Exploring Resource Mapping Policies for Dynamic Clustering on NoC-based MPSoCs

Gustavo Girão, Thiago Santini, Flávio R. Wagner
Federal University of Rio Grande do Sul
Institute of Informatics - Porto Alegre, Brazil
{ggbsilva, tcsantini, flavio}@inf.ufrgs.br

***Abstract*—The dramatic increase in the number of processors, memories and other components in the same chip calls for resource-aware mechanisms to improve performance. This paper proposes four different resource mapping policies for NoC-based MPSoCs that leverage on distinct aspects of the parallel nature of the applications and on architecture constraints, such as off-chip memory latency. Results show that the use of these policies can improve performance up to 22.5% in average, and, in some cases, depending on the parallel programming model of each application, the improvement may reach up to 32%.**

I. INTRODUCTION

Essentially, there are two types of parallel programming models: shared variables and message passing. Each of these models is more suitable for a specific memory organization. Shared variables are used in a shared memory organization, since this model relies on communication through reads and writes on variables that can be accessed by any processor. On the other hand, message passing uses explicit communication through primitives for sending and receiving data. Message passing is largely used in a distributed memory organization.

There are significant differences on the use of each of these models, and it is usually the programmer that, based on his/her expertise, decides which one to use, for instance considering which one would be more computationally efficient or more easily programmed for a particular application. The main differences are the programmability (i.e. how easy it is to develop an application) and scalability (i.e. how does the system performance degrades when adding components) [1].

The rapid increase in the number of cores in a single chip requires programming techniques that make adequate use of these resources. These are solutions that cover aspects of both parallel programming and task allocation and migration. Considering that most existing parallel applications would inevitably be developed using one of these models, it is desirable for an MPSoC platform to support both of them. In a previous work [6], it has been shown that the use of different parallel programming models in distinct memory organizations leads to different performance and energy results for the same application. Furthermore, distinct memory organizations impact directly the costs of task migration in different ways.

With this knowledge, this paper proposes four resource mapping policies for NoC-based MPSoCs to take advantage of the different impact that workload and task migration cost may have over clusters with different parallel programming models. Results show that each of these policies affects the overall performance in different ways and, in average, the performance improvement reaches 22.5%, but, in some cases, may reach up to 32%.

II. DYNAMIC CLUSTERING SOLUTION

To provide hardware support for distinct parallel programming models, this work modifies a virtual platform, already used by our group in previous works [5]. This is a cycle-accurate virtual platform described in SystemC. It is configurable in terms of number of cores, memory organization [8], cache size and placement of components on the NoC. The Processor Element (PE) used is a MIPS R2000, and all components are interconnected through a mesh NoC [9]. This NoC implements wormhole packet switching to reduce energy consumption. It also uses XY routing, to avoid deadlock situations, and a handshake control flow.

The support for distinct parallel programming models comes from a simple solution to provide shared memory and distributed memory organizations depending on the necessity of the application. The data can be stored locally (and privately) thus creating a distributed memory organization, or the data can be stored in a memory on another node of the NoC and the local memory will be used as L1 cache. For more details, please refer to our previous work [6].

In this work, each allocated application generates, and it is treated as, a cluster. Clusters are dynamic regarding the number of processors (the size of the cluster). This occurs based on a work stealing policy in which, every time a task finishes, a multi-level scheduler tries to steal a task from a processor that has more tasks. This multi-level scheduler also adapts the memory organization to the application programming model being allocated. This allocation takes into account, amongst other characteristics, the dynamic change of degree of parallelism in an application [9]. Every time a task finishes its execution, the scheduler modifies a Resource Occupation Map in order to keep track of how busy a processor is. This map holds the information of which resources are currently allocated to each application and how many tasks each processor has. If there are no more tasks to run, the scheduler looks in the Resource Occupation Map for a task to be migrated to this currently idle processor. Primarily, the choice of the task to be migrated is based on the following criteria:

- A task from an overloaded processor (i.e. a processor with more than one task) located in the same cluster;
- Preference is given to a migration between two processors that are closer to each other;
- Preference is given to a migration from a more overloaded processor (number of tasks).

The processor that best fits these conditions will receive a work-stealing message and a task migration will take place. This is a very simple way to decide the task migration and in this work it will be used as the baseline against which more elaborated policies will be compared.

978-3-9815370-0-0/DATE13/©2013 EDAA

III. RESOURCE MAPPING POLICIES

This section presents four policies used to map resources that become available. These are policies based on characteristics of the applications that certainly impact on the overall performance of the system. Each one of these policies determines the main rule to be applied to perform the resource allocation. In the case where the sole use of this policy points to more than one eligible cluster, the criteria presented in the previous section are used as a tie-break.

A. Shared Variables First (SVF)

This policy tries to take advantage of the fact that Shared Variables applications are easier to migrate due to the lack of huge amounts of data to be sent from one local memory to another. Experiments showing evidences of this behavior are shown in Section 6.

In this policy, the first criterion is to migrate a Shared Variable task to occupy the resource that becomes available. With this policy it is expected that scenarios where the largest applications are based on Shared Variables will take more advantage of the resources.

B. Higher Workload (HWL)

Typically, applications with higher workload take more time to finish, and the intuition in this policy is to prioritize this kind of application by giving it as much resources as it needs (provided the resources are available). In this case, the scheduler will have a pre-determined rank of the applications based on their workload. This will create a behavior where, as resources become available, they will be allocated to clusters that are running the application with higher workload and that have at least one overloaded processor. If the cluster does not have any overloaded processor, the resource will be given to the application with the next higher workload, and so on.

C. Cluster Shape (CS)

Given the nature of the Dynamic Clustering process, the cluster size may change (having more or less nodes). However, given the fact that the architecture presented is interconnected by a Network on Chip, this dynamic cluster size can lead to a non-efficient shape of the cluster. It is well known that the minimal average distance between nodes of a 2-D mesh NoC is obtained when the system topology forms a square. Any other rectangle topology is less efficient and, therefore, undesired.

With that knowledge in mind, the Cluster Shape policy tries to give the available resources to a cluster as long as the resulting topology becomes as close as possible to a square, thus guaranteeing smaller fluctuation on the average distance between the nodes of the same cluster.

D. Off-Chip Memory (OCM)

Typical embedded systems have very limited on-chip memory due to their constraints of area, power and energy. However, nowadays, applications have ever increasing amounts of data to be dealt with. Typical streaming applications have to store chunks of the stream in an off-chip memory (most commonly a flash memory). Due to limitations on the number of pins in a chip and also on the number of memory ports, the number of memory controllers must also be limited. In large MPSoCs the distance between the actual core and the memory controller location on the NoC may have an impact on the overall latency. In this policy, we try to take advantage of that fact by mapping applications as close as possible to a memory controller. For this particular experiment, we consider four memory controllers on the corners of the NoC in order to have an overall worst case scenario.

IV. EXPERIMENTAL SETUP

The experiments consider four applications: a Matrix Multiplication (MM), a Motion Estimation (ME) algorithm, a Mergesort (MS) algorithm, and a JPEG encoder. Table I shows the resource requirements (in this case, only processors) of each application throughout its execution (NT means the initial total number of tasks). With this table, we are trying to predict how fast an application releases processors which can be used by other applications according to the mapping policies.

TABLE I. RESOURCES REQUIRED BY EACH APPLICATION.

Application	Sequence of resources	Progression
Matrix Multiplication	NT, 1, 0	Constant
Motion Estimation	NT, 0	Constant
Mergesort	NT, NT/2, NT/4, …, 1, 0	Exponential
JPEG	NT, 1, 0	Constant

According to this table, the MS algorithm, due to its divide and conquer nature, has tasks finishing at an exponential pace. Meanwhile, MM, ME and JPEG are applications in which the degree of parallelism is almost constant throughout their execution. This means that, for these three applications, if one task finishes, it is quite probable that all tasks of that application will end very soon.

Considering the data inputs for the applications described above, Figure 1 represents their communication workload regarding different numbers of processors. Based on this chart, it is expected, for instance, that the ME algorithm will generate a larger amount of data exchanges.

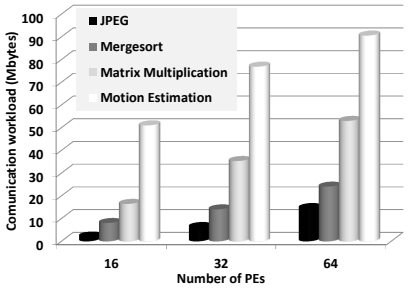

Fig. 1. Communication workload.

Depending on the parallel programming model in which these applications were developed, the task migration used to perform Dynamic Clustering can have different costs. For instance, when shared memory applications migrate, they only transfer their context, whereas distributed memory ones have to transfer the whole contents of the instruction and data memories. At the same time, memory organizations that lead to low-cost migrations (such as shared memory) can also intrinsically have higher memory latency and, therefore, generate higher execution times. The configuration used in all experiments is presented in Table II.

TABLE II. CONFIGURATION OF THE VIRTUAL PLATFORM.

Number of Processors	16; 32; 64	L1 Cache Size	8192 bytes
Number of L2 caches	4	Block Size	32 bytes
Total Number of Initial Tasks per Application			32

V. RESULTS

A. Results without applying policies

Table III presents the combinations of programming models and applications. The terminology used to define each combination is in the format *nD-mS*, meaning that, in this specific situation, the MPSoC was running n Distributed memory applications and m Shared memory applications. Excluding the situations where all applications use the same parallel programming model, the other three groups (3D-1S, 2D-2S and 1D-3S) can have multiple permutations of applications.

Figure 2 presents the execution time results for all combinations (higher means worst) for 16, 32 and 64 cores. The overhead as indicated represent the amount of time spent on task migrations. Inside each grouping it is possible to see very different results, depending on the combination of applications. In the 3D-1S group, the best combination overall is the one where ME uses a Shared Memory environment (3D-1S-C). This is mainly due to the fact that ME has a higher workload and the Shared Memory model provides the fastest task migration. Therefore, the migration overhead is lower.

TABLE III. COMBINATIONS OF APPLICATIONS.

Grouping	Permutation	Distributed	Shared
3D-1S	A	MM, ME, Mergesort	JPEG
	B	MM, ME, JPEG	Mergesort
	C	MM, Mergesort, JPEG	ME
	D	ME, Mergesort, JPEG	MM
2D-2S	A	MM, JPEG	ME, Mergesort
	B	MM, ME	Mergesort, JPEG
	C	MM, Mergesort	ME, JPEG
	D	ME, JPEG	MM, Mergesort
	E	ME, Mergesort	MM, JPEG
	F	Mergesort, JPEG	MM, ME
1D-3S	A	ME	MM, Mergesort, JPEG
	B	JPEG	MM, ME, Mergesort
	C	Mergesort	MM, ME, JPEG
	D	MM	MM, Mergesort, JPEG

In the 2D-2S group, some of the combinations resulted in the worst results. This is due to the fact that, with two applications using each parallel programming model, it is easier to have situations where migration costs will be higher, depending on which applications are developed using these high migration-cost parallel programming models. At the same time, there is also room for situations where the workload-heavy applications were developed in low migration-cost programming models. This is the specific case of the 2D-2S-F case. In this case, smaller applications like MS and JPEG use the Distributed Memory organization and, therefore, will finish even earlier than the other two applications. Also, the larger applications use the Shared Memory organization and can thus migrate faster.

As for the 1D-3S group, the situation is fairly even, but the best results are the ones where the ME and MM applications use the Shared Memory organization (1D-3S-B). This reduces the task migration cost, and the migration happens earlier due to the fact that the smallest application (JPEG) was developed using Distributed Memory organization.

Overall, these results show that, by changing which parallel programming version of an application is executed, the execution time can vary in up to 50% (the difference between 2D-2S-B and 2D-2S-F).

A. Results on the use of policies

Since the grouping 2D-2S presented the most diverse results, we simulated the use of the three policies only for permutations in this group. These results are a normalized performance improvement based on the results of Figure 2, which use a more simple migration policy as explained in Section II.

Figure 3 presents the results of performance gains regarding the use of the SVF policy, when compared to the baseline. There is a reasonable improvement especially in the cases where the applications with higher workload are programmed using shared variables, creating a situation where the tasks chosen to migrate are the lighter ones. These results show that 2D-2S-A and 2D-2S-F present better results. These two combinations have ME as a shared variable application, which, as seen in Fig. 5, is the application with higher workload.

Analyzing the results of the HWL policy, presented in Figure 4, it is possible to notice that in some cases the use of this policy hurts the overall performance of the system. This is due to the fact that this policy creates a high priority for applications with higher workload. This may lead to a situation where high workload applications may have higher migration costs depending on their parallel programming paradigm. On the other hand, the results for the 2D-2S-A and 2D-2S-F combinations show that, depending on the combination of cheaper task migration and high workload, the results of this policy can be even better than those of the SVF policy for the same combination.

Figure 5 presents the results for the CS policy. In this case, the best combination (2D-2S-F) does not present better results than the SVF policy. However, the average results for each policy presented in Table IV shows that this policy has better results than the previous two ones in situations of 32 and 64

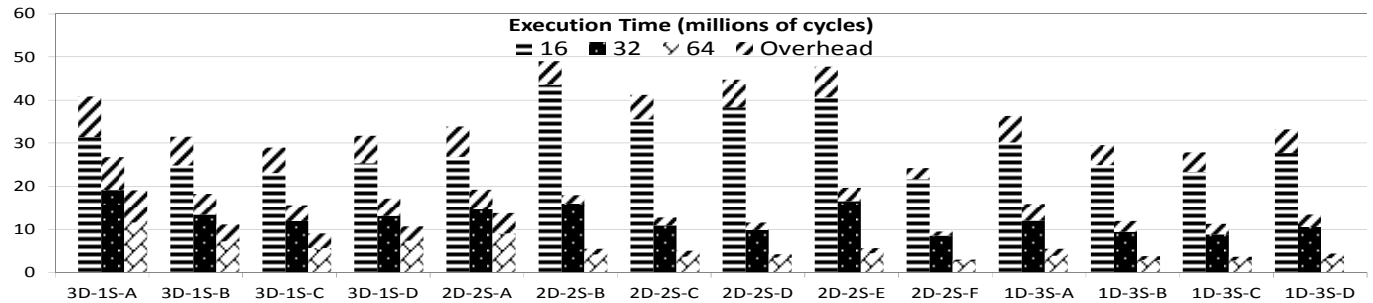

Fig. 2. Execution time results for the combination of applications.

cores. These results suggest that this policy may be a better solution if the nature of the applications is unknown.

Figure 6 present the results for the OCM policy. In this case, combination 2D-2S-F still presents the better results. However, differently from the previous policies, the 64-core scenario presents the best results. This is most likely due to the fact that a larger amount of cores increases the average distance between two cores in the NoC. Therefore, the latency resulting from the off-chip memory access has a larger impact on the overall results. By keeping tasks closer to the memory controller, we manage to reduce this latency.

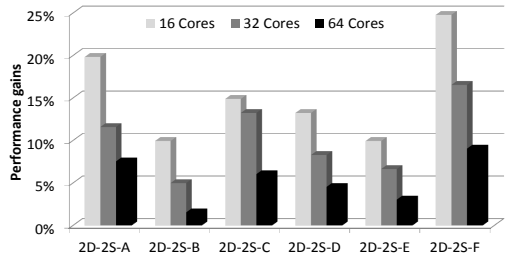

Fig. 3. Shared Variables First policy.

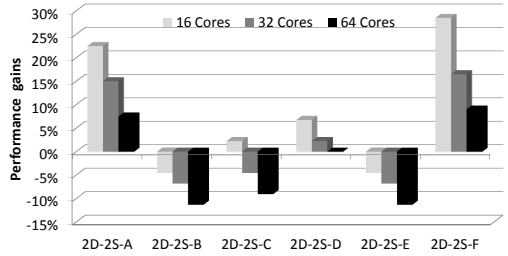

Fig. 4. Higher Workload policy.

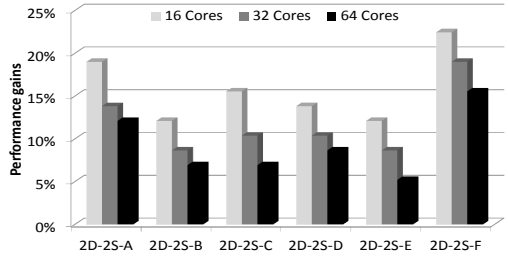

Fig. 5. Cluster Shape policy.

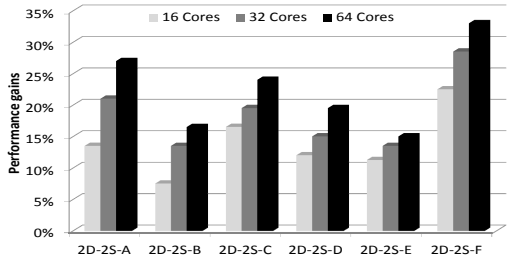

Fig. 6. Off-Chip Memory policy.

VI. Related Works

Recent works propose a paradigm of multicore programming called Invasive Computing [2,3,4]. The proposal is to take advantage of a large many-core processor in the best possible way. The idea leverages on malleable applications where the degree of parallelism can change on the fly. At the application level, the programmer uses functions that trigger the invasion process. When an application sends an invade request, a distributed resource manager evaluates the request based on the amount of resources required and the estimated speed-up per core.

This proposal may seem similar to the one presented in this work. However, Invasive Computing works at different levels of abstraction, and, therefore, the programmer must have some notion of the methodology. In our work, all mechanisms are transparent to the programmer. With that, Dynamic Clustering offers support to legacy parallel programmed codes. Also, Dynamic Clustering does not rely on middleware in order to run applications programmed using distinct parallel programmed applications. To the best of our knowledge, this is the only work that offers this feature.

VII. Conclusions and Future Works

Results show that different aspects can be considered regarding the task migration in order to effectively distribute resources. For instance, aspects like task migration overhead cost for different parallel programming models, the workload of the applications, and the average distance between nodes running tasks from the same application have impact on performance. Experimental results using policies that consider these aspects show that resource management of tasks in an MPSoC can be improved up to 22.5% in average. Furthermore, these policies result in different gains (or losses), depending on the combination of parallel programming models used by distinct, concurrent applications.

As for future works, it is our intention to extend the scheduler, for instance including the dynamic allocation of tasks that arrive to the system. In addition, the solution presented here must be compared to other ones to assess overhead costs and overall energy spent and performance.

References

[1] M. Forsell. “A Scalable High-Performance Computing Solution for Networks on Chips”. IEEE Micro 22, Vol. 5, Sept. 2002.

[2] J. Teich et al. “Invasive computing: An overview”. In Multiprocessor System-on-Chip – Hardware Design and Tool Integration. Springer, Berlin, Heidelberg, 2011.

[3] J. Henkel. “Invasive manycore architectures”. In Proc. ASP-DAC, 2012.

[4] S. Kobbe. “DistRM: Distributed resource management for on-chip many-core systems”. In Proceedings of CODES+ISSS, 2011.

[5] E.T.Silva Jr., D.Barcelos, F.R.Wagner, and C.E.Pereira. "An MPSoC Virtual Platform for Real-Time Embedded Systems". In Proc of JTRES, 2008.

[6] G. Girao, T. Santini and F.R. Wagner. “Dynamic Clustering for Distinct Parallel Programming Models on NoC-based MPSoCs”. In Proc of NoCArc, 2011.

[7] G.Girao, D.Barcelos, and F.R.Wagner. "Performance and Energy Evaluation of Memory Hierarchies in NoC-based MPSoCs under Latency". In Proc. of VLSI-SoC, 2009.

[8] C.A.Zeferino, M.E.Kreutz, and A.A.Susin. “RASoC: a Router Soft-core for Networks-on-chip”. In Proc. of DATE, 2004.

[9] B. Xu and D. H. Albonesi. “Runtime Reconfiguration Techniques for Efficient General-Purpose Computation”. IEEE Design & Test 17, 2000.