A Scenario-based Run-time Task Mapping Algorithm for MPSoCs

Wei Quan[†,‡] Andy D. Pimentel[†]

[†]Informatics Institute
University of Amsterdam
The Netherlands
{w.quan,a.d.pimentel}@uva.nl

[‡]School of Computer Science
National University of Defense Technology
Hunan, China
quanwei02@gmail.com

ABSTRACT

The application workloads in modern MPSoC-based embedded systems are becoming increasingly dynamic. Different applications concurrently execute and contend for resources in such systems which could cause serious changes in the intensity and nature of the workload demands over time. To cope with the dynamism of application workloads at run time and improve the efficiency of the underlying system architecture, this paper presents a novel scenario-based run-time task mapping algorithm. This algorithm combines a static mapping strategy based on workload scenarios and a dynamic mapping strategy to achieve an overall improvement of system efficiency. We evaluated our algorithm using a homogeneous MPSoC system with three real applications. From the results, we found that our algorithm achieves an 11.3% performance improvement and a 13.9% energy saving compared to running the applications without using any run-time mapping algorithm. When comparing our algorithm to three other, well-known run-time mapping algorithms, it is superior to these algorithms in terms of quality of the mappings found while also reducing the overheads compared to most of these algorithms.

Categories and Subject Descriptors

C.4 [**Performance of Systems**]: Performance Attributes

General Terms

Algorithm, Design, Performance

Keywords

Embedded systems, KPN, MPSoC, task mapping, simulation

1. INTRODUCTION

Modern embedded systems, which are more and more based on MultiProcessor System-on-Chip (MPSoC) architectures, often require supporting an increasing number of applications and standards, where multiple applications can run simultaneously. For each single application, there are typically also different execution modes (or program phases) with different requirements. For example, a video application could dynamically lower its resolution to decrease its computational demands in order to save battery. As a consequence, the behavior of application workloads executing on the embedded system can change dramatically over time. Here, one can distinguish two forms of dynamic application behavior: inter-application dynamism and intra-application dynamism. These forms of dynamism are often captured using *scenarios* [13, 8]. This means that there are two different kinds of scenarios: inter-application scenarios to describe the simultaneously running applications in the system, and intra-application scenarios that define the different execution modes for each application. The combination of these inter- and intra-application scenarios are called *workload scenarios*, and specify the application workload in terms of the different applications that are concurrently executing and the mode of each application.

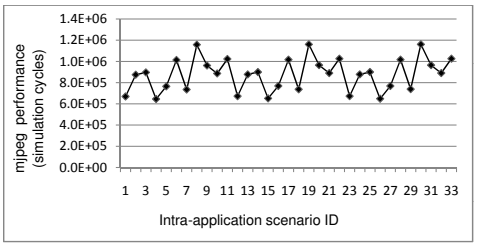

Figure 1: Intra-application scenario performance of MJPEG.

At design time of an embedded system, a designer could aim at finding the optimal mapping of application tasks to MPSoC processing resources for each inter- and intra-application scenario. However, when the number of applications and application modes increase, the total number of workload scenarios will explode exponentially. Considering, e.g., 10 applications with 5 execution modes for each application, there will be 60 million workload scenarios. If each scenario takes one second to find the optimal mapping at design time, then one would need nearly two years to obtain all the optimal mappings. Moreover, storing all these optimal mappings such that they can be used at run time by the system to remap tasks when a new scenario is detected would also be unrealistic as this would take up too much memory storage.

An approach to solve this problem is by clustering workload scenarios and only storing a single mapping per cluster of workload scenarios to facilitate run-time mapping [8]. Such clustering implies a significant space reduction needed to store the mappings. Moreover, so-called scenario-based design space exploration [17] can be deployed to efficiently find these mappings by only evalu-

DAC'13 May 29 - June 07 2013, Austin, TX, USA.

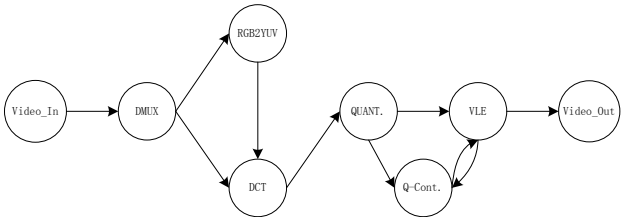

Figure 2: KPN for MJPEG application.

ating a representative subset of scenarios for each cluster. In this paper, we consider a clustering method[1] in which we find and store a single mapping for each inter-application scenario that yields, on average, best performance for all possible intra-application scenarios within the inter-application scenario. However, as we can see from the behavior of a Motion-JPEG (MJPEG) encoder application in Figure 1, using such a single mapping to represent an entire inter-application scenario shows considerable performance variations for the different intra-application scenarios that exist in this inter-application scenario. In this particular example, the inter-application scenario contains three simultaneously running multimedia applications: MJPEG, a MP3 decoder, and a Sobel filter for edge detection in images. The use of cluster-level mappings (i.e., mappings found to be good for an entire cluster of workload scenarios) can provide a run-time mapping system with enough information to quickly find an adequate mapping for a detected workload scenario but it will not immediately lead to finding the optimal system mapping for any identified workload scenario. Therefore, we propose a novel run-time Scenario-based Task Mapping algorithm (*STM*) that uses the cluster-level mapping information derived from design-time design space exploration (DSE) but, additionally, performs run-time mapping optimization by continuously monitoring the system and trying to perform (relatively small) mapping customizations to gradually further improve the system performance.

The remainder of this paper is organized as follows. Section 2 gives some prerequisites and the problem definition for this paper. Section 3 provides a detailed description of the scenario-based run-time mapping algorithm. Section 4 introduces the experimental environment and presents the results of our experiments. Section 5 discusses related work, after which Section 6 concludes the paper.

2. PREREQUISITES AND PROBLEM DEFINITION

In this section, we explain the necessary prerequisites for this work and provide a detailed problem definition.

2.1 Application Model

In this paper, we target the multimedia application domain, as was already illustrated in Figure 1. For this reason, we use the Kahn Process Network (KPN) model of computation [11] to specify application behaviour since this model of computation fits well to the streaming behaviour of multimedia applications. In a KPN, an application is described as a network of concurrent processes that are interconnected via FIFO channels. This means that an application can be represented as a directed graph $KPN = (P,F)$ where P is set of processes (tasks) p_i in the application and $f_{ij} \in F$ represents the FIFO channel between two processes p_i and p_j. Figure 2 shows the KPN of the MJPEG application.

[1]We note, however, that other clustering methods would also be possible and that our run-time mapping algorithm is independent on the clustering method used.

2.2 Architecture Model

In this work, we restrict ourselves to homogeneous MPSoC target architectures[2]. An architecture can be modeled as a graph $MPSoC = (PE,C)$, where PE is the set of processing elements used in the architecture and C is a multiset of pairs $c_{ij} = (pe_i, pe_j) \in PE \times PE$ representing a communication channel (like Bus, NOC, etc.) between processors pe_i and pe_j. Combining the definition of application and architecture models, the computation cost of task (process) p_i on processing element pe_j is expressed as T_i^j and the communication cost between tasks p_i and p_j on channel c_{xy} is $C_{ij}^{c_{xy}}$.

2.3 Task Mapping

The task mapping defines the corresponding relationship between the tasks in a KPN application and the underlying architecture resources. For a single application, given the KPN of this application and a target $MPSoC$, a correct mapping is a pair of unique assignments $(\mu : P \rightarrow PE,\ \eta : F \rightarrow C)$ such that it satisfies $\forall f \in F, src(\eta(f)) = \mu(src(f)) \wedge dst(\eta(f)) = \mu(dst(f))$. In the case of a multi-application workload, the state of simultaneously running applications that are distinguished as inter- and intra-application scenarios should be considered in the task mapping. Let $A = \{app_0, app_1, ..., app_m\}$ be the set of all applications that can run on the system, and $M^i = \{md_0^i, md_1^i, ..., md_n^i\}$ be the set of possible execution modes for $app_i \in A$. Then, $SE = \{se_0, se_1, ..., se_{n_{inter}}\}$, with $se_i = \{app_0 = 0/1, ..., app_m = 0/1\}$ and $app_i \in A$, is the set of all inter-application scenarios. And $sa_j^i = \{app_0 = md_{j_0}^0, ..., app_m = md_{j_m}^m\}$, with $app_i \in A \wedge app_i = 1 \in se_i$ and $md_{j_x}^i \in M^i$, represents the j-th intra-application scenario in inter-application scenario $se_i \in SE$. The set of all workload scenarios can then be defined as the disjoint union $S = \sqcup_{i \in SE} SA^i$, with $SA^i = \{sa_1^i, sa_2^i, ..., sa_{n_{intra}^i}^i\}$.

As already explained in the previous section, we propose to perform the run-time mapping of applications in two stages. In the first stage, which is performed at design time, we cluster workload scenarios (similar to [8]) and perform DSE for each of these scenario clusters to find a mapping that shows the best average performance for that particular cluster. More specifically, in this paper, we consider each $se_i \in SE$ as a different cluster of scenarios (i.e., we cluster all intra-application scenarios of an inter-application scenario). The mappings derived from design-time DSE are stored so they can be used by the run-time mapping algorithm to re-map applications when a workload scenario is detected that belongs to a different scenario cluster. Since these statically determined mappings may not be optimal for the current active intra-application scenario, the second stage of the run-time mapping algorithm tries to perform (relatively small) mapping customizations to gradually further improve the system performance. In our goal to optimize mappings, we recognize two kinds of objectives: system-level objectives and application-dependent objectives. System-level objectives, denoted as $O_\alpha = \{O_{\alpha 0}, O_{\alpha 1}, ...\}$, define the system-wide metrics such as system energy consumption, total system execution time, etc. Application-dependent objectives, denoted as $O_\beta = \{O_{\beta 0}, O_{\beta 1}, ...\}$, are mainly used to define the performance requirements of each separate application like throughput, latency, etc. As will be explained in the next section, the first stage of our run-time mapping approach uses system-level objectives to find mappings per scenario cluster. Here, we use system energy consumption and total workload scenario execution time as metrics: $E_{s_i}, s_i \in S$ represents the system energy consumption of workload scenario s_i

[2]In subsequent work, we will show how scenario-based run-time mapping can also be applied to heterogeneous MPSoCs.

and $X_{s_i}, s_i \in S$ is the execution time of scenario s_i. For the second stage, during which the mapping is gradually optimized, we apply application-specific objectives – in our case throughput requirements for each application – for the optimization process. However, to measure the results of the run-time optimization process, we also use the system-level metrics E_{s_i} and X_{s_i}.

Under these definitions and given the $KPN = (P,F)$ for each application and an $MPSoC = (PE,C)$, our goal is to continuously customize the mapping at run time such that the system-level and/or application-specific objectives under every workload scenario $s_i \in S$ are satisfied.

3. SCENARIO-BASED TASK MAPPING

The STM algorithm, which is outlined in Algorithm 1, can be divided into a static part and a dynamic part. The static part is used to capture application dynamism at the granularity of inter-application scenarios. For each inter-application scenario $se_i \in SE$, we have determined – using design-time DSE – a mapping that on average performs best for all intra-application scenarios SA^i of se_i. That is, for each se_i we search for a mapping by solving the following multi-objective optimisation problem:

$$min[\sum_{sa^i_j \in SA^i} E_{sa_j}, \sum_{sa^i_j \in SA^i} X_{sa_j}]. \quad (1)$$

To this end, we have deployed the scenario-based DSE approach presented in [17], which is based on the well-known NSGA-II genetic algorithm and allows for effectively pruning the design space by only evaluating a representative subset of intra-application scenarios of SA^i for each $se_i \in SE$. As this design-time DSE stage is not the main focus of this paper, we refer the interested reader to [17] for further details. The mappings derived from this design-time DSE are used by the STM algorithm as shown in lines 1-3 of Algorithm 1. When the system detects the execution of a different inter-application scenario, the static part of the STM algorithm will choose the corresponding mapping as derived from the design-time DSE stage and which has been stored in a so-called *scenario database*. Because this database only stores mappings for entire scenario clusters, its size can be controlled by choosing a proper granularity of scenario clusters (e.g., inter-application scenarios).

The dynamic part of our STM algorithm is active during the entire duration of an inter-application scenario. As explained in the previous section, it uses application-specific objectives, specified for each separate application, to continuously optimize the mapping. When the algorithm detects that an objective is unsatisfied, it will try to find a new task mapping for that particular application that missed the performance goal. If multiple applications miss their performance goal, then the STM algorithm will start optimizing the most problematic application first. The main steps of the dynamic part of the STM algorithm are described below.

3.1 Finding the Critical Task

The first step of the dynamic part of the STM algorithm is to find the so-called *critical task* for the application that missed its objective, as shown in lines 10-13 of Algorithm 1. The rationale behind this is that by remapping this critical task and possibly its neighbouring tasks (forming a bottleneck in the application), the resulting effect will be optimal. To find the critical task, the STM algorithm maintains three lists. The first list stores the task costs (TC). For every application, it contains the cost of the application's tasks, where the cost is determined by the sum of the execution and communication times of a task. These task costs are arranged in descending order in the list. The two other lists concern the storing of two other metrics for each task: the proportion of task cost in

Algorithm 1 STM algorithm

```
Input: KPN_{app_0,...,app_m}, MPSoC, O_α, O_β, μ, η
Output: New(μ,η)
list: TC, CIC, CIB, PU
pCIC = δ_c, pCIB = δ_b
1: if detectScenario() == true : //new inter-application scenario
2:   New(μ,η) = getMapping();
3:   return New(μ,η);
4: else :
5:   results[] = getStatistics();
6:   if (i = objectiveUnsatisfied(results, O_α, O_β)) != -1:
7:     taskCost(KPN_{app_i}, results, TC, CIC, CIB);
8:     peUsage(results, PU);
9:     while(1) :
10:      if (apptype = getType(KPN_{app_i})) == DATA_PARALLEL :
11:        critical = findDPCritical(KPN_{app_i}, CIC, CIB, pCIB, pCIC);
12:      else :
13:        critical = findCritical(KPN_{app_i}, CIC, CIB, pCIB, pCIC);
14:      reason = findReason(critical, CIC, CIB, pCIB, pCIC);
15:      if reason == POOR_LOCALITY :
16:        MCC[] = minCircle(KPN_{app_i}, results, critical);
17:        if GetSubstitute(PU, μ, η, MCC, apptype) == true :
18:          return New(μ,η);
19:        else failed;
20:      else if reason == LOAD_IMBALANCE :
21:        if GetSubstitute(PU, μ, η, apptype) == true :
22:          return New(μ,η);
23:        else failed;
24:      else :
25:        pCIB += ε;
26:        pCIC -= ε;
```

the total busy time of the PE (i.e., processor) onto which the task is currently mapped (CIB), and the proportion of task communication time (read and write transactions) in the task cost (CIC).

Using the TC list, the algorithm checks the task at the top of the list to find the critical task, taking the following two conditions into account: 1) whether or not the task's CIB proportion is lower than a specific threshold, defined by *pCIB*. Here, the rationale is that a high-cost task receiving only a small fraction of processor time may imply that the processor is overloaded. If the task satisfies this condition, then this task is considered as the critical task and the process of finding the critical task ends. Otherwise, the algorithm continues to check the other tasks in the TC list with lower costs until it finds the critical task. If there is no task in the application that satisfies the first condition, then the second condition will be used: 2) Whether or not the CIC proportion is higher than the threshold *pCIC*. The algorithm checks all the tasks using this second condition just like it did for the first condition. If all the tasks do not satisfy these two conditions, then the algorithm will, respectively, increase and decrease the pCIB and pCIC thresholds by ε, after which the above process is restarted again.

For data parallel applications, the process of finding the critical task has one additional test as compared to regular applications. This extra test (performed in the function *findDPCritical*) involves the check whether or not all data-parallel tasks are mapped onto different PEs. If there are data-parallel tasks that are mapped onto the same processor, then those tasks with higher task costs will be treated as critical tasks. Otherwise, the process of finding the critical task will be the same as for regular applications.

3.2 Remapping the Critical Task

After the critical task has been found, the STM algorithm tries to analyze the reason for missing the application's performance goal. In this respect, we recognize two different reasons: *poor locality* and *load imbalance*. Here, we use the process of determining the critical task to also determine the reason for not meeting the performance goal: If the CIC proportion of the critical task is higher than the value of the current pCIC threshold, then the algorithm assumes that poor locality is the reason. Otherwise it takes load imbalance as the reason for not meeting the application demands. This means that poor locality has a higher priority than load imbalance as a reason for not meeting the application demands, which is helpful to reduce the energy consumption due to communications.

Subsequently, the function *GetSubstitute* in the STM algorithm can follow different strategies to find a target PE to which the critical task will be remapped. The selection of remapping strategy depends on the reason for not meeting the application's performance demands as well as on the type of application (data parallel or not). The strategies that are used to find the substitute PE for data-parallel applications are similar to the ones for regular applications except that one additional condition is taken into account for finding the substitute PE: the substitute PE should not be a PE onto which its parallel tasks are mapped.

3.2.1 Poor locality

In the case of poor locality, the STM algorithm will try to find a better mapping for the application in question based on a *minimal cost circle (MCC)* approach. A situation that has been identified as "poor locality" is mainly due to the communication overhead between tasks. Evidently, if the communicating frequency between two tasks is very high or the communicating data size is very large, then these two tasks should preferably be mapped onto the same PE or onto two different PEs that contain a more efficient interconnect between each other. The MCC strategy aims at redistributing the critical task and its neighbouring tasks over PEs such that communication overhead is reduced while trying to avoid creating new computational bottlenecks. To this end, it first finds the minimal cost circle based on equation (2) for the critical task p_i:

$$min(Circle\_Cost(p_i)_{mn}),\ with\ 0 \leq n,m \leq |P| \qquad (2)$$

where:

$$Circle\_Cost(p_i)_{mn} = \sum_{m \leq i \leq n} T_i^z + \sum_{0 \leq i < |P|} \sum_{m \leq j \leq n} C_{ij}^{c_{xy}} \qquad (3)$$

where T_i^z denotes the execution time of task i for PE z onto which task i is currently mapped, and $C_{ij}^{c_{xy}}$ denotes the communication overhead between tasks i and j (see Section 2.2). This strategy is applicable for heterogeneous MPSoC architectures. However, in this paper, our focus is on homogeneous architectures using a shared bus interconnect. This means that each task will have a constant computational cost irrespective of the PE it is mapped on, and that communication overhead only involves internal communication within a single PE (i.e., when the communicating tasks are mapped to the same PE) or external communication between PEs via shared memory. Clearly, internal communication costs are much lower than external communication costs. Figure 3.a shows an example of an MCC (indicated by the red oval) that contains two tasks, including the critical task (red task), whereas Figure 3.b illustrates an MCC that only contains the critical task itself.

After the MCC of the critical task has been determined, the function *GetSubstitute* will choose a substitute PE for all the tasks included in the identified MCC to achieve a new mapping. For this purpose, the PU list is used, containing the processor utilisations

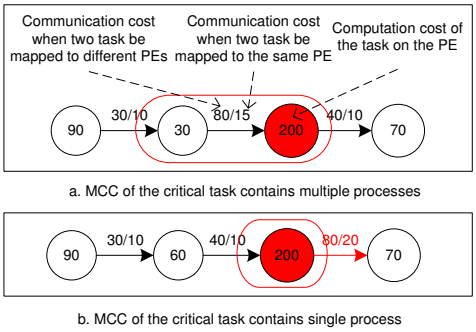

Figure 3: Examples of an MCC for a critical task (red task).

for each PE. The substitute PE is the PE with the lowest utilization in the PU list that is different from the PE onto which the critical task is currently mapped. If the MCC solely consists of the critical task itself, then the critical task will be mapped onto the PE of a neighboring task that has the heaviest communication with the critical task. This is, e.g., shown in Figure 3.b, where the critical task will be mapped onto the same PE as the task with cost 70. Moreover, the substitute PE should be different than the PE the critical task is currently mapped on. Otherwise, the algorithm fails to find a new mapping. After the substitute PE has been found, the FIFO channels between the tasks that need to be remapped are either mapped as internal communication onto the new PE (if communicating tasks are mapped onto this PE) or onto the system bus.

3.2.2 Load imbalance

In the case a load imbalance has been identified as the reason for not meeting the application demands, a load balancing strategy is used to remap the critical task. The substitute PE should satisfy the condition that it is different from the current PE of the critical task and should have the lowest processor utilization in the PU list. If such a substitute does not exist, then the algorithm cannot find a better mapping.

4. EXPERIMENTS

4.1 Experimental Framework

To evaluate the efficiency of our STM algorithm and the mappings found at run time by this algorithm, we deploy the (open-source) Sesame system-level MPSoC simulator [14]. To this end, we have extended this simulator with our run-time resource scheduling framework, as illustrated in Figure 4. Our extension includes the Scenario DataBase (SDB), a System Monitor (SM) and a run-time Resource Scheduler (RS). The SDB is used to store the mappings for each inter-application scenario as derived from design-time DSE. The SM is in charge of recording the running statistics for each active application as well as monitoring system-wide statistics. The RS uses the run-time task mapping algorithm and the statistics provided by the SM to dynamically remap application tasks when needed, as explained in the previous section.

4.2 Experimental Results

In this subsection, we present several experimental results in which we investigate various aspects of our STM algorithm and compare it to three well-known mapping algorithms: First-Fit Bin-Packing (FFBP) [7] which has been frequently adapted to do task mapping by means of modelling it as a bin-packing problem, Output-Rate Balancing (ORB) [5] and Recursive BiPartition and Refining (RBPR) [18]. We modified these algorithms to fit our mapping

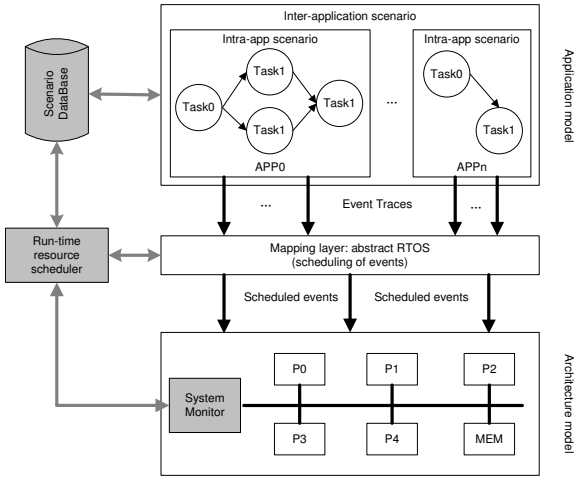

Figure 4: Extended Sesame framework.

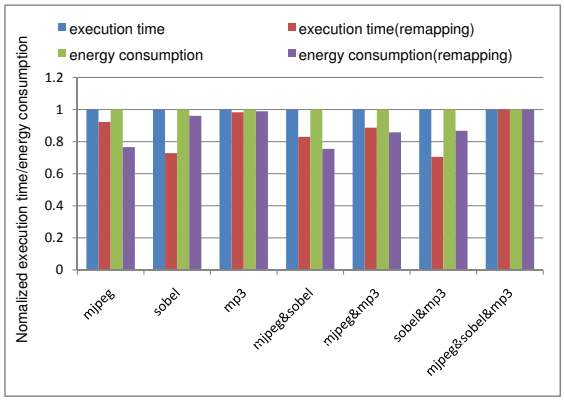

Figure 5: Performance and energy consumption of each inter-application scenario.

problem and extended them to also allow for mapping data-parallel applications by constraining the data-parallel tasks so that they have to be mapped onto different processing elements. For the FFBP algorithm, the PE with the lowest utilization is taken as the first-fit bin and the computational cost of each task in the target application is considered as the object that needs to be packed into the bins.

For our experiments, we use the three typical multi-media applications that were already introduced in Section 1: MJPEG, Sobel and MP3. The KPN of the MJPEG application contains 8 processes and 18 FIFO channels, Sobel contains 6 processes and 6 FIFO channels, and MP3 contains 27 processes and 52 FIFO channels. In the Sobel and MP3 applications, data parallelism is exploited. Moreover, MJPEG has 11 intra-application scenarios, MP3 has 3 intra-application scenarios, whereas Sobel only has 1 intra-application scenario. This results in a total of 95 different workload scenarios. At design time, we have determined the on-average best mapping for each possible inter-application scenario as explained in Section 3. With respect to the target architecture, we modeled a homogeneous MPSoC containing 5 processors, connected to a shared bus and memory. The model also includes the required components for our run-time scheduling framework.

As there are just three applications and each application contains a limited number of intra-application scenarios, we are able to exhaustively evaluate all workload scenarios. For each workload scenario, we have simulated the system using two methods: one is deploying *only the static part* of our STM algorithm to deal with the dynamism at the level of different inter-application scenarios, whereas the other one is running all the workload scenarios under a single, fixed mapping: the on-average best mapping found for the inter-application scenario in which all three applications are concurrently executing. The results of this experiment are shown in Figure 5. From this figure, we can see that the static part of our STM algorithm already yields both performance improvements and energy savings by dynamically adjusting the mapping based on the variation in inter-application scenarios. For this specific test case, the performance improvements for the different inter-application scenarios range from 1.69% to 29.49% and the energy savings range from 1.09% to 24.51%. Overall, for the execution of all 95 workload scenarios, the improvements in terms of performance and energy saving are 7.4% and 9.4%, respectively.

Figures 6.a and 6.b show the intra-application scenario execution times and energy consumption for the FFBP, ORB, RBPR and STM run-time mapping algorithms for a single inter-application scenario in which all three applications are concurrently executing. Moreover, these two graphs also contain the results when using optimal mappings (OPT) for each intra-application scenario (we derived these mappings in a design-time DSE experiment). The results in these two graphs have been ordered in a monotonically increasing fashion based on the results from the OPT mappings. Figures 6.c-e show the overall (for the entire inter-application scenario) performance, energy consumption and overhead. Here, the overhead includes the run-time calculation of new mappings as well as the migration of tasks. From Figure 6, we can see that our STM clearly performs better than the other algorithms in terms of the execution time of scenarios. For several intra-application scenarios, the STM algorithm even approaches the OPT results. With respect to energy consumption and overhead, the STM algorithm also performs well: it ranks second closely behind the ORB algorithm. The reason for a low overhead of ORB is that it only needs to migrate a few tasks in our experiment which means a very low task migration cost.

In our last experiment, we used the full STM algorithm, including the static and dynamic parts and thus combining the dynamism of inter-application as well as intra-application scenarios, to test all the 95 workload scenarios of our three applications. Our algorithm could achieve a 11.3% performance improvement and an energy saving of 13.9% compared to an approach in which we run the applications using the (static) on-average best mapping for the inter-application scenario in which all three applications are active. Comparing these results to those when only using the static part of our STM algorithm (improvements of 7.4% and 9.4%, respectively; see above), this means that the dynamic part of the STM algorithm is capable of significantly further improving the mappings.

5. RELATED RESEARCH

In recent years, much research has been performed in the area of run-time mapping for embedded systems. The authors of [6] propose a run-time mapping strategy that incorporates user behavior information in the resource allocation process. An agent based distributed application mapping approach for large MPSoCs is presented in [1]. The work of [9] proposes a run-time spatial mapping technique to map streaming applications on MPSoCs. In [3], dynamic task allocation strategies based on bin-packing algorithms for soft real-time applications are presented. A Dynamic Spiral Mapping (DSM) algorithm for mapping an application on an MPSoC arranged in a 2-D mesh topology is proposed in [2]. The authors from [4] present network congestion-aware heuristics for mapping tasks on NoC-based MPSoCs at run-time. The work of [16] uses a Smart Nearest Neighbour approach to perform run-time task mapping. In [10], a run-time task allocator is presented that

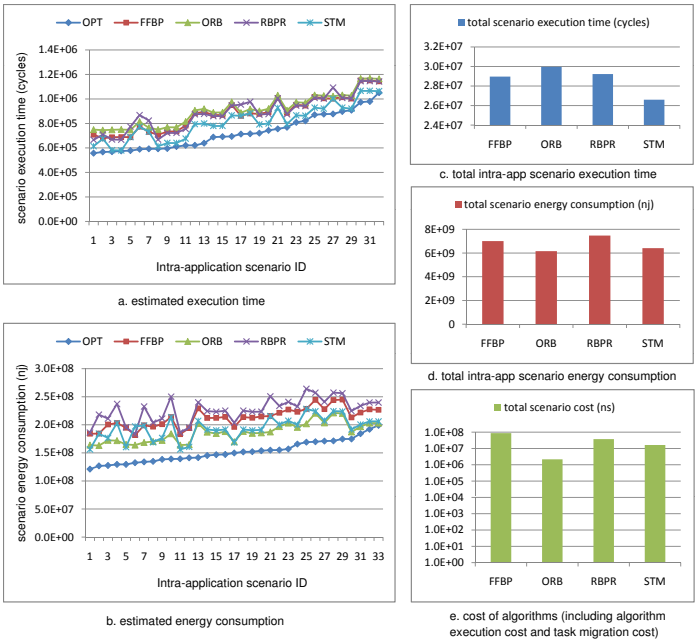

Figure 6: Comparing different run-time mapping algorithms.

uses an adaptive task allocation algorithm and adaptive clustering approach for efficient reduction of the communication load. Mariani et al. [12] proposed a run-time management framework in which Pareto-fronts with system configuration points for different applications are determined during design-time DSE, after which heuristics are used to dynamically select a proper system configuration at run time. Compared with these algorithms, our STM algorithm takes a scenario-based approach, and takes computational and communication behavior into account to make (re-)mapping decisions. Recently, Schor et al. [15] also proposed a scenario-based run-time mapping approach in which mappings derived from design-time DSE are stored for run-time mapping decisions, but they do not cluster mappings to reduce mapping storage nor do they dynamically optimize the mappings at run time.

6. CONCLUSION

We have proposed a run-time mapping algorithm for MPSoC-based embedded systems to improve their performance and energy consumption by capturing the dynamism of the application workloads executing on the system. This algorithm is based on the idea of application scenarios and consists of a design-time and run-time phase. The design-time phase produces mappings for clusters of application scenarios after which the run-time phase aims to optimize these mappings by continuously monitoring the system and trying to perform (relatively small) mapping customizations to gradually further improve the system performance. In various experiments, we have evaluated our algorithm and compared it with three other algorithms. The results show that our algorithm can yield considerable improvements as compared to just using a static mapping strategy. Comparing our algorithm with three other, well-known run-time mapping algorithms, it shows a better trade-off between the quality and the cost of the mappings found at run time.

7. REFERENCES

[1] M. A. Al Faruque, R. Krist, and J. Henkel. Adam: run-time agent-based distributed application mapping for on-chip communication. In *Proc. of DAC'08*, pages 760–765, 2008.

[2] M. Armin, K. Ahmad, and S. Samira. Dsm: A heuristic dynamic spiral mapping algorithm for network on chip. *IEICE Electronics Express*, 5(13):464–471, 2008.

[3] E. W. Brião, D. Barcelos, and F. R. Wagner. Dynamic task allocation strategies in mpsoc for soft real-time applications. In *Proc. of DATE'08*, pages 1386–1389, 2008.

[4] E. Carvalho and F. Moraes. Congestion-aware task mapping in heterogeneous MPSoCs. In *Int. Symposium on System-on-Chip*, pages 1–4, Nov. 2008.

[5] J. Castrillon, R. Leupers, and G. Ascheid. Maps: Mapping concurrent dataflow applications to heterogeneous mpsocs. *IEEE Trans.on Industrial Informatics*, PP(99):1, 2011.

[6] C.-L. Chou and R. Marculescu. User-aware dynamic task allocation in networks-on-chip. In *Proc. of DATE'08*, pages 1232–1237, 2008.

[7] E. G. Coffman, Jr., M. R. Garey, and D. S. Johnson. Approximation algorithms for bin packing: a survey. In *Approximation algorithms for NP-hard problems*, pages 46–93. PWS Publishing Co., 1997.

[8] S. V. Gheorghita, M. Palkovic, J. Hamers, A. Vandecappelle, S. Mamagkakis, T. Basten, L. Eeckhout, H. Corporaal, F. Catthoor, F. Vandeputte, and K. D. Bosschere. System-scenario-based design of dynamic embedded systems. *ACM Trans. Design Autom. Electr. Syst.*, 14(1), 2009.

[9] P. K. Hölzenspies, J. L. Hurink, J. Kuper, and G. J. Smit. Run-time spatial mapping of streaming applications to a heterogeneous multi-processor system-on-chip (mpsoc). In *Proc. of DATE'08*, pages 212–217, March 2008.

[10] J. Huang, A. Raabe, C. Buckl, and A. Knoll. A workflow for runtime adaptive task allocation on heterogeneous mpsocs. In *Proc. of DATE'11*, pages 1119–1134, 2011.

[11] G. Kahn. The semantics of a simple language for parallel programming. In *Information processing*, pages 471–475. North Holland, Amsterdam, Aug 1974.

[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 *Proc. of DATE'10*, pages 196 –201, march 2010.

[13] J. M. Paul, D. E. Thomas, and A. Bobrek. Scenario-oriented design for single-chip heterogeneous multiprocessors. *IEEE Trans. VLSI Syst.*, 14(8):868–880, 2006.

[14] A. D. Pimentel, C. Erbas, and S. Polstra. A systematic approach to exploring embedded system architectures at multiple abstraction levels. *IEEE Trans. Computers*, 55(2):99–112, 2006.

[15] L. Schor, I. Bacivarov, D. Rai, H. Yang, S.-H. Kang, and L. Thiele. Scenario-based design flow for mapping streaming applications onto on-chip many-core systems. In *Proc. of CASES'12*, pages 71–80, 2012.

[16] A. K. Singh, W. Jigang, A. Kumar, and T. Srikanthan. Run-time mapping of multiple communicating tasks on mpsoc platforms. *Procedia CS*, 1(1):1019–1026, 2010.

[17] P. van Stralen and A. D. Pimentel. Scenario-based design space exploration of mpsocs. In *Proc. of IEEE ICCD'10*, October 2010.

[18] J. Yu, J. Yao, L. Bhuyan, and J. Yang. Program mapping onto network processors by recursive bipartitioning and refining. In *Proc. of DAC'07*, pages 805 –810, june 2007.