WCET Aware Cache Locking and Task Scheduling in Single Core Embedded Systems

Tingxu Zhang\textsuperscript{1, a}, Wenguang Zheng\textsuperscript{1,b,}* and Yingyuan Xiao\textsuperscript{3,c}

\textsuperscript{1, 2, 3}School of Computer Science and Engineering, Tianjin University of Technology, Tianjin, China

\textsuperscript{a}pravez@pravez.com

\textsuperscript{b}*Corresponding author: wenguangz@tjut.edu.cn

\textsuperscript{c}yyxiao@tjut.edu.cn

Abstract—Task scheduling in embedded system needs the WCET (Worst-Case Execution Time) of the task to ensure that tasks can be completed before their deadline. The application of cache reduces the average case execution time of tasks but increases the difficulty of scheduling because WCET becomes difficult to determine. Cache locking is a technique to help determine the WCET of a task. In addition, applying suitable cache locking selection can reduce the WCET of a task, bring easier task scheduling and further reduce the total WCET of all tasks. In this paper, we propose an approach integrated dynamic cache locking and task scheduling in a single core embedded system. We choose the most valuable locking content for each task in limited cache capacity to determine the WCET, schedule the tasks so that all the tasks can be finished before the deadline and make the total WCET minimized. We have implemented this approach and compared it to no-cache task scheduling and non-preemption task scheduling model with LRU cache by using a set of tasks. The experimental results show that our approach achieves successfully task scheduling for all test data. Compared to no-cache task scheduling, our approach achieves the average improvements of 3.22%, 12.68%, 18.96% and 19.12% for the 256B, 512B, 1KB and 2KB caches, respectively. Compared to the non-preemption task scheduling model, our approach achieves the average improvements of 1.32%, 7.27%, 10.2% and 9.13% for the 256B, 512B, 1KB and 2KB caches, respectively.

1. INTRODUCTION

A typical embedded system usually runs several tasks with own release time and deadline. The release time determines the earliest time a task can start, while the deadline determines the latest time a task must complete. Reasonable task scheduling should meet all constraints of tasks and complete these tasks in the shortest time.

In order to schedule the task appropriately, it is necessary to estimate the worst-case executing time (WCET) of the task accurately. As a speed bridge between high speed CPU and relative slower off chip memory, cache has been widely used in embedded system to improve the average case executing time (ACET) of tasks. However, the WCET of tasks is difficult to calculate accurately because of the unpredictability of cache. Cache locking is a very effective technique to improve cache predictability and further, improve the calculating of WCET. When a content is locked into a cache, it will not be evicted until it gets unlocked. By loading and locking contents in the cache, the WCET can be
estimated much more easily. Furthermore, cache locking technique can also reduce the WCET of a task by selecting appropriate contents to be locked into the caches.

Every time a task's release time arrives, the embedded system needs to schedule the task appropriately. If placing task at the end of the schedule queue will make it miss the deadline, it is necessary to advance its start time. The task may have to interrupt the currently executing task to catch deadline. The application of cache locking simplified the calculation of WCET but when the interrupting occurs, a reasonable cache allocation between interrupting tasks is required. Reasonable scheduling and cache allocation can make each task complete on time according to its constraints, and make all tasks finish in the shortest time.

In this paper, we investigate the integration of dynamic cache locking and task scheduling in a single core embedded system. For a group of tasks with release time and deadline, our objective is for each task, determine the start time and the contents to be selected and locked in cache such that the task can be finished on time and the total executing time can be minimized. We make the following major contributions:

- Our approach integrates the state-of-the-art cache locking algorithm and heuristic local optimal task scheduling algorithm, makes the best use of cache and minimizes the total executing time, optimizes the task scheduling.
- We have implemented our cache locking and task scheduling approach and compared it to no-cache task scheduling and non-preemption task scheduling model with LRU cache by using a set of task groups consisted of benchmarks from the MRTC benchmark suite in [10]. Our approach achieves successfully task scheduling for all test data while the other task scheduling models failed in some of them. As for the sets that are successfully scheduled by all models, our approach achieves the average improvements of 8.94%. Compared to no-cache task scheduling, our approach achieves the average improvements of 3.22%, 12.68%, 18.96% and 19.12% for the 256B, 512B, 1KB and 2KB caches, respectively. Compared to the non-preemption task scheduling model, our approach achieves the average improvements of 1.32%, 7.27%, 10.2% and 9.13% for the 256B, 512B, 1KB and 2KB caches, respectively.

The rest of this article is organized as follows. Section II gives a brief survey of related work. Section III describes the system model and major definitions. Section IV describes the details of our approach. Section V shows the experimental results and section VI concludes the paper.

2. RELATED WORK

In recent years, there has been significant amount of work done in cache locking and task scheduling for embedded systems. Anand and Barua propose an approach for instruction-cache locking that is able to reduce the average-case run-time of the program. A cost-benefit model is used to determine contents to lock. For locking content selection, the approach iteratively selects memory block with most benefit when it is locked into cache, until the lock is full or the rest memory blocks' benefit are all negative. Falk, Sascha and Henrik propose an instruction cache locking approach in [2] that aims at minimizing the WCET of the task by iteratively select basic blocks with biggest benefit to be locked into the cache.

In this paper, we investigate the integration of dynamic cache locking and task scheduling in a single core embedded system. For a group of tasks with release time and deadline, our objective is for each task, determine the start time and the contents to be selected and locked in cache such that the task can be finished on time and the total executing time can be minimized. We make the following major contributions:

- Our approach integrates the state-of-the-art cache locking algorithm and heuristic local optimal task scheduling algorithm, makes the best use of cache and minimizes the total executing time, optimizes the task scheduling.
- We have implemented our cache locking and task scheduling approach and compared it to no-cache task scheduling and non-preemption task scheduling model with LRU cache by using a set of task groups consisted of benchmarks from the MRTC benchmark suite in [10]. Our approach achieves successfully task scheduling for all test data while the other task scheduling models failed in some of them. As for the sets that are successfully scheduled by all models, our approach achieves the average improvements of 8.94%. Compared to no-cache task scheduling, our approach achieves the average improvements of 3.22%, 12.68%, 18.96% and 19.12% for the 256B, 512B, 1KB and 2KB caches, respectively. Compared to the non-preemption task scheduling model, our approach achieves the average improvements of 1.32%, 7.27%, 10.2% and 9.13% for the 256B, 512B, 1KB and 2KB caches, respectively.

The rest of this article is organized as follows. Section II gives a brief survey of related work. Section III describes the system model and major definitions. Section IV describes the details of our approach. Section V shows the experimental results and section VI concludes the paper.

2. RELATED WORK

In recent years, there has been significant amount of work done in cache locking and task scheduling for embedded systems. Anand and Barua propose an approach for instruction-cache locking that is able to reduce the average-case run-time of the program. A cost-benefit model is used to determine contents to lock. For locking content selection, the approach iteratively selects memory block with most benefit when it is locked into cache, until the lock is full or the rest memory blocks' benefit are all negative. Falk, Sascha and Henrik propose an instruction cache locking approach in [2] that aims at minimizing the WCET of the task by iteratively select basic blocks with biggest benefit to be locked into the cache. They introduce the execution flow graph (EFG) to model the possible execution paths and selects the basic blocks. Zheng and Wu elaborate a min-cut based dynamic cache locking approach in [3]. They point out that the longest path may change after some contents on the longest path are locked into the cache. Thus, it is not effective to iteratively select contents on the longest path as locked cache contents. Their min-cut based approach select a minimal cut of the control flow graph of a task, such that the longest path and other potential longest paths are reduced simultaneously. Zheng, Wu and Yang propose two algorithms in [4] to find appropriate locking point for selected memory blocks. The ILP algorithm achieves optimal solution and the heuristic algorithm obtains approximate optimal solution faster.

Kafshdooz and Alireza present a dynamic SPM reuse scheme in [5] which allows SPM space be used by other tasks during runtime without requiring any static SPM partitioning. Their approach is unlike previous works that either static schemes or SPM incoherent space allocation and is able to
reduce WCET by up to 55% as compared to recent previous works. Zhuo and Chaitali present dynamic task scheduling algorithms in [6] to minimize the system-level energy by using a combination of optimal speed setting and limited preemption. Their algorithms achieve up to 43% energy saving compared to other approach over the DVS scheduling. Staschulat, Simon and Rolf present a conservative polynomial algorithm to perform task scheduling analysis that considers cache effects due to the preemptions. Significant improvement in analysis are observed in their experiments. Shekhar et al. propose a hard-real-time tasks scheduling strategy in [8]. They allow tasks statically choose and lock a subset of their memory lines in the core's private cache to improve the predictability. In their simulations, their approach achieves an average increase in utilization of 37.31% compared to purely partitioned task allocation. Zheng, Wu and Nie propose an integrating task scheduling approach in [9], brought significantly improve the cache utilization and up to 15% WCRT improvement compared to the extended version of the preemption tree-based approach.

### 3. System Model

![Figure 1. System Model](image)

We investigate the WCET aware cache locking and task scheduling in single core embedded systems. A typical processor models of an embedded system is shown in Fig. 1. The scheduled tasks enter the processing unit in the order of scheduling queue and execute. When a new task enters the processing unit, if the previous task has not completed, the previous task will be put into the interrupted stack. After the current task is completed and no new task enters the processing unit from the queue, the tasks in the stack will continue execution. The target processor is consisted of I-cache (instruction cache) and D-cache (data cache). In this paper, in terms of cache locking, we focus on I-cache locking problem only because the approach used to lock I-cache is also effective for locking D-cache. The I-cache is a fully-associative cache. The cache line size is 32B. Every memory block is mapped into a cache line. The locking unit of our approach is one memory block. Locked cache space of a task can't be unlocked until the task is finished. We define cache allocation unit Unit to facilitate the representation of the allocated cache space of tasks. A Unit of cache allocation for a task represents for locking one memory block of the task into the cache.

Some tasks have a late release time and an earlier deadline. If they are scheduled to be executed at the end of the execution queue, miss deadline may occur. In order to avoid this, it is necessary to interrupt the currently running tasks with relatively late deadline when the release time of the urgent task arrives, and resume the original tasks after the urgent task's execution. Tasks with interruption relationship share the cache due to overlapping runtime. In order to allocate cache reasonably for these tasks during scheduling, we define the task interrupt tree to represent the interruption relationship between tasks as follows.
**Task Interruption Tree:** For a set of tasks with interruption relationship, its task interruption tree is tree \( TIT = <T,R,H> \) where \( T = \{ t_i : t_i \) denotes a task in the set \}, \( R = \{ (t_i, t_j) : t_i \) is interrupted by \( t_j \} \), \( H = \{ h_i : h_i \) denotes the depth of \( t_i \) in the tree \}.

Since the operation of the lock instruction also takes time, it is not worth locking contents that is not in the loop. Therefore, the content of each task that is worth locking is bound to be limited. With the increase of cache capacity, the increase of the reduction of WCET increases first and then decreases. For a set of tasks sharing the same cache space, achieving the best cache utilization as far as possible for each task can minimize the overall execution time of these tasks. We define \( DTW \) to represent the reduction of WCET of the task as the cache space for locking increases. Through the comparison of \( DTW \) of different tasks, the limited cache space can be allocated to tasks that can bring more total WCET reductions.

**4. Our Approach**

In this section, we describe task scheduling approach. Given a set of tasks with release time and deadline, the objective of our approach is to reasonably arrange the start time of each task and the cache capacity they own, lock appropriate contents into cache so that each task can be completed on time according to its limited conditions, and all tasks can be finished in the shortest time.

Firstly, for each task in the task set, we use the min-cut based cache locking approach proposed in [3] to calculate the content to be locked and the corresponding WCET applying the cache locking approach under certain cache capacities, and the \( DTW \) with the increase of cache capacity. Then, according to the release time of each task, the tasks are scheduled in turn, and the cache locking condition of each task is also determined. When the scheduling is completed, the tasks with cache locking decisions and schedules are burned into the embedded system.

**4.1. Task Scheduling Framework**

In a single core embedded system, as shown in Fig. 2, if deadline and release time are not considered, WCET is the shortest and sum is the smallest when each task monopolizes all caches. Therefore, there must be no interruption and the task can be run in turn. Every task has the smallest WCET when owning the whole cache, and the WCET sum of all tasks are smallest too. Therefore, no interruptions are needed and tasks can be executed successively.

![Figure 2. The impact of interruption on task running time](image)

However, some tasks have a late release time and an earlier deadline. If they are scheduled to be executed at the end of the execution queue, miss deadline may occur. So, the task needed to be brought forward to ensure that the tasks can be completed before the deadline. The operation of brought forward may cause life cycle overlaps between tasks and the task need to share the cache space with other tasks.

We use a heuristic dynamic programming algorithm to schedule tasks, making each step locally optimal, so that the overall task scheduling tends to global optimization. The tasks are scheduled in the order of its actual release time.
The framework algorithm for task scheduling considering release time and deadline is given below.

**Algorithm 1: Task Scheduling (Ts, DTWs)**

- **Input:**  
  - Ts: a set of tasks to be scheduled  
  - DTWs: the DTWs of Ts
- **Output:** The scheduled tasks’ queue Q with actual start time, and the cache size allocated for each task

**Begin:**

- Q = new task scheduling queue;
- sort Ts from small to large according to tasks' release time;
- for each task $T_i$ in Ts do
  - put $T_i$ at the end of the Q;
  - forward_moving($T_i$, Q);
  - if $T_i$'s deadline can be caught then
    - continue;
  - else
    - preemption_tasks_scheduling ($T_i$, Q);
- return Q;

4.2. Task Forward Moving

Putting the task with earlier deadline at the front position of the queue can avoid missing deadline caused by task orders. For the task $T_i$ that just entered the queue, we perform the following operation repeatedly till the actual start time of $T_i$ is earlier than its release time or the deadline of the task swapped with $T_i$ can't be caught.

- let $T_j$ be the task before $T_i$ in the queue;
- swap $T_i$ and $T_j$;

4.3. Preemption Tasks Scheduling

After moving forward task $T_i$, if the deadline still can't be caught, the task running at $T_i$'s release time needs to be interrupted.

To schedule tasks with interruptions, we generate a TIT using $T_i$ and the tasks it is going to be interrupted. It can be seen from the definition of TIT that the executing time of the inner level task will affect the completion time of the outer level task, so the deadline of the lower level task should be caught first. The child task nodes belonging to the same parent node can share the cache space due to the no overlapping of the life cycles. From the innermost node to the outermost node of TIT, we allocate each task the least amount of cache which can catch the deadline. On the premise that all tasks can be completed on time, the total running time can be minimized by allocating the remaining cache to the tasks in the level with the largest DTW sum in the TIT.
As shown in Fig. 3, there are three tasks A, B and C. B interrupted A and C interrupted B. Assume that the cache capacity is 4 Unit, every task can catch the deadline with 1 Unit of cache allocate. By allocating the remain 1 Unit of the cache to task B, which has the largest decline of WCET, smallest overall WCET can be achieved.

5. EXPERIMENTS
We have implemented our approach and compared them to no-cache task scheduling and non-preemption task scheduling model with LRU (NP-LRU) cache.

5.1. Setup

| Task Set | Number of Tasks | Number of code size | Task in each set |
|----------|-----------------|---------------------|------------------|
| Set 1    | 6               | 1083                | bsort100 cnt cover crc expint matmult |
| Set 2    | 6               | 7013                | adpcm jfdctint ldnum ns nischneu sqrt statemate |
| Set 3    | 15              | 8676                | adpcm bs bsort100 cnt crc duff fdct fir insertsort matmult ns nischneu sqrt statemate ud |
| Set 4    | 15              | 8676                | adpcm bs bsort100 cnt crc duff fdct fir insertsort matmult ns nischneu sqrt statemate ud |
| Set 5    | 21              | 9750                | adpcm bs bsort100 cnt cover crc duff expint fdct fibcall fir insertsort jfdctint ldnum matmult ns nischneu sqrt sqrt statemate ud |

The benchmark sets, as shown in Table. 1, consist of several tasks from the MRTC benchmark suite with manually defined release time and deadline. Because there are too few memory blocks in loops in the benchmarks, we add an outer loop to all benchmarks, which contains the whole benchmark, and set the number of iterations of the outer loop to 10.
We use Chronos [11] to estimate the WCET of a task. We use four different types of cache sizes: 256B, 512B, 1KB, and 2KB. The cache line size is fixed to 32B.

5.2. DTW Calculation

![Figure 4. Change of WCET in different cache size (b)](image)

The change of WCET with the increase of cache size is shown in Fig. 4. As can be seen from the figure that when the allocated cache capacity reaches 32 Unit for most tasks, the reduction of WCET reaches the maximum, continue to increase the cache size barely bring more WCET reduction. For task nsichneu and stateamate, the allocated cache capacity is basically proportional to the reduction of WCET because there are abundant memory blocks in loops.

5.3. Task Scheduling Performance

![Figure 5. Comparison of total WCET with 256B cache size](image)

![Figure 6. Comparison of total WCET with 512B cache size](image)
All experimental results are shown from Fig. 5 through Fig. 8, where each horizontal axis denotes benchmarks and each vertical axis shows the total WCET of tasks scheduled by different model. Gray bars with X marked above stands for schedule failure.

Compared to no-cache task scheduling, our approach achieves the average improvements of 3.22%, 12.68%, 18.96% and 19.12% for the 256B, 512B, 1KB and 2KB caches, respectively. Compared to the non-preemption task scheduling model, our approach achieves the average improvements of 1.32%, 7.27%, 10.2% and 9.13% for the 256B, 512B, 1KB and 2KB caches, respectively.

Because the application of cache locking brings smaller WCET, our approach successfully schedules all task sets. For those task sets that all models have successfully scheduled, when the cache size is small, the improvement of our approach is limited compared with other models due to the limited number of locking contents. With the increase of cache size, the WCET improvement brought by cache locking and dynamic scheduling is considerable. As the cache size continues to increase, the improvement of our approach compared with NP-LRU is somewhat reduced because for NP-LRU, the larger cache makes the cache hold more memory blocks, thus reducing the WCET. Still, our approach has better performance because it always chooses the best locking content and makes approximate optimal task scheduling.

6. CONCLUSION
In this paper, we investigate the integration of dynamic cache locking and task scheduling in a single core embedded system. For a group of tasks, our approach determines the most appropriate task running order and for each task, our approach determines the optimal cache capacity usage and most valuable contents to be locked into cache. Each task can be completed before its deadline and the total execution time is minimized. The experimental result shows that our approach successfully scheduled all tasks in benchmarks, and compared to non-preemption task scheduling model with LRU cache, our
approach achieves the average improvements of 2.27%, 9.67%, 14.09% and 14.56% for the 256B, 512B, 1KB and 2KB caches, respectively.

For multi core embedded system which is much more complicated, there are not only interruptions but migrations between tasks over cores. In addition, shared L2 cache also has influence on tasks' WCET and then affect the task scheduling. Our future work is to extend the cache locking technique to multi-core multi-level cache embedded system and put forward approaches to solve the calculation of WCET and make task scheduling.

ACKNOWLEDGMENT
This work was supported by National Nature Science Foundation of China (61702368), Natural Science Foundation of Tianjin, China (18JCQNJC0070).

REFERENCES
[1] Anand, Kapil, and Rajeev Barua. "Instruction cache locking inside a binary rewriter." Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems. 2009.
[2] Falk, Heiko, Sascha Plazar, and Henrik Theiling. "Compile-time decided instruction cache locking using worst-case execution paths." Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis. 2007.
[3] Zheng, Wenguang, and Hui Wu. "WCET: aware dynamic instruction cache locking." Proceedings of the 2014 SIGPLAN/SIGBED conference on Languages, compilers and tools for embedded systems. 2014.
[4] Zheng, Wenguang, Hui Wu, and Qing Yang. "WCET-aware dynamic I-cache locking for a single task." ACM Transactions on Architecture and Code Optimization (TACO) 14.1 (2017): 1-26.
[5] Kafshdooz, Morteza Mohajjel, and Alireza Ejlali. "Dynamic shared SPM reuse for real-time multicore embedded systems." ACM Transactions on Architecture and Code Optimization (TACO) 12.2 (2015): 1-25.
[6] Zhuo, Jianli, and Chaitali Chakrabarti. "System-level energy-efficient dynamic task scheduling." Proceedings of the 42nd annual Design Automation Conference. 2005.
[7] Staschulat, Jan, Simon Schliecker, and Rolf Ernst. "Scheduling analysis of real-time systems with precise modeling of cache related preemption delay." 17th Euromicro Conference on Real-Time Systems (ECRTS'05). IEEE, 2005.
[8] Shekhar, Mayank, et al. "Semi-partitioned hard-real-time scheduling under locked cache migration in multicore systems." 2012 24th Euromicro Conference on Real-Time Systems. IEEE, 2012.
[9] Zheng, Wenguang, Hui Wu, and Chuanyao Nie. "Integrating task scheduling and cache locking for multicore real-time embedded systems." ACM SIGPLAN Notices 52.5 (2017): 71-80.
[10] M. research group, WCET benchmark programs, http://www.mrtc.mdh.se/projects/wcet/benchmarks.html.
[11] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. Science of Computer Programming, 69(1-3), 2007. http://www.comp.nus.edu.sg/~rpembed/chronos