# Adaptive on-the-fly Application Performance Modeling for Many Cores

Sebastian Kobbe, Lars Bauer, Jörg Henkel
Chair for Embedded Systems (CES), Karlsruhe Institute of Technology (KIT), Germany

*Abstract* – **Resource management for a many-core system entails allocating cores to applications and binding tasks of the applications to particular cores. Accurate on-the-fly estimates of different core allocations w.r.t. application performance are required before binding the tasks to cores for execution efficiency. We propose an adaptive on-the-fly application performance model that largely alleviates this increasingly important problem. It allows reacting to spontaneous workload variations and it considers topological properties of resources. Extensive evaluations show that the average estimation error is reduced from 14.7% to 4.5%, resulting in high quality of on-the-fly adaptive application mappings. Our work is a first milestone towards optimality of systems that exhibit a high degree of spontaneous workload variations.**

## I. Introduction

Many-core systems pose the challenge of efficient system utilization, especially if application properties and the set of concurrently executing applications rapidly vary at runtime. Therefore, when running multiple applications in parallel, it is essential to allocate the *right* set of cores to the *right* applications at runtime and thus to achieve a high degree of efficiency. This not only entails the number of cores but also their topological location on the chip. For example, the work in [1] has shown a 42% reduction of application execution time through sophisticated application mapping. To be able to select the *right* cores dynamically, accurate performance estimates are necessary. These estimates rely on so-called application performance models. The performance of an application depends on the system on which it is executed, its input data, and the system load caused by concurrently running applications. As the number and type of concurrently executing applications is not known a priori, estimations that are solely based on offline analysis will most likely result in inaccurate estimates and thus in disproportionate mappings.

### A. Problem Definition and Motivation

A many-core system of $N$ cores is described by the set $\mathcal{C}$ of all cores $c_i$, i.e. $\mathcal{C} = \{c_1, \dots, c_N\}$. Each core $c_i$ has a unique topological location on the chip. Multiple applications $(A_1, \dots, A_M)$ are executed in parallel. Without loss of generality, applications are represented as task-graphs where tasks are represented as nodes $t_x \in A_i$ in a directed acyclic graph. Resource management entails the allocation of pairwise disjoint subsets of cores $C_{A_i} \subseteq \mathcal{C}, \forall i,j, i \neq j: C_{A_i} \cap C_{A_j} = \emptyset$ to each application $A_i$. An estimation of the speedup $S_{A_i}(C)$ is used to decide which application $A_i$ should use which particular cores $C$ to optimize the overall execution efficiency. The speedup $S_{A_i}(C)$ is the quotient of the time $T_{A_i}(c_k)$ that application $A_i$ requires to perform its computations on a single core $c_k$ and the time $T_{A_i}(C)$ when executed on the set of cores $C$ (see (1)).

$$S_{A_i}(C) = T_{A_i}(c_k)/T_{A_i}(C) \tag{1}$$

In the remainder of this paper, the speedup of an application is used as a relative performance metric. $T_{A_i}(C)$ not only depends upon the number of cores $|C|$ but also upon their relative topological location to each other (see Section IV). However, determining the speedup $S_{A_i}(C)$ while considering the topological location of the cores in $C$ is in general computationally more intensive than $S_{A_i}(|C|)$ where only the number of cores is considered. Due to the number of different subsets of cores, it is *not* feasible to pre-compute and store all combinations. Even when only considering different subsets of cores with an identical number of cores ($|C| = |\hat{C}|$, but $C \neq \hat{C}$), there are ~$10^{47}$ possible ways to select 40 different cores out of 256 cores (binomial coefficients). The metric $S_{A_i}(|C|)$ estimates the same speedup for all these subsets, but actually, they differ depending on the topological location of the cores. Therefore, ignoring the topology of the resources assigned to each application leads to inefficient application mappings. Accurately calculating $S_{A_i}(C)$ by using the scheduling heuristic presented in [2] requires several milliseconds (validated by measurements of our implementation on an Intel i5-2500 at 3.3 GHz), depending on the number of tasks in the application task-graph and the number of cores in $C$. However, many estimations have to be calculated per mapping decision – the hill-climbing heuristic used in our evaluations estimates more than 10,000 combinations of $C$ to find a good distribution of 256 cores to 10 different applications (and even 62,000 combinations to distribute 256 cores to 20 applications). Optimistically assuming that there is no additional overhead and that the average computation time of determining $S_{A_i}(C)$ is just 1 ms, this results in a total runtime of 10-60 seconds to determine the resource allocation for 10-20 applications on a 256-core system. This is an unacceptable high latency, e.g. when starting a new application.

## II. Related Work

Both, Amdahl's law (that describes the theoretical maximal speedup an application can achieve based on its sequential code segments) and Gustafson's law (that describes how much the problem size could be increased when using multiple cores) do not explicitly consider the on-chip interconnect of todays and future many-core systems. The authors of [3] propose an updated version of both laws; however, they do not consider the (absolute or relative) topological location of the cores. A generic application performance model for parallel workloads has been presented in [4]. It models the achievable speedup of real-world applications when executed on computer clusters. Again, the model does not consider the topological location of the allocated cores. Other approaches to estimate the achievable performance of an application use machine-learning techniques like artificial neuronal networks [5]. Similar to offline-parameterized models, they require an exhaustive training phase before they are able to predict the application performance.

The importance of application performance models for many-core resource management ranges from on-chip many-core systems [6, 7] over high-performance computing [8], to cloud computing environments [9]. Due to the lack of adaptive models, these approaches rely on static models and – as it is not part of their models and/or not important for their target architectures – they do not consider the topological location of cores. In contrast, typical NoC mapping approaches (e.g. [1, 10, 11]) in fact do consider the topological location of cores but they do not dynamically determine the number of cores allocated to applications. In [11] the weighted communication of an application (WCA) is defined as the sum of the products of the communication volumes of the messages sent between all tasks of the application $A_i$ and the communication distance between the cores to which the tasks have been mapped. The WCA is used as optimization function to achieve a good application performance. Determining the WCA of an application $A_i$ given an allocation of cores $C$ requires to first map the tasks to the cores, a computationally intensive operation that hampers the evaluation of many different allocations $\hat{C}$.

SEEC [12] is a combination of an adaptive performance model and a resource manager. At runtime, it combines predefined mapping combinations in a way that maximizes the estimated performance. In contrast to the previous approaches, SEEC adapts to changes in application and system models. However, the number of predefined mapping combinations gets too large in systems with hundreds of cores – especially when not only the number of cores but also their topological location would be considered. Originally, SEEC is tailored to shared-memory systems and it only considers the number of cores.

In summary, there are application performance models that either adapt at runtime, consider the topological properties of the cores allocated to an application, or allow on-the-fly performance estimates. However, there is no adaptive model suitable for on-the-fly application performance estimation on NoC-based many-core systems considering the topology of cores.

**Therefore, we propose** an adaptive on-the-fly application performance model that considers the topological properties of the cores allocated to an application in NoC-based many-core systems. It uses a simple metric of $C_{A_i}$ that can easily be determined to estimate the achievable speedup based on the lower bound and the upper-bound speedup of the application. To handle highly dynamic behavior of workloads not known a priori, our performance model is continuously adapted at runtime.

## III. System Overview and Models

To enable our adaptive on-the-fly application speedup estimation, we use several interacting components shown in Fig. 1.

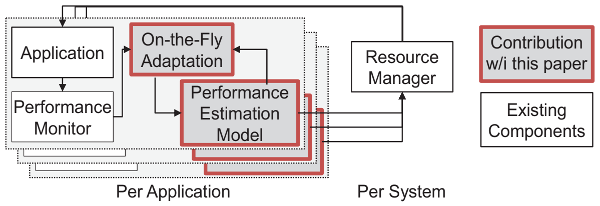
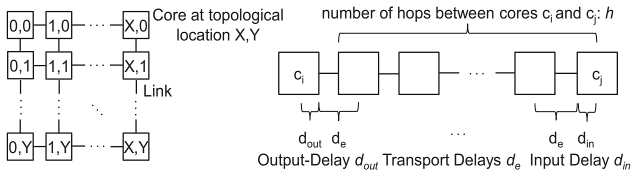


Fig. 1 Components used at runtime and their interactions

Each application is instrumented to allow the on-the-fly adaptation to monitor the performance, similar to the work presented in [13]. The resource manager uses the performance estimation models of the individual applications to decide which application $A_i$ should use which cores $C_{A_i}$ to improve the overall execution efficiency. Actually, any resource manager that considers an estimated relative application performance (e.g. [6-8]) can make use of our model and would benefit from the improved accuracy and adaptability with only minor modifications.

### *A. System and Network Model*

We target homogeneous many-core systems, as depicted in Fig. 2 and assume more cores $N$ than concurrently running applications $M$ (an application may have a large number of tasks, though). As recommended by recent many-core operating system research (e.g. [14]), this allows allocating cores temporally exclusively to applications and to avoid multiplexing overhead.



0,0
1,0
X,0
Core at topological location X,Y
0,1
1,1
X,1
Link
0,Y
1,Y
X,Y
number of hops between cores $c_i$ and $c_j$: $h$
$c_i$
$c_j$
$d_{out}$ $d_e$
$d_e$ $d_{in}$
Output-Delay $d_{out}$ Transport Delays $d_e$ Input Delay $d_{in}$

Fig. 2 System and Network Delay Model Components

The task-graph of an application $A_i$ is represented as a directed acyclic graph where the nodes $t_x$ correspond to actual computation tasks (expressed as workload $w_x$ for each task $t_x \in A_i$) that the application performs and where the edges correspond to the communication volume $v_{a,b}$ between two tasks $t_a$ and $t_b$ (cf. Fig. 3a). To model applications with periodic pattern, we extend the graph to allow cycles. This is shown in Fig. 3b where an inner task-graph is encapsulated in a larger *block* (flanked by a *begin node* and an *end node*) that is repeatedly executed in an outer loop.

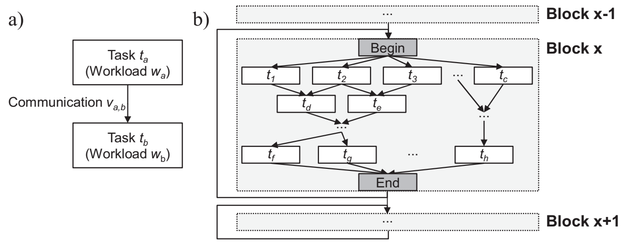


Fig. 3 Task-graph structure a) main components and b) extension that allows modeling applications with periodic patterns

A task is ready for execution as soon as all its predecessors have finished execution and their transmitted data has been received. Communication volume $v_{a,b}$ is set to zero if both tasks $t_a$ and $t_b$ are executed on the same core. Otherwise, the communication causes a delay in the NoC, as communication takes place by message passing in a 2D-mesh NoC. In [11] it has been argued that on average the link contention and buffer utilization in the routers lead to a linear relation between the delay $d_{NoC}$ of a message and its size $v$ multiplied by the required number of hops $h(c_i, c_j)$, as shown in (3).

$$h(c_i, c_j) = (|c_i.x - c_j.x| + |c_i.y - c_j.y|) \quad (2)$$

$$d_{NoC}(v, c_i, c_j) \approx d_e \times v \times h(c_i, c_j) \quad (3)$$

## IV. On-the-Fly Speedup Estimation

Based on the communication delay equation given in (3), we propose to improve the accuracy of our speedup estimation based on the average number of hops between the cores in $C$: the sum of the number of hops $h(c_i, c_j)$ between any two cores $c_i, c_j$ divided by the number of these combinations (4).

$$h_{avg}(C) = \frac{\sum_{c_i \in C} \sum_{c_j \in C \setminus c_i} h(c_i, c_j)}{|C| \times (|C| - 1)} \quad (4)$$

The average number of hops in $C$ depends on the selection of the cores. E.g. for $|C| = 40$ out of $|\mathcal{C}| = 256$ cores, the difference between the best case (spatially as close as possible to each other, 4.1 hops in average) and the worst case (spatially as far as possible from each other, 14.5 hops in average) is significant. The values of $h_{avg}^{min}(n)$ for the best case selection of cores $C$ (see (5)), and $h_{avg}^{max}(n)$ for the worst case selection (see (6)) are shown in Fig. 4.

$$h_{avg}^{min}(n) = \min\left\{h_{avg}(C) \mid C \subseteq \mathcal{C}, |C_{A_i}| = n\right\} \quad (5)$$

$$h_{avg}^{max}(n) = \max\left\{h_{avg}(C) \mid C \subseteq \mathcal{C}, |C_{A_i}| = n\right\} \quad (6)$$

To obtain $h_{avg}^{min}(n)$ and $h_{avg}^{max}(n)$, $C$ is iteratively extended by the core with the lowest/highest spatial distance to the cores already included in $C$, starting from the core topologically in the middle of the system for the best-case respectively the core in the top-left corner in the worst-case. Mappings should achieve $h_{avg}(C)$ close to the respective best case $h_{avg}^{min}(n)$ to optimize application performance. The worst-case average distance $h_{avg}^{max}(n)$ represents the worst-case selection of $n$ cores in $C$ that might occur when mapping the application, e.g. in a system that is heavily loaded with other concurrent applications.

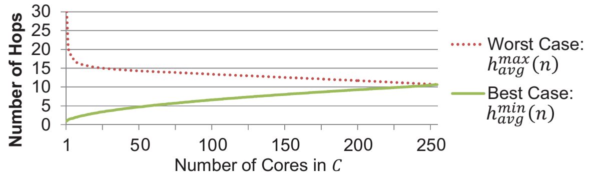


Fig. 4 Average number of hops required to send a message in $C$

Fig. 5 shows for two applications the influence that different sets of cores $C$ have on the achievable speedup. The first application task-graph is based on execution traces of a robotic vision application, the second application represents a task-graph with a high communication density between the individual tasks, resulting in a high correlation between the achievable speedup and the selection of $C$.

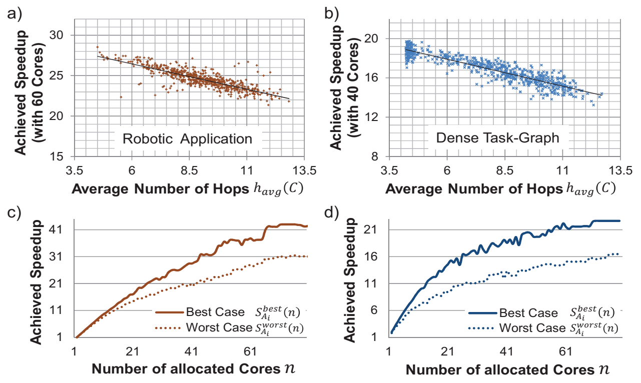


Fig. 5 Influence of different metrics [average number of hops in a), b) and number of allocated cores in c), d)] on the speedup compared to execution on a single core

Fig. 5 a) and b) show the influence of the average number of required hops $h_{avg}(C)$ on the speedup of the applications. We found that the average number of required hops $h_{avg}(C)$ correlates almost linearly with the speedup $S_{A_i}(C)$. The speedup curves $S_{A_i}^{best}(n)$ and $S_{A_i}^{worst}(n)$ resulting from executing both applications on the best case (see (7)) respectively worst case (see (8)) selections of cores in $C$ are shown in Fig. 5 c) and d).

$$S_{A_i}^{best}(n) = \max\left\{S_{A_i}(C) \mid C \subseteq \mathcal{C}, |C| = n, h_{avg}(C) \triangleq h_{avg}^{min}(n)\right\} \quad (7)$$

$$S_{A_i}^{worst}(n) = \min\left\{S_{A_i}(C) \mid C \subseteq \mathcal{C}, |C| = n, h_{avg}(C) \triangleq h_{avg}^{max}(n)\right\} \quad (8)$$

## A. *Considering the Topology for Speedup Estimation*

Based on the almost linear relationship between the achieved speedup of an application $A_i$ and the average number of hops $h_{avg}(C)$ between the cores in $C$ for any number of cores $n = |C|$, we propose to use this simple metric to allow our performance model to consider the relative topological location of the cores $C$ allocated to $A_i$. As the influence of $h_{avg}(C)$ on the speedup is different for each application, we combine the two corner-cases: the highest achievable speedup $S_{A_i}^{best}(n)$, and the worst case speedup $S_{A_i}^{worst}(n)$. We then linearly interpolate between both values based on the value of $h_{avg}(C)$, as shown in (9).

$$S_{A_i}(C) = S_{A_i}^{worst}(n) + \frac{h_{avg}^{max}(n) - h_{avg}(C)}{h_{avg}^{max}(n) - h_{avg}^{min}(n)} \times \left(S_{A_i}^{best}(n) - S_{A_i}^{worst}(n)\right) \quad (9)$$

To efficiently represent the speedup curves $S_{A_i}^{best}(n)$, and $S_{A_i}^{worst}(n)$ we use the application performance model introduced by Downey [4], as it captures the behavior of real applications running on real parallel machines very well and it is widely used in the field of many-core resource management, e.g. [6-8]. It uses two parameters (the average parallelism $P$ of an application and its variance in parallelism $\sigma$) to describe typical application speedup curves, similar to the ones shown in Fig. 5c) and d). Downey's application model does not consider the topological location of the cores, i.e. it is topology-agnostic.

## B. *Offline Parameterization of Model Parameters*

To initialize the model parameters, applications are individually profiled offline. Therefore, the application is executed on different sets of cores $C$ starting from one core ($|C| = 1$) through all cores in the system ($C = \mathcal{C}$). For each number of cores $n = |C|$ the cores are once selected to be spatially as close as possible to each other (see (7)) to obtain $S_{A_i}^{best}(n)$ and once to be spatially as far as possible from each other (see (8)) to obtain $S_{A_i}^{worst}(n)$. Then Downey's model is parameterized to match the observed $S_{A_i}^{best}(n)$ and $S_{A_i}^{worst}(n)$. The parameters for Downey's model are stored in the tuple $\left(P_{A_i}^{best}, \sigma_{A_i}^{best}, P_{A_i}^{worst}, \sigma_{A_i}^{worst}\right)$ for each *block* (see Fig. 3b) of the application. The offline analysis obviously does not cause runtime overhead. Applications may be profiled at any time before they are executed on the system, i.e. it is not required to know all applications at design time.

## C. *On-the-fly Adaptation of Model Parameters*

To improve the accuracy of the performance predictions, we propose to adapt the parameters of our application performance model continuously to react to spontaneous workload variations. Therefore, the measured performance of multiple executions is used to adapt the parameters of the model in a way that minimizes the error between the measured speedup $S_{A_i}^{m}(C)$ (see (10)) and the estimated speedup $S_{A_i}(C)$. We adopt the heartbeat framework [13] to measure the time required by the application to complete each *block* (see Fig. 3b). Applications emit heartbeats in the *start* and *end* nodes that flank each block. The time that passes between those two heartbeats, including the communication delays, is the runtime $T_{A_i}(C)$. The execution time $T_{t_x}$ of each task $t_x \in A_i$ is determined by subtracting the task's start time from its end time, to obtain the momentary dynamic workload of the application. Whenever the end node triggered its heartbeat, $S_{A_i}^{m}(C)$ is calculated according to (10), $S_{A_i}^{m}(C)$ and $C$ are stored in the sample history $Hist_{A_i}$, and the model parameters of $S_{A_i}$ are adapted to improve the accuracy of subsequent estimations of $S_{A_i}(\hat{C})$ for different allocations $\hat{C}$.

$$S^m_{A_i}(C) = \left(\sum_{t_x \in A_i} T_{n_x}\right) \Big/ T_{A_i}(C) \qquad (10)$$

Due to the monotonic increasing behavior of the speedup curves $S^{best}_{A_i}(n)$ and $S^{worst}_{A_i}(n)$ represented with Downey's performance model, we are able to use a fast hill climbing heuristic to adapt the parameters. The goal of the adaptation (Listing 1) is to minimize the weighted estimation error that is determined by comparing the measured values $S^m_{A_i}(C)$ with $S_{A_i}(C)$ for the samples stored in $Hist_{A_i}$ (Listing 2). To focus on recent changes in the workload, the weight of older samples is reduced exponentially.

```
Input: $S_{A_i}$ as $(P^{best}_{A_i}, \sigma^{best}_{A_i}, P^{worst}_{A_i}, \sigma^{worst}_{A_i})$
Output: Updated $S_{A_i}$
1.  $W_\sigma = 0.1$  // initial step width for parameters σ
2.  $W_P = 1$  // initial step width for parameters P
3.  $\delta = 1$  // start with initial step width
4.  $gain = 1$
5.  while ($gain > 0$ and $\delta > 0.2$) do  // at most 15 iterations
6.      // assemble neighboring configurations NC
7.      $NC = (P^{best}_{A_i} \pm \delta W_p, \sigma^{best}_{A_i}, P^{worst}_{A_i}, \sigma^{worst}_{A_i})$
8.      $NC = NC \cup (P^{best}_{A_i}, \sigma^{best}_{A_i} \pm \delta W_\sigma, P^{worst}_{A_i}, \sigma^{worst}_{A_i})$
9.      $NC = NC \cup (P^{best}_{A_i}, \sigma^{best}_{A_i}, P^{worst}_{A_i} \pm \delta W_p, \sigma^{worst}_{A_i})$
10.     $NC = NC \cup (P^{best}_{A_i}, \sigma^{best}_{A_i}, P^{worst}_{A_i}, \sigma^{worst}_{A_i} \pm \delta W_\sigma)$
11.     $\hat{S}_{A_i} = \mathrm{argmin}_{S \in NC}\{HistoryError_{A_i}(S)\}$
12.     $gain = HistoryError_{A_i}(S_{A_i}) - HistoryError_{A_i}(\acute{S}_{A_i})$
13.     if ($gain > 0$) then $S_{A_i} = \hat{S}_{A_i}$
14.     $\delta = 0.9 \times \delta$ // reduce step width with each step
15. end while
```

Listing 1: On-the-fly Model Adaption

```
Input: Speedup estimation function $S_{A_i}$, History $Hist_{A_i}$
Output: Weighted sum of the squared estimation errors
1.  $error = 0; \alpha = 1$
2.  for each ($s \in Hist_{A_i}$) do  // ordered chronologically
3.      $error = error + \alpha\left(S_{A_i}(s.C) - s.S^m_{A_i}(s.C)\right)^2$
4.      $\alpha = 0.9 \times \alpha$  // reduce the weight of older measurements
5.  end for each
6.  return $error$
```

Listing 2: History Sample Model Error

To avoid miss-adaptions caused by an empty sample ry $Hist_{A_i}$, measurements from previous executions of the application remain in the history. The actual adaptation works by repeatedly selecting a new configuration $\hat{S}_{A_i}$ based on $S_{A_i}$ that reduces the estimation error until no further error reduction is achieved or the step width $\delta$ is below a threshold. One of the model parameters is either increased or decreased, resulting in eight possible ways to configure $\hat{S}_{A_i}$ based on $S_{A_i}$. The configuration $\hat{S}_{A_i}$ with the lowest estimation error for the values in $Hist_{A_i}$ is chosen as the new configuration for $S_{A_i}$ if it resulted in less error than the current configuration. The step width $\delta$ by which the parameters are changed is reduced in each iteration to allow for a fast adaptation of the parameters at the beginning and a fine-tuning of the parameters at the end. As the meaningful range of the parameters $\sigma$ and $P$ differs, the value $\delta$ by which they are changed is fit by the constant factors $W_\sigma$ and $W_P$.

## V. Evaluation and Results

We evaluate the on-the-fly estimation accuracy of our application performance model (APM) as well as the induced runtime overhead to show its ability to adapt to spontaneous workload variations, which results in highly efficient utilization of many-cores. We compare with state-of-the-art multi-application multi-step (MAMS) mapping [10] as well as Downey's topology agnostic application performance model [4]. All results were obtained through mixed-level simulation of the NoC-based many-core system with $N = 256$ (16x16) cores. Computation is simulated at the abstraction level of task-graphs, i.e. at a much more detailed level than our on-the-fly application performance estimation model. Communication is simulated for a NoC that employs XY-Routing and considers link congestion for an accurate and mapping-dependent simulation. We use the task scheduling presented in [2] to map application tasks to the cores $C_{A_i}$ allocated to $A_i$ and to schedule their execution. The scheduler selects the task $t_x \in A_i$ with the highest *upward rank* value (i.e. the length of the critical path to the exit task, including computation costs and communication costs) and assigns that task to the core that provides the earliest finish time.

### A. Estimation Accuracy

To evaluate the accuracy of our application performance model, we execute three different application scenarios plus a full robotic vision application on various allocations of cores $C$. The total workload of the three scenarios and the number of tasks is almost the same; but they differ in their communication density, i.e. the number of edges in the application task-graphs. The first/second/third scenario represents an application with sparse/medium/dense communication between the individual tasks, respectively.

We compare to Downey's topology-agnostic application performance model from [4]. Additionally, very accurate (but computationally intensive) estimates of the achievable speedup are obtained by mapping the application tasks to the allocated cores [2] and using the offline profiled execution times to determine the expected execution time of the applications. To compare the accuracy of these three estimates, they are compared to the real measured speedup $S^m_{A_i}(C)$.

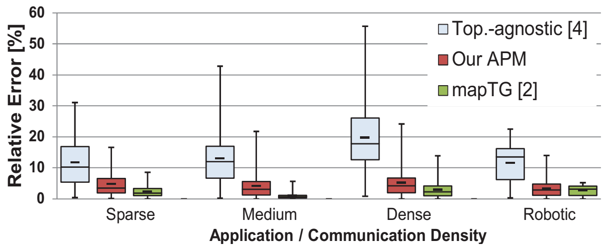


Fig. 6 Relative estimation error when estimating the speedup of the same applications executed on different sets of cores

Fig. 6 shows the relative error of estimations (note that the mapping efficiency with MAMS [10] is compared later on, but as MAMS does not perform speedup estimates, it cannot be compared here). The instances of $C$ were selected randomly. In all cases, the accuracy of our model is significantly better compared to using the topology-agnostic estimate [4] that does not consider the relative topological location of the allocated cores. At comparable computational effort (evaluated in Section V.B), our model predicts the speedup with an average relative error of 4.5% compared to 14.7% [4]. The computational intensive estimation [2], achieved by mapping the applications task-graph to the different allocations $C$, still results in an average relative error of 1.9% caused by dynamic execution behavior. The maximum estimation error is significantly lower than the errors that occur with the topology-agnostic speedup estimation, leading to more efficient application mappings, as shown in Section V.D.

## B. *Overhead Analysis*

An application performance model that is suitable for runtime resource management should have low computational effort to allow for evaluating many different resource allocations on-the-fly (e.g. thousands of different resource allocations in tens of milliseconds) for frequent adaptations of the application mapping. Determining the best-case speedup $S_{A_i}^{best}(n)$ and the worst-case speedup $S_{A_i}^{worst}(n)$ requires only a small number (≤10) of additions/multiplications and thus completes in a few CPU cycles. An ample computation of $h_{avg}(C)$ is within $\Theta(n^2)$ with $n = |C_{A_i}|$. In practice, often only one core is added to or removed from $C_{A_i}$ such that the complexity of determining $h_{avg}(C)$ is reduced to $\Theta(n)$ by adding/removing only the connections to the added/removed core. This results in a runtime that is approximatively $n$ times higher than the topology-agnostic model, but compared to the runtime of the task-mapping heuristic [2] it is negligible. We measured the execution time of the implementations on an Intel i5-2500 CPU and found that for estimating the speedup of an application running on 40 cores, Downey's topology-agnostic model [4] required 31ns (i.e. about 100 cycles) and our resource-aware approach required 1208ns. The task-mapping heuristic [2] required even 2ms, a factor 2000 slower per speedup estimation. The execution time does not depend on the number of cores in the system; only the number of cores that is assigned to an application has an influence – additional measurement results for 10 cores and 100 cores are shown in Fig. 7.

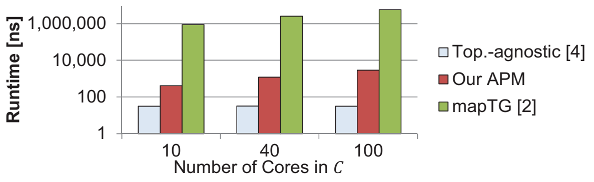


Fig. 7 Time (in ns) required to estimate the speedup of an application

Whenever an application completed execution of a block (see Fig. 3b), the model parameters are adapted. The adaptation needs to perform at most 15 iterations of the outer loop in which eight configurations of $S_{A_i}(\hat{C})$ are evaluated against the measured samples stored in $Hist_{A_i}$. As $h_{avg}(C)$ is already stored in $Hist_{A_i}$, the computational complexity of $S_{A_i}(C)$ is reduced to the complexity of Downey's performance model. Therefore, the adaptation overhead is within $\Theta(|Hist_{A_i}| \times 15 \times 8)$. By limiting $|Hist_{A_i}|$ to 10, adaptation of the model parameters can be accomplished within 50µs, which is practicable and feasible.

## C. *Evaluating Application Mapping*

To evaluate the influence of the performance estimation accuracy on the actual system performance, we use a hill climbing heuristic to decide the application mapping for different scenarios. Note, that any other resource manager that relies on an application performance model may be used as well. The heuristic determines the resource allocation $\{C_{A_1}, C_{A_2}, \dots, C_{A_m}\}$ by (re-)allocating the cores in $C$ among the applications $\{A_1, A_2, \dots, A_m\}$ as long as this (re-)allocation improves the overall system performance determined by the sum of all applications' estimated speedup $\sum_{i=1}^{m} S_{A_i}(C_{A_i})$. We use it to compare our on-the-fly performance model, the topology-agnostic application performance model presented in [4], and the computationally intensive task-graph mapping heuristic [2] to evaluate potential allocations.

As discussed in the related work section, there is no directly comparable model that estimates the speedup of an application for a set of cores $C$ that also considers the topological properties of the cores. However, there are runtime application-mapping approaches for many-core systems that aim to optimize the communication latencies without explicitly estimating the application performance. We choose the Multi-Application Multi-Step (MAMS) mapping method presented in [10] as reference implementation, as it has a similar optimization goal and achieves state-of-the-art application mapping quality. Similar to our approach, in the first step the cores $C_{A_i}$ are selected for each application $A_i$. In the second step, the tasks of the applications are mapped to these cores. The selection of $C_{A_i}$ works by finding rectangular areas on the chip that are not assigned to an other application and that can accommodate the tasks of each application $A_i$. The optimization of the communication latencies is achieved by focusing on rectangles as square as possible. MAMS only optimizes the topological location – for a fair comparison, we therefore improved MAMS to determine the ideal number of cores for each application by using the static topology-agnostic application performance model presented in [4]. We further found that in most cases the allocation of rectangles for each application leads to a fragmentation of the application mapping, i.e. there are cores left over that cannot be allocated to an application. As the application mappings achieved by the hill climbing heuristic consumes all available cores, we improved MAMS in the way that, after the rectangles for each application have been defined, we distribute the unallocated cores to spatially adjacent applications. Again, the performance model presented in [4] is used to decide which of the adjacent applications would benefit the most from additional cores.

## D. *Adaptation to Workload Variations*

We use the presented mapping heuristics to evaluate the benefits of the improved accuracy and the adaptability of our speedup estimation model for various combinations of different applications in different scenarios. These scenarios have been selected to demonstrate the adaptability of our model as well as the broad applicability to different and typical kinds of system utilization. The speedup is determined through simulating the applications. As comparison metric, the efficiency of the application mapping is obtained through the average speedup per core (see (11)).

$$Efficiency = \frac{\sum_{i=1}^{m} S_{A_i}(C_{A_i})}{|C|} \qquad (11)$$

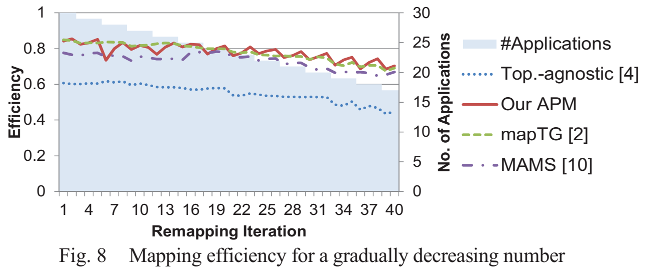


Fig. 8 Mapping efficiency for a gradually decreasing number of applications; the adaptation of our model is visible in the small improvements after each change in the workload

In the first scenario, the number of concurrent applications is gradually reduced from 30 to 17 (Fig. 8). The resulting application mappings obviously benefit from the more accurate estimates of our resource-aware adaptive model. The application mappings for this scenario are 8.0% more efficient when using our resource-aware adaptive estimates compared to state-of-the-art (MAMS) runtime mapping [10].

In the second scenario, the number of applications increases abruptly (Fig. 9). Again, the resulting application mappings achieved when using our application performance model are almost always better than the mappings from MAMS. The adaptation of our model to the new operating conditions is clearly visible, i.e. the application mapping efficiency increases with each application-re-mapping iteration after the abrupt workload change. The application mapping achieved by using our resource-aware estimation model resulted on average in a 4.3% improvement compared to MAMS.

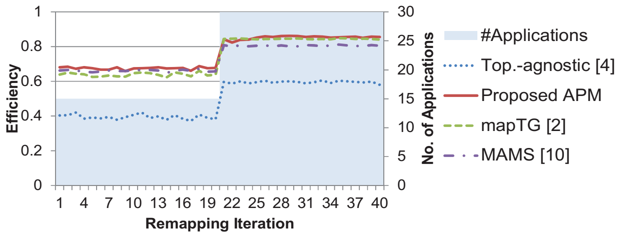


Fig. 9 Mapping efficiency for an abrupt change in the workload

For the third scenario (shown in Fig. 10), we analyze the mapping efficiency for random changes in the workload which is common for interactive system utilization. Especially in the cases of multiple concurrent applications, mappings achieved through our adaptive on-the-fly performance model outperform state-of-the-art [10]. This is because not all applications necessarily benefit to the same degree from spatially close cores. In contrast to MAMS, our resource-aware model allows some applications to use slightly scattered sets of cores to allow allocating larger coherent sets of cores to those applications that benefit more from the smaller average number of hops.

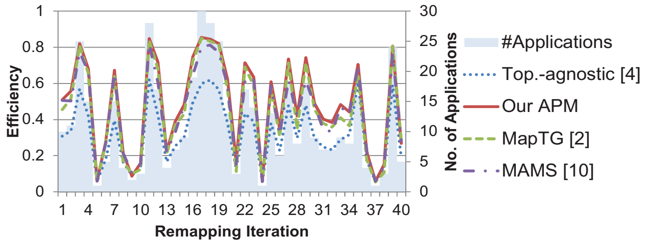


Fig. 10 Mapping efficiency for random changes in the workload, e.g. as in interactive utilization

On average, our resource-aware estimation achieves 7.4% more efficient application mappings than when using MAMS and it provides almost the same mapping quality compared to the most accurate but computationally unfeasible estimation that calculates the task-graph mapping for each estimation. On average, the application mapping efficiency is improved by 6.4% compared to state-of-the-art MAMS runtime application mapping for many-cores [10]. The resulting application mappings are significantly (32%) more efficient than the application mappings that result from using the topology-agnostic speedup estimation model [4].

## VI. Conclusions

A novel adaptive on-the-fly application performance model has been presented. We have demonstrated that the accuracy of our performance estimations results in overall high execution efficiency when using application performance estimates for application mapping decisions. Our evaluations show that the average estimation error is reduced from 14.7% to merely 4.5% while at the same time significantly improving the accuracy, i.e. the worst-case error is reduced from 57% to 24%. As a result, we profit from better application mappings compared to state-of-the-art. Our work is the first of its kind for managing many-core systems that exhibit rapid and spontaneous workload variations while still maintaining high mapping quality.

## 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] C. Marcon, A. Borin, A. Susin, L. Carro, and F. Wagner. Time and energy efficient mapping of embedded applications onto NoCs. In *Asia and South Pacific Design Automation Conference (ASP-DAC)*, Jan. 2005, pp. 33–38.

[2] H. Topcuoglu, S. Hariri, and M.-y. Wu. Performance-effective and low-complexity task scheduling for heterogeneous computing. *Trans. on Parallel and Dist. Systems*, pp. 260–274, 2002.

[3] T. Huang, Y. Zhu, M. Qiu, X. Yin, and X. Wang. Ext. Amdahl's law and Gustafson's law by evaluating interconnections on multi-core processors. *Journal of Supercomp.*, pp. 305–319, 2013.

[4] A. B. Downey. A parallel workload model and its implications for processor allocation. *Cluster Computing*, pp. 133–145, 1998.

[5] E. Ipek, B. R. De Supinski, M. Schulz, and S. A. McKee. An approach to performance prediction for parallel applications. In *Parallel Processing (Euro-Par)*, 2005, pp. 196–205.

[6] S. Kobbe, L. Bauer, D. Lohman, W. Schröder-Preikschat, and J. Henkel. DistRM: Distributed resource management for on-chip many-core systems. In *Int. Conf. on Hard-/Software Codesign and System Synthesis (CODES+ISSS)*, 2011, pp. 119–128.

[7] I. Anagnostopoulos, V. Tsoutsouras, A. Bartzas, and D. Soudris. Distributed run-time resource management for malleable applications on many-core platforms. In *Design Automation Conference (DAC)*, 2013, pp. 168–174.

[8] 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, pp. 94–114.

[9] Q. Zhu and G. Agrawal. Resource provisioning with budget constraints for adaptive applications in cloud environments. In *Int. Symp. on High Perf. Dist.Computing (HPDC)*, 2010, pp. 304–307.

[10] B. Yang, L. Guang, T. Säntti, and J. Plosila. Mapping multiple applications with unbounded and bounded number of cores on many-core networks-on-chip. *Microprocessors and Microsystems*, vol. 37, no. 4, pp. 460–471, 2013.

[11] B. Yang, T. Xu, T. Säntti, and J. Plosila. Tree-model based mapping for energy-efficient and low-latency Network-on-Chip. In *Int. Symp. on Design and Diagnostics of Electronic Circuits and Systems (DDECS)*, April 2010, pp. 189–192.

[12] H. Hoffmann, M. Maggio, M. D. Santambrogio, A. Leva, and A. Agarwal. SEEC: A framework for self-aware management of multicore resources. Computer Science and Artificial Intelligence Laboratory, MIT, Tech. Rep., Mar. 2011.

[13] H. Hoffmann, J. Eastep, M. D. Santambrogio, J. E. Miller, and A. Agarwal. Application heartbeats: A generic interface for specifying program performance and goals in autonomous computing environments. In *Int. Conf. on Autonomic Computing (ICAC)*, 2010, pp. 79–88.

[14] J. A. Colmenares, S. Bird, H. Cook, P. Pearce, D. Zhu, J. Shalf, S. Hofmeyr, K. Asanovic, and J. Kubiatowicz. Resource management in the tessellation manycore OS. In *USENIX Workshop on Hot Topics in Parallelism (HotPar)*, 2010.