Distributed run-time resource management for malleable applications on many-core platforms

Iraklis Anagnostopoulos, Vasileios Tsoutsouras, Alexandros Bartzas, Dimitrios Soudris
School of Electrical and Computer Engineering, National Tech. Univ. of Athens, Greece

ABSTRACT

Todays prevalent solutions for modern embedded systems and general computing employ many processing units connected by an on-chip network leaving behind complex superscalar architectures In this paper, we couple the concept of distributed computing with parallel applications and present a workload-aware distributed run-time framework for malleable applications on many-core platforms. The presented framework is responsible for serving in a distributed way and at run-time, the needs of malleable applications, maximizing resource utilization avoiding dominating effects and taking into account the type of processors supporting platform heterogeneity, while having a small overhead in overall inter-core communication. Our framework has been implemented as part of a `C` simulator and additionally as a runtime service on the Single-Chip Cloud Computer (SCC), an experimental processor created by Intel Labs, and we compared it against a state-of-art run-time resource manager. Experimental results showed that our framework has on average 70% less messages, 64% smaller message size and 20% application speed-up gain.

1. INTRODUCTION

The current trend in computing and embedded architectures is to replace complex superscalar architectures with many processing units connected by an on-chip network. Modern embedded, server, graphics and network processors already include tens to hundred of cores on a single die. Intel has already created platforms with 80, 48 and 50 processing cores [10, 16, 18, 11], while Tilera currently features up to 100 cores per chip. Also, Networks-on-Chip (NoC) capable of supporting such large number of cores, are already supported by the industry (such as the Æthereal NoC [9] from NXP and the STNoC [17] from STMicroelectronics). Moreover, the development of such multi-core platforms is driven by the increase of highly parallel/multi-threading and demanding applications. Thus, the challenge focuses on finding an efficient way to deal with this varying amount of resources at run-time leading to performance improvements.

This work is partially supported by the E.C. funded FP7-248716 2PARMA Project, www.2parma.eu and ENIAC-2010-1 TOISE project, www.toise.eu

The run-time resource management paradigm has been revealed as a key challenge to modern multi-core systems and it has become prominent due to the run-time dynamicity of modern parallel applications and platforms. In modern execution environments run-time resource availability may vary due to system dynamism as resources can be added or removed from such environments at any time. As described in [12] malleability is used for autonomous application reconfiguration in response to dynamic changes in platform's resource availability, thus allowing applications to optimize the use of platform's features (e.g., number of processors). In other words, malleable applications use varying amounts of platform resources during their execution and they may specify the minimum and maximum number of processors they require. As minimum can be considered the minimum number of processors a malleable job needs while maximum describes the maximum one. Any allocation of cores more than the maximum number is a waste or platform resources.

As applications and computing systems are continuously getting more and more complex, it becomes harder to manage them from one central point. Traditionally, a central core periodically analyzes applications' malleability and platform resources and tries to find the best match between them. However, such centralized approaches [15] limit scalability due to bottlenecks appeared from processing and communication functions, especially in environments that require frequent configuration changes. Also, the large number of cores in modern systems, increase the failure rate of single processors resulting in system erros when parallel applications are executing [3]. Last, centralized run-time managers lack the concept of self-adaptation and self-organization, actions that trend to be a solution to modern platforms [13].

In this work, we couple the concept of distributed computing with parallel applications and we present a workload-aware distributed run-time framework for malleable applications running on many-core platforms. The proposed framework is based on the idea of local controllers and managers while an on-chip intercommunication scheme ensures decision distribution. The presented framework is responsible (i) for serving, at run-time, the needs of malleable applications, in terms or processing cores; (ii) makes sure that the application will get the optimum number of cores avoiding dominating effects; (iii) it takes into account the type of processors best utilizing any platform's heterogeneity; and (iv) it has a small overhead in overall core intercommunication.

The rest of the paper is organized as follows. An overview of previous works is presented in Section 2, while the proposed methodology framework is presented in Section 3. The evaluation results are presented in Section 4, and finally conclusions are drawn in Section 5.

2. RELATED WORK

The run-time resource management on many-core platforms can be either centralized or distributed. Nollet et al. [14] present a centralized runtime resource management scheme that is able to efficiently manage an NoC containing fine grain reconfigurable hardware tiles and two task migration algorithms. The resource management heuristic consists of a basic algorithm completed with reconfigurable add-ons. Al Faruque et al. [1] present a distributed cluster oriented framework for homogeneous platforms which is based on agents for the task-to-cluster mapping. Anagnostopoulos et al. [2] present a divide and conquer based distributed run-time mapping framework for both homogeneous and heterogeneous platforms with the introduction of a matching factor. In order to reduce the on-chip node intercommunication Cui et al. [4] present a decentralized cluster-based scheme for task mapping, designed for reduction of the communication traffic between agents. *However, even though these approaches handle application mapping at run-time in a good way, they are designed for fixed-size applications without any malleability aspect.* Thus, no application reconfiguration is performed in response to any dynamic changes of platform's available resources and no resizing or remapping of the applications is allowed.

On the field of self-organized and dynamic systems and from the aspect of malleable or parallel applications in general, Sabin et al. [15] present a greedy centralized scheduling strategy and demonstrate that the importance of efficiency varies with respect to the characteristics of the workload a scheduler encounters. Desell et al. [6] show that the application malleability provides up to a 15% speedup over component migration alone on a dynamic cluster environment. Kobe et al. [12] present a distributed agent-based task mapping for malleable applications supporting also self-organization. The agent is assigned at run-time to a random core when a new application arrives having the disadvantage of a possible communication bottleneck when a randomly selected already occupied core serves the new request.

In this work we present a distributed run-time resource management framework for malleable applications. The novelty of the paper is threefold: (i) it takes into account platform's heterogeneity by best utilizing any cores' types thus offering flexibility; (ii) it has a small overhead in overall core intercommunication; and (iii) it has been integrated as a run-time service on a real many-core platform.

3. METHODOLOGY FRAMEWORK

The goal of the proposed methodology framework is to perform run-time resource management on many-core platforms, both homogeneous and heterogeneous, for parallel applications in a distributed way. According to the parallel job classification scheme presented in [8] we can separate parallel applications into two categories based on their capabilities. *Moldable applications* can be stopped at any point but the number of processors that occupy cannot be changed during run-time. *Malleable applications* can be stopped at any point of execution at run-time and they have the flexibility to change the number of assigned processors at run-time.

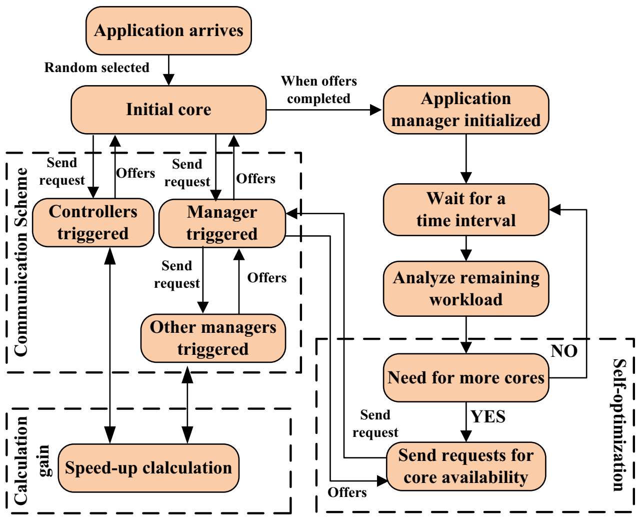

Figure 1: Overall flow of the proposed methodology.

The proposed framework is designed for malleable applications. Even though it can support both moldable and migratable ones, it best utilizes the platform available resources for malleable applications. In this work each core can have one of the following roles concerning the resource management of the platform: *(i) initial core*; *(ii) controller core*; and *(iii) manager core*. The *initial core* is randomly chosen and triggered when a new application arrives in the system being responsible for determining the cores on which the application will start running. The *controller core* is responsible for handling all the unoccupied cores of a certain region of the platform. It is defined at the initialization of the platform and cannot be changed in run-time. It also maintains a list of all manager cores that possess a core in its region. This information is provided to initial or manager cores if and when it is asked for. Last, the *manager core* manages an application searching for new cores and instructing the resizing of the application whenever it has more or less cores to run on. This core does not execute any part of the application. If it does not possess any other cores for the actual application to run on, a self optimization is necessary in order to for the manager to acquire at least one working node. Although an initial core can be a manager one and vice versa, a controller core cannot change its functionality.

An overview of our methodology framework in terms of node intercommunication is presented in Figure 1. Once a new application arrives on a random core (this is the initial core), this core sends messages to controller cores found near it and asks for an available core to serve as the actual application manager. Then the controller cores search into their area for any unoccupied cores and also send requests to any managers into this area. According to the application's characteristics, cores with the appropriate type respond, provided that the speed-up gain of the requesting application is greater than the speed-up loss of the offering manager. After that, the initial core receives all offers and determines the new manager which is initialized by a signal. Then, the new manager distributes evenly the workload to

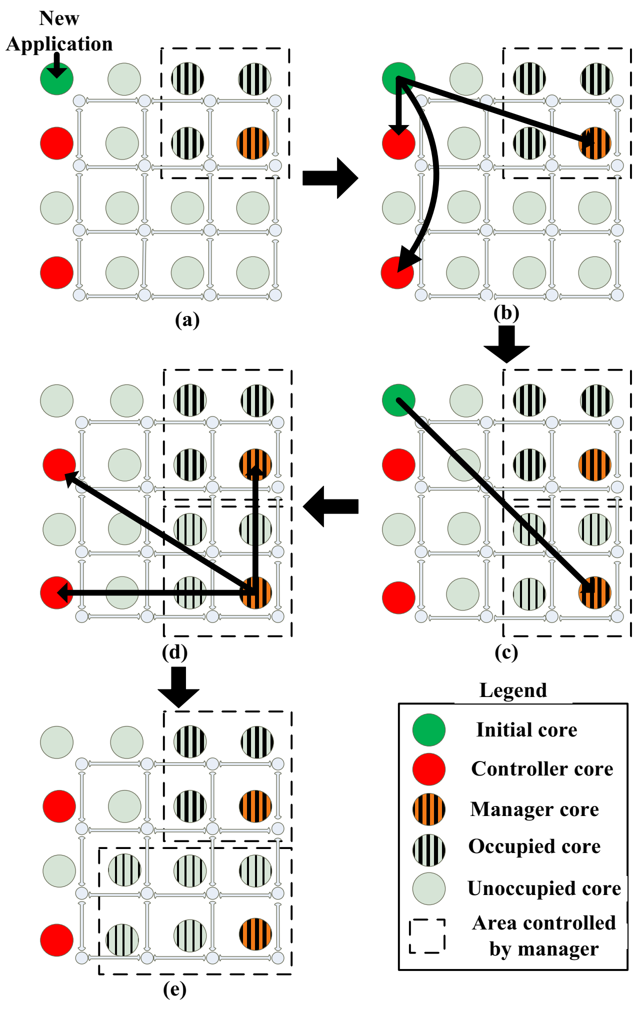

Figure 2: Example of the communication scheme.

the working node he manages. After a predefined time interval the manager, if there is a need and the application is not close to its finish, requests for more cores in order to increase application's performance giving to the framework a self-optimization aspect.

3.1 Definitions

The application and the speed-up model used in this work is the common malleable application model described in [7, 5] and used by other distributed approaches [12]. Each application is described by four parameters W, var, A and Q, where W is the workload, var is the parallelism variance, A is the average parallelism and Q is most preferred Processing Element (PE) type that the application is supposed to be executed on. A many-core platform topology and its communication infrastructure can be uniquely described by a strongly connected directed graph $P(I, N)$. The set of vertices N is composed of two mutually exclusive subsets N_{PE} and N_C containing the available platform's PEs and the platform's on-chip interconnection elements (C), such as routers in an NoC technology. Each platform's PE can be of a specific type and differ from the other platform types (supporting heterogeneous platforms) or all PEs can be of the same type and thus have the same functionality (homogeneous platform). In our framework $T_{pe_i} \forall pe_i \in N_{PE}$ specifies the type of the PE pe_i. In an heterogeneous platform, the T_{pe_i} varies for each PE while in an homogeneous platform T_{pe_i} is the same for all PEs. In order to comply with the application's requirements in terms of required PE classes and have the best $Q \rightarrow T_{pe_i}$ utilization, we define $Util[pe_i] \in [0,1]$ that implies how good the $app(W, var, A, Q)$ is served by the pe_i with T_{pe_i} type. $Util[pe_i]$ can also be considered as a priority factor when an application is asking for additional cores, meaning that when $Util[pe_i] = 1$ we want the best match while when $Util[pe_i] = 0$ the application is not requesting any specific T_{pe_i}. Last, we define the sets F and $offers[]$, which describe all the nodes that can appropriately serve an application based on their type and all the cores offered to the application respectively.

3.2 Communication Scheme

In order to be consistent with the rest of the document and with Algorithms 1 and 2 we declare that the index dst specifies the manager core that is requesting more resources while src is the manager core that is offering them. Also, we define the set R which contains, for each controller core, the PEs in a manhattan distance specified by the equation 1 where $size$ is the size of the platform and $num\_controllers$ is the number of controllers cores on it on the X dimension.

$$distance = size/num\_controllers_X \qquad (1)$$

Algorithm 1 Communication scheme algorithm

```
    // Initial core actions
 1: $analyze(W, var, A, Q)$
 2: $req\_send(control[], core\_id, app, R)$
 3: $start\_timer()$
 4:     $offers[] = receive\_offers$
 5: $end\_timer()$
 6: $sel_{pe} = best\{offers[]\}$
 7: $initialize\_manager(sel_{pe}, offer, R)$

    // Controller core actions
 8: $analyze(W, var, A, Q, R)$
 9: for each ($N_{PE} \in \overline{F}$ && $N_{PE} \in R$)
10:     If $calculate(gain(app)) > 0$ // Algorithm 2
11:         $offers[] = offers[] + new\_offer$
12: $send\_offers(offers, core\_id)$

    // Manager core actions
13: // Actions for offering cores
14: $analyze(W, var, A, Q, R)$
15: If $calculate(gain(app)) > 0$ // Algorithm 2
16:     $offers[] = offers[] + new\_offer$
17:     $send\_offers(offers, core\_id)$
18: // Actions for self-optimization
19: while ($app(W, var, A, Q)! = finished$) {
20:     $analyze(W, var, A, Q, offers)$
21:     $timer()$
22:     If (($app.threshold = max$) || ($app.left\_time < timer()$))
23:         $continue$
24:     else
25:         $req\_send(control[], R)$
26:         $start\_timer()$
27:             $offers[] = offers[] + receive\_offers$
28:         $end\_timer()$
29: end while
```

As aforementioned, when a new application arrives on a core, the initial core task is executed and the communication inside the platform takes place in order to establish a manager core for the application. The communication between initial, controller and manager cores is described in Algorithm 1.

Initial core (lines 1-7): When a new application arrives on an initial core (Figure 2a), this core analyzes the application's characteristics, sends a message to the controllers and managers (Figure 2b) that are inside its region R and fires a timer in order to check for their responses. After the end of the timer, the initial core selects the best offer and sends a signal, according to the offer, and starts the initialization of the manager that will handle the application (Figure 2c).

Controller core (lines 8-12): A controller cores has a variety of responsibilities. Besides maintaining regional information about the managers existing on its region, it has to provide this information to any cores requesting it. It also informs these cores about the position of controller cores in other regions. When the controller receives the signal form the initial core, it analyzes the application and starts to find cores to offer. The controller core can offer any unoccupied core he owns, inside the region R of the initial core, provided that it serves the application's characteristics.

Manager core (lines 13-29): The manager core has two tasks. During the first one, the manager checks if it can offer a core to the new application without loosing more in terms of application speed-up than the gain that the new application will have with the new addition. The second task has to do with the self-optimization process of the already running application. Specifically, there is a time threshold in which the manager checks if the application has taken all the necessary resources it needs or it is near to its completion. If the application has maximized its speed-up [7] there is no need to search for more cores. The same happens if the application is almost finished. In other words, if the remaining time of the application is less than the time interval there is no need to search for more cores. Otherwise, the application enters a self-optimization phase and the manager follows the same communication scheme and sends a message to the controllers (Figure 2d) that are inside its region R and fires a timer in order to check for their responses. After the end of the timer, the manager core checks the offers form the controller cores or from any other manager cores and starts the resize of the application (Figure 2e). Both the controller and manager cores use a function (lines 10 and 15 respectively) in order to calculate the gain or the loss to the application speed-up when offering a core (Algorithm 2).

3.3 Gain calculation

Algorithm 2 describes the required steps that both the controller and manager cores take in order to decide which cores should be offered when an application starts its self-optimization process asking for cores. It has three discrete parts: (i) the actions regarding the calculation of the speedup of the destination node requesting application (lines 4-8); (ii) the actions regarding the calculation of the speedup of the source node offering application (lines 9-13); (iii) and the calculation of the final total gain of this core trade (lines 14-21). During the actions regarding the destination node, for each $N_{PE} \in ((PE_{src} \cap R \cap F) - offers[])$ we calculate the speed-up of the application taking into account the core utilization ($Util[N_{PE}]$) in order to offer to the application the best choices in terms of PE type. Once the speed-up is calculated we check whether the gain of adding the new core to our working set results to an overall gain for the application. On the other side, the actions regarding the source node check whether the loss of a core results in a bigger performance degradation on an already running application. In order to verify it, we calculate the loss speed-up both in terms of performance power and in terms of the $Util[N_{PE}]$ that are occupied and are needed by the application. Since both destination and source actions are greedy, the source offers cores to the destination only when the gain of the destination is bigger than the loss of the source.

Algorithm 2 Gain calculation algorithm

```
 1: $offers[] = \emptyset$
 2: while ($gain > 0$) {
 3:    for each $N_{PE} \in ((PE_{src} \cap R \cap F) - offers[])$ {
    // Actions regarding the destination node
 4:       $PE_{dst} = PE_{dst} \cup N_{PE} \cup offers[]$
 5:       $ord\_PE_{dst} = order\{PE_{dst}\}$
 6:       for $pos = 1$ to $ord\_PE_{dst}.length()$ {
 7:          $SP_{dst} = Util\{ord\_PE_{dst}[pos]\} * (SP[pos] - SP[pos - 1])$
 8:       $gain_{dst} = SP_{dst} - previous\_SP_{dst}$
    // Actions regarding the source node
 9:       $PE_{src} = PE_{src} - offers[] - N_{PE}$
10:       $ord\_PE_{src} = order\{PE_{src}\}$
11:       for $pos = 1$ to $ord\_PE_{src}.length()$ {
12:          $SP_{src} = Util\{ord\_PE_{src}[pos]\} * (SP[pos] - SP[pos - 1])$
13:       $loss_{src} = previous\_SP_{src} - SP_{src}$
    // Calculate total gain
14:       $gain\_temp = gain_{dst} - loss_{src}$
15:       if (($gain\_temp > gain$) || (($gain\_temp = gain$) && $D(manger, N_{PE}) < D(manger, prev\_N_{PE})$))
16:          $gain = gain\_temp$
17:          $prev\_N_{PE} = N_{PE}$
18:    end for
19:    if ($gain > 0$)
20:       $offers[] = offers[] \cup N_{PE}$
21: end While
```

4. EXPERIMENTAL RESULTS

In order to validate our framework we have performed extensive simulation experiments in two steps: (i) using a C-based simulator (Section 4.1) and (ii) integrating the framework on the Intel Single Cloud Chip (SCC) many-core platform [10] (Section 4.2). In both cases, we compared the performance of the presented framework to the state-of-art distributed run-time resource manager DistRM [12]. As malleable applications input we have used the benchmarks provided by the parallel workload archive [8] and produced a file representing scenarios of applications arriving to our system. Each scenario consists of the time the application arrives on the system, its parameters as defined in section 3.1 and a random workload.

4.1 Evaluation on C simulator

Firstly, as aforementioned, we have developed a C-based simulator capable of simulating the behavior of malleable applications and all the necessary actions required by the on-chip communication scheme. For more accurate simulation, every node was represented by a different process. The inter-node communication is implemented using traditional Linux signals, while messages are passed using a pipe between the sender and the receiver. For synchronization semaphores are used. The goal of this step was to have a quick and abstract view of our methodology and estimate the cost of the developed distributed on-chip communication scheme. Also, the simulation method offers the capability of functional error correction and fast debugging in temps of possible communication deadlocks. The simulator sup-

Table 1: Comparison of the proposed technique to the DistRM [12] in the C simulator.

	Msg. cnt.		Msg. size		Avg. sp.		Comp. eff.	
Plat.	Number of applications							
sizes	32	64	32	64	32	64	32	64
6x6	72.3	71.4	73.2	72.4	3.8	13.8	86.8	86.6
8x8	64.3	63.4	64.8	63.6	4.5	9.3	83.3	83.0
12x12	49.1	52.3	44.8	48.7	-1.4	1.5	66.0	68.3
16x16	42.6	45.8	40.0	41.4	0.5	3.1	61.3	64.9
20x20	32.6	36.1	27.7	30.5	3.1	2.7	53.4	57.8
24x24	25.1	27.2	18.5	19.2	2.0	2.4	50.1	51.7
28x28	20.6	22.3	13.5	14.1	1.5	2.8	42.7	48.7
32x32	17.9	14.6	9.9	2.5	1.6	2.8	41.9	42.4
Average	41%		37%		3%		62%	

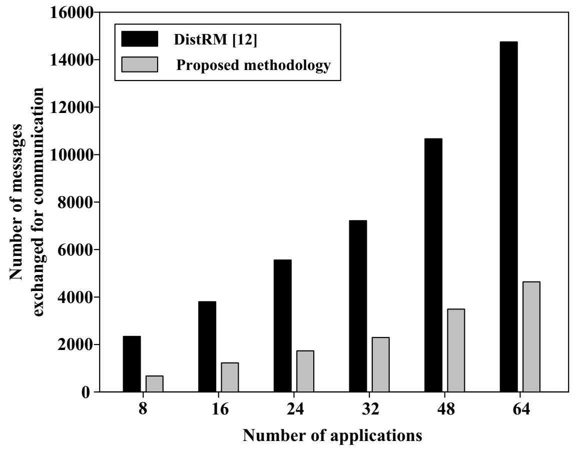

Figure 3: Total number of messages sent for inter-communication by all nodes for various applications.

ports big topologies (up to 32×32 core system), numerous application inputs.

Table 1 presents the results of the comparison of the proposed technique against the DistRM [12] distributed run-time manager. We evaluated the two run-time managers for various platform platform sizes, from 6 × 6 up to 32 × 32, and for 32 and 64 applications. The comparison metrics are: (i) message count (*Msg. cnt.*), which is the total number of messages sent by all nodes during the whole duration of the simulation both for resource management and application execution; (ii) message size (*Msg. size*), which is the total size of sent messages; (iii) application average speed-up (*Avg. sp.*); and (iv) computational effort (*Comp. eff.*), which is the total number of speed-up function calls. Table 1 presents the *percentage gains* of the presented framework in comparison to DistRM. Concerning the message count, simulation results showed an average gain of 41% for the presented methodology due to the fact that the core request messages are sent only inside the area R while DistRM sends messages in many areas, smaller than R and probably overlapping, thus increasing the number of messages used for sending requests and receiving answers. Also, the total size of these messages, measured in *bytes*, for our framework is on average 37% smaller that then size needed by DistRM.

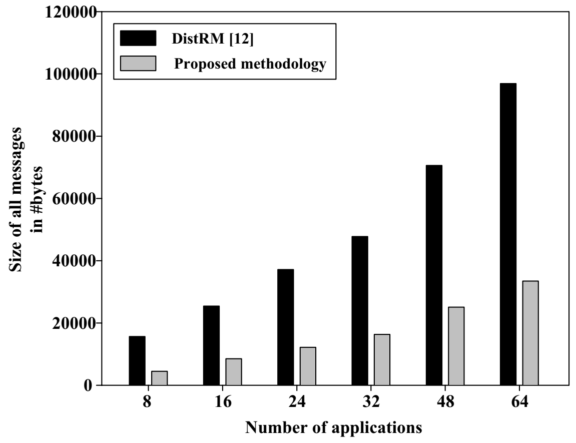

Figure 4: Total size of sent messages in *bytes*.

Thus, the network burden of our framework is on average 38% smaller. In terms of average application speed-up, the proposed framework achieves on average 3% better results than DistRM. The speed-up function used for this metric is the application speed-up function presented in [12]. Last, the gain of our methodology regarding the total computational effort is on average 62% compared to the DistRM.

4.2 Evaluation on a many-core platform

After the first evaluation of our framework in the C simulator, we integrated the whole framework as a run-time service on the many-core Intel SCC platform [10] in order to test our framework on a real platform and not only at the simulation level. The Single-Chip Cloud Computer (SCC) experimental processor [10] is a 48-core “concept vehicle” created by Intel Labs as a platform for many-core software research. Since the platform size is fixed (48 cores) we compared the performance of our distributed run-time manager to the DistRM [12] for various number of applications running on the platform (from 8 to 64).

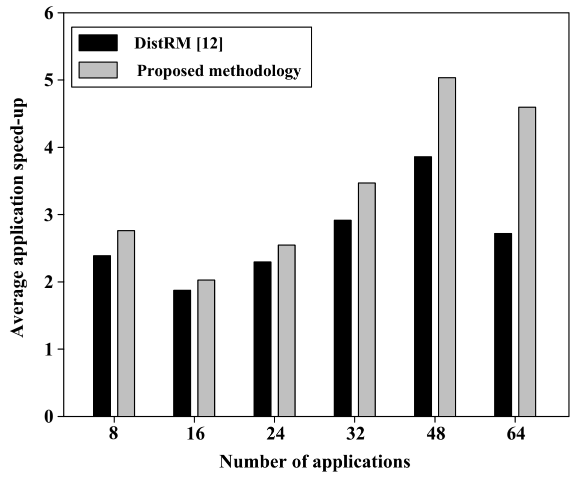

Figure 5: Average application speed-up using the speed-up function presented in [12].

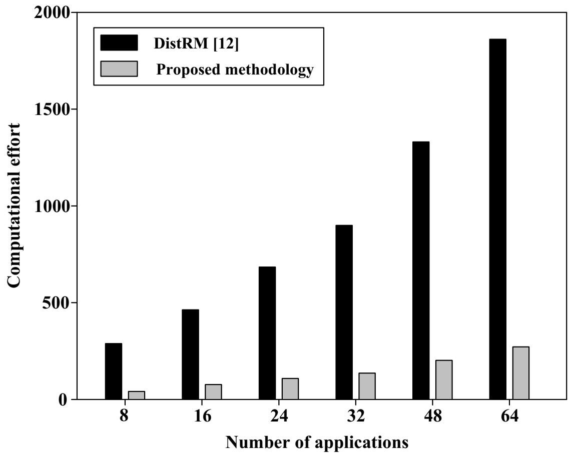

Figure 6: Computational effort comparison.

Figures 3 and 4 present the total number of messages sent by all nodes during the whole duration of the simulation and the total size of these messages in *bytes* respectively. The messages are used for application initialization, self-optimization and application resizing by manager cores and application execution. The proposed framework sends on average 70% less messages in order to perform all the necessary actions since the messages are sent only inside the area R. Whereas, DistRM searches for available cores in more areas, smaller than R, thus increasing the number of messages used for sending requests and receiving answers. Another reason that the presented algorithm has less messages than DistRM, is the fact that the framework performs application self-optimization under very specific criteria, only when the application has not maximized its speed-up or it is not near to its completion. Also, the size of the messages sent are on average 64% less than the ones used by the DistRM. The size is not proportional to the number of the messages since each message varies in size. For example, an offer message about four cores can have up to 20 *bytes* while the answer to this offer is 1 *byte*. Last, the proposed framework burdens network resources 66% less than DistRM.

Figure 5 presents the average application speed-up using the speed-up function presented in [12]. Speed-up is defined as the ratio of the total number of turnarounds performed for all applications divided by the total workload. The presented framework achieves on average 20% better application speed-up than DistRM. This can be explained by the fact that in the presented framework cores are not disturbed so often by messages during their application execution and thus completing the applications faster. On the other hand, due to the large number of messages sent by the application agents in DistRM, cores stop their functionality more frequently in order to answer to these messages, thus delaying the execution.

Figure 6 shows the comparison of the computational effort between the presented framework and DistRM. Computational effort is defined as the total number of speed-up function calls during the whole simulation. The presented framework has on average 85% less speed-up function calls than DistRM. This can be explained by the fact that, as aforementioned, the presented framework performs less application self-optimizations due to specific criteria. So the manager cores calculate the speed-up function less frequently.

5. CONCLUSIONS

In this paper we presented a distributed run-time manager for malleable applications. The presented framework has a small communication overhead, takes into account platform's heterogeneity and makes sure that the application will maximize its speed-up function. Our framework has been implemented as part of a `C` simulator and additionally as a run-time service on a real many-core platform and we compared it against the DistRM [12] state-of-art run-time resource manager. Experimental results showed that our framework has on average 70% less messages, 64% smaller message size and 20% application speed-up gain.

6. REFERENCES

[1] M. A. Al Faruque et al. Adam: run-time agent-based distributed application mapping for on-chip communication. In *Proc. of DAC*, pages 760–765. ACM, 2008.

[2] I. Anagnostopoulos et al. A divide and conquer based distributed run-time mapping methodology for many-core platforms. In *Proc. of DATE*, pages 111–116, 2012.

[3] A. Beguelin et al. Application level fault tolerance in heterogeneous networks of workstations. *J. Parallel Distrib. Comput.*, 43(2):147–155, June 1997.

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

[5] D. Feitelson. Parallel Workloads Archive, `http://www.cs.huji.ac.il/labs/parallel/workload`.

[6] T. Desell et al. Malleable applications for scalable high performance computing. *Cluster Computing*, 10(3):323–337, 2007.

[7] A. B. Downey. A model for speedup of parallel programs. Technical report, 1997.

[8] D. G. Feitelson and L. Rudolph. Toward convergence in job schedulers for parallel supercomputers. In *Proc. of JSSPP*, pages 1–26. Springer-Verlag, 1996.

[9] K. Goossens et al. Æhereal network on chip: Concepts, architectures, and implementations. *IEEE Des. Test*, 22(5):414–421, 2005.

[10] J. Howard et al. A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In *Proc. of ISSCC*, pages 108 –109, feb. 2010.

[11] Intel. The Intel® Xeon Phi^{TM} Coprocessor, `http://www.intel.com/content/www/us/en/high-performance-computing/high-performance-xeon-phi-coprocessor-brief.html`.

[12] S. Kobbe et al. DistRM: distributed resource management for on-chip many-core systems. In *Proc. of CODES+ISSS*, pages 119–128. ACM, 2011.

[13] S. Kounev et al. Towards self-aware performance and resource management in modern service-oriented systems. In *Proc. of SCC*, pages 621–624. IEEE CS, 2010.

[14] V. Nollet et al. Centralized run-time resource management in a network-on-chip containing reconfigurable hardware tiles. In *Proc. of DATE*, pages 234–239. IEEE CS, 2005.

[15] G. Sabin et al. Moldable parallel job scheduling using job efficiency: an iterative approach. In *Proc. of JSSPP*, pages 94–114. Springer-Verlag, 2007.

[16] L. Seiler et al. Larrabee: a many-core x86 architecture for visual computing. *ACM Trans. Graph.*, 27:18:1–18:15, August 2008.

[17] STMicroelectronics. STNoC: Building a new system-on-chip paradigm. White Paper, 2005.

[18] S. Vangal et al. An 80-Tile 1.28 TFLOPS Network-on-Chip in 65nm CMOS. In *Proc. of ISSCC*, pages 98–589. IEEE, 2007.