Leveraging NoC-based Many-core Performance Through Runtime Mapping Defragmentation

Angelo Elias Dalzotto, Caroline da Silva Borges, Marcelo Ruaro, Fernando Gehm Moraes
School of Technology, Pontifical Catholic University of Rio Grande do Sul – PUCRS – Porto Alegre, Brazil
{angelo.dalzotto, marcelo.ruaro, caroline.s}@edu.pucrs.br, fernando.moraes@pucrs.br

***Abstract*—Memory fragmentation occurs when non-continuous areas are allocated, requiring the adoption of memory defragmentation policies. Likewise, in NoC-based many-core systems, processing elements (PEs) that communicate with each other may be located in non-contiguous areas, characterizing PE fragmentation. The literature approaching defragmentation often acts on fragmented regions instead of fragmented applications, generating unnecessary task migrations. We propose a reactive and fine-grain defragmentation method. Two sets of experiments demonstrate the effectiveness of the proposal. The first evaluates the defragmentation in an 8x8 system with all PEs executing tasks. The communication cost starts at 2.11 (average hop distance), reaching 1.13 after defragmentation. The second one evaluates the execution time of applications in an actual many-core, showing that fragmentation penalizes the execution time of applications. By applying the defragmentation heuristic, the execution time overhead reduces from 10.6% to 4.2% for an AES benchmark, considering the execution time of task migrations.**

***Index Terms*—Runtime defragmentation, Application Performance, NoC-based many-core.**

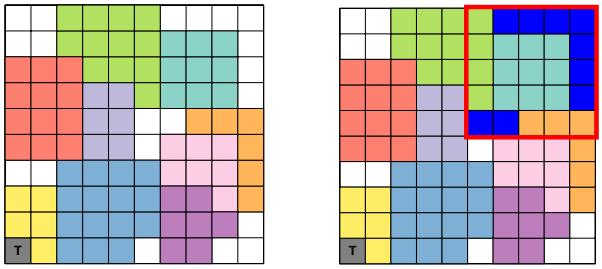
I. Introduction and Related Work

In NoC-based many-core systems, processing elements (PEs) that communicate with each other may be located in non-contiguous areas, characterizing PE fragmentation. PE fragmentation occurs in dynamic workloads, i.e., applications entering and leaving the system at any moment. Such a process leads to fragmentation in the application mapping since the availability of free cores usually will be scattered through non-contiguous regions [1]. This is observed in Figure 1. Scenario (a) shows a 10x10 system, with 80% of the PEs used to execute applications. In scenario (b), a new application is admitted to the system, and due to the lack of contiguous resources, its mapping becomes fragmented.

The adverse effects of a fragmented mapping in a Network-on-Chip (NoC)-based many-core include:

- Degradation of the fragmented application performance due to increased hop count;
- Interference in the performance in other applications due to the new traffic crossing regions with already mapped applications.

Ng et al. [2] identify scattered free cores and migrate mapped applications when the scattering reaches a certain threshold. This way, a contiguous region with free cores is created before the next application arrival. Their defragmentation algorithm computes a central free core with the minimum Manhattan distance to other free cores to obtain a free core area. Afterward, each free core is migrated one hop at a time, i.e., the running tasks are shifted one by one to minimize their communication cost penalty after a migration of free cores. Their algorithm relies on predicting the number of tasks of the next incoming application and assumes that non-fragmented applications can increase their communication cost by migrating tasks to reduce system fragmentation.



(a) (b)

Fig. 1. (a) Mapping in a 10x10 system, with high resource usage (80%); (b) Admission of a new application (dark blue), resulting in fragmentation. The red box highlights that it disturbs the orange, green, and turquoise applications. **T**: mapping task.

Pathania et al. [3] also defragment a many-core prior to the application mapping, but in a cached system, that induces cold cache misses when migrations occur. Their application mapping demands several constraints to its shape, such as being limited to using a power of two cores and limiting the mapping space to polyominoes. This way, the number of free cores in a contiguous area will also be a power of two. To defragment the mapping proposed in [3], an area with free cores, called *fragment*, can be joined with another of the same size to be suitable for the next incoming application. The migrations involved in this process occur by swapping the fragment with tasks from a running application instead of sliding the application as in [2]. Despite that, their proposal is only suited to applications that fit their constraints or have the ability to scale its number of workers to comply with them. They also enable migrations of non-fragmented applications, as in [2].

Modarressi et al. [4] propose a defragmentation that can migrate all tasks of a fragmented application. Their work aims to reduce the communication energy on a circuit-switched many-core. It is also stated that larger applications benefit more from task migrations due to the likelihood of being mapped in non-contiguous regions. On average, each migration step moved a task by 2.69 hops. Their work assumes that the communication volume between communicating tasks changes at runtime. The defragmentation heuristic executes when a given application ends its execution. The selected fragmented application is remapped using the updated communication volume between tasks. This procedure may migrate all tasks of the fragmented

application. This method differs from our proposal since we defragment the system when a task finishes, migrating only one task, not remapping the application.

The presented proposals seek to solve two problems to defragment the many-core: (*i*) *what* task to migrate; and (*ii*) *where* to migrate such task. The first problem, selecting the task to migrate, is solved by our heuristic using the cost functions detailed in Section II. The second problem is not an issue in our proposal since the task that finished its execution is at the location that will receive the migrated task. Note that all the presented algorithms are evaluated using NoC emulators while we evaluate the heuristics in an actual many-core.

The *goal* of the present work is to present a reactive and fine-grain defragmentation method. Reactive because the heuristic executes when a given task finishes, not when there are fragmented regions in the system. Fine-grain because it acts at the task level and not at the application level. This heuristic adopts two metrics to evaluate if an application is fragmented: communication cost and interference in other applications. Despite using the mapping algorithm presented in [5], the proposal may use different mapping algorithms as reference.

The *original* contributions of our proposal include:

- The defragmentation acts on fragmented applications instead of unallocated areas, reducing the number of migrations;
- There is no threshold to trigger the defragmentation process;
- The defragmentation is a fine-grain process, i.e., for each finished task, the proposal evaluates if it is possible to use the freed resource to improve the mapping quality of another application.

II. Defragmentation Heuristic

Definition 1 and Definition 2 detail the application model adopted by the current work.

Definition 1. Application (App) – is a directed and connected Communication Task Graph $CTG(T, E)$ that models each application. Each vertex $t_i \in T$ represents a task, and each edge $e_{ij} \in E$ represents communication from t_i to t_j. Assuming that edges e_{ij} are modeled implicitly in t_i (see Definition 2), an application is represented as: $App = \{t_1, t_2, ..., t_{|T|}\}$.

Definition 2. Task (t_i) – is a vertex of the CTG. Each task t_i is a tuple with its identification and a list of communicating tasks connected by its edges e_{ij}. Communicating tasks are divided into successors and predecessors. Successors $su_n \in S_i$ are tasks receiving data from t_i. Predecessors $pr_n \in P_i$ generate data to t_i. Therefore, a task is represented as: $t_i = \{id, \{su_1, su_2, ..., su_{|S_i|}\}, \{pr_1, pr_2, ..., pr_{|P_i|}\}\}$.

Figure 2 illustrates an application modeled as a CTG. It contains 5 tasks and 5 edges corresponding to communicating pairs. Task t_1 has successors t_2 and t_3, task t_5 has predecessors t_3 and t_4, while the remaining tasks have only one successor and one predecessor.

The defragmentation heuristic adopts two metrics: (*i*) communication cost (Definition 3); and (*ii*) bounding box (Definition 4).

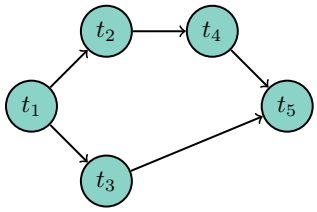


Fig. 2. Example CTG $App = \{t_1, t_2, t_3, t_4, t_4\}$. $t_1 = \{1, \{t_2, t_3\}, \{\}\}$; $t_2 = \{2, \{t_4\}, \{t_1\}\}$; $t_3 = \{3, \{t_5\}, \{t_1\}\}$; $t_4 = \{4, \{t_2\}, \{t_5\}\}$; $t_5 = \{5, \{\}, \{t_3, t_4\}\}$.

Definition 3. Communication cost (cc) – is the total number of hops between communicating pairs using a Manhattan distance (d), divided by the number of communicating pairs, corresponding to the CTG edges. The cc of an App is computed using the application CTG, while the cc of its tasks t_i is computed using the edges connected to its vertex.

Equation (1) defines the communication cost for an application. It is obtained by the average of the Manhattan distance from all vertices (tasks) to all its successors (edges directed from the vertex to another one). Therefore, it is the average Manhattan distance of all edges of an application.

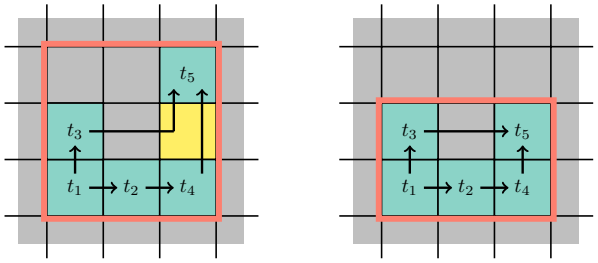
$$cc(App) = \frac{\sum_{i=1}^{|T|} \sum_{n=1}^{|Si|} d(t_i, t_i.su_n)}{\mid E \mid} \quad (1)$$

Equation (2) defines the communication cost computation for one task of an application. It is obtained by considering only the edges connected to the vertex in which the communication cost is computed. Therefore, it sums the Manhattan distance of t_i to all its predecessors and successors and obtains the average.

$$cc(t_i) = \frac{\sum_{n=1}^{|Si|} d(t_i, t_i.su_n) + \sum_{n=1}^{|Pi|} d(t_i, t_i.pr_n)}{\mid Si \mid + \mid Pi \mid} \quad (2)$$

Definition 4. Bounding box (BB) – is the minimum rectangular-shaped area comprising a set of PEs in which the application is mapped.

Figure 3 illustrates the mapping of the CTG presented in Figure 2, in a zoomed 3x3 area of the many-core. In Figure 3(a), t_5 is fragmented, i.e., it is not contiguously mapped to the remaining tasks. This is also indicated by the higher communication cost of t_5 in relation to other tasks of the same application, as $cc(t_1) = 1$, $cc(t_2) = 1$, $cc(t_3) = 2$, $cc(t_4) = 1.5$, and $cc(t_5) = 2.5$.



(a) Fragmented (b) Defragmented

Fig. 3. Application mapping in an area of a many-core, highlighting its bounding box and the communication assuming XY routing.

Assuming that the PE highlighted in yellow in Figure 3(a) becomes available due to the end of the task running on it, the defragmentation heuristic picks the task with the highest cc of the fragmented application to move to the freed PE. Thus, t_5 is migrated to the highlighted PE by the defragmentation heuristic, given the reduction in the communication cost. Figure 3(b) shows this scenario, where the cc of t_5 and its communicating pairs are updated to $cc(t_3) = 1.5$, $cc(t_4) = 1$, and $cc(t_5) = 1.5$, representing a reduction of 40% in the communication cost of the previously fragmented task.

Equation (3) presents the original communication cost of the fragmented application, while Equation (4) presents the communication cost of the same application when defragmented. In this example scenario, the communication cost of the application is reduced by 25%.

$$cc(a) = \frac{d(t_1,t_2) + d(t_1,t_3) + d(t_2,t_4) + d(t_3,t_5) + d(t_4,t_5)}{5} = \frac{1+1+1+3+2}{5} = \frac{8}{5} = 1.6 \quad (3)$$

$$cc(b) = \frac{d(t_1,t_2) + d(t_1,t_3) + d(t_2,t_4) + d(t_3,t_5) + d(t_4,t_5)}{5} = \frac{1+1+1+2+1}{5} = \frac{6}{5} = 1.2 \quad (4)$$

The operating system of each PE notifies the mapping task (**T** in Figure 1) when a given task finishes its execution. The end of a given task frees up resources in this PE, which can be used to reduce the communication cost of other applications. The mapping task executes Algorithm 1 when it receives a task termination notification in a given PE.

Algorithm 1 details the defragmentation heuristic. It has two inputs: $freed\_pe$ and $Applications$. When a given task finishes its execution, it frees a memory segment in the allocated PE, called $freed\_pe$. $Applications$ are sorted in the reverse order by their communication cost into the $SortedApps$ set (line 1) to initially evaluate the most fragmented applications, i.e., with the higher communication cost.

Algorithm 1: Defragmentation algorithm.

```
   Input: freed_pe, Applications
1  SortedApps ← sort(Applications)
2  foreach a_i in SortedApps do
3      if freed_pe ∈ a_i.bounding_box then
4          SortedTasks ← sort(a_i.tasks)
5          foreach t_i ∈ SortedTasks do
6              if compute_cost(t_i, freed_pe) <
                 compute_cost(t_i, current_pe) then
7                  migrate(t_i, freed_pe)
8                  return
9              end
10         end
11     end
12 end
```

After creating the $SortedApps$ set, a loop starts looking for the candidate application to have a migrated task. The initial condition to evaluate a given application is if the freed position ($freed\_pe$) is inside the application bounding box (line 3). If this condition is satisfied, a set named $SortedTasks$ orders the application tasks according to the hop distance between its predecessors and successors (line 4). Thus, tasks with a higher hop count are prioritized to migrate.

The loop between lines 5-10 evaluates the cost to migrate t_i from its current position to the freed PE, according to the communication cost between the t_i and its successors and predecessor tasks. If the new cost is lower than the previous one, the task can migrate. When migration occurs, the algorithm finishes. Thus, only a single migration can occur per finished task. Note that other tasks from the same application may also exit the system when a given task is completed. Thus, an application leaving the system can trigger multiple migrations.

Note that Algorithm 1 does not restrict the migration to fragmented scenarios. When an application terminates, migrations can occur to enhance another application mapping, even if it is not fragmented. Thus, another benefit of this method is migrating tasks to locations where the mapping quality is better, translated by its lower mapping cost.

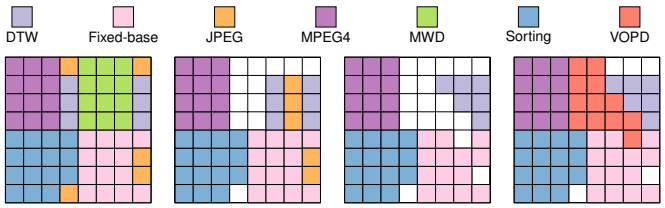
III. Defragmentation Results

Section III-A evaluates the communication cost and average communication cost of a set of applications considering the system load. Section III-B evaluates the effect of the fragmentation on the execution time of applications in an actual NoC-based many-core [6], modeled at the RTL level, with a dedicated multitasking operating system.

A. Communication cost

Experiments are conducted on a 8x8 many-core, with each PE executing a single task. The mapper task executes at the same PE as an application task, being the only PE running two tasks simultaneously. A set of 6 parallel applications with different communication models and task numbers is used.

Figure 4 presents four mapping states of the experiment. The first state, shown in Figure 4(a), is the initial state of the experiment. Initially, the many-core is loaded with six applications, using 100% of the mapping space. In this case, applications DTW, in violet, and JPEG, in orange, are fragmented. The initial average application $cc = 2.11$.



(a) $cc = 2.11$ (b) $cc = 1.87$ (c) $cc = 1.56$ (d) $cc = 1.13$

Fig. 4. Mapping states before defragmentation (a), after defragmentation (b,c), and with a new application mappped after defragmentation (d).

The first set of task migrations occurs when MWD ends its execution – Figure 4(b). Applications DTW and JPEG achieve better communication cost through 6 migrations. Although both applications are still fragmented, DTW reduced its communication cost from 3 hops to 2 hops, while JPEG

reduced from 3.75 hops to 2.5 hops. Overall, both migrated applications had their communication cost reduced by 33%, resulting in a $cc = 1.87$, 11.4% smaller than the initial value.

Figure 4(c) shows the mapping when JPEG ends and after 3 desfragmentation rounds, corresponding to two migrations for DTW and one for Fixed-base. DTW had its communication cost reduced by 40%, resulting in 1.5 hops. Note that despite not being fragmented, Fixed-based also had one migration that enhanced its communication cost, reducing by 6.2%, from 1.78 hops to 1.67 hops. The average cc reduced to 1.56, 16.6% smaller.

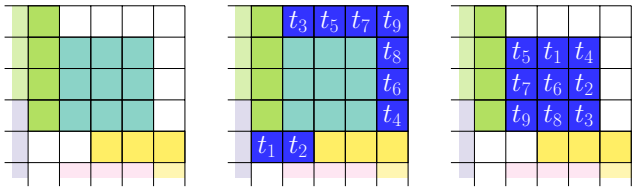
Figure 4(d) shows the mapping of a new application, VOPD. The VOPD is mapped in a system without fragmentation, resulting in a $cc = 1.13$, with a system load equal to 92.2%.

This experiment shows the main advantage of using a runtime defragmentation heuristic. As tasks finish their execution, the defragmentation reorganizes the system to generate contiguous areas for new applications.

B. Application performance

The communication cost impacts the energy consumed in the NoC and the average latency for sending messages [7]. But what is the impact on the application performance due to task fragmentation? This section seeks to answer this question. Applications run on the Memphis NoC-based many-core [6].

We evaluate the execution time of two instances of an AES application (AES1 and AES2), using a master-slave model, with tasks $t_{i>1}$ communicating with t_1. Three scenarios are evaluated: (1) AES1 running without fragmentation (turquoise color in Figure 5(a)); (2) AES2 mapped in a fragmented way due to the lack of available PEs (dark blue in Figure 5(b)), impacting the performance of both AES instances; (3) AES2 defragmented after the end of AES1 (Figure 5(c)). Note that in Figure 5 the defragmentation is showed when AES1 finished for sake of simplicity. In fact, Algorithm 1 executes when every AES1 task ends.



(a) Initial mapping (b) Fragmented (c) Defragmented

Fig. 5. Application mapping in a 5x5 area of the many-core. (a) initial mapping with AES1 executing without disturbing traffic (turquoise); (b) fragmented AES2 mapping (dark blue); (c) AES2 defragmentation after the end of AES1.

Table I presents the execution time. The first scenario is the baseline execution time (11.97 ms), where the AES application executes without the presence of disturbing traffic. In the fragmented scenario, both applications have the execution time penalized (AES1: 7.9% and AES2: 10.6%). Note that AES1 mapping is not fragmented, but due to the disturbing traffic, its performance is penalized. The last table row presents the AES2 execution time after executing Algorithm 1. As we can observe, the execution time penalty was reduced to 4.2%, despite the cost of the task migration heuristic.

TABLE I
APPLICATION EXECUTION TIME IN DIFFERENT SCENARIOS (ms).

	AES1	AES2
AES1 running without disturbing traffic	11.97	
Fragmented scenario – Figure 5(a)	12.91	13.24
AES2 defragmented – Figure 5(b)		12.47

The defragmentation algorithm does not impact the applications performance since it executes in parallel to them. The average task migration time is 140 μs, with the hop distance having a negligible impact on the migration time.

Note that task migration execution time is small compared to the AES execution time – roughly 1.2%. However, task migration may hinder the defragmentation benefits if the defragmentation heuristic executes migrations frequently, in non-fragmented applications, as in [2], [3], without the control proposed in this work.

IV. Conclusion

We showed the adverse effects of a fragmented mapping in NoC-based many-cores by evaluating the applications execution time. A fragmented mapping degrades the performance of fragmented and non-fragmented applications and increases the consumed energy due to the longer execution time.

The proposed defragmentation heuristic is triggered when a task finishes its execution. This event-driven proposal dismisses the need to control system utilization thresholds and limits the number of task migrations.

Therefore, defragmentation heuristics should be part of mapping heuristics for many-cores, in the same way that operating systems manage memory to avoid memory fragmentation.

Acknowledgment

This work was financed in part by Coordenação de Aperfeiçoamento de Pessoal de Nível Superior – CAPES (Finance Code 001), CNPq (grant 309605/2020-2), and FAPERGS (grant 21/2551-0002047-4).

References

[1] B. Li, X. Wang, A. K. Singh, and T. S. T. Mak, "On Runtime Communication and Thermal-Aware Application Mapping and Defragmentation in 3D NoC Systems," *IEEE Trans. Parallel Distributed Syst.*, vol. 30, no. 12, pp. 2775–2789, 2019.

[2] J. Ng, X. Wang, A. K. Singh, and T. Mak, "Defragmentation for Efficient Runtime Resource Management in NoC-Based Many-Core Systems," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 11, pp. 3359–3372, 2016.

[3] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Defragmentation of Tasks in Many-Core Architecture," *ACM Trans. on Architecture and Code Optimization*, vol. 14, no. 1, pp. 1–21, 2017.

[4] M. Modarressi, M. Asadinia, and H. Sarbazi-Azad, "Using task migration to improve non-contiguous processor allocation in NoC-based CMPs," *Elsevier J. of Systems Architecture*, vol. 59, no. 7, pp. 468–481, 2013.

[5] A. E. Dalzotto, M. Ruaro, L. V. Erthal, and F. G. Moraes, "Dynamic Mapping for Many-cores using Management Application Organization," in *ICECS*, 2021, pp. 1–6.

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

[7] J. C. S. Palma, C. A. M. Marcon, F. G. Moraes, N. L. V. Calazans, R. A. L. Reis, and A. A. Susin, "Mapping Embedded Systems onto NoCs: the Traffic Effect on Dynamic Energy Estimation," in *SBCCI*, 2005, pp. 196–201.