Preprint from 29th International Conference on Architecture of Computing Systems (ARCS), April 2016, Nuremberg, Germany

Task Variants with Different Scratchpad Memory Consumption in Multi-Task Environments

Martin Böhnert and Christoph Scholl

Department of Computer Science
University of Freiburg, Germany
{boehnert|scholl}@informatik.uni-freiburg.de

Abstract. We present an approach which schedules task sets using scratchpad memory (*SPM*) in an embedded multi-task system with real-time constraints. A new task model is introduced, where each task is represented by different pre-compiled *variants* which differ in the amount of scratchpad memory used. A higher use of SPM leads to smaller runtimes of a task. Moreover, the energy consumption is reduced by replacing memory accesses by SPM accesses. Our heuristic method assembles a task set of these variants by choosing one variant per task. After selecting candidates from the pre-computed set of task variants, the task set can be handled by a real-time scheduler like EDF. Our approach is able to build a new incremental task set and feasible transition in dynamically changing environments. Furthermore we show an extension of our approach to multicore environments.

1 Introduction

Designing embedded systems with minimal energy consumption is becoming increasingly important. The need for energy savings has manifold reasons. Apart from problems stemming from limited energy resources on earth, there are also problems with heat dissipation by processors and corresponding cooling problems. Moreover, embedded systems are used in more and more application areas where their energy supply has to rely on batteries.

There are several approaches to minimize energy in embedded processors and controllers. One particular aspect we consider in this paper is the usage of scratchpad memories (*SPM*) [3,15] instead of conventional caches. Scratchpad memories (*SPM*) are fast and located near to the CPU. Since they usually have no access penalty through wait states at accesses, they provide fast access to often used data. Moreover, SPM accesses need less energy than main memory accesses. In contrast to caches, SPMs are managed by the programmer / compiler, who decides which data is located in this memory. SPMs which are equal in memory size to caches, need no parallel comparison logic and therefore less chip area and less energy.

We use scratchpad memory to reduce processor utilization and energy consumption in a multi-task environment. More precisely, for each task we provide several pre-compiled *variants* using different amounts of scratchpad memory, i.e. we assume a framework like [24] which is able to map memory addresses either to main memory or scratchpad memory and produces different executables depending on a given upper limit to the amount of SPM provided to the task. Usually, the WCET (worst case execution time) of a task is shorter and its worst case energy consumption is lower, if it has more SPM at its disposal.

[19], e.g., provides a heuristical method which minimizes run-times and energy consumption by analyzing the code of a task and by assigning code and/or data memory to available SPM of a given size. Thus, each tasks variant is connected with a triple of WCET, SPM, and energy consumption. Several tasks have to share both the CPU and the scratchpad memory. Given a certain amount of scratchpad memory in the system, we select for each task a *variant* such that

1. the sum of the SPM consumptions of all selected task variants does not exceed the available SPM,
2. the CPU utilization caused by the selected task variants does not exceed 1 and
3. the energy consumption is minimized under those constraints.

If the resulting CPU utilization factor is smaller than 1, then we have the additional opportunity to decrease the processor frequency (until the utilization factor is 1) which minimizes the energy consumption further.

The given problem resembles the Multiple Choice Knapsack Problem (*MCKP*) [16, 10] and can be solved exactly by 0-1-Integer-Linear-Programming. Since we need fast solutions in dynamically changing real-time system, we developed a heuristic which is much faster than the exact solution, and it compares well in quality though. Additionally our algorithm provides incremental solutions when a new task enters a system. This solution incorporates a feasible transition from the first set of task variants to the second without exceeding the available amount of SPM or processor resources. We extended our heuristics to multicore systems where exactly one task variant has to be allocated to exactly one processor.

Related Work In [15] a first approach distributing data between SPM and external DRAM was presented which accelerates memory accesses for memory architectures using both SPM and caches. Later on, [3] showed, that SPMs have significant lower area and power consumption compared to caches. They also presented a simple SPM allocation method outperforming caches. [19] statically mapped global data memory objects and program code to SPM of a single task. They formulated a Knapsack problem with energy minimization as the optimization goal.

Many publications confirm that SPMs are useful for reducing of area, run-times and energy consumptions. Most methods use profile runs to determine frequently accessed data and executed code as a basis for SPM allocation. *Static* SPM allocation is used in [2, 18] for global data and stack variables, and in [1, 23] for code. *Dynamic* SPM allocation is used in [21, 5]. Here, global data, stack, or even heap variables are dynamically allocated to SPM at runtime.

More recent methods looked into the minimization of WCETs as well which is more important for embedded real-time systems [8, 9, 20]. The results show, that by increasing the amount of available SPM, it is possible to decrease the WCET and the energy consumption of a task. This is the basic assumption underlying our approach.

Our approach is neither restricted to any special method for assigning data and / or code to an SPM nor to the application of static or dynamic methods for allocating SPM. For our method it is only important that for each task variant there exists a fixed upper bound on the size of the SPM used by this variant; our method is completely orthogonal to the question *how* this amount of SPM is used by the task variant.

More recent methods consider SPM usage also in multi-task environments. A first approach was presented in [22]. In this work, a statically scheduled system

with a fixed number of processes was considered. In the 'saving' approach, the whole SPM is given to the currently active task, the SPM contents are saved and restored at each context switch. Our method is closer to the so called 'non-saving' approach. Here, the SPM is split into disjoint regions, at most one for each task. In contrast to this method, our approach considers several task variants to select from instead of one, it can handle a dynamic number of tasks, it is integrated into the scheduler, and selects a schedulable set of variants for the tasks with the goal of minimizing the energy consumption.

The work in [7] uses the same SPM sharing strategies as [22], but is suitable for *dynamic* multitasking systems. The shares are redistributed whenever a task enters or leaves the system. This approach relies on the existence of an MMU in the memory architectures as it uses page faults to copy new pages into the SPM. In contrast to our approach, [7] has difficulties in real-time systems, since it is hard to provide tight WCET estimates with a strategy based on page faults. Furthermore, this approach is not designed to give guarantees regarding schedulability of a real-time task set.

[25] and [26] look into the 'saving approach' discussed above for sharing SPM between several tasks. They propose and analyze refined schemes for saving and restoring SPM data during context switches. An earlier work [17] proposes hardware support by DMA (Direct Memory Access) to reduce the cost of copying between scratchpad and main memory.

The remaining paper is structured as follows: We discuss the preliminaries of this work in Sect. 2. Sect. 3 presents our basic idea and gives an exact formulation of the resulting optimization problem. We give a heuristical solution to this problem in Sect. 4. In Sect. 7 the approach is evaluated by experiments. Finally, Sect. 8 concludes the paper with a summary.

2 Preliminaries

We assume a set of periodic tasks which can be preempted and are scheduled by the Earliest Deadline First (EDF) scheduler [13]. If we do not consider task *variants*, then a task τ_i is specified by its worst-case computation time C_i and its period T_i. We assume that the relative (hard) deadline D_i of a task is the same as the period T_i. The utilization factor U of a task set of n independent periodic tasks is defined as $U = \sum_{i=1}^{n} \frac{C_i}{T_i}$. Such a task set is schedulable with EDF if and only if $U \leq 1$. This holds for EDF scheduling for a single processor system. For simplicity we first assume a single processor in this work.

Our task variant selection problem is a generalization of the Multiple Choice Knapsack Problem (*MCKP*). In the Multiple Choice Knapsack Problem (*MCKP*) [16] exactly one item from each of n classes N_i is selected such that the profit is maximized:

Multiple Choice Knapsack Problem
Given: n classes N_i of items with weights $w_{ij} \in \mathbb{N}$ and profits $p_{ij} \in \mathbb{N}$ $(1 \leq i \leq n,\, j \in N_i)$, a capacity $c \in \mathbb{N}$.
Find: Maximize $\sum_{i=1}^{n} \sum_{j \in N_i} x_{ij} p_{ij}$ with
$\sum_{i=1}^{n} \sum_{j \in N_i} x_{ij} w_{ij} \leq c$, $\sum_{j \in N_i} x_{ij} = 1$ $(1 \leq i \leq n)$, $x_{ij} \in \{0,1\}$ $(1 \leq i \leq n,\, j \in N_i)$.
($x_{ij} = 1$ iff the item j from class N_i is selected.)

MCKP is an NP-complete problem. There is a number of exact solution methods such as dynamic programming, branch-and-bound, or 0-1 Integer Linear Programming [6, 12, 16].

3 Basic Idea and Problem Formulation

As already mentioned in the introduction, our basic idea is to use different code variants for each task which differ in the amount of SPM that can be used. If a variant is allowed to use more SPM, this usually means that the energy consumption and the WCET is reduced [9]. The selection of variants for the different tasks has to be done in a way that *a)* the sum of the SPM consumptions of all selected task variants does not exceed the available SPM, *b)* the CPU utilization caused by the selected task variants does not exceed 1, and *c)* the energy consumption is minimized under those constraints. If the utilization factor resulting from the selected variants is larger than 1, then the task set with its variants is not schedulable with the given amount of SPM in the system. If the utilization factor is smaller than 1, it may be possible to reduce the processor frequency in order to save even more energy.

In the following we assume that all computations are done with worst case execution times resulting from the maximal processor frequency. If a schedulable selection of task variants has been found, then the processor frequency can be adjusted accordingly. So our algorithm works with worst case execution *times* instead of cycles and we always mean the worst case execution times under the maximal processor frequency.

Furthermore, we assume a system without caches to avoid unstable WCET estimations for different task variant combinations and thus differing cache hit and miss patterns.

We assume a size M of the scratchpad memory in the system. Each task τ_i is connected with a task period T_i and a set of variants V_i. Each variant j in V_i is connected with the following properties:

- An SPM consumption M_{ij} which represents a fraction $s_{ij} = \frac{M_{ij}}{M}$ of the total SPM in the system.
- A (worst case) computation time C_{ij} leading to a contribution $u_{ij} = \frac{C_{ij}}{T_i}$ to the processor utilization factor.
- A (worst case) energy consumption E_{ij}. When computing the average energy consumption of the whole system, the energy consumptions of single task variants have to be weighted by the fraction of time the task variant is running on the processor, which is equal to u_{ij}. Thus, the task variant contributes $e_{ij} = u_{ij} \cdot E_{ij}$ to the average energy consumption.

Definition 1. *We call* $s_{ij} = \frac{M_{ij}}{M}$ *the* scratchpad memory share *(or* memory share *for short),* $u_{ij} = \frac{C_{ij}}{T_i}$ *the* utilization share*, and* $e_{ij} = u_{ij} \cdot E_{ij}$ *the* energy contribution *of the task variant.*

Altogether, if a task τ_i has v_i variants, it is represented by a set

$$V_i = \{(s_{ij}, u_{ij}, e_{ij}) \mid 1 \leq j \leq v_i\}.$$

Definition 2. *For all* $1 \leq i \leq n$ *let* $curr_i$ *(with* $1 \leq curr_i \leq v_i$*) be the selected task variant for task* τ_i*. By analogy to the processor utilization factor we define the* SPM utilization factor *of the selected task variants by* $S := \sum_{i=1}^{n} s_{i curr_i}$*. The* average energy consumption *is given by* $E := \sum_{i=1}^{n} e_{i curr_i}$*.*

Now we have the requirements that the sum of scratchpad memory shares s_{ij} and the sum of the utilization shares u_{ij} over all selected variants is ≤ 1. The average energy consumption has to be minimized. Thus, we arrive at the following problem formulation:

Task Variant Selection (TVS)
Given: n tasks τ_i with sets
$V_i = \{(s_{ij}, u_{ij}, e_{ij}) \mid 1 \leq j \leq v_i\}$ of task variants, $s_{ij}, u_{ij} \in (0,1] \subseteq \mathbb{Q}$, $e_{ij} \in \mathbb{Q}_{\geq 0}$ $(1 \leq i \leq n,\ 1 \leq j \leq v_i)$.
Find: Minimize $\sum_{i=1}^{n} \sum_{j=1}^{v_i} x_{ij} e_{ij}$ with
$\sum_{i=1}^{n} \sum_{j=1}^{v_i} x_{ij} s_{ij} \leq 1$, $\sum_{i=1}^{n} \sum_{j=1}^{v_i} x_{ij} u_{ij} \leq 1$, $\sum_{j=1}^{v_i} x_{ij} = 1$ $(1 \leq i \leq n)$,
$x_{ij} \in \{0,1\}$ $(1 \leq i \leq n,\ 1 \leq j \leq v_i)$.
($x_{ij} = 1$ iff variant j of task τ_i is selected.)

It is easy to see that the TVS problem generalizes the Multiple Choice Knapsack Problem, since it has two cost constraints (for processor *and* SPM utilization factors) instead of one. It can be solved exactly by 0-1-ILP.

4 Heuristical Solution

Especially if we apply our approach in the context of dynamically changing real-time embedded systems, we need a fast solution to the Task Variant Selection (TVS) problem. For this reason we present a fast heuristical solution here.

In a first step we compute a feasible solution, i.e., a solution where both the sum of selected s_{ij} and the sum of selected u_{ij} is less or equal to 1. If we find a feasible solution, then we further minimize the energy consumption while maintaining feasibility in step 2.

4.1 Step 1: Computing a Feasible Solution

In the first step we do not yet look at energy contributions e_{ij}, but concentrate on computing a feasible solution.

Step 1.1: Filtering wrt. Pareto Optimality At first, we remove all task variants which are not Pareto optimal wrt. (s_{ij}, u_{ij}). If the set $V_i = \{(s_{ij}, u_{ij}) \mid 1 \leq j \leq v_i\}$ of task variants for task τ_i contains a pair of variants x and y with $s_{ix} \geq s_{iy}$ and $u_{ix} \geq u_{iy}$, then we remove variant x from V_i, since variant x is not helpful in finding feasible solutions.

Step 1.2: Initial Solution For a feasible solution, we have to competing goals: Reduction of the processor utilisation below 1, while staying inside the SPM limit.

So we construct a *balanced* initial solution which tries to avoid large scratchpad memory and utilization shares *at the same time*. To do so, for a fixed task τ_i we compute the Euclidean lengths $l(s_{ij}, u_{ij}) := \sqrt{s_{ij}^2 + u_{ij}^2}$ of all vectors (s_{ij}, u_{ij}) $(1 \leq j \leq v_i)$, and select the task variant $start_i$ with the minimal length to be included into the initial solution.

The intuition is that we try to avoid long vectors which have a large memory share or a large utilization share or both. For this step to make sense, it

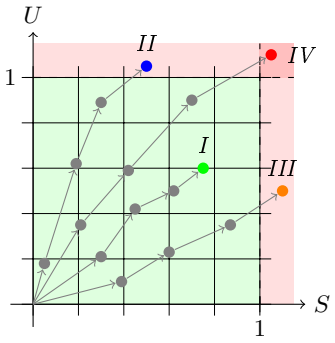

Fig. 1: The graph illustrates 4 different cases for the exchange based algorithm. I corresponds to a feasible solution, II and III violate only a single constraint (either the utilization U or the memory S), IV violates both.

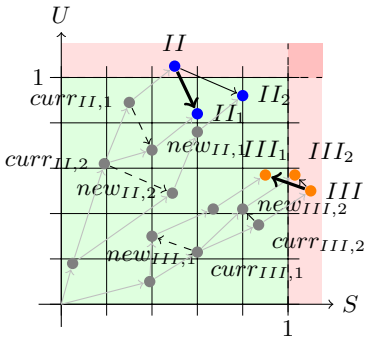

Fig. 2: The choice of possible exchange partners is illustrated. In case II the two choices $curr_{II,1}$ and $curr_{II,2}$ are considered for exchange with either $new_{II,1}$ or $new_{II,2}$. Because the difference vector $diff_{new_{II,1},curr_{II,1}}$ has the minimum gradient, this exchange will be taken. Case III is analogous.

is important that the memory and utilization shares are *normalized* in a way that we require both $\sum_{i=1}^{n} s_i start_i \leq 1$ and $\sum_{i=1}^{n} u_i start_i \leq 1$ with the same upper bound of 1 for a feasible solution. Nevertheless, this approach is only a heuristical method to obtain a good initial solution. It still may be the case that $\sum_{i=1}^{n} s_i start_i > 1$, $\sum_{i=1}^{n} u_i start_i > 1$ or both. For this reason, we try to improve the initial solution by exchanging task variants.

Step 1.3: Exchange Based Algorithm The exchange based algorithm always holds a current solution consisting of one task variant $curr_i$ ($1 \leq curr_i \leq v_i$) for each task τ_i. It starts with the initial solution, i.e, with $curr_i := start_i$ for all $1 \leq i \leq n$. The exchange based algorithm is a greedy approach; a task variant which has previously been removed from the solution is never brought back into the solution.

The algorithm differentiates between 4 cases for the current solution which are illustrated by Fig. 1. Each task variant $curr_i$ corresponds to a vector $(s_i curr_i, u_i curr_i)$. The sum of all vectors over all tasks corresponds to the pair $(\sum_{i=1}^{n} s_i curr_i, \sum_{i=1}^{n} u_i curr_i)$ which gives for the current solution the scratchpad memory utilization factor S and the processor utilization factor U. If we already have a feasible solution (case I), we stop exchanging in step 1 and continue with Step 2. Otherwise we are looking for a new exchange candidate.

An exchange candidate is rated by the difference vector $diff_{new_i,curr_i} := (s_i new_i - s_i curr_i, u_i new_i - u_i curr_i)$, where new_i of task τ_i is considered a candiate for $curr_i$. Note that in all difference vectors $diff_{new_i,curr_i}$ either the first or the second component is negative. If both components were negative (positive), then variant $curr_i$ (new_i) would not be Pareto-optimal and would have been removed in Step 1.1. The next exchange is determined based on the gradients of vectors $diff_{new_i,curr_i}$ depending on the following cases II, III, and IV, see Fig. 2.

– Case II: $\sum_{i=1}^{n} s_i curr_i \leq 1$, $\sum_{i=1}^{n} u_i curr_i > 1$: We choose an exchange candidate which reduces the processor utilization factor, i.e., a candidate new_i

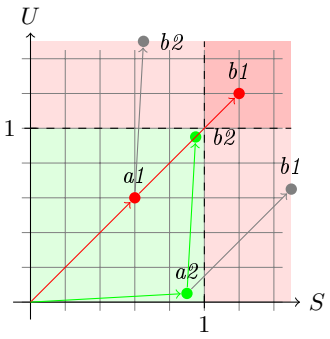

Fig. 3: Case IV of the initial solution: In the example task a has variants $a1$ and $a2$, task b variants $b1$ and $b2$. The initial solution consisting of $a1$ and $b1$ violates both constraints. After the first exchange of $a1$ with $a2$ only the U-constraint is violated, after the second exchange of $b1$ with $b2$ we obtain a feasible solution.

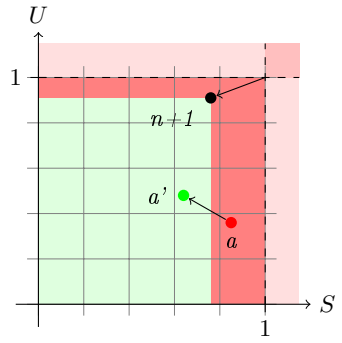

Fig. 4: Dynamic system case: We start from a feasible solution a. A new task $n+1$ is dynamically added to the task set. Our method tries to reach the green area with $0 \leq S \leq 1 - s_{n+1start_{n+1}}$ and $0 \leq U \leq 1 - u_{n+1start_{n+1}}$ without violation of the constraints $S \leq 1$ and $U \leq 1$ in the exchange process.

with $u_inew_i - u_icurr_i < 0$. Among those candidates we choose the one with minimal (negative) gradient

$$\frac{u_inew_i - u_icurr_i}{s_inew_i - s_icurr_i} \tag{1}$$

of the difference vector $diff_{new_i,curr_i}$. The intuition for this decision is that the gain wrt. processor utilization factor U *relative to* the penalty wrt. SPM untilization factor S is maximized.

– Case III: $\sum_{i=1}^{n} s_icurr_i > 1$, $\sum_{i=1}^{n} u_icurr_i \leq 1$: Here we have to choose an exchange candidate which reduces the SPM utilization factor, i.e., a candidate new_i with $s_inew_i - s_icurr_i < 0$. Similarly to case II, among those candidates we choose the one with minimal gradient

$$\frac{s_inew_i - s_icurr_i}{u_inew_i - u_icurr_i} \tag{2}$$

of the difference vector $diff_{new_i,curr_i}$.

– Case IV: $\sum_{i=1}^{n} s_icurr_i > 1$, $\sum_{i=1}^{n} u_icurr_i > 1$: Since we cannot reduce SPM and processor utilization at the same time, we arbitrarily choose one component first and try to reduce the other one later on, depending on the following cases II, III and IV. Fig. 3 illustrates an example with an initial solution violating both the processor utilization and the SPM utilization constraints (case IV). An exchange reducing S leads us into case II, a second exchange leads us to case I where both S and U are reduced to values ≤ 1.

In general, during the search for a feasible solution, it may happen that our algorithm switches back and forth between the exchange directions, towards more SPM usage or more processor usage, for several times. However, it always stops, if a feasible solution is found or at latest if all task variants have been moved once. If we have found a feasible solution, the solution is postprocessed in step 2 with the goal of further minimizing the average energy consumption.

4.2 Step 2: Postprocessing for energy minimization

We start with a feasible solution and perform further exchanges for energy minimization. We start with an application specific observation: As seen in step 1, there are only two cases wrt. exchanges: Either processor utilization is reduced and SPM utilization is increased or processor utilization is increased and SPM utilization is decreased. Increases in processor utilization and decreases in SPM utilization usually *both* increase the energy consumption. So we completely neglect exchange steps of the first type (i.e. we always proceed as in case II of step 1). There are only two differences compared to case II of step 1: (1) We omit exchanges which would produce an infeasible solution with SPM utilization larger 1. (2) For choosing exchange candidates we consider the gradients $\frac{e_i new_i - e_i curr_i}{s_i new_i - s_i curr_i}$ instead of $\frac{u_i new_i - u_i curr_i}{s_i new_i - s_i curr_i}$, since now our goal is to minimize the average energy. (Moreover, we remove task variants which are not Pareto-optimal wrt. (s_{ij}, e_{ij}) before we start exchanging in step 2.)

4.3 Complexity

Let $V = \sum_{i=1}^{n} v_i$ be the number of task instances in the original problem, $v_{max} = \max_{i=1}^{n} v_i$ the maximum number of variants per task. The main observation for worst case complexity estimation of step 1 or 2 is the fact that each of the V task instances is exchanged at most once. Each exchange step is in $O(v_{max} + n)$, leading to an overall worst case complexity of $O(V \cdot (v_{max} + n))$.

5 Dynamic systems

Many real-time systems change behaviour dynamically over time. Our algorithm is suitable for an incremental use where a new task comes into the system. It exchanges task variants until a new feasible solution is found. If not, the newly arrived tasks is rejected.

However, there is an important additional constraint to be fulfilled: Even if we are able to compute a feasible solution including the new task, it is not guaranteed that there is a *transition* from the previous set of task variants to the new set of task variants (including a variant of the new task) which does not violate the processor utilization constraint or the SPM utilization constraint in between. Here we make the realistic assumption that task variants can not be exchanged instantaneously, but the code of one task variant i_j can only exchanged to the code of another variant $i_{j'}$, when the current instance of i_j has finished, i.e., no task variant of i_j is active.

In the following we present an approach that constructs a sequence of task instance exchanges guaranteeing $S \leq 1$ and $U \leq 1$ for the intermediate configurations as well. The 'schedule of task instance exchanges' follows the steps in a modified exchange algorithm.

In a first sequence of exchanges we try to 'make room' for the newly arriving task τ_{n+1}, such that adding a variant of τ_{n+1} leads to a feasible solution. Since we do not yet know which variant of τ_{n+1} will be selected in the end, we proceed similar to Step 1.2 in Sect. 4.1, i.e., we select the variant $start_{n+1}$ of τ_{n+1} with the smallest Euclidean vector length. Now we start from the current set of task variants and try to reach a pair (S, U) of SPM utilization factor and processor utilization factor with $S \leq 1 - s_{n+1 start_{n+1}}$ and $U \leq 1 - u_{n+1 start_{n+1}}$. Of course,

we have to ensure that for each intermediate step on this way we have $S \leq 1$ and $U \leq 1$. Fig. 4 illustrates the approach.

The implementation of the approach is similar to Step 1.3 in Sect. 4.1 with the difference that we replace the limit 1 for S by $1 - s_{n+1start_{n+1}}$ and the limit 1 for U by $1 - u_{n+1start_{n+1}}$. Moreover, we only accept exchange steps leading to an intermediate solution with $S \leq 1$ and $U \leq 1$. If we have finally reached the situation that $0 \leq S \leq 1 - s_{n+1start_{n+1}}$ and $0 \leq U \leq 1 - u_{n+1start_{n+1}}$, we add τ_{n+1} with the current variant $(s_{n+1start_{n+1}}, u_{n+1start_{n+1}})$ into the solution. Then, exactly the postprocessing step for energy minimization from Sect. 4.2 follows which minimizes the average energy further without violating $S \leq 1$ and $U \leq 1$.

6 Multicore Systems

Since many embedded systems feature multicore CPUs, we extend our approach to the use with these systems. We use a simple multicore system where each CPU has its own local SPM (of the same size) and there is a fixed assignment of the tasks to CPUs. For multicore CPUs, TVS is generalized to the following problem TVSM which can be solved exactly by 0-1-ILP:

Task Variant Selection Multicore (TVSM)
Given: m processor cores $p_1, \ldots, p_k$, n tasks τ_i with sets
$V_i = \{(s_{ij}, u_{ij}, e_{ij}) \mid 1 \leq j \leq v_i\}$ of task variants, $s_{ij}, u_{ij} \in (0, 1] \subseteq \mathbb{Q}$, $e_{ij} \in \mathbb{Q}_{\geq 0}$ $(1 \leq i \leq n,\ 1 \leq j \leq v_i)$.
Find: Minimize $\sum_{k=1}^{m} \sum_{i=1}^{n} \sum_{j=1}^{v_i} x_{ijk} e_{ij}$ with
$\sum_{i=1}^{n} \sum_{j=1}^{v_i} x_{ijk} s_{ij} \leq 1$ $(1 \leq k \leq m)$,
$\sum_{i=1}^{n} \sum_{j=1}^{v_i} x_{ijk} u_{ij} \leq 1$ $(1 \leq k \leq m)$,
$\sum_{k=1}^{m} \sum_{j=1}^{v_i} x_{ijk} = 1$ $(1 \leq i \leq n)$,
$x_{ijk} \in \{0, 1\}$ $(1 \leq i \leq n,\ 1 \leq j \leq v_i,\ 1 \leq k \leq m)$.
($x_{ijk} = 1$ iff variant j of task τ_i is selected and assigned to processor k.)

For the heuristical solution we adapt our heuristics from the single-core case: The initial task variant selection from Step 1.2, Sect. 4.1, is extended by a distribution over the m available processor cores. The tasks are assigned to the cores one by one. For each core we sum up the Euclidian lengths of task variant vectors which are already assigned to it and we always greedily assign the next task to the core with the smallest sum.

Then, each task set on each core is optimized individually as in the single-core case.

For the dynamic system case, the new task is added to the core with the smallest average energy consumption. This core will then be treated as in Sec. 5.

7 Experimental Results

To evaluate our algorithm, we considered benchmarks with data generated from the MiBench [11] benchmark suite. SPM allocations for the MiBench programs were made with the help of the MACC framework [24], which only allocates global data but also tries to relocate local data to global data. These SPM enabled variants are then run in the MPARM/MEMSIM simulator [4] to obtain the corresponding energy and runtime values. For our experiments we used seven

Table 1: Results MiBench data Benchmarks

Algorithm	SPM usage [%]	Processor utilization [%]	Energy share	Solution time [ms]
Baseline initial	0.00	54.54	168,971,415	—
Gurobi initial	99.45	48.38	142,724,205	2.9984
Heuristic initial	68.90	48.30	145,104,542	1.0592
Baseline	0.00	63.63	317,414,842	—
Gurobi dynamic	99.73	56.43	275,237,960	6.4964
Heuristic dynamic	69.85	56.55	277,601,029	0.1171

programs with two to five obtained variants. These programs were *bitcount*, *dijkstra*, *stringsearch* (large and small), *rijndael*, *sha* and *crc*. *bitcount* acts as the task joining the system dynamically. For computing exact solutions to the task variant selection problem we used the 0-1-ILP solver Gurobi [14].

Table 1 shows the results of these experiments. The first three lines in Table 1 give results for the initial task set. The line labeled 'baseline' shows the processor utilization and the average energy for the task set without using the SPM at all. Our heuristics is able to compute a solution with an average energy which is only about 2 percent higher than the energy obtained by the exact solution (Gurobi), but with only one third of the solution time.

Our solution for the dynamic system is within a 1 percent range of the exact solution regarding the average energy. Here our heuristics is even faster with a factor of about 60. It is important to keep in mind, that our heuristics gives not only a feasible solution but also a feasible transition from the first task set to the second one. Gurobi only computes the new optimal task set without any guarantee that such a transition exists.

In order to set the evaluation onto a broader basis we also performed experiments with synthetic benchmarks. We randomly generated tasks with variants whose triplets of SPM usage, WCET, and energy consumption show characteristics derived from results in the literature discussed in Sect. 1, whereas SPM usage is reciprocal to runtime, and runtime correlates to energy. We generated initial task sets consisting of 15 tasks, each with 4 variants. Starting from a solution of such a task set, we added another task to evaluate the behavior for dynamic systems. 1000 of these task sets were created for the evaluation. Finally, we applied our approach also in a multicore setting, where 4 cores obtained a total of 60 tasks with 4 variants each. As before we add another task to the initial set to evaluate the dynamic behavior. Because of high run times of the exact solution we confined ourselves to 11 test runs with a multicore data set.

Table 2: Results synthetic Benchmarks

Algorithm	SPM usage [%]	Processor utilization [%]	Energy share	Solution time [ms]	Solutions
Gurobi initial	99.73	42.42	61.7597	56.2189	998
Heuristic initial	99.48	42.32	70.2280	0.7359	998
Gurobi dynamic	99.80	50.58	84.7071	114.9391	879
Heuristic dynamic	99.50	50.05	90.5008	0.4214	871
Gurobi (4 cores) initial	99.97	40.87	228.8178	492,301.5050	11
Heuristic (4 cores) initial	99.45	41.17	265.2159	4.1255	11
Gurobi (4 cores) dynamic	99.98	42.86	245.0765	1,528,642.3016	11
Heuristic (4 cores) dynamic	99.50	43.17	283.1873	0.4718	11

As we can see in Table 2, in 998 out of 1000 cases our algorithm finds a feasible solution for the initial task set. In the remaining 2 cases there exists no feasible solution as the exact result computed by Gurobi shows. On average the energy consumption in our solution is about 12 percent higher than in the exact

solution. SPM usage and processor utilization are almost equal to the values of Gurobi. However, our approach needs less than 2 percent of the runtime of the exact solution.

In the dynamic system case, Gurobi finds 879 solutions, 8 more than our heuristics. However, we do not know whether there is a feasible transition between the old and the new task sets in these 8 cases. Again remember that our heuristics additionally guarantees a feasible transition from the initial task set with 15 tasks to the new one with 16 tasks. The solutions of our heuristics are still in a 10 percent range of the exact ones regarding the average energy. The CPU times of our heuristics are by a factor of about 300 lower than those of Gurobi.

The last four rows in Table 2 show results for the multicore case. The values for SPM and processor utilization are the average values over all 4 cores. Both Gurobi and the heuristics found feasible solutions for all 11 multicore benchmarks. The energy values of the exact solutions are about 15 percent better on average. The focus here lies on the run times, where our heuristical solution clearly outperforms the exact solution by several orders of magnitude in the initial, as well as in the dynamic case.

8 Conclusions

In this work we presented a new approach for using scratchpad memories in embedded systems with the goal of reducing the average energy consumption in a multi task environment.

We use a new task model, where a task can have multiple pre-compiled variants which differ in their use of scratchpad memories and therefore also in energy usage and worst-case execution times. Apart from an exact solution based on 0-1-Integer-Linear-Programming we presented a fast heuristics for the solution. The heuristical solution is well suited for incremental use in a dynamic environment when tasks enter or leave the system. It also computes a transition from an old to a new task set which guarantees that no system constraints regarding SPM usage and processor utilization are violated in between.

The algorithm presented here works completely *orthogonal* to the method for assigning data and / or code to an SPM.

The experiments show that the solutions computed by the heuristics are competitive to the exact solutions wrt. the energy consumptions. The CPU times for computing the solutions are much faster (sometimes by several orders of magnitude) than those for the exact method and are thus suitable for an application in embedded real-time systems.

References

1. Angiolini, F., Menichelli, F., Ferrero, A., Benini, L., Olivieri, M.: A post-compiler approach to scratchpad mapping of code. In: CASES (2004)
2. Avissar, O., Barua, R., Stewart, D.: An optimal memory allocation scheme for scratch-pad-based embedded systems. ACM Trans. Embedded Comput. Syst. 1 (2002)
3. Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M., Marwedel, P.: Scratchpad memory: a design alternative for cache on-chip memory in embedded systems. In: CODES (2002)

4. Benini, L., Bertozzi, D., Bogliolo, A., Menichelli, F., Olivieri, M.: Mparm: Exploring the multi-processor soc design space with systemc. J. VLSI Signal Process. Syst. 41 (2005)
5. Dominguez, A., Udayakumaran, S., Barua, R.: Heap data allocation to scratch-pad memory in embedded systems. J. Embedded Comput. 1 (2005)
6. Dudziński, K., Walukiewicz, S.: Exact methods for the knapsack problem and its generalizations. European Journal of Operational Research 28 (1987)
7. Egger, B., Lee, J., Shin, H.: Dynamic scratchpad memory management for code in portable systems with an mmu. ACM Trans. Embed. Comput. Syst. 7 (2008)
8. Falk, H., Kleinsorge, J.: Optimal static wcet-aware scratchpad allocation of program code. In: DAC (2009)
9. Falk, H., Lokuciejewski, P.: A compiler framework for the reduction of worst-case execution times. Real-Time Systems 46 (2010)
10. Garey, M.R., Johnson, D.S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W. H. Freeman (1979)
11. Guthaus, M.R., Ringenberg, J.S., Ernst, D., Austin, T.M., Mudge, T., Brown, R.B.: Mibench: A free, commercially representative embedded benchmark suite. In: WWC-4 (2001)
12. Kellerer, H., Pferschy, U., Pisinger, D.: Knapsack problems. Springer (2004)
13. Liu, C.L., Layland, J.W.: Scheduling algorithms for multiprogramming in a hard-real-time environment. J. ACM 20 (1973)
14. Optimization, G., et al.: Gurobi optimizer reference manual. URL: http://www. gurobi. com (2012)
15. Panda, P.R., Dutt, N.D., Nicolau, A.: Efficient utilization of scratch-pad memory in embedded processor applications. In: ED&TC (1997)
16. Pisinger, D.: Algorithms for Knapsack Problems. Ph.D. thesis, DIKU, University of Copenhagen, Denmark (1995)
17. Poletti, F., Marchal, P., Atienza, D., Benini, L., Catthoor, F., Mendias, J.M.: An integrated hardware/software approach for run-time scratchpad management. In: DAC (2004)
18. Sjödin, J., von Platen, C.: Storage allocation for embedded processors. In: CASES (2001)
19. Steinke, S., Wehmeyer, L., Lee, B., Marwedel, P.: Assigning program and data objects to scratchpad for energy reduction. In: DATE (2002)
20. Suhendra, V., Mitra, T., Roychoudhury, A., Chen, T.: Wcet centric data allocation to scratchpad memory. In: RTSS (2005)
21. Udayakumaran, S., Barua, R.: Compiler-decided dynamic memory allocation for scratch-pad based embedded systems. In: CASES (2003)
22. Verma, M., Petzold, K., Wehmeyer, L., Falk, H., Marwedel, P.: Scratchpad sharing strategies for multiprocess embedded systems: a first approach. In: Embedded Systems for Real-Time Multimedia (2005)
23. Verma, M., Wehmeyer, L., Marwedel, P.: Cache-aware scratchpad allocation algorithm. In: DATE (2004)
24. Verma, M., Wehmeyer, L., Pyka, R., Marwedel, P., Benini, L.: Compilation and simulation tool chain for memory aware energy optimizations. In: Embedded Computer Systems: Architectures, Modeling, and Simulation, Lecture Notes in Computer Science, vol. 4017. Springer Berlin Heidelberg (2006)
25. Whitham, J., Audsley, N.: Explicit reservation of local memory in a predictable, preemptive multitasking real-time system. In: RTAS (2012)
26. Whitham, J., Davis, R.I., Audsley, N.C., Altmeyer, S., Maiza, C.: Investigation of scratchpad memory for preemptive multitasking. In: RTSS (2012)