the application level, requiring two modules inside the OS, which allows the MA to access low-level system resources through the management task API.

a set of circuit switching (CS) routers used for real-time flows. Memory-mapped registers (MMR) expose hardware functions to the OS, such as monitoring power consumption, the configuration of interrupt masks, scheduler interrupt counters. MMR also allows the OS to configure a hardware module, like changing the cores' voltage and frequency cores, and sending a packet using a given CS sub-network.

OS: the OS should provide the following components to support the MA paradigm (Section 4.1):

(i) *Low-level Monitor* (**LLM**) – a monitor that periodically pulls data from the hardware, user tasks, and OS modules, sending them to the MA tasks. LLM does not execute any complex computation.
(ii) *Adaptation Enforcer* (**AE**) – physically applies the adaptations coming from the MA tasks. Some examples include the control of the core voltage and frequency and task migration.
(iii) *Management Task API* (**MA API**)– provides a set of customizable management functions that allows the MA's tasks to communicate with each other and send commands to AE.

Only authorized tasks access the management task API. The OS assigns a unique ID for each application and task (user or MA tasks). The first application to be mapped is the MA application, which receives the ID 0. Thus, the OS only allows access to the management task API from tasks belonging to the application ID 0.

Application: the *application* level does not require changes. Each application (either management or user) is modeled by a set of n tasks, described through a Communicating Task Graph (CTG).

The MA implementation in the userspace brings advantages related to modularity and portability. As shown in Figure 2, the OS receives a small number of modules (LLM, AE, MA API). Therefore, the management technique may be applied to different systems, with slight changes in

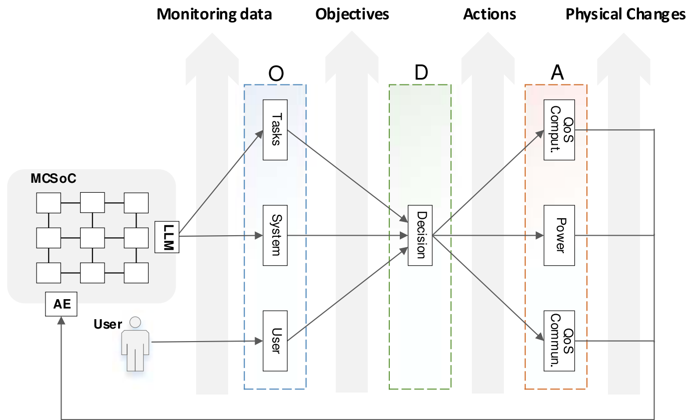


Fig. 3. DMOM-MA Framework implementing the ODA control loop.

the original OS. Another advantage is that the OS of all cores is the same, without specializing a given core, as in the CBM approach.

As the management techniques run in parallel with user's applications, they should be designed to create a minimum overhead. Such a feature differentiates the management in MCSoCs from the management in other systems, as in server systems, since MCSoCs have resources' constraints, as power, performance, and area.

4 DMOM-MA FRAMEWORK

Figure 2 presents the MA components along with a typical MCSoC organization. This Section details the MA-based DMOM framework (**DMOM-MA**). The framework covers all phases of the ODA control loop and addresses three categories of resource management: communication, computation, and physical.

Figure 3 depicts the framework model. In summary, the LLM running at the OS of each core generates messages periodically. Observation tasks handle these messages, jointly with user commands. The Observation tasks know the system and applications' constraints, and, based on the monitored data, can convert raw monitoring data to objectives. Objectives are sent periodically to the Decision task, which converts the objectives into actions [23], implementing the algorithms required to detect when and what resource needs adaptation. If necessary, the Decision task calls an Adaptation task, which implements the protocols to dynamically change the resources by interacting with the Adaptation Enforcer at the OS level.

4.1 OS Support

This Subsection presents the support required by the OS to implement the DMOM-MA framework, detailing the Low-Level Monitor, Adaptation Enforcer, and the management API functions.

4.1.1 Low-Level Monitor–LLM. The LLM sends raw monitored data to the Observation Tasks periodically. The LLM supports the monitoring of the following parameters:

- Task deadline: the OS scheduler detects when a given soft real-time task misses a deadline (the LLM receives the task's deadlines during the application loading).
- Task communication latency: tasks communicate by exchanging messages through the NoC. Based on a timestamp embedded in the packets, the OS detects when a given packet exceeds a threshold latency (the LLM receives the threshold latency during the application loading).

Table 2. API functions of DMOM-MA framework

API Call	Description
Generic	
*ReceiveMsg(unsigned int * msg, int size)*	Wait for a message sent either by the LLM or a MA task
*SendToMA(unsigned int * msg, int size, int taskMA)*	Sends a message to another MA task
DVFS	
ApplyDVFS(int v, int f, int dest_core)	Send a message to AE configuring a new VF-pair
Task Migration / Mapping	
MigrateTask(int taskID, int dest_core)	Send a message to AE ordering to start a task migration
LoadTask(int taskID, int dest_core)	Send a message to an off-chip application loader, ordering to load a given task to a core
Communication Switching	
SetupCS(int subnet, int dest_core)	Send a message to AE to configure CS in a given CS router and subnet
SetSwitching (ctp = {prod, cons}, swtc = PS, CS)	Send a message to AE, ordering a communicating task pair (*ctp*) to assume the new switching mode of communication

- Task profile: by combining the scheduling and task communication information, the LLM traces the task profile, classifying it as computation-intensive, communication-intensive, or hybrid [17].
- Application heartbeat: typical applications running in MCSoCs are cyclic, such as video processing. The endpoint task of the CTG emits a pulse to identify the heartbeat of the application [7], which characterizes the application hyper-period. The LLM records such pulse and uses it to identify deadline misses at the application level.
- Power: the LLM monitors the power consumption and estimates the core's temperature by observing the CPU activity (power per instruction), the number of memory accesses, and the number of flits transmitted by the router [10].
- Core's utilization: the OS scheduler monitors the percentage of time that the core is in use.

4.1.2 Adaptation Enforcer–AE. The AE receives and enforces the following orders sent by the Adaptation tasks:

- Task Migration: AE starts a task migration protocol based on task recreation [18], which moves a task to another core, including its code and data memory.
- Task Mapping: AE allocates a new task to its core, initializing the task control block (TCB) with parameters, such as task execution time, period, deadline, and communication latency.
- CS setup: the AE configures the task TCB to start sending and receiving messages using CS.
- PS setup: the AE configures the task TCB to send/receive messages using PS.
- DVFS: AE sets a memory-mapped register with the voltage-frequency pair embedded into the message. The MMR configuration is acknowledged by the DVFS hardware that changes the voltage and frequency of the core.

4.1.3 Management Task API. Table 2 details the Management Task API. The API functions are called by MA tasks to receive messages from the LLM and AE, to communicate with each other, and to send commands to the AE.

4.2 Exchanged Messages

Table 3 details the messages exchanged between the LLM and the Observation tasks, the Observation tasks and the Decision task, and the Decision task and the Adaptation tasks. As illustrated by the vertical gray arrows in Figure 3 and detailed in Table 3, the exchanged messages are divided into three classes: monitoring data, objectives, and actions.

Table 3. Message types of the DMOM-MA framework

Monitoring data: from LLM to Observation Tasks	
TASK_MONITOR_DATA	Contains the user's task constraints and real-time status: {task profile, task deadlines miss, task latency miss, application heart-beat}
SYS_MONITOR_DATA	Contains the system status: {power and core utilization}
USER_MONITOR_DATA	Contains the user request: {high performance, battery saving}
Objectives: from Observation Tasks to Decision Task	
SYS_OBJECTIVE	Report a system objective: {temp. = normal, high; core utilization = low, normal, high}
TASK_OBJECTIVE	Report a user's task objective: {communication QoS, computation QoS}
USER_OBJECTIVE	Reports a user objective: {high performance, battery saving}
Actuation: from Decision Task to Adaptation Tasks	
TASK_MAPPING_REQUEST	Requests a task mapping adaptation
TASK_MIGRATION_REQUEST	Requests a task migration adaptation
CIRCUIT_SWITCHING_REQUEST	Requests a CS establishment between a communicating task pair (*ctp*)
PACKET_SWITCHING_REQUEST	Requests a *ctp* to return to PS communication mode
POWER_SETUP_REQUEST	Requests the power configuration of a given core

Monitoring messages are generated from the LLM to Observation tasks containing raw monitoring data. As the LLM is implemented at the OS level, it does not use any Management Task API functions.

The Observation tasks receive the LLM messages calling the *ReceiveMsg*() function and classify the monitoring messages, converting them into *objectives messages* (Table 3). The objectives are sent from the Observation to the Decision tasks using the *SendToMA*() function.

The Decision task receives the Observation messages calling the *ReceiveMsg*(), and can generate an *action message*. Action messages are sent using API calls as DVFS, Task Migration/Mapping and Communication Switching of Table 3. The adaptation tasks and AE communicate during the execution of a given adaptive protocol. Therefore, there are also messages involved in such protocols that physically change the system. Such messages are out of the scope of this work being part of the adaptive protocols: DVFS [10], dynamic communication switching [4, 16], and task migration and mapping [17].

4.3 Observation Tasks

The Observation tasks' role is to convert raw monitoring data to objectives since they know the system and application constraints. The Observation tasks generate *objective* messages periodically to the Decision task, even when there is no constraint violation. These "normal" messages are also important to endorse a past decision of the Decision task. Three classes of Observation tasks are adopted: system, user's task, and user's commands.

System tasks transmit system performance figures, such as power, temperature, and CPU utilization. Based on the power value, it is possible to detect when a given core reaches its thermal safe power [12], classifying the temperature as normal or high (Table 3). Temperature values can trigger a DVFS adaptation. Based on the core utilization, the system task classifies the utilization of a core as low, normal, and high. The core utilization is used during task mapping and task migration decisions, helping to distribute the system's load.

User task observes the profile and performance of real-time tasks, by receiving deadlines/latency misses, heartbeats, and profiling [7, 17]. The User task is concerned with the runtime QoS fulfillment of real-time tasks (computation and communication). The task deadline miss and heartbeats information can lead to task migrations, moving tasks of the affected application to cores with more resources. A task latency miss can lead to a dynamic CS establishment. The task profile is

used to generate objectives according to the profile of the task proactively. Computation-intensive tasks can be early migrated to free cores. Communication-intensive tasks can early receive a CS connection [17].

User commands handle user messages that are generated by external user commands. The user can change the objective of the system dynamically, requesting for battery saving or higher performance [23].

4.4 Decision Task

The decision task periodically executes algorithms (heuristics or learning-based techniques) to decide when and which system resource will be the target of a runtime adaptation. This task hosts the main intelligence of DMOM, which is based on the state awareness provided by observation tasks, decides for a balance between application requirements, user experience, and physical budgets. The objectives are converted to actions and receive different priorities [23] over time.

Here, ideas as proposed by the centralized management of Shamsa et al. [23] can be implemented. In this work, decisions are taken by a learning-based algorithm (Q-learning) which access the current status of the system, user's task, and user's commands together with a reward function that can support or contest past adaptations. The method uses a state detector to gather all necessary information. Following the MA paradigm, the state detector could be divided into different observation tasks (as previously described), parallelizing system monitoring.

After making a decision, the decision task sends a message (detailed in Table 3) to the corresponding actuation task to start the system's adaptation.

4.5 Actuation Tasks

Actuation tasks manage the physical reconfiguration of the system handling the actuation messages and implementing the corresponding adaptive technique to change the system. Three classes of actuation tasks are proposed to cover the three resource categories: Task Mapping/Migration Manager (computation QoS), CS Manager (communication QoS), DVFS management (physical). Actuation tasks interact with the adaptation enforcer (at the OS level), using the API calls detailed in Table 2.

5 EXPERIMENTAL RESULTS

We compare the MA paradigm with state-of-the-art works that implement distributed management approaches: cluster-based management (CBM) [10], and per-application management (PAM) [25]. The first experiment (Section 5.1) compares the management approaches using task mapping as case-study. We evaluate the following parameters: (*i*) *resource utilization*, how much system resources each approach requires to work; (*ii*) *mapping latency*; how fast each approach executes the resource management; and (*iii*) *communication volume*, data volume exchanged by each approach. The second experiment (Section 5.2) evaluates CBM and MA approaches considering all phases of the ODA loop to assess the scalability of both approaches.

The MA, CBM, and PAM distributed management uses the Memphis MCSoC platform [15]. The platform is modeled in RTL (VHDL and SystemC-RTL), and adopts a homogeneous core architecture shown in the hardware layer of Figure 2. The CPU at each core is a Plasma processor [14] (MIPS processor), running at 100 MHz. The software is modeled in C code (mips-gcc cross-compiler, version 4.1.1, optimization O2).

5.1 Case-Study: Task Mapping

The CBM approach in [10] supports a wide range of management roles, while the PAM work [25] focuses on task mapping. This experiment addresses dynamic task mapping, allowing a fair comparison between the three distributed management approaches.

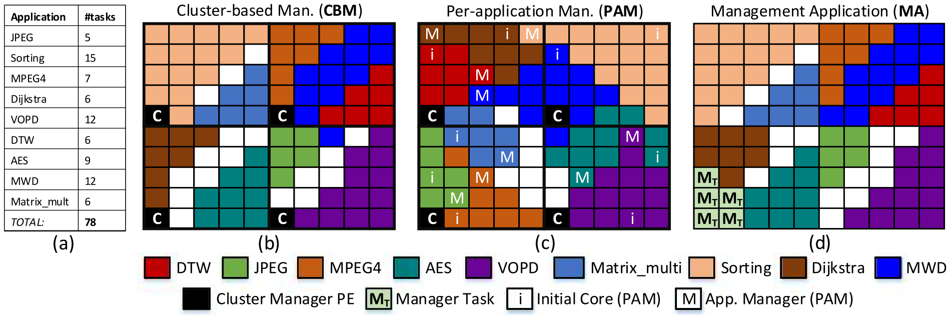


Fig. 4. Mapping overview of one random scenario. (a) list of applications, number of tasks, and the mapping sequence in one of the experiments. (b) CBM mapping [10]. (c) PAM mapping [25]. (d) MA mapping (proposed).

The task mapping algorithm was proposed in [25] along with the PAM management organization. In this experiment, we implemented the work of [25] leveraging the PAM-based task mapping model. As the task mapping is a distributed algorithm, in CBM, we implemented it in each C core (four in total), and for MA, we implemented it with five tasks, four to execute the task mapping for a cluster of cores, and one to manage the application admission from an external peripheral that contains the tasks' code (*app_injector*). The application admission management is also executed in CBM and PAM. One cluster manager C (usually the one closest to the external interface) includes the methods to interface with the *app_injector*. For fairness of comparison, the MA's tasks run on different cores since CBM and PAM assume their managers running on dedicated cores.

The experiment adopts a 10x10 MCSoC, partitioned into four 5x5 clusters, with eight applications requesting to be sequentially mapped into the system. Note that the goal is not to evaluate the quality of the task mapping heuristic but the management protocol cost. Therefore, the task mapping heuristic is the same for all approaches, consisting of: (*i*) select one core to map the first task of the application based on the maximum average distance from other applications [25]; and, (*ii*) starting from this core, mapping each application task based on a diamond search between the free cores.

Figure 4(a) presents the applications' list used in the experiment. The application set comprises 78 tasks for a 100-core system (78% of nominal utilization). Since we are using task mapping as a case-study, two criteria affect the management protocol performance: (*i*) the number of tasks per application; (*ii*) application admission order. We adopt a heterogeneous set of applications regarding the number of tasks, varying from 5 up to 15 tasks. Regarding the application order, we evaluate 10 different scenarios with the application admission order randomly selected. Figure 4(a) depicts the mapping order of one of the random scenarios.

5.1.1 Resource Utilization. Figure 4(b, c, d) shows the final task mapping for CBM, PAM, and MA, respectively. The mapping order scenario corresponds to the one presented in Figure 4(a) (other scenarios have identical results regarding resource utilization). The goal is to evaluate how many management resources each technique requires. CBM requires one manager core (C) per cluster. Assuming a 5x5 cluster, the resulted resource utilization (allocated cores) required for management was 4% of the total number of cores.

PAM requires one C per cluster, one manager per application (M), and one initial core (i) per application. Initial cores (i) are temporary managers that are created for each application to map its tasks to free cores. During task mapping, the i core communicates with C and M cores to achieve

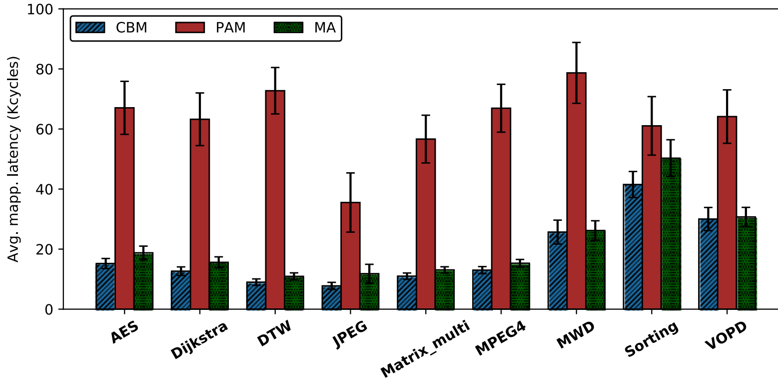


Fig. 5. Average mapping latency for all scenarios (bars). Lines represent standard error.

a consistent and temporary global view of the system resources. We optimize the PAM resource utilization by reusing core *i* after it finishes the application mapping, allocating to this core one task of its application. Despite this optimization, PAM needs a higher number of management resources, corresponding to 13% of the total number of cores.

With five MA tasks, the resources required for management are 5% of the total cores (61.5% fewer resources than PAM). MA could be implemented with four tasks, resulting in the same resource utilization as CBM.

Note that in Figure 4(d), MA's tasks are statically mapped in neighbors cores and not distributed over the system. However, the designer can choose to map a given MA task in a specific place, for instance, place a monitoring task close to an application that produces monitoring traffic with an intensive rate, or map a decision task in a region with faster cores. As the MA's tasks communicate with each other according to the messages previously presented, mapping them closer to each other brings two main benefits:

- Reducing the Manhattan distance among management tasks helps to reduce the communication latency among them and also reduce communication energy into the chip (the same rule valid for mapping the user's tasks [25]);
- Grouping management tasks helps to open space in the many-core to map users' tasks with less fragmentation than when the managers are spread over the system (like CBM).

The flexibility to map management tasks is the same as to map users' applications, enabling the computational load distribution among several tasks.

5.1.2 Mapping Latency. Figure 5 shows the average mapping latency in kilo clock-cycles (Kcycles) for all scenarios. The mapping latency comprises the time from the beginning of the mapping for the first task until the end of the mapping for the last task, measured in the manager processor. As each approach uses the same task mapping heuristic, this result reflects the protocol cost between different management components.

MA exhibits an average latency 66% lower than PAM and 16% higher than CBM. The difference to CBM is due to the implementation of the MA management at the application level, which incurs in context saving (due to the system calls) and scheduling overheads.

PAM has a higher overhead, being 3.4 times more costly than CBM and 2.9 times more costly than MA. While PAM has a scalable and modular design (assuming one manager per application), it leads to more heterogeneous management with three different managers, requiring higher synchronization among managers, increasing the mapping latency.

5.1.3 Communication Volume. The communication volume of the approaches corresponds to: PAM = 14.7 KB, CBM = 0.33 KB, and MA = 0.55 KB. MA presents a reduction of 96% compared to PAM. The communication volume between CBM and MA has a low difference because they adopt a similar management protocol, with one mapping manager (either a dedicated core or a MA task) in charge of a cluster of cores. PAM produces more messages due to the synchronization among three different managers.

5.1.4 Distributed Management Approaches - Results Overview. This first evaluation of the management approaches, using task mapping as a case-study, showed that PAM imposes a high management overhead compared to MA and CBM. MA and CBM are similar related to communication volume, resource utilization, and slightly different in mapping latency. The advantage of the proposed MA approach, compared to CBM and PAM, is flexibility and modularity. The next Subsection evaluates the CBM and MA approaches considering all phases of the ODA loop.

5.2 DMOM Management Latency

This Subsection evaluates the DMOM-MA framework presented in Section 4, using MA and CBM implementations. Definitions presented below are required to create a fair comparison between both approaches.

- Each core sends a set of monitoring messages m_t, m_u, m_s; where: m_t, corresponds to the task monitored data (TASK_MONITOR_DATA in Table 3); m_s, contains the system status (SYS_MONITOR_DATA); m_u, contains the user commands (USER_MONITOR_DATA).
- ***M:*** set of monitoring messages defined according to the number of objectives to meet: $M = \{m_t\}$ for 1 objective – **1-OBJ**, $M = \{m_t, m_s\}$ for **2-OBJ**, $M = \{m_t, m_s, m_u\}$ for **3-OBJ**.
- ***O_W:*** observation window in which the monitoring messages are sent. In the MA approach, m messages are sent to the corresponding Observation task. In CBM, M messages are sent to the manager core C. O_W is assumed as 1 ms in the experiments, which corresponds to monitoring periods typically found in literature, especially for power management [10, 22];
- The time to handle one monitoring message is on average 1,000 clock cycles, a value obtained from the RTL simulation;
- ***$t(D_T)$:*** execution time of the decision task, assumed as $t(D_T) = 0.01 * N_C * N_O$ ms, where N_C is the number of cores in a cluster, and N_O is the number of objectives to meet. The choice of this equation comes from the fact that the execution time of a decision heuristic increases according to the number of cores and the objectives to meet;
- ***$t(A_T)$:*** execution time of an adaptation task, assumed as $t(A_T) = 1$ ms, which roughly corresponds to a switching modification, or task migration or task mapping [17];
- ***A_R:*** the execution of the decision task triggers an adaption when occurs a violation in an objective. Experiments adopt $A_R = \{10\%, 25\%, 50\%\}$. An $A_R = 10\%$ is a typical rate observed in experiments related to QoS actuation [17], and an $A_R = 50\%$ is a pessimistic scenario, where half of the decision executions triggers an adaptation, representing a system that is constantly adapting its resources due to constraints violations.

The observation, decision, and actuation have a synthetic behavior to create controllable and reproducible experiments, i.e., O_W, $t(D_T)$, and $t(A_T)$ control each step's latency. Additionally, the adaptation execution is a function of the adaptation rate – A_R. An $A_R = 50\%$ means that for two decision executions, one adaptation is triggered. In practice, A_R will vary at runtime according to

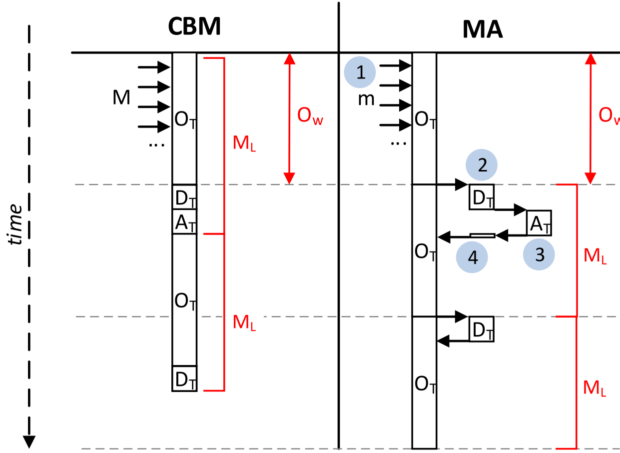


Fig. 6. ODA management latency (M_L) and observation window (O_W) for CBM and MA approaches. In CBM M_L corresponds to the summation of the O/D/A latencies, while in MA the O and D/A latencies are parallelized.

the system's resource availability and constraints violation, leading to a higher or lower adaptation rate.

The main performance figure to evaluate is the ODA management latency – M_L. Figure 6 presents the ODA execution behavior for both CBM and MA approaches, where O_T corresponds to an observation task, D_T to a decision task, and A_T to an actuation task.

In CBM, M arrives from the cores to the C core periodically. The C core waits to receive all expected monitoring messages within O_W. Once M is received, occurs the execution of D_T and A_T. Note that D_T may start before the end of O_W if all M messages arrive before O_W ends. At the end of A_T, the C core sends a monitoring release message to the cores allowing them to transmit new monitoring messages. The release message is required to avoid cores sending monitoring messages while C core executes D_T or A_T. Note that according to A_R, A_T may not run.

In MA, cores transmit m monitoring messages to the respective O_T (step 1). Each O_T sends an objective message to D_T after receiving all monitoring messages within O_W. Once D_T receives all objective messages (up to three), D_T effectively starts (step 2). If the decision requires an adaptation, an actuation message is sent to the respective A_T. The A_T sends an acknowledging message to D_T when it finishes (step 3). The D_T informs the O_Ts that they can send new objective messages (step 4). There is no need for a monitoring release message because O_Ts executes in parallel, not stalling due to D_T or A_T execution, as in CBM.

Due to the parallel execution of O_Ts with D_T and A_T, $M_L(MA)$ may be expressed according to Equation 1. In Equation 1(a), $M_L(MA)$ corresponds to the execution time of D_T and A_T. Otherwise, Equation 1(b), $M_L(MA)$ corresponds to O_W.

$$M_L(MA) = \begin{cases} t(D_T + A_T), & t(D_T + A_T) > O_W \quad (a) \\ O_W, & t(D_T + A_T) \leq O_W \quad (b) \end{cases} \tag{1}$$

Figure 7 evaluates M_L and the communication latency (time to handle the monitoring messages), varying the cluster size, A_R, and the number of objectives to meet. The goal of this experiment is to evaluate the scalability of both approaches.

Figure 7(a) presents M_L for A_R equal to 10%, 25%, and 50%. Considering all results, $M_L(MA)$ is, on average, 17.2% lower than $M_L(CBM)$.

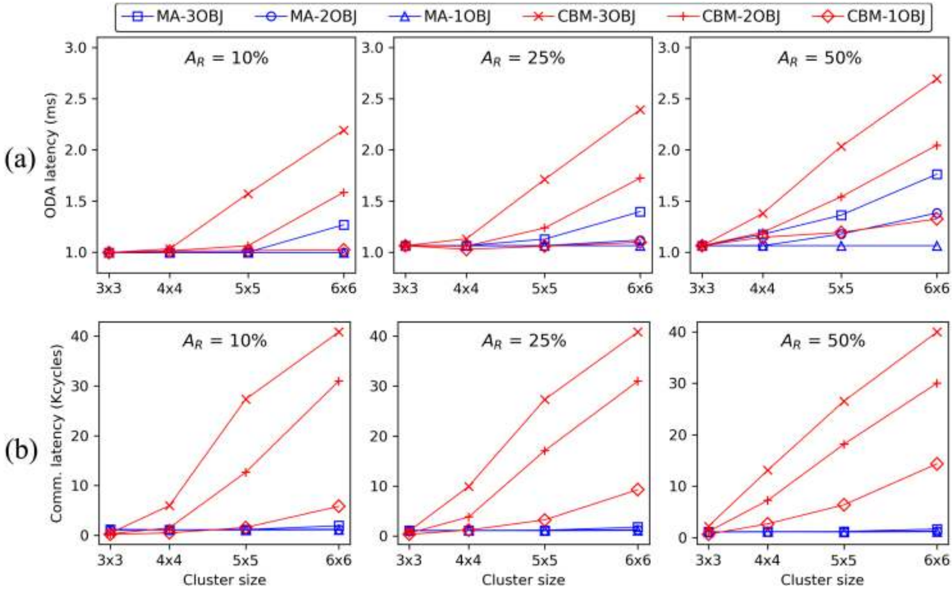


Fig. 7. Comparison of MA and CBM average monitoring latency (M_L) implementing the DMOM framework proposed in Section 4, considering four cluster sizes (3x3, 4x4, 5x5, 6x6). (a) Management Latency M_L. (b) Average communication latency per O_W.

Consider the first plot of Figure 7(a), with A_R = 10%, a typical adaptation rate. The $M_L(MA)$ follows Equation 1(b) for 1 and 2 objectives for all cluster sizes. Only for a 6x6 cluster, with 3 objectives to meet, $M_L(MA)$ follows Equation 1(a), due to the increase on the $t(D_T)$. The CBM approach meets O_W for small cluster sizes (up to 4x4), with a significant increase in $M_L(CBM)$ for 2 and 3 objectives to meet. As O_W defines the monitored data sampling period, it is expected that the adaptation occurs within this period. *This graph demonstrates the CBM approach's limitation to deal with multiple objectives, even considering a small A_R.*

A_R evaluation. $M_L(CBM)$ is 18.6%, 19.4%, and 23.9% higher than $M_L(MA)$ for an A_R equal to 10%, 25%, and 50%, respectively. The reason explaining why $M_L(CBM)$ increases with A_R comes from the sequential execution of the ODA loop. In CBM, the execution of an adaption stops the monitoring message reception, while in MA the O_Ts continue to receive monitoring messages, even if an adaptation is executing.

Number of objectives to meet. The number of objectives impacts M_L with CBM increasing at a rate of 21.6% to each added objective, while MA presents an increasing rate of 6.6%. $M_L(CBM)$ is 4.1%, 20.6%, and 35.5% higher than $M_L(MA)$ to meet 1, 2, and 3 objectives, respectively. These results show that the MA is suited to meet several objectives simultaneously, while $M_L(CBM)$ increases rapidly with their increase.

Cluster size evaluation: CBM presents an average increase rate of 19.1% to each added XY dimension, while MA increases at a rate of 5.4%. Considering the simultaneous fulfillment of two objectives, it is possible to observe the rapid increase of $M_L(CBM)$ with the increase in the cluster size, while $M_L(MA)$ is only penalized for A_R = 50%. On the other hand, the simultaneous fulfillment of three objectives, both approaches are penalized, with a smaller impact on the proposed MA approach. These results show that O_W must be adjusted according to the cluster size, and a given A_R.

Figure 7(b) evaluates the average communication latency of the monitoring messages. This latency is measured within an O_W, comprising the time from the generation of a monitoring message by LLM until its respective handling by an O_T or by a C core. The MA communication latency follows O_W, being on average 90.2% lower than CBM. By using multiple and dedicated O_Ts to handle the large amount of monitoring data generated by multi-objective management, the monitoring messages are handled in parallel (see Figure 6) avoiding being stalled during decision and actuation and reducing the bottlenecks in the NoC.

6 CONCLUSION

We proposed the Management Application (MA) and a framework of multi-objective management (DMOM) based on the MA, and evaluate it in comparison to two other distributed resource management paradigms for MCSoCs (CBM, PAM). A core aspect of MA is its implementation as a distributed management application (at the user's application level), decoupling several management components from the operating system, and bringing properties of flexibility, modularity, and parallelism.

We presented two experiments on resource management, one focused on task mapping, and the other one focused on the complete DMOM framework. The task mapping case-study showed that MA and CBM have a significant improvement compared to PAM in all performance aspects (latency, resource utilization, and communication volume). MA and CBM presented similar communication volume and resource utilization, with MA achieving a management latency 16% higher than a CBM, which occurs due to the overheads of context saving and scheduling of MA task, which CBM does not have due to its implementation at the OS level. The second experiment compares CBM and MA in a DMOM framework and shows that MA can reduce the management latency on average by 17.2% and message latency by 90.2%. The advantage of MA for multi-objective management is the flexibility allowing the system developer to split the management tasks, which favors the scalability of the management system regarding cluster size and the ability to manage several objectives simultaneously.

Directions for future words include: (*i*) evaluate the impact of the decision algorithms in the management techniques; (*ii*) assess methods to allocate ODA tasks according to the system load dynamically.

REFERENCES

[1] M. Al Faruque, J. Jahn, T. Ebi, and J. Henkel. 2010. Runtime thermal management using software agents for multi- and many-core architectures. *IEEE Design Test of Computers* 27, 6 (2010), 58–68. https://doi.org/10.1109/MDT.2010.94

[2] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris. 2013. Distributed run-time resource management for malleable applications on many-core platforms. In *DAC*. ACM, 168:1–168:6. https://doi.org/10.1145/2463209.2488942

[3] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas. 2017. KiloCore: A 32-nm 1000-processor computational array. *J. Solid-State Circuits* 52, 4 (2017), 891–902. https://doi.org/10.1109/JSSC.2016.2638459

[4] E. Carara, N. Calazans, and F. G. Moraes. 2014. Differentiated communication services for NoC-Based MPSoCs. *IEEE Trans. Computers* 63, 3 (2014), 595–608. https://doi.org/10.1109/TC.2012.123

[5] B. D. de Dinechin, R. Ayrignac, P. Beaucamps, P. Couvert, B. Ganne, P. G. de Massas, F. Jacquet, S. Jones, N. M. Chaisemartin, F. Riss, and T. Strudel. 2013. A clustered manycore processor architecture for embedded and accelerated applications. In *HPEC*. IEEE, 1–6. https://doi.org/10.1109/HPEC.2013.6670342

[6] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila. 2011. Exploration of MPSoC monitoring and management systems. In *ReCoSoC*. IEEE, 1–3. https://doi.org/10.1109/ReCoSoC.2011.5981544

[7] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A Agarwal. 2010. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *ICAC*. ACM, 79–88. https://doi.org/10.1145/1809049.1809065

[8] H. Khdr, S. Pagani, M. Shafique, and J. Henkel. 2018. Chapter four - dark silicon aware resource management for many-core systems. In *Dark Silicon and Future On-chip Systems*, Ali R. Hurson and Hamid Sarbazi-Azad (Eds.). Elsevier, 127–170. https://doi.org/10.1016/bs.adcom.2018.03.002

[9] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel. 2011. DistRM: Distributed resource management for on-chip many-core systems. In *CODES+ISSS*. ACM, 119–128. https://doi.org/10.1145/2039370.2039392

[10] A. Martins, A. L. da Silva, A. Rahmani, N. Dutt, and F. G. Moraes. 2019. Hierarchical adaptive multi-objective resource management for many-core systems. *Journal of Systems Architecture* 97 (2019), 416–427. https://doi.org/10.1016/j.sysarc.2019.01.006

[11] A. Miele, A. Kanduri, K. Moazzemi, D. Juhász, A. Rahmani, N. D. Dutt, P. Liljeberg, and A. Jantsch. 2019. On-Chip dynamic resource management. *Foundations and Trends in Electronic Design Automation* 13, 1-2 (2019), 1–14. https://doi.org/10.1561/1000000055

[12] S. Pagani, H. Khdr, J. Chen, M. Shafique, M. Li, and J. Henkel. 2017. Thermal safe power (TSP): Efficient power budgeting for heterogeneous manycore systems in dark silicon. *IEEE Trans. Comput.* 66, 1 (2017), 147–162. https://doi.org/10.1109/TC.2016.2564969

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

[14] S. Rhoads. 2016. Plasma - most MIPS I(TM). https://opencores.org/projects/plasma.

[15] M. Ruaro, L. Caimi, V. Fochi, and F. G. Moraes. 2019. Memphis: A framework for heterogeneous many-core SoCs generation and validation. *Design Automation for Embedded Systems* 23, 3 (2019), 103–122. https://doi.org/10.1007/s10617-019-09223-4

[16] M. Ruaro, E. Carara, and F. Moraes. 2015. Runtime adaptive circuit switching and flow priority in NoC-Based MPSoCs. *IEEE Trans. Very Large Scale Integr. Syst* 23, 6 (2015), 1077–1088. https://doi.org/10.1109/TVLSI.2014.2331135

[17] M. Ruaro, A. Jantsch, and F. G. Moraes. 2019. Self-Adaptive QoS management of computation and communication resources in many-core SoCs. *ACM Transaction on Embedded Computing Systems* 18, 4 (2019), 37:1–37:21. https://doi.org/10.1145/3328755

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

[19] S. Saponara, T. Bacchillone, E. Petri, L. Fanucci, R. Locatelli, and M. Coppola. 2014. Design of an NoC interface macrocell with hardware support of advanced networking functionalities. *IEEE Trans. Comput.* 63, 03 (2014), 609–621. https://doi.org/10.1109/TC.2012.70

[20] S. Sarma, N. D. Dutt, P. Gupta, A. Nicolau, and N. Venkatasubramanian. 2014. On-chip self-awareness using Cyberphysical-Systems-on-Chip (CPSoC). In *CODES+ISSS*. ACM, 22:1–22:3. https://doi.org/10.1145/2656075.2661648

[21] M. Shafique and S. Garg. 2017. Computing in the dark silicon era: Current trends and research challenges. *IEEE Design & Test* 34, 2 (2017), 8–23. https://doi.org/10.1109/MDAT.2016.2633408

[22] M. Shafique, B. Vogel, and J. Henkel. 2013. Self-adaptive hybrid dynamic power management for many-core systems. In *DATE*. ACM, 51–56. https://doi.org/10.7873/DATE.2013.025

[23] E. Shamsa, A. Kanduri, A. M. Rahmani, P. Liljeberg, A. Jantsch, and N. Dutt. 2019. Goal-Driven autonomy for efficient on-chip resource management: transforming objectives to goals. In *DATE*. IEEE, 1397–1402. https://doi.org/10.23919/DATE.2019.8715134

[24] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel. 2013. Mapping on multi/many-core systems: Survey of current and emerging trends. In *DAC*. ACM, 1–10. https://doi.org/10.1145/2463209.2488734

[25] V. Tsoutsouras, I. Anagnostopoulos, D. Masouros, and D. Soudris. 2018. A hierarchical distributed runtime resource management scheme for NoC-Based many-cores. *ACM Transactions on Embedded Computing Systems* 17, 3 (2018), 65:1–65:26. https://doi.org/10.1145/3182173

[26] S. Wildermann, M. Glaß, and J. Teich. 2014. Multi-objective distributed run-time resource management for many-cores. In *DATE*. IEEE, 1–6. https://doi.org/10.7873/DATE.2014.234

[27] Y. Xiao, S. Nazarian, and P. Bogdan. 2019. Self-Optimizing and self-programming computing systems: A combined compiler, complex networks, and machine learning approach. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems* 27, 6 (2019), 1416–1427. https://doi.org/10.1109/TVLSI.2019.2897650

[28] Y. Xue, Z. Qian, G. Wei, P. Bogdan, C. Tsui, and R. Marculescu. 2014. An efficient Network-on-Chip (NoC) based multicore platform for hierarchical parallel genetic algorithms. In *NOCS*. ACM, 17–24. https://doi.org/10.1109/NOCS.2014.7008757

[29] H. Zhang and H. Hoffmann. 2016. Maximizing performance under a power cap: A comparison of hardware, software, and hybrid techniques. In *ASPLOS*. ACM, 545–559. https://doi.org/10.1145/2872362.2872375

Received January 2020; revised January 2021; accepted March 2021