DistRM: Distributed Resource Management for On-Chip Many-Core Systems

Sebastian Kobbe[x], Lars Bauer[x], Daniel Lohmann[+], Wolfgang Schröder-Preikschat[+], Jörg Henkel[x]
[x]Karlsruhe Institute of Technology (KIT), Chair for Embedded Systems, Karlsruhe, Germany
[+]Friedrich-Alexander-Universität (FAU), Erlangen-Nuremberg, Germany
{sebastian.kobbe, lars.bauer, henkel} @ kit.edu
{lohmann, wosch} @ cs.fau.de

Abstract

The trend towards many-core systems comes with various issues, among them their highly dynamic and non-predictable workloads. Hence, new paradigms for managing resources of many-core systems are of paramount importance. The problem of resource management, e.g. mapping applications to processor cores, is NP-hard though, requiring heuristics especially when performed online. In this paper, we therefore present a novel resource-management scheme that supports so-called *malleable applications*. These applications can adopt their level of parallelism to the assigned resources. By design, our (decentralized) scheme is scalable and it copes with the computational complexity by focusing on local decision-making. Our simulations show that the quality of the mapping decisions of our approach is able to stay near the mapping quality of state-of-the-art (i.e. centralized) online schemes for malleable applications but at a reduced overall communication overhead (only about 12,75% on a 1024 core system with a total workload of 32 multi-threaded applications). In addition, our approach is scalable as opposed to a centralized scheme and therefore it is practically useful for employment in large many-core systems as our extensive studies and experiments show.

Categories and Subject Descriptors

D.4.7 [**Operating Systems**]: Organization and Design – *Distributed systems*

General Terms

Algorithms, Management

Keywords

Multicore, Manycore, MPSoC, Scalability, Resource management, Multi-Agent-System

CODES+ISSS'11, October 9–14, 2011, Taipei, Taiwan.

1 Introduction and Related Work

Current trends strongly point to on-chip many-core system designs. Prominent industrial examples are Intel's SCC [1] and Tilera's TILE-Gx processor family [2]. Both architectures have in common that they feature a large set of cores that are connected via a network on chip (NoC) [3]. Intel's SCC features 48 cores and Tilera currently features up to 100 cores per chip. Thousand core chips are a technology perspective that is expected to become reality within one decade [4]. Hence, the problem of scalability is real and of significant relevance. If no new paradigms are developed, complex future many-core systems will suffer from low efficiency since these systems will tend to spend large portions of their communication and computation capacities with managing their own resources instead of releasing the resources for efficient application execution.

1.1 Resource Management in Many-Core Systems

Hundreds or even thousands of cores might solve the demand for computational performance, but also lead to the problem of resource allocation, i.e. *which application* should use *which and how many* cores to get the most out of the system. The general problem of mapping applications to cores can be expressed as the *quadratic assignment problem*, which is known to be NP-hard. The size of the search space for an optimal mapping grows factorial with the number of cores [5].With high dynamic workloads, these mappings cannot be predetermined and have to be decided online.

A promising approach to utilize the available cores efficiently is the principle of *malleable applications* [6, 7] that are able to adapt their degree of parallelism to the number of assigned cores dynamically. This means that they are able to fulfill the same functionality on a variable number of cores and are designed in a way that allows it to enlarge or shrink the set of cores the application is using. The programmer of an application could either directly design the application to be malleable (e.g. using client/server architectures with varying amounts of clients per server to distribute the work among the clients and/or use work-stealing techniques [8]) or use frameworks that support the automated creation of malleable applications. E.g., Intel's Threading Building Blocks [9] allow the creation of malleable applications by applying techniques similar to work-stealing to efficiently make use of varying amounts of available cores.

1.2 Multi Agent Systems

As computing systems are continuously getting more complex, it becomes harder to manage them from one central point. Starting

with IBM's autonomic computing initiative [10], there has been a clear consent that future computing systems should be self-organizing and self-optimizing to be able to handle the always-growing complexity. This means that the components within a large system have to configure and optimize themselves independently to operate efficiently as a whole. Multi agent systems [11] are a promising approach to achieve these goals. A multi agent system is composed of multiple interacting intelligent agents and is typically used to solve problems that are difficult or impossible for an individual agent or a centralized decision authority to solve. In [12] the major characteristics of multi agent systems are defined as follows:

- each agent has only incomplete (i.e. local) information and is restricted in its capabilities
- system control is distributed
- computation is asynchronous, i.e. the agents operate independent of each other (except rather seldom communications)

1.3 Related work

Related work can be categorized into resource management schemes that support malleable applications and schemes that only support fixed-size applications. Both kinds of schemes further split into centralized and distributed approaches.

If the workload of the system is known at design time and does not change at runtime, the mapping can be decided offline for malleable as well as for fixed-size applications. An optimal or near-to-optimal solution can be found by applying exhaustive and stochastic search methods and heuristic approaches [5]. To speed up the offline search, distributed approaches have been presented [13]. However, all offline approaches require a priori knowledge about system states and thus cannot react to unforeseen situations.

Unknown workload, system states, or unpredictable issues like failures require online approaches to react during runtime. Because of the large search space, online approaches are typically not able to deliver optimal mappings. Approaches like presented in [14] apply heuristics to perform mapping of applications. They manage temporarily unused resources centrally and always choose the best fitting region on the chip to map an application. The approach does not allow resizing or remapping of applications. In [15] a centralized, iterative, greedy selection scheme for malleable applications is presented that achieves better average turnaround times than other recently proposed malleable application-scheduling schemes. The turnaround time is defined as the total time starting from a program being ready for execution until its completion. It is one of the commonly used metrics to evaluate scheduling algorithms. Due to the advantageous average turnaround times from the approach in [15], we have chosen it as our reference implementation to evaluate the decision quality of our proposed DistRM scheme. A brief description of the algorithm from [15] is presented in Section 3.1.

In [16] a distributed virtual cluster based online approach is presented where the mapping complexity is reduced by splitting the large search space into smaller virtual clusters. The scheme only supports static applications and focuses on communication. Grid computing uses and manages resources of many different, spatially distributed computing systems that are linked together using a middleware to solve the mapping of large tasks jointly [17]. Because of the size of the tasks, the spatial distribution and the (typically) unpredictable network in the grid-computing domain, their resource management tends to have a higher latency and a lower frequency of utilization changes. The rather seldom task changes in a grid computing environment allow complex procedures for the resource management. Distributed approaches for this management exist [18, 19], but due to their high complexity, they are not promising for embedded on-chip many-core systems.

Concluding the related work, there are distributed online approaches to tackle the enormous computational effort that arises in systems with many cores. In addition, there are online approaches that can utilize many available cores by supporting malleable applications, but they rely on centralized schemes with their beforementioned scalability issues. Altogether, no distributed online scheme exists that is suitable for on chip many-core systems supporting malleable applications.

1.4 Our DistRM Approach

In this paper, we present our distributed online resource management scheme for on-chip many-core systems. We designed it from scratch to be a distributed system without any global synchronization or global communication. This allows our management system to scale with the size of the many-core system. Our simulations show that it works as well in 64 (8x8) core systems as in 1024 (32x32) or even 4096 (64x64) core systems. To achieve this high scalability, we employ the principles of a multi agent system [12] to perform the resource management. We use one dedicated agent per application as *resource manager*. The necessary computations of each agent are performed on the same cores as the application it manages and thus the overall computational effort for the whole resource management is distributed throughout the many-core system.

Each agent autonomously aims at increasing the speedup of its application by searching for cores on the chip that can be used by its application. Therefore, it utilizes the ability of malleable applications to adapt – to some degree – to additional cores.

As soon as there is more than one application running on the system, the available cores have to be shared among the different applications. That leads to the situation that an application might lose a core to another application, for example, if an application occupies $n+1$ cores that only result in a small speedup improvement for this application compared to using n cores and another application would benefit significantly from one additional core.

The resources are no longer managed in one central place but in many places spread all over the chip. This provides important advantages like inherent parallelism and the avoidance of computational bottlenecks. Additionally, the necessary communications for resource management occur mainly in local areas. These areas are distributed over the whole chip instead of concentrating in one point. All these advantages help to make the multi agent system scalable and less intrusive to the actual applications running on the system.

The paper is structured as followed: In the next section, we will describe our system in detail. In Section 3, we give a detailed presentation of our evaluation setup before we present the results. Finally, we conclude this paper in Section 4.

2 Agent based Resource Management

In this section, we describe the architecture and algorithms of our agent based resource management scheme for on chip many-core systems in detail. We require an *application model* (described in Section 2.1) which is used by the application's agent to evaluate the application's speedup for a given set of cores.

In Section 2.2 we describe how new applications are initiated, how their agents allocate cores, how the agents communicate

(Section 2.3), and how the trading of cores takes place (Section 2.4 and 2.5). Finally, we briefly describe how the agents continuously try to optimize their applications set of cores at the applications runtime in Section 2.6 and how the management of unallocated cores is done (Section 2.7).

2.1 Application Model

Our application model consists of a workload W, which represents the work that the application has to perform during its lifetime, independent of how many cores are used for it. We assume our applications to be malleable, i.e. they can adapt to different amounts of cores at runtime. The high amount of available cores on future chips allows us to dedicate whole cores to applications (i.e. to make binary mapping decisions). This significantly simplifies the decision making and application performance estimation. The speedup of an application running on n cores is defined by the runtime on one core divided by the runtime on n cores. The finishing time T_{finish} of an application is calculated by applying the speedup function $S(n)$ on the remaining work $W_{remaining}$ whenever n changes. Applications without knowledge of their remaining workload (i.e. interactive applications) are assumed to run forever after their initialization. Currently our simulations only consider applications that provide a workload W.

$$S(n) = \frac{T(1)}{T(n)};\ T_{finish} = \frac{W_{remaining}}{S(n)} \tag{1}$$

In most applications, there is no linear correlation between the speedup and the number of cores. We use the speedup model for parallel programs introduced by Downey [20]. The authors have shown, that their model captures the behavior of real programs (e.g. benchmark suites) running on real parallel machines very well. The model is based on two parameters, the average parallelism A of a program and its variance in parallelism σ. Eq. 2, taken from [20], shows how the speedup $S(n)$ of an application is calculated if these parameters are known. Depending on the variance in parallelism σ a different calculation has to be performed. In the case of no variance at all, the number of cores n or the average parallelism A determines the speedup, depending on which of both is lower. The values A and σ for real applications can be obtained by evaluating benchmarks of the intended application running on the target platform [20]. The model is intended to and used to provide synthetically generated workload to evaluate many core scheduling schemes.

We use this speedup model to compare our scheme to a state-of-the-art malleable application-scheduling scheme [15], as it has been used by them and others (e.g. [21-23]) as well.

$\sigma < 1$ (low variance):

$$S(n) = \begin{cases} \dfrac{nA}{A + \frac{\sigma}{2(n-1)}}, & 1 \le n \le A \\ \dfrac{nA}{\sigma\left(A - \frac{1}{2}\right) + n\left(1 - \frac{\sigma}{2}\right)}, & A \le n \le 2 * A - 1 \\ A, & n \ge 2A - 1 \end{cases} \tag{2}$$

$\sigma \ge 1$ (high variance):

$$S(n) = \begin{cases} \dfrac{nA(\sigma+1)}{\sigma(n + A - 1) + A}, & 1 \le n \le A + A\sigma - \sigma \\ A, & n \ge A + A\sigma - \sigma \end{cases}$$

To allow multi-objective optimization, we use an enhanced version of the speedup model, which also considers the NoC communication, as bandwidth bottlenecks can reduce the practical speedup of an application. To consider this *network penalty* for calculating the communication-aware speedup $S'(n,C)$ for a subset C of cores (with $|C|=n$) we use the formula shown in Eq. 3. The formula is only an approximation of the real expected speedup and it is based on the idea that long distance communication should be avoided. Therefore, the available NoC bandwidth $BW_{maxAvailable}$ between two neighboring cores is divided by the average required bandwidth $BW_{avgRequired}(n)$ between two tasks (that shall execute on the n cores) and the average distance $Dist_{avg}(C)$ between two cores of the subset. Note that the average required bandwidth depends on the parallelization for n cores (i.e. a property of the application, independent of the actual cores) whereas the average distance depends on the particular subset C (i.e. *which* cores are evaluated). The average required bandwidth has to be annotated by the application designer. The values can be estimated by analyzing the used algorithms or measured by using on- or offline profiling methods. As the performed calculation of $S'(n,C)$ is only an approximation, it is not necessary to annotate the exact values if not possible.

$$S'(n,C) = S(n) * \min\left\{1,\ \frac{BW_{maxAvailable}}{BW_{avgRequired}(n) * Dist_{avg}(C)}\right\} \tag{3}$$

Our DistRM mapping scheme aims at minimizing the network penalty by reducing the average distance between two cores. If an application requires a large degree of communication bandwidth, its agent will focus on choosing neighbored cores, which typically limits the number of suitable cores. Without such communication requirements, the agent could acquire as many cores as available (including those with a larger average distance), as then the larger application speedup $S(n)$ (due to the larger number of cores) also results in a larger communication-aware speedup $S'(n,C)$.

2.2 Initialization of new Applications

When a new application is about to start running on the system, its agent is initiated on a random core. We apply the randomness to achieve a kind of load balancing without the need for global system state knowledge. However, the agent will eventually move to the actual cores it found for its application. The randomly chosen initial core acts as a seed for searching resources and it will not necessarily be chosen by the agent for its application. In fact, it could be any core. The agent then searches for a suitable set of cores within the system for its application.

Our algorithm for determining this set of cores is shown in the upper half of Listing 1 (lines 1-13). Searching the initial set of cores for the application is performed in a loop until at least one core is found. As the agent is not aware of the global system state, it randomly selects *regions* on the chip and tries allocating resources there. A region of size r is a set of cores that is defined as a particular core (the center of the region) and all cores that are within a Manhattan distance[1] of r to that core. Up to `MAX_PAR_REQS` requests are performed in parallel to speed up the search. Therefore, the agent first randomly selects multiple potential regions on the chip and then performs a pre-selection of re-

[1] The Manhattan distance equals the sum of the horizontal and vertical hops required to get from one point to another in a mesh network.

gions before initiating the actual requests on the remaining regions. To reduce the average communication distance, regions near to the randomly chosen seed are preferred in the pre-selection process. The longer the agent (unsuccessfully) tries to allocate resources for its application, the higher the probability becomes to use regions that are more distant from the initial seed core. Therefore, nearby regions are removed from the list of potential regions with an always-increasing probability, as the agent most probably examined them before. If after this randomized elimination of nearby regions more than `MAX_PAR_REQS` regions remain within the list of randomly chosen regions, also the most distant ones are removed. The agent will then send a resource request to the remaining regions as described in the next section and shown in the lower half of Listing 1 (lines 14-29).

```
1.  tries ← 0;
2.  Offers ← ∅;
3.  while Offers = ∅ and tries < TRY_LIMIT do
4.      tries ← tries + 1;
5.      PotentialRegions ← selectRandomRegions;
        // now pick the most promising regions. The longer
           we search, the more distant they could be
6.      foreach Region ∈ PotentialRegions do
7.          if distance(Region) < (tries * MAX_DIST / TRY_LIMIT)
                and rand(0,1) > (1/tries) then
8.              PotentialRegions = PotentialRegions – Region;
9.          end if
10.     end foreach
11.     while PotentialRegions.size > MAX_PAR_REQS do
12.         deleteMostDistantRegion(PotentialRegions);
13.     end while
        // after we have selected the regions to ask for
           resources, try to actually request resources
14.     foreach Region ∈ PotentialRegions do parallel
15.         if distance(Region) > LOCALITY_THRESH then
16.             Offers ← Offers ∪ Region.dispatchRequest;
17.         else
18.             if size(Region) > SIZE_THRESH then
19.                 SubRegions = split(Region);
20.                 foreach Region ∈ SubRegions do parallel
21.                     Offers ← Offers ∪
                            Region.dispatchRequest;
22.                 end foreach
23.             else
24.                 Offers ← Offers ∪ self.handleRequest;
25.             end if
26.         end if
27.     end foreach
28.     wait(REQUEST_TIMEOUT);
29. end while
```

Listing 1 Simplified pseudo code showing how an agent tries to get offers for cores after its initialization

2.3 Agent Communication Scheme

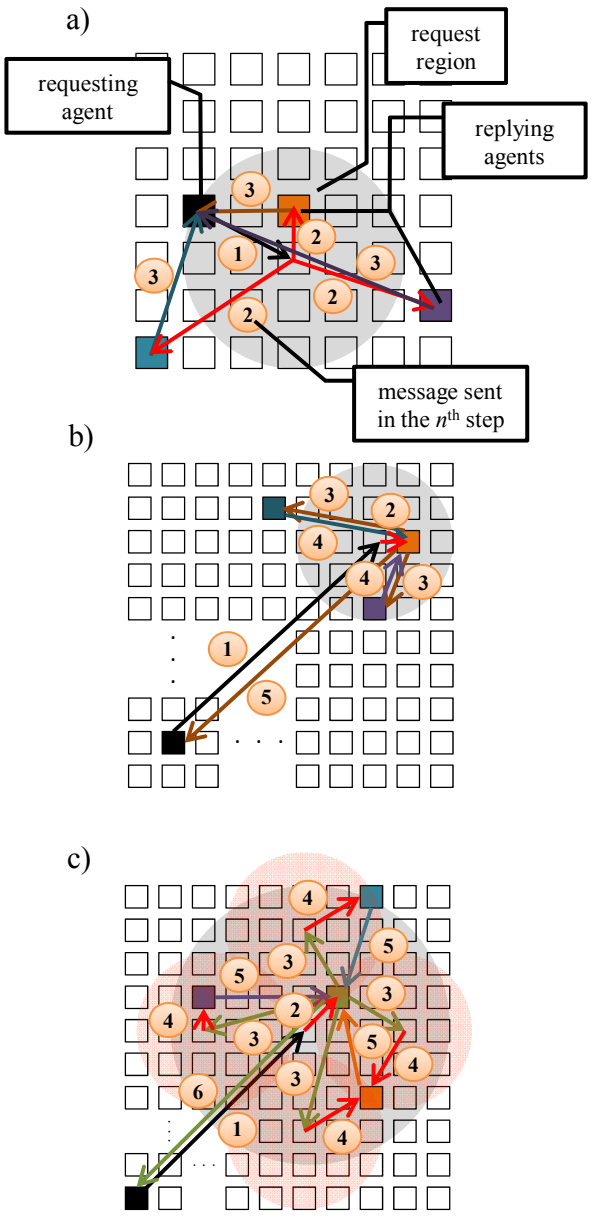

Figure 1 Communication between agents for a) local and b) distant target regions; c) shows the divide-and-conquer behavior for larger target regions

We have designed our multi agent system to work in a fully decentralized manner to allow it operating on many-core systems of any size. In addition, we avoid using broadcast communication, which could lead to an over-utilization of the communication system. The agents need to know how to communicate with other agents. A centralized directory of all current locations of active agents could solve that problem, but would contradict the design principles of our DistRM scheme and potentially reduce the scalability of the approach. Instead, an important part of the infrastructure of our multi agent system is a distributed directory service, which allows the agents to answer the question, which agents have allocated cores within a local region. All available cores are split into evenly distributed clusters, while each cluster contains one di-

rectory running on one of the cores. The agents register themselves at the directories corresponding to the cores occupied by the own application. These directories provide a regional knowledge to the system on a particular core about the agents that are located nearby at a certain point in time.

We have implemented some mechanisms to reduce long-distance communication as much as possible. Whenever an agent requests resources for its application in a distant region of the chip, it sends a request to a core in the middle of that region. The system running on that core utilizes the knowledge of the distributed directory service about the local agents and forwards the request to the agent with the most cores in the desired region. This agent then becomes the local request manager and distributes the request to the other agents in that region, collects the replies and sends them back to the initial agent. When an agent wants to request resources from its own proximity, the step of forwarding the request to another agent is skipped and the agent acts directly as the request manager. We apply a divide-and-conquer scheme to handle larger requests. The request manager splits the request into four sub-requests and handles these like new requests in parallel. Each agent keeps track of the last requests it already handled to make sure that no loops are created.

Figure 1 visualizes how this scheme helps avoiding long-distance NoC communication. Figure 1a) shows how a request in the proximity of an agent is performed. The agent handles the request by itself. Therefore, it sends the request directly to the agents that occupy cores in the desired region using the knowledge of the distributed directory. The answers to that request are directly sent back to the requesting agent. Figure 1b) shows how the request for a more distant region is handled. One of the agents within the region becomes the local request handler and collects the replies of the other agents in step (4). It accumulates the replies of the other agents in that region (including his own reply) and sends them back to the agent that was interested in that region (5). Figure 1c) finally shows how the request handler splits the request into smaller requests, which are then handled in parallel. Again, the results to these new requests are accumulated by the request handler before they are sent back to the initial agent.

2.4 Interplay of Resource Management Agents and Applications

After we have described the most important parts of the agent system, we now present the flow through the components of the agent system right after the initialization of a new application. Figure 2 shows how the components of the systems interact with each other.

The first step (1) represents the instantiation of the agent of the new application on a random core. The agent then randomly chooses regions on the chip (2) and tries to allocate cores there. The agent first tries regions near to its own position, but with each try, the probability to use more distant regions increases. In each try, a limited number of possible regions on the chip are then handled in parallel. Depending on the distance, the agent itself starts to bargain for these resources or it dispatches the request to one of the agents in the given region (3), (4). The agent that actually handles the request then looks up the actual agents currently located in that region (5) and then asks these agents (6) to evaluate which of their own cores could be given to the requesting agent (7). They send their offer (containing information about their own loss in speedup by providing the cores) back to the managing agent (8). All offers for the region are collected by this agent (9) before they are sent back to the requesting agent (10). Note that this step is not required if the requesting agent handles the request by itself. Steps (3) to (10) can be repeatedly applied to cover larger regions on the chip. At the end, the requesting agent evaluates all offers (11). All offers that help increasing the speedup of the own application are taken, as long as the speedup of the own application increases more than the speedup for the offering agents application decreases. Not all offered cores have to be chosen, e.g. if the agent has received another offer before and its own applications situation has changed. The agent will inform all agents that made an offer, whether or not their offer had been taken and which cores are affected. (12). The agents that have to release cores then have to tell their own applications to reconfigure and resize (13) while in parallel the requesting application (14) is informed and reconfigured for the new set of cores.

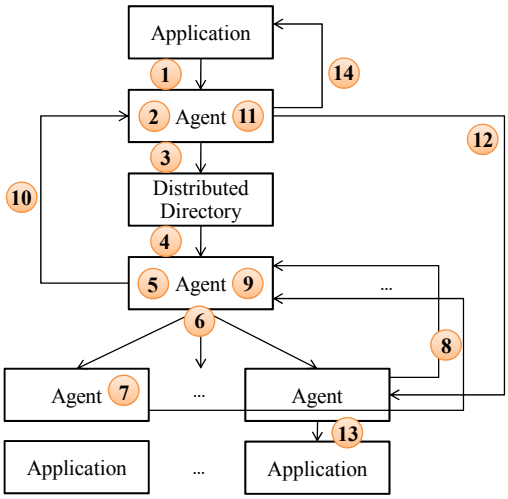

Figure 2 Interaction of the components of the system

2.5 Request Handling Algorithm

We deploy a greedy heuristic to decide which and how many of the cores of an application are offered if another agent sends a request (see Listing 2). The key idea behind the algorithm is to transfer cores within a given region from one application to another that maximize the total gain in speedup expressed through Eq. 4.

$$gain_{total} = gain_{receiver} - loss_{giver} \quad (4)$$

The greedy selection of cores continues until $gain_{total}$ drops below zero, i.e. when $loss_{giver}$ (see Listing 2, line 10) is larger than $gain_{receiver}$ (see Listing 2, line 9). All offered cores are reserved for the requesting agent and thus not offered to another requesting agent. The reservation is removed after the agent knows the decisions, which from the offered cores have to be transferred to the requesting agent.

To allow requesting resources from multiple agents within one region in parallel, the amount of cores each agent manages within the desired region is considered. The *share* of an agent that may potentially give away cores within a region ($share_{giver}$) is defined as the number of cores in that region that are used to execute the agent's application in comparison to the total number of cores in that region (see Eq. 5).

```
1.  OfferedCores ← ∅;
2.  gain_total ← ∞;
3.  while gain_total > 0 do
4.      gain_total ← 0;
5.      GreedyChoice ← ∅;
6.      base_receiver ← Speedup(Cores_receiver ∪ OfferedCores);
7.      base_giver ← Speedup(Cores_giver − OfferedCores);
8.      foreach Core ∈ (Cores_giver∩Region) –OfferedCores
        do
9.          gain_receiver ← share_giver *
                Speedup(Cores_receiver ∪ OfferedCores ∪ Core)
                − base_receiver;
10.         loss_giver← base_giver –
                Speedup (Cores_giver – OfferedCores – Core);
11.         if (gain_receiver – loss_giver > gain_total) then
12.             gain_total ← gain_receiver – loss_giver;
13.             GreedyChoice ← Core;
14.         end if
15.     end foreach
16.     if gain_total > 0 then
17.         OfferedCores ← OfferedCores ∪ GreedyChoice;
18.     end if
19. end while
```

Listing 2 Pseudo code of the greedy selection algorithm used by the agents

$$share_{giver} = \left|Cores_{giver} \cap Region\right| / \left|Region\right| \qquad (5)$$

Each agent within a region performs the selection with a slightly difference in the gain function (i.e. with a different $gain_{total}$, see Eq. 6).

$$gain_{total} = share_{giver} * gain_{receiver} - loss_{giver} \qquad (6)$$

Without this modification, multiple agents would offer more cores than actual beneficial for an overall balanced speedup as they do not know about the offers from the other agents. With this modification, it becomes harder to obtain rather many cores from only one agent. Instead, the offer typically contains cores from different agents of different applications, which results in a rather small loss for all of them and thus a more balanced distribution of cores among applications.

2.6 Continuous Self-Optimization

Requesting resources from other agents not only happens during the initialization phase of an application. After an agent has found the initial set of cores for its application, it does not stop trying to increase the speedup of its application by acquiring additional cores. Therefore, at runtime of its application each agent sends requests for cores to nearby regions. These requests are then handled like the initial request. Again, the regions are chosen randomly, but this time no distant regions are considered. To avoid back-and-forth trading between two applications (what would lead to a serious performance impact due to permanent reconfiguration effort of the applications), the transfer of cores only happens if the gain in speedup is significant higher than the loss for the giving application. This scheme significantly helps to optimize the mappings of applications over time, e.g. if new resources are available because another application that was running in a given region finished its task. Figure 3 shows how three applications running for some time optimize their mapping iteratively through local reconfigurations.

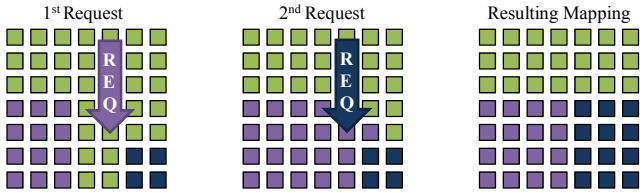

Figure 3 After two optimization requests from different applications the initial mapping changed to a more balanced share of resources

To make sure that requesting additional resources does not happen too often, the maximum delay between two requests is doubled after each request. A random delay between a fixed minimum and the always increasing maximum is chosen to avoid a synchronization of the optimization cycle of different agents, i.e. 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. The idea behind this is to optimize the set of cores of an application as soon as possible and keep the cores as long as possible to benefit the most of the additional speedup. This approach keeps the self-optimization overhead of the agent system low.

2.7 Management of Unallocated Cores

We employ a specialized kind of agents (called *idle agents*) to manage the cores that are not allocated by any application at any point of time. This kind of agent behaves like a regular applications agent, but it will never get any benefit from the cores it is managing, i.e. $loss_{giver}$ from Eq. 4 is always zero. With other words: Whenever an idle agent is managing cores and another (regular) agent requests cores from it, the request will be granted. In difference to the applications agents that might move with their application, the idle agents for managing the unallocated cores are located at fixed positions, forming a regular grid distributed all over the system. Each of these idle agents is responsible for a fixed set of cores within a rectangular region in the grid. For any system size, the size of this region defines how many idle agents have to be initiated.

There are two cases, in which cores are not used by an application and thus have to be handed by an idle agent. The first (trivial) case is directly after the initialization of the system. The second case is whenever an application has finished its calculations and does no longer require the allocated cores. The applications agent hands the cores over to the responsible idle agents before its own termination. The previously (Section 2.6) introduced self-optimization scheme of each application makes sure, that the idle agent does not manage these cores forever, but that they are used for actual computation.

3 Evaluation and Results

To be able to evaluate how the multi agent system performs, we have created a complete system-level simulation environment that

is capable of simulating arbitrary configurations of on-chip many-core systems. By running the simulation, we are able to tell which core had to perform how many calculations and how many messages of which size the NoC had to transfer. The multi agent system and the centralized resource manager (presented in section 3.1) are implemented in our simulation environment. To evaluate our system we use synthetically generated workload that is generated using the widely used Downey model [20, 24, 21-23]. The workload model comes with a workload generator, which generates a set of jobs with distributions of starting time T_0, total Workload W and parallelism parameters (A and σ matching to the speedup model shown in Eq. 2) similar to those of real workloads on parallel high performance computers. The workload generator can be configured to match the size of the system (workload generated for a 1024 core system is not suitable for evaluating a 25 core system). In all analyzed cases, the input data fed to the multi agent system and the centralized manager is the same to allow a fair comparison. Additionally, our simulations assume resizing of a malleable application and task migrations (required by our implementation of the centralized manager) without any overhead (which is not the case in real world systems [25]).

For evaluating our distributed resource management scheme, we run several system-level simulations. We use different system sizes ranging from 5x5 to 32x32 cores and synthetically generated workload consisting of 16, 32 and 64 parallel applications. We generate the workload matching to every system configuration ten times and then evaluate the centralized scheme and our distributed scheme also ten times, resulting in 100 runs per configuration. Parameters for `MAX_PAR_REQS` and the request region size have been found empirically. All following results are based on the same configuration (the size of the initial request is set to a Manhattan distance of 3, the size of the periodic requests for continuously self optimization is set to 2, and `MAX_PAR_REQS` is set to 3). The distance (grid size) between the *idle agents* (see Section 2.7) has been set to 5, i.e. each *idle agent* is responsible for 25 cores.

3.1 Reference Implementation: Centralized Resource Manager

```
1.  foreach application do
2.      application.cores ← 1;
3.      unmark(application);
4.  end foreach
5.  while unmarked application exists and cores available do
6.      greedily choose the application J that would benefit
        the most from an additional core;
7.      J.cores ← J.cores + 1;
8.      calculate finishing time of all applications;
9.      if overall finishing time did not improve then
10.         J.cores ← J.cores – 1;
11.         mark(J);      // J is not a good candidate for
                             additional cores
12.         recalculate finishing time of all applications;
13.     end if
14. end while
```

Listing 3 Pseudo code showing the iterative greedy optimization algorithm used by the central resource manager [15]

To evaluate our distributed and self-organizing resource management scheme, we compare it with the centralized scheme presented in [15]. This scheme has been chosen because it produces competitive, near-to-optimal schedules for malleable applications for a broad range of input data without the need for fine-tuning. It is an iterative greedy selection scheme that achieves efficient resource utilization, as cores are not wasted on poorly scalable applications. The used base algorithm is shown in Listing 3. Details on the implementation and the evaluation of this algorithm are presented in [15]. The algorithm itself does not create mapping decisions, but it creates near-to-optimal decisions on how many cores each application should use to optimize the overall average turnaround time. A heuristic algorithm like the one presented in [26] (that is also used in [14] to translate the decisions made by the selection scheme into actual mapping decisions) needs to be applied to achieve an actual mapping of these cores on the chip.

3.2 Comparison

In the following subsections, we present the results of the comparison between the centralized and our DistRM scheme. We analyze the decision quality, the computational overhead, and the communicational overhead of both schemes.

3.2.1 Decision Quality

The metric we used to compare both schemes is the total workload of all applications divided by the sum of the turnaround times of all applications, resulting in the average speedup of all applications in each simulation run as shown in Eq. 7.

$$Speedup_{avg} = \frac{\sum Application\ Workload}{\sum Application\ Turnaround\ Time} \quad (7)$$

Our results show that our distributed approach performs better, the more cores are in the system. In average, we achieve about 84% of the mapping quality of the centralized scheme, as shown in Figure 4.

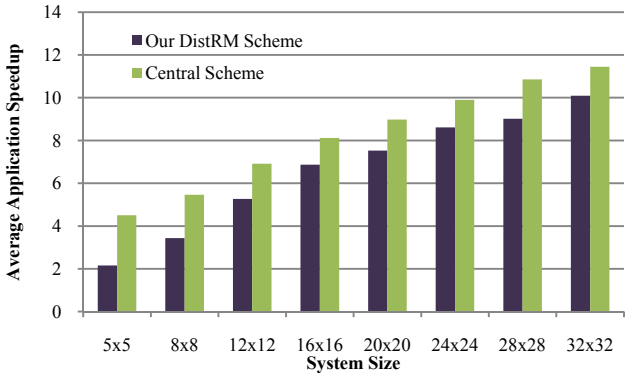

Figure 4 Average application speedup for various system sizes, comparing the centralized scheme with our proposed DistRM scheme

However, remember that we assume free task migrations, which is not realistic for real world systems. Our implementation of the centralized scheme always aims at the globally best solution independent of the current task mapping (resulting in big changes in the mapping whenever a new application enters the system or an application finishes its workload), whereas our DistRM scheme performs local changes such that no complete application has to

be migrated to another set of cores. In fact, task migrations are very expensive in terms of network bandwidth and migration latency [25], thus our comparison with the centralized scheme is conservative. This also means that a more sophisticated mapping scheme would be required for the centralized resource manager to reduce the amount of task migrations. This would come with a much higher (potentially infeasible) computational effort compared to the one we used. In both cases (i.e. frequent task migration or more sophisticated mapping scheme), the real speedup of the centralized scheme would be reduced compared to our conservative comparison, which shows that the slightly decreased speedup of our DistRM scheme (on large systems) is acceptable when considering its significantly reduced overhead (evaluated in Subsections 3.2.2 and 3.2.3).

3.2.2 Computational Overhead

The computational complexity of the centralized resource manager per decision is within $\mathcal{O}(n*m^2+m*log(m))$ where m is the number of applications to be mapped and n is the number of cores in the system. The inner loop (Listing 3, lines 5-14) requires $\mathcal{O}(n)$ iterations. Within each iteration, there is one greedy selection (line 6) in $\mathcal{O}(m)$ and one simplified many core schedule needs to be calculated (line 8) in again at least $\mathcal{O}(m)$. The selection scheme determines which application should use how many cores. It does not solve the mapping problem – thus an additional mapping heuristic with additional computational overhead of $\mathcal{O}(m*log(m))$ is required [14, 26]. The central manager has to perform these calculations whenever an application starts or finishes. The value of m changes over time and depends on the workload of the system. The authors of the algorithm [15] did not state which scheduling algorithm they used within their algorithm, how many applications they processed, or which machine they used to run the algorithm.

Our multi agent system distributes its computations all over the chip. Each agent performs up to `MAX_PAR_REQS` requests in parallel when it searches the initial set of cores for its application (see Section 2.2). Afterwards, requests are sent randomly distributed over the lifetime of its application to optimize its set of cores (see Section 2.6). The more applications are running in parallel, the more agents exist. The more often the self-optimization occurs, the more requests are generated. The larger the request region size gets, the more calculations have to be performed per request. At most $\mathcal{O}(RegionSize)$ calculations have to be performed by an agent to answer a request. This results in a total estimated complexity of $\mathcal{O}(m*(\texttt{MAX\_PAR\_REQS}*RegionSize+SelfOptimizations*RegionSize))$. However, the computational complexity of the agent system is independent of the size of the many-core system, which makes it scalable for future systems.

Figure 5 shows the simulated accumulated computational effort of the centralized approach and the multi agent system for managing a synthetically generated workload consisting of 16, 32, or 64 applications. Again, both schemes use the same workload data and the same speedup function. The values show how often the speedup function is evaluated in average during our simulations. Calling the speedup function is the innermost loop of both approaches and thus acts as the chosen indicator of computational effort. Note, that the centralized scheme has to perform additional computations in the order of $\mathcal{O}(m*log(m))$ to calculate the mapping that are not shown in the graph. While the effort grows with the size of the chip for the centralized scheme, it stays constantly low for our multi agent system. While all those computations have to be performed on a single core in the centralized scheme, the effort of our multi agent system is distributed over the whole system, as each agent has to perform only some of the calculations.

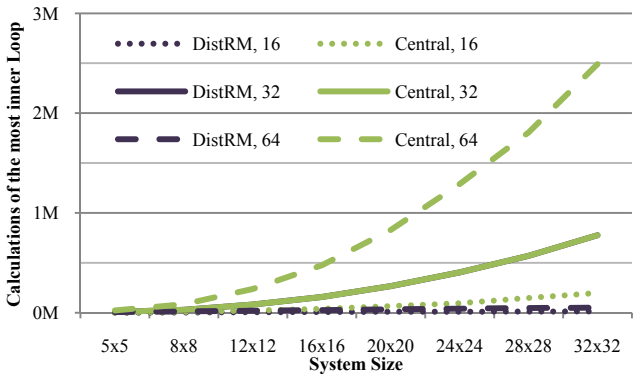

Figure 5 Accumulated computational effort (number of calls to the speedup calculation function) of both schemes for a workload consisting of 16, 32, and 64 applications on various system sizes

For example, on a 1024 (32x32) core system with a workload consisting of 32 applications, the centralized scheme requires in average about 777,000 calculations of the speedup function, whereas our distributed scheme only requires about 27,000 calculations or – in average – only about 850 (less than 1%) calculations per agent.

3.2.3 Communication Overhead

In our analysis of the communication overhead, we are only looking at the communications directly related to the resource management scheme. We assume a 2D mesh network in our simulations. In a real world deployment, additional network utilization would be caused by the applications reconfigurations. Again, our comparison is conservative because the centralized scheme causes more of these reconfigurations (whenever an application starts or finishes, all other currently running applications have to be resized and potentially remapped) compared to our multi agent system which only causes local reconfigurations.

The centralized scheme requires one short message that is sent to the resource manager whenever an application requests resources or when it finishes execution. In both cases, all running applications in the system need to be informed about the cores they are allowed to use after this change in utilization. Our multi agent system utilizes more messages to achieve its behavior as described in Section 2 and in particular in Figure 2. The results of our simulations are presented in Figure 6, showing the total number of messages and the average communication distance required for both schemes for handling the workload for various system sizes. As the larger systems' workload contains more parallel executions of applications, the number of required messages grows with the system size. Because of the random influences in our multi agent system, no direct correlation between the system size and the number of required messages is notable. To determine the average communication distance, the centralized resource manager is placed in the middle of the system. We use the Manhattan distance (as typical in a 2D mesh network) to calculate the distance between two cores. The average communication distance directly correlates to the system size in both schemes. It grows twice

as fast with the system diameter in the centralized scheme than in our DistRM scheme.

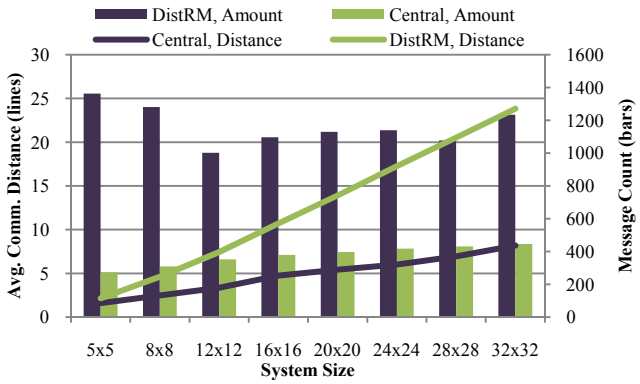

Figure 6 Total number of required messages (bars) for handling a workload of 32 applications and the resulting average communication distance (lines) on a 2D mesh network and various system sizes

The average message size in the centralized scheme directly correlates to the system size because the average amount of cores per application grows with the size of the system. The messages of our multi agent system are shorter in average, as only information about a few cores is transmitted. Still some messages containing the current set of cores of an application are required. Again, the size of these messages is correlated to the size of the set of cores, which typically grows with the size of the many-core system. As such, in total the average message size grows with the system size in our multi agent system. The amount of bytes each represented core within these messages actually requires depends on how the information about each core is expressed. We use two 8 bit values to express the X and Y coordinates of each core, resulting in 16 bits per core per message. The way cores are represented within a message could be optimized, but these optimizations would improve both schemes equally. Because of this, the values presented below are relative to each other and not absolute.

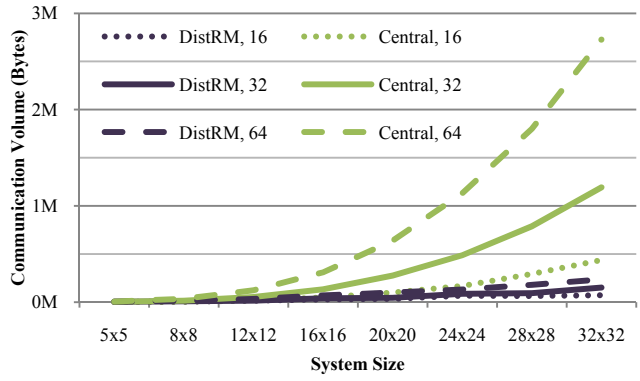

Figure 7 The network utilization of our DistRM multi-agent system compared to a centralized scheme on a 2D mesh network for 16, 32, and 64 applications for various system sizes

The total network utilization is calculated by multiplying the amount of messages, the average message size, and the average communication distance. The resulting communication volume required for handling the workload of 16, 32, and 64 applications is shown in Figure 7. As soon as the system size reaches 16x16 cores, our DistRM scheme requires less total communication volume. Additionally, the messages required by our scheme are scattered all over the NoC, which helps to avoid communicational bottlenecks which might occur with a centralized scheme.

Looking at the values of a 1024 (32x32) core system, we can see, that our DistRM scheme only requires 12,75% of the communication volume of the centralized scheme for managing 32 applications, 30,35% on a 256 (16x16) core system and about the same (96,39%) as the centralized scheme for a 25 (5x5) core system. More concurrent applications or larger systems make our scheme even more beneficial.

3.3 Implementation of our DistRM Scheme

To evaluate the effort of a real world deployment of our scheme, we have implemented it as a plain C program in addition to the implementation in our system level simulation environment. Therefore, we assume that there is already an operating system running on each core that supports basic functions like sending and receiving messages through the NoC. The agent code is running on operating system level, realized as a software implementation of a finite state machine triggered by software interrupts. The agent is triggered by a) application level function calls, b) incoming messages from other agents, and c) a timeout mechanism provided by the operating system. The realization of the agents as a software interrupt allows implementing the operating system in a lightweight manner, i.e. without the need for a sophisticated thread management per core. The timeout trigger is required to allow the application running on a core to continue its work during the time where the agent waits for replies from other agents and it allows optimizing the set of cores available to the application continuously. No additional hardware is required to support our DistRM scheme.

The software implementation consisting of a finite state machine with 8 states requiring less than 500 lines of C code. The code was configured for an agent capable of residing in a 32x32 core system. Compiled for the i686 architecture, in total about 6 KBytes of static memory are required, split into 2 KBytes of program code and 2 KBytes of global state keeping variables. Additionally, a 2 KByte buffer for NoC messages is used. The required dynamic memory depends on the state of the agent, the size of the system and the number of incoming requests.

Overall, the footprint of our multi agent system is quite small, making it a perfect match for large on-chip many-core systems where the number of cores is high but the local memory of each core is limited.

3.4 Interpretation of the Results

Although our simulation method favors the centralized scheme, we have shown that our DistRM scheme is able to manage the resources of a many-core system in a way that leads to an average overall application speedup near to the one a centralized scheme is able to achieve. The more cores are available, the smaller the difference becomes. Our results show, that the accumulated computational and communicational demands of our scheme drop below the demands of a centralized scheme as soon as the many-core system is bigger than 12x12 cores. The larger the system becomes, the higher the benefits of using our DistRM scheme for managing the resources of an on-chip many-core system become. As the computational effort of our scheme is not correlated to the amount of cores within the system, it scales well for 16x16, 32x32, or even 64x64 systems. Despite that, a centralized scheme

eventually becomes a bottleneck for rather large systems. All decisions of our DistRM scheme are based on local knowledge and only have a local influence on the resource utilization of the entire system. Thus, our scheme opens the doors to exploit this fast, lightweight local exchange of cores between applications, which is not be possible if the global system state would have to be examined or even changed for every local change.

4 Conclusion

In this paper, we have shown that managing resources in an on-chip many-core system in a distributed manner is feasible and fruitful, even if the system has hundreds or thousands of cores. We have shown that the decision quality (in terms of average application speedup) is almost as good as it would be with a state-of-the art centralized heuristic with global knowledge when using a conservative comparison. We have shown that managing the resources of a large on-chip many-core system (a NP-Hard problem) in only one central place leads to serious issues in terms of computational complexity when the number of cores gets too high. Our agent-based distributed resource management scheme allows distributing this effort over many cores in parallel. As each agent only has to take care of a reduced subset of the large search space, the computational complexity for each of these parallel working agents is also significantly smaller (less than 1% of the calculations have to be performed on a 1024 core system with 32 running applications). Our approach requires more messages sent trough the NoC, but we were able to show that the messages are shorter and most often only require a few hops on the NoC and are distributed over the entire chip (in contrast to a centralized solution). Therefore, our approach requires notably less network bandwidth (12,75% for a 1024 core system with 32 running applications) compared to a centralized solution.

5 Acknowledgment

This work was partly supported by the German Research Foundation (DFG) as part of the Transregional Collaborative Research Centre "Invasive Computing" (SFB/TR 89).

References

[1] J. Howard, S. Dighe, Y. Hoskote *et al.*, “A 48-Core IA-32 message-passing processor with DVFS in 45nm CMOS”, in *IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC)*, February 2010, pp. 108–109.

[2] Tilera Corporation, “Tile-GX Processor Family”, 2011.

[3] L. Benini and G. De Micheli, “Networks on chips: a new SoC paradigm”, *IEEE Computer*, vol. 35, no. 1, pp. 70–78, January 2002.

[4] S. Borkar, “Thousand core chips: a technology perspective”, in *Proceedings of the 44th annual Design Automation Conference (DAC)*, 2007, pp. 746–749.

[5] C. Marcon, E. Moreno, N. Calazans, and F. Moraes, “Evaluation of algorithms for low energy mapping onto NoCs”, in *IEEE International Symposium on Circuits and Systems (ISCAS)*, May 2007, pp. 389–392.

[6] J. Turek, J. L. Wolf, and P. S. Yu, “Approximate algorithms scheduling parallelizable tasks”, in *Proceedings of the fourth annual ACM symposium on Parallel algorithms and architectures (SPAA)*, 1992, pp. 323–332.

[7] D. Feitelson, L. Rudolph, U. Schwiegelshohn *et al.*, “Theory and practice in parallel job scheduling”, in *Job Scheduling Strategies for Parallel Processing*, 1997, vol. 1291, pp. 1–34.

[8] R. Blumofe and C. Leiserson, “Scheduling multithreaded computations by work stealing”, *Foundations of Computer Science, Annual IEEE Symposium on*, vol. 0, pp. 356–368, 1994.

[9] J. Reinders, *Intel threading building blocks*, 1st ed. Sebastopol, CA, USA: O’Reilly & Associates, Inc., 2007.

[10] P. Horn, “Autonomic Computing: IBM’s Perspective on the State of Information Technology”, *Computing Systems*, pp. 1–40, 2001.

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

[12] N. R. Jennings, K. Sycara, and M. Wooldridge, “A Roadmap of Agent Research and Development”, *Autonomous Agents and Multi-Agent Systems*, vol. 1, pp. 7–38, 1998.

[13] J. Boillat and P. Kropf, “A fast distributed mapping algorithm”, in *CONPAR 90 - VAPP IV*, 1990, vol. 457, pp. 405–416.

[14] B. Yang, L. Guang, T. Xu *et al.*, “Multi-application multi-step mapping method for many-core network-on-chips”, in *NORCHIP*, November 2010, pp. 1–6.

[15] G. Sabin, M. Lang, and P. Sadayappan, “Moldable parallel job scheduling using job efficiency: An iterative approach”, in *Job Scheduling Strategies for Parallel Processing*, 2007, vol. 4376, pp. 94–114.

[16] M. A. Al Faruque, R. Krist, and J. Henkel, “ADAM: run-time agent-based distributed application mapping for on-chip communication”, in *Proceedings of the 45th annual Design Automation Conference (DAC)*, 2008, pp. 760–765.

[17] F. Berman, G. Fox, and T. Hey, *Grid Computing - Making the Global Infrastructure a Reality*. Chichester, UK: John Wiley & Sons, Ltd., 2003.

[18] J. Cao, S. A. Jarvis, S. Saini *et al.*, “Arms: An agent-based resource management system for grid computing”, *Sci. Program.*, vol. 10, pp. 135–148, April 2002.

[19] F. Berman, R. Wolski, H. Casanova *et al.*, “Adaptive computing on the Grid using AppLeS”, *IEEE Transactions on Parallel and Distributed Systems*, vol. 14, no. 4, pp. 369–382, April 2003.

[20] A. B. Downey, “A Model for Speedup of Parallel Programs”, Tech. Rep., 1997.

[21] D. Feitelson, “Metric and workload effects on computer systems evaluation”, *Computer*, vol. 36, no. 9, pp. 18 – 25, sept. 2003.

[22] B. Zhou, D. Walsh, and R. Brent, “Resource allocation schemes for gang scheduling”, in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin / Heidelberg, 2000, vol. 1911, pp. 74–86, 10.1007/3-540-39997-6_6. [Online]. Available: http://dx.doi.org/10.1007/3-540-39997-6_6

[23] W. Cirne and F. Berman, “Adaptive selection of partition size for supercomputer requests”, in *Job Scheduling Strategies for Parallel Processing*, ser. Lecture Notes in Computer Science, D. Feitelson and L. Rudolph, Eds. Springer Berlin / Heidelberg, 2000, vol. 1911, pp. 187–207, 10.1007/3-540-39997-6_12. [Online]. Available: http://dx.doi.org/10.1007/-3-540-39997-6_12

[24] D. Feitelson, “Parallel Workloads Archive”, 2005. [Online]. Available: http://www.cs.huji.ac.il/labs/parallel/workload/

[25] J. Jahn, M. Al Faruque, and J. Henkel, “CARAT: Context-aware runtime adaptive task migration for multi core architectures”, in *IEEE/ACM 14th Design Automation and Test in Europe Conference (DATE)*, March 2011, pp. 515–520.

[26] K. Bazargan, R. Kastner, and M. Sarrafzadeh, “Fast template placement for reconfigurable computing systems”, *IEEE Design Test of Computers*, vol. 17, no. 1, pp. 68–83, 2000.