A Workflow for Runtime Adaptive Task Allocation on Heterogeneous MPSoCs

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*—Modern Multiprocessor Systems-on-Chips (MPSoCs) are ideal platforms for co-hosting multiple applications, which may have very distinct resource requirements (e.g. data processing intensive or communication intensive) and may start/stop execution independently at time instants unknown at design time. In such systems, the runtime task allocator, which is responsible for assigning appropriate resources to each task, is a key component to achieve high system performance. This paper presents a new task allocation strategy in which self-adaptability is introduced. By dynamically adjusting a set of key parameters at runtime, the optimization criteria of the task allocator adapts itself according to the relative scarcity of different types of resources, so that resource bottlenecks can be effectively mitigated. Compared with traditional task allocators with fixed optimization criteria, experimental results show that our adaptive task allocator achieves significant improvement both in terms of hardware efficiency and stability.**

I. Introduction

Multiprocessor Systems-on-Chip (MPSoC) are believed to be the major solution for future embedded systems [1], [2]. In many emerging utilization scenarios, multiple applications are executed concurrently on the same platform and they may start and stop execution at time instants unknown at design time. For example, in a Software Defined Radio (SDR) system, different communication standards may be activated/deactivated. Another example would be a car infotainment system, in which the user may dynamically turn on/off certain services (e.g. MP3, DVD player, etc). Hence, a dynamically mixed set of applications are expected in MPSoC systems targeting these domains. The runtime task allocator is a key component of such systems, which is responsible for assigning adequate and appropriate resources (e.g. the processor, memory and communication channel) to each application.

Many MPSoCs, especially those with heterogeneous processing elements (PEs) such as the GENESYS architecture [1], are designed for cross-domain applications [3], which usually have very distinct resource requirements. For example, signal processing applications may have large execution times and control applications may communicate very often. Moreover, the set of potential applications to be executed on the platform may not be entirely known at design time, e.g. new software modules or add-ons can be downloaded to a consumer device. This makes static analysis and optimization very difficult. Moreover, a single task allocation strategy can not be expected to fit for all applications and domains. This paper presents a new approach for solving the problem by introducing self-adaptability to the runtime task allocation algorithm. This is achieved by dynamically adjusting a set of key parameters according to the current resource utilization. Experimental results from different application sets show that our self-adaptive scheme improves the performance significantly in terms of hardware efficiency and the stability with respect to different characteristics of the applications.

The rest of this paper is organized as follows. In section II, we review relevant existing work tackling the task allocation problem. In section III we first introduce the application and architecture models and then present the details of our adaptive runtime task allocator. Experimental results are presented in section IV. Section V concludes this paper.

II. Related Work

Existing task allocation approaches can be roughly classified into two branches, namely static (offline) and dynamic (online) approaches. Examples of static approaches include [4], [5], [6], in which task allocation is formulated as a static optimization problem and solved using standard techniques such as Integer Linear Programming and Genetic Algorithms, while SymTA/S [7] and DOL [3] frameworks are based on evolutionary algorithms. Although static algorithms enable thorough exploration of design space, they lack flexibility: the entire set of applications needs to be known and fixed at design time; and a major re-computation is necessary if the applications or platform change.

One possible improvement of pure static approaches is to use semi-static scenario-based algorithms [8], [9], [10], [11], [12]. The basic idea is to compute a separate schedule for each possibly coexisting application combination and store all configuration data in a table. When the application-set changes at runtime, the pre-defined configurations are looked up and applied. These approaches achieve good tradeoff between performance and flexibility but still come with some drawbacks. Firstly, the scalability is bad, since the number of configurations may grow exponentially with the number of total applications. Secondly, the flexibility is still limited, since 1) all potential applications must be entirely known at design time; 2) a major recomputation and update is needed when the application set changes. Thirdly, storing configuration data introduces additional overhead. Last but not least, since the

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

978-3-9810801-7-9/DATE11/©2011 EDAA

same task might be mapped to different PEs in different schedules, task migration is needed during reconfiguration, which makes it very difficult to provide constant quality of service. Kumar et al. introduce an alternative approach [13], in which the task-to-PE mapping is determined at design time and fixed during execution. At runtime, a resource manager coordinates the temporal resource assignment such that all applications meet their performance requirements. However, this approach considers only the PE resources and works only on non-preemptive systems. Also, the scalability and flexibility issues mentioned above still exist. In contrast to the previously discussed work, our approach handles an arbitrary number of tasks with much lower computational complexity. This results in high scalability at maximum flexibility.

Due to the high complexity of the problem, pure runtime strategies mostly rely on fast heuristics [14]. Moreira et al.[15] proposed a runtime resource allocator that is able to allocate multiple real-time jobs on homogeneous multiprocessor systems with a round-robin local scheduler. Compared to [15], our work considers a more generic system architecture, i.e. heterogeneous MPSoCs with any deterministic local task scheduling policy. Another key improvement of this work is the introduction of runtime self-adaptability. The key parameters of the task allocator are dynamically adapted based on the resource utilization, so that higher hardware efficiency can be achieved.

III. Adaptive Runtime Task Allocation

The objective of the task allocator can differ depending on the use cases. In this paper, we focus on maximum hardware efficiency, i.e. we want to allocate as many applications to the platform as possible while keeping the utilization of resources low. Nevertheless, the proposed task allocation flow is generic enough to be configured with other optimization goals, e.g. minimum energy. To achieve maximum hardware efficiency, one important issue is to map tasks to their **preferred** PEs where the resource usage can be reduced, e.g. map data processing tasks to DSPs instead of RISCs. Another issue is that the hardware resource utilization should be kept in a balanced way, since the amount of tasks that a PE can accommodate is always limited by the most scarce resource. The task allocator must also guarantee that no more resources are allocated than are available (e.g. processor time and network interface (NI) buffer) and no deadlines are violated.

A. Application and Architecture Models

We assume applications are represented in task graphs (TGs) (figure 1a), whose vertices represent **tasks** (computation kernels) and edges E capture data dependencies between tasks. For each task t_i^l of application l, the WCET ($c_{i,x}^l$) and memory consumption ($m_{i,x}^l$) on the PE x can be measured and annotated in the model. The **WCET** ($c_{i,x}^l$) in the entire paper refers to the stand-alone WCET of t_i^l on PE x. The Worst-Case Response Time (**WCRT**) ($w_{i,x}^l$) refers to maximum response time taking the waiting time caused by resource contention into account. Naturally, $w_{i,x}^l$ is in any case no less than $c_{i,x}^l$.

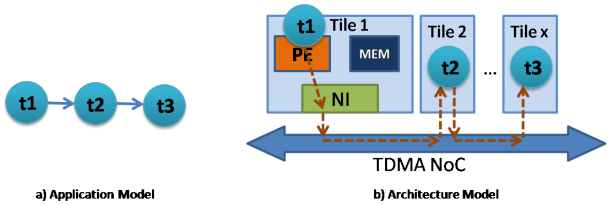

Fig. 1. Task and Architecture Models

Our target hardware platforms are tiled multiprocessor systems with heterogeneous processing elements (figure 1b). A tile in such systems typically consists of a Processing Element (PE), a local memory element (MEM), and a Network Interface (NI). As pointed out in [16], many existing multiprocessor platforms fit into this model. To achieve deterministic behavior for hard real-time applications, a predictable on-chip communication scheme is needed. As this work is based on the architecture suggested in GENESYS, we assume that the tile-to-tile communication is via TDMA-arbitrated Network-on-Chip (NoC). The time slices of the shared NoC are distributed among all tiles at design time, resulting in an equivalent local bandwidth for each tile. This bandwidth can then be assigned to different tile-to-tile communication channels. Note that a channel only consumes the local bandwidth of the sender, since the communication messages are broadcasted to all receivers. We also take the data buffer limitation at the NI into account by setting a maximum number of channels that a NI could support. Naturally, no communication channel is needed if the two communicating tasks are mapped to the same PE.

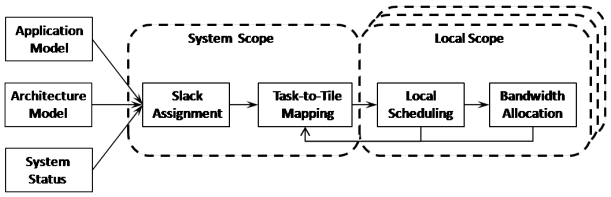

Fig. 2. Runtime Task Allocation Flow

Using a divide and conquer strategy, the task allocation flow is split into two major parts (figure 2). The first part includes *Slack Assignment* and *Task-to-tile Mapping* which are done in the system scope. Real-time applications typically have global end-to-end deadlines. The *slack assignment* step computes a *local relative deadline* (denoted using d_i^l) for each task. The end-to-end deadlines are guaranteed if all local deadlines are met. The next step maps each task to exactly one tile with enough resources. With the local deadlines, schedulability tests can be performed to make sure that the task-set on each tile is schedulable. The second part of the workflow is performed at local scope, including *Local Scheduling analysis* and *Bandwidth Allocation*. The classical theory on single processor scheduling is reused in local scheduling. The bandwidth allocation is done in the last step to enable a flexible slack borrowing strategy (section III-F).

B. Speculative Slack Assignment

The goal of slack assignment is to distribute the global deadline to individual tasks in a way that the schedulability is maximized. A classical approach is the normalized proportional deadline algorithm [17], which assigns deadlines according to the tasks' execution time and the utilization of the PEs that the tasks are mapped to. This approach can not be directly used in heterogeneous MPSoCs, since the execution time may differ for the same task on different PEs. Moreover, the task-to-PE mapping is not yet known, so we need to rely on the estimated execution time. In [18], the author evaluates three estimation approaches, namely the optimistic, pessimistic and average strategy, which use respectively the minimum execution time on all possible PEs, the maximum execution time on all possible PEs and the average execution time as the estimation. However, task allocators may have different optimization goals and a fixed estimation scheme is not expected to be optimal in all cases [18]. Another drawback of these approaches is that the run time system's utilization information is not considered, which can actually give a good hint where a task shall be mapped to. Hence, this paper proposes a speculative approach, which tries to predict the task mapping in advance and uses the execution time on predicted PEs as a basis for the slack assignment.

Naturally, the best prediction is actually to directly call the task-to-tile mapping algorithm used in later steps. However the local schedulability test requires the knowledge of the local deadlines, which is not decided at this point. In this case, we implement a reduced version of the task-to-tile mapping algorithm in the slack assignment step, which only checks resource availability. A prediction error occurs if the task turns out to be non-schedulable on the predicted PE.

Let's assume that a task t_i^l is predicted to be mapped to PE_x. We calculate the average WCRT-to-WCET ratio R_x of all tasks mapped to PE_x. With R_x, we have a computational inexpensive estimator to take into account the current load on PE_x, since it reflects how largely a task is influenced by other tasks on the same PE. Then the *speculative response time* v_i^l of t_i^l is computed as:

$$v_i^l = c_{i,x}^l * R_x \tag{1}$$

This step is done for all tasks of the application, and the slack time is distributed proportionally to the speculative execution time. In this step, slack time is also reserved for the Token Transfer (TT), which can also introduce significant latency. However, the exact TT time depends on the bandwidth allocated to the corresponding channel, which is not known during the slack assignment step. We estimate the TT time for a new channel using the average bandwidth allocated to all existing channels on the same tile.

To minimize the side effect of prediction errors, we introduce a *slack borrowing* strategy. The key idea is to lend the slack time allocated to TT firstly to its source task. The local deadlines computed in the slack assignment step are then the maximum time to execute the task *and* transfer the token to the destination. If the WCRT of the task is too large, the task can *borrow* slack time from TT. On the other hand, if the WCRT is relatively small, all the remaining slack time is used for TT so that the bandwidth consumption can be reduced. Hence, the overall schedulability is improved.

C. Adaptive Task Mapping

With the local deadlines, the next step is to map tasks to PEs. The proposed approach is a modified first-fit decreasing bin-packing heuristic, which works as follows. When a new application is activated, its tasks are sorted in decreasing order with respect to the resource usage, which is calculated as the weighted sum of all the three types of resources we consider:

$$u_{i,x}^l = \alpha \bar{c}_{i,x}^l + \beta \bar{m}_{i,x}^l + \gamma \bar{b}_{i,x}^l \tag{2}$$

The $\bar{c}_i^l$, $\bar{m}_i^l$ and $\bar{b}_i^l$ are respectively **normalized** execute time, memory consumption and bandwidth consumption of task t_i^l. The normalization is performed with respect to the total system resources. The weighting factor α, β, γ are key parameters in the algorithm, since they determine how the task allocator trade-off between the three types of resources. Starting from the biggest one, the allocator tries to map each task onto the tile of the most preferred type, which minimizes the resource consumption computed using equation 2. The task allocator checks resource availability and also the schedulability of the new task-set (using algorithms presented in section III-E). If a task fails to be mapped to the most preferred tile, other tiles with increased resource consumption are sequentially examined. If multiple tiles of the same type exist, the tile with the least resource utilization is chosen. The resource utilization of a tile is computed with respect to the local available resources and weighted by the same factors as in equation 2:

$$U_x = \alpha \frac{P_{x,used}}{P_{x,available}} + \beta \frac{M_{x,used}}{M_{x,available}} + \gamma \frac{B_{x,used}}{B_{x,available}} \tag{3}$$

The resource consumption caused by the current task to be mapped is also summed up to U_x. This procedure is continued until all tasks are mapped. If at least one task can not be mapped on any tile, the application is not accepted for execution.

Applications usually have very distinct resource requirements and it can be suboptimal ot use a set of fixed weighting factors. For example, all types of resources may be equally important in an empty system, but after mapping several control applications, the NoC bandwidth becomes the most scarce resource. This implies that reducing the bandwidth consumption becomes critical, otherwise a bandwidth bottleneck will occur, which prevents any future applications from being allocated. To solve this problem, we propose to dynamically adapt the weighting factors based on the current system utilization. Let α_0, β_0, γ_0 denote the default weighting factors. After allocating a task, the weighting factors are

updated to:

$$\alpha = \alpha_0 * \sum_x \frac{P_{x,used}}{P_{x,available}}$$
$$\beta = \beta_0 * \sum_x \frac{M_{x,used}}{M_{x,available}}$$
$$\gamma = \gamma_0 * \sum_x \frac{B_{x,used}}{B_{x,available}}$$

In the example case mentioned above, as bandwidth utilization increases faster than the other resources, the factor γ also increases faster, meaning that the importance of the bandwidth resource now becomes higher. By dynamically weighting different types of resources, the *preferred* tile of a task changes at runtime. In this way, equation 2 emphasizes the optimization of inadequate resources and equation 3 contributes to balancing the usage of inadequate resources. As shown in section IV, improved resource management and hardware efficiency are achieved using the proposed technique.

D. Adaptive Runtime Task 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. For example, a Clustering Before Packing (CBP) algorithm is proposed in [15], which attempts to contract a certain amount (e.g. 50%) of channels before submitting the application to the allocator. The basic idea is to group two communicating tasks into a single larger task, forcing them to be mapped to the same tile so that no tile-to-tile communication is needed. A major drawback of the clustering approach is that it may produce large tasks that make it more difficult to schedule and also more difficult to optimize the usage of other resources. In [15], the results show that CBP can improve the performance in communication-intensive cases (compared with non-clustering) but can also have negative effect when the communication load is relatively low.

To address this problem, we propose an adaptive clustering approach. Instead of grouping a certain amount of tasks statically, we make the clustering decision online based on the current system status. Assume a task t_1 is already mapped to tile x and we are now going to map an communicating task t_2, whose favorite tile is y. If x and y are of the same type, clustering is taken as the natural optimum solution. When x and y are of different types, a trade-off needs to be evaluated. For that, we compute the difference between the resource consumption of t_2 on tile x and that on tile y:

$$C_{diff} = (\alpha \bar{c}_{2,x} + \beta \bar{m}_{2,x}) - (\alpha \bar{c}_{2,y} + \beta \bar{m}_{2,y}) \quad (4)$$

We also compute the bandwidth resource that can be saved from clustering: $C_{save} = \gamma \bar{b}_{1,2}$. Then, the clustering is taken if $C_{save} > C_{diff}$, i.e. when the saved resources overcome the side effect of clustering. It can be seen that the weighting factors are used to compute C_{diff} and C_{save}, which makes the algorithm self-adaptive at runtime, in way that the strength of clustering is changing dynamically according to the relative scarcity of the bandwidth resource. The performance of this algorithm is verified to be very stable under various communication loads (Section IV).

E. Local Scheduling Analysis

After computing the task-to-tile mapping, the next step is to perform local scheduling of the task system on each tile. As mentioned before, the workflow allows for reuse of single processor scheduling techniques. In principle, any deterministic scheduling policy can be used, e.g. TDMA and static-priority preemptive scheduling. For TDMA arbitration, the WCRT can be found as:

$$w_{i,j} = \left\lceil \frac{c_{i,j}}{\tau_{i,j}} \right\rceil (T - \tau_{i,j}) + c_{i,j} \quad (5)$$

Where T is the TDMA time wheel, $c_{i,j}$ is the stand-alone WCET of task t_i on p_j and $\tau_{i,j}$ is the time slot allocated for t_i. For static priority scheduling, the WCRT can be found using standard techniques [19]. We also implemented some of the techniques from [20] that take intra-task and inter-task correlations into account to achieve tighter analysis bounds.

F. Bandwidth Allocation

The final step in our workflow is to assign bandwidth to all channels such that all end-to-end deadlines are enforced with TT time taken into account. As introduced in section III-B, the slack time allocated to TT is first added to its source task to allow a flexible slack borrowing strategy. After the local scheduling analysis, the WCRT of the source task is known. Then, the remaining time for token transfer from source task t_i^l to target task t_j^l can be calculated as:

$$t_{(i,j)} = (d_i^l + d_{i,j}^l) - w_{i,x}^l$$

where d_i^l is the slack time allocated to t_i^l, d_{ij}^l denotes the slack time reserved for TT of channel (i,j) and w_i^l denotes the WCRT of source task t_i^l. In the TDMA based NoC, the local time slice for a specific tile is further divided into a set of meta time slots, which can be allocated to all outgoing channels on that tile. Let B_0 denote the equivalent bandwidth (in terms of unit token size per unit time) of each meta time slot. The bandwidth allocated to channel (i,j) is the minimum integer number of B_0 such that the token transfer can be finished within the remaining slack time:

$$b_{i,j} = \left\lceil \frac{S_{i,j}}{t_{(i,j)} B_0} \right\rceil * B_0$$

where $S_{i,j}$ is the token size of channel (i,j). If the allocated bandwidth exceeds available one, the task is considered non-schedulable on the tile and a feedback is generated for the task allocator to examine other possible tiles.

IV. Experimental Results

To evaluate the performance of the approach, we generate 4 sets of syntactic task graphs (TGs) [21], each containing 100 graphs with 15 ± 5 tasks. Among the 4 sets, the TGs in $set1$ are PE-usage intensive, those in $set2$ are memory consumption

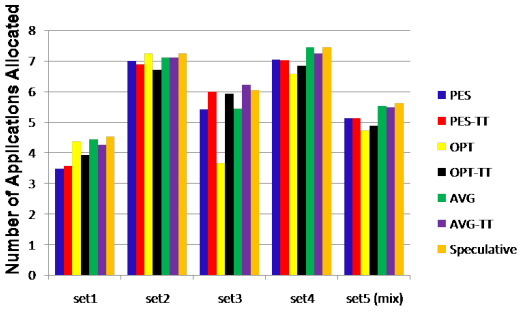

Fig. 3. Comparison of Slack Assignment Approaches

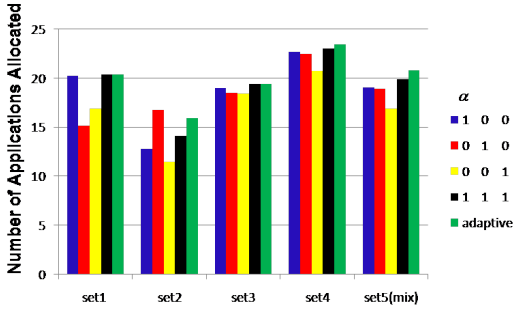

Fig. 4. Performance Comparison of Weighting Functions

	PES	PES-TT	OPT	OPT-TT	AVG	AVG-TT	Speculative
Average (%)	89.3	91.1	85.5	90.7	96.4	97.3	99.4
Std. Dev.	1.48	1.42	1.52	1.25	1.26	1.24	1.20

TABLE I
AVERAGE/STANDARD DEVIATION OF SLACK ASSIGNMENT APPROACHES

	(1,0,0)	(0,1,0)	(0,0,1)	(1,1,1)	adaptive
Average (%)	92.5	91.3	83.2	95.6	99.1
Std. Dev.	3.64	2.73	3.41	3.25	2.71

TABLE II
AVERAGE/STANDARD DEVIATION OF TASK MAPPING APPROACHES

intensive, those in $set3$ are communication intensive and the TGs in $set4$ have balanced requirements on all resources. A MPSoC consisting of 9 tiles of 3 types is used as the architecture model. On each tile, a Deadline Monotonic (DM) scheduler is used. The performance metric we considered is the number of graphs that can be mapped to the architecture model. Starting from an empty system, we randomly pick a TG and try to map it on the architecture model. This procedure ends when the first allocation failure occurs. We run such experiments for 1000 rounds and take the average results.

The first set of experiments focuses on the performance of several slack assignment schemes. The first three algorithms to be compared are the pessimistic (labeled *PES*), optimistic (*OPT*) and average (*AVG*) estimation approaches discussed in section III-B. In these three schemes, no slack time is reserved for the token transfer. The *PES-TT*, *OPT-TT* and *AVG-TT* approaches extend the original algorithms by considering the token transfer time. To make the comparison fair, the same task-to-tile mapping scheme is used in all approaches and runtime clustering is turned off. During the experiments, we observe that the performance of the slack assignment algorithms is close to each other. This is because the deadline constraint is relatively loose and the system bottleneck is resource availability. We proceed with increasing the workload in all TGs to come to a more difficult slack assignment problem. Figure 3 presents the results if the execution times are set to twice the original values. As it can be seen, the proposed speculative approach achieves the best results in the majority of scenarios. Table I summarizes the normalized performance averaging over results of all 5 TG-sets. The normalization is done with respect to the best result for each set using any of the slack assignment algorithms. Among all algorithms, the speculative approach achieves the highest average performance(99.4%). A special case is *set3* (communication intentive), for which reserving more slack time for TT can actually improve the performance significantly. For example the *OPT-TT* outperforms *OPT* by 47.3%. For this task set, the *speculative* approach maps less TGs than *AVG-TT* since more slack time is assigned for task processing according to equation 1. In different usage scenarios, the activation sequences of applications may be totally different. Hence the stability of the algorithms becomes another critical issue. Table I also compares the standard deviation of all slack assignment algorithms. With a standard deviation of 1.20, our speculative algorithm achieves the highest stability and is more preferable for arbitrary TGs. This point is also verified by the results of *set5*, in which TGs are randomly selected from *set1* to *set4*.

In the next step, we fix the slack assignment algorithm to *speculative* and evaluate the task mapping strategies. Figure 4 shows the performance of five algorithms. In the first four, fixed weighting factors are used. The weighting factor $(1, 0, 0)$ focuses on optimizing the PE consumption. Similarly, the weighting factor $(0, 1, 0)$ and $(0, 0, 1)$ optimize respectively the memory and bandwidth consumption. The fourth weighting factor $(1, 1, 1)$ has a balanced weight and the fifth case is the proposed adaptive algorithm. As expected, the weighting factor $(0, 1, 0)$ that focuses on the memory resource performs best on *set2*. The results for *set1* and *set3* are slightly different to the expectation. The reason is that the PE usage and bandwidth usage have intrinsic correlation in our setup. When the execution time is optimized and WCRT is reduced, more slack time can be used for token transfer, resulting in reduced bandwidth consumption. For this reason, optimizing PE and bandwidth usage simultaneously achieves better results, e.g. the weighting factor $(1, 1, 1)$ performs better than $(1, 0, 0)$ and $(0, 0, 1)$ for *set3*. From the results of *set1* to *set3*, we observe that our algorithm exhibits good self-adaptability, achieving results close or equal to the best result. If *set4* and *set5* are used, which have balanced or mixed resource requirements, our adaptive algorithm outperforms all the other algorithms. We can again compare the normalized performance and the standard deviation (table II). With normalized performance 99.1% and standard deviation 2.71, the *adaptive* algorithm achieves the highest performance and stability.

In previous steps, no clustering is considered during task mapping. Now, we introduce runtime clustering to

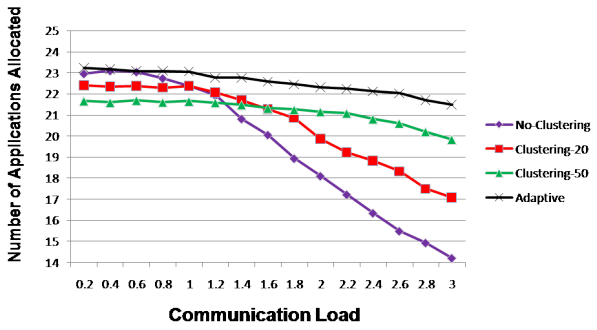

Fig. 5. Evaluation of Clustering Approaches

the task allocator so that the complete workflow is implemented. In figure 5, four approaches are compared: *No-Clustering*, *Clustering-20*, *Clustering-50* and Adaptive clustering. *Clustering-20* and *Clustering-50* are two instances of the CBP approach presented originally in [15], which tries to eliminate 20% and 50% of the communication channels. We use *set4* as the input and gradually increase the communication load by dividing the network speed by a constant factor. Figure 5 summarizes the results with different scaling factors. Comparing the first three approaches, we see that no one performs the best in all cases. When the network load is low (scaling factor less than 0.8) *No-Clustering* is actually preferable. With medium scaling factor (between 1 and 1.6) the *Clustering-20* approach is the best, since the non-aggressive clustering introduces a good tradeoff between bandwidth and other resources. When the communication load is high (factor greater than 1.6), aggressive Clustering-50 can enable allocating more TGs. The proposed adaptive algorithm can solve the problem by changing the strength of clustering at runtime. With low communication load, the bandwidth usage increases slowly and the $save$ factor becomes relatively small, resulting in a less aggressive clustering method. For high communication load, this is vice verse. The results in figure 5 clearly show that the proposed clustering approach shows very good adaptability and achieves the best results for all considered scenarios.

We measured the execution time of the algorithm on a workstation with 3GHz Intel Processor and 4GB memory. With an average allocation time of 6.6ms for one TG, the algorithm is considered to be online capable.

V. Conclusion and Future Work

This paper presents a runtime task allocator that can map and schedule multiple hard real-time tasks on MPSoC platforms. To enable cross-domain application of the MPSoC, we introduce runtime self-adaptability to the task allocation algorithm by dynamically weighting different resources based on their scarcity. We also proposed an adaptive clustering approach for efficient reduction of the communication load. Experimental results verify that the task allocator achieves more balanced and efficient use of the hardware resources. Moreover,the proposed approach works more stable than traditional task allocators with fixed configurations, making it a better strategy for mapping cross-domain real-time applications. The next step will be the investigation of adaptive algorithms for task systems with mixed criticality. Within the scope of our current research, a GENESYS MPSoC will be developed, on which real world experiments with the proposed task allocation flow can be conducted.

Acknowledgment

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

References

[1] W. Wolf, "The future of multiprocessor systems-on-chips." in *DAC*, 2004.
[2] J. Ceng, J. Castrillón, W. Sheng, H. Scharwächter, R. Leupers, G. Ascheid, H. Meyr, T. Isshiki, and H. Kunieda, "MAPS: an integrated framework for MPSoC application parallelization," in *DAC*, 2008.
[3] L. Thiele, I. Bacivarov, W. Haid, and K. Huang, "Mapping applications to tiled multiprocessor embedded systems," in *ACSD*, July 2007.
[4] M. Goraczko, J. Liu, D. Lymberopoulos, S. Matic, B. Priyantha, and F. Zhao, "Energy-optimal software partitioning in heterogeneous multiprocessor embedded systems," in *DAC*, 2008.
[5] D. Shin and J. Kim, "Power-aware communication optimization for networks-on-chips with voltage scalable links," in *CODES+ISSS*, 2004.
[6] R. Xu, R. G. Melhem, and D. Mossé, "Energy-aware scheduling for streaming applications on chip multiprocessors," in *RTSS*, 2007.
[7] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the symta/s approach," in *IEE Proceedings Computers and Digital Techniques*, 2005.
[8] M. T. J. Strik, A. H. Timmer, J. L. van Meerbergen, and G.-J. van Rootselaar, "Heterogeneous multiprocessor for the management of real-time video and graphics streams," in *IEEE journal of solid-state circuits*, Nov 2000.
[9] L. Benini, D. Bertozzi, and M. Milano, "Resource management policy handling multiple use-cases in mpsoc platforms using constraint programming," in *ICLP*, 2008.
[10] C. Yang and A. Orailoglu, "Towards no-cost adaptive mpsoc static schedules through exploitation of logical-to-physical core mapping latitude," in *DATE*, 2009.
[11] A. Schranzhofer, J.-J. Chen, L. Santinelli, and L. Thiele, "Dynamic and adaptive allocation of applications on mpsoc platforms," in *ASPDAC*, 2010.
[12] G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria, "An industrial design space exploration framework for supporting run-time resource management on multi-core systems," in *DATE*, 2010.
[13] A. Kumar, B. Mesman, B. Theelen, H. Corporaal, and H. Yajun, "Resource manager for non-preemptive heterogeneous multiprocessor system-on-chip," in *ESTMED*, Washington, DC, USA, 2006.
[14] P. K. F. Hölzenspies, J. Hurink, J. Kuper, and G. J. M. Smit, "Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (MPSOC)," in *DATE*, 2008.
[15] O. Moreira, J.-D. Mol, M. Bekooij, and J. v. Meerbergen, "Multiprocessor resource allocation for hard-real-time streaming with a dynamic job-mix," in *RTAS*, Washington, DC, USA, 2005.
[16] S. Stuijk, T. Basten, M. Geilen, and H. Corporaal, "Multiprocessor resource allocation for throughput-constrained synchronous dataflow graphs," in *DAC*, 2007.
[17] J. Sun, "Fixed priority scheduling of end-to-end periodic task," *Ph.D. thesis, University of Illinois at Urbana-Champaign*, 1997.
[18] J. Jonsson, "A robust adaptive metric for deadline assignment in heterogeneous distributed real-time systems," *Parallel Processing Symposium, International*.
[19] J. P. Lehoczky, "Fixed priority scheduling of periodic task sets with arbitrary deadlines," *RTSS*, 1990.
[20] R. Henia, A. Hamann, M. Jersak, R. Racu, K. Richter, and R. Ernst, "System level performance analysis - the SymTA/S approach," in *IEE Proceedings Computers and Digital Techniques*, March 2005.
[21] http://ziyang.eecs.umich.edu/ dickrp/tgff/, "TGFF."