Dynamic Mapping for Many-cores using Management Application Organization

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 28th IEEE International Conference on Electronics, Circuits, and Systems (ICECS) | 978-1-7281-8281-0/21/$31.00 ©2021 IEEE | DOI: 10.1109/ICECS53924.2021.9665547

***Abstract*—The increasing core count in many-core systems introduced management challenges, including scalability, portability, and reduced overhead in user applications. This work adopts the Management Application (MA) organization, where management tasks execute as applications loosely coupled to the OS, meeting the above challenges. The main management task executed in a many-core is the application mapping, where the primary cost function is the hop count. Satisfying this cost function reduces the communication energy and latency, improving the overall application performance. Despite a rich NoC mapping literature, a gap observed is the mapping implementation in an actual many-core, considering the management organization. This work has as main goals: (*i*) present the task mapping as a modular and portable management application; (*ii*) propose a dynamic mapping heuristic using a sliding window technique, which produces a low application fragmentation. Results evaluate the proposal against mapping heuristics in two management approaches: cluster-based (CBM) and per-application management (PAM). Compared to the CBM and PAM approaches, the average communication cost reduces by 27% in systems with 100 PEs. The PAM latency is 4.3 times larger than CBM and our proposal. Compared to CBM, our mapping latency presents a slight increase (3%).**

***Index Terms*—Many-core, Task Mapping, Management Application, Management Organizations.**

I. Introduction

The increased number of cores results in complex resource allocation problems. These problems are addressed by the *many-core management*. The many-core management in state-of-the-art platforms is tightly coupled (Definition 1) to the OS and hardware. Fast hardware and software evolution demands modularity (Definition 2) and portability (Definition 3) so the management strategies can evolve alongside the platforms.

Definition 1. *Coupling* is defined as the degree of interaction between modules of a system. A software is desired to have low, or loose, coupling. In the context of operating systems (OS), the coupling measures the dependency between kernel and nonkernel modules [1].

Definition 2. *Modularity*. Ability to add, modify, or remove management objectives from the many-core.

Definition 3. *Portability*. Refers of using the same management strategy, goals, and heuristics between distinct platforms while reusing the code.

The main management action executed in a many-core is the application mapping. Even with a rich literature related to task mapping [2], recent works addresses this problem. Amin et al. [3] (2020) presents a comparative analysis and categorization of application mapping approaches with current trends in NoC design implementation. Gaffour et al. [4] (2020) present a dynamic clustering mapping strategy to place all the tasks of the same application in the same region. Relevant features of this work are centralized mapping, dynamic clusters sizes, and multi-tasking. The Authors compute the cluster sizes according to the number of tasks, presenting better results than approaches using static cluster sizes. Lee et al. [5] (2021) propose an SMT (Satisfiability Modulo Theories)-based framework to find the contention-free task mapping with the minimum application schedule length.

The survey presented in [2] highlights distributed approaches since these can map more than one application in parallel and may reduce the mapping execution time by restricting the search space. However, there are two main weaknesses related to distributed mapping algorithms. The first one refers to the fact that, when restricting the search space to a particular region (e.g., a cluster), the mapping may fail due to a lack of knowledge of the other areas of the many-core. Reclustering [6] of dynamic clusters [4] increase the search space if the mapping fails, at the cost of execution time. Mapping more than one application in parallel only makes sense if the many-core can receive requests in parallel, which is usually not the case, since many-cores normally have one interface dedicated for injecting applications (e.g., network adapter, external memories).

This work has two strategic *objectives*. The first is to present the task mapping as a modular and portable management task. The second is to present a mapping heuristic using a sliding window technique, which results in low application fragmentation in large systems (100 PEs).

The proposal is centralized, allowing the mapper to have a complete many-core view, improving decision making. Despite being centralized, the mapping latency is only 3% slower than a cluster-based mapping, which is a distributed approach. Modularity and portability (Definitions 2 and 3) of the heuristic comes from its implementation loosely coupled to the OS (Definition 1), by running as a standard application. in user space.

This paper is organized as follows. Section II reviews many-core management organizations. Section III presents the main contribution of this work, the management application mapping heuristic. Section IV compares the mapping heuristic to state-of-the-art proposals. Section V concludes this paper and point-out directions for future works.

II. Many-core management organization

The many-core management organization defines *where* the management is located and *how* it is executed in a many-core. Many-cores adopts three main organization classes: centralized management [7], Cluster-Based Management (**CBM**) [8, 9], and Per Application Management (**PAM**) [10, 11].

In centralized management the many-core allocates one Processing Element (PE), called Global Manager (GM), to be the controller of all actions in a many-core. The centralized management can be quickly overloaded by answering requests from numerous PEs, also generating a considerable amount of traffic around it. The CBM approach divides the many-core into regions, named *clusters*. Besides using a GM to synchronize all the many-core management, each cluster has a Local Manager (LM). PAM is another distributed way to manage a many-core that dynamically assigns a manager for each running application.

We proposed an alternative method, named Management Application (**MA**) [12]. The MA has the following features:

- No need for dedicated cores for management execution as in centralized management, CBM, and PAM. Management tasks can share processors with user tasks.
- MA tasks are not bound to specific locations as in other paradigms and can be migrated. This possibility brings additional reliability with migration in case of violated thermal constraints or faulty cores.
- The OS has its memory footprint reduced and is easier to maintain since it is not overloaded or modified with the insertion of resource management modules.

This paper proposes a dynamic mapping algorithm for the MA management approach, and compares it with state-of-the art mapping for CBM and PAM. CBM and PAM use the mapping algorithm proposed in [13]. This heuristic consists of: (i) select one core to map the first task of the application based on the maximum average distance from other applications; (ii) starting from this core, mapping each application task based on a diamond search between the free cores. The implementation in CBM and PAM differ by the initial search space, being restricted by clusters in CBM.

III. MA Mapping heuristic

The proposed mapping heuristic, named *MAmap*, contains three phases: window selection, mapping order, and task mapping. Before executing *MAmap*, it is verified if the system can run the incoming application, i.e., if there are enough free pages[1] to map the application. A counter implements this verification, which increases with the number of application tasks to be mapped, and decreases when an application finishes.

A. Window selection algorithm

MAmap adopts the concept of *virtual clusters*. Figure 1 shows a virtual cluster, or *window*, in orange, in an 8x7 many-core, in a 2D mesh NoC topology. Therefore the window is defined by x and y coordinates of its bottom-left corner and W_x and W_y representing the window size in the x-axis and the y-axis. In Figure 1, the window x and y are located at the coordinate $(2, 2)$, and its W size is 3x3. The reasoning to adopt virtual clusters is to reduce the search space to map a given application, sliding the window similar to a convolution matrix, advancing each time by a value called *stride* (S), which in the Figure is 2.

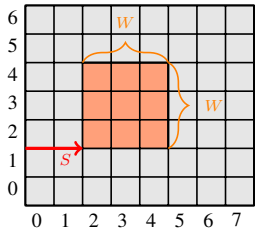

Fig. 1. A virtual cluster in the many-core.

The goal of mapping an application into a virtual window is to reduce the application fragmentation, i.e., have a small hop count between communicating tasks. The search procedure to find a window to receive the application starts from the last selected window. The reasoning for adopting this method is to avoid using the same many-core region, balancing the execution load, and in the long term, increase the system lifetime.

Algorithm 1 presents the window selection algorithm. Its inputs are the number of tasks of the application ($app.\#tasks$), the last selected window, W_x, and W_y. The algorithm always returns a window because there is a previous verification related to the availability of resources. The returned window contains the tuple $\{x, y, W_x, W_y\}$.

Algorithm 1: Window selection algorithm.

```
   Input: app.#tasks, last_window, Wx, Wy
   Output: window
1  increased_x ← false
2  while true do
3      window ← window_next (last_window, Wx, Wy, stride)
4      while window < last possible position do
5          if window_pages (window, Wx, Wy) ≥ app.#tasks then
6              return window
7          end
8          window ← window_next (last_window, Wx, Wy, stride)
9      end
10     window ← (0, 0)
11     while window ≤ last_window do
12         if window_pages (window, Wx, Wy) ≥ app.#tasks then
13             return window
14         end
15         window ← window_next (last_window, Wx, Wy, stride)
16     end
17     if increased_x then
18         Wy ← Wy + 1
19         increased_x ← false
20     else
21         Wx ← Wx + 1
22         increased_x ← true
23     end
24 end
```

Line 3 of the Algorithm advances to the next virtual window. The loop between lines 4 to 9 searches the first window with available resources to execute the application, after the selected *last_window*, returning it if it exists. The function

[1]A page is a memory region to execute a task, and each PE has a parameterizable number of pages defined at design time

`window_pages` returns the number of available memory pages in a window. If the first loop does not find a window, the same process is repeated from the first window (coordinates (0,0), line 10) up to the last window (lines 11–16).

Note that these two loops are inside a "while true" external loop. Suppose the application requires more resources than those available in the current window size, or the windows have processors executing tasks belonging to other applications. In this case, it is necessary to increase the window size.

Lines 17–23 increase W_x or W_y alternately, avoiding to increase the number of PEs in the window by a large value. The two main loops rerun after increasing the window size in one dimension. This process continues up to find a suitable window.

Figure 2 illustrates the window sliding in a 8x7 many-core. The window slides in the x-direction by adding the stride value to the current x value. After sliding in the x-direction, the y value receives the current y value plus the stride value. Note that the figure has red windows. These windows are the ones that the stride reduces to reach the boundaries of the many-core while keeping the window size.

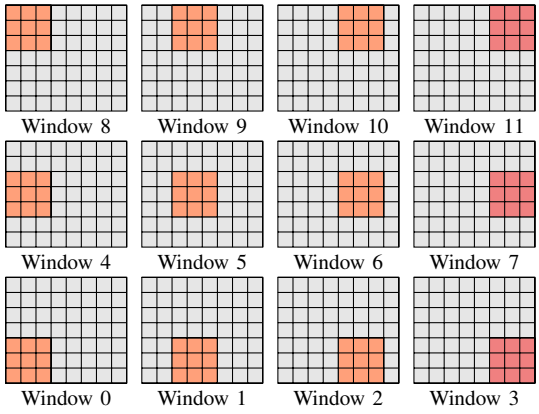

Fig. 2. Virtual window sliding in a 8x7 many-core, $W_x = 3$, $W_y = 3$, and $S = 2$.

B. Mapping order algorithm

This step is, in effect, independent of the window selection and thus could run in parallel with the first phase. The result of this phase is the mapping order used in the mapping algorithm (Section III-C). An appropriate mapping order, although not guaranteed to be optimal, reduces the mapping fragmentation.

Before detailing the mapping order algorithm, Definitions 4 to 6 detail the application model adopted by the current work.

Definition 4. *Application (App).* A directed and connected Communication Task Graph, $CTG(T, E)$, models each application. Each vertex $t_i \in T$ represents a task, and each edge $e_{ij} \in E$ represents the communication from t_i to t_j. Assuming edge e_{ij} implicitly modeled in the t_i representation, an application with N tasks is represented as:

$App = \{t_0, t_1, ..., t_{N-1}\}$

Definition 5. *Task (t_i).* A task is a vertex of the CTG. Each task t_i is a tuple with its identification, a list of successors and a list of predecessors. Successors, su_i, are tasks receiving data from t_i. Predecessors, pr_i, generate data to t_i.

$t_i = \{id, \{su_0, su_1, ...\}, \{pr_0, pr_1, ...\}\}$

Definition 6. *Initial task (in_i).* A task t_i is said to be initial, in_i, if the predecessors set is empty, i.e., there is no edge directed to it.

Figure 3 illustrates a 5-task application modeled as a CTG. In this example, t_0 is the initial task because it does not have predecessors. The t_0 successors are $\{t_1, t_2\}$, while t_4 predecessors are $\{t_2, t_3\}$.

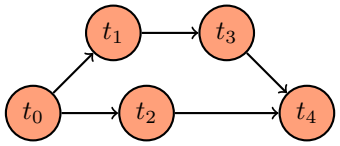

Fig. 3. Application modeled as a CTG.

Algorithm 2 details the mapping order algorithm. The loop between lines 5–9 creates the $Initials$ set, i.e., a set with all application initial tasks. Note that an application may have cyclic dependencies, making this set empty. For example, this may happen in Figure 3 if t_4 sends data to t_0. Lines 10-11 adds the first application task, t_0, to the *Initials* set when this situation happens.

Algorithm 2: Task mapping ordering algorithm.

```
   Input: App // Definition 4
   Output: Order
1  Order ← ∅
2  Initials ← ∅ // Definition 6
3  task_count ← 0
4  inserted ← 0
5  foreach t_i in App do
6      if t_i.Predecessor is ∅ then
7          Initials.insert(t_i)
8      end
9  end
10 if Initials is ∅ then
11     Initials.insert(t_0)
12 end
13 foreach in_i in initials do
14     Order[inserted++] ← in_i
15     while task_count < inserted do
16         foreach su_i in Order[task_count].Successors do
17             if su_i not in Order then
18                 Order[inserted++] ← su_i
19             end
20         end
21         task_count++
22     end
23 end
24 return Order
```

The next loop, lines 13-23, acts as a breadth-first search algorithm to traverse the CTG. At line 14, the Order set receives an initial task, in_i. Next, lines 16–20 add all non-added successors of in_i into the Order set. At line 21, the counter $task\_count$ increases, making the loop 16–20 to add the successors of the second element in the Order set into the Order set.

Table I illustrates how the application is traversed using the CTG depicted in Figure 3. The algorithm adds the initial

task, t_0, at line 14. Next, the first iteration of lines 16-20 adds the successors of the first Order element to the Order set. The counter $inserted$ is now equal to three, meaning that there are 3 elements in the Order set. At line 21, $task\_count$ increments, moving the traversal index for the next iteration. At the end of the third iteration, all tasks are in the Order set ($inserted = 5$). The loop 16-20 repeats twice, increasing $task\_count$ up to be possible to exit the loop. This process is repeated for each initial task. The resulting mapping order is t_0, t_1, t_2, t_3, t_4.

TABLE I
EXECUTION OF THE MAPPING ORDER ALGORITHM, USING FIGURE 3 AS INPUT.

Iter.	Lines	Order	task_count	inserted	remark
	1-9	$\emptyset$	0	0	initial task: t_0
	14	t_0		1	
1	16-20 21	t_0, t_1, t_2	1	3	t_0 successors
2	16-20 21	t_0, t_1, t_2, t_3	2	4	t_1 successors
3	16-20 21	t_0, t_1, t_2, t_3, t_4	3	5	t_2 successors
	21	t_0, t_1, t_2, t_3, t_4	4,5		repeats 16-20 twice, exiting

C. Task mapping algorithm

The main cost function of this phase is the hop count reduction. Satisfying this cost function reduces the communication energy and latency [14], improving the overall applications performance. This phase uses the results of the two previous phases, the *window*, and the *Order* set. The *window* reduces the mapping complexity due to the limited search space. The *Order* set defines the sequence to map tasks to minimize the communication cost.

The designer can tune the mapping cost-function using two parameters:

- `COST_DIFF_APP`: cost related to tasks not belonging to the application being mapped running in the PE under evaluation. This cost prevents PE sharing among different applications.
- `COST_SAME_APP`: cost related to tasks of the application being mapped running in the PE under evaluation. This value defines CPU sharing. A large values distributes the tasks in several PEs, increasing the application performance, while small values increase the CPU sharing, reducing the number of resources used by the application.

Algorithm 3 details the mapping algorithm. The external loop (lines 2–24) maps the tasks sequentially, according to the *Order* set. The algorithm creates, at line 5, the *Neighbor* set with all tasks communicating with t_i.

The loop between lines 6–22 evaluates all PEs in the *window*, with available resources to receive tasks. The functions `n_tasks_diff_app` and `n_tasks_same_app` get the number of tasks in the PE running different and same applications as the task to map, respectively, multiplying by the costs. Next, lines 10–15 evaluates the communication cost between the t_i and its neighbor tasks already mapped. Finally, the lines 16–20, select the PE with the smaller cost. The last step executed by the algorithm, line 23, is to add the PE address to the Mapping set.

Algorithm 3: Task mapping algorithm.

```
   Input: Order, window
   Output: Mapping
1  Mapping ← ∅
2  foreach t_i in Order do
3      cost ← ∞
4      selected_PE ← None
5      Neighbors ← t_i.Predecessors ∪ t_i.Successors
6      foreach PE_xy ∈ window do
7          if PE_xy.pages > 0 then
8              diff_app_cost ← n_tasks_diff_app(PE_xy, t_i)
                 * COST_DIFF_APP
9              same_app_cost ← n_tasks_same_app(PE_xy,
                 t_i) * COST_SAME_APP
10             comm_cost ← 0
11             foreach t_c in Neighbors do
12                 if t_c is mapped then
13                     comm_cost ← comm_cost +
                         manhattan_distance(t_i, t_c)
14                 end
15             end
16             c ← diff_app_cost + same_app_cost + comm_cost
17             if c < cost then
18                 cost ← c
19                 selected_PE ← PE_xy
20             end
21         end
22     end
23     Mapping[t_i.id] ← selected_PE
24 end
25 return Mapping
```

The current implementation of the mapping algorithm evaluates the PEs first in the y-direction and then in the x-direction (line 6 of Algorithm 3). The reasoning for this order comes from the stride parameter, which overlaps windows in the x-direction. Figure 4 shows step-by step the mapping of the CTG presented in Figure 3, considering one memory page per PE. The average communication cost of this mapping is 1.2 (6 hops divided by 5 edges).

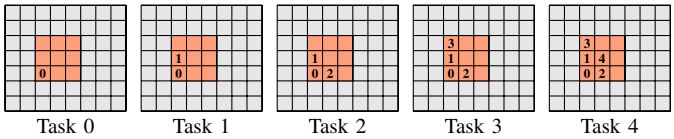

Fig. 4. Execution of the mapping algorithm related to CTG presented in Figure 3, for $W_x = 3$ and $W_y = 3$.

IV. RESULTS

A. Functional validation and computational complexity

A Python implementation enabled the initial functional validation. This implementation generates logs readable by the many-core debugger [15], allowing debugging and algorithmic optimizations.

Figure 5 displays the window of the many-core debugger, showing the application mapping for a scenario in an 6x6 many-core, with two memory pages per PE. This scenario contains eleven different applications, each one marked by a different color in the Figure. The mapping generated contiguous regions for the applications, minimizing fragmentation.

Table II presents the computational cost of each *MAmap* phase. For the window selection phase, the average complexity

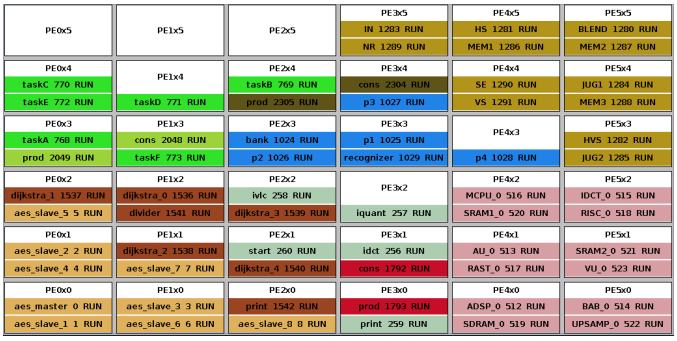

Fig. 5. *MAmap* validation on a 6x6 many-core with multitasking, $W_x = 3$, $W_y = 3$, and $S = 2$.

is $O(W^2)$, being W the window size (maximum between W_x and W_y). When the system load is high, and most PEs execute tasks, the algorithm needs to increase the window size, which may grow up to the system size. Thus, in the worst-case, the complexity is $O(N^3)$ (where N is the PE number in one dimension).

TABLE II
COMPLEXITY OF EACH PHASE OF THE HEURISTIC.

	Window Selection	Mapping Order	Task Mapping
Average Case	$O(W^2)$	$O(t)$	$O(W^2 \times t)$
Worst-Case	$O(N^3)$		$O(N^2 \times t^2)$

The third column in Table II shows the complexity of the mapping order phase. The complexity of this phase is $O(t)$, where t is the number of tasks of the application, since it verifies all tasks once.

The fourth column in Table II shows the complexity of the mapping phase. The average complexity is $O(W^2 \times t)$. The average case arises when the selected window has the initial size, and each task only has a few successors or predecessors. This phase searches for all PEs in the window for each task of the application. When the window grows to the many-core size, its complexity can rise to $O(N^2 \times t)$. When all tasks communicate with all other tasks, the worst-case can reach $O(N^2 \times t^2)$ due to each task computing the communication cost for all other tasks in the application.

B. Mapping quality

The three managements organizations were implemented using the Memphis many-core [15] as baseline platform. The experiments use a 10x10 many-core, with each PE supporting a single task. The CBM is divided into four 5x5 clusters. The PAM is also divided into the same clusters with hierarchical management. All evaluated scenarios, **14**, contains the same set of **9** applications, each one with a different number of tasks (Table III), with a total of **78** tasks, occupying 78% of the memory pages in the many-core. What differentiates the 14 evaluated scenarios is the order in which the applications enter the system. One scenario maps applications from the smaller to largest number of tasks, one from the largest to the smaller number of tasks, one balanced, and 11 random scenarios.

The performance figure used to evaluate a mapping result is its mapping average communication cost, $comm\_cost$. The

TABLE III
NUMBER OF APPLICATIONS TASKS.

Application	#tasks	Application	#tasks
JPEG	5	AES	9
DTW	6	MWD	12
Dijkstra	6	VOPD	12
Matrix multip.	6	Sorting	15
MPEG4	7	**Total tasks**	**78**

average communication cost of an application modeled as a CTG is the total number of hops between communicating pairs (Manhattan distance) divided by the number of communicating pairs (graph edges). The $comm\_cost$ is the average cost between the mapped applications.

Figure 6 shows the mapping of scenario with a balanced application arrival order in terms of task count. CBM had a $comm\_cost = 2.27$ hops. The CBM mapping produced mostly contiguous mapping, except for the MWD application, in blue, which needed reclustering to fit in the many-core. The PAM mapping solved this issued by allowing a search space equal to the system size. In addition to the 4 managers of the CBM, the hierarchical PAM needs one extra manager per application, resulting in an overhead of 13% of lost mapping space. Besides searching the entire many-core for the best location to map, PAM reduced the $comm\_cost$ by just 1.32%, resulting in a $comm\_cost = 2.24$ hops. The *MAmap* overhead is a single memory page, representing 1% of the available PEs. Moreover, the average communication cost dropped by 21% compared to CBM, with a $comm\_cost = 1.77$ hops, due to the algorithm considering a restricted search space and a mapping order that considers the communicating task pairs.

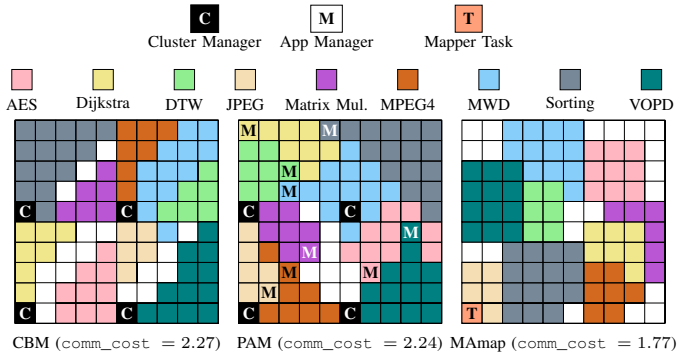

Fig. 6. Mapping comparison between CBM, PAM, and *MAmap*. *MAMap* uses $W_x = 3$, $W_y = 3$, and $S = 2$.

The analysis of the 14 scenarios showed that on average, PAM and CBM produced the same average $comm\_cost =$ **2.33** hops, indicating that clustering does not penalize the mapping. The standard deviation of 0.07 in CBM and 0.11 in PAM shows that both approaches have low fragmentation independent of the application order, indicating that the algorithm keeps the communication distance close to the average for all scenarios. For *MAmap*, the average $comm\_cost =$ **1.71** hops, is 27% smaller than CBM and PAM, for the 14 scenarios. A small standard deviation, 0.06, is observed, confirming that

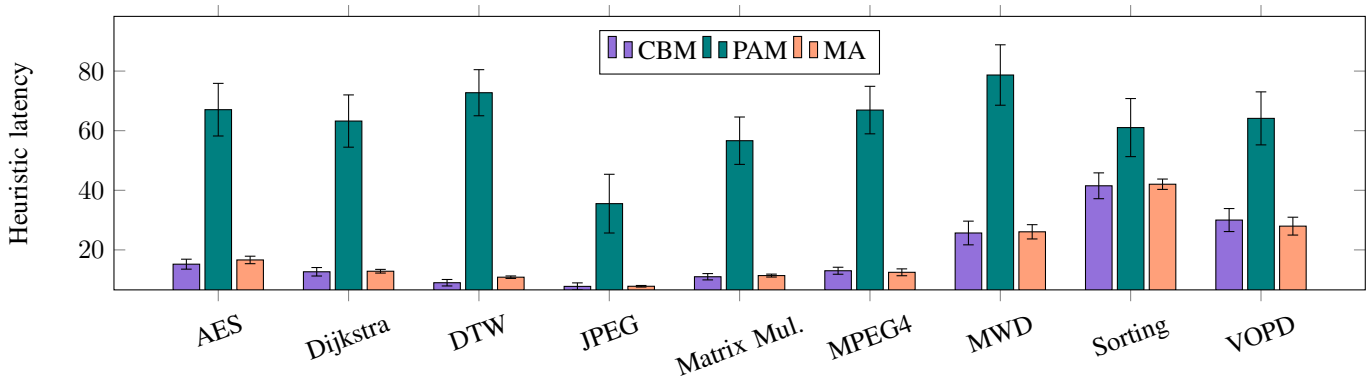

Fig. 7. Mapping latency (Kclock-cycles) for different sized applications in CBM, PAM and *MAmap*.

the *MAmap* is also independent of the order and the size of the applications entering the system.

Figure 7 shows a graph of the average mapping latency in kilo clock cycles (Kcycles) for each management paradigm. Each bar represents the average heuristic latency from all evaluated scenarios with the lines marking the standard error. The latency is measured from the start to the end of the heuristic in the manager processor (CBM, PAM) or in the mapper application (*MAmap*).

In Figure 7, the heuristic used by CBM and PAM is the same, but PAM has the disadvantage of searching the whole many-core for the initial PE, resulting in average 4.3 times higher latency. *MAmap* shows similar results than CBM, being in average just 3% slower despite being a centralized heuristic. Additionally, it is important to note that CBM runs directly at kernel level, which incurs in less execution time overhead compared to *MAmap*, which runs at the application level.

The average standard error showed by *MAmap* (1.26) is lower than CBM (2.2) and PAM (8.87). The reason explaining this result is twofold: (a) *MAmap* is centralized, thus does not need to spend time synchronizing the system status with managers, as CBM does and PAM does more heavily; (b) the PE running *MAmap* serves the single purpose of mapping, being more available than the CBM manager that runs another management goals in the same software.

V. Conclusion

This paper presented a new mapping technique, which despite being centralized, has a similar execution time to distributed implementations and presents a better cost related to communication between tasks. Another differential of the proposal is its implementation loosely coupled to the OS, which allows porting it to other many-cores.

Future works include evaluating fragmentation after insertion and removal of multiple applications, implementing fragmentation detection techniques, and automatic implementation of defragmentation through task migration.

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] L. Yu, S. R. Schach, K. Chen, and J. Offutt, "Categorization of Common Coupling and its Application to the Maintainability of the Linux Kernel," *IEE Transactions on Software Engineering*, vol. 30, no. 10, pp. 694–706, 2004, https://doi.org/10.1109/TSE.2004.58.

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

[3] W. Amin, F. Hussain, S. Anjum, S. Khan, N. K. Baloch, Z. Nain, and S. W. Kim, "Performance Evaluation of Application Mapping Approaches for Network-on-Chip Designs," *IEEE Access*, vol. 8, pp. 63 607–63 631, 2020, https://doi.org/10.1109/ACCESS.2020.2982675.

[4] K. Gaffour *et al.*, "Dynamic Clustering Approach for Run-time Applications Mapping on NoC-based multi/many-core systems," in *EDiS*, 2020, pp. 15–20, https://doi.org/10.1109/EDiS49545.2020.9296439.

[5] D. Lee, B. Lin, and C.-K. Cheng, "SMT-based Contention-Free Task Mapping and Scheduling on SMART NoC," *IEEE Embedded Systems Letters*, pp. 1–1, 2021, https://doi.org/10.1109/LES.2021.3049774.

[6] 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, https://doi.org/10.1109/ISVLSI.2013.6654651.

[7] A. M. Rahmani *et al.*, "Reliability-Aware Runtime Power Management for Many-Core Systems in the Dark Silicon Era," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 25, no. 2, pp. 427–440, 2017, https://doi.org/10.1109/TVLSI.2016.2591798.

[8] 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, https://doi.org/10.1145/1391469.1391664.

[9] 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, https://doi.org/10.1109/ACCESS.2019.2937730.

[10] 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, https://doi.org/10.1145/2463209.2488942.

[11] 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, https://doi.org/10.1145/2039370.2039392.

[12] M. Ruaro, A. Santana, A. Jantsch, and F. G. Moraes, "Modular and Distributed Management of Manycore SoCs," *ACM Transactions on Computer Systems (TOCS)*, vol. 38, no. 1-2, pp. 1–16, 2021, https://doi.org/10.1145/3458511.

[13] V. Tsoutsouras, S. Xydis, and D. Soudris, "Application-Arrival Rate Aware Distributed Run-Time Resource Management for Many-Core Computing Platforms," *IEEE Transactions on Multi-Scale Computing Systems*, vol. 4, no. 3, pp. 285–298, 2018, https://doi.org/10.1109/TMSCS.2018.2793189.

[14] J. C. S. Palma *et al.*, "Mapping Embedded Systems onto NoCs: the Traffic Effect on Dynamic Energy Estimation," in *SBCCI*, 2005, pp. 196–201, https://doi.org/10.1109/SBCCI.2005.4286856.

[15] 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, https://doi.org/10.1007/s10617-019-09223-4.