Minimizing Stack Memory for Hard Real-time Applications on Multicore Platforms

Chuansheng Dong
McGill University, Montreal, Canada
Email: chuansheng.dong@mail.mcgill.ca

Haibo Zeng
McGill University, Montreal, Canada
Email: haibo.zeng@mcgill.ca

***Abstract*—Multicore platforms are increasingly used in real-time embedded applications. In the development of such applications, an efficient use of RAM memory is as important as the effective scheduling of software tasks. Preemption Threshold Scheduling is a well-known technique for controlling the degree of preemption, possibly improving system schedulability, and allowing savings in stack space. In this paper, we target at the optimal mapping of tasks to cores and the assignment of the scheduling parameters for systems scheduled with preemption thresholds. We formulate the optimization problems using Mixed Integer Linear Programming framework, and propose an efficient heuristic as an alternative. We demonstrate the efficiency and quality of both approaches with extensive experiments using random systems as well as two industrial case studies.**

I. Introduction

Many real-time embedded systems, including automotive controls [13] (e.g., powertrain applications) are today deployed on multicore architecture platforms. As the industry faces intense cost-cutting pressure in today's hyper-competitive market, it is important to minimize hardware costs by adopting cheaper processors with limited hardware (processing and memory) resources. In most systems-on-chip and also, in general, in embedded systems, availability of RAM is a major constraint and consequently a significant factor determining chip prices, because of the hardware fabrication technology. Today, the space required to manufacture a RAM cell is 10 to 25 times that of a ROM cell, thus availability of both types of memory is typically inversely proportional with the same ratio. Finally, in many systems, and especially in the automotive market, where interchange of components and integration across the supply chain is a major requirement, compliance with standards is mandatory.

Partitioning the computing tasks over multiple cores presents several advantages in terms of power consumption, reliability, and extensibility, but forces the developers to rethink the application decomposition to leverage the availability of parallel processing. Real-time scheduling techniques are mainly classified into *partitioned*, *global*, and *hybrid* [8]. Under *partitioned scheduling*, tasks are statically assigned to cores, and the tasks within each core are scheduled by a local scheduler. This may result in under-utilization, where no core has sufficient capacity remaining to schedule further tasks even if in total a large amount of capacity is unused [8]. Under *global scheduling*, all tasks are scheduled by a single global scheduler. Tasks are dynamically allocated to cores and job migration results in significant overheads. *Hybrid scheduling* combines the strengths of both partitioned and global scheduling. Interested readers may refer to [8] for a survey on the topic of multicore real-time scheduling algorithms.

978-3-9815370-2-4/DATE14/

Partitioned scheduling is adopted by domain-specific standards like AUTOSAR and by commercial real-time operating systems (e.g., VxWorks, LynxOS, and ThreadX). In addition, most automotive controls are designed with *static priority based scheduling* of tasks, as supported by the OSEK and AUTOSAR standards. Other scheduling policies, including Earliest Deadline First (EDF) and its multicore version, are not supported by the industry (yet). In this work, we assume **partitioned scheduling with static priority**, mostly for its practical relevance. However, it should also be noted that the bounds for global scheduling policies are still quite pessimistic (compared with their counterparts for partitioned scheduling, which offer necessary-and-sufficient schedulability tests) [8].

The concept of Preemption Threshold Scheduling (PTS) is introduced in [17] [15]. PTS allows a task to disable preemption from higher priority tasks up to a specified threshold priority; only tasks with priorities higher than the given task's threshold are allowed to preempt it. Its benefits include: reducing the application stack space requirement compared to fully-preemptive scheduling; reducing the number of run-time task preemptions compared to preemptive scheduling; improving schedulability compared to both preemptive and non-preemptive scheduling.

PTS has been integrated into commercial real-time OS (e.g., ThreadX). Also, the automotive OSEK and AUTOSAR OS standards support the concept of Internal Resources, which allow the definition of non-preemption groups [10] and, to some degree, the preemption threshold mechanism. In practice, an (approximate) application-level implementation only requires an API call for changing the task priority at runtime.

Related Work. The definition of *preemption thresholds* is first proposed in [17] to improve schedulability of real-time tasks. The worst-case response time of tasks with preemption thresholds can be computed using the corrected formula in [14]. In [17] [15], the authors proposed several algorithms, including one which maximizes the task preemption threshold level. The *non-preemption groups* model makes use of a similar but not equivalent concept (as shown in [10]), where tasks are partitioned into non-preemptive groups, and tasks belonging to the same group cannot preempt each other.

[14] introduces two other scheduling abstractions: task clusters and task barriers, for better robustness. In [11] a unified framework for static and dynamic priority systems with the definition of preemption thresholds is presented. The authors demonstrate that the algorithm in [15] for the assignment of the

maximum preemption thresholds is also optimal with respect to stack usage. When scheduling offsets are known, they can be exploited to further improve the analysis and the definition of the threshold levels, as discussed in [12] [5].

[18] considers a functional model in which functions are already mapped into tasks and the priorities of tasks are given, and preemption thresholds can be assigned to functions. [21] considers the problem of task priority assignment. It also provides rules and algorithms for the optimal function threshold assignment and the function execution order inside a task when the design starts with a functional model.

Other approaches have been proposed to limit preemption among tasks, including Deferred Preemption Scheduling [3] [19] and Fixed Preemption Points[1] [6] [4]. These approaches, while reducing runtime overhead incurred by preemption and possibly improving system schedulability [7], assume that preemption can be disabled for a single time interval or between predefined locations inside the task code. Therefore, they do not save on stack space, as stacks can only be safely shared among tasks with no preemptions [11].

Our Contributions. In this paper, we target at *the design synthesis problem* of selecting task-to-core mapping, task scheduling and stack management policies to minimize system stack space requirement. We focus on the case of *partitioned fixed-priority scheduling* and *preemption thresholds*, as they are compliant with industrial standards. We first propose a Mixed Integer Linear Programming (MILP) formulation which is only feasible for relatively small task sets. We also present a heuristic algorithm, which is more efficient (1-2 magnitudes faster than simulated annealing (SA) and more than 100 times faster than MILP) with good quality results (on average 5% or less additional stack usage comparable to SA and MILP).

The rest of the paper is organized as follows. Section II presents the system model. Section III describes the algorithms of task priority and preemption threshold assignment on single-core CPUs. The MILP formulation for minimizing the stack usage on multicore platforms is presented in Section IV. In Section V, we propose an efficient and effective heuristic algorithm. Section VI presents experimental results, and finally Section VII concludes the paper.

II. System Model

We consider a set $\mathcal{T}$ of n periodic tasks *scheduled on a multicore platform* with m cores. Each core ρ_k is scheduled independently using a static-priority uniprocessor algorithm. Each task τ_i is associated with a period T_i, a deadline $D_i \leq T_i$ that is no larger than its period, a priority π_i (the higher the number, the higher the priority), and a preemption threshold $\gamma_i \geq \pi_i$ that is assumed once τ_i starts its execution and retained until it finishes. At runtime, τ_i is allowed to preempt τ_j only if $\pi_i > \gamma_j$. We assume tasks have arbitrary offsets. For each core ρ_k, task τ_i requires a worst-case execution time (WCET) $C_{i,k}$ possibly different than others as ρ_k may run at different speed, but the required stack space S_i is the same for all cores (as the generated code is the same). The task stack space can be analyzed by tools (e.g., StackAnalyzer from AbsInt) that use its object file as the input.

[1]The terminology is sometimes inconsistent in different papers. We adopt the one in [7].

In this paper, we assume that task period, deadline, WCETs, and stack are design parameters. We target at the assignment of the design variables including task to core mapping, task priority, and task preemption threshold, such that the system is schedulable and the stack usage is minimized.

III. Task Priority Assignment on Single-core

In this section, we describe the algorithms on the assignment of task priority and preemption threshold to minimize system stack usage on single-core architectures.

When task priorities are known, the algorithm **PTAA**, Preemption Threshold Assignment Algorithm, proposed in [15] defines the maximum preemption threshold assignment for all tasks. As demonstrated in [11], PTAA minimizes preemption among tasks and has minimum system stack usage. It is based on the property that *the maximum preemption threshold of a task is independent from the preemption thresholds of tasks with lower priority*. Thus, starting from the highest priority task down to the lowest priority one, PTAA tries to assign each task with the largest threshold value that will still keep the system schedulable.

The concept of task *blocking time limit* is proposed in [15] (and later redefined in [18]). The blocking time limit of task τ_i, denoted as h_i, is defined as the maximum blocking time τ_i can tolerate while still meeting its deadline. We develop a heuristic for task priority assignment based on an improved estimate on the blocking time limit [22], referred to as **PA-DMMPT**, **P**riority **A**ssignment algorithm assuming **D**eadline **M**onotonic and **M**aximum **P**reemption **T**hreshold, as summarized in Algorithm 1. Given that the computation of the task blocking time limit requires the exact priority order and preemption thresholds of higher priority tasks, we use deadline monotonic (DM) policy to estimate the priority order of higher priority tasks (line 5), and the maximum preemption threshold of these tasks are then found with the optimal PTAA algorithm (line 6). Based on this improved estimate of the blocking time limit, starting from the lowest priority level, the task with the maximum blocking time limit (or the smallest lateness) among the ones in the *unassigned set* is selected at each step. After the task priorities are assigned, PTAA is used to find the maximum task preemption thresholds (line 18).

Algorithm 1: PA-DMMPT [22]

```
   Input: Task set T.
   Output: Priority assignment for tasks in T.
1  Unassigned = T;
2  for each priority level p = 1 to |T| do
3      for each task τ_i in Unassigned do
4          assume p_i = p;
5          assume DM policy for the set Unassigned\{τ_i};
6          use PTAA to assign preemption threshold to Unassigned;
7          calculate blocking time limit h_i for τ_i;
8          if r_i ≤ d_i then
9              a_i = h_i;
10         else
11             a_i = d_i − r_i;
12         end
13     end
14     select τ_i from Unassigned with the largest a_i;
15     p_i = p;
16     Unassigned = Unassigned \{τ_i};
17 end
18 use PTAA to assign task preemption threshold;
```

3200 randomly generated systems with 5 to 20 tasks are used to validate the quality of PA-DMMPT. For all random systems, *PA-DMMPT returns a task priority assignment with the same stack usage as simulated annealing*. For 1000 systems with 5 to 9 tasks where exhaustive search is feasible, both PA-DMMPT and simulated annealing return the same optimal solution as exhaustive search. As expected, PA-DMMPT runs much faster than simulated annealing. For instance, when the number of tasks is 20, the heuristic always takes less than one second, while the simulated annealing algorithm takes 40 minutes on average. We will *use PA-DMMPT in Section V for the development of the heuristic algorithm*.

However, PA-DMMPT is still difficult to be incorporated to MILP framework, as task priorities are assigned in an iterative way. We observe that PA-DMMPT does not directly minimize stack usage; instead, it tries to maximize the possibility of higher thresholds from lower priority tasks at each priority level by picking the task with the largest block time limit. This gives the intuition that *an algorithm easing schedulability also allows for higher thresholds, less preemptability, and hence less stack usage*. Thus, we consider the simple deadline monotonic (DM) policy. For 13200 randomly generated task sets with 5 to 70 tasks, DM policy returns a solution that is unfeasible in 28 systems (but feasible with PA-DMMPT), in 321 systems it has a larger stack space requirement than PA-DMMPT, and for all the other systems the results are the same. The average and maximum differences are 0.6% and 86.7% respectively. We will *use DM priority assignment in Section IV for the MILP formulation*, which greatly simplifies the formulation without losing much optimality.

IV. MILP FORMULATION

In this section, we provide an MILP formulation that considers the task to core mapping and task preemption threshold assignment. *Task priorities are assigned according to the deadline monotonic policy*.

Task mapping. We define a set of optimization variables associated to task mapping.

$$a_{i,k} = \begin{cases} 1, & \text{if } \tau_i \text{ is mapped to core } \rho_k \\ 0, & \text{otherwise} \end{cases} \tag{1}$$

$$u_{i,j,k} = \begin{cases} 1, & \text{if } \tau_i \text{ and } \tau_j \text{ are both mapped to core } \rho_k \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

Each task is mapped to exactly one core

$$\textstyle\sum_{\rho_k} a_{i,k} = 1 \tag{3}$$

Also, u and a variables should be consistent

$$u_{i,j,k} \le a_{i,k},\ \ u_{i,j,k} \le a_{j,k},\ \ u_{i,j,k} \ge a_{i,k} + a_{j,k} - 1 \tag{4}$$

Preemption threshold assignment. For each pair of tasks τ_i and τ_j, τ_i cannot preempt τ_j if and only if $\pi_i \le \gamma_j$. A set of binary variables is used to encode this condition

$$q_{i,j} = \begin{cases} 1 & \text{if } \pi_i \le \gamma_j \\ 0 & \text{otherwise} \end{cases} \tag{5}$$

If task τ_j has a priority higher than or equal to τ_i, then τ_i cannot preempt τ_j.

$$\forall i, j : \pi_i \le \pi_j, \quad q_{i,j} = 1 \tag{6}$$

If τ_i cannot preempt τ_j, then any task τ_k with priority $\le \pi_i$ cannot preempt τ_j, too; conversely, if τ_i can preempt τ_j, any task with priority $\ge \pi_i$ can preempt τ_j.

$$\begin{array}{l} \forall k : \pi_k \le \pi_i, \quad q_{k,j} \ge q_{i,j} \\ \forall k : \pi_k \ge \pi_i, \quad q_{k,j} \le q_{i,j} \end{array} \tag{7}$$

$v_{i,j,k}$ is defined as the product of two binaries $u_{i,j,k}$ and $q_{i,j}$, and $w_{i,j,k}$ is defined as the product of $u_{i,j,k}$ and $(1 - q_{i,j})$.

$$\begin{array}{l} v_{i,j,k} \le u_{i,j,k},\ \ v_{i,j,k} \le q_{i,j},\ \ v_{i,j,k} \ge u_{i,j,k} + q_{i,j} - 1 \\ v_{i,j,k} + w_{i,j,k} = u_{i,j,k} \end{array} \tag{8}$$

Blocking time. Each task τ_i can only be blocked once, with a worst-case blocking time b_i equal to the maximum WCET of any lower priority task τ_j mapped to the same core with a preemption threshold $\gamma_j \ge \pi_i$.

$$\forall j : \pi_j \le \pi_i, \quad b_i \ge \textstyle\sum_{\rho_k} (C_{j,k} \cdot v_{i,j,k}) \tag{9}$$

Real-time Schedulability. We make use of a method for the efficient encoding of schedulability conditions in an MILP framework [20]. Instead of directly calculating the task response time, real-time feasibility can be checked at a set of given point pairs of start and finish times. The feasibility region for the first instance of τ_i is expressed as

$$\bigvee_{(s,f)\in\mathcal{I}_i} \begin{cases} b_i + \displaystyle\sum_{j:\pi_j>\pi_i} rbf_j(s) < s \\ b_i + \displaystyle\sum_{j:\pi_i<\pi_j\le\gamma_i} rbf_j(s) + \sum_{j:\pi_j>\gamma_i} rbf_j(f) + c_i \le f, \end{cases} \tag{10}$$

where $\mathcal{I}_i$ is the set of candidate start and finish time point pairs for τ_i, and $rbf_j(t) = \left\lceil \frac{t}{T_j} \right\rceil c_j$ denotes the request bound function of τ_j within the interval of length t. [20] describes the formulation of (10) in MILP framework, the calculation of $\mathcal{I}_i$ and its simplification.

The second item in the first inequality of (10) should sum over the higher priority tasks that are mapped to the same core as τ_i (thus $\exists k, u_{j,i,k} = 1$), which can be rewritten as

$$\sum_{j:\pi_j>\pi_i} \sum_{\rho_k} \left(\left\lceil \frac{s}{T_j} \right\rceil C_{j,k} \cdot u_{j,i,k} \right) \tag{11}$$

Likewise, the second item in the second inequality of (10) should sum over the higher priority tasks that are mapped to the same core as τ_i but not able to preempt τ_i (thus $\exists k, u_{j,i,k} = 1$ and $q_{j,i} = 1$, or equivalently $v_{j,i,k} = 1$)

$$\sum_{j:\pi_j>\pi_i} \sum_{\rho_k} \left(\left\lceil \frac{s}{T_j} \right\rceil C_{j,k} \cdot v_{j,i,k} \right) \tag{12}$$

Similarly, the third item in the second inequality of (10) is

$$\sum_{j:\pi_j>\pi_i} \sum_{\rho_k} \left(\left\lceil \frac{f}{T_j} \right\rceil C_{j,k} \cdot w_{j,i,k} \right) \tag{13}$$

The task execution time is

$$c_i = \textstyle\sum_{\rho_k} (C_{i,k} \cdot a_{i,k}) \tag{14}$$

Stack usage. For each pair of tasks τ_i and τ_j, if there exists k such that $w_{i,j,k} = 1$, it must be that τ_i and τ_j are mapped to the same core k, and τ_i can preempt τ_j. We denote $\tau_i \preccurlyeq \tau_j \Leftrightarrow w_{i,j,k} = 1$. This relationship is *transitive*. A *preemption graph* is built for each core, where each task is represented as

a vertex, with the weight equal to its stack usage. An edge is added from τ_i to τ_j if $\tau_i \preccurlyeq \tau_j$. The maximum stack usage is the largest total stack usage for a set of tasks where each pair has $\tau_i \preccurlyeq \tau_j$, or equivalently, the longest path in the graph [5].

A naive way to calculate the total stack space is to add a set of constraints restricting that the stack usage is no smaller than the length of any path. However, the number of possible paths is exponential to the number of nodes (tasks) in the graph, resulting an exponential number of constraints. In addition, the preemption graph is not known a-priori. We notice that the preemption graph is *directed acyclic* with only edges from higher priority tasks to lower priority ones. We define a variable $x_{i,k}$ for each task τ_i and each core ρ_k: if $a_{i,k} = 1$, $x_{i,k}$ is the sum of stack usages for τ_i and the ones that can preempt τ_i; otherwise, it is zero. In addition, $y_{j,i,k}$ is defined as the product of the binary $w_{j,i,k}$ and the real variable $x_{j,k}$ (which can be linearized with the standard "big-M" formulation).

If τ_i is mapped to core k ($a_{i,k} = 1$), $x_{i,k}$ is its own stack S_i plus the largest $y_{j,i,k}$, or equivalently, the largest $x_{j,k}$ of τ_j that is mapped to the same core and can preempt τ_i.

$$\begin{aligned} & x_{i,k} \le M \cdot a_{i,k} \\ \forall j : \pi_j \ge \pi_i, \quad & x_{i,k} \ge S_i \cdot a_{i,k} + y_{j,i,k} \end{aligned} \tag{15}$$

The stack usage m_k of core k is

$$\forall i, \quad m_k \ge x_{i,k} \tag{16}$$

Objective function. In addition to satisfying the constraints, we seek to minimize the system stack usage.

$$\min \quad \sum_{\rho_k} m_k \tag{17}$$

V. Heuristic for Multicore Mapping

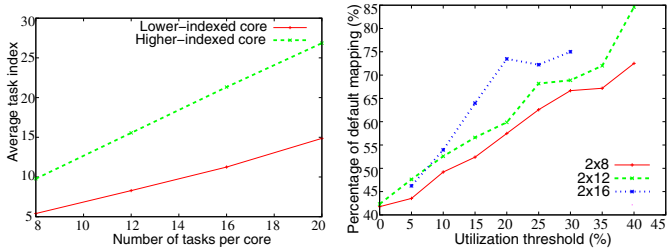
In this section, we propose a heuristic approach as an alternative to the MILP formulation, which are more scalable for large designs. We use the results from simulated annealing to study possible systematic behaviors of the (close to) optimal solutions. We leverage the observations on the solutions of SA for the task mapping heuristic (see Algorithm 2). After the task mapping is found, Algorithm 1 is used to assign task priority and preemption threshold.

Task default mapping. We first study the average task index mapped to each core. The tasks are indexed (starting from 1) according to rate monotonic policy, i.e. the larger the period, the smaller the index. As an example, Figure 1(a) shows the average task index of a dual-core platform with total numbers of tasks $n = 16, \cdots, 40$, where the line "higher-indexed core" ("lower-indexed core") plots the core with higher (lower) average index. For each n, we generate 100 random systems. For example, when a total of 16 tasks is mapped on two cores, the lower-indexed core has an average index of 5.407 while the higher-indexed core is 9.788. The ratio between the average index of the two cores is 0.5224. This ratio remains relatively constant for different number of tasks: for n =24, 32, and 40, the ratios are 0.5333, 0.5276, and 0.5536 respectively.

The average task index demonstrates a strong differentiation between the tasks mapped to the two cores. This core-index ratio reveals the preference of task mapping – *tasks with similar periods are possibly preferred to be mapped together*. We use it as a first step in our heuristics: order all tasks by increasing periods, order the cores by decreasing processing power, and then divide the tasks into m groups with roughly the same utilization. Such a mapping becomes the default mapping for each task, and we denote the default core of task τ_i as p_i. This is done in lines 1–2 of the heuristic Algorithm 2.

Algorithm 2: Heuristic Algorithm for Task Mapping

Input: Task set $\mathcal{T}$.
Output: m groups of tasks for each core
1 order $\mathcal{T}$ by increasing period, and order cores by decreasing speed;
2 finding default mapping for $\mathcal{T}$;
3 $U^T = \frac{100\% \times 3}{m}$;
4 **for** *each* $\tau_i \in \mathcal{T}_{heavy}$ **do**
5 map τ_i to its default core p_i;
6 **end**
7 $\mathcal{T} = \mathcal{T} \setminus \mathcal{T}_{\text{heavy}}$;
8 order each task τ_i in $\mathcal{T}$ by decreasing g_i as defined in (18);
9 select first N tasks $\mathcal{T}_{\text{exhaustive}}$ in $\mathcal{T}$ for exhaustive search;
10 $\mathcal{T} = \mathcal{T} \setminus \mathcal{T}_{seed}$;
11 **for** *each possible mapping of* $\mathcal{T}_{exhaustive}$ **do**
12 **for** *each task* $\tau_i \in \mathcal{T}$ **do**
13 **for** *each core* k **do**
14 temporarily map τ_i to core k;
15 $z_{i,k} = +\infty$;
16 **if** *system is schedulable* **then**
17 $z_{i,k}$ = increase in stack usage;
18 **end**
19 **end**
20 map τ_i to the core k_i with the smallest $z_{i,k}$;
21 **end**
22 **end**



(a) Average task index (b) Percent of default mapping

Fig. 1. Patterns from the solutions of SA.

Heavy tasks. We now study the probability of tasks in the SA solutions mapping to their default cores. For the systems containing 2×8 (dual-core with averagely 8 tasks on each core), 2×12, and 2×16 tasks, the percentage of tasks with default mapping are around 40%, as shown in the first point of the lines in Figure 1(b).

However, this behavior may depend on the utilization of the tasks. To validate this observation, we set a utilization threshold U^T and define the tasks with its utilization on the default core higher than U^T as *heavy tasks* $\mathcal{T}_{\text{heavy}}$. As shown in Figure 1(b), the percentage that the heavy tasks with default mapping increases with the utilization threshold, meaning higher utilization tasks would have a larger probability to be mapped on their default cores. In our heuristic Algorithm 2, we fix the heavy tasks (which also have a larger influence on the system schedulability) on their default cores, as detailed in lines 3–6 of the algorithm, where the utilization threshold U^T is set to be three times the average task utilization.

Mapping of remaining tasks. After fixing the mapping of the heavy tasks $\mathcal{T}_{\text{heavy}}$, we define a cost metric which combines the

consideration of task stack and utilization, as follows:

$$\forall \tau_i,\ \ g_i = \frac{S_i}{\max_j S_j} + \frac{\max_k U_{i,k}}{\max_{j,k} U_{j,k}} \tag{18}$$

Essentially, for task τ_i, its cost g_i is defined as the sum of two items: the first item is the ratio of its stack usage compared to the largest one among all the tasks in the system, and the second item is similarly defined for the task utilization. Intuitively, a task τ_i with a high cost g_i potentially has large impact on both the system stack usage (because of the large value of the first item) and the system schedulability (as captured by the second item).

We order the remaining tasks $\mathcal{T} \setminus \mathcal{T}_{\text{heavy}}$ by their decreasing cost g_i, and select the first few tasks $\mathcal{T}_{\text{exhaustive}}$ for exhaustive search. This is because of the difficulty to find a good heuristic for these tasks, as the system stack usage and schedulability are sensitive to their mapping. The number N of tasks selected for exhaustive search depends on the number of cores: for dual-core architectures, we select $N = 6$; for 4-core systems, $N = 4$. N is quite small compared with the number of all possible mapping combinations. This step is described in lines 8–10 of Algorithm 2.

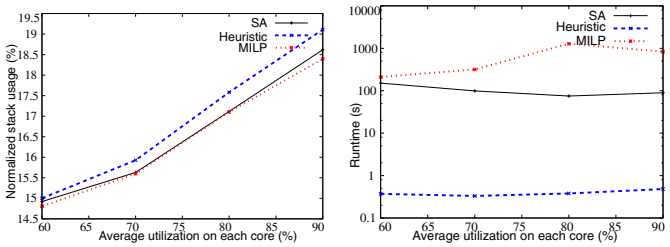
For all the other tasks, we use the following strategy. We temporarily map τ_i to each core k, and analyze the task system with τ_i and all the tasks already mapped to core k. We calculate the associated system stack usage increase $z_{i,k}$ (or the increase on the longest path in the preemption graph). If the system becomes unschedulable, $z_{i,k}$ is defined as infinity. After tried all the cores, τ_i is mapped to the core k_i with the smallest stack increase (the smallest value among all $z_{i,k}$). In the case that there are multiple cores with the same $z_{i,k}$, the tie breaker is to use the increase on the second longest path in the preemption graph, and so on. This step is detailed in the loop in lines 11–22 of Algorithm 2.

VI. Experimental Results

In this section, we use randomly generated task sets as well as industrial case studies, to assess the effectiveness and efficiency of our algorithms (including both MILP and the heuristic) in minimizing the stack usage. As an additional comparison, we develop a simulated annealing (SA) algorithm to achieve close to optimal solutions. In SA, for each transition, a task is randomly selected and moved to a new core. After the transition, the task priority is assigned using the deadline monotonic policy and the task preemption threshold is calculated using the optimal Algorithm PTAA [15]. We select the multicore platforms with 2 to 4 cores, as is the case for most hard real-time applications (see e.g., the product roadmap for automotive market from Freescale [1], and factory automation and automotive products from Infineon [2]).

1) Random Systems: We generate systems consisting of random task sets with $n = 16$ to 48 tasks. 100 task sets are generated and then examined for schedulability for each n. The periods of the tasks are generated by the product of one to three factors, each randomly drawn from three harmonic sets $(2, 4), (6, 12), (5, 10)$. This creates periods that are integer multiples of 2, 3, or 5, as is the case for most applications of practical interest. The task stack usage is uniformly distributed between 80 and 512 bytes.

The *first experiment* is to check the performance of the algorithms with respect to the number of tasks in the system.



(a) Normalized stack usage (b) Runtime

Fig. 3. Performance and runtime vs. system utilization for 2×12 tasks.

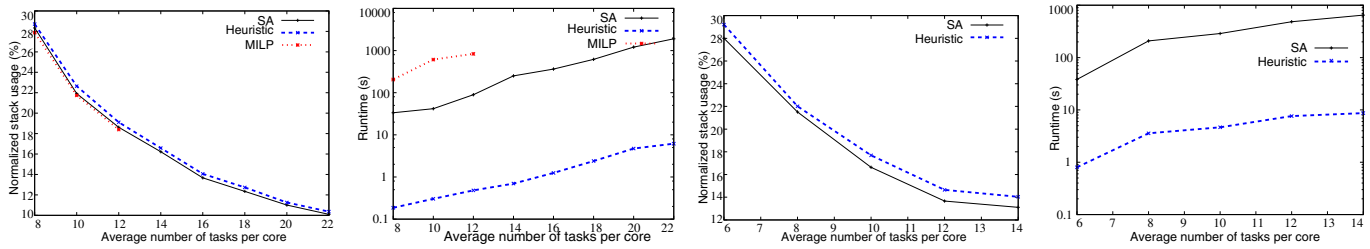
We keep the average CPU utilization constant ($U = 90\%$) while varying the number of tasks in the system. For dual-core architectures, Figures 2(a) and 2(b) plot the stack usage and runtime for MILP, simulated annealing and the heuristic algorithm (Algorithm 2), where the x-axis is the average number of tasks per core, and the y-axis is the normalized stack usage (divided by the sum of the task stacks, or the required stack space by fully-preemptive scheduling [11]). Algorithm 2 provides a reasonable solution both in terms of runtime and accuracy. Its stack usage is on average only 2.5% higher than SA. However, it runs about 249 times faster on average. For example, for the 2×22 case (dual-core architecture with 22 tasks on each core on average), the average runtime for the heuristic is 6.2 seconds, while SA takes about 1903 seconds. As MILP is time consuming, we are only able to run it for systems with no more than 2×12 tasks. It provides 1.2% stack usage saving than SA, but runs about 10 times slower.

We also compare the algorithms for systems with 4-core architectures. Figure 2(c) shows the normalized stack usage of the solutions found by the heuristic algorithm and SA for systems with 90% average core utilization, while Figure 2(d) plots their runtimes. Compared to the simulated annealing solution, on average the heuristic requires about 5.1% more stack spaces, but the runtime is 60 times shorter.

The *second experiment* is to check the stack usage with respect to the system utilization. We fix the number of tasks to be $n = 2 \times 12$, and vary the system utilization from $U = 60\%$ to 90%. For each U, 100 schedulable task sets are generated. The results are shown in Figure 3. The quality of the results from the heuristic does not depend much on the system utilization: it requires 1.3% more stack than MILP for 60% utilization, and 3.9% more for 90% utilization. In addition, the runtime is also almost independent from the system utilization.

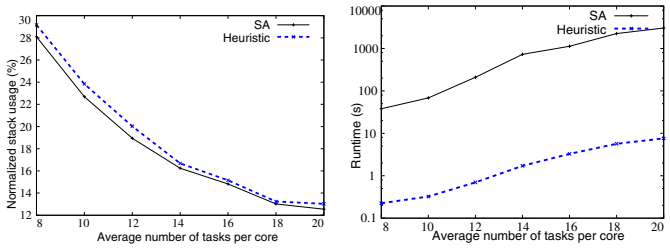
For all the above experiments, using preemption threshold scheduling to selectively disable preemption and allow stack space sharing can significantly save on system memory. For example, the normalized stack space is only about $14\% - 19\%$ of the sum of task stacks for systems with $n = 2 \times 12$ tasks.

The above experiments are done for systems with symmetrical cores, so that mapping a task on either core would require the same WCET and stack space. In order to evaluate the performance of the algorithms in systems with asymmetrical cores, we select as an example dual core processors where one core has a processing speed that is 50% of the other. Figure 4 compares normalized stack usage and runtime for the heuristic and simulated annealing. As can be seen from the figure, the heuristic can achieve a solution with similar quality as SA (only 3.8% more stack spaces) while the runtime is about two



(a) Stack size on 2-core platform (b) Runtime on 2-core platform (c) Stack size on 4-core platform (d) Runtime on 4-core platform

Fig. 2. Average system stack usage and algorithm runtime vs. number of tasks for CPU utilization = 90%.



(a) Stack usage (b) Runtime

Fig. 4. Performance and runtime on asymmetrical 2-core platform.

magnitudes smaller.

2) Industrial Case Studies: We also apply the optimization algorithms to two industrial case studies. The first one (available in [16]) consists of 42 tasks, and we map them to a dual-core processor. The task execution times are scaled to an average core utilization of 86.4%. All the three optimization algorithms (MILP, SA, and Algorithm 2) return the same solution with 8000 bytes of stack usage. The runtime for MILP is about 302 seconds, SA takes 170 seconds, while the heuristic is much faster with 1.8 seconds of runtime.

The second industrial case study (available in [9]) consists of a fuel injection embedded controller, which is a simplified version of the full control system with 90 function blocks (out of 200 in the real system), executed with 7 different periods (in ms): 4, 5, 8, 12, 50, 100, and 1000. The execution times are scaled up to an average core utilization of 94.1%. We assume a one-to-one mapping from function blocks to tasks. Due to the lack of data, the stack usage is set to be the maximum between 32 bytes and 2 times of the total size of its outgoing communication links. Because of the complexity of MILP, we only apply the heuristic and SA to the case study. The results are the following: the heuristic finds a solution of 1632 bytes in 52 seconds, which is much faster than SA with a small loss (3.0%) of solution quality, as SA takes more than 30 hours to finish, with a best solution of 1584 bytes.

VII. Conclusions

In this paper, we discuss the problem of design synthesis to minimize stack usage for systems with preemption threshold on multicore platforms. We present a mixed integer linear programming formulation, as well as a heuristic algorithm for task mapping and assigning task scheduling parameters. We perform extensive experiments using random systems and two industrial case studies to demonstrate the time efficiency and result quality of both optimization algorithms.

References

[1] Freescale. Automotive MCUs and MPUs. http://cache.freescale.com/files/microcontrollers/doc/roadmap/BRAUTOPRDCTMAP.pdf.

[2] Infineon. Microcontrollers. http://www.infineon.com/microcontrollers.

[3] S. Baruah. The limited-preemption uniprocessor scheduling of sporadic task systems. In *Proc. 17th Euromicro Conf. Real-Time Systems*, 2005.

[4] M. Bertogna, G. Buttazzo, and G. Yao. Improving feasibility of fixed priority tasks using non-preemptive regions. In *Proc. 32nd IEEE Real-Time Systems Symposium*, 2011.

[5] M. Bohlin, K. Hanninen, J. Maki-Turja, J. Carlson, and M. Nolin. Bounding shared-stack usage in systems with offsets and precedences. In *Proc. 20th Euromicro Conference on Real-Time Systems*, 2008.

[6] R. J. Bril, J. J. Lukkien, and W. F. Verhaegh. Worst-case response time analysis of real-time tasks under fixed-priority scheduling with deferred preemption. *Real-Time Systems* 42(1-3):63–119, Aug. 2009.

[7] G. Buttazzo, M. Bertogna, and G. Yao. Limited preemptive scheduling for real-time systems. a survey. *IEEE Trans. Industrial Informatics* 9(1):3–15, 2013.

[8] R. I. Davis and A. Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35, Oct. 2011.

[9] M. Di Natale, L. Guo, H. Zeng, and A. Sangiovanni-Vincentelli. Synthesis of multitask implementations of simulink models with minimum delays. *IEEE Trans. Industrial Informatics* 6(4):637–651, 2010.

[10] P. Gai, G. Lipari, and M. D. Natale. Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In *Proc. 22nd IEEE Real-Time Systems Symposium*, 2001.

[11] R. Ghattas and A. G. Dean. Preemption threshold scheduling: Stack optimality, enhancements and analysis. In *Proc. 13th IEEE Real Time and Embedded Technology and Applications Symposium*, 2007.

[12] K. Hanninen, J. Maki-Turja, M. Bohlin, J. Carlson, and M. Nolin. Determining maximum stack usage in preemptive shared stack systems. In *Proc. 27th IEEE International Real-Time Systems Symposium*, 2006.

[13] H. Kopetz, R. Obermaisser, C. El Salloum, and B. Huber. Automotive software development for a multi-core system-on-a-chip. In *Proc. 4th Workshop on Software Engineering for Automotive Systems*, 2007.

[14] J. Regehr. Scheduling tasks with mixed preemption relations for robustness to timing faults. In *Proc. 23rd IEEE Real-Time Systems Symposium*, 2002.

[15] M. Saksena and Y. Wang. Scalable real-time system design using preemption thresholds. In *Proc. Real-Time Systems Symposium*, 2000.

[16] K. Tindell, A. Burns, and A. Wellings. Allocating hard real-time tasks: an NP-hard problem made easy. *Real-Time Systems* 4(2):145–165, 1992.

[17] Y. Wang and M. Saksena. Scheduling fixed-priority tasks with preemption threshold. In *Proc. 6th International Conference on Real-Time Computing Systems and Applications*, 1999.

[18] G. Yao and G. Buttazzo. Reducing stack with intra-task threshold priorities in real-time systems. In *Proc. 10th ACM international conference on Embedded software*, 2010.

[19] G. Yao, G. Buttazzo, and M. Bertogna. Bounding the maximum length of non-preemptive regions under fixed priority scheduling. In *Proc. 15th Conf. Embedded & Real-Time Computing Systems & Applications*, 2009.

[20] H. Zeng and M. Di Natale. An efficient formulation of the real-time feasibility region for design optimization. *IEEE Trans. Computers* 62(4):644–661, 2013.

[21] H. Zeng, M. Di Natale, and Q. Zhu. Optimizing stack memory requirements for real-time embedded applications. In *Proc. ETFA*, 2012.

[22] H. Zeng, M. Di Natale, and Q. Zhu. Minimizing Stack and Communication Memory Usage in Real-time Embedded Applications. *Accepted to ACM Trans. Embedded Computing Systems*.