Scratchpad Memory Aware Task Scheduling with Minimum Number of Preemptions on a Single Processor

Qing Wan Hui Wu Jingling Xue

School of Computer Science and Engineering
University of New South Wales
Sydney, NSW 2052, Australia
e-mail: {qingwan, huiw, jingling@cse.unsw.edu.au}

Abstract— We propose a unified approach to the problem of scheduling a set of tasks with individual release times, deadlines and precedence constraints, and allocating the data of each task to the SPM (Scratchpad Memory) on a single processor system. Our approach consists of a task scheduling algorithm and an SPM allocation algorithm. The former constructs a feasible schedule incrementally, aiming to minimize the number of preemptions in the feasible schedule. The latter allocates a portion of the SPM to each task in an efficient way by employing a novel data structure, namely, the preemption graph. We have evaluated our approach and a previous approach by using six task sets. The results show that our approach achieves up to 20.31% on WCRT (Worst-Case Response Time) reduction over the previous approach.

I. Introduction

In a typical embedded system, there are multiple concurrent tasks. Tasks may be subject to release times, deadlines, and precedence constraints. The release time and the deadline of a task specify its earliest start time and the latest completion time in a feasible schedule. The precedence constraints specify the data and control dependencies between tasks. In hard real-time embedded systems, it is essential to find a feasible schedule for all the tasks at the design stage.

The problems of scheduling a set of tasks with various constraints have been extensively studied [8,15,16]. Most scheduling problems are NP-complete. On a single processor. if tasks are preemptible, the EDF (earliest deadline first) strategy is guaranteed to find a feasible schedule for a set of tasks with individual release times, deadlines and precedence constraints whenever one exists [8]. However, if tasks are not preemptible, the problem of finding a schedule with minimum lateness for a set of independent tasks with individual release times and deadlines on a single processor is NP-complete [8].

Scratchpad memory is the on-chip SRAM managed by the compiler. It is an attractive alternative to cache in embedded systems due to its three major advantages. Firstly, it consumes less energy than cache. Secondly, it is easier to compute the WCET (Worst-Case Execution time) of a task because the access time of each variable or instruction is known at compile time. Thirdly, the compiler can usually hide data hazards in modern RISC processors without any hardware support as the latency of each data access to SPM is known at compile time. However, SPM also introduces additional challenges. One major challenge is that the task scheduling problem and the SPM allocation problem are mutually dependent. On one hand, the WCET of a task is dependent on the size of the SPM allocated to it. On the other hand, the size of the SPM allocated to each task is dependent on whether this task is preempted in the schedule or not. If a task T_i is preempted by another task T_j, then T_i and T_j cannot use the same section of SPM to store data assuming there is no dynamic SPM reallocation.

In this paper, we study the problem of scheduling a set of tasks with individual release times, deadlines and precedence constraints on a single processor of an embedded system where SPM is used to replace data cache, and the problem of allocating the SPM to each task. We assume that the target embedded system is a hard real-time system where the deadline of each task must be met. We make the following major contributions.

1. We propose a novel unified approach to the task scheduling problem and and the SPM allocation problem. The unified approach consists of a task scheduling algorithm and an SPM allocation algorithm. The task scheduling algorithm aims at minimizing the number of preemptions in a feasible schedule for the task set. The SPM allocation algorithm employs a novel data structure, namely, the preemption graph, to efficiently allocate SPM to tasks.

2. We have evaluated our approach and the one proposed by Suhendra *et al.* [19] by using six task sets with tasks selected from three benchmark suites: Powerstone [13], Mälardalen WCET Benchmarks [7], SNU real-time benchmarks [14], and an open-source UAV (Unmanned Aerial Vehicle) control application from PapaBench [5]. For all the task sets, our approach achieves a maximum improvement of 20.31% on the WCRT reduction.

The rest of this paper is organized as follows. Section II describes the system model and key definitions. Section III shows how to determine the maximum SPM size for each task. Section IV describes our unified approach to task scheduling and SPM allocation. Section VI describes related work. Section V presents our experimental results, followed by the conclusion section in Section VII.

II. System Model and Definitions

The target hard real-time embedded system uses a single processor where an SPM is used to replace data cache. The size

978-1-4673-3030-5/13/$31.00 ©2013 IEEE

of the SPM is m bytes. The SPM occupies a contiguous section of the processor's memory space. The start address of the SPM is 0. The SPM is only used to store the local (stack) data of tasks. The problem of allocating the global data, heap data and code of a task set to SPM will be studied in future work. There is a set $V = \{T_1, T_2, \cdots, T_n\}$ of n tasks to be executed on the processor. Each task is preemptible by any other tasks. However, our unified algorithm for task scheduling and SPM allocation preempts a task only if it is necessary. Tasks have individual release times, deadlines and precedence constraints. The precedence constraints are represented by a DAG (directed acyclic graph) $G = (V, E)$, where $V = \{T_1, T_2, \cdots, T_n\}$ is the set of tasks, $E = \{(T_i, T_j) : T_j$ can be executed only after T_i finishes $\}$ is a set of precedence constraints between tasks. Each task T_i has the following attributes:

1. Pre-assigned release time $R(T_i)$,
2. Pre-assigned deadline $D(T_i)$,
3. The maximum size $s_i (s_i \leq m)$ of the SPM space needed by T_i, and
4. The worst-case execution time $wcet(T_i, x)$ when an SPM with size of x bytes is allocated to T_i.

Given a schedule for a set of tasks with individual release times, deadlines and precedence constraints, a task T_i is ready at time t if all its predecessors have been completed by t and t is greater than or equal to the release time of T_i. The ready time of T_i is the earliest time at which T_i is ready.

EDF is a classical scheduling strategy. There are two versions, preemptive EDF (pEDF) and non-preemptive EDF (npEDF). The npEDF schedules a task only when the current running task is completed. The pEDF performs scheduling whenever a new task is ready. Both pEDF and npEDF schedule a ready task with the smallest deadline.

Definition 1 *Given a schedule σ and a task T_i, the live range of T_i, denoted as $L(T_i)$, is a time interval $[S(T_i), F(T_i)]$, where $S(T_i)$ and $F(T_i)$ are the start time and the finish time, respectively, of T_i in σ.*

Given two tasks, they can share a section of SPM iff their live ranges do not overlap.

Definition 2 *Given a schedule σ for a set of tasks, the interference graph of σ is an undirected graph $G(\sigma) = (V, E)$, where $V = \{T_1, T_2, \cdots, T_n\}$ is the set of tasks, and $E = \{(T_i, T_j) : T_i, T_j \in V$ and $L(T_i) \cap L(T_j) \neq \emptyset\}$.*

Definition 3 *Given a schedule, a task T_i is said to preempt a task T_j iff one of the following conditions holds: 1) T_i preempts T_j directly. 2) T_i is scheduled immediately after the completion of another task T_s, and T_s preempts T_j in the schedule.*

Notice that our definition of preemption is a generalization of the traditional one.

Definition 4 *Given a schedule σ for a set of tasks, the preemption graph of σ is a directed graph $G = (V, E)$, where $V = \{T_1, T_2, \cdots, T_n\}$ is the set of tasks, and $E = \{(T_i, T_j) : T_i, T_j \in V$ and T_j preempts T_i in $\sigma\}$.*

It is easy to see that the preemption graph of any schedule constructed by using an EDF scheduler is a forest. The preemption graph is a key data structure of our unified algorithm for task scheduling and SPM allocation. We can easily prove that for each path in a preemption graph, the live ranges of any two tasks on the path overlap.

Definition 5 *Given a set of tasks with precedence constraints, individual release times and deadlines, the edge-consistent deadline of a task T_i, denoted $D'(T_i)$, is recursively defined as follows. $D'(T_i) = \min\{D(T_i), \min\{D'(T_j) - wcet(T_j, s_j)$: T_j is an immediate successor of T_i in the precedence graph$\}\}$.*

III. Determining The SPM Sizes of Individual Tasks

Our unified algorithm for task scheduling and SPM allocation needs to know the maximum SPM size s_i of each task T_i. The impact of SPM on the WCET of each task may vary. For some tasks, SPM may drastically reduce their WCETs. For some other tasks, SPM may not be very effective. Therefore, it is very important to determine the maximum SPM size of each task in a fair manner.

The approaches proposed in [20, 21] assume that the maximum SPM size of each task is known without proposing any approach to determining the maximum SPM size of each task in a fair manner. The approaches proposed in [18, 19] use a heuristic based on ILP (Integer Linear Programming). Since the tasks in [18, 19] do not have individual deadlines, their ILP based heuristics are not applicable to our task model.

Next, we propose a new approach for determining the maximum size of the SPM for each task based on our previous work on allocating variables of a single task to SPM [23]. For each variable v_i, we define a benefit vector $benefit(v_i)$ as follows.

$$benefit(v_i) = \frac{l - l(v_i)}{size(v_i)} \quad (1)$$

where l is the vector of the lengths, in non-increasing order, of the k longest paths of the task immediately before allocating v_i to the SPM, $l'(v_i)$ is the vector of the lengths, in non-increasing order, of the k longest paths of the task immediately after allocating v_i to the SPM, and $size(v_i)$ is the size of v_i. Intuitively, the benefit vector of a variable v_i is the normalized contribution of v_i to the k longest path lengths of the task. To compare any two benefit vectors, we use lexicographical ordering.

In our definition of benefit vector, k is a parameter. On one hand, the larger the value of k, the more accurate a benefit vector. On the other hand, the larger the value of k is, the higher time complexity for computing a benefit vector.

In order to determine the maximum SPM size for each task in a fair manner, we introduce a threshold benefit vector α_{min} for all the tasks. For each task, we select a variable as an SPM resident only if its benefit vector is greater than α_{min}. The threshold benefit vector is a parameter of our approach. Given a specific task set, its value needs to be tuned for the best performance of a given task set.

We determine the maximum SPM size s_i of each task T_i as follows: Keep selecting a variable of T_i on the longest path with the maximum benefit vector being greater than α_{min}, and

allocating it to the SPM of the task until no variable can be selected. For more details on selecting a most beneficial variable and allocating it to SPM, we refer to [23].

IV. Unified Task Scheduling and SPM Allocation

Given a set S of tasks with individual release times, deadlines, and precedence constraints, our objective is to find a feasible schedule for S on a single processor with an SPM with a size of m bytes to store local data of the tasks. A feasible schedule is the one satisfying all the constraints.

Our unified approach to task scheduling and SPM allocation consists of two major parts: the task scheduling algorithm and the SPM allocation algorithm. The task scheduling algorithm aims at minimizing the number of preemptions when finding a feasible schedule for the task set. By default, it uses the npEDF scheduling. It uses the pEDF scheduling only if a task misses its deadline under the npEDF scheduling. Initially, no task is preempted. Therefore, the whole SPM is allocated to each task T_i. When a task currently scheduled meets its deadline, the task scheduling algorithm calls the SPM allocation algorithm to allocate SPM to the task and each of the predecessors of the task in the preemption graph. During the execution of our unified approach, if a task is preempted. the SPM size of each predecessor of the task may decrease.

Our unified approach uses the following variables:

- $D(T_i)$: the deadline of T_i.
- $wcet(T_i)$: the current worst-case execution time of task T_i,
- $accu\_time(T_i)$: the accumulated execution time of task T_i,
- $preempted(T_i)$: a Boolean variable, denoting if task T_i has been preempted before,
- $miss(T_i)$: a Boolean variable, denoting if task T_i has missed its deadline before, and
- $start$: storing successive scheduling points. A scheduling point is a time point at which a task is scheduled.

Our unified approach works as follows:

1. Compute the edge-consistent deadlines for all the tasks, and initialize the relevant data structures.
2. If a task T_i meets its deadline under the npEDF, do the following:
   (a) If T_i is not in the preemption graph, add T_i to the preemption graph.
   (b) If T_i has not been preempted before, do the following:
      i. Find the task T_j that is most recently preempted and has not finished.
      ii. If T_j exists, add the directed edge (T_j, T_i) to the preemption graph. and call our incremental SPM allocator to allocate SPM to T_i and each of its predecessors in the preemption graph.
3. If a task T_i misses its deadline by the npEDF scheduling. Let T_j be the task scheduled at the release time of T_i. The following two cases are distinguished:
   (a) The deadline of T_j is not larger than T_i. In this case, no feasible schedule exists.
   (b) The deadline of T_j is larger than that of T_i. In this case, do the following:
      i. Preempt T_j at the ready time of T_i.
      ii. Find the set C of all the tasks scheduled after T_j in the current schedule.
      iii. Remove all the edges incident to the tasks in C in the current schedule from the preemption graph.
      iv. Undo the current schedule for C, and continue to schedule all the unscheduled tasks, including the tasks in C.
4. Repeat steps 2 and 3 until all the tasks have been scheduled or a task cannot meet its deadline.

The details of our unified algorithm for task scheduling and SPM allocation are shown in Algorithms 1 and 2.

Algorithm 1: Our unified approach to task scheduling and SPM allocation

Input: A set S of tasks with individual release times, deadlines and precedence constraints, an SPM with a size of m bytes, the maximum SPM size s_i needed by each task T_i, and a processor P

Output: A feasible schedule and an SPM allocation scheme for S

```
1  Compute the edge-consistent deadline for each task;
2  foreach task T_i ∈ S do
3      preempted(T_i) = false;
4      Compute wcet(T_i, s_i);
5      wcet(T_i) = wcet(T_i, s_i);
6      accu_time(T_i) = 0;
7      D(T_i) = D'(T_i);
8  Create an empty preemption Graph G;
9  start = the earliest release time of all the tasks in S;
10 Scheduler_Allocator(S, start);
```

The SPM allocation algorithm works incrementally based on the current partial SPM allocation scheme. It is called by the task scheduling algorithm whenever a new task T_i is successfully scheduled. When being called, it starts with T_i and works toward the source (root) task along the path from T_i to the root in the preemption graph. For each task T_j visited in the preemption graph, our SPM allocation algorithm tries to allocate s_j bytes to it. If s_j bytes is not available, it allocates the remaining free SPM space to T_j considering the interference constraints. Once a task T_k cannot be allocated s_k bytes, all its predecessors in the preemption graph will not be allocated any SPM space. For each task T_i, we introduce four variables, $start\_addr(T_i)$, $end\_addr(T_i)$, $spm\_size(T_i)$, and $wcet(T_i)$, where $start\_addr(T_i)$ and $end\_addr(T_i)$ are the start address and the end address of T_i, respectively, in the

Algorithm 2: $Scheduler\_Allocator(S, start)$

Input: A set S of tasks, the earliest release time $start$ of all the tasks
Output: Task schedule and SPM allocation results

while $S \neq \emptyset$ **do**
  Find a task T_i in S that is ready at time $start$ and has the earliest deadline among all the ready tasks;
  if $start + wcet(T_i) - accu\_time(T_i) \leq D(T_i)$ **then**
    Schedule T_i at time $start$;
    $start = start + wcet(T_i) - accu\_time(T_i)$;
    $accu\_time(T_i) = wcet(T_i)$;
    $S = S - \{T_i\}$;
    if T_i *is not in* G **then**
      Add T_i to G;
    if $preempted(T_i) = false$ **then**
      Let T_j be the most recently preempted task that has not finished by time $start$;
      if T_j *exists* **then**
        Add (T_j, T_i) to G;
        `// Re-allocate SPM to` T_i `and all its predecessors`
        $Incr\_SPM\_Allocator(T_i)$;
  else
    if *the ready time of* $T_i = start \parallel miss(T_i) = true$ **then**
      return No feasible schedule;
    else
      $start$ = the release time of T_i;
      Let T_j be the task executing at time $start$ in the current schedule;
      if $D(T_j) \leq D(T_i)$ **then**
        return No feasible schedule;
      else
        $miss(T_i) = true$;
        $preempted(T_j) = true$;
        $C = \{T_j\} \cup \{T_k : T_k$ is scheduled after T_j in the current schedule$\}$;
        Preempt T_j at time $start$;
        Undo the partial schedule for all the tasks scheduled after time $start$;
        foreach *task* $T_k \in C$ **do**
          Recalculate $accu\_time(T_k)$;
          Remove T_k and all its incident edges from G;
          $S = S \cup \{T_k\}$;
        $Scheduler\_Allocator(S, start)$;
        break;
return;

Algorithm 3: $Incr\_SPM\_Allocator(T_i)$

Input: A preemption graph G, an allocation scheme for all the tasks in the preemption graph, and a new task T_i
Output: A new SPM allocation scheme for all the tasks in G

$start\_addr(T_i) = 0$;
$end\_addr(T_i) = \min\{m - 1, s_i - 1\}$;
T_j = the parent of T_i in G;
while $T_j \neq null$ **do**
  $temp = \max\{end\_addr(T_s) : T_s$ is a child of T_j in $G\}$;
  if $temp \geq m - 1$ **then**
    `// No SPM space for` T_j
    $start\_addr(T_j) = m$;
    $end\_addr(T_j) = m$;
    $spm\_size(T_j) = 0$;
    $wcet(T_j) = wcet(T_j, 0)$;
  else
    $start\_addr(T_j) = temp + 1$;
    $end\_addr(T_j) = \min\{m - 1, temp + s_i\}$;
    $spm\_size(T_j) =$ $end\_addr(T_j) - start\_addr(T_j) + 1$;
    if $spm\_size(T_j) < s_i$ **then**
      `// Not enough SPM space for` T_j
      Compute $wcet(T_j, spm\_size(T_j))$;
      $wcet(T_j) = wcet(T_j, spm\_size(T_j))$;
  T_j = the parent task of T_j in G;

SPM, spm_size(T_i) is the size of the SPM allocated to T_i, and $wcet(T_i)$ is the WCET of T_i. For a leaf task, its start address is 0. For a non-leaf task that can be allocated to SPM, its start address is one plus the maximum end address of all its children. Our SPM allocation algorithm is shown in Algorithm 3.

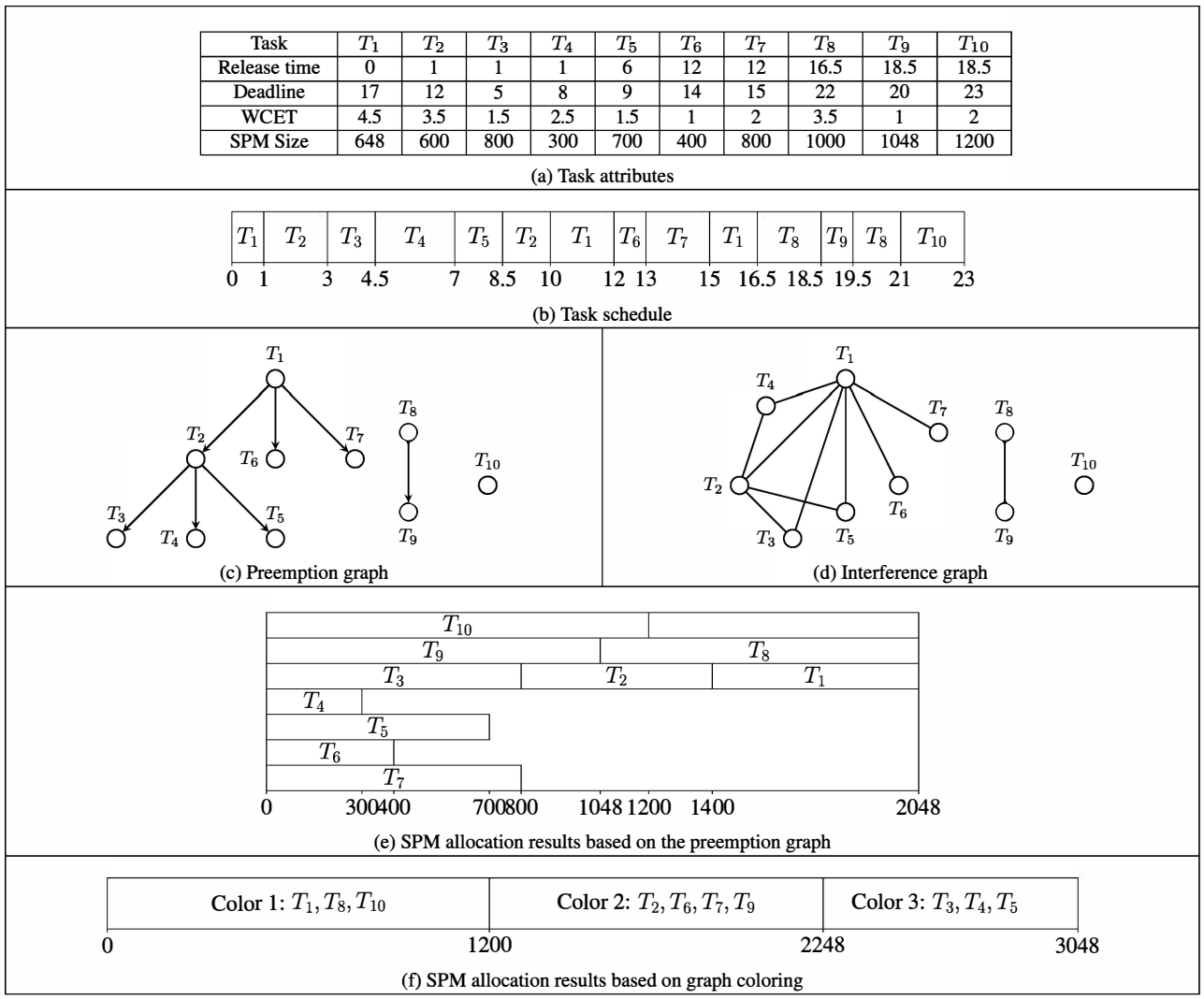
Next, we use an example to explain how our unified approach to task scheduling and SPM allocation works. We also use it to compare our SPM allocation algorithm with the graph coloring based SPM allocation technique proposed in [19].

There are a set of 10 independent tasks to be executed on a single processor where an SPM of $2K$ bytes is used to store local data of the tasks. The task attributes are shown in Figure 1a, where the SPM size is the size of the SPM space needed by each task, and the WCET of each task is its WCET when its SPM size requirement is satisfied. For example, if T_1 is allocated 648 bytes of SPM space, its WCET is 4.5.

A feasible schedule found by our task scheduling algorithm is shown in Figure 1b, and the preemption graph of this schedule is constructed as in Figure 1c. Based on the preemption graph, the SPM allocation scheme computed by our SPM allocation algorithm is shown in Figure 1e. As we can see, all the tasks are allocated to the SPM.

Notice that by our SPM allocation algorithm, a task may not be fully allocated to the SPM. In this example, if we change the SPM size requirement of T_1 to $1K$ bytes, our algorithm will allocate only 648 bytes of SPM space to T_1.

Consider the final schedule shown in Figure 1b. For simplicity, we ignore the start times and finish times of all the tasks, and only consider their execution order. Next, we will show how the graph coloring based SPM allocation technique proposed by in [19] works. The interference graph of all the tasks in the final schedule is shown in Figure 1d. After applying the



Task	T_1	T_2	T_3	T_4	T_5	T_6	T_7	T_8	T_9	T_{10}
Release time	0	1	1	1	6	12	12	16.5	18.5	18.5
Deadline	17	12	5	8	9	14	15	22	20	23
WCET	4.5	3.5	1.5	2.5	1.5	1	2	3.5	1	2
SPM Size	648	600	800	300	700	400	800	1000	1048	1200

Fig. 1.: An example comparing preemption graph based and graph coloring based SPM allocation

coloring algorithm, we have three colors. T_1, T_8 and T_{10} are assigned color 1, T_2, T_6, T_7 and T_9 are assigned color 2, and T_3, T_4 and T_5 are assigned color 3. The SPM is partitioned into three disjoint sections for color 1, color 2 and color 3, respectively, and all the tasks with the same color share a section of the SPM, as shown in Figure 1f. As we can see, in order to place all the tasks in the SPM, the size of the SPM must be at least 3048 bytes, in contrast to the SPM size of $2K$ bytes needed by our SPM allocation algorithm. As a result, their approach cannot find a feasible schedule given an SPM size of $2K$ bytes.

V. Experimental Results

A. Experiment Setup

In order to evaluate our unified approach, we created six task sets as shown in Table I. We selected 20 applications from three benchmark suites: Powerstone [13], Mälardalen WCET Benchmarks [7] and SNU real-time benchmarks [14]. The statistics of all the applications are given in Table II. Each application is a task. Each task set consists of a subset of tasks from the 20 applications. There are no precedence constraints for the first five task sets.

TABLE I
: Task sets

Groups	Applications
Set1	minver, jfdctint, fdct, statemate, ludcmp, compress, nsichneu, qurt, fir, select
Set2	lms, adpcm, crc, engine, pocsag, matmult, jpeg, fft1k, edn, v42
Set3	all the tasks from Set1 and Set2
Set4	adpcm, jfdctint, ludcmp, edn, fft1k, fdct, lms, minver, jpeg, matmult
Set5	statemate, nsichneu, qurt, select, fir, crc, engine, pocsag, v42, compress
Set6	PapaBench

The 6th task set comes from a real-life open-source UAV control application, the PapaBench [5]. It consists of 28 tasks and operates in two modes: fly by wire and autopilot, which means that the aircraft can be controlled both manually and automatically. Each mode consists of several tasks to control the

TABLE II
: Tasks information

Benchmark	Data size (bytes)	WCET with SPM (cycles)
ludcmp	21680	12449
qurt	92	17856
minver	2604	14264
engine	478	4859160
pocsag	1216	1268830
jpeg	77561	74621273
statemate	227	21326
nsichneu	1588	202212
fft1k	16484	5223830
lms	1268	1762300
jfdctint	340	58678
adpcm	2212	256275
fir	592	150293
crc	1079	47786
compress	1837	14758
matmult	4840	167420
fdct	220	6275
select	124	6644
edn	1884	164364
v42	40973	40869080

aircraft and communicate with ground station. For our evaluation purposes, we separated the tasks from the original implementation, and maintained the control dependencies among these tasks. The statistics of all the tasks in the 6th task set can be found in [19].

We implemented both our unified approach and the CR approach proposed in [19] . When determining the maximum SPM size for each task, we set k to 2, and the threshold benefit vector α_{min} to $(0.1, 0)$. Since the CR approach does not handle individual deadlines, we revised it such that the interference between two tasks cannot be eliminated if delaying one task causes its deadline to be missed. In addition, we set the priority of each task to its deadline, and a smaller deadline implies a higher priority.

We manually assigned each task in all the six task sets a release time and a deadline for every SPM configuration in such a way that many preemptions are needed in order to find a feasible schedule.

We modified Chronos 4.0 [12] to calculate the WCET of each application with different SPM sizes. The infeasible path detection is enabled in Chronos. The target architecture is an out-of-order, pipelined processor, with an instruction cache and perfect branch prediction. If the instruction cache is hit, an instruction fetch takes 1 cycle. Otherwise, it takes 100 cycles. The target processor uses scratchpad memory to replace data cache. The latencies of scratchpad memory and off-chip memory accesses are 1 cycle and 100 cycles as in [19], respectively. The execution time of each instruction is 1 cycle.

B. Results and Analysis

We evaluated both our approach and the CR approach under three different SPM size configurations: 10%, 20% and 30% of the total data size. We use two performance metrics, namely WCRT and feasible schedule, to compare both approaches. The WCRT of a schedule is the maximum completion time minus the minimum start time of all the tasks. The WCRTs produced by both approaches under various configurations are shown in Figure 2.

In each figure, the black bars are for our approach and the light bars for the CR approach. Each bar represents the relative WCRT increase $WCRT_{inc}$ which is computed as follows:

$$WCRT_{inc} \quad = \quad (WCRT_{base} - WCRT_{alg})/WCRT_{base}$$

where $WCRT_{base}$ is the WCRT of a schedule, computed by using the pEDF, for the same task set without any SPM, and $WCRT_{alg}$ is the WCRT computed by the two approaches.

For the 10% SPM size in Set1, the CR approach cannot find a feasible schedule that meets all the deadlines while our approach does. Therefore, the second bar is empty. For Set3, the WCRTs computed by our approach and the CR approach are close. For this task set, the schedules computed by both have the same number of preemptions. However, our approach achieves a slightly better SPM utilization due to our more efficient SPM allocation algorithm. As a result, our approach performs slightly better. For all the other task sets, our approach performs significantly better. The maximum improvement on WCRT of our approach over the CR approach is 20.31%, which occurs in Set2 under 30% SPM size configuration.

There are two major reasons that our approach performs better. The first reason is that our approach preempts a task only if it is necessary. The second reason is that our SPM allocation algorithm is more efficient as we demonstrated in an example in Section IV.

It is worth noting that SPM is much less effective for an out-of-order processor than for an in-order processor used in [17]. The reason is that an out-of-order processor can hide off-chip memory access latencies by executing other ready instructions.

VI. Related Work

The problems of scheduling tasks with various constraints have been extensively studied [8, 15, 16]. Various scheduling techniques have been proposed. One common assumption made by all the previous scheduling techniques is that the WCET of each task is known. If SPM is used to replace cache, this assumption does not hold any more. As a result, all the previous scheduling techniques without considering SPM are not applicable to the processors with SPM.

A number of research groups have studied the SPM allocation for a single task [1–3, 9–11, 17, 24]. All the techniques proposed assume that the amount of the SPM allocated to each task is known, which is not true for typical embedded systems with concurrent tasks. As a result, those techniques cannot be used to solve the SPM allocation problem.

Recently, several research groups studied the SPM/cache allocation problem for concurrent tasks. [22] exploits both cache partitioning and dynamic cache locking to to provide worst-case performance estimates for multitasking systems. [6] studies the problem of placing multiple tasks in the cache to improve cache performance. It proposes an ILP based approach

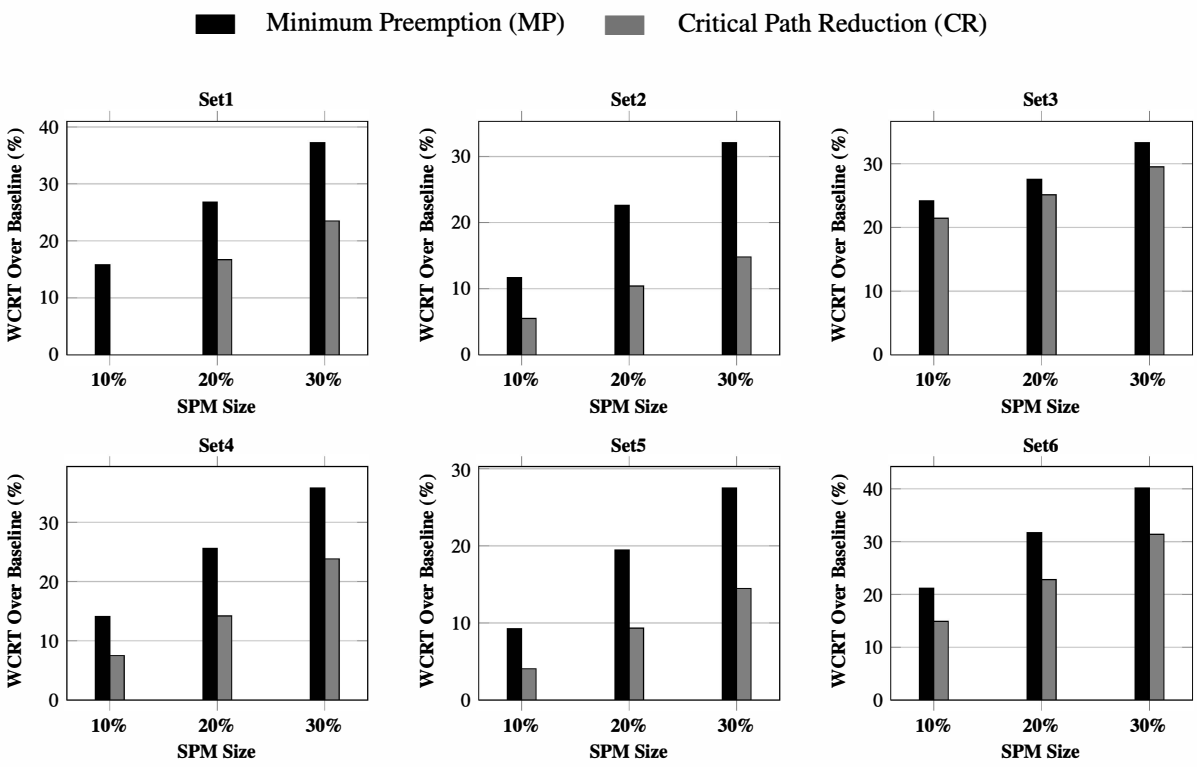


Fig. 2.: WCRT comparison between MP and CR for six task sets and under three SPM configurations

to optimally placing multiple tasks in the cache. The ILP formulations aim to minimize a cost function which is the total conflicts multiplied by a weight assigned to each task. [4] proposes a dynamic scratchpad memory code allocation technique that supports dynamically created processes. Their approach partitions SPM into pages. At runtime, an SPM manager loads code pages of the running applications into the SPM on demand. It supports different sharing strategies that determine how the SPM is distributed among the running processes.

[21] proposes scratchpad memory management techniques for priority-based preemptive multitasking systems. The techniques are applicable to a real-time environment. It proposes three methods: spatial, temporal, and hybrid methods, with an objective to achieve energy reduction in the instruction memory subsystems. It formulates each method as an ILP problem that simultaneously determines (1) partitioning of scratchpad memory space for the tasks, and (2) allocation of program code to scratchpad memory space for each task.

All the above-mentioned approaches do not consider the mutual impacts between task scheduling and SPM allocation. As a result, they cannot achieve the best SPM utilization. [18] and [19] consider the mutual impacts between task scheduling and SPM allocation. [18] proposes an integrated task mapping, scheduling, SPM partitioning, and data allocation technique based on ILP. All the tasks are free of timing constraints and subject to precedence constraints. The ILP formulation explores the optimal performance limit and shows that integrated task scheduling and SPM optimization improves performance by up to 80% for embedded applications.

[19] presents several dynamic scratchpad allocation techniques that take the process interferences into account to improve the performance and predictability of the memory system. It models the application as a MSC (Message Sequence Chart) to capture the interprocess interactions. It proposes an iterative allocation algorithm that consists of two critical steps: (1) analyzing the MSC along with the existing allocation to determine potential interference patterns, and (2) exploiting this interference information to tune the scratchpad reloading points and content so as to best improve the WCRT.

The approach proposed in [19] is the most related to ours. Both their approach and ours take into account the mutual impacts between task scheduling and SPM allocation. Both consider real-time tasks with precedence constraints. However, there are four key differences between our approach and theirs. Firstly, our SPM allocation algorithm is more efficient than their graph coloring based approach. Secondly, by our approach, all the tasks are initially non-preemptible, which means that each task occupies the whole SPM. A task is preempted only if another task with a smaller deadline misses its deadline. By their approach, all the tasks are preemptible at the beginning, and not allocated any SPM space. Detailed analysis in CR (Critical Path Interference Reduction) algorithm is used to reduce the number of preemptions. As a result, our approach leads to fewer preemptions and higher SPM utilization. Thirdly, the task models are different. Under their task model, all tasks are periodic tasks without any additional release times, and all tasks have the same deadline. Our task model assumes that each task has its own release time and deadline. Lastly,

their approach aims to minimize the worst-case response time. In contrast, our approach aims to minimize the number of preemptions while constructing a feasible schedule.

VII. Conclusion

We have proposed a unified approach to the problem of scheduling a set of tasks with individual release times, hard deadlines, and precedence constraints on a single processor where an SPM is used to replace data cache to store stack data of each task, and the problem of allocating SPM to each task. Our approach consists of two algorithms: a task scheduling algorithm and an SPM allocation algorithm. The former aims at minimizing the number of preemptions by using a mix of preemptive and non-preemptive EDF scheduling strategies. The latter employs a novel data structure, namely, the preemption graph, to allocates SPM to each task. Our simulation results show that our unified approach performs better than the approach proposed in [19]. Our future work is to extend our approach to multiprocessor systems.

Acknowledgments

This research is supported by the Australian Research Grants: DP0881330 and DP110104628.

References

[1] Jean-Francois Deverge and Isabelle Puaut. WCET-directed dynamic scratchpad memory allocation of data. In *ECRTS*, pages 179–190, 2007.

[2] Bernhard Egger, Chihun Kim, Choonki Jang, Yoonsung Nam, Jaejin Lee, and Sang Lyul Min. A dynamic code placement technique for scratchpad memory using post-pass optimization. In *CASES*, pages 223–233, 2007.

[3] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Dynamic scratchpad memory management for code in portable systems with an MMU. *In TECS*, 7(2), 2008.

[4] Bernhard Egger, Jaejin Lee, and Heonshik Shin. Scratchpad memory management in a multitasking environment. In *EMSOFT*, pages 265–274, 2008.

[5] Fadia Nemer, Hugues Cass, Pascal Sainrat, Jean-Paul Bahsoun, Marianne De Michiel. Papabench : A free real-time benchmark. In *Workshop on Worst-Case Execution Time Analysis*, 2006.

[6] Gernot Gebhard and Sebastian Altmeyer. Optimal task placement to improve cache performance. In *EMSOFT*, pages 259–268, 2007.

[7] Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. The Mälardalen WCET Benchmarks – Past, Present and Future. In *Workshop on Worst-Case Execution Time Analysis*, pages 137–147, 2010.

[8] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, 1:343–362, 1977.

[9] Lian Li, Hui Feng, and Jingling Xue. Compiler-directed scratchpad memory management via graph coloring. *TACO*, 6(3):9:1–9:17, October 2009.

[10] Lian Li, Quan Hoang Nguyen, and Jingling Xue. Scratchpad allocation for data aggregates in superperfect graphs. In *LCTES*, pages 207–26, 2007.

[11] Lian Li, Jingling Xue, and Jens Knoop. Scratchpad memory allocation for data aggregates via interval coloring in superperfect graphs. *TECS*, 10(2):28:1–28:42, January 2011.

[12] Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3):56–67, 2007.

[13] Jeff Scott, Lea Hwang Lee, John Arends, and Bill Moyer. Designing the low-power M*CORE architecture. In *IEEE Power Driven Microarchitecture Workshop*, pages 145–150, 1998.

[14] SNU. SNU Real-Time Benchmarks. `http://www.cprover.org/goto-cc/examples/snu.html`.

[15] John A. Stankovic. *Deadline scheduling for real-time systems: EDF and related algorithms*. Springer, 1998.

[16] John A. Stankovic. *Scheduling algorithms*. Springer, 2007.

[17] Vivy Suhendra, Tulika Mitra, and Abhik Roychoudhury. WCET centric data allocation to scratchpad memory. In *RTSS*, pages 223–232, 2005.

[18] Vivy Suhendra, Chandrashekar Raghavan, and Tulika Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *CASES*, pages 401–410, 2006.

[19] Vivy Suhendra, Abhik Roychoudhury, and Tulika Mitra. Scratchpad allocation for concurrent embedded software. *In TOPLAS*, 32(4), 2010.

[20] Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada. Allocation of scratchpad memory in priority-based multi-task systems. In *VLSI-DAT*, pages 68–71, 2009.

[21] Hideki Takase, Hiroyuki Tomiyama, and Hiroaki Takada. Partitioning and allocation of scratchpad memory for priority-based preemptive multi-task systems. In *DATE*, pages 1124–1129, 2010.

[22] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *TECS*, 7(1):4:1–4:38, December 2007.

[23] Qing Wan, Hui Wu, and Jingling Xue. WCET-aware data selection and allocation for scratchpad memory. In *LCTES*, pages 41–50, 2012.

[24] Hui Wu, Jingling Xue, and Sri Parameswaran. Optimal WCET-aware code selection for scratchpad memory. In *EMSOFT*, pages 59–68, 2010.