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.

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 runtime 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 [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 $T$ of $n$ periodic tasks scheduled on a multicore platform with $m$ cores. Each core $p_k$ is scheduled independently using a static-priority uniprocessor algorithm. Each task $\tau_i$ is associated with a period $T_i$, a deadline $D_i \leq T_i$ that is not larger than its period, a priority $\pi_i$ (the higher the number, the higher the priority), and a preemption threshold $\gamma_i \geq \pi_i$, that is assumed once $\tau_i$ starts its execution and retained until it finishes. At runtime, $\tau_i$ is allowed to preempt $\tau_j$ only if $\pi_i > \gamma_j$. We assume tasks have arbitrary offsets. For each core $p_k$, task $\tau_i$ requires a worst-case execution time (WCET) $C_{i,k}$ possibly different than others as $p_k$ 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 AbInt) that use its object file as the input.

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 $\tau_i$, denoted as $h_i$, is defined as the maximum blocking time $\tau_i$ 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, Priority Assignment algorithm assuming Deadline Monotonic and Maximum Preemption Threshold, 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]**

<table>
<thead>
<tr>
<th>Line</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>$Unassigned = T$;</td>
</tr>
<tr>
<td>2</td>
<td>for each priority level $p = 1$ to $</td>
</tr>
<tr>
<td>3</td>
<td>for each task $\tau_i$ in $Unassigned$ do</td>
</tr>
<tr>
<td>4</td>
<td>$p_i = p$;</td>
</tr>
<tr>
<td>5</td>
<td>assume DM policy for the set $Unassigned\setminus{\tau_i}$;</td>
</tr>
<tr>
<td>6</td>
<td>use PTAA to assign preemption threshold to $Unassigned$;</td>
</tr>
<tr>
<td>7</td>
<td>calculate blocking time limit $h_i$ for $\tau_i$;</td>
</tr>
<tr>
<td>8</td>
<td>if $h_i \leq \beta$ then</td>
</tr>
<tr>
<td>9</td>
<td>$a_i = h_i$;</td>
</tr>
<tr>
<td>10</td>
<td>else</td>
</tr>
<tr>
<td>11</td>
<td>$a_i = \beta - h_i$;</td>
</tr>
<tr>
<td>12</td>
<td>end</td>
</tr>
<tr>
<td>13</td>
<td>end</td>
</tr>
<tr>
<td>14</td>
<td>select $\tau_i$ from $Unassigned$ with the largest $a_i$;</td>
</tr>
<tr>
<td>15</td>
<td>$p_i = p$;</td>
</tr>
<tr>
<td>16</td>
<td>$Unassigned = Unassigned \setminus {\tau_i}$;</td>
</tr>
<tr>
<td>17</td>
<td>end</td>
</tr>
<tr>
<td>18</td>
<td>use PTAA to assign task preemption threshold;</td>
</tr>
</tbody>
</table>

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

\[ a_{i,k} = \begin{cases} 1, & \text{if } \tau_i \text{ is mapped to core } \rho_k \\ 0, & \text{otherwise} \end{cases} \quad (1) \]

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

Each task is mapped to exactly one core

\[ \sum_{\rho_k} a_{i,k} = 1 \quad (3) \]

Also, \( u \) and \( a \) variables should be consistent

\[ u_{i,j,k} \leq a_{i,k}, \quad u_{i,j,k} \leq a_{j,k}, \quad u_{i,j,k} \geq a_{i,k} + a_{j,k} - 1 \quad (4) \]

Preemption threshold assignment. For each pair of tasks \( \tau_i \) and \( \tau_j \), \( \tau_i \) cannot preempt \( \tau_j \) if and only if \( \pi_i \leq \pi_j \). A set of binary variables is used to encode this condition

\[ q_{i,j} = \begin{cases} 1, & \text{if } \pi_i \leq \pi_j \\ 0, & \text{otherwise} \end{cases} \quad (5) \]

If task \( \tau_j \) has a priority higher than or equal to \( \tau_i \), then \( \tau_i \) cannot preempt \( \tau_j \).

\[ \forall i,j : \pi_i \leq \pi_j, \quad q_{i,j} = 1 \quad (6) \]

If \( \tau_i \) cannot preempt \( \tau_j \), then any task \( \tau_k \) with priority \( \leq \pi_i \) cannot preempt \( \tau_j \), too; conversely, if \( \tau_i \) can preempt \( \tau_j \), any task with priority \( \geq \pi_i \) can preempt \( \tau_j \).

\[ \forall k : \pi_k \leq \pi_i, \quad q_{k,j} \geq q_{i,j} \]

\[ \forall k : \pi_k \geq \pi_i, \quad q_{i,j} \leq q_{k,j} \quad (7) \]

\( v_{i,j,k} \) 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 \( (1 - q_{i,j}) \).

\[ v_{i,j,k} \leq u_{i,j,k}, \quad v_{i,j,k} \leq q_{i,j}, \quad v_{i,j,k} \geq u_{i,j,k} + q_{i,j} - 1 \]

\[ v_{i,j,k} + w_{i,j,k} = u_{i,j,k} \quad (8) \]

Blocking time. Each task \( \tau_i \) can only be blocked once, with a worst-case blocking time \( b_i \) equal to the maximum WCET of any lower priority task \( \tau_j \) mapped to the same core with a preemption threshold \( \gamma_j \geq \pi_i \).

\[ \forall j : \pi_j \leq \pi_i, \quad b_i \geq \sum_{\rho_k} (C_{j,k} \cdot v_{i,j,k}) \quad (9) \]

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 \( \tau_i \) is expressed as

\[ \bigvee_{(s,f) \in I_i} \left\{ b_i + \sum_{j : \pi_j < \pi_i} rb_{f_j}(s) < s \quad (10) \right\} \]

where \( I_i \) is the set of candidate start and finish time point pairs for \( \tau_i \), and \( rb_{f_j}(t) = \frac{f_j}{T_j} \) denotes the request bound function of \( \tau_j \) within the interval of length \( t \). \[20\] describes the formulation of (10) in MILP framework, the calculation of \( I_i \) 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 \( \tau_i \) (thus \( \exists k, u_{j,i,k} = 1 \)), which can be rewritten as

\[ \sum_{j : \pi_j > \pi_i} \sum_{\rho_k} \left( \frac{s}{T_j} C_{j,k} \cdot v_{j,i,k} \right) \quad (11) \]

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 \( \tau_i \) but not able to preempt \( \tau_i \) (thus \( \exists k, u_{j,i,k} = 1 \) and \( q_{j,i} = 1 \), or equivalently \( v_{j,i,k} = 1 \))

\[ \sum_{j : \pi_j > \pi_i} \sum_{\rho_k} \left( \frac{f_j}{T_j} C_{j,k} \cdot v_{j,i,k} \right) \quad (12) \]

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

\[ \sum_{j : \pi_j > \pi_i} \sum_{\rho_k} \left( \frac{f_j}{T_j} C_{j,k} \cdot w_{j,i,k} \right) \quad (13) \]

The task execution time is

\[ c_i = \sum_{\rho_k} (C_{i,k} \cdot a_{i,k}) \quad (14) \]

Stack usage. For each pair of tasks \( \tau_i \) and \( \tau_j \), if there exists \( k \) such that \( u_{i,j,k} = 1 \), it must be that \( \tau_i \) and \( \tau_j \) are mapped to the same core \( k \), and \( \tau_i \) can preempt \( \tau_j \). We denote \( \tau_i \preceq \tau_j \leftrightarrow u_{i,j,k} = 1 \). 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 $\tau_i$ to $\tau_j$ if $\tau_i \preceq \tau_j$. The maximum stack usage is the largest total stack usage for a set of tasks where each pair has $\tau_i \preceq \tau_j$, 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 $\tau_i$ and each core $\rho_k$; if $a_{i,k} = 1$, $x_{i,k}$ is the sum of stack usages for $\tau_i$ and the ones that can preempt $\tau_i$; 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 $\tau_i$ 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 $\tau_j$ that is mapped to the same core and can preempt $\tau_i$.

$$\forall j : \pi_j \geq \pi_i, \quad x_{i,k} \leq M \cdot a_{i,k}$$ (15)

The stack usage $m_k$ of core $k$ is

$$\forall i, \quad m_k \geq x_{i,k}$$ (16)

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

$$\min \sum_{\rho_k} m_k$$ (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 $\tau_i$ as $p_i$. This is done in lines 1–2 of the heuristic Algorithm 2.

**Heavy tasks.** We now study the probability of tasks in the SA solutions mapping to their default cores. For the systems containing $2 \times 8$ (dual-core with averagely 8 tasks on each core), $2 \times 12$, and $2 \times 16$ 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 $T_{\text{heavy}}$. 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 $T_{\text{heavy}}$, we define a cost metric which combines the

![Algorithm 2: Heuristic Algorithm for Task Mapping](image_url)
consideration of task stack and utilization, as follows:

\[ \forall \tau_i, \ g_i = \frac{S_i}{\max_j S_j} + \frac{\max_k U_{i,k}}{\max_j \max_k U_{j,k}} \]  

(18)

Essentially, for task \( \tau_i \), 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 \( \tau_i \) 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 \( T \setminus T_{\text{heavy}} \) by their decreasing cost \( g_i \), and select the first few tasks \( T_{\text{heavy}} \) 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 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 \( \tau_i \) to each core \( k \), and analyze the task system with \( \tau_i \) 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, \( \tau_i \) 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 task usage. As an additional comparison, we develop a simulated annealing (SA) algorithm to achieve close to optimal solutions. In SA, for each 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. 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 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, 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
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 thresholds 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