Formal Probabilistic Analysis of Distributed Resource Management Schemes in On-Chip Systems

Shafaq Iqtedar*, Osman Hasan*, Muhammad Shafique†, Jörg Henkel†
*School of EE and CS, National University of Sciences and Technology (NUST), Islamabad, Pakistan
†Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Germany
Email: {10besesiqtedar, osman.hasan}@seecs.nust.edu.pk; {muhammad.shafique, henkel}@kit.edu

***Abstract*—New paradigms for managing resources in on-chip many-core systems come with various issues. Among them is the key demand for robust verification of (distributed) resource management (RM) schemes before deployment. Moreover, it is important to have a unified framework where different RM schemes can be formally analyzed and compared for their performance efficiency and robustness. Traditional techniques, like simulation or emulation, are inherently in-exhaustive and thus compromise the completeness and accuracy of the analysis results. In this work, we present a formal approach, based on probabilistic model checking, for evaluating and comparing the performance of different distributed RM schemes. To illustrate the benefits and applicability of our formal verification and comparative analysis approach, we perform a case study on the comparison of two state-of-the-art distributed RM schemes using the PRISM model checker.**

I. Introduction

With current trends in hardware design, on-chip many-core systems [1] have emerged as a new paradigm. These multicore architectures execute highly parallel and resource demanding applications. This leads to the problem of runtime resource allocation, i.e., which application should use which and how many cores to get the most efficient utilization of available resources [2] [3]. These resources can no longer be managed by one central entity (i.e., in a centralized resource management paradigm) due to increased design optimization space of large-scale many-core systems. Therefore, many distributed Resource Management (RM) schemes have been proposed, e.g., [2] [4] [5] [6], over the past few years, to ensure an efficient utilization of the available resources and for maximizing the overall throughput of on-chip many-core systems in a scalable fashion.

Just like any other algorithm, these distributed RM schemes are also susceptible to functional and design errors. But catching all such errors using traditional techniques, such as simulation or emulation [7], can never be guaranteed due to the distributed nature of RM schemes and the in-exhaustive nature of these analysis methods. This is a severe limitation since an uncaught bug in the analysis phase may lead to runtime failures. Especially, when considering distributed RM schemes, the possibilities of mapping decision (i.e., task-to-core mappings) increase exponentially with the number of cores. Furthermore, as depicted in [10], the exhaustive verification of even a simple 32-bit comparator would take about 584,941 years to complete using simulation. Therefore, no matter how intelligent the test-bench and generator are, validating the design intent through simulation [16] is inherently incomplete for large and complex systems. On the other hand, formal verification methods [10] have been well known to solve such problems due to their soundness and completeness. Moreover, it is of great significance to have a single framework where different distributed RM schemes can be compared to each other and evaluated for their design and performance efficiency. Again, given the sampling-based nature of simulation and emulation, the comparison results cannot be considered as completely reliable. Finally, another challenge while analyzing distributed RM schemes is the absence of a global system state knowledge due to their decentralized nature. Therefore, in some cases the resources are allocated in a randomized manner, and with continuous self-optimization, the system attains a stable and efficient configuration (i.e., closer to the mapping quality of centralized schemes). Due to such randomness in the behavior of distributed RM schemes, we have to use probabilistic analysis methods. Simulation-based probabilistic-analysis only ascertains the correctness of the states that are reached in a particular simulation path and as the reachable state-space of the system increases, the limited sampling-based simulation results become less and less accurate.

The accuracy of functional and performance analysis of distributed RM schemes and their reliable comparison is a dire need due to the extensive applications of many-core systems in a variety of safety-critical systems, such as medicine and transportation, and the high performance demands of the present era. As a more accurate and complementary approach to simulation and emulation methods, we propose to employ probabilistic model checking [12], i.e., a formal method for verifying systems that exhibit randomized behaviors, for analyzing and comparing various distributed RM schemes. To the best of our knowledge, *so far no formal probabilistic verification method, with comprehensive quantitative analysis of functional and performance-evaluation properties, has been proposed in the context of verifying distributed RM schemes in on-chip many-core systems*.

A. Our Novel Contributions and Concept Overview

In particular, we have developed a generic discrete time Markovian model for distributed RM schemes, such that most of them can be formally modeled based on our generic model through simple updates. This model can then be used with a probabilistic model checker to formally verify various probabilistic properties that provide very useful insights for the many-core system designer. For instance, *the probability of the event when an application fails to get the desired number of cores* or *the probability of an event when one/more cores will never be used for actual computations*, is a useful piece of information for the distributed RM designers. Thus, statistical knowledge can play a vital role in comparison and performance evaluation of different distributed RM schemes and will help in the detection of unwanted behaviors during early design stages.

We have identified a set of probabilistic properties which are aimed to formally address the following most critical questions posed on distributed RM community:

1) *Quality of mapping decision*, i.e., how close is the quality of mapping decision compared to the centralized approaches.
2) *Communication/computation overhead*, i.e., how the system states and transitions (i.e., messages, requests, calculations, etc.) increase as we increase the number of cores and applications.
3) *Time to a stable system configuration*, i.e., how the number of steps to a stable system configuration increases with the increase in the resource demand.
4) *Identification of other miscellaneous bugs*, i.e., deadlocks, back and forth trading, and other functional design errors.

To illustrate the practical effectiveness and utilization of the above-mentioned contributions, we have employed the PRISM model checker [8] in this paper to formally verify functional and timing properties of two state-of-the-art distributed RM schemes, namely "distributed run-time resource management for malleable applications on many-core platforms" (DRRMM) [5] and "distributed RM for on-chip many-core systems" (DistRM) [2], and thus compare them. The main reason for choosing these two schemes is that both of these recently proposed schemes are highly scalable while achieving comparable results to the centralized RM schemes.

II. Preliminaries

A. Probabilistic Model Checking and the PRISM Model Checker

Probabilistic model checking [13] is an extension of traditional model-checking techniques [9] for the integrated analysis of both qualitative and quantitative properties of systems that exhibit stochastic behavior. A model checker exhaustively traverses the entire state-space of a design in ascertaining correctness. The 100% completeness of analysis coupled with the consideration of randomized behaviors and provision of a detailed quantitative knowledge makes probabilistic model checking quite suitable for the proposed work.

PRISM [8] is a widely used probabilistic model checker. The system to be verified by PRISM is first described as a probabilistic variant of Reactive Modules [11]. PRISM provides support for analyzing discrete-time Markov chains (DTMCs), continuous-time Markov chains (CTMCs) and Markovian Decision Process (MDPs). An advantage of using PRISM is that it allows accurate computation for a wide range of numerical properties and it performs a complete analysis, which is a very useful feature for analyzing the best-and-worst-case scenarios. For instance, in case of distributed RM schemes, the designer can analyze the upper and lower bounds for the average application speedup and the probability of their occurrence. Such quantitative knowledge indicates the usefulness of probabilistic formal analysis in the context of verifying distributed RM schemes.

B. Agent-Based Distributed RM Schemes

The run-time RM schemes often utilize the concept of *malleable applications* [15] that are able to adapt their degree of parallelism to the number of assigned cores dynamically. This means that they are designed in a way that allows them to enlarge or to shrink the set of cores that are used by the application. In a distributed environment, the chip is usually divided into different *regions*. A region of size r is a set of cores that contains a particular center-of-the-region core and all cores that are within a Manhattan distance of r to that core. The resources are handled locally in different regions, rather than being handled globally from a single central point. In order to do so, these schemes often employ the principles of multi-agent systems [14] to perform the resource management. An *agent* is a situated computational entity that makes decisions autonomously. Depending on the RM scheme, the agents can either be associated with different applications or they can be distributed throughout the chip.

III. Markovian Model for distributed RM schemes

We used a set of guarded commands in PRISM to develop a generic model for the distributed RM schemes. A guard is a predicate over all variables of the system and if it is true then a transition takes place with the associated probabilities. A command takes the form:

```
[action] guard-><prob>:<updt1>+...+<prob>:<updt2>;
```

where each *updt (i.e., update)* describes a transition that the module can make if the guard is true. Each update is also assigned a probability, considering the corresponding transition. If no probability is assigned to a transition, then it is implicitly assumed to be 1. A command can also be labeled with *actions*, which force two or more modules to make transitions simultaneously. For example:

```
[] x=1 -> 0.3:(x′=2) + 0.7:(x′=3);
```

The above-mentioned command states that if the expression '$x = 1$' holds true, then ($->$) with probability 0.3, the next value of x (x') would be equal to 2 or with probability 0.7, the next value of x (x') would be equal to 3. If two or more unsynchronized commands are activated simultaneously, then one of them is chosen probabilistically.

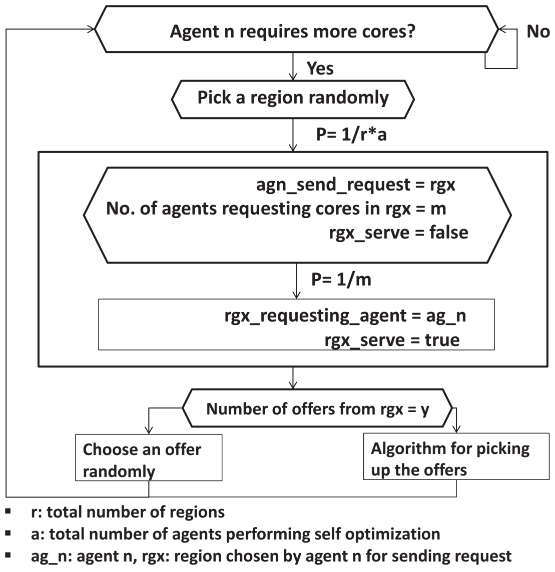
Agent n requires more cores?
No
Yes
Pick a region randomly
P= 1/r*a
agn_send_request = rgx
No. of agents requesting cores in rgx = m
rgx_serve = false
P= 1/m
rgx_requesting_agent = ag_n
rgx_serve = true
Number of offers from rgx = y
Choose an offer randomly
Algorithm for picking up the offers
r: total number of regions
a: total number of agents performing self optimization
ag_n: agent n, rgx: region chosen by agent n for sending request

Fig. 1. Application Self Optimization

Our generic model is composed of the behaviors depicting the applications, agents and regions in a distributed decision making paradigm (e.g., in case of a distributed RM scheme).

A. Modeling Applications and Agents

The characteristics of input applications/agents, like average parallelism (A) of an application, its variance in parallelism (σ), currently occupied cores, etc., can be stored in a set of variables in PRISM. During the execution of algorithm, the values of these variables would be updated in accordance with the underlying conditions. The speedup of an application can be modeled by a set of formulas which comprise of a name and an expression. For example, the speedup calculation formula, given in [17] and used by the distributed RM schemes of [2] [5], can be expressed as follows:

$$\textbf{Formula } curr\_speedup\_1 = n * A/(A + (\sigma/2 * (n - 1));$$

where n is the number of current cores occupied by an application and *formula* is the keyword used in PRSIM to express mathematical formulas. This expression usually varies depending upon the distributed RM scheme. Applications are initialized in a main module from where the agents can send requests to different regions across the chip with a certain probability. Algorithm 1 presents how an agent sends requests to different regions across the chip (Line 5) and how the request is either accepted or kept in the wait queue if the corresponding region is already serving a request (Lines 6-7). For instance, if 3 agents are sending requests in a particular region at the same time, one agent will be picked up with a probability equal to 1/3 and the request of the chosen agent will be served first. This is also illustrated in Fig.1 where an agent chooses a region with a certain probability and then the corresponding region serves the request depending upon the total number of requesting agents. We have not synchronized the commands in which agents send requests for self-optimization (Line 5) and hence PRISM is allowed to probabilistically choose which agent will initiate a request at a particular time interval. This caters for the fact that in some distributed RM schemes, a random delay is chosen to avoid a synchronization of the optimization cycle of different agents such that two agents that had been initialized at the same time will not always try to optimize their set of cores at the same points of time in the future.

B. Modeling Regions

We have modeled regions across the chip as a set of modules running in parallel (Algorithm 2). Each module contains the information of total number of cores in the given region, local unoccupied cores, the occupied cores and the agents/application that occupy cores within the region. Lines 1-3 of Algorithm 2 show the initialization of the cores of different agents inside a region. The initialization values (i.e., $rg1\_ag1\_init$, $rg1\_ag2\_init$, etc.,) are provided later on when model checking is performed. The size of a region (i.e., the total number of cores inside a region) may vary in different RM schemes. In this way, the designer can analyze the effects of the size of region on the performance of the given distributed RM scheme. Note that the size of the region would remain constant in a particular run of model checking. The applications or their agents can send signals requesting cores inside a region. The necessary computations will be performed inside the region module as the agents within the region will respond to the request.

Algorithm 2 presents how different agents respond to the request of a new application. We have considered two cases:

1) *Agent0* is shown as a manager of unoccupied cores. Therefore, it simply relinquishes the cores to the requesting agent without calculating the speedups (Line 4). At startup, we initiate this agent with the total number of available cores as no application is running on the chip. Depending on the distributed RM scheme, multiple agents can be initiated in different regions to handle the unoccupied cores.
2) *Agent2* is shown as a manager of some running application. Therefore, it performs the speedup calculation before losing its own cores to the requesting agent. (Line 5)

Line 8 of Algorithm 1 is synchronized with Line 5 of Algorithm 2 by using the same label, i.e., '$[rgi\_agentj\_offered\_k]$'. This allows an agent to update the value of its total current cores whenever it occupies more cores inside a new region. Line 8 of Algorithm 1 executes whenever the requesting agent occupies new cores, inside any region, relinquished by other agents that have cores within the same region. Similarly, Line 7 of Algorithm 1 is synced with Line 4 of Algorithm 2. This allows an agent to update the value of its total number of current cores whenever it receives any cores from the manager of unoccupied cores inside any region.

Customization for Different Distributed RM Schemes: The main motivation of developing a generic Markovian model for distributed RM schemes, described above, is to facilitate the construction of a Markovian model of any arbitrary RM scheme. The customization step requires simple modifications. For instance, the speed up formulas would usually vary for different distributed RM schemes. Similarly, if the agents of distributed RM scheme only send requests inside a region r, we can restrict Line 5 of Algorithm 1 to be executed only once and then the value of region would remain constant throughout the life of the application. Moreover, we have shown that the agents perform self-optimization under very specific criteria, i.e., only when the application has not maximized its speed-up. If this is not the case and the agents keep on sending requests throughout their lives, then we can simply remove some conditions from the guard of Line 5 in Algorithm 1. In a similar manner, our generic model can be tailored for most of distributed RM schemes and for illustration purposes, we will present the formal modeling of two state-of-the-art distributed RM schemes in Section V.

IV. Probabilistic Properties of Distributed RM Schemes

In this section, we provide a set of probabilistic properties that can be used to formally analyze and compare the functionality and performance of any arbitrary RM scheme.

Algorithm 1 Application self-optimization

```
Module optimize
1: $agent1\_current\_cores : [0..agent1\_max\_cores];$
2: $agent1\_send\_request : [0..maxRegion];$
   ⋮    //The region in which agent 1 is sending request.
          The initial value 0 means no region is chosen as yet
   $agentn\_current\_cores : [0..agentn\_max\_cores];$
   $agentn\_send\_request : [0..maxRegion];$
3: $rg1\_serve : bool; . . . rgn\_serve : bool;$
                    //Is region 1 already serving a request?
4: $rg1\_requesting\_agent : [0..maxAgents];$
   ⋮          //the agent requesting resources in region 1
   $rgn\_requesting\_agent : [0..maxAgents];$

5: $[]\ agent1\_send\_request = 0\ \&\ agent1\_demand! = 0 -->$
   $1/total\_region : (agent1\_send\_request' = 1)+$
   $1/total\_region : (agent1\_send\_request' = 2)+$
   ⋮                        //if an agent require more cores
                            then pick a random region
   $1/total\_region : (agent1\_send\_request' = n);$
6: for region i= 1 to n
   $[]\ agent1\_send\_request = i\ \&\ rgi\_serve = false -->$
   $(rgi\_serve' = true)\ \&\ (rgi\_requesting\_agent' = 1);$
   ⋮                   //if region i is not serving any request
                  then send a request signal
   $[]\ agentn\_send\_request = i\ \&\ rgi\_serve = false -->$
   $(rgi\_serve' = true)\ \&\ (rgi\_requesting\_agent' = n);$
   End for
7: for region i= 1 to n
   let agent 0 be the manager of unoccupied cores
   Let j be the requesting agent
   $[rgi\_agent0\_offered\_j]\ true -->$
   $(agent0\_current\_cores' = agent0\_current\_cores-$
   $min(agentj\_required\_cores, rgi\_agent0\_cores))$
   $\&\ (agentj\_current\_cores' = agentj\_current\_cores+$
   $min(agj\_required\_cores, rgi\_agent0\_cores));$
             //if the cores are offered from agent 0 in region
               i to agent j then update the current cores
   End for
8: for region i= 1 to n
   for each agent j= 1 to n, inside region i
   Let k be the requesting agent
   $[rgi\_agentj\_offered\_k]\ true -->$
   $(agentj\_current\_cores' = agentj\_current\_cores - 1$
   $(agentk\_current\_cores' = agentk\_current\_cores + 1$
             //if the cores are offered from agent j in region
               i to agent k then update the current cores
   End for
   End for
endmodule
```

A. Functional properties

Functional properties of a RM scheme can be expressed in probabilistic Linear Temporal Logic (LTL). For instance, we can determine the probability that *eventually an application will acquire the maximum desired number of cores for optimizing its speedup*, as follows:

$$P =?\ [F\ agentn\_current\_cores = agentn\_max\_cores]$$

Similarly, for the comparison of average application speedup, we can calculate the probability that *eventually the average application speedup will be greater than or equal to a certain threshold.*

$$P =?\ [F\ averageSpeedup >= Threshold]$$

Another interesting aspect for the designer can be the analysis of resource utilization in comparison with demand. It can be useful to know, if there are any cases in which even though the demand is greater than

Algorithm 2 A Region serving an Agent's request

```
Module region_i
1: $rgi\_agent0\_cores : [0..total\_cores\_in\_region] init\ rgi\_ag0\_init;$
2: $rgi\_agent1\_cores : [0..total\_cores\_in\_region] init\ rgi\_ag1\_init;$
    :
    :                    //the cores of different agents inside region 1
3: $rgi\_agentn\_cores : [0..total\_cores\_in\_region] init\ rgi\_agn\_init;$
4: for requesting agent j= 1 to n
    let agent 0 be the manager of unoccupied cores
    $[rgi\_agent0\_offered\_j] rgi\_agent0\_cores > 0$ &
    $rgi\_serve = true \& rgi\_requesting\_agent = j -->$
    $(rgi\_agent0\_cores' = rgi\_agent0\_cores -$
    $min(agentj\_required\_cores, rgi\_agent0\_cores))$
    & $(rgi\_agentj\_cores' = rgi\_agentj\_cores +$
    $min(agj\_required\_cores, rgi\_agent0\_cores));$
    End for
            //agent0 offering the unoccupied cores to agenti
5: for offering agent j=1 to n
    for requesting agent k=1 to n
    $[rgi\_agentj\_offered\_k] rgi\_agentj\_cores > 0$ &
    $rgi\_serve = true$ & $rgi\_requesting\_agent = k$ &
    $((gain\_speedup\_k) - (loss\_speedup\_j) > 0) -->$
    $(rgi\_agentj\_cores' = rgi\_agentj\_cores - 1)$
    $(rgi\_agentk\_cores' = rgi\_agentk\_cores + 1);$
    End for
    End for
     //if gain in speedup of agent k is greater than the loss
   in speedup of agent j, agentj will offer the core to agentk
endmodule
```

the available resources but the resources are still not utilized for actual computations. Thus, we can calculate the probability that *eventually all the cores will be utilized for actual computation*, as follows:

$$P = ?\ [F\ unallocatedCores = 0]$$

B. Performance evaluation properties

Performance properties can be verified in PRISM by augmenting the models with costs and rewards, i.e., real values associated with certain states or transitions. Using this feature, the designer can reason about a wide range of quantitative measures relating to the behavior of distributed RM schemes. For instance, we can utilize this feature for finding the number of requests initiated by an application/agent for self-optimization. In order to do so, we first have to extend the model with rewards.

rewards "num_requests_by_agentn"
$[]agentn\_send\_request = 0$ & $agentn\_required\_cores! = 0 : 1;$
endrewards

The above-mentioned reward structure assigns a real value of 1 to all transitions from the state(s) which satisfy the guard, i.e., *'agentn_send_request = 0 & agentn_required_cores !=0*. The reward will be accumulated over time and in order to calculate the cumulative reward after certain time interval, we can express the following property:

$$R\ \{\text{"}num\_requests\_by\_agentn\text{"}\} = ?\ [C <= 100]$$

which would return, for a given state of the model, "*the expected number of requests initiated by agent n within 100 time units of operation*". Thus, the designer can calculate the communication/computation overhead by assigning real values to different transitions of the model. Similarly, we can use the reward structure for evaluating timing properties. For example, *expected time required to reach a state where an agent has acquired the maximum desired number of cores, from a state s*, can be obtained by:

$$R = ?\ [F\ agentn\_current\_cores = agentn\_max\_cores]$$

The above-mentioned property can be expressed in different ways to evaluate and compare the number of steps to a stable configuration in different distributed RM schemes.

In order to evaluate and compare the quality of mapping decision, the designer can be interested in analyzing the average applications speedup over time. This can be done by augmenting the model with rewards:

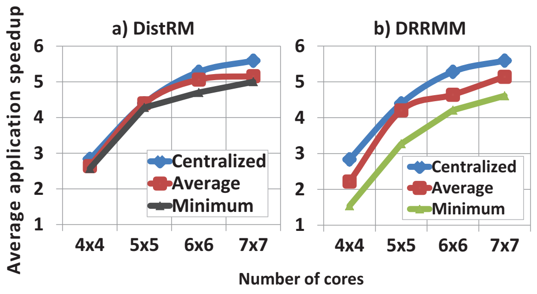

Fig. 2. Average application speedup

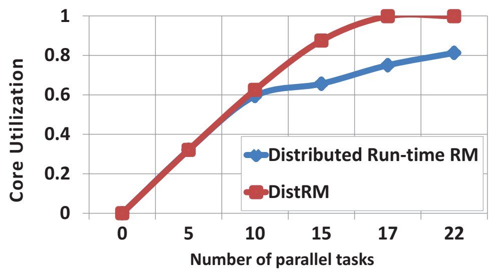

Fig. 3. Average core Utilization

rewards "average_application_speedup"
true : *expression/formula for average speedup*;
endrewards

The above-mentioned reward structure assigns a real value equal to the average application speedup to every state of the given model. This time we will not calculate the accumulated reward, as done in the previous two cases, but rather the value of this reward in the long-run or steady state. This can be done by expressing the following property which will return the *long-run average application speedup*

$$R\ \{\text{"}average\_application\_speedup\text{"}\} = ?\ [S]$$

where S implies the steady state. The properties, mentioned in this section, can be specialized to formally analyze and compare most of the distributed RM schemes, as will be depicted in the next section.

V. Evaluation and Results

For illustration purposes, we used the proposed generic model, given in Algorithm 1 and 2, to formally analyze and compare the following two state-of-the-art distributed RM schemes:

1) Distributed RM for on-chip many-core systems (DistRM) [2]
2) Distributed run-time RM for malleable applications on many-core platforms (DRRMM) [5].

A. Formal modeling of DistRM [2]

DistRM [2] is a fully decentralized agent-based RM scheme for on-chip many-core systems. In distRM, a dedicated agent per application is used as a resource manager. The agent randomly chooses regions on the chip and tries to allocate cores there. The actual agents, currently located inside regions, evaluate which of their own cores could be given to the requesting agent. They send their offer (containing information about their own loss in speedup) back to the requesting agent. All offers that help increasing the speedup of the own application are taken, as long as the speedup increases more than the speedup for the offering agent's application decreases. As an agent can send requests outside the initial region, we added the following behavior in the generic code, described in Section 3, to set the *selected-region* back to *null* once the request has been served by a particular region:

$[rgi\_release\_requesting\_agent]\ rgi\_requesting\_agent = j$
$-> (agentj\_send\_request' = 0)$ & $(rgi\_serve' = false)$
& $(rgi\_requesting\_agent' = 0);$

In this way, an agent can randomly choose a new region for future requests (see Line 5 of Algorithm 1). Moreover, since the agents in DistRM keep

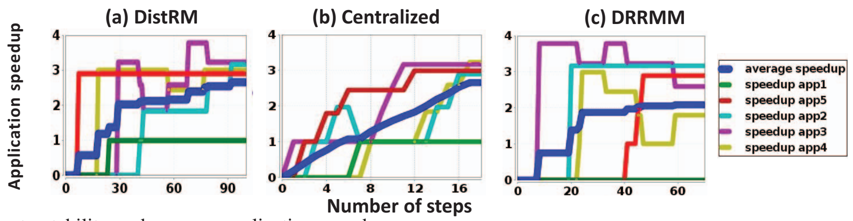

Fig. 4. Comparison of steps to stability and average application speedup

on looking for cores even when the application has optimized its speedup, we removed all the commands that prevent from doing so from our generic code described in Section 3. The speedup formulas of our generic code are replaced by the speedup model used in distRM. Moreover, in distRM, an *Idle agent* is located in every region for handling the unoccupied cores. The implementation of these agents is already explained in Section 3.

B. Formal modeling of DRRMM [5]

Our second chosen RM scheme [5] is also a fully distributed RM scheme but instead of agents, it rather utilizes the concept of controller and manager cores. The controller cores and manager cores have similar functionality as that of idle agents and application agents in distRM, respectively. Therefore, they can be modeled in a very similar manner. Whenever there is a new application, a core inside a random region is selected as its initial core. The requests for more cores are only sent inside the initial region. Therefore, once a random region is chosen at start-up, we keep it constant throughout the execution. Moreover, the requests are only initiated if the speedup of an application is not already optimized. Therefore, we strengthened the guards of all the commands in which the requests are initiated, to fulfill this criterion.

C. Experimental Setup

We used the version 4.1 of PRISM model checker along with Windows 7 professional OS running on a core i5-3210 CPU at 2.50 GHz with 8.00 GB of RAM. The verification is done starting from a 3x3 to upto a 7x7 grid using probabilistic model checking. In order to compare our results with a centralized scheme, we have also modeled a centralized-resource mapper using the speedup functions and greedy algorithm used in [2].

D. Results and Discussion

1) Comparison of Average Applications Speedup and Steps to Stability: The most interesting property in the analysis of distributed RM schemes is to verify that eventually the agent negotiation would result in an efficient resource distribution that is closer to the mapping quality of centralized schemes. Thus, maximizing the speedups of all applications and the overall average application speedup. Moreover, it is also important that the system attains this stable configuration (i.e., when the agents are no longer sending requests for more cores and the best possible distribution of cores has been attained) in minimum number of steps. In order to analyze this behavior, we performed a detailed analysis on different grid sizes and for the three considered RM schemes. For the calculation of average application speedup, we used reward based properties. PRISM by default only returns the value of reward for the single initial state. Therefore, in order to calculate the value of average-application speedup for all possible states we applied filters [8].

$$filter(min, R\,\{\text{“}average\_speedup\text{”}\} =?\ [S])$$
$$filter(avg, R\,\{\text{“}average\_speedup\text{”}\} =?\ [S])$$
$$filter(max, R\,\{\text{“}average\_speedup\text{”}\} =?\ [S])$$

The above-mentioned filters give the minimum, average and the maximum values of rewards over all possible states of the model, respectively. These properties give the value of average applications speedup in steady state. We evaluated 9 such properties for every grid-size (i.e., 54-properties) in order to compare the performance of the three RM schemes. These upper and lower bounds can never be attained using simulation-based techniques. The authors of [2], evaluated the RM schemes for only 100 runs/configuration. Whereas, using model checking we have exhaustively verified over all possible system states and thus, identified the range in which average speedups would fall. The results of this analysis are presented in Figs.4 and 2. Fig.4 presents how the applications interact with each other to optimize their speedups overtime. Fig.2 presents analysis of average application speedup over different grid-sizes. **The following observations were made:**

1) It was noticed that in DRRMM, applications compromise to a distribution in fewer steps whereas, agents in distRM require more steps to attain a stable distribution. It can also been seen in Fig.2 that the agents in distRM negotiate and exchange cores more often as compared to DRRMM.
2) Even though it requires more number of steps and larger number of requests but it can be seen that eventually applications in distRM end up getting more cores and thus a higher value of speedup. Moreover, the final solution is closer to that of the centralized RM scheme.

The above-mentioned behavior can be explained by the fact that agents in distRM send request outside region R, which increases the potential options for getting more cores. Whereas, applications in DRRMM stays inside the region R. Therefore, the applications have no access to the available cores, which are outside this region R. Thus the latter attains a stable configuration in smaller number of steps while compromising on the possible better distribution of cores. These results indicate the usefulness of our approach over traditional analysis techniques as it not only saves the time and money spent on simulations but the model checker has exhaustively traversed through all possible system states and identified the worst possible speedup that the given distributed RM schemes may attain which is evident from the minimum-curve in Fig. 2 and also the speedups that would be attained on average by the underlying distributed RM schemes. As a simulator only provides results based on particular simulation paths, it is very much possible that we only check on the paths that end up giving a better solution. And there is no way that a simulator can exhaustively determine that how a distributed RM scheme would perform on average.

2) Average Core Utilization: An important piece of knowledge for distributed RM scheme designer can be the amount of average core utilization after a certain time interval when different applications are running in parallel. In order to perform this analysis, we augment the model with the following reward structure:

rewards “avg_core_utilization”
$[]true : total\_occupied\_cores/total\_cores;$
endrewards

The above-mentioned reward structure assigns a real value equal to the average core utilization to every state of the model. For instance, in order to calculate the average core utilization after 100 steps, we verify the following property:

$$R\,\{\text{“}avg\_core\_utilization\text{”}\} =?\ [I = 100]$$

where ‘I’ is known as the instantaneous reward. Unlike other reward structures, it does not give the accumulated value of reward, rather the instantaneous value at a particular time interval. Fig.3 shows the results of this analysis. It can be seen that in terms of average core utilization, distRM makes a more optimal utilization of available cores. These statistics can help the designers in optimizing a distributed RM scheme during early design stages where the designer can tweak the underlying distributed RM algorithm and analyze the resulting effects on average core utilization. Simulation-based results can again be inaccurate as we cannot exhaustively determine the average core utilization.

3) Overhead of Requests Initiated by Agents: Apart from making an optimal utilization of resources, a distributed RM scheme should also be efficient in terms of communication/computation overhead. We performed

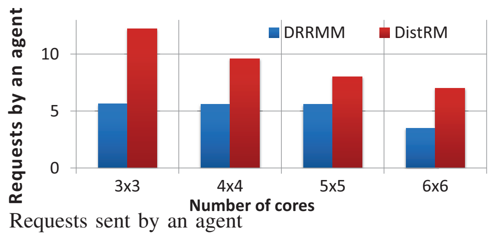

Fig. 5. Requests sent by an agent

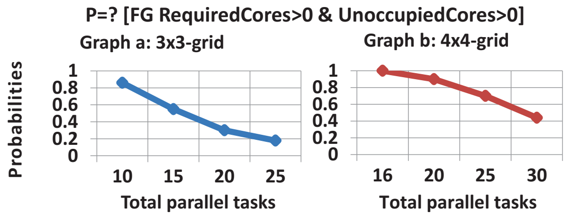

Fig. 6. Analysis of unoccupied cores

an analysis on the average number of requests sent by an agent and the comparison of the results of the two distributed RM schemes is shown is Fig.5. It can be seen that the agents in distRM send more requests compared to the DRRMM scheme.

4) Analysis of Unoccupied Cores: One of the main purposes of a distributed RM scheme is to make sure that all the cores will be used for actual computations and the *idle agents* will not manage the unoccupied cores forever. Therefore, we performed an analysis on resources vs demand. The results were evaluated for a $3x3$ and a $4x4$ grid size. We divided the grid into two regions and each region contains a manager for unoccupied cores. We kept the demand greater than the available resources for this experiment. For evaluating the probabilities, we calculated the *probability that in future one or more available cores will remain globally unoccupied even when there is a high resource demand*:

$$P =?[F\ G\ required\_cores > 0\ \&\ unoccupied\_cores > 0]$$

Fig.6 presents the results for the DRRMM scheme. It shows that the probability is very high for small number of parallel tasks and it gradually goes towards zero as there are more parallel applications on the chip. These results indicate that on some cores actual computations will never be performed and hence the system is not making an optimal use of the available resources. The probability of such an event is very important as the distributed RM scheme must ensure that all the cores will be utilized for actual computations and should not remain unoccupied for infinitely long interval. In distRM, this probability remains zero as long as the applications are allowed to periodically increase the distance for sending the requests. These results clearly indicate the usefulness of probabilistic model checking in the verification and analysis of distributed RM scheme as it not only allows the designer to identify the design problems but also compute actual probabilities associated with occurrence of faulty events. These cases have not been caught using the traditional analysis techniques.

The verification results show that there is a trade-off between the performance of the two RM schemes. One scheme attains better mapping configuration, which is closer to centralized while compromising on the communication/computation overhead. On the other hand, the other scheme is highly efficient in terms of complexity and it reaches stability in less time but it compromises on the possible number of better configuration options. Here, probabilities plays a very important role as the designer can determine how much a distributed RM scheme is compromising on one performance factor to attain another. These corner cases in which a core will remain unoccupied forever cannot be determined using simulations as simulation-based analysis only ascertains the correctness of the states that are reached within a particular simulation path and it is always possible that we may miss the path containing a fault. The probability of occurrence of such an event is highly useful for the designer to determine how efficiently the given distributed RM scheme is utilizing the available resources. These statistics corroborate the benefits of probabilistic model checking in verifying distributed RM.

VI. Conclusions

The paper presents a formal probabilistic methodology for analyzing and comparing distributed RM schemes. The proposed method utilizes the PRISM model checker to verify the functional and performance evaluation properties. For illustration purposes, a successful probabilistic analysis and comparison of two state-of-the-art distributed RM schemes is presented in this work. The rigorous nature of the analysis coupled with the ability to identify the outer bounds for the performance efficiency of the distributed RM schemes, the probability of occurrence of corner cases and detailed quantitative insights are the distinguishing features of our approach compared to the other traditional techniques.

One can optimize the design of distributed RM scheme by integrating our proposed methodology in early design phases. A designer can verify useful quantitative properties and compare the results of different alternate design solutions for the resource management problem and hence come up with a solution that utilizes the resources in an on-chip many-core system in the most efficient manner.

Acknowledgment

This work is supported in parts by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre Invasive Computing (SFB/TR 89 http://invasic.de) and in parts by the DAAD "Deutsch-Pakistanische Forschungskooperationen" project.

References

[1] ITRS, http://www.itrs.net, 2015.

[2] S. Kobbe, L. Bauer, D. Lohmann, W .S. Preikschat, and J. Henkel, "DistRM: Distributed Resource Management for On-Chip Many-Core Systems", Hardware/Software Codesign and System Synthesis, pages 119-128, 2011

[3] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on Multi/many-core Systems: Survey of Current and Emerging Trends", Design Automation Conference, pages 1-10, 2013.

[4] M. A. Al Faruque et al, "Adam: Run-time Agent-based Distributed Application Mapping for On-chip Communication." Design Automation Conference, pages 760-765, 2008.

[5] I. Anagnostopoulos et al, "Distributed Run-time Resource Management for Malleable Applications on Many-core Platforms", Design Automation Conference. (168), pages 1-6, 2013.

[6] M. Fattah, M. Daneshtalab, P. Liljeberg, and J. Plosila, "Smart Hill Climbing for Agile Dynamic Mapping in Many-core Systems", Design Automation Conference. (39), pages 1-6, 2013.

[7] W. K. Lam, "Hardware Design Verification: Simulation and Formal Method-Based Approaches", Prentice Hall, 2008.

[8] PRISM web site, www.prismmodelchecker.org, 2015.

[9] C. Baier and J. P. Katoen, "Principles of Model Checking", MIT Press, 2008.

[10] O. Hasan and S. Tahar, "Formal Verification Methods. In Encyclopedia of Information Science and Tech." pages 7162-7171, 2014.

[11] R. Alur and T. Henzinger, "Reactive Modules", Formal Methods in System Design, 15(1), pages 7-48, 1996.

[12] A. Biere, "Tutorial on Model checking, Modeling and Verification in computer science", Algebraic Biology, LNCS, (5147), pages 16-21, 2008.

[13] M. Kwiatkowska and D. Parker, "Advances in Probabilistic Model Checking", Software Safety and Security: Tools for Analysis and Verification. (33), pages 126-151, 2012.

[14] G. Weiss, Ed, "Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence", MIT Press, 1999.

[15] T. Desell et al, "Malleable Applications for Scalable High Performance Comp.", Cluster Computing. 10(3), pages 323-337, 2007.

[16] D. L. Dill, "What's Between Simulation and Formal Verification", Design Automation Conference, pages 328-329, 1999.

[17] A. B. Downey, "A Model for Speedup of Parallel Programs, Tech. Rep, 1997.