Towards Self-adaptive MPSoC Systems with Adaptivity Throttling

Wei Quan[†,‡]

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

Andy D. Pimentel[†]

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

***Abstract*—Today's multi-processor system-on-chip (MPSoC) systems increasingly have to deal with dynamically changing application workload scenarios. To cope with such dynamic application behavior, these systems could dynamically adapt the mapping of application tasks onto the underlying system resources to improve the system's performance. However, such performance improvement comes at the cost of a system reconfiguration in which application tasks may have to be migrated between processors. This trade-off implies that reconfiguring the system is only beneficial when the performance gains outweight the reconfiguration overhead. To address this problem for MPSoCs, this paper presents a scenario-based run-time resource management framework with the ability of adaptivity throttling that uses the history of application scenario execution behavior to predict the actual benefit of a system reconfiguration to allow for explicitly deciding (at runtime) whether or not to reconfigure. Experimental results reveal that our proposed approach substantially improves the system's efficiency as compared to MPSoCs that do not provide such intelligent reconfiguration control.**

I. Introduction

Due to the ever-increasing performance demands of modern embedded applications, the use of heterogeneous MPSoC systems has become increasingly popular in the embedded systems domain. Today's MPSoC systems often require supporting an increasing number of applications and standards, where multiple applications can run simultaneously. For each single application, there may also be different execution modes (or program phases) with different requirements. 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, which are often captured using *scenarios* [7], [2]. This means that there are two different kinds of scenarios: inter-application scenarios describe the simultaneously running applications in the system, while intra-application scenarios 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. The change of workload scenarios over time on a certain MPSoC system typically depends on environmental behaviour, such as initiated by a user.

The mapping of application tasks onto the underlying system resources plays a crucial role in achieving high performance in MPSoC systems. The performance of a workload scenario may vary greatly among different mappings. To enable MPSoC systems to support the application dynamism more efficiently, a state-of-the-art solution would provide a light-weight resource scheduler that allows for reconfiguring the system at run time based on pre-optimised system configurations, such as task mappings, derived at design time [6], [18], [14], [10], [9], [16]. This type of methods can be divided into two stages. The first stage is the design time preparation which determines one or multiple system configurations for each possible scenario that may appear on the target system. For example, these configurations could be different task mappings optimizing the system for e.g. performance and/or energy consumption. The second stage is the run-time stage in which a resource scheduler chooses the appropriate system configuration from the pre-optimised configurations based on the current active workload scenario on the system.

However, these dynamic MPSoC solutions typically always try to reconfigure the system when a new workload scenario has been detected and, doing so, they do not explicitly take the *reconfiguration costs* into account. These reconfiguration costs may be substantial as they include the overhead of application tasks that may need to be migrated between different processors in the MPSoC. Especially in the case where the *duration of a certain workload scenario* is not long enough (i.e., different workload scenarios appear after each other with a relatively high frequency), such overheads may easily eliminate the benefits of reconfiguring the system: the reconfiguration itself may take longer than the performance gain that is obtained after reconfiguration. Therefore, this paper presents a novel run-time management framework that tries to predict whether or not reconfiguration of the system actually is beneficial. In this paper, we refer to this ability as *adaptivity throttling*. To make a good trade-off, we need to be able to predict the potential performance benefits and costs of a specific system reconfiguration. Here, especially predicting the performance benefits is challenging because this also entails predicting the duration of workload scenarios. For this purpose, we use an approach in which the history of application scenario execution behavior is stored to predict future scenario execution behavior. Experimental results reveal that our approach is able to clearly improve the system's efficiency as compared to MPSoCs that do not provide such

intelligent reconfiguration control.

The remainder of the paper is organized as follows. Section II discusses some prerequisites, after which Section III provides a detailed description of our approach. Section IV presents the experimental environment and the results of our experiments. Section V discusses related work, after which Section VI concludes the paper.

II. Prerequisites

As introduced in the previous section, we use the concept of scenarios to capture application dynamism. A workload scenario is defined as one combination instance of the execution mode of target applications. Here, we denote S as the set of all possible workload scenarios for the target applications. For a number of n target applications where each application has m execution modes, the total number of workload scenarios in S is $(m+1)^n - 1$. In each workload scenario $s_i \in S$, the application tasks and communications between tasks are described as a directed graph $G_i = (T, C)$ where T is the set of tasks in scenario s_i and C represents the set of communication channels between any pair of communicating tasks.

In this work, we restrict ourselves to heterogeneous MPSoC architectures with shared memory. An architecture can be modelled as a graph $MPSoC = (PE, M)$, where PE is the set of processing elements used in the architecture and M is a multiset of pairs $m_{ij} = (pe_i, pe_j) \in PE \times PE$ representing a buffered communication medium between processors pe_i and pe_j, composed of a network channel (Bus or NoC) and a buffer located in shared memory. We note, however, that our proposed approach is not limited to the architecture we assume here.

The task mapping defines the binding of the components in a workload scenario (tasks and the communication channels) to the underlying architecture resources. Given a workload scenario and a target MPSoC, a correct mapping is a pair of unique assignments ($\mu : T \to PE$, $\eta : C \to M$) such that it satisfies $\forall c \in C, src(\eta(c)) = \mu(src(c)) \wedge dst(\eta(c)) = \mu(dst(c))$. For each workload scenario $s_i \in S$, the possible task mappings are denoted as TM_i with each single mapping $tm_i^j \in TM_i$ complying with the mapping constraint. In this work, we assume that the task mapping of applications on the target system can only be changed by task migration.

Our target applications belong to the domain of streaming applications (like multimedia applications) that continuously process an incoming stream of data elements. To capture the duration of a workload scenario in this case, we use the concept of *scenario frames*. Here, we define one *scenario frame* as the time it takes for each active application within a specific workload scenario to process a single unit (frame) of data (e.g., processing a single MP3 frame, an H264 frame, etc.). This means that the frame execution time p_i^j of a workload scenario is defined as the maximum of frame execution times of all active applications within the scenario:

$$p_i^j = max(p_{app_k}^j) \tag{1}$$

where $p_{app_k}^j$ represents the frame execution time of application app_k that is active in scenario s_i under mapping tm_i^j. Consequently, the total execution duration of scenario s_i under mapping tm_i^j is calculated as $p_i^j * n_i$ where n_i is the number of scenario frames executed on the system.

When a new workload scenario has been detected, this means that one or more applications may have stopped and/or new ones have started. Here, we assume that when a new application is started, it is added to the system using a predetermined, default task mapping. Given a newly detected scenario, the complete task mapping of those applications that persist in the new scenario and any newly added applications needs to be reconsidered. Remapping of the application tasks in the newly detected workload scenario can be beneficial performance wise (i.e., every workload scenario has an optimal task mapping) but this depends on both the actual performance gain of reconfiguring the system and the reconfiguration costs. The reconfiguration costs include two parts: 1) the overhead of the resource scheduler which includes the time of finding a new mapping and making a reconfiguration decision, and 2) the task migration cost that may occur during system reconfiguration. Here, we denote the reconfiguration costs for scenario s_i to change from mapping tm_i^j to $tm_i^{j'}$ as $c_i^{jj'}$, which includes the time of finding the new mapping $tm_i^{j'}$ and making the reconfiguration decision for the new mapping. This implies that the system reconfiguration benefit B can now be expressed as:

$$B = (p_i^j - p_i^{j'}) * n_i - c_i^{jj'} \tag{2}$$

Our objective is to maximize the system performance for a sequence of workload scenarios S'. This means that we want to maximize the total system reconfiguration benefit $\sum_{s_k \in S'} b_k * B_k$, where $b_k \in \{0,1\}$ is the migration decision made by the run time scheduler and B_k is the reconfiguration benefit of workload scenario $s_k \in S'$. Obviously, the solution to this problem is $b_k = 0$ if $B_k <= 0$ and $b_k = 1$ if $B_k > 0$. Consequently, to achieve our objective, the system needs to correctly predict the system reconfiguration benefit B for each workload scenario.

III. Run-time Task Mapping

A. Design-time Preparation

To achieve the objective of run-time adaptivity for our target MPSoC system, a design-time preparation stage is needed to provide the necessary information for the run-time manager. For each possible workload scenario, the optimal mapping needs to be found at design time. To this end, we have deployed a so-called scenario-based design space exploration (DSE) approach [17]. 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 design-time DSE yields a performance optimised mapping for each workload scenario, which are stored in system memory for run-time usage.

To determine the performance gain of reconfiguring the system (remapping application tasks) versus keeping the currently active mapping in case of a newly detected workload scenario, we must know the performance of both the current mapping and the pre-optimised mapping of the newly detected scenario. In this work, we assume that these two performance numbers have also been determined at design time and are stored in system memory. This means that the frame execution time of each workload scenario under its pre-optimised

mapping (as found by design-time DSE) is stored. Moreover, we assume that we also have a look-up table (generated at design time) to determine the performance of a newly detected scenario under the old mapping of the previously active application scenario. Evidently, to avoid the relatively large memory overheads of such look-up tables, we intend to study light-weight run-time mapping performance prediction techniques in the future. Finally, we also statically store the size of data that needs to be migrated during task migration for every task in all target applications for the run-time prediction of migration costs.

B. Run-time Resource Reconfiguration

To construct the adaptive resource scheduler as introduced before, the problem that needs to be solved is to correctly predict the system reconfiguration benefit B. It consists of three parts: the performance improvement $p_i^j - p_i^{j'}$, the reconfiguration cost $c_i^{jj'}$ and the workload execution duration n_i. These three parts are unknown before the system reconfiguration. Thus, prediction models should be used to determine a reconfiguration decision based on the benefit B. In this paper, we mainly focus on the prediction of the workload scenario duration n_i.

1) Mapping Performance Prediction: As explained in the previous section, the performance of each workload scenario under the mappings derived at design time are stored on the system. In this case, the run-time scheduler does not need to dynamically predict the performance of workload scenarios which saves additional overhead, at the cost of additional design-time preparation and the system storage needed for storing the performance information.

2) Reconfiguration Cost Prediction: The reconfiguration cost of our target MPSoC consists of two parts: the overhead of the resource scheduler and the task migration cost during system reconfiguration. When a new workload scenario is detected, the system scheduler will first determine a new mapping (in our case the stored pre-optimised mapping) for this workload scenario, and then make a reconfiguration decision. The overhead of these two steps can be determined by means of measurements. However, the other part of the reconfiguration cost, the cost of task migration, should still be predicted. As mentioned before, the amount of data that needs to be migrated between processors for each task is known at design time. Subsequently, we use a simple linear analytic model for the migration cost:

$$CMig = (\sum_{t_i \in T_{mig}} ms_i)/r_{mem} \quad (3)$$

where T_{mig} is the set of migrating tasks, ms_i represents the amount of migrating data for task t_i, and r_{mem} is the memory access speed. This model is based on two assumptions: the migrating data is transferred via the MPSoC's shared memory and the resource scheduler sequentially controls task migrations.

3) Reconfiguration Decision Prediction: For a newly detected workload scenario s_i, the potential performance improvement due to system reconfiguration can be calculated by using the stored mapping performance information that was derived at design time. Combining the performance improvement

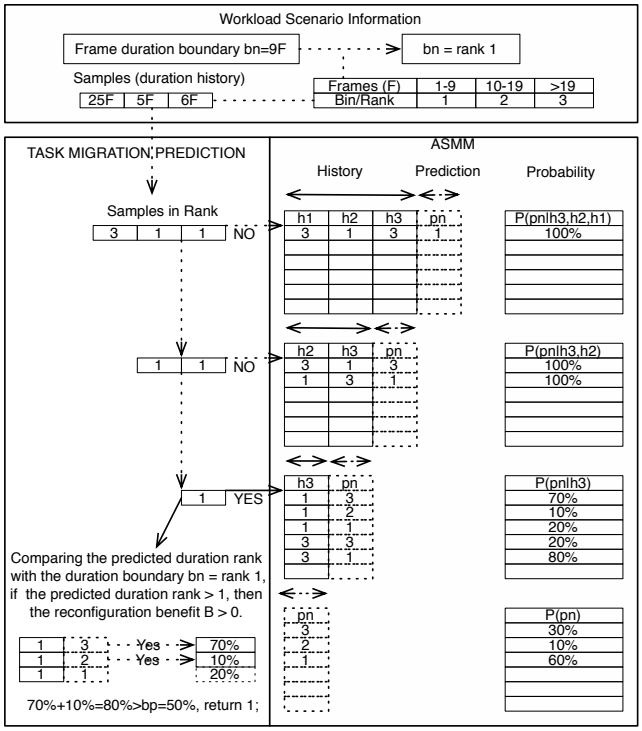

Figure 1: ASMM example with 3 history samples.

and the (predicted) system reconfiguration cost, the scheduler can determine a lower bound bn_i for the execution duration of this workload scenario:

$$bn_i = c_i^{jj'}/(p_i^j - p_i^{j'}) \quad (4)$$

According to the derivation in Section II, one can easily see that the system should be reconfigured only if the execution duration of s_i is larger than bn_i ($B > 0$). In this work, we propose an Accumulated Statistical Metric Model (ASMM), which is based on the Statistical Metric Model (SMM) [13], to predict the scenario execution duration. The SMM is a probability distribution over application patterns of varying length. It models the conditional distribution on the identity of the $i-th$ (quantized) sample given the identities of all previous (quantized) samples in a metric sequence. The difference between our ASMM and the original SMM concerns the way of how the prediction value is generated, as will be explained below.

Using our ASMM, we build a metric model based on the probability distribution of scenario execution duration for each workload scenario, which means each workload scenario has its own ASMM. When a new workload scenario is detected, the system scheduler uses the ASMM of this workload scenario to predict its duration. Figure 1 gives an example of the ASMM-based prediction of a reconfiguration decision when using 3 history samples. In our problem, the samples of scenario execution duration are measured in the number of *frames* (F). These frame numbers are quantized using a limited number of bins (or *ranks*) to reduce the complexity of our predictor, see the upper part of Figure 1 for an example. The lower right part of Figure 1 shows a simple instance of our ASMM. It

includes three kinds of tables: the execution duration history tables, duration prediction tables and tables with probabilities for all possible duration predictions. The first two types of tables store *rank numbers*. The width and depth of the history tables usually determine the prediction ability of the ASMM and should be set based on the target problem. At the end of a workload scenario, all tables of the workload scenario are updated according to its actual execution duration. We refer the interested reader to [13] for further details about (A)SMMs.

To illustrate the ASMM-based reconfiguration decision, please consider the lower left part of Figure 1. The input of our ASMM is the (quantized) execution duration sample history (top of Figure 1) of the detected scenario and the duration bound bn_i. According to the detected workload scenario, the corresponding ASMM will be used to determine the reconfiguration decision. In our example, the ASMM first checks the history table with 3 history samples to see if there is a pattern match regarding the scenario's duration history. If there is no match, like in the case of our example, the ASMM will continue to search the history tables with a smaller width of history samples (i.e., using a shorter history). This process continues until there is a history pattern match or it ends up at the direct duration prediction without any execution duration history (the table at the bottom of the ASMM in Figure 1). In both cases, the duration bound bn_i is compared with all the possible duration prediction values and for those prediction values bigger than bn_i their probabilities will be accumulated: hence the name *Accumulated* SMM. This accumulated probability represents the chance of $B > 0$ if the system is reconfigured for this workload scenario. Only if the accumulated probability is larger than the probability bound bp (set by the designer), the ASMM will return a positive reconfiguration decision. In our example, the probability bound bp is set to 50%.

IV. Experiments

A. Experimental Setup

To evaluate the efficiency of our proposed run-time adaptive resource scheduler, we deploy the system-level MPSoC simulation framework from the work of [11] which is based on the open source Sesame simulator [8]. This Sesame-based modeling and simulation environment facilitates efficient performance analysis of embedded (media) system architectures. The most important feature of this simulator for this work is its ability to support the simulation of run-time system reconfiguration of MPSoC systems. We have extended this simulator with our run-time resource scheduling framework such that it includes a Scenario DataBase (SDB), a Run-time System Monitor (RSM) and a Run-time Resource Scheduler (RRS). The SDB is used to store the task mappings as derived from design-time DSE as well as the mapping performance information of each workload scenario and the migrating data size of each task. The RSM is in charge of detecting and identifying the active workload scenario, and also collects statistics (e.g., the actual execution duration of a workload scenario) from the underlying system during the execution of a certain workload scenario. To detect and identify workload scenarios in our Sesame simulation framework, we need to instrument the source code of target applications with scenario related events such as *STARTSCENARIO* and *ENDSCENARIO*. When a processing element in a Seseme system model encounters a *STARTSCENARIO* event of an application, it will register this application with its execution mode information in the RSM to notify that a new application has started execution on the system. Similarly, for an *ENDSCENARIO* event of an application, the processing element will unregister the application in the RSM. According to the registered application information in the RSM, the active workload scenario on the target system can be identified. The RRS uses our proposed approach to do resource reconfiguration for the identified workload scenario. As the function of the RRS is performed on a CPU, we can determine the overhead of finding the pre-optimised mapping and making a migration decision (including the cost of updating the information in the scenario duration predictor) using the CPU timer. This cost is then normalized with respect to the simulator's frequency to get the final reconfiguration cost.

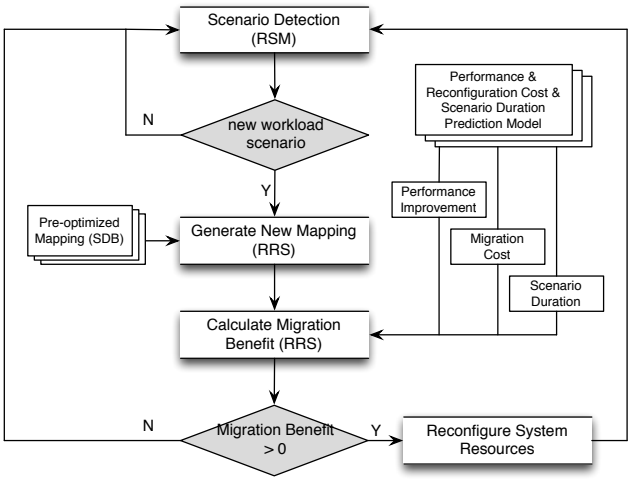

Figure 2: Workflow of run-time adaptive resource scheduler.

Figure 2 illustrates the workflow of our adaptive run-time MPSoC system. In our target MPSoC system, when the RSM detects a new workload scenario, the RRS obtains the pre-optimised mapping for the current active scenario from the SDB. Hereafter, the RRS makes a reconfiguration decision based on the reconfiguration benefit B of changing the current mapping to the new mapping. According to this decision, the RRS will either reconfigure the system based on the new mapping or continue the system's execution under the current mapping. When the current workload scenario finishes, the RRS updates the information in the corresponding ASMM based on the actual execution duration collected by the RSM.

As the actual functionality of the applications is not important in our experiments, we use five synthetic streaming applications to simplify the simulation process. Each application contains only 1 execution mode. In this case, the total number of workload scenarios is 31 ($2^5 - 1$). The number of tasks in each application ranges from 4 to 8. We assume that each task can be executed on each processor of the target MPSoC using the corresponding pre-compiled code (stored in the shared memory). The task execution time and migration data size of each task on each processor have been randomly generated and range between 10,000 and 100,000 time units (simula-

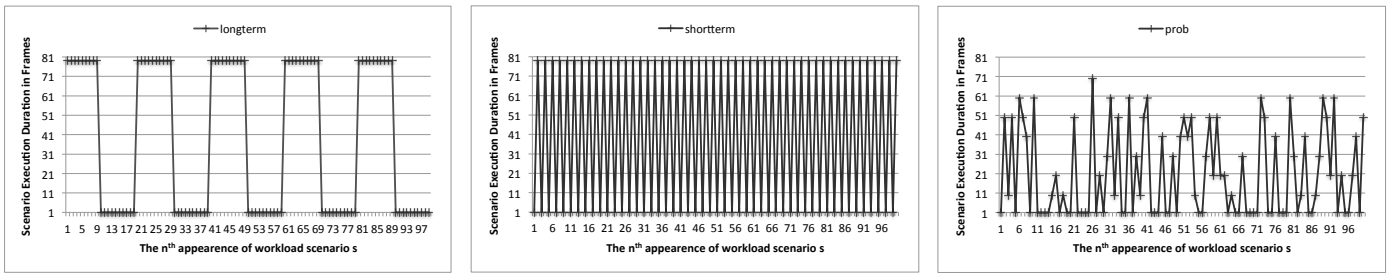

Figure 3: The scenario duration distributions used for generating workload scenarios. (longterm: the execution pattern of a scenario does not frequently change; shortterm: the execution pattern of a scenario does frequently change; prob: the execution pattern of a scenario randomly changes.)

tion cycles) and between 50K and 500K Bytes respectively. Communications between tasks range from 1,000 to 10,000 Bytes in size. Regarding the target architecture, we target a heterogeneous MPSoC containing 5 different processors with different computational characteristics, connected to a shared bus and memory.

To model dynamic application behavior over time (e.g. due to user behavior), we generate three workload scenario sequences. These sequences are generated in two steps. The first step is to choose a workload scenario from the total 31 workload scenarios considered in our experiment. Each workload scenario has the same probability to be selected. The second step is to generate the duration in *frames* of the selected workload scenario. This process iterates until a pre-defined total frame number (100,000 frames in our case) has been achieved for the scenario sequence. As the workload scenarios considered in our test case need around 40 frames on average to neutralize the reconfiguration cost, we limit the duration of each workload scenario to a value between 1 and 80 frames. Here, the reason of limiting the duration of a workload scenario in the range of 1 to 80 is that our simulated system is very sensitive when the scenario execution duration is around 40. If the execution duration of each workload scenario would be very large, then we would not need to use our proposed adaptivity throttling technique anymore and just alway reconfiguring the system when a scenario changes. In our three scenario sequences, the duration of each workload scenario and frequency of changes to this duration are generated using different distributions as shown in Figure 3. Here, the x-axis represents the *n*th appearance of one specific workload scenario and the y-axis represents the execution duration in *frames* for that particular appearance of the workload scenario. In the *longterm* distribution, the duration of a workload scenario is either long or short and does not frequently change, whereas in the *shortterm* distribution the scenario execution duration does frequently change. Like in *shortterm*, the frequency of changes in the *prob* distribution is high but the actual scenario execution duration now has been generated from the following probability distribution: 1→30%, 11→10%, 21→10%, 31→10%, 41→10%, 51→10%, 61→10%, 71→10%. That is, a workload scenario has a probability of 30% that it will be executed for only 1 frame and 10% for each of the other duration times.

With regard to the parameters of our ASMM, the maximal width and depth of the ASMM table is 4 (2-history, 1-

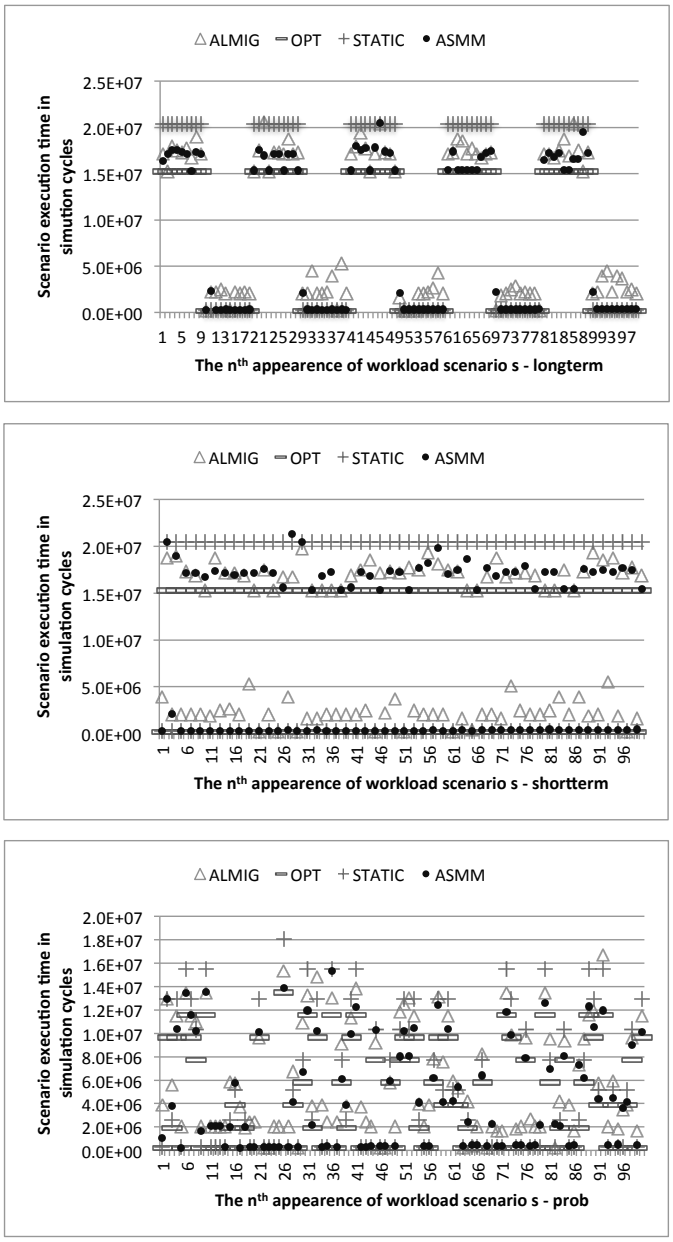

Figure 4: Scenario execution time of a selected workload scenario in the three generated workload scenario sequences under different scheduling approaches.

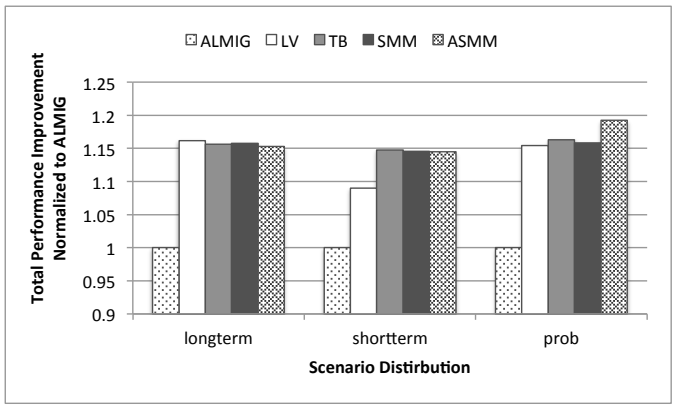

Figure 5: Performance comparison of adaptivity throttling approaches with different predictors.

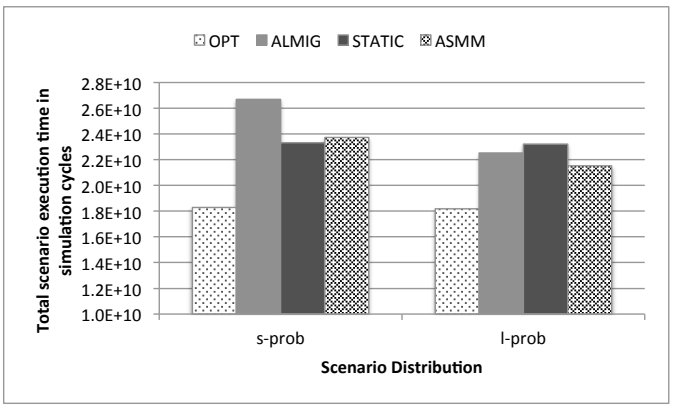

Figure 6: Performance of different resource scheduling techniques in two *prob* scenario sequences.

prediction and 1-probability) and 1024 respectively, which is large enough for our test cases. The probability bound *bp* for our ASMM is 50%. This will not lead to either a pessimistic or aggressive decision. As we limit the scenario execution duration from 1 to 80 frames, we divide the scenario execution durations into 8 bins/ranks in our ASMM, each containing a frame range of 10.

B. Experimental results

In the first experiment, we compare the scenario execution time (including the run-time reconfiguration cost) of each of the three workload scenario sequences under four resource scheduling approaches. The first approach (ALMIG) is executing each workload scenario under its pre-optimised mapping stored in the SDB. This means that the run-time scheduler will always reconfigure the system based on the pre-optimised mapping whenever a new workload scenario appears. The second approach (STATIC) is executing all the workload scenarios under a single pre-optimised mapping that has been optimised to work best – on average – for all workload scenarios. In STATIC, no system reconfiguration takes place. The third approach (ASMM) uses our ASMM-based run-time adaptive resource scheduler. Finally, as a baseline, we also compare to the ideal case (OPT) that applies ALMIG but for which all run-time reconfiguration costs have been discarded. At the beginning of each simulation, the target system is initialised by the mapping of the STATIC approach.

Figure 4 shows the results of the scenario execution time of the first 100 appearances of a single selected workload scenario in our three workload scenario sequences. Clearly, in all cases the OPT approach performs best, and STATIC performs worst in those cases where the system should have been reconfigured. We can also see that the scenario execution time is influenced by the reconfiguration cost. For example, consider the bottom parts of the three graphs, i.e., small scenario execution durations. Here, one can clearly see that ASMM consistently outperforms ALMIG since the latter is negatively affected by the reconfiguration overhead whereas our ASMM approach is not because it predicted that reconfiguration is not beneficial. For larger scenario execution durations (top parts of the three graphs), the performance of ALMIG and ASMM is similar (i.e., they both reconfigure the system).

In the second experiment, we compare the total execution time of each workload scenario sequence under our adaptive resource scheduler with different scenario duration predictors. We compare three application behaviour predictors to our ASMM approach: a Last Value predictor (LV), a Table-Based predictor (TB) [5] and the original SMM [13]. For a fair comparison, the parameters used for TB and SMM are the same for ASMM. Figure 5 shows the results of this experiment. In this figure, the total scenario execution time is normalized to the ALMIG approach. As one can see, in the *longterm* case, all predictors show a good performance as they are all able to accurately predict the execution pattern. For the *shortterm* distribution, the LV predictor shows a poorer performance improvement. This is mainly because of a high prediction error (nearly 100%) of the LV predictor in this distribution. In the *prob* test case, our ASMM predictor clearly outperforms the other three predictors. This is because the scenario execution does not have a fixed pattern anymore which induces a high prediction error for the LV and TB predictors. Compared with the SMM predictor, our ASMM uses the accumulated probability to determine a reconfiguration decision, which increases the chance of making the right decision when reconfiguration is actually needed. Among these predictors, the LV predictor needs the least system memory (only the last value) to record the execution history of a workload scenario. The SMM/ASMM predictor consumes the most system memory which will be introduced in the last part of this section.

Where the *longterm* and *shortterm* scenario sequences more or less reflect extreme cases in workload scenario behavior, the *prob* sequence possibly exhibits a more realistic view on dynamic behavior in application workloads. Therefore, we focus on the *prob* case in the third experiment. Figure 6 shows the total execution time of executing all workload scenarios in the entire scenario sequence for two instances of the *prob* scenario sequence: in the *s-prob* sequence, 70% of all scenarios has a short duration of less than 40 frames (note that our test cases need around 40 frames on average to neutralize the reconfiguration cost), whereas the *l-prob* sequence contains only 40% of scenarios with a short duration. In the *s-prob* and *l-prob* scenario sequences, the average execution duration of scenarios is approximately 30 and 60 frames, respectively. This explains why the *ALMIG* approach

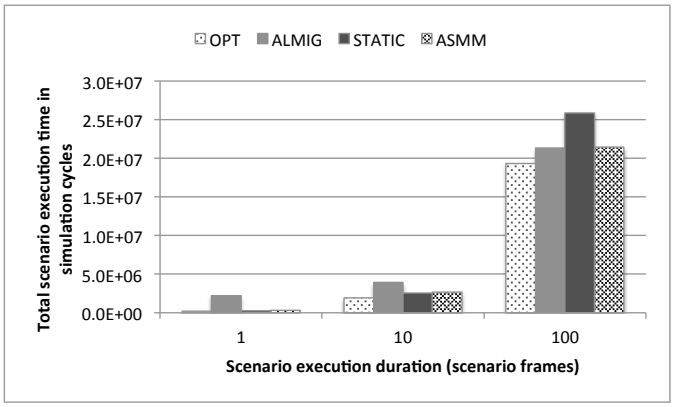

Figure 7: Performance comparison of resource scheduling approaches under different execution duration for a certain workload scenario.

performs worse than *STATIC* for the *s-prob* scenario sequence, as shown in Figure 6. However, when the average scenario duration is large enough, such as in *l-prob*, an approach such as *ALMIG* starts to perform better than *STATIC*. Note that the gap between *ALMIG* (or *ASMM*) and *STATIC* will become larger with an increasing average scenario duration. *ASMM* shows good behaviour for both cases because its ability to avoid unnecessary system reconfigurations. In the case of *s-prob*, only the static mapping approach works slightly better than *ASMM* as the former approach does not suffer from any run-time overheads. As the average execution duration in these two scenario sequences is fairly small, the percentages of performance improvement of *ASMM* comparing with *ALMIG* and *STATIC* are relatively low. However, we should stress that the task migration enabled approaches, i.e., *ASMM* and *ALMIG*, will show better performance with the increasing of average scenario execution duration as shown in Figure 7. This figure gives the performance results of executing a workload scenario with different scenario duration (scenario frames) under the above mentioned scheduling approaches. In the ultimate case, the performance of *ASMM* and ALMIG will very close to *OPT*.

The overhead of our approach involves the run-time computational overhead and the system memory consumption. Comparing the scenario execution time of *ASMM* and *STATIC* in the bottom part of each figure in Figure 4, we can see that the *ASMM* results are close to those for *STATIC*. This means that the run-time overhead of our approach (hundreds of thousands cycles on a 2.1GHZ CPU) does not have a major impact on system performance. Regarding the run-time system storage consumption, we store all the design-time prepared information (pre-optimised mappings and application/architecture information) in the shared memory. For storing the pre-optimised mappings, we assume that the mapping information of each task and each communication channel between tasks can be stored in one *byte*. Each piece of application/system information needs one *word* of system memory and each history scenario duration is encoded using one *byte*. Under these assumptions, our approach needs 880 *bytes* and 920 *bytes* for storing the total of 31 mappings and the application/architecture information, respectively. The memory consumption of pre-optimised mappings is related to the number of possible workload scenarios and the number of tasks in each workload scenarios. And the memory usage for storing the detailed scenario and architecture information is dependent on the number of application tasks and processing elements. Besides storing these types of information, we also need to record the execution history of workload scenarios in the ASMM predictor. The memory usage of our ASMM predictor is dynamic and based on the execution pattern of workload scenarios. In the worst case, for each ASMM with the parameters considered in our experiments, it needs 584 *bytes* to record all possible patterns. As we have 31 possible workload scenarios in our experiments, the total memory consumption of all ASMMs is 18104 *bytes*. The memory usage of each ASMM will, however, increase exponentially with the width of history tables and the number of bins divided for recording scenario execution duration. This problem will be addressed in our future work.

V. Related Research

In recent years, much research has been performed in the area of run-time task remapping for embedded systems to achieve better performance or save energy consumption. In these efforts, a hybrid task mapping approach is commonly used which combines a design-time preparation stage with a run-time dynamic mapping policy to perform task reallocation. For example, Mariani et al. [6] propose a run-time management framework in which Pareto-fronts with system configurations for different applications are determined during design-time DSE, after which heuristics are used to dynamically select a proper system configuration at run time. In [18], a fast and light-weight priority based heuristic is used to select near-optimal configurations explored at design time for the active applications according to the available platform resources. [16] proposes DSE strategies that perform exploration to optimize throughput and energy consumption by considering a generic platform in which design points derived by DSE are selected efficiently at run time. In [15], Schranzhofer et al. propose static and dynamic task mapping approaches for probabilistic applications based on static and dynamic power components. Statically pre-computed template mappings for each execution probability are stored on the system and applied at run time, allowing the system to adapt to changing environment conditions. Based on this work, [4] presents an extension that considers only the static mapping and takes into account the communication and reconfiguration energy component. Schor et al. [14] and Quan et al. [10], [12] also propose scenario-based run-time mapping approaches in which mappings derived from design-time DSE are stored for run-time mapping decisions. However, in these approaches, none of them consider the problem of whether the system will actually benefit from the system reconfiguration when e.g. the system workload changes frequently.

Sarikaya et al. [13] proposed an SMM to predict the run-time application behavior, and applied this technique to an adaptive dynamic power management scheme. In our work, we modified their SMM to predict the duration of workload scenarios in our adaptive run-time framework. In [1], user behavior information is used to adapt the strategy used for resource allocation at run time. Based on the user behavior, an online machine learning model predicts which kind of

communication contention should be minimized on a NoC-based MPSoC. The authors of [3] propose a customer-aware task allocation and scheduling for MPSoCs. At design time, an initial task allocation and scheduling solution (TAS) to optimize energy consumption and system lifetime for each execution mode is generated. At run time, they conduct online adjustment of the TAS based on the processor usage history to guarantee the lifetime reliability and/or reduce the energy consumption. Different to these works, we use the user system execution history to control the task allocation process.

VI. Conclusion

To increase the efficiency of MPSoC systems, we have proposed a run-time adaptive resource scheduler that reconfigures the system based on past and future (predicted) application workload behavior. At design time, we explore performance optimal task mappings for different workload scenarios. These pre-optimised mappings are used at run time by the resource scheduler to reconfigure the system resources. The decision of whether or not to reconfigure is made based on the scenario execution pattern. By using the proposed approach, the system can adapt its behavior according to e.g. user behavior. Experimental results confirm the effectiveness of our approach.

References

[1] C.-L. Chou and R. Marculescu. Run-time task allocation considering user behavior in embedded multiprocessor networks-on-chip. *Trans. Comp.-Aided Des. Integ. Cir. Sys.*, 29(1):78–91, Jan. 2010.

[2] 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.

[3] L. Huang, R. Ye, and Q. Xu. Customer-aware task allocation and scheduling for multi-mode mpsocs. In *Proceedings of the 48th Design Automation Conference*, pages 387–392. ACM, 2011.

[4] A. Hussien, A. Eltawil, R. Amin, and J. Martin. Energy aware task mapping algorithm for heterogeneous mpsoc based architectures. In *Computer Design (ICCD), 2011 IEEE 29th International Conference on*, pages 449–450, 2011.

[5] C. Isci, G. Contreras, and M. Martonosi. Live, runtime phase monitoring and prediction on real systems with application to dynamic power management. In *Microarchitecture, 2006. MICRO-39. 39th Annual IEEE/ACM International Symposium on*, pages 359–370, Dec 2006.

[6] 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.

[7] 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.

[8] 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.

[9] W. Quan and A. Pimentel. An iterative multi-application mapping algorithm for heterogeneous mpsocs. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*, pages 115–124, Oct 2013.

[10] W. Quan and A. D. Pimentel. A scenario-based run-time task mapping algorithm for mpsocs. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, pages 131:1–131:6, New York, NY, USA, 2013. ACM.

[11] W. Quan and A. D. Pimentel. A system-level simulation framework for evaluating task migration in mpsocs. In *Proceedings of the 2014 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '14, pages 13:1–13:9, New York, NY, USA, 2014. ACM.

[12] W. Quan and A. D. Pimentel. A hybrid task mapping algorithm for heterogeneous mpsocs. *ACM Trans. Embed. Comput. Syst.*, 14(1):14:1–14:25, Jan. 2015.

[13] R. Sarikaya, C. Isci, and A. Buyuktosunoglu. Runtime application behavior prediction using a statistical metric model. *Computers, IEEE Transactions on*, 62(3):575–588, March 2013.

[14] 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.

[15] A. Schranzhofer, J.-J. Chen, and L. Thiele. Dynamic power-aware mapping of applications onto heterogeneous mpsoc platforms. *Industrial Informatics, IEEE Transactions on*, 6(4):692–707, 2010.

[16] A. K. Singh, A. Kumar, and T. Srikanthan. Accelerating throughput-aware runtime mapping for heterogeneous mpsocs. *ACM Trans. Des. Autom. Electron. Syst.*, 18(1):9:1–9:29, Jan. 2013.

[17] P. van Stralen and A. D. Pimentel. Scenario-based design space exploration of mpsocs. In *Proc. of IEEE ICCD'10*, pages 305–312, October 2010.

[18] C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. *Computers Digital Techniques, IET*, 5(2):123–135, 2011.