Management Application - a New Approach to Control Many-Core Systems

Angelo Elias Dalzotto*, Marcelo Ruaro†, Leonardo Vian Erthal*, Fernando Gehm Moraes*
*PUCRS – School of Technology, Porto Alegre, Brazil
angelo.dalzotto@edu.pucrs.br, leonardo.e@edu.pucrs.br, fernando.moraes@pucrs.br
†Univ. Bretagne-Sud, UMR 6285, Lab-STICC, Lorient, France – marcelo.ruaro@univ-ubs.fr

2021 34th SBC/SBMicro/IEEE/ACM Symposium on Integrated Circuits and Systems Design (SBCCI) | 978-1-6654-2170-6/21/$31.00 ©2021 IEEE | DOI: 10.1109/SBCCI53441.2021.9529989

***Abstract*—The increasing core count in many-core systems introduced management challenges, including scalability, portability, and reduced overhead in the user's applications. Works available in the literature seek to manage a given objective, such as power, temperature, communication, and quality-of-service. These management strategies are tightly coupled to the hardware platform and the operating system (OS) running on it. This coupling implies the lack of management modularity, resulting in low flexibility related to modifying management strategies at runtime, and low portability. State-of-the-art shows that few works propose management strategies or frameworks, only evaluating the proposed objective's quality. This work aims to present a new approach to control many-core systems, named Management Application (MA), which can implement multi-objective management decoupled from the hardware and the OS through a set of high-priority tasks. MA transforms the management problem into a distributed application, allowing the management to truly benefit from the high parallel power of many-cores. The MA approach is demonstrated with a proof-of-concept framework. Results evaluate the cost to adopt MA, compared to the cluster management, and the benefits of adopting MA to manage a benchmark with real-time constraints revealing improved memory footprint and higher management throughput due to its parallelization.**

***Index Terms*—Many-core; resource management; system management.**

I. Introduction

The increasing number of cores inside a chip results in increased system complexity [1]. In a many-core, the communication mechanism must serve all cores with reduced delays, and the performance must meet the requirements of tasks that are often real-time. The power must also be under the designed budget, which is even more limited for IoT applications. The temperature should not exceed the physical limits to prevent excessive silicon wear out. These challenges must be met by the hardware and by the system managing the many-core. Using the Operating System (OS) and/or dedicated management components, state-of-the-art many-cores are increasingly evolving to meet multiple management objectives that can be conflicting, like power versus performance.

Many-core management scalability has been tackled initially by separating the system into clusters, reserving processors to manage these areas. Another approach is to dedicate one manager processor for each application running in the system. Both methods present weaknesses, and neither is inherently modular. The main drawbacks include the specialization of a set of processors for management, requiring dedicated OS support, which implies a lack of modularity, i.e., it is impossible to add new management functions at runtime. State-of-the-art management strategies revealed few-to-none concerns with modularity or portability features. These works use consolidated management organizations to evaluate one or more system constraints.

This work *proposes* a new management approach, Management Application (MA), decoupled from hardware and OS targeting portability and modularity while keeping similar or even lower management overheads compared to state-of-the-art approaches. MA brings benefits to designers. First, it decouples the management tasks from the OS, making it light and equal for all Processing Elements (PE). Second, the MA makes the management infrastructure agnostic to the hardware, making it possible to replace the OS or the processor.

Experiments revealed an increase in management throughput by leveraging the many-core parallelism not just in user tasks but also in management tasks. The memory footprint is reduced w.r.t. other management strategies due to the simpler many-core organization.

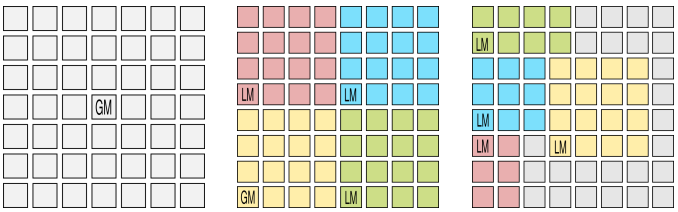
This paper is organized as follows. Section II presents related work. Section III details the MA proposal. Section IV presents a proof-of-concept for the MA. Section V presents results comparing the MA proposal to the cluster organization. Finally, Section VI concludes this paper and points out directions for future works.

II. Related Work

To fully exploit the parallelism offered by many-cores it is necessary to execute several *management tasks*, such as: (*i*) map tasks to cores aiming reduced latency and communication energy; (*ii*) migrate tasks to avoid hot spots, fragmentation, and meet deadlines; (*iii*) control Dynamic Voltage and Frequency Scaling (DVFS) to keep the execution under power and temperature constraints, preventing the introduction of dark cores; (*iv*) increase reliability and lifetime. The *many-core management organization* defines *where* the management is located and *how* it runs in a many-core.

Figure 1 presents the three main *management organizations*: centralized management, Cluster-Based Management (CBM), and Per Application Management (PAM). In centralized management, shown in Figure 1a, the many-core allocates one PE, called Global Manager (GM), to be the controller of all management actions. Figure 1b presents the CBM approach, which despite using a GM to synchronize the management, divides the many-core into regions, named ***clusters***, controlled by Local Managers (LMs). PAM is another distributed way

978-1-6654-2170-6/21/$31.00 ©2021 IEEE



(a) Centralized (b) CBM (c) PAM

Fig. 1: Management organizations.

to manage a many-core shown in Figure 1c that dynamically assigns a manager for each running application.

One possible way to implement system management is to use the *ODA control loop* [2]. ODA is a modular method to divide the many-core management roles into three phases. These phases are:

- **Observe** - Extracts information about the system or task status, like deadline violations, temperature, and communication latency.
- **Decide** - Algorithms that decide what resources and when they will be reconfigured to meet application constraints or system budget. A multi-objective decision can be made from multiple different data from the *observe* phase.
- **Act** - Enforces the decision made from the previous step, integrating the ODA loop to the hardware or the OS through techniques as DVFS, task migration, the establishment of communication paths, changes in priority of task scheduler, among others.

A. Centralized approaches

Centralized management is the most straightforward organization architecture for a many-core. The works [3, 4] adopt centralized management, where the global resource manager is located outside the many-core fabric.

Rahmani et al. [4] connect a many-core to a host machine controlling runtime mapping and Dynamic Power Management (DPM). Their work focus on a multi-objective DPM that also considers performance fulfillment, reliability, and system lifetime. They claim to have achieved enhanced throughput, Thermal Design Power and Thermal Safe Power constraints, and improved lifetime compared to state-of-the-art management policies. Mariani et al. [3] propose the ARTE framework that uses a general-purpose host processor called "fabric controller" connected to a many-core fabric with 16 MIPS-like processors.

According to Castilhos et al. [5], the central manager can be quickly overloaded by answering requests from numerous PEs, also generating a considerable amount of traffic around it. Therefore, centralized management is only suited for lower core counts due to its poor scalability.

B. CBM approaches

CBM stands out as the most widespread organization in the literature. It divides the management into clusters, managed by a cluster manager, implemented in a dedicated core. The Authors in [6, 7] assume two hierarchical management levels with slave cores that execute user tasks and a system core assigned to the management of each slave cluster. The works [8, 9] 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.

Agent-based Distributed Application Mapping (ADAM) [10] was the first hierarchical CBM proposal, using a "global agent" and "cluster agents". The Authors' goal is to present a distributed runtime application mapping, which showed more than seven times less computational effort than a centralized mapper. Gregorek et al. [11] present Dracon. Dracon has a RunTime Management (RTM) PE connected to each core of its cluster, acting as an OS serving system calls and scheduling tasks, and communicates with each other RTMs using a dedicated management network.

The CBM's main advantage is that it is easily scalable to a large number of cores. However, the trade-offs are: (*i*) the resulting overhead of lost PEs used only for management purposes, mainly occurring on huge many-cores with smaller cluster sizes; and (*ii*) the cluster size is essentially static, with its size and location being decided at design time, impairing the system dynamics.

C. PAM approaches

Liao and Srikanthan [12] adopt the PAM approach, with a global manager to create rectangular sub-meshes and assign one core of this area to be a local manager that will control the application hierarchically. DistRM [13] also uses the PAM organization but is fully decentralized without any global synchronization by assigning each manager called "agent" randomly at application arrival. These managers are based on the concept of multi-agent systems, distributing the control among agents that have local information about the system and act independently of each other.

Anagnostopoulos et al. [14] use PAM in a more deep hierarchical way. They first separate the many-core into clusters based on "controller cores", using "initial cores" for temporary resource management at application arrival, and then finally creating "manager cores" for each application management and resource exchange. Compared to DistRM, this work shows reduced communication overhead with 70% fewer messages and 64% less message size while gaining up to 20% speed-up.

PAM addresses the CBM's problem of lacking runtime mutability by creating and killing managers at runtime. However, two main trade-offs exist: (*i*) having a considerable overhead for applications with a small number of tasks; and (*ii*) focusing mainly on applications QoS instead of meeting the whole system goals.

D. Final remarks

Most of the evaluated works, including the most recent ones [15, 16], adopt a given management organization to evaluate an algorithm, hardware, or framework. This fact resulted in many centralized resource management works due to its straightforward implementation, with a single PE to execute the management procedures, resulting in poor scalability.

The management organizations (centralized, CBM, and PAM) rely on dedicated cores for management, which requires special support from the OS kernel, imposing challenges to reuse the management procedures in other systems and making

it difficult to add new goals. CBM organization that defines statically a many-core division, some researched centralized organizations [3, 4] that rely on a external manager hardware, and the work of [11] that uses a hardware architecture for management, are tightly coupled to its hardware, making it unfeasible to port an already verified management strategy to different platforms. Despite being scalable, PAM and CBM result in an overhead of reserved resources for management, resulting in a potential loss of parallelism.

III. Management Application – MA

The principle of the MA organization is the *absence* of processors dedicated to system management. All management functions are removed from the OS, running as distributed tasks in user-space. This new paradigm transforms the management problem in a distributed application, allowing the management to truly benefit from the high parallel power of many-cores. Figure 2 presents an example of management tasks according to the MA approach. Blue, green and red PEs represent Observation, Decision, and Actuation tasks, respectively. 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. All PEs running management tasks can be shared with with user tasks.

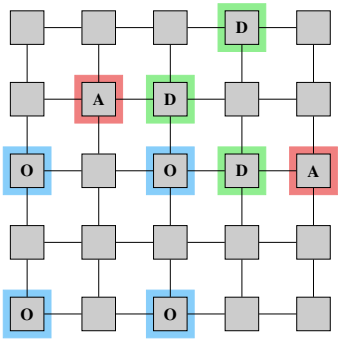


Fig. 2: MA organization. **O** - Observation, **D** - Decision, **A** - Actuation.

Note that the mapping of MA tasks presented in Figure 2 is just an example. The tasks can be mapped close to each other, allocated close to strategical regions of the system where the monitoring load is higher, or allocated in cores with more computing power (especially for decision tasks that need to run heuristics at runtime).

The main advantages of the MA paradigm include:

- No need for dedicated cores for management execution as in centralized management, CBM, and PAM. Management tasks can share processors with user tasks.
- Management tasks are not bound to specific locations as in other paradigms. They can also be migrated, bringing additional reliability in case of violated thermal constraints or faulty cores.
- The OS becomes lighter and easier to maintain since it is not overloaded or modified with new resource management modules insertion.

To adopt the MA organization, it is necessary to include in the OS of each PE:

- **Low-Level Monitors (LLM)**: periodically pulls raw data from hardware and redirects to Observation tasks without executing complex computation. LLM examples include deadline miss detection, communication latency, task profiling (computation- or communication-intensive, or hybrid), heart-beat for periodic applications, power and thermal, and core utilization.
- **Adaptation Enforcer (AE)**: provides the drivers to physically apply the requests from Actuation tasks. Examples are DVFS and task migration drivers (which require kernel privileged memory access).
- **Management Communication API**: the OS must provide a secure communication method for MA tasks, ensuring that user tasks do not tamper the system.

Figure 3 depicts the MA framework model. The LLM generates messages periodically. Observation tasks handle these messages, jointly with user commands. The Observation tasks know the system and applications constraints and can convert raw monitoring data into objectives. The objectives are periodically sent to the Decision task that converts them into goals [17] by using algorithms that detect when and what resource needs actuation. If necessary, the Decision task triggers an Actuation task, which implements the protocols to dynamically change the resources by interacting with the AE at the OS level.

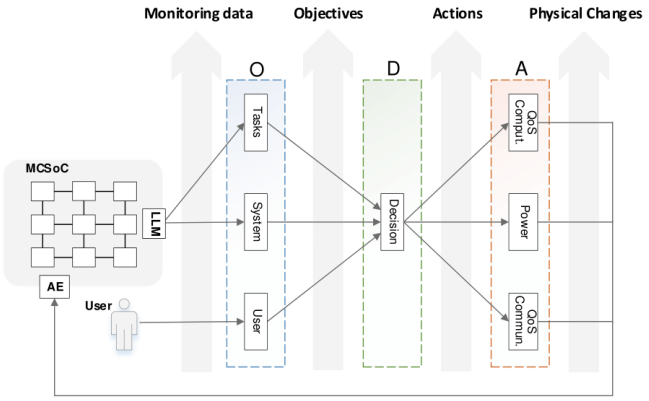


Fig. 3: ODA model used MA. MCSoC: Many-core SoC.

IV. Proof-of-concept Implementation

Sections IV-A and IV-B present components of the MA approach, the Management Communication API and the MA task injection method, respectively. Section IV-C presents the proof-of-concept, with the ODA tasks.

A. Management Communication API

Message exchange mechanisms, such as MPI, use send-receive methods between communicating task pairs. However, management tasks are reactive, triggered by multiple sources at any time (for example, many observation messages can be directed to a single decision task). Additionally, there is communication between MA tasks and the kernel, and usually MPI primitives are targeted to task-to-task communication.

This work proposes a *management* communication API that is used by MA tasks. This mechanism is similar to

the MPI_ANY_SOURCE directive of MPI, differing from it to allow task-to-kernel communication. Figure 4 presents the Management Communication API sequence diagram.

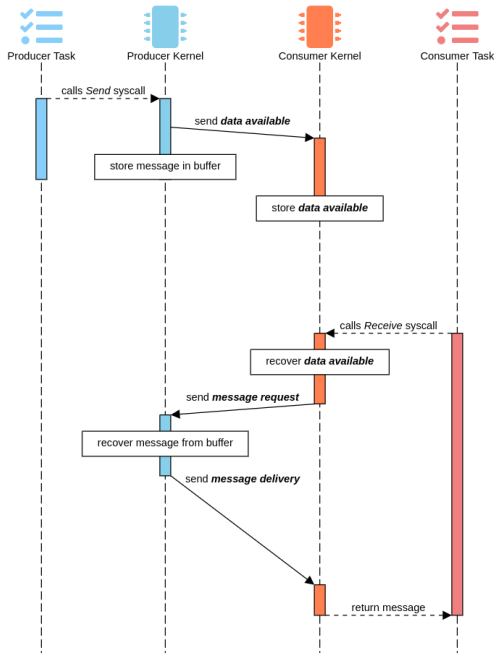


Fig. 4: Sequence diagram of the new message-passing API for management communication.

In Figure 4 when a producer wants to send a message, a *data available* packet is generated containing the producer task *id* and location. When the consumer calls the receive message function, e.g., when all decision algorithms are done and the task is put in a blocked state, it checks data availability sent by producers. For each producer, the consumer then sends a *message request*, knowing its task id and location from the *data available* message. Finally, each producer kernel dispatches the stored message inside a *message delivery* packet. User tasks do not have access to this API, ensuring that the management system is not disturbed by malicious tasks.

B. MA Injector

The public-available Memphis many-core [18] is the baseline system used to build the proposed MA approach. Memphis has a homogeneous region with PEs, and peripherals connected to the many-core borders. Each PE contains: a 32-bit RISC processor (CPU); a true dual-port scratchpad memory for instructions and data; a Direct Memory Network Interface (DMNI), integrating a Network Interface and a DMA modules; and the NoC router.

Memphis has a default peripheral, *Application Injector*, that deploys applications into the system. This peripheral can be seen as an Ethernet interface receiving applications to be executed in the platform. For security reasons, it is necessary to separate the deployment of management tasks from user tasks. Thus, a second peripheral was added to the Memphis platform, called *MA Injector*.

At system startup, all external interfaces of the many-core are disabled, except the MA Injector interface. This peripheral transmits the initial code of a set of ODA management tasks to a given processor, including the mapping task, acting as a flash memory with a trusted boot code. Once the MA tasks are loaded in the system, the mapping task releases the external interfaces. Note that the method does not exclude distributed mapping since multiple mapping tasks can co-exist, being hierarchically managed.

C. MA Proof-of-concept

The MA framework proof-of-concept contains ODA tasks working together to guarantee the QoS of a real-time application, triggering task migration when deadline violations occur. The ODA task set contains: (i) a real-time task monitor – Observer; (ii) a QoS Decider; (iii) a task migration Actuator.

Section III presented the three components required by the platform to support the MA framework (LLM, AE, Communication API). With the Management Communication API, and the existing OS migration support acting as the AE, the LLM is implemented in the kernel of each PE. The LLM sends raw messages periodically with deadline, slack time, and remaining execution time of all PE real-time tasks.

The OS knows whom to send LLM messages to by coupling an Observer task to each monitored user task when the mapper releases the user task to run, notifying the nearest Observer task. The mapper does this by checking a *Task Type Tag* inserted into each task's binary file, indicating if the task is O, D, A or user, and its O, D, and A capabilities, like QoS, migration, DVFS, or power management. This feature also allows the mapper to answer service discovery messages issued by the O, D, and A tasks that request the task that serves each step in the ODA loop.

For the proof-of-concept, the real-time task monitor Observer checks for missed deadlines for each monitored task, and in case of occurrence, sends a message to the QoS Decider. The QoS Decider stores in a buffer with a Least Recently Used (LRU) replacement policy the latest monitored tasks by the Observer. After a parameterizable number of deadlines misses, the mapper starts a task migration. Note that the real-time task monitor Observer could also send information to the QoS Decider about tasks meeting deadlines with sufficient slack time and processor usage to possibly execute a DVFS Actuator to lower the frequency of these tasks' processors and improve power efficiency.

V. Results

Section V-A evaluates the cost to adopt MA, in terms of memory footprint, number of management packets and discusses modularity. Section V-B compares the effect of CBM and MA in a real benchmark with QoS constraints. Section V-C also compares both approaches, but in terms of their performance.

A. MA Cost

Figure 5 compares the CBM and MA management binaries sizes. The CBM has all management tasks inside its kernel, while the MA has the management split into real-time task Observer, QoS Decider, and the migration Actuator, which also is a mapper task. The ODA task set is 78.9% smaller than the CBM kernel, while keeping the same functionality. This occurs

because the MA has the advantage of no clusters to manage, which needs more structures in memory and more code for the reclustering procedures. The size difference between the kernels running in PEs executing user tasks is negligible, even with the new communication API required in MA paradigm.

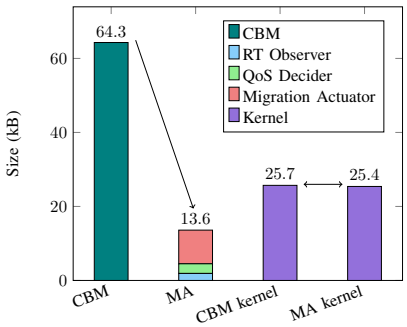


Fig. 5: Management binaries size: CBM, MA, CBM kernel, MA kernel.

The CBM requires dedicated PEs for management, with a 64.3 kB management kernel at each of these processors. In the MA approach, all PEs can run user tasks, with a kernel having roughly the same size as the CBM kernel for PEs that run user tasks. The proof-of-concept MA application requires 13.6 kB, being distributed in several PEs. This result shows that the MA makes better use of system resources without requiring more memory.

Table I shows the number of messages exchanged for CBM and MA, related to the experiment described in the next section. Despite resulting in about 87% more exchanged messages, the MA only increased the total volume of flits by about 20% because the added monitoring and management messages are small. Note that this case study is simple and contains only one application and one set of MA tasks. Real scenarios execute several applications and may use many ODA sets. Such scenarios reduce the overhead of the MA messages since the distance, in hops, between user tasks and management tasks reduces by using the MA approach.

TABLE I: Message exchange in CBM and MA.

	CBM	MA
Number of messages	1,443	2,698
Number of flits	77,257	92,830

Another advantage of the MA is modularity. While the CBM approach can only vary the size of its clusters, changing the number of managers, the MA platform can vary the number of ODA tasks and change the management goals at runtime by adding a new set of ODA tasks. Besides modularity, another advantage is portability, allowing the reuse of the ODA tasks in other platforms.

B. MA Case Study

Experiments in this Section adopt a 3x3 many-core to verify the MA feasibility. This small system corresponds to one cluster in the CBM and is scalable to larger systems by increasing the number of clusters. The benchmark is a Dijkstra's shortest path algorithm, partitioned in 7 tasks. In the MA approach, PEs 0x0, 0x1, and 0x2 receive the task mapping, QoS Decider, and real-time task monitor tasks. The CBM manager is mapped at PE 0x0. Both management approaches allow up to 4 32 kB tasks per PE.

The CPU load of each Dijkstra's task is 25%. The initial task mapping of this application is one PE executing 4 tasks and 3 PEs for the remaining tasks. The reason to adopt this mapping is to induce deadline misses. To make a fair comparison, the adopted migration heuristic [18] and the monitoring window are the same for both management approaches.

Table II evaluates the time required for the management approaches to detect and react to deadline misses and the application execution time. The 1st column details the events where the time was measured. The 2nd and 3rd columns detail the measured timestamp for CBM and MA, respectively.

TABLE II: Timestamps for the Dijkstra's application using CBM and MA (ms).

Event	CBM	MA
1st migration request	4.99	5.29
end of 1st migration	5.16	5.48
2nd migration request	9.22	5.37
end of 2nd migration	9.39	5.58
end of application execution	11.80	11.72

CBM reacts quickly than MA for the first acting condition (configured to 3 deadline misses), firing the migration before MA. This happens due to the MA pipeline structure, with messages sent from LLM to the Observer tasks, then from this task to the Decider task that decides the migration. *But this pipeline behavior is the MA strength*. Observe the second acting condition. CBM misses this event because it is finishing computing the previous decision and actuation procedures (remember that CBM executes all ODA actions in the same processor). Thus, CBM acts only in a third acting condition. As the MA has the ODA tasks split into several processors, it can detect violations from different tasks almost simultaneously.

Even with the increased number of messages due to the separated management tasks and the increased complexity of the management communication API, the application executed faster using MA (last Table row). The reason to speed up the application is the faster MA detection and actuation.

Figure 6 depicts this parallelization on a scenario where many LLM (one for each PE) sends QoS monitoring messages to the nearest Observer task, while the Observer tasks gather these data and pack to the correct Decider task that sends messages based on the chosen action. Finally, the Act tasks apply the decisions via the AE of a chosen target PE. In Figure 6 each entity is running in parallel, showing how the ODA loop introduces parallelism to the management processes with a pipeline model which truly exploits the parallel computing power provided by the many-core.

C. Management throughput

The third experiment aims to saturate the management infrastructure, using the previous experimental setup, but with

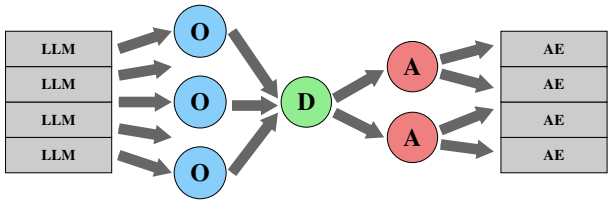


Fig. 6: MA pipeline model. The number of LLMs and AEs is one per PE. Observer tasks are defined at design time, while deciders and actuators are a function of the management objectives.

a synthetic task. The synthetic task generates bursts of deadline miss messages, and for each message, there is an actuation (task migration).

Figure 7 shows on the x-axis the sequential number of the monitoring messages and on the y-axis the time for the management technique to handle an event (time to execute the observation, decision, and migration). This chart has three regions. The first region (messages 1 to 5) corresponds to the MA warm-up, i.e., fill the MA pipeline. In this first region, CBM acts quickly, as observed in Table II. For a short period (messages 5 to 9), the CBM processor can still process the observation messages and execute the decision. However, from the ninth message onwards, the system reaches a steady-state, with the throughput adapted to the processing capacity of each management method. MA is faster than CBM due to its parallel nature.

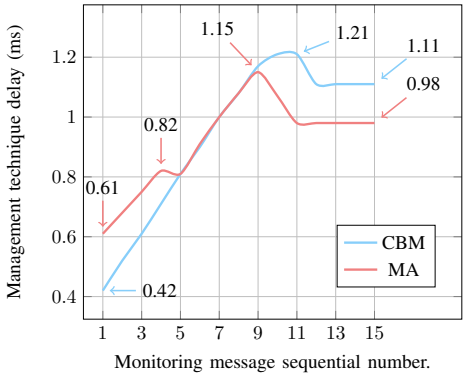


Fig. 7: Delay from monitoring message emission until task migration completion in CBM and MA.

Although small (11.7%), the observed gain was obtained in a system with one task, seeking to illustrate the behavior of management techniques. Such gain will increase in actual scenarios, where there are multiple tasks generating monitoring data (QoS, temperature, faults), which leads to several decisions. Naturally, the management structure should be scalable, such as the MA proposed in this work.

VI. Conclusion

In this paper, we presented a new method to manage many-core systems, with the following advantages w.r.t. the state-of-the-art: (*i*) there is no computational resources reservation for management; (*ii*) implementation of the management method as a distributed application. As a consequence of adopting this method, we observed that the MA reacts quickly to missed deadlines and does not penalize the execution of applications.

Future works include the MA evaluation in large systems (e.g., 10x10), comparing it to CBM, and most important, propose heuristics for multi-objective decisions.

Acknowledgment

This work was financed in part by CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), grant 309605/2020-2; and CAPES (Coordenação de Aperfeiçoamento de Pessoal de Nível Superior), Finance Code 001.

References

[1] O. Peckham, "Esperanto Unveils ML Chip with Nearly 1,100 RISC-V Cores," 2020. [Online]. Available: https://www.hpcwire.com/2020/12/08/esperanto-unveils-ml-chip-with-nearly-1100-risc-v-cores/

[2] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal, "A Generalized Software Framework for Accurate and Efficient Management of Performance Goals," in *EMSOFT*, 2013, pp. 1–10.

[3] G. Mariani, G. Palermo, V. Zaccaria, and C. Silvano, "ARTE: An Application-specific Run-Time managEment framework for multi-cores based on queuing models," *Parallel Computing*, vol. 39, no. 9, pp. 504–519, 2013.

[4] A. M. Rahmani *et al.*, "Reliability-Aware Runtime Power Management for Many-Core Systems in the Dark Silicon Era," *IEEE Transactions on VLSI Systems*, vol. 25, no. 2, pp. 427–440, 2017.

[5] G. Castilhos, M. Mandelli, G. Madalozzo, and F. Moraes, "Distributed resource management in NoC-based MPSoCs with dynamic cluster sizes," in *ISVLSI*, 2013, pp. 153–158.

[6] B. D. de Dinechin *et al.*, "A clustered manycore processor architecture for embedded and accelerated applications," in *HPEC*, 2013, pp. 1–6.

[7] Y. Xiao, S. Nazarian, and P. Bogdan, "Self-Optimizing and Self-Programming Computing Systems: A Combined Compiler, Complex Networks, and Machine Learning Approach," *IEEE Transactions VLSI Systems*, vol. 27, no. 6, pp. 1416–1427, 2019.

[8] M. Al Faruque, J. Jahn, T. Ebi, and J. Henkel, "Runtime Thermal Management Using Software Agents for Multi- and Many-Core Architectures," *IEEE Design & Test of Computers*, vol. 27, no. 6, pp. 58–68, 2010.

[9] W. Quan and A. D. Pimentel, "A Hierarchical Run-time Adaptive Resource Allocation Framework for Large-scale MPSoC Systems," *Design Automation for Embedded Systems*, vol. 20, no. 4, pp. 311–339, 2016.

[10] M. A. A. Faruque, R. Krist, and J. Henkel, "ADAM: Run-time agent-based distributed application mapping for on-chip communication," in *DAC*, 2008, pp. 760–765.

[11] D. Gregorek, J. Rust, and A. Garcia-Ortiz, "DRACON: A Dedicated Hardware Infrastructure for Scalable Run-Time Management on Many-Core Systems," *IEEE Access*, vol. 7, pp. 121 931–121 948, 2019.

[12] X. Liao and T. Srikanthan, "A scalable strategy for runtime resource management on NoC based manycore systems," in *ISICir*, 2011, pp. 297–300.

[13] S. Kobbe, L. Bauer, D. Lohmann, W. Schröder-Preikschat, and J. Henkel, "DistRM: Distributed resource management for on-chip many-core systems," in *CODES+ISSS*, 2011, pp. 119–128.

[14] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris, "Distributed run-time resource management for malleable applications on many-core platforms," in *DAC*, 2013, pp. 1–6.

[15] X. Huang, X. Wang, Y. Jiang, A. K. Singh, and M. Yang, "Dynamic Allocation/Reallocation of Dark Cores in Many-Core Systems for Improved System Performance," *IEEE Access*, vol. 8, pp. 165 693–165 707, 2020.

[16] M. H. Haghbayan, A. Miele, Z. Zouv, H. Tenhunen, and J. Plosila, "Thermal-Cycling-aware Dynamic Reliability Management in Many-Core System-on-Chip," in *DATE*, 2020, pp. 1229–1234.

[17] E. Shamsa *et al.*, "Goal-Driven Autonomy for Efficient On-chip Resource Management: Transforming Objectives to Goals," in *DATE*, 2019, pp. 1397–1402.

[18] M. Ruaro, L. L. Caimi, V. Fochi, and F. G. Moraes, "Memphis: a framework for heterogeneous many-core SoCs generation and validation," *Design Automation for Embedded Systems*, vol. 23, no. 3-4, pp. 103–122, 2019.