Integrating Task Scheduling and Cache Locking for Multicore Real-Time Embedded Systems

Wenguang Zheng
Tianjin University of Technology, China
wenguangz@tjut.edu.cn

Hui Wu
The University of New South Wales, Australia
huiw@cse.unsw.edu.au

Chuanyao Nie
The University of New South Wales, Australia
c.nie@unsw.edu.au

Abstract

Modern embedded processors provide hardware support for cache locking, a mechanism used to facilitate the WCET (Worst-Case Execution Time) calculation of a task. We investigate the problem of integrating task scheduling and cache locking for a set of preemptible tasks with individual release times and deadlines on a multicore processor with two-level caches. We propose a novel integrated approach that schedules the task set and allocates the locked cache contents of each task to the local caches ($L1$ caches) and the level-two cache ($L2$ cache). Our approach consists of three major components, the task scheduler, the $L1$ cache allocator, and the $L2$ cache allocator. The task scheduler aims at minimizing the number of task preemptions. The $L1$ cache allocator converts the interference graph of all the tasks scheduled on each core into a DAG by considering the preemptions between tasks and allocates the $L1$ cache space to each task. The $L2$ cache allocator converts the interference graph of all the tasks into a DAG by using a k-longest-path-based graph orientation algorithm and allocates the $L2$ cache space to each task. Both cache allocators significantly improve the cache utilization for all the caches due to the efficient use of the interference graphs of tasks. We have implemented our approach and compared it with the extended version of the preemption tree-based approach and the static analysis approach without cache locking by using a set of benchmarks from the Mälardalen WCET benchmark suite and SNU real-time benchmarks. Compared to the extended version of the preemption tree-based approach, the maximum WCRT (Worst Case Response Time) improvement of our approach is 15%. Compared to the static analysis approach, the maximum WCRT improvement of our approach is 37%.

Categories and Subject Descriptors C.3 [*Special-purpose and application-based systems*]: Real-time and embedded systems

Keywords Cache locking, task scheduling, cache allocation, multicore processor

LCTES'17, June 21–22, 2017, Barcelona, Spain
© 2017 ACM. 978-1-4503-5030-3/17/06...$15.00
http://dx.doi.org/10.1145/3078633.3081033

1. Introduction

Caches are widely used in modern processors to bridge the speed gap between processors and off-chip memory. However, caches make it significantly harder to compute the WCET of a task due to the unpredictable memory access latency. Cache locking is employed in modern embedded processors to alleviate the unpredictability problem of caches. Examples of processors with a cache locking mechanism are Power-PC 604e, 405 and 440 families, Intel-960, Motorola MPC7400, and ARM Cortex-A9 Processors.

Multicore processors with two-level caches are increasingly used in modern high performance embedded systems. Examples include MIPS32 74K and IBM Xenon. In a multicore processor with two-level caches, each core has a local I-cache and a local D-cache, and all the cores share an $L2$ cache. A typical embedded system consists of a set of tasks. Tasks may be subject to timing constraints such as release times and deadlines. For hard real-time embedded systems, the designers need to construct a feasible schedule satisfying all the constraints, including timing constraints, at design stage.

However, cache locking makes it more complicated to construct a feasible schedule for a set of tasks on such a processor architecture with cache locking mechanism. On the one hand, in order to construct a feasible schedule, the task scheduler needs to know the WCET of each task. On the other hand, the WCET of each task depends on several factors, including the size of each cache allocated to each relevant task, the set of data and the set of instructions selected as locked cache contents for each task, the locking points of the selected data and instructions of each task, and whether two tasks can share a section of a cache or not. Two tasks can share a section of a cache if they have disjoint lifetimes. As a result, the task scheduling problem and the cache locking problem are intertwined and should be solved in an integrated way.

In this paper, we investigate the problem of integrating task scheduling and cache locking for a set of preemptible tasks with individual release times and deadlines on a multicore processor with two-level caches. For ease of descriptions, for $L1$ caches, we consider $L1$ I-cache allocation only. Our cache allocation algorithms can be easily extended to $L1$ D-cache. Our objective is to construct a feasible schedule, select a set of memory blocks of each task as locked cache contents in the $L1$ cache and the $L2$ cache, and find a layout of all the selected cache contents in each cache. We make the following major contributions.

1. We propose a novel approach to this integrated task scheduling and cache locking problem. Our approach includes a task scheduling algorithm that aims at minimizing the number of the task preemptions in the schedule of the tasks, and novel, efficient cache allocation algorithms for the $L1$ cache and the $L2$ cache. Both cache allocation algorithms consider the interfer-

ences between tasks and convert the task interference graphs to DAGs (Directed Acyclic Graphs) to make efficient use of the $L1$ cache and the $L2$ cache.

2. We have implemented our approach and compared it with the extended version of the preemption tree-based approach and the static analysis approach without cache locking by using a set of benchmarks from the Mälardalen WCET benchmark suite and SNU real-time benchmarks. Compared to the extended version of the preemption tree-based approach, the maximum WCRT improvements of our approach is 15%. Compared to the static analysis approach, the maximum WCRT improvements of our approach is 37%.

The rest of this paper is organized as follows. Section 2 gives a survey of the related work. Section 3 describes the system model and key definitions. Section 4 presents our integrated approach. Section 5 shows the experimental results and analysis, and Section 6 concludes this paper.

2. Related Work

Cache locking problem for a single task has been extensively studied, and various approaches have been proposed. [7, 8, 13, 16, 17, 24, 25]. The most recent approaches are proposed in [24] and [25]. Zheng and Wu [24] propose a min-cut based, dynamic I-cache locking approach for a single task. For a non-nested loop, the approach selects a minimum set of basic blocks as the locked cache contents by using the min-cut algorithm. For a loop nest, the approach finds a good loading point for each selected basic block. They also investigate the dynamic D-cache locking problem for a single task [25], and propose two ILP (Integer Linear Programming)-based approaches to select a near-optimal set of variables as locked cache contents and a k-longest-path-based approach to allocate the selected variables to a D-cache.

A number of approaches have been proposed to integrate task scheduling and cache allocation for single core processors. Puaut and Decotigny [18] study the static I-cache locking problem for multitask real-time systems. They propose two algorithms, one aiming at minimizing the CPU utilization and the other attempting to minimize the interferences between tasks. Campoy et al. [4] propose a genetic algorithm for the problem of selecting instructions to be locked into the I-cache to reduce the response time of multitasks. They also propose a dynamic I-cache locking approach for multitask systems in [5]. The approach uses the response time analysis approach proposed in [3] for the schedulability test, and combines the schedulability analysis with cache locking using a genetic algorithm to improve the performance of the I-cache on a multitasking, preemptive real-time system. Liu et al. [15] propose three I-cache locking approaches: static locking, semi-dynamic locking and dynamic locking, aiming at minimizing the worst-case CPU utilization (WCU) for the embedded systems with multi-tasks.

Several approaches to the integrated task scheduling and cache locking problem have been proposed for multicore processors or multiprocessors. Ding et al. [6] propose an ILP-based task mapping approach to minimize the WCRT of concurrent tasks on a multicore architecture with $L2$ caches. However, they assume that the tasks are executed in a non-preemptive fashion. Therefore, each task can use the entire $L1$ caches. As a result, the $L1$ cache allocation problem does not exist. Liu et al. [14] combine cache locking with task assignment to reduce the WCETs of a set of tasks on a multiprocessor system with two-level caches. They apply cache locking to both I-cache and D-cache by using the algorithms proposed in [13] and [22] to reduce the WCET for each task. Then, they optimize the task assignment considering the locked cache size. They propose a greedy algorithm for $L2$ cache partitioning to assign cache units one by one to the core with the largest completion time. However, the tasks in their task model do not have any timing constraints, which significantly simplifies task scheduling and cache allocation. Furthermore, they do not consider the interferences between tasks on the $L2$ cache. Suhendra and Mitra [20] propose different combinations of the cache locking and partitioning schemes with a focus on $L2$ cache allocation. Particularly, they propose two partitioning approaches for $L2$ cache, namely, task-based partitioning approach and core-based partitioning approach. By task-based partitioning approach, the $L2$ cache is divided into disjoint sections, one section for each task. By core-based partitioning approach, the $L2$ cache is partitioned into disjoint sections, one section for each core. However, this paper does not propose any specific partitioning algorithm for the core-based partitioning approach. There are two major problems with these partitioning approaches. Firstly, they do not take task preemptions into account, resulting in low cache utilization. Secondly, the reloading cost is high for the core-based partitioning approach when a task preemption occurs.

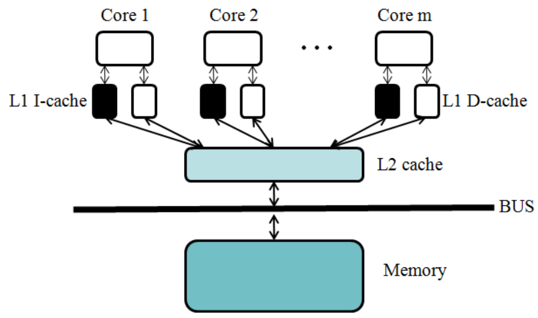


Figure 1. The architecture of a multicore processor

SPM (Scratchpad Memory) management problem is closely related to the cache locking problem. A number of approaches have been proposed to integrate the task scheduling and SPM management. Wan et al. [23] study the task scheduling problem for a single processor with an SPM. They propose two algorithms for task scheduling and SPM allocation, respectively. The first algorithm is to find a feasible schedule for a set of tasks with individual release times, deadlines and precedence constraints, aiming at minimizing the number of preemptions in the schedule. The second algorithm allocate SPM to each task by using the preemption tree. Salamy and Ramanujam [19] investigate the problem of scheduling a set of tasks with precedence constraints on an MPSoC (MultiProcessor System-on-Chip) where all the processors share an SPM. They present a heuristic approach that finds a schedule of the tasks, partitions the SPM memory among the processors, and assigns the variables of each task to the SPM allocated to the processor where the task is scheduled. Suhendra et al. [21] propose an ILP-based approach for the problem of integrating task mapping, scheduling, SPM partitioning, and data allocation. They assume that each core has a private SPM and that a core can access another core's private SPM albeit with an increased latency. The ILP constraints are constructed based on the task graph representing the precedence constraints. The optimization objective of the ILP formulation is to minimize the overall completion time of the task set.

3. System Model and Definitions

The target real-time embedded system uses a homogeneous multicore processor with a set C=$\{c_1, c_2, ..., c_m\}$ of m identical cores. Each core c_i has an $L1$ I-cache and all the cores shared an $L2$ cache which stores both data and instructions. All the caches are set associative caches. The multicore processor uses a bus architecture. We only consider code allocation for both the $L1$ caches and the $L2$

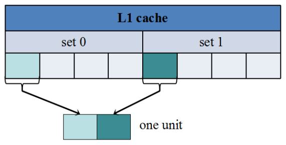


Figure 2. One unit of the $L1$ cache

cache. The data allocation algorithms for the $L1$ caches and the $L2$ cache are similar. Figure 1 illustrates the architecture of the target multicore processor.

We do not model the bus delay caused by multiple simultaneous bus accesses, and also do not consider the task preemption costs. One simplified solution to the bus delay problem is to add the worst-case bus delay to the memory access latency when computing the WCET of each task. The cost of a preemption between two tasks is negligible.

We consider sporadic task model where each task is executed only once, and has a release and a deadline. Our approach can be easily extended to the periodic task model by constructing a set of tasks within the hyperperiod of all the tasks where each task may have multiple instances. Furthermore, each task is preemptible.

We use the dynamic cache locking approach proposed in [24] to select locked cache contents of the code of a task. The locking unit is a cache line. The main memory is partitioned into contiguous memory blocks such that the size of each memory block is mapped to exactly one cache line.

Each task $t_i \in T$ has the following attributes.

1. d_i: the pre-assigned deadline of t_i.
2. r_i: the pre-assigned release time of t_i.
3. W_i: the WCET of t_i without any caches.

Given a schedule of a set T of tasks, the lifetime of a task $t_i \in T$ is an interval $[S_i, F_i)$, where S_i and F_i are the start time and the finish time of t_i in the schedule, respectively.

3.1 Execution Time Functions

The WCET of a task depends on the sizes of the $L1$ cache space and the $L2$ cache space allocated to it for locking the selected cache contents. To help assign a proper size of the $L1$ cache space and a proper size of the $L2$ cache space to each task, we define two discrete functions for each task to record its WCET reductions when different cache sizes are used for its selected locked cache contents.

For the $L1$ cache, we introduce a discrete function $f_{t_i}(x)$ to denote the WCET reduction of a task t_i by using x units of the $L1$ cache to store the locked cache contents of t_i. A unit of the $L1$ cache consists of l_1 cache lines, one cache line in each set, where l_1 is the number of sets of the $L1$ cache. For example, assume that the $L1$ cache is a 4-way set associative cache and the number of sets is 2. Figure 2 shows one unit of the $L1$ cache which consists of two cache lines, one cache line in each set.

We compute $f_{t_i}(x)$ for $x = 1, 2, \cdots, k$ at a fixed interval with a length of 1 unit by using the algorithm proposed in [24], where k is the associativity of the $L1$ cache. For a specific x, $f_{t_i}(x) = W_i - w_i^x$, where W_i is the WCET of t_i without any caches, and w_i^x is the WCET of t_i when using x units of the $L1$ cache to store the locked cache contents of t_i.

For each task t_i, we also introduce a different discrete function $g_{t_i}(x, y)$ to represent the WCET reduction of t_i by using x units of $L1$ cache and a size y of $L2$ cache to store the locked cache contents of t_i. Since the $L2$ cache is used only after a task does not have sufficient cache space in its $L1$ cache for its locked cache contents,

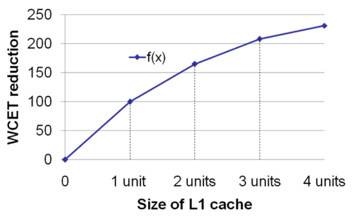


Figure 3. An example of function $f_{t_i}(x)$

x is a fixed value when we compute $g_{t_i}(x, y)$. Notice that, it is not effective to employ the unit-based strategy to compute $g_{t_i}(x, y)$. The reason is that the size of one unit of the $L2$ cache is much larger than that of the $L1$ cache so that some cache contents with much less benefit for reducing the WCET of the task may be locked into the $L2$ cache by using the unit-based strategy. In order to improve the $L2$ cache utilization, we propose a different strategy to compute $g_{t_i}(x, y)$ for each sampling point $y = sp_0, sp_1, \cdots, sp_s$ as follows:

- $sp_0 = 0$. In this case, $g_{t_i}(x, sp_0) = f_{t_i}(x)$.
- $sp_{j+1} = sp_j + s_j$, where s_j represents the size of the $L2$ cache needed to store all the selected memory blocks in one iteration by using the algorithm proposed in [24] for each loop nest of the task t_i. In this case, $g_{t_i}(x, sp_{j+1}) = W_i - w_i^{(x,sp_{j+1})}$, where $w_i^{(x,sp_{j+1})}$ denotes the WCET of the task t_i by allocating x unit of the $L1$ cache and sp_{j+1} size of the $L2$ cache to store the locked cache contents of t_i.

For each task, we define two critical sampling points as follows:

1. sp_m: the smallest sampling point that produces the largest WCET reduction of t_i.
2. sp_b: the sampling point with the largest benefit with respect to the baseline between $(0, f_{t_i}(x))$ and $(sp_m, g_{t_i}(x, sp_m))$, where x is the number of units of the $L1$ cache allocated to t_i, and sp_b is the size of the $L2$ cache allocated to t_i to store its locked cache contents, which is computed as follows:
   (a) Draw a straight line from the point $[0, f_{t_i}(x)]$ to the point $[sp_m, g_{t_i}(x, sp_m)]$ on $g_{t_i}(x, y)$. Assume that $f_{t_i}(x) = \alpha$ and $g_{t_i}(x, sp_m) = \beta$. The straight line can be formulated as follow: $(\beta - \alpha)X - sp_m Y + \alpha * sp_m = 0$.
   (b) For each sampling point smaller than sp_m, calculate the distance from the point $[sp_i, g_{t_i}(x, sp_i)]$ to the straight line as follow:
   $$d_i = \frac{|(\beta - \alpha) * sp_i - sp_m * g_{t_i}(x, sp_i) + \alpha * sp_m|}{\sqrt{(\beta - \alpha)^2 + sp_m^2}}$$
   (c) sp_b is the sampling point with the largest distance.

The sampling point sp_b denotes the size of the $L2$ cache allocated to the task t_i. We do not allocate additional size of the $L2$ cache to the task t_i even though there is free space in the $L2$ cache. The reason is that locking the rest of the cache contents of t_i into the $L2$ cache has much less benefit for reducing the WCET of the task. Thus, we only lock the sp_b size of the $L2$ cache for the task t_i, and the free space of the cache behaves as a cache without cache locking.

Next, we use an example to illustrate how to generate the discrete functions $f_{t_i}(x)$ and $g_{t_i}(x, y)$. Given a task t_i, its WCET and WCET reduction at each sampling point are shown in Table 1, where W_i denotes the WCET of t_i without any caches. Obviously, the WCET reduction of t_i is 0 when no cache is allocated to it. The

Table 1. Sampling results for the task t_i

	$L1$ cache					$L2$ cache					
	W_i	1 unit	2 units	3 units	4 units	sp_1	sp_2	sp_3	sp_4	sp_5	sp_6
t_i's WCET	672	572	507	464	441	421	405	392	386	386	386
t_i's WCET reduction	0	100	165	208	231	251	267	280	286	286	286

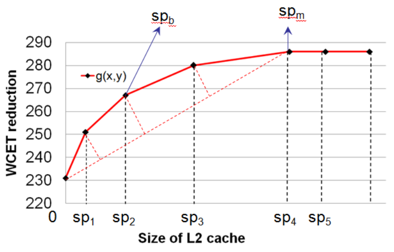


Figure 4. An example of function $g_{t_i}(x, y)$

functions $f_{t_i}(x)$ and $g_{t_i}(x, y)$ of task t_i are shown in Figure 3 and 4, respectively. Clearly, both $f_{t_i}(x)$ and $g_{t_i}(x, y)$ are non-negative and non-increasing with x and y.

3.2 Task Interference Graphs

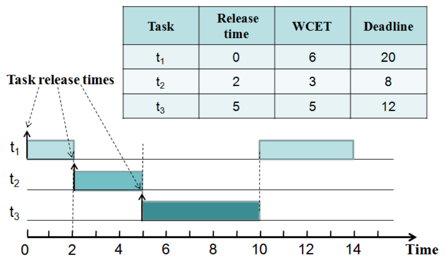
When the locked cache contents of each task are determined by using the cache locking approach proposed in [24], we need to allocate them to the $L1$ cache and the $L2$ cache. For each task, the $L2$ cache is used only if the $L1$ cache does not have enough space to store its locked contents. If two tasks executed on the same core have disjoint lifetimes, their locked cache contents can share a section in both the $L1$ cache and the $L2$ cache. If two tasks executed on different cores have disjoint execution lifetimes, their locked cache contents can share a section in the $L2$ cache. In order to determine if two tasks can share a section of the $L1$ cache and a section of the $L2$ cache, we construct two types of task interference graphs, a global task interference graph for all the tasks and a local task interference graph for each core.

Definition 1. *Given a schedule of a set of n tasks on m cores, the global task interference graph I is an undirected graph where each node t_i denotes a task, and each undirected edge (t_i, t_j) represents that the lifetimes of t_i and t_j overlap.*

The global task interference graph is used to allocate the locked cache contents of all the tasks to the $L2$ cache. Each node of the global task interference graph is labelled with the size of the $L2$ cache allocated to the corresponding task in each cache set. Consider a set of 7 tasks scheduled on a processor with two cores as shown in Figure 6(a), its global task interference graph is shown in Figure 6(b). Assume that the $L2$ cache has only 2 sets. $t_3(0[8], 1[8])$ denotes that the sizes assigned to task $t3$ are 8 bytes in both set 0 and set 1 of the $L2$ cache.

Definition 2. *Given a set of tasks scheduled on a core c_i, the local task interference graph $I(c_i)$ is an undirected graph, where each node t_j denotes a task scheduled on c_i, and each undirected edge (t_j, t_k) represents that the lifetimes of t_j and t_k overlap.*

A local task interference graph for each core is used to handle the $L1$ cache allocation for that core. The weight of each node in a local task interference graph is the number of units of the $L1$ I-cache assigned to the corresponding task. Consider the local task interference graph shown in Figure 6(e). The weight of each node is shown in the bracket. For example, $t_2(1)$ represents that the number of the units of the $L1$ cache allocated to task t_2 is 1.



Task	Release time	WCET	Deadline
t_1	0	6	20
t_2	2	3	8
t_3	5	5	12

Figure 5. An example of task preemption

Notice that, for a particular task t_i, it may have different weights in the global task interference graph and the local interference graph. The reason is that we use different strategies to handle the allocations of the locked cache contents in the $L1$ cache and the $L2$ cache. The weights of the local task interference graph and global task interference graph represent the sizes occupied by the tasks in the $L1$ cache and the $L2$ cache, respectively. Given the global task interference graph I for all the tasks, the local task interference graph $I(c_i)$ can be constructed as follows. Firstly, remove the undirected edge between each pair of tasks t_i and t_j where t_i and t_j are executed on different cores. Secondly, change the weight of each node of the local task interference graph to the number of units of the $L1$ cache allocated to the corresponding task.

3.3 y-Projection Graphs for $L2$ Cache

The allocation of the locked cache contents of each task in the $L2$ cache is performed on individual cache sets. For each set, we need to consider the interferences between tasks that are mapped into this set. Hence, we construct the y-projection graph of a task interference graph for each set y of the $L2$ cache.

Definition 3. *Given the global task interference graph, its y-projection graph is a subgraph of the task interference graph where the locked cache contents of the corresponding task of each node has a portion mapped into the set y of the $L2$ cache, and the node weight of each task is equal to the size of its portion mapped into the set y of the $L2$ cache.*

For each $L1$ cache, we assign the same number of cache lines to each task for all the cache sets. Therefore, no y-projection graph of a local task interference graph is needed.

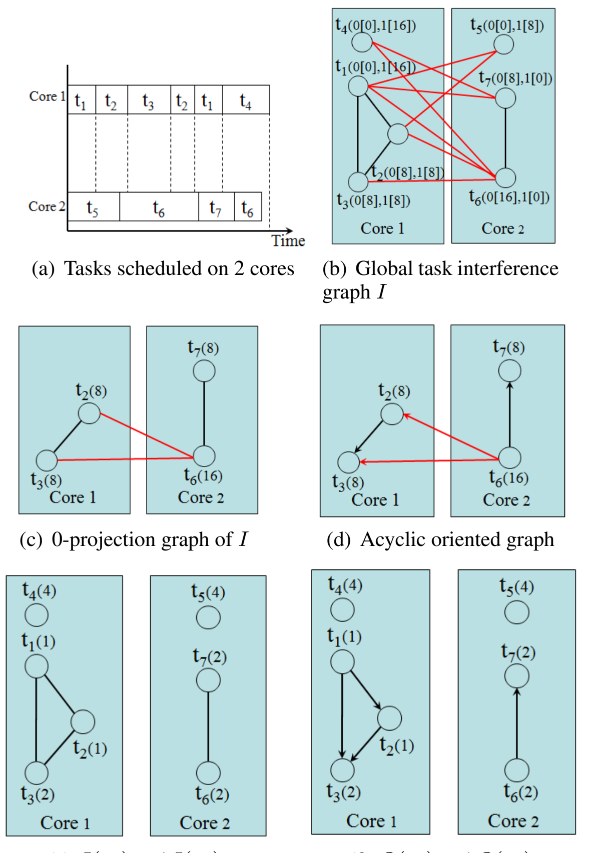
3.4 Acyclic Oriented Graphs

In order to allocate the locked cache contents of each task to a cache, we resort to graph orientation, that is, converting each task interference graph into an acyclic oriented graph, where each edge denotes the precedence between the locked cache contents of the two tasks in the cache.

Definition 4. *Given an undirected graph, its oriented graph is a directed graph obtained by assigning a direction to each edge. An acyclic oriented graph is an oriented graph without any cycle.*

We use different strategies for handling the global task interference graph and the local task interference graphs. For the global task interference graph, we compute its y-projection graph for each set y and convert the y-projection graph into an acyclic oriented graph for the $L2$ cache allocation. For each local task interference graph, we convert it into an acyclic oriented graph for all the sets of the $L1$ cache.

Given a y-projection graph of the global task interference graph, we use the k-longest-path-based approach proposed in [25] to convert it into an acyclic oriented graph, denoted by O_y. Then, we use



(a) Tasks scheduled on 2 cores (b) Global task interference graph I

(c) 0-projection graph of I (d) Acyclic oriented graph

(e) $I(c_1)$ and $I(c_2)$ (f) $O(c_1)$ and $O(c_2)$

Figure 6. An example of task interference graphs, y-projection graphs, and acyclic oriented y-projection graphs

the acyclic oriented graph O_y to determine the layout of the locked cache contents of all the tasks in the set y of the $L2$ cache. Specifically, a directed edge (t_i, t_j) represents that the locked cache contents of task t_j are stored after the locked cache contents of t_i in the cache. The reader may refer to [25] for the details.

Next, we describe how to construct an acyclic oriented graph for each local task interference graph. A task t_i preempts a task t_j in a schedule if one of the following conditions holds:

1. t_i directly preempts t_j.
2. t_i is scheduled after another task that preempts t_j.

Notice that this definition is different from the traditional definition of task preemption. It is used to facilitate the $L1$ and $L2$ cache allocations. In the example shown in Figure 5, t_3 is scheduled after the completion time of t_2, and t_2 preempts t_1. Therefore, t_3 also preempts t_1.

We convert a local task interference graph $I(c_i)$ into an acyclic oriented graph $O(c_i)$ as follows:

- Given a local task interference graph $I(c_i)$, if a task t_i preempts a task t_j in the schedule, we add a direction from t_j to t_i.

We will describe how to allocate the $L1$ cache and the $L2$ cache to each task in Section 4.

Consider a set of 7 tasks scheduled on a processor with two cores as shown in Figure 6(a), its global task interference graph I is shown in Figure 6(b). Figure 6(c) and Figure 6(d) show the 0-projection graph of the global task interference graph, and the acyclic oriented graph of the 0-projection graph of the global interference graph. Figure 6(e) shows the local task interference graphs for core 1 and core 2, respectively. Figure 6(f) shows the acyclic oriented graphs of the local task interference graphs for core 1 and core 2, respectively.

4. Our Approach

Given a set $C = \{c_1, c_2, ..., c_m\}$ of identical cores and a set $T = \{t_1, t_2, ..., t_n\}$ of independent tasks with individual release times and deadlines, our objective is to allocate the $L1$ cache and the $L2$ cache to each task and construct a feasible schedule.

ALGORITHM 1: *IntegratedApproach*

```
input :  A homogeneous multicore processor with a set
         $C = \{c_1, c_2, ..., c_m\}$ of $m$ cores, and a set
         $T = \{t_1, t_2, ..., t_n\}$ of $n$ independent tasks
         with individual release times and deadlines
output:  A feasible schedule of the $n$ tasks
begin
    for each task $t_j \in T$ do
        Allocate the entire $L1$ cache to $t_j$;
        Allocate $L2$ cache to $t_j$ using our $L2$ cache
          allocation algorithm;
        Compute the WCET of $t_i$;
    for each core $c_i$ do
        Create an empty local task interference graph;
    Create an empty global task interference graph;
    Sort all the tasks in $T$ in a non-pre-emptive EDF order;
    for each task $t_j \in T$ in a non-pre-emptive EDF order do
        $TaskScheduling(t_j)$;
```

Any integrated approach to this problem needs to solve the following two problems. The first problem is to assign the n tasks to m cores and construct a feasible schedule for the tasks such that all the timing constraints are satisfied and the maximum completion time of the m cores is minimized. The second problem is to determine the sizes of the $L1$ cache and the $L2$ cache for each task to store its locked cache contents, and find a layout of all the selected cache contents in each cache such that the utilization of each cache is maximized. Unfortunately, these two problems are intertwined with each other. On the one hand, the task scheduler needs to know the WCET of each task. On the other hand, the WCET calculation needs to know the schedule in order to determine the sizes of $L1$ caches and $L2$ cache assigned to each task. Next, we describe how our approach solves these two intertwined problems elegantly in an integrated way.

In order to construct a feasible schedule, we need to minimize the number of task preemptions. The reason is that if a task is not preempted during its execution, it can use the whole $L1$ cache. In order to minimize the number of task preemptions, our approach preempts a task only if it is necessary.

We define a non-pre-emptive EDF (Earliest Deadline First) order for the tasks as the order in which the tasks are scheduled by using the non-pre-emptive EDF strategy.

Initially, our approach allocates the whole $L1$ cache and the $L2$ cache of a proper size to each task, and computes its WCET. Next, it incrementally schedules each task in non-preemptive EDF order such that the maximum completion time of all the tasks scheduled so far is minimized. Algorithm 1 shows the framework of our approach. The task scheduling algorithm and the $L1$ and the $L2$ cache allocation algorithms will be described subsequently.

The algorithm $TaskScheduling(t_j)$ assigns task t_j to a core c_s and incrementally constructs a feasible partial schedule for all

ALGORITHM 2: $TaskScheduling(t_j)$

```
input : A task $t_j$ to be scheduled
output: A new schedule containing $t_j$
begin
    Add $t_j$ to the global interference graph;
    for each core $c_i \in C$ do
        Add $t_j$ to the local interference graph for $c_i$;
        Call our cache allocation algorithms to allocate the
          locked contents of $t_j$ into both $L1$ cache and $L2$
          cache;
        Tentatively schedule $t_j$ on $c_i$ by using the
          non-pre-emptive EDF scheduling;
    Find a subset $C'$ of all the cores on which $t_j$ can meet its
      deadline by using the non-pre-emptive EDF scheduling
      algorithm;
    if $C' \neq \emptyset$ then
        /* No preemption is needed */
        for each core $c_i \in C'$ do
            Compute the completion time $f_i$ of $t_j$ on $c_i$;
        Find the core $c_{min}$ with $f_{min} = \min\{f_i : c_i \in C'\}$;
        for each core $c_i \in C'(c_i \neq c_{min})$ do
            Remove $t_j$ from $c_i$ and restore the previous
              schedule without $t_j$;
            Remove $t_j$ and all its incident edges from the
              local interference graph for $c_i$ and the global
              interference graph;
            Undo the $L1$ allocation on $c_i$ and the $L2$
              allocation for $t_j$;
    else
        /* Preemption is needed */
        for each core $c_i \in C$ do
            Let $T_s$ be the task scheduled to run at the release
              time $r_j$ of $t_j$ on $c_i$;
            if $d_s \leq d_j$ then
                /* No feasible schedule on $c_i$ */
                $f_i = +\infty$;
            else
                /* Compute a new schedule by preempting
                  $t_s$ */
                Let $S_j$ be the set of tasks scheduled at or
                  after the release time $r_j$ of $t_j$ on $c_i$;
                $start\text{-}time = r_j$;
                for each task $t_s \in S_j$ do
                    Remove $t_s$ and all its incident edges
                      from the local interference graph for
                      $c_i$ and the global interference graph;
                    Undo the $L1$ allocation on $c_i$ and the
                      $L2$ allocation for $t_s$;
                $f_j = Re\text{-}Scheduling(S_j, start\text{-}time)$;
        $f_{min} = \min\{f_i : c_i \in C\}$;
        if $f_{min} = +\infty$ then
            /* Cannot find a feasible schedule */
            exit;
        for each core $c_i \in C'(c_i \neq c_{min})$ do
            Remove $t_j$ from $c_i$ and restore the previous
              schedule without $t_j$;
            Remove $t_j$ and all its incident edges from the
              local interference graph for $c_i$ and the global
              interference graph;
            Undo the $L1$ allocation on $c_i$ and the $L2$
              allocation for $t_j$;
```

the tasks assigned to c_s such that the maximum completion time of all the tasks scheduled so far is minimized. Its objective is to minimize the total number of task preemptions while constructing a feasible schedule. It considers the following two cases.

1. There is a core c_r such that t_j can be scheduled on c_r to meet its deadline without preempting any previously scheduled tasks. In this case, since t_j's lifetime does not overlap with any other task's lifetime, t_j can use the whole $L1$ cache. Therefore, previous $L1$ cache allocation result is unchanged. However, we need to do $L2$ cache re-allocation for t_j and other relevant tasks due to the inclusion of t_j in the global task interference graph.
2. t_j cannot be scheduled on any core to meet its deadline without preempting other tasks. In this case, for each core c_i, $TaskScheduling(t_j)$ calls $Re\text{-}Scheduling(S_j, start\text{-}time)$ to construct a new schedule for a set S_j of tasks and re-allocate the $L1$ cache and the $L2$ cache to each task in S_j, where S_j consists of t_j and all the tasks scheduled at or after the release time of t_j on core c_i, and $start\text{-}time$ is the release time of t_j. Then, it selects a core that minimizes the maximum completion time of all the tasks, and undoes the schedules for all the other cores by removing t_j.

The details of $TaskScheduling(t_j)$ are shown in Algorithm 2. The algorithm $Re\text{-}Scheduling(S, start\text{-}time)$ aims to construct a new schedule for all the tasks in S. Recall that $Re\text{-}Scheduling(S, start\text{-}time)$ is called when t_j fails to meet its deadline by using the non-preemptive EDF scheduling. Therefore, $Re\text{-}Scheduling(S, start\text{-}time)$ attempts to preempt the task that runs at the release time of t_j. If the deadline of that task is not larger than that of t_j, our approach cannot construct a feasible schedule. Otherwise, $Re\text{-}Scheduling(S, start\text{-}time)$ preempts that task, and constructs a new schedule recursively for all the tasks in S, aiming at minimizing the number of task preemptions. Whenever $Re\text{-}Scheduling(S, start\text{-}time)$ reschedules a task, it undoes the $L1$ cache and $L2$ cache allocations for the task, and re-allocates the $L1$ cache and $L2$ cache allocations for the task. The details are shown in Algorithm 3.

In the next two subsections, we present our cache allocation algorithms for the $L1$ cache and the $L2$ cache, respectively.

4.1 $L1$ Cache Allocation

According to our system model, each core c_i has a private, k-way set associate $L1$ cache. We divide the $L1$ cache into k units with equal size as described in Section 3.

Assume that a set $T(c_i) = \{t_1, t_2, .., t_g\}$ of g tasks have been scheduled on a core c_i, and a new task t_j is to be scheduled on the core c_i. Our $L1$ cache allocation algorithm adds t_j and all its incident edges to the acyclic oriented local task interference graph $O(c_i)$, discards the previous $L1$ cache allocations of all the tasks in $T(c_i)$, and allocates a proper number of units of the $L1$ cache to t_j and each task in $T(c_i)$. The locked cache contents of each task are allocated to its assigned units of the $L1$ cache.

We define the level of each task in $O(c_i)$ as its in-degree. It can be shown that tasks with the same level have disjoint lifetimes, and thus can share the same section of the $L1$ cache. According to the definition of acyclic oriented local task interference graph, a task with a higher level implies that it directly or indirectly preempts more tasks, and thus should have a higher priority in the $L1$ cache allocation. Consider 5 tasks scheduled on c_i as shown in Figure 7(a). The acyclic oriented local task interference graph is shown in Figure 7(b), where the number in each bracket is the in-degree of the corresponding task. The levels of all the tasks are shown in Figure 7(c).

Our $L1$ cache allocation algorithm consists of four phases. In the first phase, it works from the highest level to the lowest level.

ALGORITHM 3: $Re\text{-}Scheduling(S,\ start\text{-}time)$

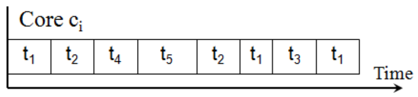
```
input  : A set $S$ of tasks to be re-scheduled on core $c_i$
         $start\text{-}time$: the start time of the first task in $S$ in the
         new schedule for $S$ on $c_i$
output: The maximum completion time of all the tasks in $S$
         in the new schedule
begin
    while $S \neq \emptyset$ do
        Find a task $t_s \in S$ that is ready at $start\text{-}time$ and
          has the smallest deadline;
        Schedule $t_s$ at $start\text{-}time$;
        Add $t_s$ to the local task interference graph and the
          global task interference graph;
        Call our cache allocation algorithms to allocate the
          $L1$ cache and the $L2$ cache to $t_s$;
        Compute the WCET of $t_s$;
        if $t_s$ does not meet its deadline then
            if $r_s = start\text{-}time$ then
                /* Cannot find a feasible schedule */
                return $+\infty$;
            else
                Let $t_r$ be the task scheduled to run at the
                  release time $r_s$ of $t_s$ on $c_i$;
                if $d_r \leq d_s$ then
                    /* Cannot find a feasible schedule on $c_i$
                      */
                    return $+\infty$;
                else
                    /* Compute a new schedule by
                      preempting $t_r$ */
                    Let $S_s$ be a set of tasks scheduled at or
                      after the release time $r_s$ of $t_s$ on $c_i$;
                    $start\text{-}time = r_s$;
                    for each task $t_p \in S_s$ do
                        Remove $t_p$ and all its incident
                          edges from the local interference
                          graph for $c_i$ and the global
                          interference graph;
                        Undo the $L1$ allocation on $c_i$ and
                          the $L2$ allocation for $t_p$;
                    $f_i = Re\text{-}Scheduling(S_s,\ start\text{-}time)$;
                    if $S = \emptyset$ then
                        return $f_i$;
        else
            Remove $t_s$ from $S$;
            if $S = \emptyset$ then
                return the completion time of $t_s$;
            $start\text{-}time = start\text{-}time+$ the WCET of $t_s$;
```

For each level, it allocates a minimum number of units of the $L1$ cache to all the tasks of that level such that all of them meet their deadlines. In the second phase, it repeatly finds a level that results in the maximum reduction of the maximum completion time of all the tasks on the core c_i if all the tasks of that level receive one more unit of the $L1$ cache, and allocates one more unit of the $L1$ cache to all the tasks of that level. In the third phase, for each sink task in $O(c_i)$ whose level is less than the highest level, adds the total number of units of the $L1$ cache allocated to all the higher levels to this sink task. Lastly, it assigns a start address in the $L1$ cache to each task. The details of our $L1$ cache allocation algorithm are shown in Algorithm 4.

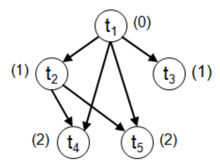
ALGORITHM 4: $L1\text{-}Cache\text{-}Allocation(t_j, c_i, O(c_i))$

```
input  : A new task $t_j$, a core $c_i$ where $t_j$ is to be scheduled,
         and the acyclic oriented local task interference graph
         $O(c_i)$ for all the tasks previously scheduled on $c_i$
output: An $L1$ cache allocation scheme for all the tasks
         scheduled on $c_i$
begin
    Add $t_j$ and all its incident edges to $O(c_i)$;
    /* Phase 1 */
    $L_{max} = 0$; /* $L_{max}$ is the number of units of the $L1$
      cache currently used */
    for each level $l$ in decreasing order do
        Find $w_{min}$, the minimum number of units of the $L1$
          cache such that all the tasks of level $l$ meets their
          deadlines if they receive $w_{min}$ units of the $L1$
          cache;
        if $L_{max} + w_{min} \leq c_{max}$ then
            /* $c_{max}$ is the maximum number of units of the
              $L1$ cache on the core $c_i$ */
            Allocate $w_{min}$ units of the $L1$ cache to each
              task of level $l$;
            $L_{max} = L_{max} + w_{min}$;
        else
            Allocate $c_{max} - L_{max}$ units of the $L1$ cache to
              each task of level $l$;
            $L_{max} = c_{max}$;
            /* No more $L1$ cache space is available */
            break;
    /* Phase 2 */
    while $L_{max} < c_{max}$ do
        Find the level $l$ that results in the maximum
          reduction of the maximum completion time of all
          the tasks on $c_i$ if they receive one more unit of the
          $L1$ cache;
        Allocate one unit of the $L1$ cache to each task of
          level $l$;
        $L_{max} = L_{max} + 1$;
    /* Phase 3 */
    for each sink task $t_s$ whose level is less than the highest
      level do
        Add the total number of units of the $L1$ cache
          allocated to all the higher levels to $t_s$;
    /* Phase 4 */
    for each task $t_s$ in topological order do
        /* $addr^y_{L1}(t_s)$ is the start address of $t_s$ in each set $y$
          of the $L1$ cache */
        if $t_s$ is a source node then
            $addr^y_{L1}(t_s) = 0$;
        else
            $addr^y_{L1}(t_s) = addr^y_{L1}(t_k) + u_k$, where $t_k$ is an
              immediate predecessor of $t_s$ and $u_k$ is the
              number of units of the $L1$ cache allocated to
              $t_k$;
```

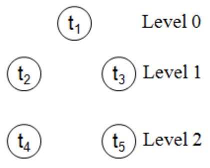
Next, we use an example to illustrate how our task scheduling algorithm and $L1$ cache allocation algorithm work. Assume that the $L1$ cache is a 4-way set associative cache and has 2 sets. Three tasks $\{t_1, t_2, t_3\}$ are to be scheduled on a core c_i, and the WCETs of tasks are shown in Table 2, where the column w_i is the WCET



(a) A schedule for core c_i



(b) The acyclic oriented local interference graph for c_i



(c) Task levels

Figure 7. Task levels in $O(c_i)$

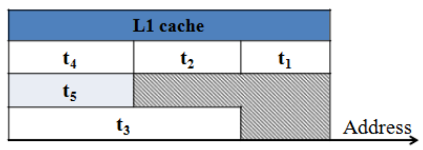


Figure 8. Allocation results of the example in Figure 7 in the $L1$ cache

Table 2. WCETs of the tasks

Tasks	Release Time	Deadline	w_i	WCETs							
				$L1$ cache				$L2$ cache			
				one unit	two units	three units	four units	units used in $L1$ cache (x)	start point ($y = 0$)	largest benefit	smallest WCET
t_1	0	1300	765	664	591	532	489	$x = 1$	664	510	456
								$x = 2$	591	475	441
								$x = 3$	532	470	439
								$x = 4$	489	466	432
t_2	100	900	874	749	642	568	509	$x = 1$	749	500	491
								$x = 2$	642	490	480
								$x = 3$	568	487	471
								$x = 4$	509	478	467
t_3	700	1015	694	573	472	385	323	$x = 1$	573	350	328
								$x = 2$	472	315	310
								$x = 3$	385	312	307
								$x = 4$	323	309	303

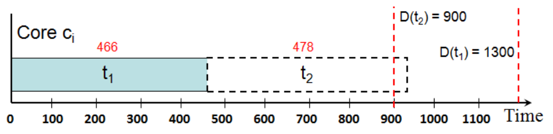
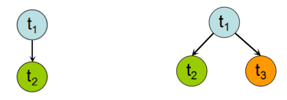


Figure 9. Assigning t_2 to c_i without preemption

of each task without caches, the columns of the $L1$ cache denote the WCETs of each task by using different numbers of units of the $L1$ cache to store its corresponding locked cache contents, and the columns of the $L2$ cache represent the WCET of each task by using different numbers of units of the $L1$ cache and the $L2$ cache.

Initially, t_1 is scheduled on the core c_i and its deadline is met. When t_2 is assigned to the core c_i without preemption, t_2 misses its deadline as shown in Figure 9. Thus, t_2 preempts t_1 at the release time of t_2. The oriented local task interference graph $O(c_i)$ for $\{t_1, t_2\}$, is shown in Figure 10(a). Assign t_2 one unit of $L1$ cache so that it meets its deadline. Therefore, both t_1 and t_2 meet their deadlines now, and there are still 3 free units of the $L1$ cache. Next, assign one more unit of the $L1$ cache to the task which has



(a) (b)

Figure 10. $O(c_i)$ for task sets $\{t_1, t_2\}$ and $\{t_1, t_2, t_3\}$

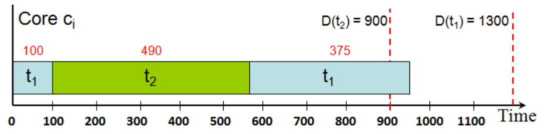


Figure 11. t_2 preempts t_1 at 100

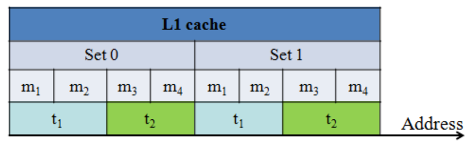


Figure 12. Allocation result of $\{t_1, t_2\}$ in the $L1$ cache

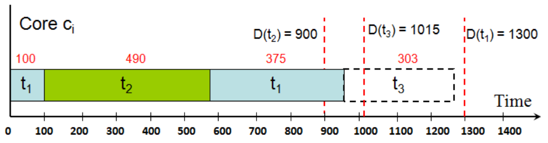


Figure 13. Assigning t_3 to c_i without preemption

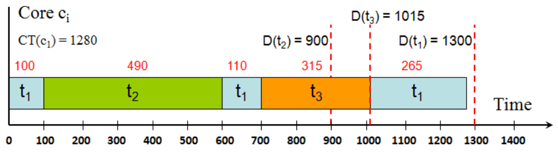


Figure 14. Final schedule of $\{t_1, t_2, t_3\}$

the maximum WCET reduction after adding one more unit of the $L1$ cache to store its locked cache contents. For t_1, if one more unit of the $L1$ cache is assigned to it, its WCET will reduce by $510-475 = 35$. For t_2, if one more unit of the $L1$ cache is assigned to it, its WCET will reduce by $500 - 490 = 10$. Therefore, assign one more unit of the $L1$ cache to t_1. The schedule and the $L1$ cache allocation result of $\{t_1, t_2\}$ are shown in Figure 11 and Figure 12, respectively.

When assigning t_3 to the core c_i without preemption, t_3 misses its deadline as shown in Figure 13. Hence, t_3 preempts t_1 at the release time of t_3, 700. The oriented local task interference graph $O(c_i)$ for $\{t_1, t_2, t_3\}$ is shown in Figure 10(b). Now both t_2 and t_3 are at level 1. After assigning two units of the $L1$ cache to level 1, both t_2 and t_3 meet their deadlines, and the accumulated execution times of t_2 and t_3 are 490 and 315, respectively. Next, assign t_1 two units of the $L1$ cache to guarantee that t_1 will meet its deadline. The final schedule and the $L1$ cache allocation result of $\{t_1, t_2, t_3\}$ are shown in Figure 14 and Figure 15, respectively.

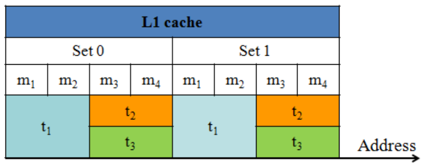


Figure 15. Allocation result of $\{t_1, t_2, t_3\}$ in the $L1$ cache

4.2 $L2$ Cache Allocation

After our $L1$ cache allocation algorithm determines the size of $L1$ cache for a new task t_j, our $L2$ cache allocation algorithm will allocate the $L2$ cache to t_j as follows. Firstly, it constructs the function $g_{t_j}(x, y)$ for t_j, where x is the number of the units of the $L1$ cache allocated to t_j, and y is the proper size of the $L2$ cache allocated to t_j, which is determined by using the approach described in Section 3. Secondly, it allocates the locked cache contents of t_j and the locked cache contents of all the tasks that have been scheduled to the $L2$ cache as follows:

1. Add t_j and its incident edges to the global task interference graph I.
2. Construct the y-projection graph I_y for each set $y(y = 0, 1, ..., m-1)$ of the $L2$ cache, where m is the number of sets in the $L2$ cache.
3. Convert each I_y into an acyclic oriented graph O_y for each set y of the $L2$ cache using the k-longest-path-based approach proposed in [25].
4. Allocate the locked cache contents of the tasks in O_y into the $L2$ cache by using the k-longest-path-based approach proposed in [25].

5. Experimental Results

In order to evaluate the effectiveness of our approach, we select 20 tasks from the Mälardalen WCET benchmark suite [9] and SNU real-time benchmarks [1] to form 7 task sets with different numbers of tasks as shown in the Table 3. We compare our approach with two approaches: STA and PRT.

1. STA: Static analysis approach proposed in [10, 12]. The approach proposed in [12] assumes non-preemptive scheduling. We extend it by combining the preemption delay analysis technique proposed in [10] to calculate the WCET of each task.
2. PRT: An extended approach to the preemption tree approach for a single core processor proposed in [23]. By using the task assignment algorithm and the $L2$ cache allocation algorithm described in Section 4, we modify the approach in[23] to perform task scheduling for a multicore processor.

The performance metric we use is the WCRT (Worst Case Response Time) which is the maximum WCRT of all the cores. The WCRT of a core is the total WCET of all the tasks scheduled on the core.

5.1 Setup

We use the 7 task sets as shown in the Table 3. The number of cores of the target processor is either 2 or 4. We only consider the I-caches. The $L1$ I-caches and the $L2$ I-cache are 4-way set associative caches each with a size of 1KB or 16KB. The hit latency for each $L1$ cache is 1 cycle. The hit latency for the $L2$ cache is 10 cycles, and its miss latency is 50 cycles. We modify the open source WCET analysis tool, Chronos [11], to calculate the WCRTs. SimpleScalar PISA instruction set [2] is used to compile tasks. We perform all the experiments on Intel i7-4770 CPU with 3.40GHz frequency and 16GB memory. For all the task sets, the longest running time of our approach is 362 seconds, which occurs for the task set with 20 tasks, using 4 cores. The results are shown in Figures 16 and 17, where each horizontal axis denotes the 7 task sets, each vertical axis represents the WCRT of each task set, by using the three approaches, and the unit for WCRT is K cycles.

Table 3. Task sets

Tasks	Code size (byte)	Tasks	Code size (bytes)	Tasks	Code size (bytes)	Tasks	Code size (bytes)
cnt	2880	st	3857	fir	8863	select	4494
bsort100	2779	fft1k	6244	ludcmp	5160	qsort-exam	4535
expint	4288	ud	6380	matmult	3737	bs	4248
edn	10563	crc	5168	ndes	7345	insertsort	3892
fdct	8863	qurt	4898	ns	10436	adpcm	26852

Task sets	Number of tasks	Tasks in each set
Set 1	8	qsort-exam, insertsort, ns, bsort100, qurt, adpcm, expint, select
Set 2	10	adpcm, edn, fdct, crc, ludcmp, cnt, ndes, qurt, st, matmult
Set 3	12	bsort100, qsort-exam, insertsort, select, cnt, ud, expint, edn, crc, ludcmp, ns, adpcm
Set 4	14	qsort-exam, insertsort, ns, bsort100, qurt, ud , expint, select, cnt, edn, ludcmp, crc ,fdct, bs
Set 5	16	ludcmp, ud, bsort100, insertsort, adpcm, ns, qsort-exam, select, bs, fdct, qurt, st, matmult, expint, fir, cnt
Set 6	18	adpcm, edn, fdct, crc, ludcmp cnt, ndes, qurt,,st, matmult, qsort-exam, insertsort, select, fir, ns, expint, bs, fft1k
Set 7	20	all the 20 tasks

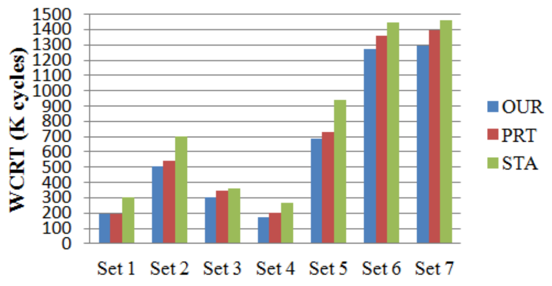


Figure 16. Comparisons of WCRTs for a 2-core processor

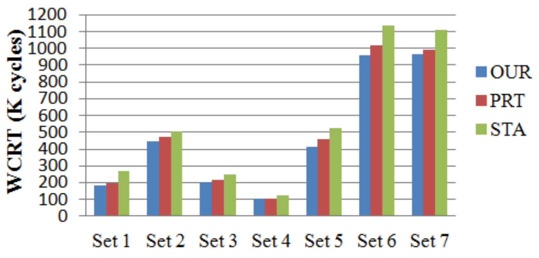


Figure 17. Comparisons of WCRTs for a 4-core processor

5.2 Our Approach vs. STA

Figure 16 and 17 show that the average WCRT improvements of our approach over the static WCRT analysis approach are 24% and 18% for the 2-core processor and the 4-core processor, respectively. On average, our approach achieves dramatic improvements for the task sets containing many tasks with sequentially executed non-nested loop, such as Set 4. With a fixed task set, our approach performs much better for the 2-core processor. The key reason is that more tasks are assigned to each core, resulting in more preemptions for STA in order to meet the deadlines of the tasks. With a fixed number of cores, the WCRT improvements of our approach initially increase as the number of the tasks increases. However, the WCRT improvements decrease when the number of tasks further increases. This is because more tasks compete for the $L2$ cache. For the 2-core processor, Set 4 with 14 tasks has the largest WCRT improvement of 37%. For the 4-core processor, Set 5 has the largest WCRT improvement of 21%.

5.3 Our Approach vs. PRT

Compared with PRT, our approach has nearly no improvement for the task set with a small number of tasks. The reason is that fewer preemptions are needed and more tasks can monopolize the entire $L1$ cache. Therefore, Figure 16 and 17 show small average WCRT improvements, 8% and 6% for the 2-core processor and the 4-core processor, respectively. The maximum WCRT improvement is 15%, which occurs for Set 4 using 2 cores. On average, our approach achieves better performance for a small number of cores. The main reason is that more tasks are assigned to each core, and our approach performs better $L1$ cache allocations. Overall, the major reason that our approach performs better than the preemption tree approach is that our approach simultaneously considers all the tasks currently scheduled to determine the size of the $L1$ cache for each task such that all the tasks can meet their deadlines and the maximum completion time of all the tasks is minimized. In contrast, the preemption tree-based approach PRT determines the size of the $L1$ cache for each task without collectively considering all the related tasks.

6. Conclusion

We investigate the problem of integrating task scheduling and cache allocation for a set of tasks with individual release times and deadlines on a multicore processor with two-level caches, and propose a novel approach to this problem. Our approach aims at minimizing the number of task preemptions and the total utilization of each $L1$ cache and the $L2$ cache by using an efficient preemption-aware task scheduling algorithm and acyclic oriented task interference graphs-based cache allocation algorithms. Experimental results show the effectiveness of our approach. Our future work includes integrating task scheduling and cache locking for a set of tasks with timing constraints and conditional precedence constraints on heterogeneous multiprocessors with two-level caches.

References

[1] Snu real-time benchmarks. `http://archi.snu.ac.kr/realtime/benchmark/`.

[2] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News*, 25(3):13–25, 1997.

[3] J. V. Busquets-Mataix, J. J. Serrano-Martin, R. Ors-Carot, P. Gil, and A. Wellings. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *Proceedings of the Eighth Euromicro Workshop on Real-Time Systems*, pages 271–276. IEEE, 1996.

[4] A. M. Campoy, A. Ivars, and J. Busquets-Mataix. Using genetic algorithms in content selection for locking-caches. In *Proceedings of the international symposium on applied informatics*, pages 271–276, 2001.

[5] A. M. Campoy, A. P. Ivars, and J. Busquets-Mataix. Dynamic use of locking caches in multitask, preemptive real-time systems. In *Proceedings of the 15th Triennial World Congress of the International Federation of Automatic Control*, 2002.

[6] H. Ding, Y. Liang, and T. Mitra. Shared cache aware task mapping for wcrt minimization. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 735–740. IEEE, 2013.

[7] H. Ding, Y. Liang, and T. Mitra. Wcet-centric dynamic instruction cache locking. In *Proceedings of the conference on Design, Automation & Test in Europe*, pages 1–6. IEEE, 2014.

[8] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th international conference on Hardware/software codesign and system synthesis*, pages 143–148. ACM, 2007.

[9] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The mälardalen wcet benchmarks: Past, present and future. In *the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*, pages 136–146, 2010.

[10] J. C. Kleinsorge, H. Falk, and P. Marwedel. A synergetic approach to accurate analysis of cache-related preemption delay. In *Proceedings of the ninth international conference on Embedded software*, pages 329–338. ACM, 2011.

[11] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, pages 56–67, 2007.

[12] Y. Liang, H. Ding, T. Mitra, A. Roychoudhury, Y. Li, and V. Suhendra. Timing analysis of concurrent programs running on shared cache multi-cores. *Real-Time Systems*, 48(6):638–680, 2012.

[13] T. Liu, M. Li, and C. J. Xue. Minimizing wcet for real-time embedded systems via static instruction cache locking. In *Real-Time and Embedded Technology and Applications Symposium, 2009. RTAS 2009. 15th IEEE*, pages 35–44. IEEE, 2009.

[14] T. Liu, Y. Zhao, M. Li, and C. J. Xue. Task assignment with cache partitioning and locking for wcet minimization on mpsoc. In *Parallel Processing (ICPP), 2010 39th International Conference on*, pages 573–582. IEEE, 2010.

[15] T. Liu, M. Li, and C. J. Xue. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Systems*, 48(2):166–197, 2012.

[16] S. Plazar, J. C. Kleinsorge, P. Marwedel, and H. Falk. Wcet-aware static locking of instruction caches. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, pages 44–52. ACM, 2012.

[17] I. Puaut and A. Arnaud. Dynamic instruction cache locking in hard real-time systems. In *Proceedings of the 14th International Conference on Real-Time and Network Systems*, 2006.

[18] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the 23rd IEEE Real-Time Systems Symposium*, pages 114–123. IEEE, 2002.

[19] H. Salamy and J. Ramanujam. A framework for task scheduling and memory partitioning for multi-processor system-on-chip. In *High Performance Embedded Architectures and Compilers*, pages 263–277. Springer, 2009.

[20] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *Proceedings of the 45th annual Design Automation Conference*, pages 300–303. ACM, 2008.

[21] V. Suhendra, C. Raghavan, and T. Mitra. Integrated scratchpad memory optimization and task scheduling for mpsoc architectures. In *Proceedings of the 2006 international conference on Compilers, architecture and synthesis for embedded systems*, pages 401–410. ACM, 2006.

[22] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. In *ACM SIGMETRICS Performance Evaluation Review*, volume 31, pages 272–282. ACM, 2003.

[23] Q. Wan, H. Wu, and J. Xue. Scratchpad memory aware task scheduling with minimum number of preemptions on a single processor. In *Design Automation Conference (ASP-DAC), 2013 18th Asia and South Pacific*, pages 741–748, Jan 2013. .

[24] W. Zheng and H. Wu. Wcet aware dynamic instruction cache locking. In *Proceedings of the conference on Languages, compilers and tools for embedded systems*, pages 53–62. ACM, 2014.

[25] W. Zheng and H. Wu. Wcet-aware dynamic d-cache locking for a single task. In *Proceedings of the 16th Languages, Compilers and Tools for Embedded Systems*, pages 8:1–8:10. ACM, 2015.