A Hybrid Task Mapping Algorithm for Heterogeneous MPSoCs

Wei Quan, University of Amsterdam; National University of Defense Technology
Andy D. Pimentel, University of Amsterdam

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 article presents a hybrid task mapping algorithm that combines a static mapping exploration and a dynamic mapping optimization to achieve an overall improvement of system efficiency. We evaluate our algorithm using a heterogeneous MPSoC system with three real applications. Experimental results reveal the effectiveness of our proposed algorithm by comparing derived solutions to the ones obtained from several other run-time mapping algorithms. In test cases with three simultaneously active applications, the mapping solutions derived by our approach have average performance improvements ranging from 45.9% to 105.9% and average energy savings ranging from 14.6% to 23.5%.

Categories and Subject Descriptors: C.4 [**Performance of Systems**]: Performance Attributes

General Terms: Algorithm, Design, Performance

Additional Key Words and Phrases: Embedded systems, KPN, MPSoC, task mapping, simulation

ACM Reference Format:
Wei Quan and Andy D. Pimentel. 2013. A Hybrid Task Mapping Algorithm for Heterogeneous MPSoCs. *ACM Trans. Embedd. Comput. Syst.* 0, 0, Article 00 (0000), 25 pages.
DOI:http://dx.doi.org/10.1145/0000000.0000000

1. INTRODUCTION

Modern embedded systems, which are more and more based on Multi-Processor System-on-Chip (MPSoC) architectures, often require supporting an increasing number of applications and standards. In these systems, multiple applications can run concurrently and are thus simultaneously contending for system resources. For each single application, there are often 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 the battery. As a consequence, the behaviour of application workloads executing on the embedded system can change dramatically over time. Typically, the target MPSoC architecture platforms are heterogeneous in nature, as such systems are capable of providing better performance and energy trade-offs than their homogeneous counterparts [Kumar et al. 2004]. Here, the process of application task mapping plays a crucial role in exploiting the system properties such that applications can meet their, often diverse, demands on performance and energy efficiency [Sun and Sugawara 2011].

Author's addresses: W. Quan and A. D. Pimentel, Informatics Institute, University of Amsterdam, Science Park 904, 1098XH Amsterdam, The Netherlands; W. Quan, School of Computer Science, National University of Defence Technology, Yanwachi Main Street 47, Changsha, Hunan, China.

DOI:http://dx.doi.org/10.1145/0000000.0000000

The problem of optimally mapping tasks onto a given set of heterogeneous processors for maximal throughput (performance) or minimal overall energy consumption has been known, in general, to be NP-complete. When considering mapping multiple applications onto a target architecture, this problem is exacerbated as the resource contention between applications should be carefully considered in this case. State-of-the-art methods for solving this problem can be divided into three categories: static, dynamic and hybrid task mapping algorithms which, respectively, work at design time, run time and both design time and run time. Traditionally, the task mapping problem is solved statically at design time for which there are many known task mapping algorithms targeting different application domains and different hardware architectures. These algorithms typically use computationally intensive search methods to find the optimal mapping or near optimal mapping for the applications that may run on the system. Dynamic task mapping techniques, on the other hand, cannot be computationally intensive as they have to efficiently make task mapping decisions at run time. Therefore, these techniques typically use heuristics to find good task mappings. Evidently, static task mapping techniques usually obtain mappings of higher quality compared to those derived from dynamic algorithms as the former allow for exploring a larger design space for the underlying architecture. This, of course, at the cost of consuming more time. Another drawback of static mapping techniques is that they cannot cope with dynamic application behaviour in which different combinations of applications can be executing concurrently over time that are contending for system resources. To overcome the shortcomings of pure static and dynamic task mapping algorithms, hybrid (semi-static) approaches have become increasingly popular in recent years. Usually, in this kind of approaches, multiple mapping solutions are found at design time and applied at run time based on the current state of the system. However, most hybrid mapping approaches still suffer from shortcomings regarding the support of adaptivity to cope with application dynamism. For example, many approaches do not support the handling of new, incoming applications that were not known at design time, or they cannot capture fine-grained application dynamism, such as the dynamism that exists within each application. In this work, we propose a novel hybrid task mapping (HTM) algorithm for heterogeneous multimedia MPSoCs that tries to capture dynamic behaviour both *between and within applications* and which exploits the advantages from both static and dynamic mapping algorithms.

Like all hybrid approaches, our proposed approach distinguishes two stages. Firstly, the design-time stage performs design space exploration (DSE) to find two optimal mappings for each application with the objectives of maximizing the throughput and maximizing the throughput under a predefined energy budget respectively. Secondly, the run-time stage dynamically optimizes the mapping of the running application(s) based on the optimal mappings of the corresponding applications explored in the first (design-time) stage and the system execution state. The second stage can be further split into two steps, mapping initialization and mapping customization, which optimize the mapping with the objective of maximizing the throughput under the predefined energy budget and further improve the performance of the mapping using an application-dependent objective respectively. Using a range of experiments, we show the effectiveness of our proposed approach by comparing derived solutions to the ones obtained from several other run-time mapping algorithms.

The remainder of this article is organised as follows. Section 2 presents a motivational example for our work. Section 3 gives some prerequisites and the problem definition for this work. Section 4 provides a detailed description of our hybrid task mapping (HTM) algorithm. Section 5 introduces the experimental environment and presents the results of our experiments. Section 6 discusses related work, after which Section 7 concludes the paper.

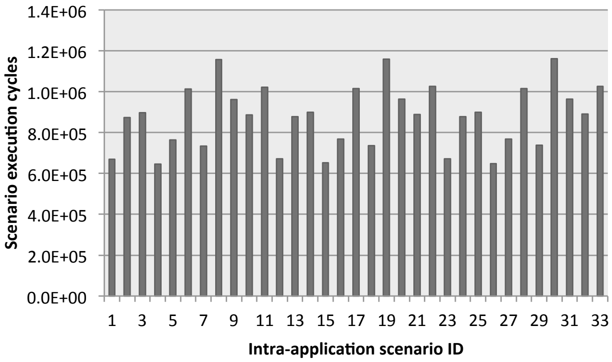


Fig. 1. Intra-application scenario performance of MJPEG

2. MOTIVATIONAL EXAMPLE

To support dynamism between and within applications, we use the concept of *scenarios* [Paul et al. 2006; Gheorghita et al. 2009; van Stralen and Pimentel 2010b]. Here, one can distinguish two forms of scenarios to capture dynamic application behaviour: inter-application scenarios describe the simultaneously running applications in the system, while intra-application scenarios define the different execution modes within 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. At design time, a system designer could aim at finding the optimal mapping of application tasks to MPSoC processing resources for each workload scenario with different objectives (performance/energy). However, when the number of applications and application modes increase, the total number of workload scenarios will explode exponentially. Consider, e.g., 10 applications with 5 execution modes for each application. In this case, there will be 60 million workload scenarios. If it takes one second to find the optimal mapping for each scenario at design time, then one would need nearly two years to obtain all the optimal mappings. Moreover, storing all these optimal mappings (6^{10}-1 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.

A general hybrid 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 [Gheorghita et al. 2009; Quan and Pimentel 2013b]. Such clustering implies a significant space reduction needed to store the mappings. Moreover, so-called scenario-based design space exploration [van Stralen and Pimentel 2010a] can be deployed to efficiently find these mappings by only evaluating a representative subset of scenarios for each cluster. For example, let us consider a clustering method in which we find and store a single mapping for each inter-application scenario that yields, on average, the best performance for all possible intra-application scenarios within the inter-application scenario [Quan and Pimentel 2013b; Schor et al. 2012]. However, as we can see from the behaviour 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, besides the MJPEG encoder, two other simultaneously running multimedia applications: a MP3 decoder and a Sobel filter for edge detection in images.

From this example, we can observe that the use of cluster-level mappings (i.e., mappings found to be good for an entire cluster of workload scenarios) could 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. Besides this, there are two additional drawbacks of this hybrid approach, as was already discussed in the previous section: it lacks the adaptivity of supporting new applications (i.e., adding an application would require to redo the entire process of clustering and design-time DSE) and it still suffers from relatively high memory usage for storing all the pre-optimized mappings when the number of applications increases (for the example clustering method, 2^{10}-1 mappings need to be stored for 10 applications with 5 modes in each application).

In this work, we solve the first problem by splitting the handling of application dynamism using two run-time steps: mapping initialization and mapping customization. In the first step, an adequate mapping for a detected workload scenario is found (as above), after which the second step 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. To address the aforementioned problems of supporting new applications and storage requirements, the design-time phase of our approach explores two optimal mappings for each intra-application scenario in every single application. It does so for two different objectives: maximizing the throughput and maximizing the throughput under a predefined energy budget. By using this method, the number of mappings that need to be determined and stored at design time is greatly reduced (100 mappings need to be stored for 10 applications with 5 modes in each application). Also, if a new application needs to be supported on the system, only two pre-optimized mappings for each intra-application scenario of this application need to be provided to the system, avoiding the need of exploring the mappings for all possible new inter-application scenarios.

3. PREREQUISITES AND PROBLEM DEFINITION

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

3.1. Application Model

In this paper, we target the multimedia application domain. For this reason, we use the Kahn Process Network (KPN) model of computation [Kahn 1974] 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)[1] 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 Motion-JPEG (MJPEG) decoder application used in the previous section.

3.2. Architecture Model

In this work, we restrict ourselves to heterogeneous MPSoC architectures with shared memory. An architecture can be modelled 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

[1] We use the terms process and task interchangeably in this article.

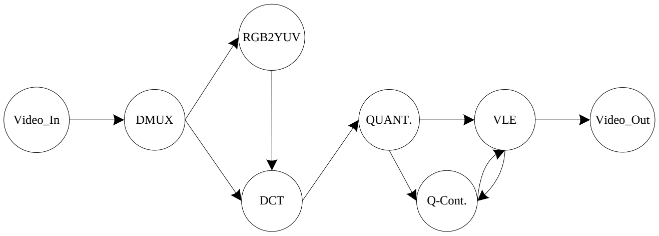


Fig. 2. KPN for MJPEG

$c_{ij} = (pe_i, pe_j) \in PE \times PE$ representing a buffered communication medium, composed of a network channel (like a Bus, NoC, etc.) and a buffer located in shared memory, 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 via channel c_{xy} that connects pe_x and pe_y is $C_{ij}^{c_{xy}}$. Here, the time units for the communication cost and the computation cost should be unified to the cycles under the same clock frequency in case of different clock frequencies on the target system. With respect to power consumption, SP_i and DP_i refer to the static and dynamic power consumption for pe_i. Besides processing elements, another main component of energy consumption in our target system is the shared memory. For this component, we denote the static and average dynamic power consumption (for read/write transactions) as SM and DM respectively.

3.3. Task Mapping

The task mapping defines the binding of the components in a KPN application (including the processes and the FIFO communication channels) to 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 ($\mu : P \to PE$, $\eta : F \to C$) such that it satisfies $\forall f \in F, src(\eta(f)) = \mu(src(f)) \wedge dst(\eta(f)) = \mu(dst(f))$. When tasks are mapped onto the underlying architecture, the usage U_k of each pe_k can be calculated by equation 1, where $p_i \mapsto pe_k$ and $p_j \mapsto pe_y$ mean that tasks p_i, p_j are mapped onto processors pe_k and pe_y respectively. Note that, if two tasks are mapped onto the same processor (k equals to y in equation 1), then the communication cost between these two tasks will be neglected as it uses the internal memory on the processor for communication.

$$U_k = \sum_{p_i \mapsto pe_k, p_j \mapsto pe_y} (T_i^k + C_{ij}^{c_{ky}}) \tag{1}$$

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 $app_i \in A$ (if app_i is active then $app_i = 1$, else $app_i = 0$), is the set of all inter-application scenarios. And $sa_j^i = \{...,\ app_k = md_{j_k}^k,\ ...\}$, with $0 \leq k \leq m$, $app_k \in A \wedge app_k = 1 \in se_i$ and $md_{j_k}^k \in M^k$, 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 task mapping of applications in two stages. In the first stage, which is performed at design time, we perform DSE for each intra-application scenario of each application (denoted by scenario s_i in the whole workload scenario space S) to find two mappings that show i) the maximal throughput (optimization objective O_t) and ii) the maximal throughput under a certain energy budget b_i (optimization objective O_{tb}) respectively. Here, b_i is defined as the energy budget for workload scenario s_i. As will be explained in detail in the next section, the two mappings derived from design-time DSE are stored so they can be used for mapping initialization in the second (run-time) stage to get a final mapping – either by directly using the stored mappings (if a newly detected workload scenario only contains a single active application) or by deriving a new system mapping from the stored per-application mappings – when a new workload scenario has been detected. As mentioned above, both the O_t and O_{tb} objectives are used in design-time mapping exploration, whereas only O_{tb} will be used for run-time mapping optimization. To allow our design-time DSE to construct a Pareto Front with regard to the performance and energy consumption of mapping solutions, we change the objective of maximal throughput into a minimal objective, i.e. $O_p = 1/O_t$. Consequently, the system objectives turn into minimizing O_p and O_{pb}. More specifically, we use system energy consumption $E_{s_i}, s_i \in S$ and total workload scenario execution time $X_{s_i}, s_i \in S$ for workload scenario s_i to find the optimal or near optimal mappings that satisfy the above two objectives at design time. For the purpose of run-time mapping customization, we also use an application-specific objective (besides the system-wide objective O_{pb}), denoted as $O_{\beta i}$ for application app_i. This objective defines the performance requirements of each separate application, which in our case is still defined in terms of throughput.

Under these definitions and given the $KPN = (P, F)$ for each application and an $MPSoC = (PE, C)$, our goal is to find the optimal or near optimal mapping at run time for each detected workload scenario $s_i \in S$ with the objective to minimize O_{pb} and where each application should also satisfy its own objective O_β.

4. A NOVEL HYBRID TASK MAPPING (HTM) ALGORITHM

As shown in Figure 3, the entire workflow of our approach can be divided into three steps: design-time preparation, run-time mapping initialization and run-time mapping customization. As mentioned before, in the step of design-time preparation, two optimized mappings for each intra-application scenario are prepared by exploring the corresponding mapping space. These pre-optimized mappings will be stored in system memory for run-time mapping initialization/optimization. At run time, when the system detects a new workload scenario, our HTM algorithm will try to produce a good mapping for the active applications in the scenario using (a combination of) the stored per-application mappings derived from the design-time preparation step. Here, we make the assumption that each workload scenario will execute long enough to justify a possible remapping of application tasks. Otherwise, a trade-off needs to be made between the cost of remapping and the mapping performance improvement, which is beyond the scope of this article. This process of determining a new mapping for all applications when a new workload scenario has been detected, is referred to as mapping initialization. The objective of this process is to maximize the system throughput under the predefined energy budget (or, in other words, minimize O_{pb}). The mapping initialisation, which uses the stored mapping information of isolated applications, may not immediately lead to finding the optimal system mapping for a complete identified workload scenario (i.e., the combination of applications that form the scenario). Therefore, during the execution of a certain workload scenario, the HTM algorithm will try to actively further improve the mapping performance when application-specific objec-

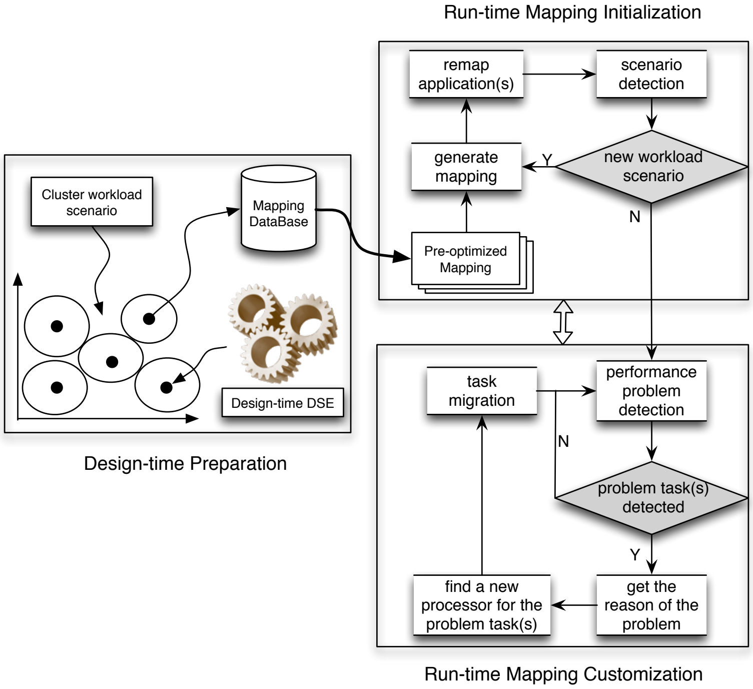


Fig. 3. The workflow of HTM

tives are (about to be) violated. To this end, it continuously monitors the system and tries to perform relatively small mapping customizations to gradually further improve the system performance. Evidently, to reduce migration overheads, the algorithm aims at keeping the number of required task migrations as low as possible. This process is called mapping customization. The details of these three steps will be explained in the following subsections.

4.1. Design-time Preparation

At design time, the mappings with minimal O_p and O_{pb} will be searched for all intra-application scenarios in each isolated application (i.e., in those inter-application scenarios with only a single active application). As shown in Figure 3, it would also be possible to cluster intra-application scenarios of applications, and only determine mappings with minimal O_p and O_{pb} for an entire cluster of intra-application scenarios. This would further reduce the number of mappings that need to be explored and stored. However, in this article, we assume that mappings with minimal O_p and O_{pb} are searched for all separate intra-application scenarios of applications. To find these mappings, we deploy a scenario-based DSE approach [van Stralen and Pimentel 2010a], which is based on the well-known NSGA-II genetic algorithm (GA). As our target MPSoC platform is known, we can use a simplified version of the approach in [van Stralen and Pimentel 2010a]. The implementation of the genetic algorithm is explained below.

4.1.1. Chromosome Representation. We use a standard approach for representing mappings within the GA's chromosomes [Erbas et al. 2006]. It contains two parts. The first part contains the mapping of the processes in the application, whereas the second part describes the mapping of the communication channels of the application. Implicitly,

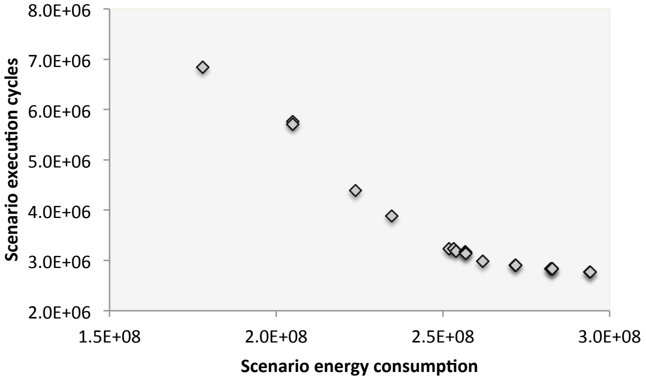


Fig. 4. Pareto Front of a workload scenario with application Sobel active

this also contains the resource allocation for the platform. Resources that are not used in any binding (processing or communicating resources) are also not allocated on the platform. This representation method could also be applied to store the pre-optimized mapping on the target system. In that case, for each process and FIFO channel in the application KPN, it would need $\log_2^{|PE|}$ bits and $\log_2^{|C|}$ bits respectively to encode the mapping information. To store a mapping in system memory for application app_i that contains $|P_i|$ processes and $|F_i|$ FIFO channels, it would need $|P_i| * \log_2^{|PE|} + |F_i| * \log_2^{|C|}$ bits.

4.1.2. Fitness Function. To find the mappings with minimal O_p and O_{pb} for workload scenario s_i in the mapping space in question, the Pareto Front of mapping performance and mapping energy consumption is generated by solving the following multi-objective optimization problem:

$$min[E_{s_i}, X_{s_i}]. \tag{2}$$

Evaluating the fitness value of each individual (i.e., design point) is performed using the Sesame system-level MPSoC simulation framework [Pimentel et al. 2006]. After the GA-based DSE, a Pareto Front of solutions is generated, as illustrated in Figure 4. Looking at the Pareto Front, one can easily obtain the mappings satisfying the minimal O_p and O_{pb}, which equal to the mappings with minimal X_{s_i} and minimal X_{s_i} under the energy budget of b_i. This energy budget of b_i is calculated using equation 3. The first part α in equation 3 is a constant scaling factor set for the energy budget and the second part represents the minimal energy consumption for workload scenario s_i. Here, we assume that the energy budget b_i should be higher than the minimal energy consumption of the solution mappings found for the workload scenario s_i.

$$b_i = \alpha * argmin(E_{s_i}), with\ \alpha > 1 \tag{3}$$

4.1.3. Operators for NSGA-II. To effectively search for global optimal mapping solutions, and escape possible local ones, the crossover and mutation operators are important components of a GA. With respect to the crossover operator, the most common methods are one-point crossover, two-point crossover and uniform crossover. In our work, the one-point crossover is used as it is simple and yields more or less the same effect as the other approaches. Regarding mutation, we use an operator that randomly selects an application task that is subsequently moved to a randomly selected processor. The last step is to set appropriate parameters for the GA, such as population size, crossover and mutation probabilities, etc. The parameters used in our work will be introduced in the experiment section.

4.2. Run-time Mapping Initialization

In the mapping initialization stage, we use an Energy-aware Iterative multi-application Mapping (EIM) algorithm [Quan and Pimentel 2013a] to find a good initial mapping for a newly detected workload scenario. Our EIM 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 the intra-application dynamism in those inter-application scenarios with only a single active application.

The mappings derived from the design-time preparation stage are used by the EIM algorithm, as shown in lines 1-3 of Algorithm 1. To this end, these mappings are stored in a so-called *scenario database*. Besides storing these two mappings, the estimated minimum energy consumption for each intra-application scenario of each application is also stored in the database. This value is based on the most energy efficient mapping found in the Pareto Front generated by design-time DSE (e.g., the left-most point in Figure 4) and is calculated using equation 4. The calculation and use of this value will be explained later on. When the system detects a new workload scenario, the EIM algorithm will first choose the corresponding optimal mapping – as stored in the scenario database – for each application active in the workload scenario as the initial mapping. As the database only stores mappings for the intra-application scenarios of each single application, its size typically is relatively small. However, if its size becomes too large, then the size can be controlled by clustering intra-application scenarios (as explained in Section 4.1) and choosing a proper granularity of scenario clusters.

If there is only a single application active in the workload scenario, then the mapping selected from the scenario database as the initial mapping is the mapping with the maximal throughput under a given energy budget for that particular application. Hereafter, the algorithm directly returns this initial mapping as a final mapping decision. Otherwise, if there are multiple applications active simultaneously, then the mapping with maximal throughput for each active application will be chosen as initial mappings. These initial per-application mappings will then simply be merged together to form the initial mapping for the complete workload scenario. Here, there are two reasons for not choosing the mapping with maximal throughput under a certain energy budget as the initial mapping. First, the communication locality behaviour of the mapping with maximal throughput under an energy budget typically is not as good as the one with maximal throughput without an energy budget. Our run-time algorithm exploits this locality incorporated in the initial per-application mappings for further improvement of the workload scenario mapping. Second, we will consider the energy constraints during the mapping optimization process at run time, so we do not yet have to consider an energy budget for the initial mapping in the case of a workload scenario with multiple active applications.

The dynamic part of our EIM algorithm is only used for those workload scenarios that contain multiple simultaneously active applications. It aims at further optimizing the initial mapping found during the static part of the EIM algorithm, as described above. The strategy used in the dynamic part of the EIM algorithm is described below.

For the active multi-application scenario, our algorithm will optimize the mapping with the objective to minimize the system metric O_{pb}. Consequently, the optimal mapping for each scenario is the one that has the minimal O_p among all the possible mappings under energy budget of b_i for workload scenario s_i. It is, however, extremely hard to find the optimal mapping for each workload scenario at run time because of

ALGORITHM 1: EIM algorithm

```
Input: $KPN_{app_{active}}$, $MPSoC$, scenario_id($s_i$)
Output: $(\mu, \eta)$
(μ, η) = getInitMapping($s_i$);
if singleAppActive($s_i$) == true:
    return $(\mu, \eta)$;
else:
    $U$ = peUsage($KPN_{app_{active}}$, $MPSoC$, $\mu$, $\eta$);
    $M_p$ = maxPUsage($U$);
    $V_p$ = varPUsage($U$);
    $b_i$ = eBudget($s_i$);
    return iterativePOpt($\mu$, $\eta$, $M_p$, $V_p$, $b_i$);
```

the following reasons. Firstly, as one cannot obtain the true value of O_p before actually executing the application on the target platform, an estimated $O_p^{'}$ needs to be used to guide the algorithm to find the optimal mapping. Here, there exists of course a clear accuracy/overhead trade-off between different estimation techniques. Efficient but less accurate run-time mapping-performance estimation techniques may lead to sub-optimal mappings, while the high overhead of more accurate techniques may neutralise the performance benefits of the mapping optimization itself. Secondly, the mapping problem is NP-complete, as was mentioned before. It is unrealistic for a run-time mapping algorithm to explore the entire mapping space to determine the optimal mapping for a scenario.

To solve the above problems, we use an alternative method using heuristics to search a part of the mapping space which may contain the optimal or a near optimal mapping. To this end, we change the objective of performance into the optimization of two alternative metrics: M_p and V_p that, respectively, represent the maximal usage and usage variation in $U_k, pe_k \in PE$ (see also equation 1). In this case, we do not need to use the metric O_p as the optimization objective, thereby addressing the first of the two above problems. Regarding the second problem, by using an optimization heuristic based on the metrics M_p and V_p, we aim at finding an optimal or near optimal mapping in a computationally efficient fashion. The rationale behind this heuristic is that a better mapping for the objective of high throughput usually has smaller M_p and V_p values. For the purpose of restricting the energy consumption of the resulting mapping, we use the estimated energy consumption of a mapping (μ_i, η_i) for workload scenario s_j given by equation 4 and the system energy budget b_j calculated by equation 5, to control the search space of possible mappings. Here, E_{mk} represents the estimated minimal energy consumption for application app_k which is stored in the scenario database.

In equation 4, $E_p^{'}$ is the dynamic and static energy consumed by all active processors and $E_m^{'}$ represents the dynamic and static energy consumption of the shared memory. This relatively simple energy model is built on several assumptions of the target architecture: 1) the power model used for the shared memory in the system already includes the power consumption of the bus connected to it; 2) for simplicity, we ignore the energy consumption caused by resource contention and communication delays. We make these assumptions to control the complexity of the analytical energy model and reduce the computation cost at run time. It is hard (and computationally much more expensive) to analytically derive the task stalls incurred by delays due to communication and resource contention in heterogeneous MPSoCs as considered in this work. So, here we make a tradeoff between accuracy and computation complexity. Under these assumptions, the system active time for a specific workload scenario is simply assumed to be $argmax(U_i)$, which is subsequently used to calculate the static energy consumption. Notice that the energy budget defined here is different from the energy budget used for design-time DSE (equation 3). In equation 5, the *estimated* minimal energy consumption is used instead of the actual minimal energy consumption determined (using the Sesame simulator) by design-time DSE like is used in equation 3. The estimated minimal energy is computed by applying equation 4 to the mapping that has been found to yield the highest energy efficiency after performing design-time DSE and is stored in the scenario database (as was explained earlier). This is motivated by the fact that the energy consumption of possible mapping solutions explored at run time are also estimated using equation 4. Therefore, we also make a projection of the energy budget using equation 4.

$$E_{ij} = E_p^{'} + E_m^{'} \tag{4a}$$

$$E_p^{'} = \sum_{active\_pe_k} (DP_k * U_k + SP_k * argmax(U_k)) \tag{4b}$$

$$E_m^{'} = DM * \sum_{\substack{c_{xy}=mem \\ f_{rt} \mapsto c_{xy} \in \eta_i}} (C_{rt}^{c_{xy}}) + SM * argmax(U_k) \tag{4c}$$

$$b_j = \alpha * \sum_{active\_app_k \in s_j} E_{mk} \tag{5}$$

The mapping algorithm for the workload scenarios with multiple active applications is outlined in Algorithm 2, which will be executed in an iterative fashion. The starting mapping used in this algorithm is the one derived from Algorithm 1. In each iteration, the algorithm first proposes a new mapping for each active application as shown in line 2 of Algorithm 2. In this process, the algorithm searches the mapping space using the following greedy pattern: it checks the processors in U_k in descending order to determine whether the KPN application in question has a task or a bundle of adjacent, communicating tasks[2] resident on this processor. If so, then the algorithm finds a possible substitute processor for the task/adjacent tasks that satisfies the following conditions:

(1) The $M_p^{'}$ of the new mapping is smaller than the M_p of the old mapping.
(2) If the previous condition cannot be satisfied, then the algorithm tries to find a substitute processor for which the resulting $M_p^{'}$ is equal to M_p and $V_p^{'}$ is smaller than V_p. If the first condition was satisfied, then this condition will never be used in this particular iteration.
(3) The estimated energy consumption of the new mapping should be smaller than the energy budget b_i of the (intra-application) scenario s_i in question.

The above process proposes new mappings for those applications that satisfy the conditions (for the other applications, the mapping remains unaltered). These newly proposed mappings are either a mapping that has a minimal $M_p^{'}$ (if condition 1 has been satisfied) or a mapping with minimal $V_p^{'}$. However, in the above process, it can also be the case that there are multiple new mappings proposed for an application, e.g. when there are multiple tasks (or task bundles) that can be remapped and for which the

[2] Mapping such a task bundle to a single processor is the outcome of the design-time mapping optimization to reduce communication overhead.

ALGORITHM 2: IPO algorithm

```
iterativePOpt($\mu$, $\eta$, $M_p$, $V_p$, $b_i$):
1:  for each active $app_j$:
2:      $(\mu_j, \eta_j)$ = getPSubstitute($\mu$, $\eta$);
3:      if $(\mu_j, \eta_j)$ != $(\mu, \eta)$:
4:          $U$ = peUsage($KPN_{app_{active}}$, $MPSoC$, $\mu_j$, $\eta_j$);
5:          $M_p^j$ = maxPUsage($U$);
6:          $V_p^j$ = varPUsage($U$);
7:          $L^j$ = perfLoss($app_j$, $\mu$, $\eta$, $\mu_j$, $\eta_j$);
8:  $M_p^k$ = $argmin(M_p^j)$;
9:  if $M_p^k < M_p$:
10:     $(\mu^*, \eta^*)$ = $(\mu_k, \eta_k)$;
11:     iterativePOpt($\mu^*$, $\eta^*$, $M_p^k$, $V_p^k$, $b_i$);
12: else:
13:     $V_p^t$ = $argmin(V_p^j + L^j)$;
14:     $(\mu^*, \eta^*)$ = $(\mu_t, \eta_t)$;
15:     if $(\mu^*, \eta^*)$ == $(\mu, \eta)$:
16:         return $(\mu, \eta)$;
17:     else:
18:         iterativePOpt($\mu^*$, $\eta^*$, $M_p^t$, $V_p^t$, $b_i$);
```

above conditions hold. In these cases, we use another metric, L, to decide on the final proposed mapping, where the value of L needs to be minimized. The metric L tries to capture the performance loss of a task remapping for the application in question[3] and is calculated using equation 6.

$$L = \sum_{p_k \in B_i^j} (T_k^j - T_k^i) + (C_{kt}^{c_{jl}} - C_{kt}^{c_{il}}) \tag{6}$$

Here, we denote the task/task bundle that needs to be remapped from pe_i to pe_j as B_i^j.

After the algorithm has proposed a new mapping for each application, the next step is to select the *most effective* among these remapping proposals to be used for the next optimization iteration of the algorithm or return a mapping as the final one, as shown in lines 8-18 of Algorithm 2. If no new mapping has been proposed for any of the applications in the workload scenario in the previous step, then the input mapping will be returned as the final optimized result. Otherwise, we use the following conditions to select the most effective remapping for the next iteration of the algorithm:

(1) If there is one and only one proposed mapping that has the minimal $M_p^{'}$ and this $M_p^{'}$ is smaller than the M_p of the original mapping, then this mapping will be passed to the next mapping optimization iteration.

(2) If the first condition has not been satisfied, then the proposed mapping with $argmin(V_p^{'} + L)$ will be taken as the input mapping for the next iteration. The rationale behind this is that the algorithm tries to gradually optimize the mapping for the entire workload scenario while keeping the performance loss for a single application due to task remapping as small as possible (i.e., taking into account the processor affinity of the tasks proposed to be remapped).

[3] We note that L can be negative, implying that the task/task bundle has a higher affinity with the processor it is proposed to be mapped on.

The time complexity of our EIM algorithm is highly dependent on the diversity of each pre-optimized (i.e., statically derived) mapping stored in system memory, especially considering the iteration count of our EIM algorithm. The diversity $|D_i|$ of an application (app_i) mapping is defined as the number of pipeline segments in this mapping. Under this definition, $|D_i| = 1$ and $|D_i| = |P_i|$ mean that all the tasks in app_i are mapped onto a single processor and different processors respectively. Here, $|P_i|$ represents the number of tasks in app_i. In the function on line 2 in Algorithm 2, the maximal number of possible new mappings is $|PE| * |D_i| * |PE|$. For each possible new mapping, the time consumed for computing the values of M_p and V_p is $O(|PE||P| + |F||C| + 2|PE|)$. Consequently, the time complexity of each active application in each iteration is $O(|PE|^2|D_i|(|PE||P| + |F||C| + 2|PE|))$. The approximate time complexity of each iteration then is $O(|PE|^2|D|(|PE||P| + |F||C| + 2|PE|))$, where $|PE|$, $|D|$, $|P|$, $|F|$ and $|C|$ respectively represent the total number of processor elements, the sum of $|D_i|$ of each active application app_i, the total number of active tasks, the total number of active FIFO channels and the total number of communication channels. As the algorithm searches the mapping space to minimize M_p and V_p simultaneously, the maximal iteration count of Algorithm 2 is $argmax(|D|, |D||PE|)$, where the first and the second argument represent the maximal iteration count needed for searching each of the above metrics. Then, the overall time complexity of Algorithm 2 is $O(|PE|^3|D|^2(|PE||P| + |F||C| + 2|PE|))$.

4.3. Run-time Mapping Customization

After the mapping is initialized for the active workload scenario, the system will monitor the execution of this workload scenario. As mentioned before, the application-specific objective of each active application will be applied at run time, which will be used to determine whether or not a performance problem arises. Here, we assume that the target MPSoC should, in principle, be dimensioned such that it can accommodate all possible target applications but that a particular application's performance objective may be violated due to a bad mapping. When the system detects that an objective is unsatisfied, a Scenario-based run-time Task Mapping (STM) algorithm is applied 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. This STM algorithm is based on our previous work in [Quan and Pimentel 2013b], but has been extended in this work to also work for heterogeneous MPSoC architectures. The main steps of our STM algorithm are described below.

4.3.1. Finding the Critical Task. The first step of our STM algorithm is to find the so-called *critical task* for the application that missed its objective, as shown in lines 5-8 of Algorithm 3. 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 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

ALGORITHM 3: STM algorithm

```
Input: $KPN_{app_i}$, $\mu$, $\eta$
Output: $New(\mu, \eta)$
list: TC, CIC, CIB, PU
pCIC = $\delta_c$, pCIB = $\delta_b$
results[] = getStatistics();
taskCost($KPN_{app_i}$, results, TC, CIC, CIB);
peUsage(results, PU);
while(1) :
    if (apptype = getType($KPN_{app_i}$)) == DATA_PARALLEL :
        critical = findDPCritical($KPN_{app_i}$, TC, CIC, CIB, pCIB, pCIC);
    else :
        critical = findCritical($KPN_{app_i}$, TC, CIC, CIB, pCIB, pCIC);
    reason = findReason(critical, CIC, CIB, pCIB, pCIC);
    if reason == POOR_LOCALITY :
        MCC[] = minCircle($KPN_{app_i}$, results, critical);
        if GetSubstitute(PU, $\mu$, $\eta$, MCC, apptype) == true :
            return $New(\mu, \eta)$;
        else failed;
    else if reason == LOAD_IMBALANCE :
        if GetSubstitute(PU, $\mu$, $\eta$, apptype) == true :
            return $New(\mu, \eta)$;
        else failed;
    else :
        pCIB += $\varepsilon$;
        pCIC -= $\varepsilon$;
```

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.

4.3.2. 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

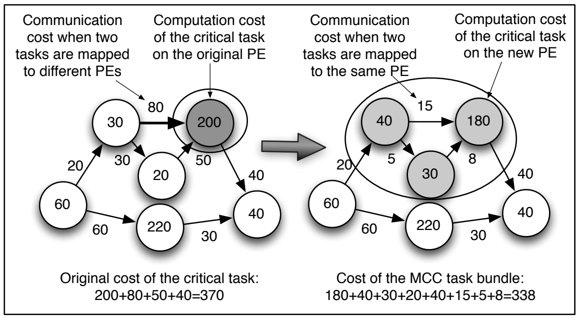

Communication cost when two tasks are mapped to different PEs
Computation cost of the critical task on the original PE
Communication cost when two tasks are mapped to the same PE
Computation cost of the critical task on the new PE
30
80
200
20
30
20
50
40
60
60
220
30
40
40
15
180
20
5
30
8
40
60
60
220
30
40
Original cost of the critical task: 200+80+50+40=370
Cost of the MCC task bundle: 180+40+30+20+40+15+5+8=338

Fig. 5. Example of an MCC for a critical task (gray task on the left-hand side)

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.

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 possibly 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 7 for the critical task p_i:

$$min(Circle\_Cost(p_i)^z_{mn}),\ with\ 0 \le m,n < |P|, m \le i \le n,\ 0 \le z < |PE| \tag{7}$$

where:

$$Circle\_Cost(p_i)^z_{mn} = \sum_{\substack{m \le k \le n \\ p_k \mapsto pe_z}} T^z_k + \sum_{\substack{m \le k \le n \\ p_k \mapsto pe_z}} \sum_{0 \le j < |P|} C^{c_{zy}}_{kj} \tag{8}$$

where T^z_k denotes the execution time of task k for PE z, and $C^{c_{zy}}_{kj}$ denotes the communication overhead between tasks k and j (see Section 3.2). Figure 5 shows an example of an MCC (indicated by the oval on the right-hand side) consisting of a task bundle of three tasks, including the critical task (gray task).

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. As the task to processor binding is part of the calculation of the MCC of the critical task, implying that the binding with the minimal MCC is known after

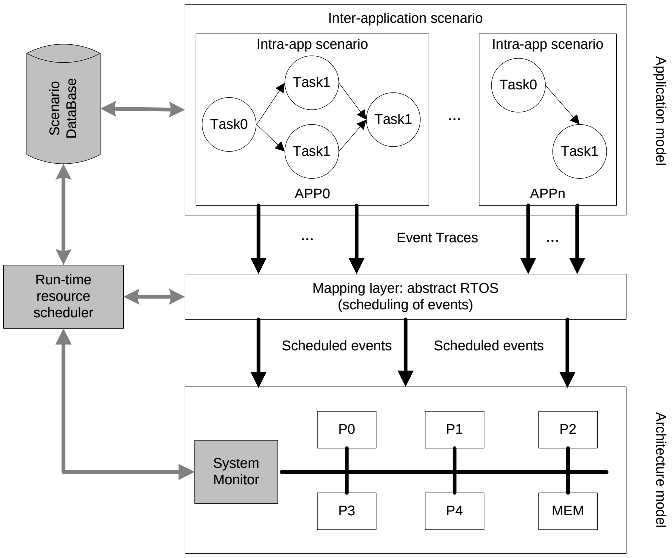


Fig. 6. Extended Sesame framework

this calculation, the substitute PE is the processor used in this binding. However, if the MCC solely consists of the critical task itself, then the critical task will be mapped together with the neighbouring task with which the critical task has the heaviest communication to the processor that yields the minimal task cost for the combined tasks. 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.

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. For this purpose, the PU list is used, containing the processor utilisations for each PE. The substitute PE for the critical task is the PE with the lowest utilisation in the PU list that is different from the PE onto which the critical task is currently mapped. If such a substitute does not exist, then the algorithm cannot find a better mapping.

In the STM algorithm, the most time consuming part is the function for finding the MCC of the critical task in line 11 of Algorithm 3. The time consumption of an exhaustive search method to find the result of equation 7 is $O(|P_i|^{|PE|}|F_i|^{|C|})$. To reduce the time complexity of this function, we find the MCC by gradually adding a communicating task into the bundle of tasks around the critical task to see whether the minimal cost of the new MCC task bundle (if the task bundle is mapped to a different processor) is larger than the previous task bundle. If so, this communicating task will not be added to the MCC task bundle. By using this greedy method, the maximal time complexity is reduced to $O(|P_i||PE| * N)$ where $|P_i||PE|$ represents the maximal number of times of selecting the MCC task bundle and N represents the time complexity of calculating the cost of a new MCC task bundle.

5. EXPERIMENTS

5.1. Experimental Framework

To evaluate the efficiency of our HTM algorithm and the mappings found at run time by this algorithm, we deploy the Sesame system-level MPSoC simulator [Pimentel et al. 2006]. To this end, we have extended this simulator with our run-time resource scheduling framework, as illustrated in Figure 6. Our extension includes the Scenario DataBase (SDB), a Run-time System Monitor (RSM) and a Run-time Resource Scheduler (RRS). The SDB is used to store the mappings for intra-application scenarios of each application as derived from design-time DSE as well as the application-specific information like the performance objective, energy budget, etc. The RSM is in charge of detecting and identifying the active workload scenario, and also collects the statistics (e.g., performance of each application, system execution information, etc.) from the underlying system during the execution of a certain workload scenario. Here, we would like to note that the mechanism for the actual scenario detection and identification is beyond the scope of this article. The RRS uses the HTM algorithm (the EIM part) and the identified workload scenario by the RSM to do mapping initialization at the beginning of each new workload scenario. During the execution of a workload scenario, the RRS uses the HTM algorithm (the STM part) and the statistics collected by the RSM to do mapping customization when there is a pre-defined application-specific performance objective violated (triggered by RSM).

When a new application needs to be added to the framework, two design-time preparations for this application are required. First, as mentioned in Section 2, two mappings for each intra-application scenario of this application need to be explored at design time. These pre-optimized mappings will be stored in the SDB. Second, a standard function needs to be provided that helps the RSM to collect the appropriate application execution statistics and to decide whether or not the application violates its performance objective. Also, the user-defined application-specific performance objective and energy budget need to be stored in the system memory.

5.2. Experimental Results

5.2.1. Experiment Setup. In this subsection, we present several experimental results in which we investigate various aspects of our HTM algorithm. For our experiments, we use three typical multi-media applications: a Motion-JPEG (MJPEG) encoder, an MP3 decoder, and a Sobel filter for edge detection in images. These applications are denoted as A1, A2 and A3 respectively in Table II and Figure 7. 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. 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. Here, the 11 intra-application scenarios (execution modes) of MJPEG represent 11 different levels of computation complexity due to differences in the characteristics of the images being processed. For MP3, however, the intra-application scenarios are defined by the way of how the input music is decoded (left mono, right mono and stereo). The simultaneously active tasks are therefore different among the different intra-application scenarios of MP3. At design time, we have determined the optimal mappings for each intra-application scenario of each application targeting the two throughput objectives O_p and O_{pb}, as explained in Section 4.1. For all the three applications, there are 15 intra-application scenarios (MJPEG : 11, Sobel : 1 and MP3 : 3) in total. That means that we need to store 30 optimal mappings in system memory (i.e., the scenario database).

Table I. Parameters of NSGA-II

parameter	value
initial population size	256
generation size	256
generations	512
crossover probability	0.8
mutation probability	0.2

Table II. Studied application workload scenarios

Inter-application scenario	Workload scenario
A1	mjpeg 7
A2	sobel 0
A3	mp3 2
A1A2	mjpeg 7, sobel 0
A1A3	mjpeg 7, mp3 2
A2A3	sobel 0, mp3 2
A1A2A3	mjpeg 7, sobel 0, mp3 2

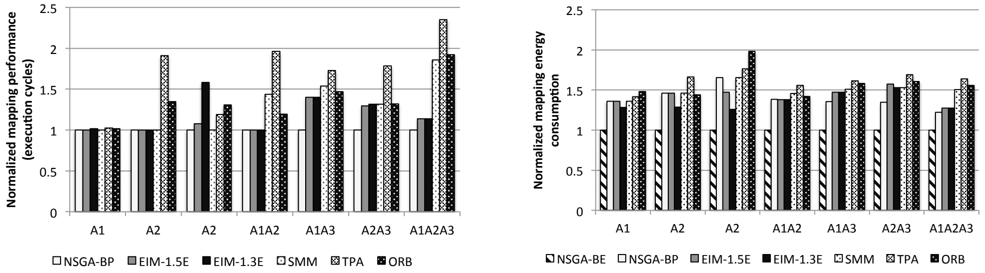
The parameters of the NSGA-II genetic algorithm we have used for design-time DSE are listed in Table I, which have been tuned for obtaining high-quality mappings for each workload scenario in our benchmark set.

With respect to the target architecture, we target a heterogeneous MPSoC containing 5 different processors, connected to a shared bus and memory. In this work, we assume that these 5 processors can execute all application tasks. However, we want to stress that our framework is not restricted to this assumption: dedicated processors could also be used in our framework. In that case, some additional information like the possible target processor of each task is needed for the RRS to derive a correct mapping. The architecture model also includes the required components for our run-time scheduling framework.

In the following experiments, we will evaluate the effectiveness of our HTM algorithm by studying how the run-time part of HTM (the EIM algorithm for mapping initialization and the STM algorithm for mapping customization) improves the mapping quality in terms of scenario execution time and scenario energy consumption.

5.2.2. Mapping Initialization. In this experiment, we compare the EIM algorithm to three different run-time mapping algorithms: Simple Mapping Merge (SMM) which simply merges together the (statically derived) optimal mappings of each active application for the corresponding intra-application scenario, Task Processor Affinity (TPA) which uses the affinity between tasks and processors to greedily determine a mapping without considering resource contention, and Output-Rate Balancing (ORB) [Castrillon et al. 2011] which aims at balancing the computation and communication load of each processor. Moreover, we also compare the run-time mapping results to the results of optimal mappings for each workload scenario. These optimal mappings have been statically determined by means of design-time DSE using the NSGA-II genetic algorithm.

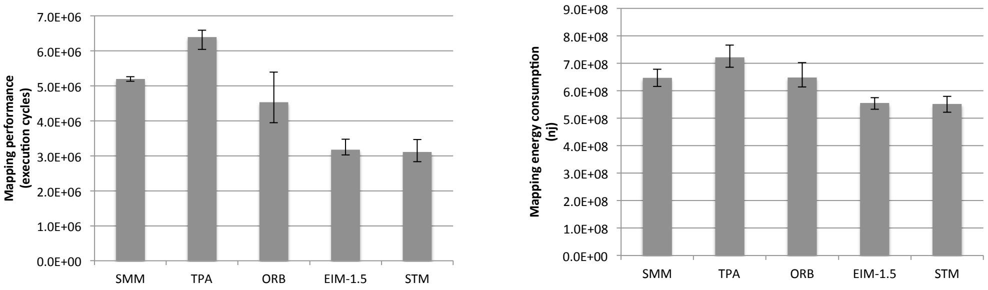
Here, we focus on those workload scenarios that have the heaviest computational demands, instead of all workload scenarios. These workload scenarios are listed in Table II, where the first column specifies the encoded (in terms of A1, A2 and A3) inter-application scenarios and the second column specifies the intra-application scenarios (labeled by the integer following the application name) used to form the workload scenario. For the scaling factor α of the energy budget in our EIM algorithm (see equation 5) we use the values 1.5 and 1.3 in our experiments.



(a) Performance of mappings from different algorithms

(b) Energy consumption of mappings from different algorithms

Fig. 7. Comparing the quality of mapping solutions derived from different run-time mapping algorithms for different inter-application scenarios



(a) Performance of mappings from different algorithms

(b) Energy consumption of mappings from different algorithms

Fig. 8. Comparing the quality of mapping solutions derived from different run-time mapping algorithms for the intra-application scenarios of A1A2A3

The experimental results are shown in Figure 7. In Figure 7(a), we compare the performance of the mappings resulting from the EIM, SMM, TPA, and ORB algorithms as well as from NSGA-II-based design-time DSE. The energy consumption of these mappings is shown in Figure 7(b). In these two figures, the bars of NSGA-BP and NSGA-BE respectively represent the mappings with best performance and minimal energy consumption found by the NSGA-II-based design-time DSE. These are used as a baseline for comparison. From Figure 7(a), we can see that our EIM algorithm in most cases produces a better mapping for the tested workload scenarios than the SMM, TPA and ORB algorithms. For the workload scenarios in which only a single application is active (i.e., bars for A1, A2 and A3) our EIM algorithm directly uses the mapping from design-time DSE, which results in a mapping performance that is very close or even equivalent to the optimal mapping. However, although the mappings have similar performance, they could still have a different energy consumption behaviour. In the case of our EIM algorithm, we use the energy budget in the search for an efficient mapping to limit the energy consumption of the resulting mapping. Consequently, and as shown in Figure 7(b), the EIM algorithm can yield mappings for single-application workload scenarios that are more energy efficient than the ones obtained by NSGA-BP.

In the workload scenarios with multiple simultaneously active applications, we can see that the EIM algorithm yields clear performance improvements compared to the

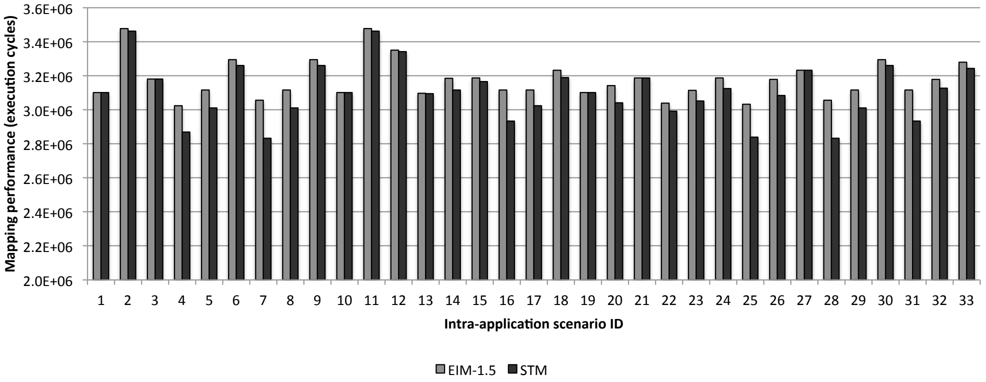


Fig. 9. Final mapping comparison of EIM and STM for all intra-application scenarios of A1A2A3

other three run-time mapping algorithms, especially in the case of workload scenario A1A2A3. By setting the parameter α of our EIM algorithm to different values, we can notice that in some workload scenarios, like A1, A3 and A2A3, the mapping performance with a higher energy budget is better than the one with a lower energy budget. However, in other workload scenarios, there is no such behaviour. This can be explained by the fact that for the latter workload scenarios the energy budget is high enough for the algorithm with a lower energy budget to find a mapping that is as good as the one found by EIM with a higher energy budget. In Figure 7(b), we can see that even if we have an energy budget in our EIM algorithm, the actual energy consumption of the final mapping may still exceed the energy budget: like for EIM-1.5E in the A2A3 workload scenario and for EIM-1.3E in a few other workload scenarios. This is caused by estimation inaccuracies of the energy model used in our algorithm. Even if the estimated energy consumption of a new mapping is under the predefined energy budget, the actual resulting system energy consumption after the remapping has taken place may still not fully satisfy our desired energy budget.

5.2.3. Mapping Customization. The experiment in this subsection shows the results of further mapping optimization by applying the mapping customization process. Figures 8.a and 8.b show the scenario execution time and energy consumption of mappings found by the SMM, TPA, ORB, EIM and STM algorithms in all the intra-application scenarios of a particular inter-application scenario, namely A1A2A3. In the case of the STM algorithm, the algorithm uses and tries to improve on the results of the EIM algorithm, as sketched in Figure 3. Using inter-application scenario A1A2A3, there are 33 workload scenarios in total that are considered as the application workload in this experiment. The error bars in the graphs show the variability of the results. From Figure 8.a, we can see that the mappings from our STM and EIM algorithm with a scaling factor $\alpha = 1.5$ achieve the best average performance among the investigated five algorithms. Note that the mapping customization process is applied during the execution of a certain workload scenario after the mapping initialization process. The STM algorithm is therefore used to further optimize the mapping solutions derived from the EIM algorithm. Comparing the results from STM and EIM, we found that the STM algorithm can achieve an additional performance improvement of 2.2% on average for all 33 considered intra-application scenarios. The reason for this relatively small performance improvement is twofold. First, the mapping derived by EIM for each new workload scenario is already a near optimal solution, which im-

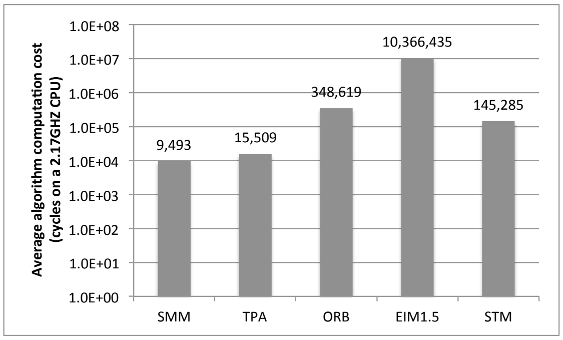


Fig. 10. Average algorithm computation cost (cycles on a 2.17GHZ CPU) for intra-application scenarios of A1A2A3

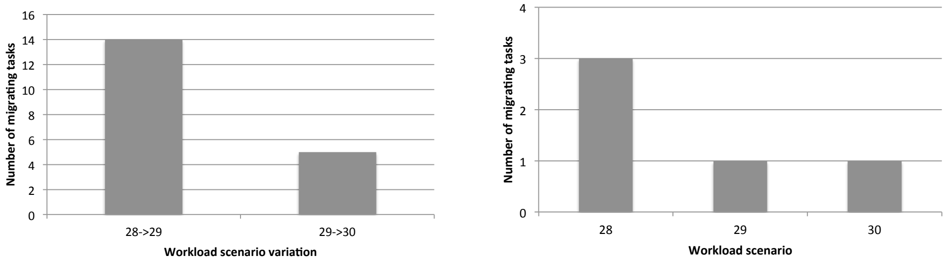
plies that the potentials for further improvement by the STM algorithm are limited. Second, the STM algorithm is designed for the situation in which the (user-defined) application-specific performance objective is violated. However, as the EIM algorithm can handle fine-grained application dynamism, such performance objective violations for an application will typically not be very large. When there is a (small) performance objective violation, the STM algorithm tries to find a new mapping by only making small changes to the old mapping in an on-the-fly manner. If the new mapping satisfies the pre-defined performance objective, then the STM algorithm will stop[4]

Figure 9 shows the details of the mapping performance comparison between the STM and EIM algorithms. Figure 8.b shows the average energy consumption of the final mappings as shown in Figure 8.a. The results in this figure illustrate that the mappings from our STM and EIM algorithms have the lowest average energy consumption, where the STM algorithm achieves an additional energy improvement of 0.5% on average compared to the EIM algorithm. Overall, in this experiment, we have demonstrated that the STM algorithm is capable of further improving the mapping performance compared with the EIM algorithm, without sacrificing the energy consumption of the mapping.

Compared with the mapping solutions derived from the SMM, TPA and ORB algorithms, the average performance and energy consumption of the final mapping solutions generated by applying the HTM algorithm (the mapping is first optimized by EIM followed by STM at run time) for the 33 workload scenarios in A1A2A3 improve by 67.2%, 105,9%, 45.9% (performance) and 14.6%, 23.5%, 14.9% (energy) on average respectively.

5.2.4. Algorithm Computation Cost. In this subsection, we investigate the computation cost of the run-time stage (EIM and STM) of our HTM algorithm and compare it to the overhead of the other run-time mapping algorithms. The results of an experiment in which we average the algorithmic overhead for executing different intra-application scenarios of A1A2A3 (the most complicated workload scenario where all three applications are active) are shown in Figure 10. Note that the time unit (cycles on a 2.17GHZ CPU) used in this figure is different from the time unit used for the mapping performance as presented in the previous figures, which are based on simulation cycles measured by the Sesame simulator. From Figure 10, we can see that the SMM approach has the smallest algorithmic cost. As in this approach there is no actual computation

[4] In case the violation cannot be remedied by the STM algorithm, the user can be notified and/or a strategy for graceful termination of applications could be used, but this is beyond the scope of this article.



(a) Task migration cost of EIM

(b) Task migration cost of STM

Fig. 11. Run-time task migration costs when applying the EIM and STM algorithms

of a new mapping, it just involves memory access time to retrieve the pre-optimized mapping from the SDB for each active application. On the other hand, the EIM part of our HTM algorithm has the heaviest computational cost to find a new mapping, which happens at the detection of a new workload scenario (mapping initialization). According to the cycle count for a 2.17GHZ CPU, the average computation time of the EIM algorithm is in the order of a few milliseconds, which is in our opinion still acceptable for the workload scenario initialization process. For the other part of HTM algorithm – the STM algorithm – the computation cost is much smaller.

5.2.5. Task Migration Cost. Here, we would like to give an intuition of the run-time cost in terms of the number of tasks that need to be migrated when applying our proposed algorithm. The actual time needed to perform these migrations highly depends on the mechanism used to implement such task migrations. A relatively simple implementation of task migration for our heterogeneous platform could be that each processor on the system keeps a copy of each task (the binary code compiled for the processor) or loads the corresponding task from shared memory when needed [Cannella et al. 2012]. When the scheduler decides to do task migration, it sends a stop and start command to the old processor and the new processor of the migrating task respectively, and also redirects the communication channels from the old processor to the new processor. As we are targeting the multi-media application domain, the task migration event could be triggered at the end of each input 'frame' which means the old processor will stop after having processed a whole input frame and the new processor will start to process the next new frame. In this case, there is no need to save and migrate the intermediate state of the migrating task. Other task migration mechanisms can also be implemented, but this is beyond the scope of this article.

In this experiment, the intra-application scenarios 28, 29 and 30 from Figure 9, in which the STM algorithm achieved larger performance improvements, will be considered as the target scenarios. In these three scenarios, the MP3 application has a different execution mode, while the execution modes of MJPEG and Sobel do not change. As mentioned before, the number of tasks in each scenario is 41. For the purpose of determining the migration costs for the STM algorithm, each scenario is executed 10 times before changing to the next scenario. The results of this experiment are shown in Figure 11, where Figure 11.a illustrates the number of tasks that need to be migrated by applying our EIM algorithm when the workload scenario changes and Figure 11.b shows the number of migrations caused by the STM algorithm during the execution of a certain workload scenario. We can see that the number of migrations due to our EIM algorithm can be relatively high, depending on the workload scenario. This is why we make the assumption in this article that each workload scenario will execute for a long enough time so that the system is able to benefit from our EIM algorithm. However, more research is needed on efficient migration implementations and effective migration policies (deciding whether or not to migrate) to improve the potential impact of run-time mapping algorithms in general. As can be seen from Figure 11.b, the migration cost due to the STM algorithm is quite low.

6. RELATED RESEARCH

In recent years, much research has been performed in the area of run-time task mapping for embedded systems. Singh et al. [Singh et al. 2013b] provide a nice survey of current and emerging trends for the task mapping problem on multi/many-core systems. In the context of performance optimization, the authors of [Chou and Marculescu 2008] propose a run-time mapping strategy that incorporates user behaviour information in the resource allocation process. An agent based distributed application mapping approach for large MPSoCs is presented in [Al Faruque et al. 2008]. The work of [Hölzenspies et al. 2008] proposes a run-time spatial mapping technique to map streaming applications onto MPSoCs. In [Brião et al. 2008], dynamic task allocation strategies based on bin-packing algorithms for soft real-time applications are presented. A run-time task allocator is presented in [Huang et al. 2011] that uses an adaptive task allocation algorithm and adaptive clustering approach for efficient reduction of the communication load. The approach proposed in [Schranzhofer et al. 2010] produces multiple mappings for each application with a trade-off between resource requirement and throughput. Mariani et al. [Mariani et al. 2010] 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. A similar approach is presented in [Singh et al. 2013a] targeting a generic architecture. In [Ykman-Couvreur et al. 2011], the authors propose a lightweight run-time manager, linked with an automated design-time exploration and incorporated in the host processor of the platform, to dynamically and efficiently configure the applications according to the available platform resources. Compared with these algorithms, our hybrid algorithm takes an application scenario-based approach, and takes computational and communication behaviour embodied in design-time optimized mappings into account when optimizing the mapping at run time.

Recently, Schor et al. [Schor et al. 2012] and Quan et al. [Quan and Pimentel 2013b] also proposed scenario-based run-time mapping approaches in which mappings derived from design-time DSE are stored for run-time mapping decisions. However, [Schor et al. 2012] does not address the reduction of mapping storage (all workload scenarios are stored) and does not dynamically optimize the mappings at run time. In [Quan and Pimentel 2013b], an approach is proposed in which mappings for inter-application scenarios are stored and used as a basis for run-time mapping decisions, after which a run-time algorithm aims at gradually further optimizing these mappings. The work presented in this article is based on the work of [Quan and Pimentel 2013b] but extends it in several directions in order to address several limitations of the latter work. A first limitation of the work in [Quan and Pimentel 2013b] is that it only works for homogeneous multi-processor systems. Another drawback is that the method from [Quan and Pimentel 2013b] needs to search for optimal mappings for *inter-application* scenarios at design time, which implies that it should already been known at design time which applications can execute on the target platform. For example, extending the system with a new application would require to redo the entire design-time DSE for all inter-application scenarios. In our approach, this problem is avoided by taking intra-applications as the basis for doing design-time DSE (i.e., performing DSE on applications in isolation) similar to the approach in [Quan and Pimentel 2013a]. However, in [Quan and Pimentel 2013a], the mapping is optimized only at the beginning of each new workload scenario. There is no run-time performance constraint set for each application, which means that the mapping is fixed during the execution of a certain workload scenario.

7. CONCLUSION

We have proposed a hybrid mapping algorithm, called HTM, for MPSoC-based embedded systems to improve their performance by capturing the dynamism of the application workloads executing on the system. Our approach is based on the idea of application scenarios and consists of three steps: design-time preparation, run-time mapping initialization and run-time mapping customization. The design-time preparation exploits optimal mappings for each mode of each application which will be stored on the target platform for further mapping optimization. At run time, the mapping initialization process dynamically optimizes the mapping of the running application(s) with the objective of maximizing the throughput under a predefined energy budget based on the optimal mappings of the corresponding applications stored on the system when a new workload scenario emerges. During the execution of a certain workload scenario, mapping customization is performed to further improve the performance of the mapping under an application-dependent objective. In various experiments, we have evaluated our algorithm and compared it with other run-time mapping algorithms. These experiments indicate that our proposed approach can achieve considerable performance improvements (45.9% - 105.9%) and energy savings (14.6% - 23.5%) compared with the other algorithms for workload scenarios in which multiple applications are simultaneously active.

REFERENCES

Mohammad Abdullah Al Faruque, Rudolf Krist, and Jörg Henkel. 2008. ADAM: Run-time Agent-based Distributed Application Mapping for On-chip Communication. In *Proceedings of the 45th Annual Design Automation Conference (DAC '08)*. ACM, New York, NY, USA, 760–765. DOI:http://dx.doi.org/10.1145/1391469.1391664

Eduardo Wenzel Brião, Daniel Barcelos, and Flávio Rech Wagner. 2008. Dynamic Task Allocation Strategies in MPSoC for Soft Real-time Applications. In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, New York, NY, USA, 1386–1389. DOI:http://dx.doi.org/10.1145/1403375.1403709

Emanuele Cannella, Onur Derin, Paolo Meloni, Giuseppe Tuveri, and Todor Stefanov. 2012. Adaptivity Support for MPSoCs Based on Process Migration in Polyhedral Process Networks. *VLSI Des.* 2012, Article 2 (Jan. 2012), 1 pages. DOI:http://dx.doi.org/10.1155/2012/987209

Jeronimo Castrillon, Rainer Leupers, and Gerd Ascheid. 2011. MAPS: Mapping Concurrent Dataflow Applications to Heterogeneous MPSoCs. *IEEE Trans.on Industrial Informatics* PP, 99 (2011), 1. DOI:http://dx.doi.org/10.1109/TII.2011.2173941

Chen-Ling Chou and R. Marculescu. 2008. User-Aware Dynamic Task Allocation in Networks-on-Chip. In *Design, Automation and Test in Europe, 2008. DATE '08*. 1232–1237. DOI:http://dx.doi.org/10.1109/DATE.2008.4484847

Cagkan Erbas, Selin Cerav-Erbas, and Andy D. Pimentel. 2006. Multiobjective Optimization and Evolutionary Algorithms for the Application Mapping Problem in Multiprocessor System-on-Chip Design. *IEEE Transactions on Evolutionary Computation* 10, 3 (2006), 358–374.

Stefan Valentin Gheorghita, Martin Palkovic, Juan Hamers, Arnout Vandecappelle, Stelios Mamagkakis, Twan Basten, Lieven Eeckhout, Henk Corporaal, Francky Catthoor, Frederik Vandeputte, and Koen De Bosschere. 2009. System-scenario-based Design of Dynamic Embedded Systems. *ACM Trans. Des. Autom. Electron. Syst.* 14, 1, Article 3 (Jan. 2009), 45 pages. DOI:http://dx.doi.org/10.1145/1455229.1455232

Philip K. F. Hölzenspies, Johann L. Hurink, Jan Kuper, and Gerard J. M. Smit. 2008. Run-time Spatial Mapping of Streaming Applications to a Heterogeneous Multi-processor System-on-chip (MPSoC). In *Proceedings of the Conference on Design, Automation and Test in Europe (DATE '08)*. ACM, New York, NY, USA, 212–217. DOI:http://dx.doi.org/10.1145/1403375.1403427

Jia Huang, A. Raabe, C. Buckl, and A. Knoll. 2011. A workflow for runtime adaptive task allocation on heterogeneous MPSoCs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*. 1–6. DOI:http://dx.doi.org/10.1109/DATE.2011.5763189

Gilles Kahn. 1974. The semantics of a simple language for parallel programming. In *Information processing*. North Holland, Amsterdam, 471–475.

Rakesh Kumar, Dean M. Tullsen, Parthasarathy Ranganathan, Norman P. Jouppi, and Keith I. Farkas. 2004. Single-ISA Heterogeneous Multi-Core Architectures for Multithreaded Workload Performance. In *Proceedings of the 31st annual international symposium on Computer architecture (ISCA '04)*. IEEE Computer Society, Washington, DC, USA, 64–. http://dl.acm.org/citation.cfm?id=998680.1006707

G. Mariani, P. Avasare, G. Vanmeerbeeck, C. Ykman-Couvreur, G. Palermo, C. Silvano, and V. Zaccaria. 2010. An industrial design space exploration framework for supporting run-time resource management on multi-core systems. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*. 196–201. DOI:http://dx.doi.org/10.1109/DATE.2010.5457211

JoAnn M. Paul, Donald E. Thomas, and Alex Bobrek. 2006. Scenario-oriented design for single-chip heterogeneous multiprocessors. *IEEE Trans. VLSI Syst.* 14, 8 (2006), 868–880. DOI:http://dx.doi.org/10.1109/TVLSI.2006.878474

Andy D. Pimentel, Cagkan Erbas, and Simon Polstra. 2006. A Systematic Approach to Exploring Embedded System Architectures at Multiple Abstraction Levels. *IEEE Trans. Computers* 55, 2 (2006), 99–112.

Wei Quan and A.D. Pimentel. 2013a. An iterative multi-application mapping algorithm for heterogeneous MPSoCs. In *Embedded Systems for Real-time Multimedia (ESTIMedia), 2013 IEEE 11th Symposium on*. 115–124. DOI:http://dx.doi.org/10.1109/ESTIMedia.2013.6704510

Wei Quan and Andy D. Pimentel. 2013b. A Scenario-based Run-time Task Mapping Algorithm for MPSoCs. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 131, 6 pages. DOI:http://dx.doi.org/10.1145/2463209.2488895

Lars Schor, Iuliana Bacivarov, Devendra Rai, Hoeseok Yang, Shin-Haeng Kang, and Lothar Thiele. 2012. Scenario-based Design Flow for Mapping Streaming Applications Onto On-chip Many-core Systems. In *Proceedings of the 2012 International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES '12)*. ACM, New York, NY, USA, 71–80. DOI:http://dx.doi.org/10.1145/2380403.2380422

Andreas Schranzhofer, Jian-Jian Chen, and Lothar Thiele. 2010. Dynamic Power-Aware Mapping of Applications onto Heterogeneous MPSoC Platforms. *Industrial Informatics, IEEE Transactions on* 6, 4 (2010), 692–707. DOI:http://dx.doi.org/10.1109/TII.2010.2062192

Amit Kumar Singh, Akash Kumar, and Thambipillai Srikanthan. 2013a. Accelerating Throughput-aware Runtime Mapping for Heterogeneous MPSoCs. *ACM Trans. Des. Autom. Electron. Syst.* 18, 1, Article 9 (Jan. 2013), 29 pages. DOI:http://dx.doi.org/10.1145/2390191.2390200

Amit Kumar Singh, Muhammad Shafique, Akash Kumar, and Jörg Henkel. 2013b. Mapping on multi/many-core systems: survey of current and emerging trends. In *Proceedings of the 50th Annual Design Automation Conference (DAC '13)*. ACM, New York, NY, USA, Article 1, 10 pages. DOI:http://dx.doi.org/10.1145/2463209.2488734

Wei Sun and Tomoyoshi Sugawara. 2011. Heuristics and Evaluations of Energy-Aware Task Mapping on Heterogeneous Multiprocessors. In *Parallel and Distributed Processing Workshops and Phd Forum (IPDPSW), 2011 IEEE International Symposium on*. 599–607. DOI:http://dx.doi.org/10.1109/IPDPS.2011.209

P. van Stralen and A. Pimentel. 2010a. Scenario-based design space exploration of MP-SoCs. In *Computer Design (ICCD), 2010 IEEE International Conference on*. 305–312. DOI:http://dx.doi.org/10.1109/ICCD.2010.5647727

P. van Stralen and A.D. Pimentel. 2010b. A trace-based scenario database for high-level simulation of multimedia MP-SoCs. In *Embedded Computer Systems (SAMOS), 2010 International Conference on*. 11–19. DOI:http://dx.doi.org/10.1109/ICSAMOS.2010.5642097

C. Ykman-Couvreur, P. Avasare, G. Mariani, G. Palermo, C. Silvano, and V. Zaccaria. 2011. Linking run-time resource management of embedded multi-core platforms with automated design-time exploration. *Computers Digital Techniques, IET* 5, 2 (2011), 123–135. DOI:http://dx.doi.org/10.1049/iet-cdt.2010.0030

Received xx xxxx; revised xx xxxx; accepted xx xxxx