RUNTIME ADAPTIVE ALLOCATION OF DYNAMICALLY MIXED TASKS ON A HETEROGENEOUS MPSOC PLATFORM

Jia Huang, Andreas Raabe, Christian Buckl

fortiss GmbH
Guerickestr. 25, 80805 Munich, Germany
{huang,raabe,buckl}@fortiss.org

Alois Knoll

Technische Universität München
Boltzmannstr. 3, 85748 Garching, Germany
knoll@in.tum.de

ABSTRACT

Multiprocessor System-on-Chip platforms are typically used for co-hosting multiple tasks, which may start and stop execution independently at time instants unknown at design time. In such systems, the runtime resource manager is responsible for allocating adequate and appropriate resources to each task. We identify a key issue in existing work that the resource management algorithms consider the problem only at task-level, i.e. the optimization is performed for each individual task upon activation. However, it can be shown that such strategies are suboptimal from the system point of view. In contrast, we propose in this paper a new task allocation flow that considers the system-level resource management. Comparing with traditional techniques, significant performance improvement (up to 29.5%) is observed during evaluation using a standard benchmark set. In addition, the proposed task allocator features runtime self-adaptability with respect to changes in hardware and/or applications.

Index Terms— Processor scheduling, Resource management

1. INTRODUCTION

As the complexity of today's embedded applications increases continuously, there is trend toward adopting Multiprocessor System-on-Chip (MPSoC) platforms in embedded system design. Such platforms are typically integrated with multiple heterogeneous processing elements (PEs), memory blocks and a communication infrastructure. MPSoCs provide adequate computation power and flexibility for centralized execution of a wide range of applications.

In many application domains, multiple tasks can be executed concurrently on a shared MPSoC. For example, a Software Defined Radio (SDR) platform can co-host several independent communication standards. Another example would be a car infotainment system, in which different services like MP3 and DVD decoding need to be provided. A key characteristic of the aforementioned use cases is that the tasks can start and stop execution independently at time instants unknown at design time. This is usually due to the switch of the use scenarios, e.g. the user just turn off the MP3 player and start the DVD. Hence, dynamically varying combinations of tasks (called *task-sets*) are expected in MPSoC platforms. When a task gets activated, a resource manager is needed to allocate appropriate hardware resources (e.g. processor, memory and communication channel) to execute the task. On one hand, the resources allocated should be adequate to meet the application requirements, e.g. worst case response time and throughput. On the other hand, unnecessary over-allocation and overload of resources should be avoided to save space for future tasks. This step is known as *task allocation* (or *resource allocation* from the platform point of view), which is of prominent importance in the MPSoC design flow, since it can influence the hardware and energy efficiency dramatically. Especially for heterogeneous MPSoCs, in which the execution efficiency of the same task may differ significantly on different PEs, task allocation must be considered with utmost care.

The task allocation problem consists of two aspects, namely the spatial *task mapping*, i.e. where to execute the task, and temporal *task scheduling*, i.e. when to execute the task. Extensive studies tackling entire or parts of this problem can be found in literature. The proposed solutions basically split in two directions, namely the static and dynamic approaches. Examples of static approaches include [1, 2, 3], in which task allocation is formulated as static optimization problems and solved using standard techniques such as Integer Linear Programming (ILP) and Genetic Algorithms. However, due to the high complexity and huge execution time of these algorithms, optimization can only be performed offline at design time. This in turn requires that the set of possible tasks must be fixed and known, which is in many cases impractical. For example, the software components may get updated and new add-on modules may be downloaded. Another limitation of these studies is that only the optimization of a single task is considered.

In the dynamic approaches [4, 5, 6], the allocation scheme is calculated at runtime upon activation of the task, typically using efficient heuristics due to the timing constraints. In [5], a *hierarchical search with iterative refinement* algorithm is

presented, which targets on discovering the energy-optimal mapping with real-time guarantee. It is not clear in literature how this approach can optimize the resource allocation for multiple tasks. In [4], Moreira et al introduced a run-time resource allocator that is able to handle multiple real-time tasks. The task mapping is computed using vector bin-packing heuristics aiming at minimum resource usage. Thereafter, local scheduling analysis is applied on each PE to enforce real-time requirements. The method proposed in [6] is half way in between static and dynamic, in sense the task mapping is determined at design time and the temporal settings are computed at runtime.

In the existing work mentioned above, the task allocation is only considered at **task-level**, that is, the optimization is performed on each task according to the activation sequence. A key issue that is neglected is that the allocation of current task has substantial impact on the allocation of future tasks due to the limited availability of shared resources. Actually, those strategies might lead to adverse system configurations, as illustrated in the following example.

Motivating Example. Consider the example scenario depicted in figure 1, where we have 4 tasks (t_1 to t_4) to be mapped onto two processors (a RISC and a DSP) with the goal of minimizing resource utilization. The left side of figure 1 shows the processor and memory consumption of each task; the right side of figure 1 shows resource allocation according to two different mapping strategies. The activation sequence is assumed to be t_1, t_2, t_3, t_4. $Scheme1$ is the results from traditional algorithms that focus on minimizing the resource consumption of each individual task. When t_1 is firstly activated, the resource consumption of t_1 on RISC and that on DSP are compared. Obviously, the DSP is a better choice since the processor usage is smaller. The same is done for t_2. However, a direct consequence of such a mapping scheme is that t_3 cannot be mapped to DSP any more due to lack of resources. Hence, t_3 must be mapped to RISC although it is very inefficient. A further consequence is that t_4 cannot be mapped at all since not enough resources are available on both DSP and RISC. If we examine the tasks t_1, t_2 and t_3, it can be seen that, although all of them *prefer* the DSP, the gain of mapping them to DSP instead of RISC is different. For t_1, there is only a small gap in the resource consumption on DSP and RISC, whereas the difference is much larger for t_2 and t_3. Hence, when the DSP cannot accommodate all of them, mapping t_1 to RISC and saving the space for future tasks t_2 and t_3 is more favorable from the system point of view. This leads to a better mapping *Scheme2*, in which all 4 tasks can be allocated.

The above example clearly shows how the mapping of one task influences the mapping of other tasks. It is also shown that the task-level allocation algorithms can lead to suboptimal decisions for certain activation sequences. It is the objective of this study to develop a technique that optimizes the **system-level** task allocation for any activation sequences. We

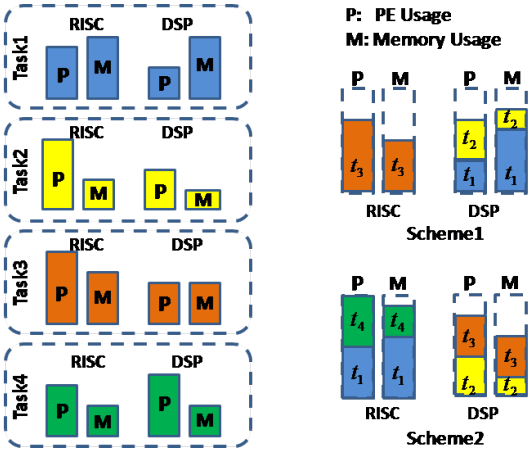

Fig. 1. Motivating Example

achieve this goal by introducing the concept *virtual resource consumption* (see section 3). The major contributions of this paper are:

- We make the key observation that the task-level optimal mapping and system-level optimal mapping can be different. An efficient heuristic is proposed, which can be used at runtime to improve the system-level resource management.
- We present a complete runtime task allocation flow targeting a realistic MPSoC Architecture *GENESYS*[1]. The allocation flow is self-adaptive to changes in hardware architecture and/or applications.

In the following section, we first describe the architecture and application models used in this work (section 2). The details of proposed task allocation flow are introduced afterwards in section 3. Experimental results are presented in section 4. A comparison of this study with most relevant work is presented in section 5. Section 6 concludes this paper.

2. SYSTEM MODELS

In this work, applications are described using the Task Graph model. A Task Graph $TG = (V, E)$ represents an independent **task**, whose vertices $v \in V$ represent **actors** (sub-tasks) of that task. For each actor a_i^l of task l, the Worst Case Execution Time (WCET) and memory consumption can be measured using simulation or external tools and can be annotated in the model. For simplicity, the **WCET** ($e_{i,x}^l$) in the entire paper refers to the stand-alone WCET for a_i^l on processor p_x without considering resource sharing. The Worst-Case Response Time (**WCRT**) ($w_{i,x}^l$) refers to maximum response time taking the waiting time caused by resource contention

[1] http://www.genesys-platform.eu/

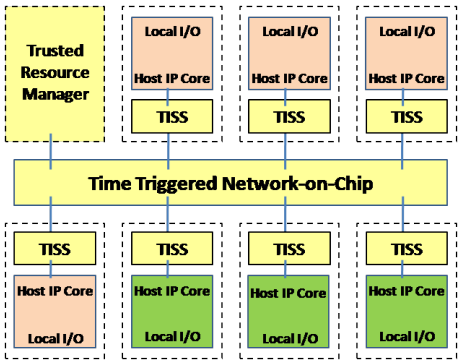

Fig. 2. Hardware Architecture

into account. Naturally, $w^l_{i,x}$ is in any case no less than $e^l_{i,x}$. The edges in E represent the dependencies between actors. For an edge (i, j), the target actor a_j can only start execution after receiving the required data (called **token**) from the source actor a_i. If the source and target actors are mapped to different PEs in the MPSoC, a communication channel needs to be established. Alternatively, if the two communicating actors reside in the same PE, token transfer can be done via local memory and no communication channel is necessary. The applications we considered are real-time tasks, which typically have global end-to-end deadlines from the source actors, where the task is activated, to sink nodes, where the response is made. The source-to-sink latency must be smaller than the deadline.

The target hardware platform in this work is the GENESYS MPSoC [7] as shown in figure 2. The center of the MPSoC is a time-trigger Network-on-Chip (NoC) for reliable communication, the time slots of which are shared by all **tiles**. The Trusted Resource Manager (TRM) is responsible for configuration of the NoC, e.g. time slot distribution, and it is the only entity allowed to do so. Each tile in GENESYS consists of a PE associated with memory and a Trusted Interface Sub-System (TISS). TISS is the bus guardian that guarantees no tile uses the bus out of its own time slot. Assume the NoC has a time wheel T and a time interval t is assigned to a communication channel c. The time needed to transfer the associated token in the worst case can be found as:

$$W = \left\lceil \frac{S_c}{tB} \right\rceil (T - t) + \frac{S_c}{B} \tag{1}$$

where S_c denotes the token size of channel c and B is the data rate of the NoC.

3. TASK ALLOCATION FLOW

This section describes the proposed task allocation flow. As shown in figure 3, based on the architecture and application models, application profile analysis is firstly performed to obtain important information for later steps, such as which types of resources are the most inadequate, etc. The analysis is done before activation of any tasks and has little impact on the runtime complexity. When a task needs to be allocated at runtime, the mapping of each of its actor is first computed. After that, local scheduling analysis is done on each PE to compute a safe WCRT of each actor. In the last step, we perform data flow analysis to check if the end-to-end deadlines can be met. If not, a feedback is generated to previous steps and other possible mappings are examined. If all deadlines are fulfilled, data flow analysis provides the slack time that can be used for token transfer. Adequate NoC bandwidth is allocated to each communication channel such that no deadline is violated.

3.1. Application Profile Analysis

The application profile analysis step aims at estimating the optimality of each mapping in the system scope. This is done as follows. Consider an actor a_i (from an arbitrary task) that can be mapped to a set of PEs denoted by P_i. For each feasible mapping, the processor utilization and *normalized* memory consumption of a_i on the PE $p_j \in P_i$ are denoted by $u_{i,j}$ and $m_{i,j}$, respectively. The utilization is defined as the ratio between the WCET of a_i on p_j and period of parent task. The normalization of memory consumption is done with respect to average available memory on each PE. To evaluate the efficiency of a certain mapping, we introduce the quantity *desirability*, which is defined as the relative difference between the resource consumption of a specific mapping and the average resource consumption over all possible mappings:

$$\bar{u_i} = \frac{\sum_{p_j \in P_i} u_{i,j}}{|P_i|}, \quad \bar{m_i} = \frac{\sum_{p_j \in P_i} m_{i,j}}{|P_i|}$$

$$d_{i,j} = \frac{(\bar{u_i} + \bar{m_i}) - (u_{i,j} + m_{i,j})}{(\bar{u_i} + \bar{m_i})} \tag{2}$$

As can be seen, the lower the resource consumption of a_i on p_j is, the higher the desirability of mapping a^l_i to p_j is. The type of PE that achieves highest desirability $d_{i,max}$ is called the most *preferred* type. The corresponding minimum utilization and memory consumption are denoted by $u_{i,min}$ and $m_{i,min}$.

An important use case of desirability is to decide the priority of actors that prefer the same PE. Particularly, if the preferred PE cannot accommodate all of those actors, tradeoffs must be made. Naturally, actors with low desirability should first be considered to move to other PEs. This is a typical situation that the runtime task allocator is facing. It must decide whether the actor should be mapped to its preferred PE or it should not since space needs to be saved for future actors. It becomes evident that the correct decision must be based on the desirability of other actors on the same PE and the amount of resources the PE can offer. To quantitatively evaluate this issue, we introduce another two quantities. Let A_j denote the

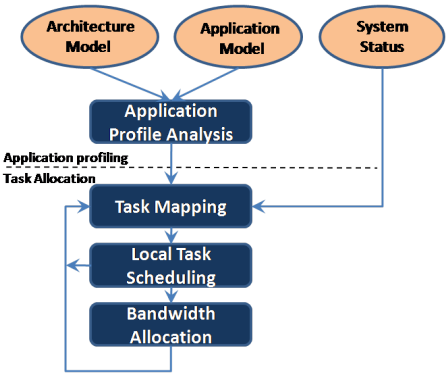

Fig. 3. Task Allocation Flow

set of actors that can be mapped to p_j. The *virtual resource demand* of an actor $a_i \in A_j$ on p_j is estimated as:

$$\hat{u_{i,j}} = u_{i,min}(1 + d_{i,j})$$

$$\hat{m_{i,j}} = m_{i,min}(1 + d_{i,j}) \tag{3}$$

The idea behind the virtual resource demand is that, with lower (or higher) desirability, the probability of mapping actor a_i on p_j is lower (or higher) according to the optimization criterion of task allocator. In other words, when the desirability is low, a_i will not use its full strength to request resources from p_j, and vice verse. Hence, the resource demand of a_i is estimated by scaling $u_{i,min}$ and $m_{i,min}$ based on the desirability $d_{i,j}$. With $\hat{u_{i,j}}$ and $\hat{m_{i,j}}$ for all $a_i \in A_j$, we can compute the overall *demand factors* of p_j by summarizing the virtual demands from all possible actors and then dividing the sum by total resources available on p_j:

$$\alpha_j = \sum_{a_i \in A_j} \hat{u_{i,j}}, \quad \beta_j = \frac{\sum_{a_i \in A_j} \hat{m_{i,j}}}{M_i} \tag{4}$$

Where M_i is the normalized memory available on p_j. The demand factors α_j and β_j represent the relative scarcity of certain types of resources. Naturally, PEs with relatively large demand factors are demanded by many actors and are likely to be heavy-loaded. To estimate the optimality of a mapping in the system scope, we compute the *virtual resource consumption* of a_i on p_j by weighting the original resource consumption by the demand factors of p_j:

$$v_{i,j} = v^u_{i,j} + v^m_{i,j} = \alpha_j u_{i,j} + \beta_j m_{i,j} \tag{5}$$

In the later task mapping step, instead of comparing the real resource consumption, the PE with smallest virtual resource consumption will be selected (see section 3.2). By computing the virtual resource consumption, we introduce a penalty on the usage of scarce resources such that allocation of scarce resource to actors with low desirability will be prevented, since the penalty overcomes the gain of such a mapping. This strategy actually allows the task allocator to look into the future and make better mapping decisions for the overall system. Concerning the motivating example, the mapping *Scheme2* will be chosen when allocating t_1, since future tasks t_2 and t_3 use the limited resources of DSP more efficiently.

3.2. Runtime Task Mapping

In the task mapping step, each actor is assigned to exactly one PE that has enough resources. As mentioned before, our goal is to optimize the system-level resource consumption. It is worth mentioning that besides minimizing the resource usage, another important aspect for optimization is to *balance* the resource consumption over PEs. The balancing consists of two aspects. First, on each tile, the usage of PE and memory should be balanced to avoid resource bottleneck [8]. Second, the workload should be balanced amongst available tiles to achieve a better energy and thermal profile [9]. This problem can be viewed as a multi-dimensional bin-packing problem. We developed a modified first-fit decreasing heuristic, the details of which are presented in the following.

When a new task is activated, its actors are visited in decreasing order according to averaged virtual resource consumption on all possible PEs. Sorting of actors needs to be performed only once in the application profile analysis step. From the biggest actor, the task allocator tries to map the actor onto the PE that minimizes the virtual resource consumption. If not enough resources are available on the most favorite PE, the task allocator sequentially examines other PEs with increased resource consumption. If multiple PEs of the same type exist, the one with least utilization is chosen. The above procedure continues until all actors are mapped. If at least one actor cannot be mapped on any PE, the task is not accepted for execution.

The procedure above guarantees feasibility of mapping from the resource availability point of view. Nevertheless, the real-time requirements must be also enforced. In the next steps, local scheduling analysis and global bandwidth allocation are performed. The former step provides an upper bound of WCRT of each actor. The later step determines the token transfer time of each channel. With this information, all end-to-end deadlines can be checked. If the timing constraints cannot be fulfilled, feedback is generated to the task mapping step to activate a timing adjustment phase, which works as follows. The task allocator firstly computes the Critical Path (CP) between the source and sink actors. For all actors in CP, the one with least average virtual resource consumption is chosen and the mapping is removed. Then, the task allocator checks if the WCRT of this actor can be reduced by mapping it to any other PEs. If yes, the new mapping is adopted and the end-to-end deadline is re-checked. If no, the mapping is kept unchanged and next actor is examined. The timing adjustment phase ends when the end-to-end deadline is fulfilled

or no timing improvement can be achieved.

3.3. Local Scheduling Analysis

With known task-to-PE mapping computed in previous step, classical single processor scheduling techniques can be reused to compute the WCRT of each actor under the resource sharing scheme. Any deterministic scheduling policy can be used here, e.g. TDMA and static priority preemptive scheduling. For TDMA arbitration, the WCRT can be found as:

$$w_{i,j} = \left\lceil \frac{e_{i,j}}{t_{i,j}} \right\rceil (W - e_{i,j}) + e_{i,j} \tag{6}$$

Where W is the TDMA time wheel, $e_{i,j}$ is the stand-alone WCET of actor a_i on p_j and $t_{i,j}$ is the time slot allocated for a_i. For static priority preemptive scheduling, standard techniques exist for WCRT computation [10, 11]. After the local analysis step, the obtained WCRTs are annotated in the model for later steps.

3.4. Global Bandwidth Allocation

The final step in the proposed workflow is to assign bandwidth to each communication channel. As mentioned before, a channel needs to be established only if the two communicating entities are mapped to two different PEs. The goal of this step is to allocate a minimal yet adequate time slot to each channel such that the end-to-end deadlines can be guaranteed with the token transfer time being taken into account. To do this, we first perform data flow analysis to obtain the timing budget for each channel. Since the WCRT is available from the previous step, the timing budget for all channels along an arbitrary path between the source and sink actors can be found as the difference between the accumulated WCRT of all actors along the path and the end-to-end deadline. The remaining slack time can then be distributed to all channels along the path proportionally to the token size. We start from the CP between source and sink and sequentially visit other paths until an appropriate timing budget is obtained for each channel. Then, the minimum possible time slot with which the token transfer can be finished within the timing budget is computed according to equation 1. If the required bandwidth is more than available, the new task is considered as non-schedulable and a feedback is generated to the task mapping step.

In the GENESYS architecture, the global NoC bandwidth is shared by all tasks. Hence, suboptimal usage of the bandwidth can degrade the system performance significantly. A bad situation that should be avoided is that the slack time for token transfer is very small, resulting in a huge request in global bandwidth. We solve the problem by two means. First, we introduce a minimum timing budget for token transfer that should be guaranteed by each task, e.g. 5% of the end-to-end deadline. Second, a runtime adaptive actor clustering technique is developed, which is presented in the next section.

3.5. Adaptive Actor Clustering

Since communication is one of the most critical issues in many MPSoC platforms, clustering approaches are developed in existing studies to reduce the bandwidth consumption. In [4], Moreira et al present a Clustering Before Packing (CBP) algorithm, which attempts to contract a certain amount of channels (e.g. 20%) before submitting the task to allocator. By actor clustering, two communicating actors are grouped into a single larger actor, forcing them to be mapped to the same PE. Clustering can also be done at runtime by intentionally mapping communicating actors to the same PE. A drawback of clustering is that large actors are produced, which make it more difficult to optimize the usage of other resources. Results in [4] show that clustering is not beneficial when the communication load is low, because of the side effect. Nevertheless, for communication-intensive tasks, the mapping success rate can be improved significantly by clustering.

Our runtime clustering technique aims at balancing the usage of *tile resources* (PE and memory) and global bandwidth. Assume an actor a_1 is already mapped to p_x and the task allocator is considering a communicating task a_2. If the preferred PE of a_2 is of the same type as p_x, clustering is taken as natural optimum. Otherwise a tradeoff needs to be evaluated. Assume p_y is the preferred PE of a_2, the loss in tile resources caused by clustering can be computed as:

$$C_{diff} = (v^u_{2,x} + v^m_{2,x}) - (v^u_{2,y} + v^m_{2,y})$$

The bandwidth that can be saved is estimated by:

$$C_{save} = S_{1,2}\bar{b}$$

Where $S_{1,2}$ is the token size and $\bar{b}$ is the average bandwidth per token size over all existing channels. Then, clustering is taken if the ratio between C_{save} and C_{diff} is larger than a threshold:

$$\frac{C_{save}}{C_{diff}} > \frac{B_{avail}}{U_{avail}} \tag{7}$$

Where B_{avail} and U_{avail} are respectively the current available bandwidth and tile resources in the system. As it can be seen, the clustering threshold is set according to the relative scarcity of bandwidth resource. When the bandwidth utilization grows faster than the tile utilization, the strength of clustering algorithm will increase, and vice verse. This makes the algorithm self-adaptive at runtime and allows for more balanced usage of bandwidth, PE and memory.

3.6. Discussion of Self-Adaptability

One advantage of the proposed workflow is self-adaptability. Given the hardware model and a set of tasks, the underlying task allocator can adapt to the application domain in the application profile analysis phase, in which the demand factors of each PE and virtual resource consumption of each actor are

approach used	#task graphs	hardware utilization	average usage
task-level	12.40	89%	7.17%
system-level	14.18	93%	6.56%

Table 1. Performance Evaluation of Task-Level and System-Level Algorithms

computed. This feature is particularly useful when changes in the system occur, e.g. when the system gets updated or when a new task is added to the system. In such cases, we just need to re-activate the analysis phase.

Our workflow can also benefit from execution probability information of tasks if available. Such information may be either known at design time or obtained by runtime profilers. For example, different users of a consumer device may have distinct favorite programs that execute more often. In such cases, a set of weighting factors can be easily added to equation 4 according to the relative occurrence probabilities. The task allocator is then adapted more to the highly probable tasks.

4. EVALUATION

To evaluate the performance of the approach, we use the standard task graph set available at [12]. The benchmarking suit contains both randomly generated TGs and graphs modeled from real world applications. The number of actors in each graph ranges from 50 to 100. A MPSoC with 9 PEs of 3 different types are used as the architecture model. On each PE, deadline monotonic scheduling is used as the local scheduling policy and the iterative algorithm from [11] is used to compute WCRT. In each experiment round, we pick up a new task set randomly from the TG suit as potential applications. Then, the tasks in this set are activated in a random sequence and submitted to the allocator. This procedure ends until the first allocation failure occurs. We run such experiment for 1000 rounds and record the average data, e.g. the number of tasks that is successfully allocated.

We first compare the performance of traditional approaches that optimize the resource consumption of each single task (labeled *task-level*) and the proposed approach that considers the system-level resource management (labeled system-level). Table 1 summarizes the results. As can be seen, the system-level algorithm allocates 14.18 TGs on average, which is 14% better than the task-level algorithm. The third and fourth columns of the table show respectively the total hardware utilization and average resource usage of each task. Clearly, the system-level algorithm achieves higher hardware efficiency. After analyzing the simulation traces, we find out that the performance gap comes from two sources. The first is unbalanced usage of the resources in the task-level algorithm. In many cases, either the processor or the memory is overloaded and becomes a bottelneck, which prevents further actors from being allocated. The second reason is that the most scarce resources are not used at their best in the task-level alogrithm. Since tasks are activated in random sequences, the task-level algorithm overlooks the side effect of a mapping on furture tasks and uses highly requested resources for actors with low desirability.

	approach used	#tasks graphs	HW util.	average usage	perf. drop
case1	task-level	10.71	86%	8.03%	14%
	system-level	13.86	97%	7%	2%
case2	task-level	10.04	85%	8.47%	19%
	system-level	13.00	95%	7.31%	8%

Table 2. Comparing the Performance Drop with Reduced Resource Availability

As explained before, self-adaptability is one of the major improvement of the proposed task allocator. In table 2, results from another two sets of experiments are presented, in which the available memory on some types of PEs are reduced by a certain amount. In *case1*, we reduce 20% of the memory for PEs with highest demand factor and 10% of memory for PEs with second highest demand factor. In the second case, 30% and 15% memory are removed, respectively. As can be seen, the number of tasks allocated using the task-level algorithm decreases by 14% in *case1*, whereas the performance of system-level algorithm decreases by only 2%. For *case2*, performance drop of 19% and 8% are observed by task-level and system-level algorithms, respectively. Clearly, the task-level algorithm is very sensitive to reduction of highly demanded resources, since the side effects of suboptimal usage of those resources become more severe. In the second case, the performance of system-level algorithm is already 29.5% better than that of the task-level algorithm. These results clearly verify that the proposed approach is capable of self-adaptation to platform changes. Also, since the candidate task sets are chosen randomly in each experiment round, adaptability to changes in the applications is already justified.

Another set of experiments are carried out to evaluate the performance of difference clustering approaches. Four algorithms are compared: *No-Clustering*, *Clustering-20*, *Clustering-50* and *Adaptive*. The *Clustering-20* and *Clustering-50* are two instances of the CBP algorithm presented in [4], which contracts 20% and 50% of total edges in each TG. In each experimental rounds, we fix the task-set and amplify the communication load by dividing the NoC data rate by a constant factor. Figure 4 summarizes the results with the x axis being the scaling factor. As can be seen, when the network load is low, *Clustering-20* and *Clustering-50* degrade the performance compared with *No-Clustering*. The reason is that bandwidth is not the current resource bottleneck and clustering introduces suboptimal usage of processor

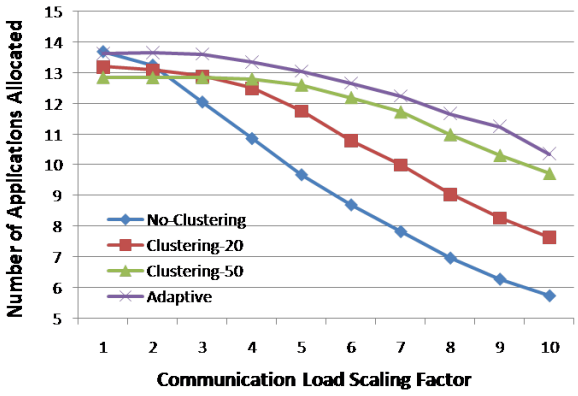

Fig. 4. Performance Comparison of Clustering Approaches

and memory resources. Nevertheless, the performance of *No-Clustering* drops rapidly as the network load increases. With a scaling factor larger than 2, *Clustering-20* already outperforms *No-Clustering* approach. When the scaling factor is between 2 and 3, the less aggressive *Clustering-20* approach is more preferable. With a large scaling factor (greater than 3), *Clustering-50* clearly shows its advantage. Concerning the proposed adaptive approach, the strength of clustering changes dynamically at runtime. With low network load and correspondingly low bandwidth usage, the threshold in equation 7 becomes relatively large, resulting in a less aggressive clustering method. Alternatively, the strength of clustering increases upon high bandwidth utilization. The results from figure 4 clearly show that the proposed algorithm features good self-adaptability and achieves best results in most cases.

5. RELATED WORK

Since task mapping/scheduling is one of the most important step during design with multiprocessor systems, much work has been published on this topic. Traditionally, designers rely more on static design time exploration to find the desired allocation scheme. Prominent tools that can handle this job include DOL[8] and SymTA/S [13]. Sophisticated optimization algorithms can be used since everything is computed offline, e.g. evolutionary algorithm are used in [8, 13], ILP is used in [1]. A prerequisite of using static optimization is that the applications must be fixed and entirely known, which limits the design flexibility with MPSoCs.

Runtime dynamic task allocation methodologies are studied more recently. One of the most viable alternatives to our solution is semi-static approach [14, 15, 16]. The basic idea of these approaches is to divide the problem into two phases, namely the offline phase and online phase. In the offline part, thorough analysis of the application set is done to identify all possible co-existing combinations of tasks (the use cases). Then, a separate mapping and schedule is computed for each use case and stored in the system. In the online phase, the resource manager looks up and adopts the pre-calculated configurations upon activation of a task. The major advantage of this strategy is that offline optimization algorithms can be used, since the execution time is not critical. However, despite the overhead for storing configuration data, this strategy has several limitations. First, the scalability is bad, since the number of use-cases can increase exponentially with the number of tasks. Second, the flexibility is limited, because 1) the application set needs to be completely known at design time; 2) a major re-calculation and system update are needed upon a change in the system, e.g. when a new application is added. Last but not least, since the same task may be mapped to different PEs in different use cases, task migration is needed during reconfiguration, which makes it very difficult to guarantee continuous QoS of the task. Our approach is more scalable and flexible compared to the semi-static approaches, since only a low-complexity application profile analysis phase needs to be repeated for different systems.

Our work can also be compared with purely dynamic approaches such as [4, 17, 5, 6]. A key improvement of the proposed approach is that, in contrast to all existing work, the task allocator considers the system-level resource management instead of optimizing the resource consumption of a single task. Another important advantage of the proposed task allocation flow is runtime self-adaptability.

6. CONCLUSION

This paper presented a runtime task allocation flow targeting the GENESYS MPSoC platform, characterized by heterogeneous processing elements connected via a global time-triggered NoC. In the existing prototype implementation of GENESYS, only static task allocation is supported [18]. We identified two important requirements in the task allocator to enable efficient dynamic task allocation, which has several advantages as discussed previously. First, since MPSoCs are typically used for co-hosting multiple tasks with unknown activation sequences, optimal resource management should be done in the system scope with future tasks taking into consideration. The proposed task allocator incorporates a global analysis step to gather the resource demand profile of all potential applications, allowing for better allocation schemes from the system point of view. Second, the GENESYS architecture is designed for cross-domain applications. Hence, adaptability to different task-sets should be supported. This is achieved by application profile analysis together with an adaptive actor clustering algorithm. We evaluated the approach using a standard benchmark set and observed significant performance gain (up to 29.5%) compared with traditional task allocators.

Our work could be extended in multiple directions. One interesting issue is to consider other important optimization

criterion, e.g. energy consumption. Moreover, we want to consider task systems with mixed criticality, e.g. to develop algorithms that guarantee 100% success rate for critical applications. Real world experiments are also planed after the GENESYS MPSoC development finishes.

Acknowledgment

This work has been supported in part by the European research project ACROSS under the Grant Agreement ARTEMIS-2009-1-100208.

7. REFERENCES

[1] Michel Goraczko, Jie Liu, Dimitrios Lymberopoulos, Slobodan Matic, Bodhi Priyantha, and Feng Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *DAC '08: Proceedings of the 45th annual Design Automation Conference*, 2008.

[2] Dongkun Shin and Jihong Kim, "Power-aware communication optimization for networks-on-chips with voltage scalable links," in *International Conference on Hardware - Software Codesign and System Synthesis (CODES+ISSS)*, 2004.

[3] Ruibin Xu, Rami G. Melhem, and Daniel Mossé, "Energy-aware scheduling for streaming applications on chip multiprocessors," in *IEEE Real-Time Systems Symposium (RTSS)*, 2007.

[4] Orlando Moreira, Jan-David Mol, Marco Bekooij, and Jef van Meerbergen, "Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix," in *11th IEEE Real Time on Embedded Technology and Applications Symposium (RTAS)*, Washington, DC, USA, 2005.

[5] Philip K. F. Hölzenspies, Johann Hurink, Jan Kuper, and Gerard J. M. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multiprocessor system-on-chip (MPSOC)," in *Design Automation and Test in Europe (DATE)*, 2008.

[6] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun, "Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip," in *IEEE/ACM/IFIP Workshop on Embedded Systems for Real Time Multimedia (ESTMED)*, Washington, DC, USA, 2006.

[7] R. Obermaisser and H. Kopetz, "Genesys book: A candidate for an artemis cross-domain reference architecture for embedded systems.," in *Publisher SVH. ISBN 3838110404*, 2009.

[8] L. Thiele, I. Bacivarov, W. Haid, and Kai Huang, "Mapping applications to tiled multiprocessor embedded systems," in *Application of Concurrency to System Design (ACSD)*, 2007.

[9] Ayse Kivilcim Coskun, Tajana Šimunic Rosing, Keith A. Whisnant, and Kenny C. Gross, "Static and dynamic temperature-aware scheduling for multiprocessor socs," *IEEE Trans. Very Large Scale Integr. Syst.*, 2008.

[10] M. Joseph and P. Pandya, "Finding response times in a real-time system," *BCS Computer Journal*, Oct 1986.

[11] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," *IEEE Real-Time Systems Symposium*, 1990.

[12] http://www.kasahara.elec.waseda.ac.jp/schedule/index.html, "Standard Task Graph Set," .

[13] Rafik Henia, Arne Hamann, Marek Jersak, Razvan Racu, Kai Richter, and Rolf Ernst, "System level performance analysis - the symta/s approach," in *IEE Proceedings Computers and Digital Techniques*, 2005.

[14] Luca Benini, Davide Bertozzi, and Michela Milano, "Resource management policy handling multiple use-cases in mpsoc platforms using constraint programming," in *24th International Conference on Logic Programming (ICLP)*, 2008.

[15] Chengmo Yang and Alex Orailoglu, "Towards no-cost adaptive mpsoc static schedules through exploitation of logical-to-physical core mapping latitude," in *DATE*, 2009.

[16] Andreas Schranzhofer, Jian-Jia Chen, Luca Santinelli, and Lothar Thiele, "Dynamic and adaptive allocation of applications on mpsoc platforms," in *Asia and South Pacific Design Automation Conference (ASPDAC)*, 2010.

[17] Orlando Moreira, Jan-David Mol, and Marco Bekooij, "Online resource management in a multiprocessor with a network-on-chip," in *ACM Symposium on Applied Computing*, NY, USA, 2007.

[18] Christian Paukovits, *The Time-Triggered System-on-Chip Architecture*, Ph.D. thesis, Technische Universität Wien, Institut für Technische Informatik, Dec. 2008.