SPA: Simple Pool Architecture for application resource allocation in many-core systems

Jayasimha Sai Koduri, Iraklis Anagnostopoulos
Department of Electrical and Computer Engineering, Southern Illinois University, Carbondale, U.S.A.

***Abstract*—The technology push by Moore's law brings a paradigm shift in the adaption of many core systems which replace high frequency superscalar processors with many simpler ones. On the software side, in order to utilize the available computational power, applications are following the high performance parallel/multi-threading model. Thus, many-core systems raise the challenges of resource allocation and fragmentation making necessary efficient run-time resource management techniques. In this paper, we propose SPA, a Simple Pool Architecture for managing resource allocation in many-core systems. The proposed framework follows a distributed approach in which cores are organized into clusters and multiple clusters form a pool. Clusters are created based on system's characteristics and the allocation of cores is performed in a distributed manner so as to take advantage of spatial features, shared resources and reduce scattering of cores. Experimental results show that SPA produces on average 15% better application response time while waiting time is reduced by 45% on average compared to other state-of-art methodologies.**

I. Introduction and Motivation

MODERN computing and embedded systems are moving from superscalar and multicore architectures and follow the many-core paradigm, which is characterized by the constant increase in the number of integrated processors [1]. The number of cores to be integrated in a single chip is expected to rapidly increase in the coming years [2] marching towards thousand core chips [3]. Currently, commercial available systems can accommodate more than 100 cores [4], [5] while academia explores new frontiers in the number of integrated processors [6].

This increased number of integrated processors requires efficient inter-core communication and data management techniques. Network-on-Chip (NoC) has been established as the communication architecture for overcoming bottlenecks [7] and provide efficient inter-core data exchange while run-time mapping techniques are used in order to allocate resources to the incoming applications. Nevertheless, modern many-core systems still face performance losses. The high number of processing elements increases the complexity of mapping decisions and traditional resource managers fail to exploit the underlying hardware. As a consequence, they do not provide the desired high performance that applications expect from such large pools of resources [8].

Additionally, cores are not fully independent processors but share resources, such as caches and memory controllers, creating contention points and resulting in performance degradation. Cache coherency scales well for a small amount of cores but it becomes the bottleneck in high core-count processors [9]. An architectural extension is clustered many-core systems, where cores are grouped together sharing a last level cache and the memory controller. The clustering principle, due to its design simplicity, allows the scaling of the number of cores (e.g. 72 cores [10]), however it raises additional challenges for the run-time managers [11].

From the software perspective and in order to take advantage of the underlying infrastructure, applications have become highly parallel, dynamic and compute-intensive. In order to handle applications' dynamicity, efficient thread-to-core mapping is required as static approaches cannot predict nor handle run-time workload variability. Specifically, performance- and communication-aware run-time mapping algorithms mostly try to allocate cores in a contiguous way so as to minimize communication cost and boost performance [12]. By having a contiguous allocation, NoC traffic congestion is reduced and inter-core communication is benefited. As the number of incoming applications increases, mapping of multiple applications can result in scattered unutilized cores which will either stay idle, because their allocation does not satisfy the application's requirements, or they will be used without exploiting platform's spatial characteristics. This problem is also referred as *fragmentation problem [12]*. Furthermore, fragmentation is greatly affected by application arrival rate and it becomes harder to manage it from one central point. Traditionally, a central core analyzes the status of the system and the application's requirements trying to find the best match between them. However, there techniques cannot adapt to frequent system changes as they suffer from communication overhead [13]. Distributed management can overcome these problems and provide efficient allocation of cores.

In this work we present SPA, a Simple Pool Architecture scheme for run-time resource allocation of multi-threaded applications on many-core systems. The proposed framework follows a distributed approach in which cores are organized into clusters while multiple clusters form a pool. Clusters are created based on system's characteristics and optimization policies and the allocation of cores is performed in a distributed way. Specifically, SPA is responsible: (i) to generate the pool-based structure and organize cores into clusters depending on the NoC architecture (it provides an abstract representation of available resources); (ii) to serve, at run-time, the needs of multi-threaded applications, in terms of processing cores; and (iii) to allocate resources in order to take advantage of spatial features, shared resources and reduce scattering of cores.

978-3-9819263-0-9/DATE18/©2018 EDAA

II. Related work

Several research works have presented run-time resource management schemes aiming at efficient resource utilization and application optimizations. Regarding run-time mapping and minimization of communication cost on many-core systems, authors in [14] present a decentralized cluster-based task-to-core scheme while authors in [15] take into account the NoC topology by using static application models. In [3] a methodology is presented to map incoming applications into continuous cores by searching for neighboring blocks of cores. If there are no free contiguous cores the mapping request will be refused and the application will be placed in a queue until system resources are freed. An effort to minimize contention and increase applications' communication cost was presented in [13], where an hierarchical run-time distributed scheme serves and monitors applications employing run-time self-adaptation. Moreover, in [12] the authors use segregated contiguous blocks of cores. When a new application arrives in the system, the requirements of that application, in terms of cores, is rounded up to the nearest power of two and the application is mapped in segregated blocks. This approach achieves good results in terms of communication cost, however may leave system resources underutilized. Additionally, the communication aspects of the mapped applications are taken into account in [16], where an Integer Linear Programming formulation is used in order to calculate the best solution and map the application. However, this solution follows a centralized approach and it is focused on specific applications.

Regarding system's fragmentation, the authors in [12] migrate a complete set of tasks initially mapped in contiguous cores in order to serve incoming applications and keep fragmentation low. However, this approach leads to high migration cost as it rounds up the applications' requirements in terms of cores. In [17], the authors propose the usage of agents that divide the cores into clusters and migrate tasks between them in order to provide space for incoming applications. However, they do not consider the migration cost. The authors in [18] use a defragmentor routine which runs when an application finishes and a fragmentation metric keeps track of the system resources. The defragmentor routine is executed whenever a fragmentation exceeds a specific threshold. However, without knowing the requirements and the duration of the incoming application, the task migration may have a negative effect on the whole system. If the incoming application has independent tasks which do not require any frequent communication, the drfragmentor should not be activated.

In this work, we present SPA, a run-time resource allocation scheme that follows a distributed approach in which cores are organized into clusters while multiple clusters form a pool. The novelty of the paper is threefold: (i) SPA can be configured either for minimizing communication-cost or contention; (ii) It allocates cores in maximum possible contiguous manner and it is resilient in terms of different arrival rates of applications; and (iii) It increases system's resource utilization, in terms of shared caches and memory controllers, by controlling the scattering of cores while mapping.

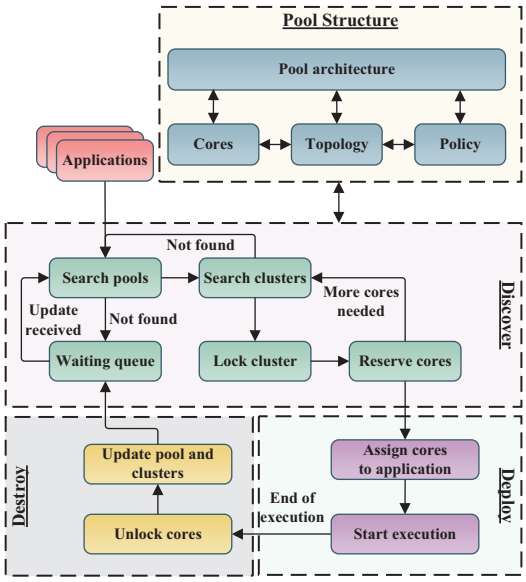


Fig. 1: Overall flow of the proposed methodology

III. Proposed methodology framework

The goal of the proposed methodology framework is to perform run-time resource allocation on many-core systems by maintaining fragmentation and increasing the resource utilization. An overview of our approach is presented in Figure 1. The first thing is the organization of the available computing resources into clusters and pools. The pool architecture (Section III-A) is generated based on the characteristics of the platform such as (i) number of cores; (ii) topology (shared resources, memory controllers etc.); and (iii) optimization policy. Currently SPA supports two policies: (i) minimization of communication-cost; and (ii) maximization of shared resources. Once the pool architecture is defined, it is stored as a shared structure among cores in order to allow distribution in resource allocation. SPA follows the *3D phase rule* in resource allocation (Section III-B): *Discover, Deploy and Destroy*. The Discover phase is triggered when a new application enters the system and the goal is to find clusters, within the same pool, that satisfy the requirements of the application in terms of computing components. The pool structure allows multiple applications to search for cores and excludes applications from competing for the same resources at the same time. During the Deploy phase, the allocated cores execute the workload of the application. Last, in the Destroy phase, the application returns the allocated cores back to the clusters and marks them as unoccupied.

A. Pool architecture

SPA organizes the computing resources of a many-core system into clusters and multiple clusters form a pool. The set $P = \{P_1, P_2, \cdots, P_N\}$ describes the available pools in SPA. Each pool $P_i = \{i, F_{pe}, N_{app}, C\}$ is characterized by (i) its id i; (ii) F_{pe}, the number of free cores inside the pool; (iii) N_{app} the number of applications inside the pool; and (iv) C

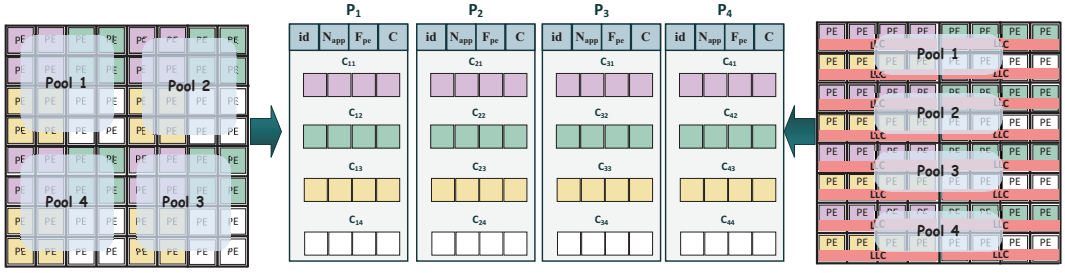


Fig. 2: Example of pool architecture. The pool-based approach offers an abstraction layer that makes SPA resilient and adaptable

the number of available clusters. Currently, C is the same for all pools. Exploration of unbalanced clusters is left as part of future work. In general, C_{ij} represents the j cluster of the pool P_i. Each $C_{ij} = \{j, F_{pe_{ij}}, L\}$ is characterized by its id j, the number of free cores $F_{pe_{ij}}$ and a lock L which describes if this cluster is currently locked by an application that searches for resources. PE represents the processing elements in cluster C_{ij}. Each $PE = \{id, S\}$ is characterized by its id and the status of the core S which indicates whether it is occupied.

The number of pools $|P|$ and clusters C play an important role in application fragmentation. High values of $|P|$ and C benefit the applications that require small number of cores. As each cluster employs few cores, applications can utilize the resources in the granularity of clusters. On the other hand, small values for $|P|$ and C benefit applications that require large number of cores. In this way, applications utilize the resources in the granularity of pools. How clusters are generated depends on the optimization policies. Currently SPA supports two policies: (i) minimization of communication-cost; and (ii) maximization of shared resources.

The first policy benefits applications that require frequent inter-core communication. Clusters are organized according to the average node distance while pools are organized according to the average cluster distance. The average node distance D of an $M \times N$ mesh is given by Equation 1. As aforementioned, in SPA the number of cores per cluster is the same. In order to form clusters, we start from the first node of the NoC and we calculate the value of D for all possible combinations that create a mesh and include the required number of cores. For each of these clusters we select the one with the lowest value of D. After clusters are formed, we follow the same procedure at the cluster level in order to form the pools. The second policy, focuses on increasing the utilization of shared resources such as caches and memory controllers and benefits applications with heavy data reuse. Clusters are formed with cores that share the Last Level Cache (LLC) while pools are formed with clusters that share the memory controller. In this way, applications can benefit from the locality of the shared LLC and the distribution of memory controllers.

$$D = \frac{1}{3}[(M - \frac{1}{M}) + (N - \frac{1}{N})] \quad (1)$$

An example of how cores are organized into clusters and pools is depicted in Figure 2. At the left part of the figure, the many-core system is organized under the cost minimization criterion while at the right part, the system is organized in order to maximize the utilization of shared resources. In both cases, even though the topology is different, SPA organizes the resources in the same abstract way.

B. SPA resource allocation

The SPA resource allocator is triggered whenever a new application enters the system or when there is a change in the number of free resources. The architecture of the pool is stored a shared structure among cores in order to allow distribution in resource allocation. Each application A_i is characterized by the $A = \{C_{rq}, C_{rcv}\}$, where C_{rq} represents the requirements in terms of processing elements and C_{rcv} is a list that indicates the allocated cores. As aforementioned, resource allocation in SPA is based on the *3D phase rule*, Discover (search for resource allocation), Deploy (execute the application's workload) and Destroy (return the resources back to the system).

The SPA resource allocation procedure is based on five rules and it is described in Algorithm 1. Rule 1: Every new application selects the pool that currently has more free cores F_{pe}. If multiple pools exist and they have the same number or free cores, then the one with the smaller number of applications inside it, N_{app}, is chosen (lines 1-8). By following this rule, applications that arrived at the same time will not compete for the resources of the same pool. Consequently, applications already inside a pool will benefit and achieve better resource utilization. When an application selected the pool it will update the N_{app} and F_{pe} of the pool. Rule 2: It follows rule 1. In this rule only one application can enter a cluster while acquiring cores (lines 10-20). When a pool P_i is found, the next step is to search for a cluster C_{ij} that has the maximum number of free cores $F_{pe_{ij}}$. When a cluster is found, the application will acquire the lock L of the cluster, so no other application can ask for these resources. The cores of that cluster are added to the C_{rcv} list and the $F_{pe_{ij}}$ and L indexes are updated. By following this rule we can reduce the chances of two applications falling into same cluster. The cluster may have shared memory resources like cache so there will be less chance of cache flushing. Rule 3: When C_{rq} of A_i is greater than $F_{pe_{ij}}$ in the selected C_{ij}, from previous rule, the application searches for clusters inside the same pool (lines 21-23). In many cases, the requirements of an application cannot fit only in one cluster. In this case, the application holds the lock L and searches for other clusters inside the

same pool by following Rule 2. By following this rule there will be less scattering of application cores and reduce the communication cost. Rule 4: When C_{rq} of A_i is greater than $F_{pe_{ij}}$ in C_{ij}, and no other clusters are available in the pool P_i, the application searches for a new pool (lines 24-26). In order to maintain locality and reduce fragmentation, if the selected pool P_i cannot serve the requirements of the application, the already acquired resources C_{rcv} of clusters are kept and the application searches the others pools for available resources in order to increase system utilization and locality. Rule 5: When an application A_i cannot enter a pool, it is placed in a waiting queue (lines 28-30). If an application cannot find a pool that satisfies the constraint $F_{pe} \geq C_{rq}$, it enters the waiting queue and unlock all C_{rcv} cores. There is a trade off here, an application can keep the cores locked and wait till the availability of remaining resources or it can unlock cores for next available application. These policies need a further investigation. SPA resource allocation is triggered whenever an application exits the system or whenever a new application enters into the system.

Once the application satisfies the constraint $C_{rcv} = C_{rq}$, it enters the Deploy phase where tasks are bounded to the cores and the execution of the application starts. The number of utilized cores is reflected in the values of $F_{pe_{ij}}$ and F_{pe}. Last, when the application ends, it enters the Destroy phase where all cores of the C_{rcv} are marked as free and the values of $F_{pe_{ij}}$, N_{app} and F_{pe} are updated.

Algorithm 1 Resource allocation for incoming applications

Input: C_{rq}*: Required cores for that application*
Output: C_{rcv}*: Received cores for Mapping*

```
1: Update(P_max):
2: for each P_i ∈ P do
3:     P_max =pool with maximum free cores;
4:     if (pools have same free cores) then
5:         P_max =pool with minimum applications;
6:     end if
7: end for
8: update N_app and F_pe_i of P_max
9:
10: Update(C_max):
11: for each cluster inside P_max do
12:     C_max = find cluster with maximum free cores;
13: end for
14: Lock(C_max)
15:
16: for i = 0 to C_rq_i do
17:     if (F_pe_max != empty) then
18:         C_rcv = Getcores(C_max);
19:         Lock(core);
20:         update F_pe_max
21:     end if
22:     if (F_pe_max == empty) then
23:         Unlock(C_max);
24:         Update(C_max);
25:         if (C_max == NULL) then
26:             Update(P_max);
27:         end if
28:         if (P_max == NULL) then
29:             Push A_i to waiting queue
30:         end if
31:     end if
32: end for
33: for each cores inside C_rcv do
34:     Bind(core to application)
35: end for
36: if (application exit) then
37:     for ∀ cores in C_rcv do
38:         Unlock(core);
39:         update pools and clusters
40:     end for
41: end if
```

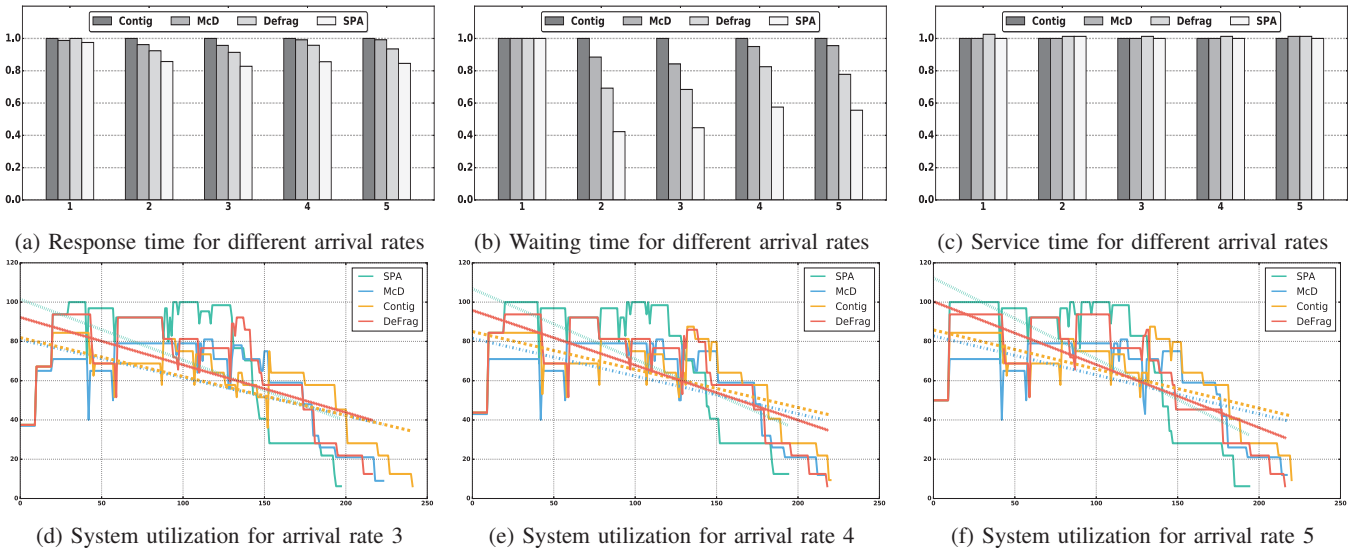
IV. Experimental Results

In order to evaluate our framework, we performed extensive simulations using the Sniper [19] many-core simulator, PARSEC [20] and NAS [21] parallel benchmarks as high performance parallel applications. Specifically, SPA was developed on top of Sniper which was modified in order to support different arrival rates and intervals for multiple applications.

We compared the performance of SPA to three run-time resource managers that also focus on reducing the fragmentation of the system: (i) *Contig* the resource allocator presented in [3]. Contig follows a smart hill climbing approach in order to allocate resources in a continuous way; (ii) *Many core Defragmenter (McD)* [12] which allocates segregated contiguous blocks of cores and supports thread migration. McD rounds up the requirements of the incoming applications to a power of two to perform exponentially separable mapping; and (iii) *Defrag* [18] which uses a fragmentation metric to monitor the status of the system.

The metrics that we used to compare all the aforementioned resource managers are: (i) *Response time:* The time from the arrival up to the exit of an application from the system; (ii) *Service time:* The actual execution time of the application; (iii) *Waiting time:* The time during which the application waited in order to get resources from the manager (it includes the time in the waiting queue); and (iv) *System utilization:* This metric shows how many of the available processing elements were active executing the workload of an application. The evaluation was performed on an 8×8 NoC where each processing element has a private 32KB L1 D-cache, 32KB private L1 I-cache, 512KB private L2 cache and 4 cores share an 8MB L3 cache. We evaluated five different scenarios in which we set the arrival interval to 10 ms and the arrival rate was changing from one (one application arrives every 10 msec) to five (five applications arrive every 10 msec).

Figure 3 presents the results for the PARSEC benchmarks. We used communicational intensive applications from the PARSEC benchmark suite to show how SPA performs under this constraint. Regarding SPA, the pool architecture that focuses on the minimization of communication cost was chosen with four pools and four clusters per pool. Specifically, Figures 3a, 3c and 3b present the performance of the resource managers regarding response, waiting and service time accordingly for different arrival rates. Regarding the response time depicted in Figure 3a, applications under SPA performed better



(a) Response time for different arrival rates (b) Waiting time for different arrival rates (c) Service time for different arrival rates

(d) System utilization for arrival rate 3 (e) System utilization for arrival rate 4 (f) System utilization for arrival rate 5

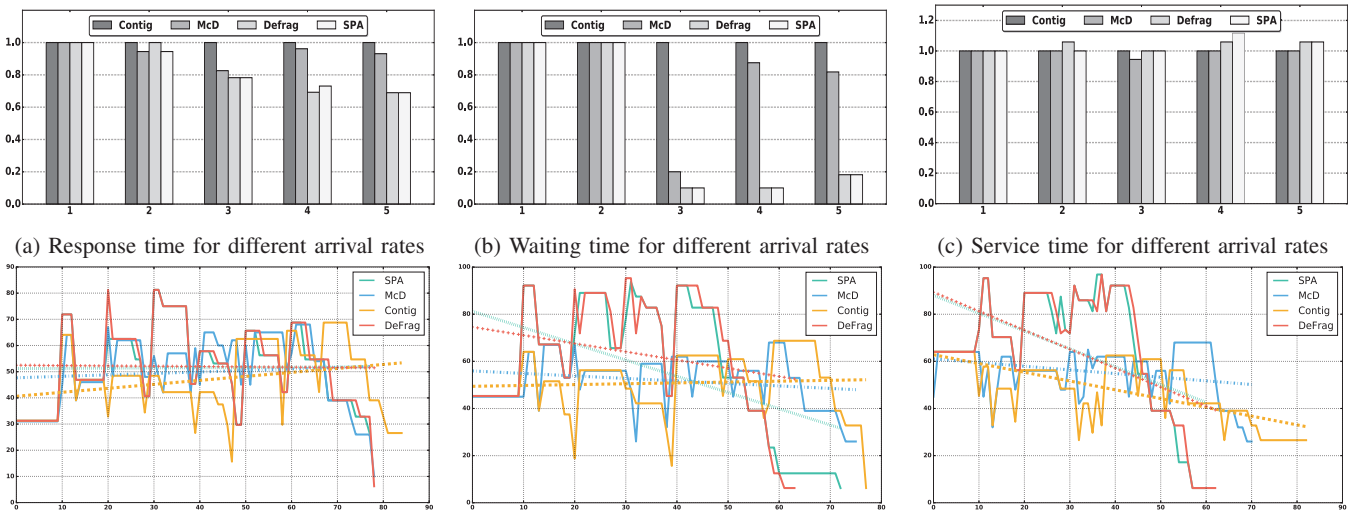
Fig. 3: Comparison of different resource allocator for PARSEC [20] benchmarks

with an average of 18% compared to Contig, 15% compared to McD and 10% compared to Defrag. As the arrival rate increases, SPA manages to perform better due to the the pool architecture. This is also reflected in Figure 3c which depicts the average waiting time for all applications in each scenario. When the arrival rate is one, applications are mapped on the system without waiting. As the arrival rate increases, waiting time for Contig and McD increases as well. This happens because Contig searches for the best continuous area for each application while McD rounds the requested cores to a power of two. Thus, if an application requires 6 cores, McD will allocate 8 for it. However, 8 cores may not be available at that time. Overall SPA reduces waiting time by an average of 55% compared to Contig, 45% compared to McD and 31% compared to Defrag. Regarding service time depicted in Figure 3b, all managers performed in a similar way. SPA has a negligible overhead due to the fact that fragmentation may occur but it will not affect the performance so much. Regarding the utilization of the system, Figures 3d, 3e and 3f present the system's utilization throughout for arrival rate three, four and five accordingly. The utilization of system's resources for arrival rates of 1 and 2 are not presented due to lack of space as all managers performed very similar to each other. According to Figure 3d, SPA achieves 100% system utilization for more time compared to other resource managers. This is also reflected to the trendline. A utilization trendline with a small slope (McD and Defrag) results in a more uniform utilization but the system needs more time to serve all the incoming applications. The slope of the the trendline is also connected with the waiting time. As the waiting time decreases, the system utilizes all the available resources in order to give to the applications the required cores.

Figure 4 presents the results for the NAS [21] parallel benchmarks. We use computational intensive applications from NAS benchmarks to show how SPA perform over this type of applications. Regarding SPA, the pool architecture that focuses on the maximization of shared resources was chosen with four pools and four clusters per pool. Specifically, Figures 4a, 4b and 4c present the performance of the resource managers regarding response, waiting and service time accordingly for different arrival rates. Regarding response time depicted in Figure 4a, applications under SPA performed better with an average of 17% compared to Contig, 12% compared to McD and 2% compared to Defrag. As the arrival rate increases, SPA manages to perform better again. This is also reflected in Figure 4b which depicts the average waiting time for all applications in each scenario. When the arrival rate is small (one and two), applications are mapped on the system without waiting. As the arrival rate increases, waiting time for Contig and McD increases as well. Defrag has the same behavior as SPA. This happens because NAS parallel benchmarks are more computational intensive than PARSEC. Overall SPA reduces waiting time by an average of 85% compared to Contig, 78% compared to McD and 9% compared to Defrag. Regarding service time depicted in Figure 4c, SPA has an overhead of 6%. This happens because SPA leverages continuity of cores with waiting time of the applications. Regarding the utilization of the system, Figures 4d, 4e and 4f present the system's utilization throughout for arrival rate three, four and five accordingly. In all cases, SPA performed similar with Defrag because NAS parallel benchmarks are highly parallel and computational intensive applications and Defrag uses thread migration to maintain continuity of cores. This results in better utilization of the available cores.

V. Conclusion

In this paper, SPA a pool-based resource allocation for multi-threaded applications was presented. The proposed framework follows a distributed approach in which cores are organized into clusters and multiple clusters form a pool. Clusters are created based on system's characteristics and the allocation of cores is performed in a distributed manner so



(a) Response time for different arrival rates (b) Waiting time for different arrival rates (c) Service time for different arrival rates

(d) System utilization for arrival rate 3 (e) System utilization for arrival rate 4 (f) System utilization for arrival rate 5

Fig. 4: Comparison of different resource allocator for NAS [21] parallel benchmarks

as to increase resource utilization and reduce fragmentation. SPA allows for scalability and adaptability of the employed policies. Experimental results show that SPA produces on average 15% better application response time while waiting time is reduced by 45% on average compared to other state-of-art methodologies. Regarding the requirements of the applications and the system configuration, a different pool architecture policy, and exploration of the number of pools and clusters can be suited resulting in better resource allocation.

REFERENCES

[1] J. Teich, J. Henkel, A. Herkersdorf, D. Schmitt-Landsiedel, W. Schröder-Preikschat, and G. Snelting, "Invasive computing: An overview," in *Multiprocessor System-on-Chip*. Springer, 2011, pp. 241–268.

[2] C. Silvano, W. Fornaciari, S. C. Reghizzi, G. Agosta, G. Palermo, V. Zaccaria, P. Bellasi, F. Castro, S. Corbetta, E. Speziale, *et al.*, "Parallel paradigms and run-time management techniques for many-core architectures: The 2parma approach," in *2011 9th IEEE International Conference on Industrial Informatics*. IEEE, 2011, pp. 835–840.

[3] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, "Smart hill climbing for agile dynamic mapping in many-core systems," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 39.

[4] J. Jeffers and J. Reinders, *Intel Xeon Phi coprocessor high-performance programming*. Newnes, 2013.

[5] T. Fleig, O. Mattes, and W. Karl, "Evaluation of adaptive memory management techniques on the tilera tile-gx platform," in *Architecture of Computing Systems (ARCS), 2014 Workshop Proceedings*. VDE, 2014, pp. 1–8.

[6] B. Bohnenstiehl, A. Stillmaker, J. Pimentel, T. Andreas, B. Liu, A. Tran, E. Adeagbo, and B. Baas, "A 5.8 pj/op 115 billion ops/sec, to 1.78 trillion ops/sec 32nm 1000-processor array," in *VLSI Circuits (VLSI-Circuits), 2016 IEEE Symposium on*. IEEE, 2016, pp. 1–2.

[7] T. Bjerregaard and S. Mahadevan, "A survey of research and practices of network-on-chip," *ACM Computing Surveys (CSUR)*, vol. 38, no. 1, p. 1, 2006.

[8] T. Desell, K. El Maghraoui, and C. A. Varela, "Malleable applications for scalable high performance computing," *Cluster Computing*, vol. 10, no. 3, pp. 323–337, 2007.

[9] D. Katz, A. Barbalace, S. Ansary, A. Ravichandran, and B. Ravindran, "Thread migration in a replicated-kernel os," in *Distributed Computing Systems (ICDCS), 2015 IEEE 35th International Conference on*. IEEE, 2015, pp. 278–287.

[10] S. Pagani, H. Khdr, J.-J. Chen, M. Shafique, M. Li, and J. Henkel, "Thermal safe power (tsp): Efficient power budgeting for heterogeneous manycore systems in dark silicon," *IEEE Transactions on Computers*, vol. 66, no. 1, pp. 147–162, 2017.

[11] A. K. Singh, P. Dziurzanski, H. R. Mendis, and L. S. Indrusiak, "A survey and comparative study of hard and soft real-time dynamic resource allocation strategies for multi-/many-core systems," *ACM Computing Surveys (CSUR)*, vol. 50, no. 2, p. 24, 2017.

[12] A. Pathania, V. Venkataramani, M. Shafique, T. Mitra, and J. Henkel, "Defragmentation of tasks in many-core architecture," *ACM Transactions on Architecture and Code Optimization (TACO)*, vol. 14, no. 1, p. 2, 2017.

[13] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris, "Distributed run-time resource management for malleable applications on many-core platforms," in *Proceedings of the 50th Annual Design Automation Conference*. ACM, 2013, p. 168.

[14] Y. Cui *et al.*, "Decentralized agent based re-clustering for task mapping of tera-scale network-on-chip system." in *Proc. of ISCAS*. IEEE, 2012, pp. 2437–2440.

[15] B. Yang *et al.*, "Mapping multiple applications with unbounded and bounded number of cores on many-core networks-on-chip," *Microprocessors and Microsystems*, vol. 37, no. 4, pp. 460–471, 2013.

[16] L. Yang, W. Liu, W. Jiang, M. Li, J. Yi, and E. H.-M. Sha, "Application mapping and scheduling for network-on-chip-based multiprocessor system-on-chip with fine-grain communication optimization," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 24, no. 10, pp. 3027–3040, 2016.

[17] M. A. Al Faruque, R. Krist, and J. Henkel, "Adam: run-time agent-based distributed application mapping for on-chip communication," in *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*. IEEE, 2008, pp. 760–765.

[18] J. Ng, X. Wang, A. K. Singh, and T. Mak, "Defrag: Defragmentation for efficient runtime resource allocation in noc-based many-core systems," in *Parallel, Distributed and Network-Based Processing (PDP), 2015 23rd Euromicro International Conference on*. IEEE, 2015, pp. 345–352.

[19] T. E. Carlson, W. Heirmant, and L. Eeckhout, "Sniper: exploring the level of abstraction for scalable and accurate parallel multi-core simulation," in *High Performance Computing, Networking, Storage and Analysis (SC), 2011 International Conference for*. IEEE, 2011.

[20] C. Bienia, S. Kumar, J. P. Singh, and K. Li, "The parsec benchmark suite: Characterization and architectural implications," in *Proceedings of the 17th international conference on Parallel architectures and compilation techniques*. ACM, 2008, pp. 72–81.

[21] D. H. Bailey, E. Barszcz, J. T. Barton, D. S. Browning, R. L. Carter, L. Dagum, R. A. Fatoohi, P. O. Frederickson, T. A. Lasinski, R. S. Schreiber, *et al.*, "The nas parallel benchmarks," *The International Journal of Supercomputing Applications*, vol. 5, no. 3, pp. 63–73, 1991.