Energy-efficient Runtime Resource Management for Adaptable Multi-application Mapping

Robert Khasanov*, Jeronimo Castrillon†
Chair for Compiler Construction, Technische Universität Dresden, Germany
Email: *robert.khasanov@tu-dresden.de, †jeronimo.castrillon@tu-dresden.de

***Abstract*—Modern embedded computing platforms consist of a high amount of heterogeneous resources, which allows executing multiple applications on a single device. The number of running application on the system varies with time and so does the amount of available resources. This has considerably increased the complexity of analysis and optimization algorithms for runtime mapping of firm real-time applications. To reduce the runtime overhead, researchers have proposed to pre-compute partial mappings at compile time and have the runtime efficiently compute the final mapping. However, most existing solutions only compute a fixed mapping for a given set of running applications, and the mapping is defined for the entire duration of the workload execution. In this work we allow applications to adapt to the amount of available resources by using mapping segments. This way, applications may switch between different configurations with varied degree of parallelism. We present a runtime manager for firm real-time applications that generates such mapping segments based on partial solutions and aims at minimizing the overall energy consumption without deadline violations. The proposed algorithm outperforms the state-of-the-art approaches on the overall energy consumption by up to 13% while incurring an order of magnitude less scheduling overhead.**

***Index Terms*—energy-efficiency, runtime systems, scheduling**

I. Introduction

Most modern computing systems are embedded in end-user devices. Some of these systems consist of many-cores, and the number of cores have already reached the thousands [1]. Many-cores allow executing multiple applications in parallel on a single device. These applications are launched at any time, making resource management particularly challenging.

The problem of optimally executing multiple applications is well-known in the embedded domain. Existing solutions might be classified into design-time, runtime and hybrid [2]. The latter combines the benefits of the first two approaches. In a hybrid mapping, the decision where to execute an application is split between design- and runtime. At design time, partial solutions for each application are generated by analyzing the applications in isolation. This is done, e.g., using established methodologies for Design Space Exploration (DSE). At runtime, the runtime manager (RM), being aware of the overall workload, transforms these solutions into final mappings. Hybrid mappings thus benefit from extensive design space exploration to find near-optimal partial solutions, and can adapt to the workload at runtime via efficient heuristics.

In runtime and hybrid approaches, when a new application arrives, the RM has to assign resources to it. An incremental RM allocates the new application on free resources [3], [4]. If available resources do not suffice, the RM either rejects the application, or uses fast heuristics to remap existing jobs. For instance in [5], [6], a joint mapping is computed for all applications at once. Authors formulate the problem as multi-choice multidimensional knapsack problem (MMKP) [7] and solve it using fast heuristics. Similarly, when an application finishes execution, more resources become available and the RM can generate new mappings. State-of-the-art solutions, as discussed above, generate *fixed* mappings, i.e., mappings that do not change during the execution of the applications. Those fixed mappings are optimized *locally* for the duration of the fixed set of applications, and at the subsequent RM activations, new local optimal mappings will be generated. However, a sequence of local optimal decisions might lead to a sub-optimal schedule in the scope of the entire runtime of the system.

Recently, Niknafs et al. [8] attempted to enlarge the scope of analysis for firm real-time applications by computing joint mappings. In the generated schedules, different applications might be mapped to the same core and executed according to the earliest deadline first (EDF) policy. Applications might be also preempted in favour of more critical and proactively predicted jobs. However, the presented approach is limited to single-threaded applications. In this paper we generalize Niknafs' approach to multi-threaded applications. We express the schedules as fragmented into mapping segments, which explicitly express resource adaptations (Section IV). We propose a fast algorithm for firm real-time multi-threaded applications, which analyzes the applications in the scope until the last application finishes, and generates the mapping segments optimized w.r.t. the overall energy consumption (Section V). Due to the enlarged scope of the analysis, the generated schedules will be near-optimal for the entire execution of the current workload (Section VI).

II. Related work

The problem of optimal execution of multiple applications is well-known. The common aim is to increase the overall throughput by using historical data [9], [10], or by exploiting the concavity of throughput of multi-threaded applications [11]. These works do not optimize energy efficiency.

Several runtime managers for energy efficient execution of applications exist. Das et al. [12] uses energy-awareness of single applications to improve both energy consumption and

Table I: Request parameters.

Req.	App.	S1-Arr.	S1-Dead.	S2-Arr.	S2-Dead.
σ_1	λ_1	0	9	0	9
σ_2	λ_2	1	5	1	4

Table II: Application parameters.

		λ_1, pr. 0% - 18.87% - 62.08%		λ_2, pr. 0%	
#L	#B	$\tau[s]$	$\xi[J]$	$\tau[s]$	$\xi[J]$
1	0	16.8 - 13.63 - 6.37	7.90 - 6.41 - 3.00	10.0	2.00
2	0	10.3 - 8.36 - 3.91	7.01 - 5.69 - 2.66	7.0	2.87
0	1	11.2 - 9.09 - 4.25	18.54 - 15.04 - 7.03	5.0	7.55
0	2	6.3 - 5.11 - 2.39	17.70 - 14.36 - 6.71	3.5	10.5
1	1	8.1 - 6.57 - 3.07	10.90 - 8.84 - 4.13	3.5	6.44
1	2	7.9 - 6.41 - 3.00	10.60 - 8.60 - 4.02	3.0	6.81
2	1	5.3 - 4.30 - 2.01	8.90 - 7.22 - 3.38	3.0	5.73
2	2	4.7 - 3.81 - 1.78	11.00 - 8.92 - 4.17	2.0	6.58

thermal dissipation. Greedy heuristics are used for thread allocations, and reinforcement learning is employed for selecting the minimum frequency. Tzilis et al. [13] uses profiling data to predict the performance and energy consumption of a single-threaded application co-scheduled with another applications, and decides on application placement and frequency settings. Similarly, Libutti et al. [14] exploits collected off-line information such as CPU demands and memory sensitivity for job co-scheduling. Libutti et al. target multi-threaded applications but without controlling frequency settings. Singh et al. [15] capture the trace information of individual applications at design-time, and then merge execution intervals of multiple applications at runtime. This approach is limited to Synchronous Data Flow Graphs (SDFG) and thus cannot be applied to dynamic workloads.

Unlike aforementioned works, hybrid approaches prepare a set of partial or complete mappings of individual applications at design-time, which correspond to Pareto-optimal configurations. Ascia et al. [16] were one of the first to propose multi-objective mapping generation, using evolutionary algorithms to find Pareto-optimal points. Approaches such as [3], [17]–[19] use heuristics for efficient exploration of Pareto-optimal configurations, while others such as [4], [20] use evolutionary algorithms. The amount of considered configuration can be also reduced by exploiting system symmetries [21], [22].

Identified Pareto-optimal points at design time are passed to the RM. The essential task of the RM is to select the operating points for multiple applications. Singh et al. [3] iteratively map applications onto the platform, whereas Weischlgartner et al. [4] enhance the iterative binding of application with a repair heuristic. More sophisticated approaches express the problem as a multiple-choice multidimensional knapsack problem (MMKP) [7]. Ykman-Couvreur et al. [5], for instance, propose a fast heuristic, which expresses the resource demands of operating points as a single value, and then use a greedy algorithm to solve the MMKP. This heuristic underlies the solutions proposed in [17], [20]. Wildermann et al. [6], [23] solve the problem by applying a Lagrangian relaxation method. Shojaei et al. [24] propose a compositional Pareto-algebraic heuristic using Pareto-algebra. However, these algorithms assume that applications are constantly running, and do not consider application reconfiguration. As mentioned in Section I, Niknafs et al. [8] do consider application reconfiguration, but limited to single-threaded applications.

III. Motivational Example

Assume a heterogeneous multi-core device with 2 little and 2 big cores that serves requests arriving according to Scenario S1 in Table I. Each request consists of the application to run, its arrival and (absolute) deadline. Table II describes configurations of the two applications of the example (λ_1, λ_2), characterized by the number of little and big cores, the execution time τ and the energy consumption ξ. For λ_1 we show time/energy values in triples, in which the first (initial) state is followed by the states states with a progress ratio of 18.87% and 62.08% correspondingly, to which we refer below. The values in the table are synthetic, but feature ratios similar to what we observed in real applications (see Section VI).

At $t = 0$, the RM receives request σ_1 to execute application λ_1. An energy-optimizing mapper decides to map it to 2 little and 1 big cores (i.e., `2L1B`), since this configuration meets the deadline ($t = 9$) with the least energy consuming (8.9 J, underlined). After 1 s, request σ_2 arrives while request σ_1 progressed to $\approx 18.87\%$. To meet the deadline, σ_2 must be executed either as `2B`, `1L1B`, `1L2B`, `2L1B` or `2L2B`. If any of these mappings is chosen, σ_1 must then continue as `1L`, `2L`, `1L1B` or `1B`. A mapper that only explores fixed mappings would choose one where both jobs meet their deadlines, e.g., `1L1B` for both σ_1 and σ_2. By $t = 4.5$, σ_2 finishes its execution and σ_1 progresses to $\approx 62.08\%$. If σ_1 continues as `1L1B` till the end, as depicted in Fig. 1(a), the overall energy consumption is 16.96 J. If the RM decides to remap application at $t = 4.5$, then it would choose the most efficient mapping `2L`, see Fig. 1(b), with overall energy consumption of 15.49 J. However, if at $t = 1$ the RM runs σ_2 on `2L1B` and suspends momentarily σ_1, then when σ_2 finishes, σ_1 may continue with `2L1B` leading to an overall energy consumption of 14.63 J (see Fig. 1(c)).

Assume now the tighter Scenario S2 in Table I. At $t = 1$, σ_2 can only choose `1L2B`, `2L1B`, or `2L2B`, which leaves at most either 1 big or 1 little core for σ_1. Since these configurations

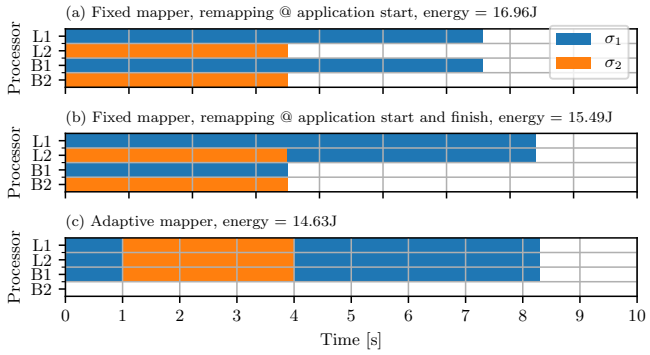

Figure 1: Three resource management scenarios.

cannot meet the deadline, a fixed mapper will be unable to find a schedule and σ_2 will be rejected. With explicit adaptations, a dynamic mapper in the RM can produce the schedule in Fig. 1(c) and meet the constraints. By allowing mapping reconfigurations and global analysis scope, the overall energy consumption and the request acceptance rate can be improved.

IV. System model and problem definition

To formalize the resource management problem illustrated above, assume a heterogeneous platform with m resource types, and core counts represented by the vector $\vec{\Theta} = (\Theta_1, \ldots, \Theta_m)$. The platform executes multi-threaded applications, in which each thread perform computations during the whole execution of the application. Additionally, we assume that in each fixed configuration all threads process the workload with a constant progress rate. Via a DSE method or benchmarking, the RM is given information about each application λ and its N_λ operating points (cf. Table II). Each operating point consists of needed resources $\vec{\theta}$, the (worst case) execution time τ and the energy consumption ξ, i.e., $c_\lambda^j = \langle \vec{\theta}, \tau, \xi \rangle$. Operating points are assumed to be already Pareto-filtered, i.e., each operating point is better than any other in at least one parameter.

Every time a request arrives, the RM is activated. We denote by $\Sigma_{t'}$ the set of unfinished jobs admitted in $t < t'$ plus the newly arrived job at time t'. For a given job, $\sigma[\alpha]$ denotes the arrival time, $\sigma[\delta]$ the (absolute) deadline, $\sigma[\lambda]$ the application, and $\sigma[\rho] \in [0, 1]$ the remaining progress ratio of the job, i.e., $\sigma = \langle \alpha, \delta, \lambda, \rho \rangle$. Given a set of requests $\Sigma_{t'}$, the RM attempts to find a schedule κ, which is defined as a list of mappings μ defined on consecutive time segments:

$$\kappa = \{\mu_i \times \Delta_{\mu_i}\}, \qquad 0 \leq i < N \tag{1}$$

where N is a number of segments, $\Delta_\mu = [\underline{\Delta_\mu}, \overline{\Delta_\mu})$ is a time interval of the mapping μ, $\underline{\Delta_{\mu_0}} = t'$ and $\overline{\Delta_{\mu_i}} = \underline{\Delta_{\mu_{i+1}}}$, $0 \leq i < N - 1$. Each mapping μ contains individual job mappings $\nu = \langle \sigma, \lambda, j \rangle$, which expresses that application λ of request σ runs with configuration j. Note that this formulation allows jobs to be remapped from one configuration to another in the schedule.

Having introduced the notation, the optimization problem can be defined as:

$$\text{minimize} \sum_{\mu \in \kappa} \sum_{\nu \in \mu} c_{\nu[\lambda]}^{\nu[j]}[\xi] \frac{|\Delta_\mu|}{c_{\nu[\lambda]}^{\nu[j]}[\tau]} \tag{2a}$$

$$\text{subject to} \sum_{\nu \in \mu} c_{\nu[\lambda]}^{\nu[j]}[\vec{\theta}] \leq \vec{\Theta}, \qquad \forall \mu \in \kappa \tag{2b}$$

$$\nu_1[\sigma] \neq \nu_2[\sigma], \qquad \forall \nu_1, \nu_2 \in \mu, \nu_1 \neq \nu_2 \tag{2c}$$

$$\sum_{\substack{\mu \in \kappa \\ \nu \in \mu \\ \nu[\sigma] = \sigma}} \frac{|\Delta_\mu|}{c_{\nu[\lambda]}^{\nu[j]}[\tau]} = \sigma[\rho], \qquad \forall \sigma \in \Sigma \tag{2d}$$

$$\max_{\substack{\mu \in \kappa \\ \nu \in \mu \\ \nu[\sigma] = \sigma}} \overline{\Delta_\mu} \leq \sigma[\delta], \qquad \forall \sigma \in \Sigma. \tag{2e}$$

Constraint (2b) ensures that the resources required for mapping in the segment μ does not exceed the available resources per type $\vec{\Theta}$. Equation (2c) specifies that at each mapping segment at most one job mapping relates to each request σ. Constraint (2d) expresses that each job will run until the end. Finally, each job must finish the execution before the deadline (2e). If there is a feasible solution to this problem, the RM admits the request and changes the schedule accordingly. Otherwise the request is rejected.

V. Fast heuristic for MMKP-based scheduling

Algorithm 1 MMKP-MDF mapping heuristic.

Input: Set of jobs Σ_t, platform $\vec{\Theta}$, application table c
Output: The schedule κ

1: $\vec{J} \leftarrow \vec{\Theta} \times \max(\sigma[\delta] - t \mid \sigma \in \Sigma_t)$
2: **for all** $\sigma \in \Sigma_t$ **do** $jc[\sigma] \leftarrow \varnothing$
3: **while** $\exists \sigma \in \Sigma_t : jc[\sigma] = \varnothing$ **do**
4: $\quad \sigma^*, cl \leftarrow$ NextJobMDF$(\Sigma_t, jc, \vec{J}, c)$
5: $\quad$ **while** $jc[\sigma^*] = \varnothing$ **do**
6: $\quad\quad$ **if** $|cl| = 0$ **then return** $\varnothing$
7: $\quad\quad j^* \leftarrow \text{argmin}_{j \in cl}\{c^j_{\sigma^*[\lambda]}[\xi]\}$
8: $\quad\quad jc^* \leftarrow jc$, $jc^*[\sigma^*] \leftarrow j^*$
9: $\quad\quad \kappa^* \leftarrow$ ScheduleJobs$(\Sigma_t, jc^*, \vec{\Theta}, c)$
10: $\quad\quad$ **if** $\kappa^* \neq \varnothing$ **then**
11: $\quad\quad\quad jc \leftarrow jc^*$, $\kappa \leftarrow \kappa^*$
12: $\quad\quad\quad \vec{J} \leftarrow \vec{J} - c^{j^*}_{\sigma^*[\lambda]}[\vec{\theta}] \times c^{j^*}_{\sigma^*[\lambda]}[\tau] \times \sigma^*[\rho]$
13: $\quad\quad$ **else**
14: $\quad\quad\quad cl \leftarrow cl \setminus j^*$
15: **return** κ

We consider the core types as knapsacks with certain *capacities*, and the job configurations c_λ^j as *items* with certain *weights*. Weights are defined as the required processing time to finish the job $c_\lambda^j[\tau]$ times the required number of resources of corresponding type $c_\lambda^j[\vec{\theta}]$, while the capacities express the available processing time per each resource type. Each job forms a group of items, in which exactly one item must be chosen. After negating the energy *values* of each item, the optimization goal becomes to maximize the overall (negative) value which can be expressed as a multiple-choice multi-dimensional knapsack problem (MMKP) [7]. Algorithm 1 describes our heuristic to solve this. It generalizes the solution in [8] for multi-threaded applications.

At each activation of the RM, the containers $\vec{J}$ are initialized with the overall processing time per resource type limited by the largest job deadline (the time scope of the analysis), which is followed by the initialization of the found job configuration dictionary jc (lines 1-2). The algorithm iterates over unmapped jobs (line 3), using NextJobMDF to select the next one to map (line 4). Similarly to [8] this function (i) filters configurations by checking whether they can meet deadlines and fit the containers $\vec{J}$, (ii) determines a job, in which the difference between the most energy-efficient feasible configuration and the second best one is maximized, i.e., Maximum Difference

First (MDF), and (iii) returns the selected job σ^* along with a list of filtered configurations cl. The MDF policy prioritizes the job that would cause the highest degradation if the best point is not chosen in this iteration. In lines 5-14, the algorithm iterates over the configurations in non-decreasing order of energy consumption. It then attempts to schedule the job σ^* with already mapped jobs in SCHEDULEJOBS, detailed in Algorithm 2. Once the job is successfully scheduled, its configuration and a new schedule is saved, and the containers $\vec{J}$ are updated (lines 11-12). Otherwise, no feasible schedule was found and the algorithm exits in line 6.

Algorithm 2 Schedule jobs.

Input: Set of jobs Σ_t, their configurations jc, platform $\vec{\Theta}$, application table c
Output: The schedule κ

1: $\widetilde{\Sigma} \leftarrow \{\sigma \in \Sigma_t \mid jc[\sigma] \neq \varnothing\}$, $\kappa \leftarrow \varnothing$, $t^e \leftarrow t$
2: **while** $\left|\widetilde{\Sigma}\right| \neq 0$ **do**
3: $\quad \sigma^* \leftarrow \text{argmin}_{\sigma \in \widetilde{\Sigma}}\{\sigma[\delta]\}$
4: $\quad j^* \leftarrow jc[\sigma^*]$, $\rho^* \leftarrow \sigma^*[\rho]$
5: $\quad$ **for all** $\mu \times \Delta \in \kappa$ **do**
6: $\quad\quad \vec{\theta^*} \leftarrow \sum_{\nu \in \mu} c^{\nu[j]}_{\nu[\lambda]}[\vec{\theta}]$
7: $\quad\quad$ **if** $\neg(c^{j^*}_{\sigma^*}[\vec{\theta}] + \vec{\theta^*} \leq \vec{\Theta})$ **then continue**
8: $\quad\quad r \leftarrow c^{j^*}_{\sigma^*[\lambda]}[\tau] \times \rho^*$
9: $\quad\quad$ **if** $r \geq |\underline{\Delta}|$ **then**
10: $\quad\quad\quad \mu \leftarrow \mu \cup \{\nu\langle\sigma^*, \sigma^*[\lambda], j^*\rangle\}$
11: $\quad\quad\quad \rho^* \leftarrow \rho^* - \frac{|\underline{\Delta}|}{c^{j^*}_{\sigma^*[\lambda]}[\tau]}$
12: $\quad\quad$ **else**
13: $\quad\quad\quad \mu_1 \times \Delta_1, \mu_2 \times \Delta_2 \leftarrow \text{SPLIT}(\mu \times \Delta, \underline{\Delta} + r\ )$
14: $\quad\quad\quad \mu_1 \leftarrow \mu_1 \cup \{\nu\langle\sigma^*, \sigma^*[\lambda], j^*\rangle\}$
15: $\quad\quad\quad \kappa \leftarrow (\kappa \setminus \{\mu \times \underline{\Delta}\}) \cup \{\mu_1 \times \Delta_1, \mu_2 \times \Delta_2\}$
16: $\quad\quad\quad \rho^* \leftarrow 0$, $t^f \leftarrow \overline{\Delta_1}$
17: $\quad\quad\quad$ **break**
18: $\quad\quad$ **if** $\rho^* = 0$ **then** $t^f \leftarrow \overline{\Delta}$, **break**
19: $\quad$ **if** $\rho^* \neq 0$ **then**
20: $\quad\quad r \leftarrow c^{j^*}_{\sigma^*[\lambda]}[\tau] \times \rho^*$, $\Delta \leftarrow [t^e, t^e + r)$
21: $\quad\quad \mu \leftarrow \{\nu\langle\sigma^*, \sigma^*[\lambda], j^*\rangle\}$, $\kappa \leftarrow \kappa \cup \{\mu \times \Delta\}$
22: $\quad\quad t^e \leftarrow t^e + r$, $\rho^* \leftarrow 0$, $t^f \leftarrow \overline{\Delta}$
23: $\quad$ **if** $t^f > \sigma^*[\delta]$ **then return** $\varnothing$
24: $\quad \widetilde{\Sigma} \leftarrow \widetilde{\Sigma} \setminus \sigma^*$
25: **return** κ

Algorithm 2 takes the job configurations as input and generates a feasible schedule on. Line 1 initialize the algorithm. The algorithm iterates over unmapped jobs (lines 2-24) in non-decreasing order of their deadlines, i.e., Earliest Deadline First (EDF) (line 3). The loop in lines 5-18 schedules the job on already constructed mapping segments in the ascending order of the time segments (with a slight abuse of notation). After checking resource constraints on the segment (line 7), the algorithm checks whether the job will execute during the whole segment (lines 10-11) or only a part of it (lines 13-17). In the last case, the mapping segment is split at time the job finishes execution (line 13), and the job is added only to the first part of it. To track the remaining progress rate of the job while iterating the mapping segments, the algorithm initialize ρ^* in line 4 and updates it in lines 11 and 16. If the job is not finished after the last mapping segment, the new mapping segment is created and added to the schedule (lines 19-22). At the end, line 23 verifies that the job meets its deadline. As a result, the algorithm puts more time-critical jobs into the earliest mapping segments as possible (EDF policy).

Note that the proposed algorithm is backward-compatible with the single-threaded version of the algorithm (without predictions) [8], and the generated schedules will be the same due to MDF and EDF policies. At the same time, due to a constraint to schedule all the threads on the same mapping segments, the original single-threaded algorithm cannot be employed for multi-threaded applications.

Table III: Amount of test cases differentiated by a number of jobs and deadline level.

Deadline level \ # Jobs	1	2	3	4
Weak	15	255	255	230
Tight	35	340	340	206

VI. EVALUATION

This section describes the experimental setup, the generation of the experimental workload and the alternative algorithms. We evaluate the scheduling success rate, the energy-efficiency of found schedules, and the overhead of our approach.

A. Experimental setup and test generation

In our experiments we used three different dataflow applications from the automotive and multimedia domains: an algorithm of *speaker recognition* with 8 processes [25], *audio filter*, a stereo frequency filter with 8 processes [21], and an algorithm of *pedestrian recognition* with 6 processes, provided by Silexica. To obtain application configurations we exhaustively benchmarked these applications with input data of different sizes on the Hardkernel Odroid XU4 featuring an Exynos 5422 big.LITTLE chip with four Cortex-A15 and four Cortex-A7 cores, fixed at frequencies of 1.8 GHz and 1.5 GHz respectively. We measured the power consumption of the Odroid-XU4 board using ZES Zimmer LMG450 Power Analyzer connected to DC input with an external readout rate of 20 Sa/s. To identify Pareto-optimal configurations we executed the variants 50 times to get average execution times and energy consumptions. In this way we obtained 36 Pareto-configurations for audio filter, 35 for pedestrian recognition, and 28 for speaker recognition.

The multi-application setup consists of 1676 test cases. Each test has one to four jobs, which are characterized by the current progress ratio and the remaining deadline. 31.9% of the test cases consist of requests of a single application (uniformly distributed among each application and input data), while the remaining 68.1% are application mixes. In around 22.6% of

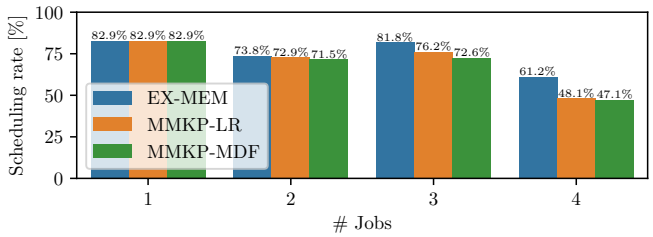

Figure 2: Scheduling rate of different schedulers for test cases with tight deadlines.

the tests we set the progress state of the jobs to zero (initial state). For all others, we randomly choose a progress rate in the range $0 - 0.9$ except for the first job, which naturally starts in the initial state. To set deadlines, we randomly select a configuration, calculate the remaining time to finish the job using this configuration and then scale it by a factor. In the case of weak deadlines, we randomly choose large factors in the range $2 - 6$. For tighter deadlines factors are selected in the range $0.6 - 2$ at random. Table III reports the amount of tests for each pair of number of jobs and deadline level.

Our proposed algorithm **MMKP-MDF** (Section V) was implemented in Python 3 and runs on a 3.20 GHz Intel Core i5-6500 CPU. Our so implemented RM prototype receives the Pareto-optimal configurations of the applications, reads a test case, and maps the applications on the Odroid XU4 platform. We implemented alternative algorithms in the RM prototype to evaluate our solution. **EX-MEM** exhaustively checks all possible mappings for each of the mapping segments. In each constructed mapping segment it cuts the segment on the shortest job, and generates the next mapping segment. To accelerate the algorithm, we use memoization by storing and re-using the best energy consumption for a given current state (a pair of jobs, their progress rates, and time). **MMKP-LR** is based on the Lagrangian Relaxation algorithm described in [6]. This method solves Lagrangian relaxations of the MMKP problem using a subgradient method (limited by 100 iterations), and iteratively maps applications in the increasing order of the minimum configuration costs. Similarly, during job mapping, the algorithm iteratively checks the configurations in the increasing order of their cost. A configuration is mapped if there are enough resources and the job can meet the deadline either using this configuration till the end, or reconfigured to another configuration at the end of the mapping segment (optimistic check). This process is repeated for the next mapping segment. Thus, the analysis scope is limited to a single mapping segment.

B. Scheduling rate and energy-efficiency

We evaluated all three implemented algorithms w.r.t the percentage of test cases they could find a feasible schedule. All algorithms scheduled 100% of the test cases with weak deadlines. The results are completely different for the tests with tight deadline as shown in Fig. 2. For test cases with one or two jobs, all three schedulers feature a similar scheduling success rate, with a difference within 2.3%. For the tests with more jobs, EX-MEM shows significantly higher rate than the other two algorithms, up to 14.1%. In all test cases, MMKP-LR and our MMKP-MDF achieve a similar scheduling success rate, with a difference within 3.6% in favour of MMKP-LR.

Table IV: Geometric mean of the relative energy consumption compared to EX-MEM.

	MMKP-LR		**MMKP-MDF**	
# Jobs	Weak	Tight	Weak	Tight
1	1.0000	1.0000	1.0000	1.0000
2	1.0480	1.1291	1.0003	1.0682
3	1.1534	1.2250	1.0031	1.0978
4	1.2648	1.3404	1.0099	1.0618
Overall	**1.1452**	**1.1923**	**1.0042**	**1.0756**
(all levels)	**1.1665**		**1.0356**	

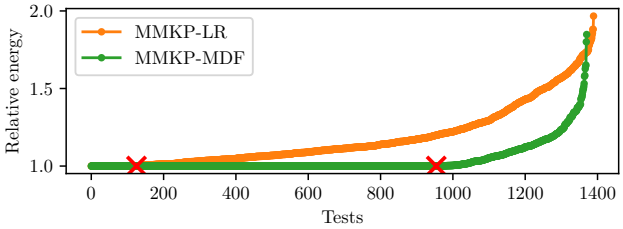

Figure 3: S-curves of the relative energy consumption compared to EX-MEM (lower is better).

In terms of energy efficiency, we compared the algorithms to the optimal solutions obtained by EX-MEM. For each successfully scheduled test case, we compute the relative energy consumption compared to EX-MEM, and report the geometric mean of these values for each test group in Table IV. All schedulers generate optimal schedules in case of a single job. For tests with weak deadlines, the relative energy consumption of MMKP-MDF schedules increases slowly from 0.03% for two jobs till 0.99% for four jobs (in geometric mean), and over all tests with weak deadline, the schedules are off by 0.42% from the optimal one. For tighter deadlines, the relative energy consumption of MMKP-MDF varies nonmonotonically with the number of jobs, and they are off by 7.56% in geometric mean. For MMKP-LR, the relative energy consumption increases with the number of jobs, and overall the geometric mean of these values are 14.52% and 19.23% for weak and tight deadlines, correspondingly. Overall, MMKP-MDF generates more energy-efficient schedules by 13.1% than MMKP-LR. Fig. 3 presents S-curves of relative energy consumption over all tests. As we see, MMKP-MDF generates optimal schedules for 954 tests (69.6% of successfully scheduled), while MMKP-LR only for 125 tests (9.0%).

C. Search time

Fig. 4 shows the boxplots and the average values of the execution times of different algorithms differentiated by the number of jobs. As we see in the figure, the scheduling time increases with the number of jobs for all three implementations. As expected, EX-MEM displays an exponential

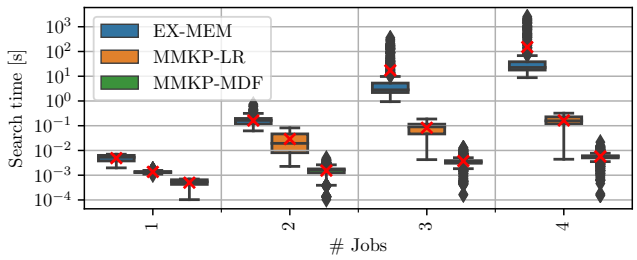

Figure 4: Box plots and the average values summarizing the scheduling overhead for different algorithms.

growth, with an average of 152 s to schedule four jobs, while the median and the worst-case values are 22.65 s and 2550 s (≈ 37.5 min) correspondingly. The scheduling time of MMKP-LR and MMKP-MDF grows less rapidly. MMKP-LR needs around 1.3 ms to schedule one job and around 163 ms for four jobs. MMKP-MDF is significantly faster, requiring only 5.7 ms in average for four jobs with a worst-case of 21.6 ms.

To summarize, our proposed MMKP-MDF algorithm achieves comparable scheduling success rate as the MMKP-LR scheduler, while outperforming it in terms of overall energy efficiency and scheduling overhead. MMKP-MDF schedules applications within 21.6 ms, which makes it a good candidate to implement in a fully-functional resource runtime manager. As mentioned in the experimental setup, the overhead analysis for all schedulers was performed on a prototyped RM written in Python 3. Better performances can be expected from a C implementation.

VII. Conclusion

We investigated how mapping analysis with global scope can improve the quality of generated schedules for multi-threaded firm real-time applications. We proposed a fast algorithm to schedule the applications on heterogeneous multi-core systems. The proposed approach achieves a scheduling rate competitive with the state-of-the-art, while improving the energy efficiency by around 13%. The generated solutions are only 3.6% off from optimal schedules obtained exhaustively. The algorithm runs an order of magnitude faster than the state-of-the-art approach, making it a good candidate for integration into other runtime resource managers.

Acknowledgment

This work was supported in part by the German Research Foundation (DFG) within the Collaborative Research Center HAEC and the Center for Advancing Electronics Dresden (cfaed). We thank Silexica (www.silexica.com) for making their SLX Tool Suite and the applications available to us.

References

[1] A. Olofsson, "Epiphany-v: A 1024 processor 64-bit risc system-on-chip," *arXiv preprint arXiv:1610.01832*, 2016.

[2] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Mapping on multi/many-core systems: Survey of current and emerging trends," in *Proceedings of DAC*. ACM, May 2013, pp. 1–10.

[3] A. K. Singh, A. Kumar, and T. Srikanthan, "Accelerating throughput-aware runtime mapping for heterogeneous mpsocs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 18, no. 1, pp. 9:1–9:29, Jan. 2013.

[4] A. Weichslgartner, D. Gangadharan, S. Wildermann, M. Glaß, and J. Teich, "Daarm: Design-time application analysis and run-time mapping for predictable execution in many-core systems," in *Proceedings of CODES+ISSS*. IEEE, 2014, pp. 1–10.

[5] C. Ykman-Couvreur, V. Nollet, F. Catthoor, and H. Corporaal, "Fast multi-dimension multi-choice knapsack heuristic for mp-soc run-time management," in *Proceedings of SOC*, Nov 2006, pp. 1–4.

[6] S. Wildermann, A. Weichslgartner, and J. Teich, "Design methodology and run-time management for predictable many-core systems," in *Proceedings of ISORCW*, April 2015, pp. 103–110.

[7] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*. New York, NY, USA: John Wiley & Sons, Inc., 1990.

[8] M. Niknafs, I. Ukhov, P. Eles, and Z. Peng, "Runtime resource management with workload prediction," in *Proceedings of DAC*. ACM, 2019, pp. 169:1–169:6.

[9] C. Gregg, M. Boyer, K. M. Hazelwood, and K. Skadron, "Dynamic heterogeneous scheduling decisions using historical runtime data," 2011.

[10] J. Castrillon *et al.*, "Trace-based kpn composability analysis for mapping simultaneous applications to mpsoc platforms," in *Proceedings of DATE*, Mar. 2010, pp. 753–758.

[11] V. Venkataramani, A. Pathania, and T. Mitra, "Scalable optimal greedy scheduler for asymmetric multi-/many-core processors," in *SAMOS*, 2019.

[12] A. Das, B. M. Al-Hashimi, and G. V. Merrett, "Adaptive and hierarchical runtime manager for energy-aware thermal management of embedded systems," *ACM TECS*, vol. 15, no. 2, pp. 24:1–24:25, Jan. 2016.

[13] S. Tzilis, P. Trancoso, and I. Sourdis, "Energy-efficient runtime management of heterogeneous multicores using online projection," *ACM TACO*, vol. 15, no. 4, pp. 63:1–63:26, Jan. 2019.

[14] S. Libutti, G. Massari, and W. Fornaciari, "Co-scheduling tasks on multi-core heterogeneous systems: An energy-aware perspective," *IET Computers Digital Techniques*, vol. 10, no. 2, pp. 77–84, 2016.

[15] A. K. Singh, M. Shafique, A. Kumar, and J. Henkel, "Resource and throughput aware execution trace analysis for efficient run-time mapping on mpsocs," *IEEE TCAD*, vol. 35, no. 1, pp. 72–85, Jan 2016.

[16] G. Ascia, V. Catania, and M. Palesi, "Multi-objective mapping for mesh-based noc architectures," in *Proceedings of CODES+ISSS*. ACM, 2004, pp. 182–187.

[17] G. Massari *et al.*, "Combining application adaptivity and system-wide resource management on multi-core platforms," in *Proceedings of SAMOS*, July 2014, pp. 26–33.

[18] W. Quan and A. D. Pimentel, "A hybrid task mapping algorithm for heterogeneous mpsocs," *ACM TECS*, vol. 14, no. 1, p. 14, 2015.

[19] G. Onnebrink, A. Hallawa, R. Leupers, G. Ascheid, and A. Shaheen, "A heuristic for multi objective software application mappings on heterogeneous mpsocs," in *Proceedings of ASPDAC*. ACM, 2019, pp. 609–614.

[20] G. Mariani *et al.*, "Using multi-objective design space exploration to enable run-time resource management for reconfigurable architectures," in *Proceedings of DATE*, March 2012, pp. 1379–1384.

[21] A. Goens *et al.*, "Tetris: a multi-application run-time system for predictable execution of static mappings," in *Proceedings of SCOPES*. ACM, Jun. 2017, pp. 11–20.

[22] A. Goens, S. Siccha, and J. Castrillon, "Symmetry in software synthesis," *ACM TACO*, vol. 14, no. 2, pp. 20:1–20:26, Jul. 2017.

[23] S. Wildermann, M. Glaß, and J. Teich, "Multi-objective distributed run-time resource management for many-cores," in *Proceedings of DATE*, 2014, pp. 221:1–221:6.

[24] H. Shojaei, T. Basten, M. Geilen, and A. Davoodi, "A fast and scalable multidimensional multiple-choice knapsack heuristic," *ACM TODAES*, vol. 18, no. 4, pp. 51:1–51:32, Oct. 2013.

[25] H. Bouraoui, J. Castrillon, and C. Jerad, "Comparing dataflow and openmp programming for speaker recognition applications," in *Proceedings of PARMA-DITAM*. ACM, 2019, pp. 4:1–4:6.