Integrated Instruction Cache Analysis and Locking in Multitasking Real-time Systems

Huping Ding[#], Yun Liang[*], and Tulika Mitra[#]
[*]Center for Energy-efficient Computing and Applications, School of EECS, Peking University
[#]School of Computing, National University of Singapore
{d-huping,tulika}@comp.nus.edu.sg, ericlyun@pku.edu.cn (contact author)

ABSTRACT

Cache locking improves timing predictability at the cost of performance. We explore a novel approach that opportunistically employs both cache analysis and locking to enhance schedulability in preemptive multi-tasking real-time systems. The cache is spatially shared among the tasks by statically locking a portion of the cache per task. To overcome the issue of limited cache space per task, we keep a portion of the cache unlocked and let all the tasks use it through time-multiplexing. Compared to locking the entire cache for each task during execution, our approach obviates the cost of reloading locked blocks at preemption. But we require static cache analysis for WCET estimation and cache related preemption delay (CRPD) analysis of the unlocked cache space. We design an algorithm to make appropriate locking decisions through accurate cost-benefit analysis. Experimental results show that our integrated approach leads to substantially improved schedulability results compared to cache analysis and cache locking employed individually.

Categories and Subject Descriptors

C.3 [**Special-purpose and Application-based Systems**]: [Real-time and embedded systems]

General Terms

Algorithm, Design, Performance.

Keywords

Real-Time, Multi-tasking, WCET, CRPD, Cache Locking.

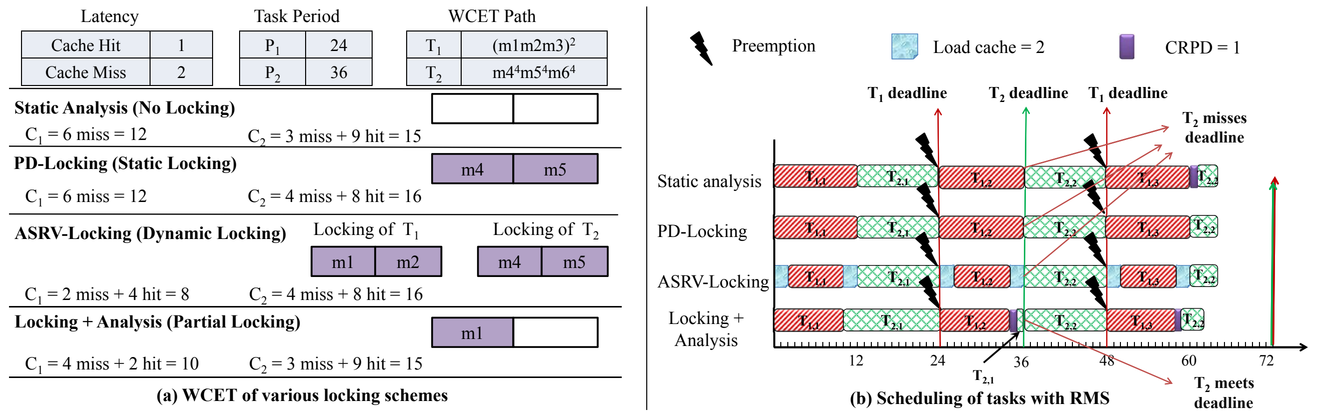
1. INTRODUCTION

Multi-tasking real-time systems demand predictable timing behavior from the underlying architectural mechanisms employed to execute the tasks. Modern micro-architectures, however, perform aggressive dynamic optimizations to improve performance at the cost of timing predictability. The instruction cache is one such architectural feature that is ubiquitous in real-time embedded systems. But the instruction cache introduces two challenging problems in multi-tasking real-time systems. First, in the presence of an instruction cache, it is difficult to estimate the *Worst-case Execution Time (WCET)* [28] of a task through program analysis as it is not known statically whether a memory access will hit or miss in the cache. Second, in a preemptive multi-tasking system, when a task T gets preempted by another task T', the memory blocks of T are replaced by those of T' in the cache. Once T resumes execution, it needs to bring in the replaced memory blocks again into the cache. This cost is known as *Cache Related Preemption Delay (CRPD)* [13] and adds a variable delay to the fixed context switching cost. The delay depends on the number of replaced memory blocks that are used again in task T. The schedulability analysis of a task set requires both the WCET and the CRPD values.

At the other end of the spectrum, we have cache locking [8, 9, 19] that offers completely predictable cache behavior and avoids the complexity of cache modeling altogether. The cache is locked with selected memory blocks from a task before execution. The memory blocks locked in the cache are guaranteed to be cache hits, while the remaining blocks are guaranteed to be cache misses, obviating the need for cache modeling in WCET analysis.

In multi-tasking systems, there exist two different locking approaches. Puaut and Decotigny [23] propose space sharing of the entire cache by locking a portion of the cache per task. We call it *PD-Locking* approach following the last names of the authors. The advantage of this approach is that the cache content remains unchanged throughout the execution of the tasks. The downside is that each task has access to only a fraction of the cache. Aparicio et al. [4] observe this limitation and introduces a time-multiplexed sharing of the cache through locking, called *ASRV-Locking* approach in the rest of the paper. In this approach, a task has exclusive access to the entire cache during execution and locks the cache with its own memory blocks leading to improved WCET per task compared to *PD-Locking* approach. However, when a task resumes execution after preemption, it has to reload the locked cache blocks leading to significantly higher (but fixed) preemption cost. Note that both approaches can bypass CRPD analysis as *PD-Locking* does not require cache reloading at preemption, while *ASRV-Locking* has fixed cache re-loading/locking cost at preemption.

We propose a non-traditional approach that judiciously combines cache locking with cache modeling in preemptive multi-tasking real-time systems and overcomes the space limitation of *PD-Locking* and cache reloading cost at preemption for *ASRV-Locking*. Similar to *PD-Locking*, we adopt space sharing by statically locking a portion of the cache per task but with a crucial difference. We leave a portion of the cache unlocked and let the tasks take advantage of this unlocked portion during execution through normal cache replacement policy. That is, the locked portion of the cache is statically shared, while the unlocked portion is time-multiplexed among the tasks. This relaxes the space constraint for each task and elimi-



(a) WCET of various locking schemes

(b) Scheduling of tasks with RMS

Figure 1: Motivating example.

nates cache re-loading at preemption. However, WCET and CRPD analysis are required for the unlocked portion of the cache.

Why do we want to give up the predictability offered through full cache locking, but not embrace static cache analysis all the way? We believe cache locking comes to rescue when cache analysis is unable to conclusively classify a memory access as hit or miss. At the same time, advancement in both WCET and CRPD analysis ensures that we can analyze the cache behavior quite precisely for most of the memory blocks. These memory blocks with predictable access patterns can reside in the unlocked portion of the cache providing improved performance. Indeed recent work [8] shows that, for a single task, a partially locked cache provides optimal WCET compared to both static analysis and full cache locking.

While Ding et al. [8] develops partial cache locking solution for a single task, it is challenging to make a locking decision for multi-tasking systems. First, the change in execution time of a task impacts the schedulability of the other tasks. Thus, we need to adopt a global optimization approach rather than a local per task approach. Second, the unlocked portion of the cache requires CRPD analysis. We propose an algorithm that employs accurate cost-benefit analysis to capture the impact of locking a memory block on the WCET, CRPD, and schedulability of all the tasks and thereby makes an informed decision to choose the appropriate memory blocks for locking. We perform detailed experimental evaluation to validate the improved schedulability results of our approach.

Motivating Example. We illustrate the benefit of our integrated cache analysis and locking using the example in Figure 1.

WCET comparison of various locking schemes. Figure 1(a) compares the WCET under various locking schemes. We assume two tasks T_1 and T_2 with periods 24 and 36. Cache hit latency is 1 cycle and miss latency is 2 cycles. The WCET paths of T_1 and T_2 are $(m_1m_2m_3)^2$ and $(m_4^4m_5^4,m_6^4)$ where m_i represent a memory block. We assume all the memory blocks are mapped to the same cache set in a 2-way set associative cache. To simplify discussion, we also assume that timing is solely determined by instruction cache effects and the WCET paths do not change after locking.

Static Analysis estimates the WCET via cache modeling with no locking [24]. For T_1, all the accesses are miss because 3 memory blocks compete for 2 cache blocks resulting in 6 cache miss and WCET $C_1 = 12$. For T_2, it has 3 cold miss while remaining accesses are hit resulting in $C_2 = 15$. *PD-Locking* statically locks the entire cache with memory blocks from T_2 by selecting memory blocks with highest $access frequency/period$ [23]. Thus, accesses of T_1 miss, while for T_2 it depends on which blocks are locked. *ASRV-Locking* employs dynamic locking where each task has exclusive access to the entire cache. WCET of T_1 is reduced; but not the WCET of T_2. Finally, our *Locking + Analysis* judiciously chooses to lock only one block of T_1 and leaves the other cache block unlocked so that T_2's accesses are still hits after the cold misses.

Scheduling results of RMS. Let us assume that the reloading overhead for *ASRV-Locking* is 2 and the CPRD for our *Locking + Analysis* is 1. The execution of the tasks over the hyper-period is shown in Figure 1(b). The tasks are scheduled with Rate Monotonic Scheduling (RMS) policy where the task with the shortest period (T_1) has the highest priority. $T_{X,Y}$ represents the Yth instance of task T_X. For *Static Analysis* and *PD-Locking*, they fail to meet the deadline due to high WCET. The CRPD for *Static Analysis* is not shown as the task already misses the deadline. For *ASRV-Locking*, each task locks the entire cache and thus have lower WCET. However, every time a new task instance starts execution or a preempted task resumes execution, we need to reload and lock the cache with 2 cycle penalty. The additional cache reloading overhead makes T_2 miss its first deadline. Due to time-multiplexing of the unlocked cache space among all the tasks and lower preemption cost, our *Locking + Analysis* solution has lower WCET compared to *Static Analysis* and *ASRV-Locking*. This enables both T_1 and T_2 to meet their deadlines. A comparison with real task sets will be presented later in the experimental evaluation section.

2. RELATED WORK

Caches are problematic for WCET estimation due to their timing unpredictability. Static analysis has been widely used to bound the WCET [15, 24] of a single task. In multi-tasking preemptive real-time systems, a number of techniques have been proposed to accurately model the CRPD [13, 25, 21, 12].

Modern architectures often feature cache locking for better timing predictability. By carefully selecting the memory blocks for locking, WCET can be improved. There exist two locking mechanisms, static cache locking [9, 19, 22, 16] and dynamic cache locking [5, 26]. However, all of the above techniques lock the entire cache. Recently, Ding et al. [8] demonstrated that by partially locking the cache, WCET can be improved significantly.

In the context of multi-tasking real-time systems, Puaut and Decotigny [23] propose two static low complexity locking algorithms. Aparicio et al. [4] propose a dynamic locking solution based on Integer Linear Programming. Campoy et al. [7] develop static locking solutions using generic algorithms. Liu et al. [20] also propose dynamic locking solution within a task. We do not consider dynamic locking within a task as it requires code modification by inserting cache reloading instructions as shown in [20]. Again, these techniques lock the entire cache. Verma et al. [27] propose a hybrid approach for scratchpad memory allocation in multiprocess systems. Each process is allocated a disjoint region while the rest portion is shared by all processes. Their approach aims to minimize the energy consumption. Scratchpad memory does not have any address mapping constraint as in cache. Moreover, there is no timing unpredictability issue in scratchpad memory, making the problem somewhat simpler. Meanwhile, real-time scheduling is not considered in their work.

3. SYSTEM MODEL

In this section, we present the basic models of caches and tasks.

Cache Model. Cache design involves a few parameters: line (block) size L, which defines the unit of data or instruction transfer between the cache and main memory; number of cache sets K that the cache is divided into; associativity A, which determines the number of cache lines in a set. Then, the capacity of the cache is $L \times K \times A$. We assume LRU (Least Recently Used) cache replacement policy.

Given a memory block m, it can be mapped to only one cache set ($m \ \ modulo \ \ K$). To simplify the following discussion, we assume there is only one cache set as the different cache sets do not interfere with each other. However, our locking algorithm works with multiple cache sets. We use M to represent the set of memory blocks mapped to a cache set. We also use $\perp$ to indicate the absence of any memory block in a cache line.

DEFINITION 1 (**Concrete Cache State**). *A concrete cache state c is a vector $\langle c[0], ..., c[A-1]\rangle$ of length A where $c[j] \in M \cup \{\perp\}$. If $c[j] = m$, then m is the j^{th} most recently used memory block in the cache set. We also define a special concrete cache state $c_\perp = \langle \perp, ..., \perp \rangle$ called the empty concrete cache state.*

DEFINITION 2 (**Abstract Cache State**). *An abstract cache state a is a vector $\langle a[0], ..., a[A-1]\rangle$ of length A where $a[j] \in 2^M$.*

An abstract cache state maps a cache block to a set of memory blocks. At a program point, the abstract cache state is a safe approximation of the concrete cache states along all the incoming program paths, and hence is a more compact representation. Both concrete and abstract cache states have been widely used in real-time systems for timing analysis [17, 24].

Task Model. We assume a preemptive multi-tasking real-time system running on uni-processor with a set of N independent periodic tasks $\mathscr{T} = \{T_1, ..., T_N\}$. For each task T_i, we use P_i to represent its period and C_i to represent its WCET. We assume the deadline $D_i = P_i$. The C_i value is obtained by performing intra-task WCET analysis for T_i. In other words, the WCET analysis is performed in isolation per task. In processors with caches, we also need to account for the delay due to preemption: the CRPD and the context switching cost. For each task T_i, we use Δ_i to denote the delay due to preemption. Let U be the total processor utilization for the task set. A necessary condition for feasible scheduling of the task set is

$$U = \sum_{i=1}^{N} \frac{C_i + \Delta_i}{P_i} \leq 1 \qquad (1)$$

The delay due to preemption for task T_i is defined as follows.

$$\Delta_i = \sum_{T_j \in pt(T_i)} (CRPD(T_i, T_j) + CSC) \times n(T_i, T_j) \qquad (2)$$

where $pt(T_i)$ is the set of tasks that may preempt T_i, $CRPD(T_i, T_j)$ is the CRPD of T_i imposed by T_j in one preemption, $n(T_i, T_j)$ is the bound for the number of preemption of T_i imposed by T_j, and CSC represents the context switching cost.

EDF Scheduling. Earliest Deadline First (EDF) is a dynamic priority based scheduling policy. The priority of a task is determined by its deadline. At any time instance, EDF chooses the ready task with the closest deadline for execution. For EDF, Equation 1 ($U \leq 1$) is both sufficient and necessary condition for feasible schedule. The task set that may preempt T consists of all the tasks that may have earlier deadline than T [11].

RMS Scheduling. Rate Monotonic Scheduler (RMS) is a static priority based scheduling policy. The priority of a task is determined statically by its period. Task T_i has higher priority than task T_j if $P_i < P_j$. Therefore, the set of tasks that may preempt T is the set of tasks with higher priority. Unlike EDF, $U \leq 1$ is not a sufficient condition for feasible schedule with RMS. There exists no polynomial time schedulability test for RMS. An iterative method is employed to estimate the response time of each task and compare it against the deadline.

$$S_i^{n+1} = C_i + \sum_{T_j \in hp(T_i)} \lceil \frac{S_i^n}{P_j} \rceil (C_j + CRPD(T_i, T_j) + CSC) \qquad (3)$$

where S_i^n is the response time of T_i in the n^{th} iteration, and $hp(T_i)$ represents the set of tasks that have higher priority than T_i.

4. FRAMEWORK OVERVIEW

We first provide an overview of our integrated analysis and locking approach. We propose to statically lock a part of the cache per task, while a part is left unlocked to be used by all the tasks. The locked cache space is spatially shared among the tasks, while the unlocked cache space is temporally shared by all the tasks.

According to Equation 1, the execution time of a task depends on both intra-task WCET and inter-task CRPD. For the locked memory blocks, they do not incur any CRPD as they can not be evicted from the cache and their impact on the WCET can be easily determined. However, for the unlocked memory blocks, we still need to perform static analysis for both intra-task WCET and inter-task CRPD analysis as they use the remaining unlocked cache space.

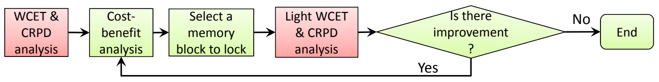


Figure 2: Framework for *Locking + Analysis* approach.

Figure 2 illustrates the flow of our *Locking + Analysis* approach. We first perform intra-task WCET analysis with abstract interpretation [24]. Meanwhile, we also perform inter-task CRPD analysis [12]. Then, for each memory block in the task set, a cost-benefit analysis on WCET and CRPD is carried out for cache locking. This cost-benefit analysis captures the impact of locking a memory block on the WCET, CRPD, and schedulability of all the tasks. Based on this cost-benefit analysis, we choose the most profitable memory block to lock. We perform intra-task WCET analysis and inter-task analysis again after locking this memory block. We call it light WCET & CRPD analysis, because it avoids some unnecessary cache analysis compared to the full-fledged WCET & CRPD analysis. If either the schedulability or the utilization improves, we continue to lock other memory blocks. Otherwise, the iterative process stops and we obtain the final solution.

5. WCET AND CRPD ANALYSIS

In the following, we present a brief description of the static analysis techniques for intra-task WCET and inter-task CRPD estimation (see Figure 3). This analysis ignores cache locking. But this

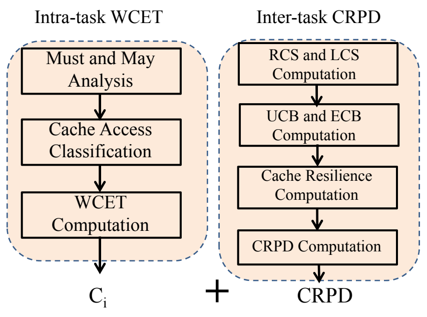


Figure 3: WCET and CRPD Analysis.

background material is required to appreciate the WCET and CRPD estimation in the presence of cache locking for our cost-benefit analysis presented in the next section.

5.1 Intra-Task WCET

As shown in Figure 3, intra-task WCET analysis involves three steps. First, it performs abstract interpretation — must and may analysis — based on abstract cache states [24]. Must analysis determines the set of memory blocks that are guaranteed to be present in the cache at a program point. May analysis captures the set of memory blocks that may be present in the cache at a program point. Next, the memory blocks are classified based on the must and may analysis. The memory blocks present in the abstract cache states of must analysis are classified as always hits; the memory blocks not present in the abstract cache states of may analysis are classified as always misses; the remaining memory blocks belong to non-classified category, i.e., they are assumed to be cache misses during WCET estimation. Finally, the WCET is derived based on the memory accesses classification.

5.2 Inter-Task CRPD

A preempted task T incurs CRPD as some of its useful memory blocks are evicted from the cache by the preempting task T'. The "useful" memory blocks are those memory blocks that have been loaded into the cache before preemption and may be accessed again after preemption. Thus, the key to CRPD analysis is to determine the useful cache blocks of the preempted task and verify whether they could survive after the preemption.

Recently, Kleinsorge et al. [12] proposed a CRPD estimation approach for set associative caches that combines techniques for direct mapped caches [21] with resilience analysis for set associative caches [3]. Concrete cache states are fundamental to inter-task CRPD analysis [12, 21]. Given a program point, it may be reached via multiple program paths, which leads to multiple concrete cache states. Thus, in general, it is infeasible to maintain all the possible concrete cache states for large programs with complex control flow. Inter-task CRPD analysis aims to identify the cache states with the largest number of useful memory blocks and thus higher preemption delay. The subsumed cache states (with lower number of useful memory blocks) can be safely removed. Hence, it is feasible to use concrete cache states for inter-task CRPD analysis as shown in [12, 21]. We adopt the technique in [12] to estimate the CRPD for a single preemption. The approach in [12] depends on the computation of UCB (useful cache blocks) and ECB (evicting cache blocks). The resilience of a UCB defines the maximum number of allowed memory accesses from the preempting task before it can be evicted. Figure 3 shows the four steps of CRPD analysis in [12] that are detailed in Appendix A.

During the execution of task T, it is possible that higher priority task T' preempts lower priority task T multiple times. The preemption bound imposed by T' on T is denoted as $n(T,T')$ in Equation 2. $n(T,T')$ depends on the scheduling policy. For EDF scheduling policy, we use the approach in [11] to bound $n(T,T')$; for RMS scheduling, $n(T,T')$ is a by-product of the response time computation as shown in [18] and Equation 3.

6. LOCKING ALGORITHM

As we have mentioned, existing locking techniques for multitasking systems allocate the entire cache for locking [4, 23]. Such locking techniques eliminate the CRPD analysis at the expense of poor performance. Thus, it is relatively easy to compute the memory blocks to be locked. While in our approach, there is a complex interplay between cache locking and its impact on schedulability analysis. When a memory block of task T is locked in the cache, T generally benefits from the locking. However, it also takes away valuable cache space from the remaining tasks and also changes their CRPD. Any exact locking algorithm for our approach will have exponential complexity. Thus we design an efficient heuristic to decide on the memory blocks to be locked.

As noted earlier, we first perform intra-task WCET analysis and inter-task CRPD for each task in the task set (see Figure 2) when no memory block is locked. Then we compute the processor utilization and response time for each task. As a by-product of intra-task WCET analysis, we have the abstract cache states (must and may) at each program point. We also collect the memory blocks along the WCET path and their execution frequencies for each task. Similarly, we record the worst-case preemption point and the corresponding UCB and ECB for each task during CRPD analysis. We design an iterative solution to select the memory blocks for locking. In each iteration, we choose the most beneficial memory block for locking. The benefit of locking a memory block is defined differently for different scheduling policy (see section 6.3). We stop this process when there is no benefit due to locking and the remaining cache space is left unlocked.

The cost and benefit of locking is based on the following observations. Given a memory block $m \in T$, if m is locked, then all the accesses to m are cache hits. But as cache size is reduced, it might have negative impact on the other memory blocks mapped to the same cache set for all the tasks including T. For task T, its intra-task WCET might be improved if the benefit of locking m is greater than the cost on other memory blocks. However, for other tasks, there is no positive effect on their intra-task WCET. Finally, the CRPDs for all the tasks are usually reduced as the effective cache size is reduced after locking m. In the following, we show how to estimate the cost and benefit of locking memory block m. We assume $m \in T$ and m is mapped to cache set s.

6.1 Cost-benefit analysis within a task

We only consider the memory blocks of task T along the WCET path for locking as locking the other memory blocks has no benefit. Let m be a memory block along the WCET path of T and f_m be the execution frequency of m along the WCET path of T. We use lat_m to denote the access latency of memory block m. lat_m is determined by the classification (cache hit or cache miss) of memory block m in must/may analysis. We use lat_{hit} and lat_{miss} to represent the cache hit and miss latency, respecitively. Then, the benefit of locking m on the WCET of T is

$$wcet\_benefit_m^T = (lat_m - lat_{hit}) \times f_m$$

However, locking m may also have negative impact on the other memory blocks of T mapped to the same cache set s as the number of cache blocks in set s is now reduced by one. Let C be the abstract cache state for must analysis of set s. If $m \in C$, m is classified as cache hit before cache locking, thus locking m does not evict any other memory block from the cache and $wcet\_cost_m^T = 0$. However, if $m \notin C$, locking m will evict out the memory block m' with age $A-1$ in C from the cache, which results in cache miss for the accesses of m'. In this case, the cost of locking m is

$$wcet\_cost_m^T = \sum_{(m' \in M_s) \wedge (age_{m'}^C = A-1)} (lat_{miss} - lat_{hit}) \times f_{m'}$$

where M_s is the set of memory blocks mapped to set s in T, and $f_{m'}$ indicates the execution frequency of m' along the worst-case path. Therefore the WCET gain of T by locking m is

$$wcet\_gain_m^T = wcet\_benefit_m^T - wcet\_cost_m^T$$

Apart from the influence on the intra-task WCET of T, locking m may also affect the CRPD of T. We assume T is preempted by another task T'. Obviously, locking m will not generate any new useful cache block because the cache size is reduced. As mentioned before, we record the UCB, ECB and the preemption point that lead to the worst-case CRPD for this preemption. Suppose M_s^u is the set of useful cache blocks in T mapped to set s at this point and $M_s^{u'}$ ($M_s^{u'} \subset M_s^u$) is the set of blocks that contribute to the CRPD before locking m. To model the effect of locking m, we update ECB of set s and the resilience of any block in M_s^u. With the new ECB and updated resilience, we can obtain $M_s^{u''} \subset M_s^u$, the new set of blocks that contribute to the CRPD after locking m. So the CRPD gain by locking m due to one preemption by T' is

$$crpd\_gain_m^{TT'} = (|M_s^{u'}| - |M_s^{u''}|) \times (lat_{miss} - lat_{hit})$$

Therefore, the total CRPD gain of T by locking m is

$$crpd\_gain_m^T = \sum_{T' \in pt(T)} crpd\_gain_m^{TT'} \times n(T,T')$$

where $pt(T)$ is the set of tasks that may preempt T and $n(T,T')$ is the bound on the number of preemptions of T imposed by T'. Finally the overall execution time gain of T by locking m is

$$time\_gain_m^T = wcet\_gain_m^T + crpd\_gain_m^T$$

6.2 Cost-benefit analysis of other tasks

Let $T' \neq T$ and $m' \in T'$ be a memory block along the WCET path of T'. We assume m' and m are mapped to the same cache set s and m' is in the abstract cache state of must analysis C. If the age of m' is $A-1$, then locking m will evict m' out of cache. Thus, locking m has negative impact on the WCET of other tasks. We define the WCET cost on T' by locking m as follows

$$wcet\_cost_m^{T'} = \sum_{(m' \in M_s') \wedge (age_{m'}^C = A-1)} (lat_{miss} - lat_{hit}) \times f_{m'}$$

where M_s' is the set of memory blocks mapped to set s in T' and $f_{m'}$ is the execution frequency of m' along the WCET path.

The CRPD gain of T' by locking m, $crpd\_gain_m^{T'}$, can be obtained via the same approach as in section 6.1. Thus, the overall execution time gain of task T' by locking m is

$$time\_gain_m^{T'} = crpd\_gain_m^{T'} - wcet\_cost_m^{T'}$$

6.3 Memory block selection strategy

We design different memory block selection strategies for EDF and RMS scheduling policies.

EDF Scheduling. Equation 1 is a sufficient and necessary condition for feasible schedule. Thus we select the memory blocks based on their impact on total processor utilization as follows

$$util\_gain_m = \frac{time\_gain_m^T}{P} + \sum_{T' \in \mathscr{T} \setminus \{T\}} \frac{time\_gain_m^{T'}}{P'}$$

where P is the period of task T, P' is the period of task T' and $\mathscr{T}$ is the task set. The utilization gain of locking a memory block is used as a metric to select the memory blocks for locking. In each iteration, we select the memory block with maximum utilization gain over all memory blocks in the task set.

RMS Scheduling. Utilization (Equation 1) is not a sufficient condition for feasible schedule in RMS. Thus, for RMS, we first need to ensure the schedulability of the task set. For each task, its response time can be computed using the iterative method provided by Equation 3. We focus on the tasks with response time greater than their deadline, and among them try to optimize the response time of the task with highest priority first. Based on Equation 3, in order to improve the response time of a task T, we can either reduce the execution time of T, or improve the execution time of the tasks with higher priority than T. So, when we try to lock a memory block $m \in T$, the corresponding response time gain of T is

$$\begin{aligned} rsp\_gain_m^T =& wcet\_gain_m^T \\ &+ \sum_{T' \in hp(T)} (crpd\_gain_m^{TT'} - wcet\_cost_m^{T'}) \times n(T,T') \end{aligned}$$

where $hp(T)$ is the set of tasks with higher priority than T. When we try to lock a memory block $m' \in T'$ with higher priority than T, the corresponding response time gain of T is

$$\begin{aligned} rsp\_gain_{m'}^T =& (crpd\_gain_{m'}^{TT'} + wcet\_gain_{m'}^{T'}) \times n(T,T') - wcet\_cost_{m'}^T \\ &+ \sum_{T'' \in hp(T) \setminus \{T'\}} (crpd\_gain_{m'}^{TT''} - wcet\_cost_{m'}^{T''}) \times n(T,T'') \end{aligned}$$

where T'' is a task with higher priority than T and $T'' \neq T'$, and $n(T,T'')$ represents the number of preemption bound imposed on T by T''. The WCET and CRPD gain are different for T' and T'' through locking of m'. But both of them contribute to the response time gain of T. Thus, $rsp\_gain_{m'}^T$ includes both of them. Because $m' \in T'$, $wcet\_gain_{m'}^{T'}$ is obtained via the approach in section 6.1. Meanwhile, $m' \notin T$ and $m' \notin T''$, thus, $wcet\_cost_{m'}^T$ and $wcet\_cost_{m'}^{T''}$ are computed similarly via the approach in section 6.2. $crpd\_gain_{m'}^{TT'}$ and $crpd\_gain_{m'}^{TT''}$ can be obtained via the same approach as in section 6.1.

We select the memory block with the maximum response time gain to lock, while at the same time we ensure the utilization gain of this block to be non-negative. After all the tasks are schedulable, we apply the same method used for EDF scheduling to further minimize the utilization. For both scheduling policies, after each iteration, we recompute the abstract cache states of set s where the selected memory block m is mapped to, and then recompute the WCET. Similarly, the cache states for CRPD computation at each program point are also updated. We then recompute the UCB and ECB for each task in task set and obtain the new CRPD. Based on the new WCET and CRPD, we derive the metric value. If there is improvement, we continue to lock. The iterative approach stops only when all the memory blocks are locked or there is no improvement after locking any memory block. The pseudo-code of the algorithms appear in Appendix C.

7. EXPERIMENTAL EVALUATION

In this section, we quantitatively compare our approach with static analysis [24, 12], *ASRV-Locking* [4], and *PD-Locking* [23].

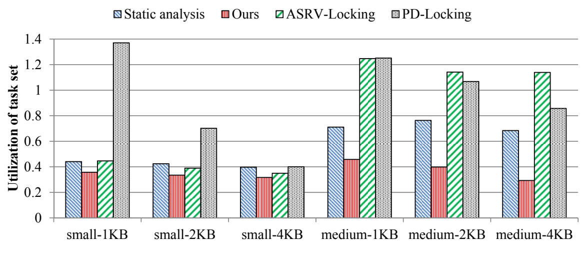
Experiments Setup. We use similar task sets used in [23, 4]. The task sets are shown in Table 1. They contain one small and one medium task set. All the tasks are from MRTC benchmark suite [10].

We assume the deadline of a task is equal to its period. Our framework is built on top of the open-source WCET analysis tool Chronos [14]. All the tasks are compiled with gcc cross-compiler for an ARM-like instruction set [6].

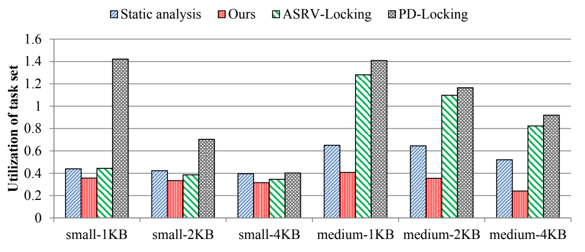
We assume there is only one level of instruction cache. Instruction hit latency is 1 cycle, while the cache miss latency is 30 cycles. The locking routine is stored in non-cacheable memory and it uses five instructions to load and lock a memory block [2, 1]. Thus, the cost of locking a memory block is 150 cycles. The cache is 4-way set-associative with block size of 32 bytes. We also assume each context switch takes 1,000 cycles per preemption for all the approaches. For a fair comparison, we assume there is no line buffer for Aparicio et al.'s approach [4].

Table 1: Characteristics of task sets

Task set	Task	Code size (bytes)	Period
small	jfdctint	5,512	1,500,000
	crc	2,032	2,000,000
	fir	1,144	4,200,000
	matmult	1,632	3,900,000
medium	minver	6,256	720,000
	qurt	2,048	44,000
	jfdctint	5,512	680,000
	fdct	5,176	370,000



(a) Utilization with EDF



(b) Utilization with RMS

Figure 4: Utilization comparison of different approaches.

Utilization Comparison. Figure 4 (a) and (b) present the utilization comparison of different approaches under EDF and RMS scheduling. *small-X KB* (*medium-X KB*) denotes small (medium) task set with cache size of *X* KB. As shown, our integrated cache analysis and locking substantially improves the utilization irrespective of task set size, scheduling policy, and cache size. *PD-Locking* has high utilization when the cache size is small. For *PD-Locking*, the locked memory blocks for each task are very limited and most of the memory accesses are serviced from main memory instead of cache. As a result, the WCET of the tasks and utilization of the task set are high. In the medium task set, the utilization of *ASRV-Locking* is also high. First, the code size for tasks in medium task set is large. Thus, there are still many unlocked memory blocks. Second, the period of task *qurt* is much smaller than the other tasks, and these tasks suffer many preemptions from *qurt*. Thus, the relocking cost also contributes a lot to the utilization.

More results are detailed in Appendix B.

8. CONCLUSION

In this paper, we present an approach that integrate instruction cache analysis and locking in multitasking preemptive real-time systems. A portion of the cache is locked by the tasks in the task set, while the remaining portion is used by all the tasks. We propose an algorithm based on accurate cost-benefit analysis to select the appropriate memory contents to lock. Experimental results show that our approach outperforms previous techniques that either time-multiplexes the cache among all the tasks or statically shares and fully locks the cache.

9. ACKNOWLEDGMENTS

This work was partially supported by Singapore Ministry of Education Academic Research Fund Tier 2, MOE2009-T2-1-033.

10. REFERENCES

[1] ARM 940T (rev 2) technical reference manual.
[2] Intel XScale core developer manual.
[3] S. Altmeyer, C. Maiza, and J. Reineke. Resilience analysis: tightening the crpd bound for set-associative caches. In *LCTES*, 2010.
[4] L. C. Aparicio et al. Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems. *J. Syst. Archit.*, 57(7), 2011.
[5] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *RTNS*, 2006.
[6] D. Burger and T. M. Austin. The simplescalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3), 1997.
[7] A. M. Campoy et al. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *ECRTS*, 2005.
[8] H. Ding, Y. Liang, and T. Mitra. WCET-centric partial instruction cache locking. In *DAC*, 2012.
[9] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS*, 2007.
[10] J. Gustafsson et al. The mälardalen WCET benchmarks - past, present and future. In *WCET*, 2010.
[11] L. Ju, S. Chakraborty, and A. Roychoudhury. Accounting for cache-related preemption delay in dynamic priority schedulability analysis. In *DATE*, 2007.
[12] J. C. Kleinsorge, H. Falk, and P. Marwedel. A synergetic approach to accurate analysis of cache-related preemption delay. In *EMSOFT*, 2011.
[13] C.-G. Lee et al. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. *IEEE Trans. Comput.*, 47(6), 1998.
[14] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury. Chronos: A timing analyzer for embedded software. *Sci. Comput. Program.*, 69(1-3):56–67, 2007.
[15] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *RTSS*, 1996.
[16] 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 Syst.*, 48(6):638–680, 2012.
[17] Y. Liang and T. Mitra. Cache modeling in probabilistic execution time analysis. In *DAC*, 2008.
[18] C. L. Liu and J. Layland. Scheduling algorithms for multiprogramming in a hard realtime enviroment. *Journal of the ACM*, 20(1), 1973.
[19] T. Liu, M. Li, and C. J. Xue. Minimizing wcet for real-time embedded systems via static instruction cache locking. In *RTAS*, 2009.
[20] T. Liu, M. Li, and C. J. Xue. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Syst.*, 48(2), 2012.
[21] H. S. Negi, T. Mitra, and A. Roychoudhury. Accurate estimation of cache-related preemption delay. In *CODES+ISSS*, 2003.
[22] S. Plazar et al. WCET-aware static locking of instruction caches. In *CGO*, 2012.
[23] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS*, 2002.
[24] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache andpath analyses. *Real-Time Syst.*, 18(2/3), 2000.
[25] H. Tomiyama and N. D. Dutt. Program path analysis to bound cache-related preemption delay in preemptive real-time systems. In *CODES*, 2000.
[26] X. Vera, B. Lisper, and J. Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1), 2007.
[27] M. Verma et al. Scratchpad sharing strategies for multiprocess embedded systems: a first approach. In *ESTIMedia*, 2005.
[28] R. Wilhelm et al. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), 2008.

APPENDIX

A. INTER-TASK CRPD ANALYSIS

In this section, we detail the steps to compute the CRPD by Kleinsorge et al. [12].

RCS and LCS Computation. The useful memory blocks are computed using two different types of cache states: the reaching cache states (RCS) and live cache states (LCS). At a program point p, RCS_p is the set of possible cache states when p is reached via any incoming program path. Conversely, at a program point p, LCS_p represents the set of possible cache states via any outgoing program path from p. The cache states RCS and LCS can be computed via forward/backward fix-point data flow analysis [12, 21].

UCB and ECB Computation. We use UCB_p to denote the set of useful memory blocks at program point p. UCB_p is computed as

$$UCB_p = \{c \cap c' | c \in RCS_p, c' \in LCS_p\}$$

where $c \cap c'$ is defined as

$$c \cap c' = \{b | \exists\, 0 \leq j < K \;\; s.t.\; c[j] = b, \exists\, 0 \leq k < K \;\; s.t.\; c'[k] = b\}$$

Evicting cache block (ECB) captures the memory blocks that may be accessed during the execution of the preempting task. Thus

$$ECB = RCS_{exit}$$

where *exit* is the exit point of the preempting task.

Cache Block Resilience. Given a useful memory block m at a program point p, its survival upon preemption depends on its resilience to the preempting task. We define its resilience res_p^m as the maximum number of allowed memory accesses from the preempting task before m can be evicted and is computed as follows. We define the distance of useful memory block m at program point p

$$distance_p^m = \begin{cases} A-1 & if\ age_m^{\downarrow} + age_m^{\uparrow} \geq A \\ age_m^{\downarrow} + age_m^{\uparrow} & otherwise. \end{cases}$$

where $age_m^{\downarrow}$ and $age_m^{\uparrow}$ denote the maximum age of m in RCS_p and LCS_p, respectively. Then, the resilience is defined as

$$res_p^m = (A-1) - distance_p^m$$

CRPD Computation. We can now bound the CRPD based on the UCB of preempted task T and ECB of preempting task T'. Let UCB_p be the set of useful cache blocks at a program p of the preempted task T and ECB be the set of evicting cache blocks of the preempting task T'. For any $u \in UCB_p$ and $e \in ECB$

$$CRPD_{(u,e)}^p = |u \setminus \{m | res_p^m \geq |e|\}| \times CRT$$

where CRT is the reloading overhead of one memory block. Then, the CRPD at this program point p is the maximum among all the possible combinations of UCB of T and ECB of T'

$$CRPD^p = \max_{u \in UCB_p, e \in ECB} CRPD_{(u,e)}^p$$

The CRPD for this preemption is the maximum CRPD over all the program points. That is

$$CRPD(T, T') = \max_{p \in PP} CRPD^p$$

where PP is the set of program points of the preempted task T.

B. EXTRA EXPERIMENTAL RESULTS

In this section, we show more experimental results with our approach, including response time speedup, utilization breakdown, unlocked cache space, and runtime of our approach. The details are shown in the following subsections.

B.1 Response Time Speed-up

We compare the different approaches using response-time speedup metric proposed in [4] for RMS policy. It is defined as follows.

$$speedup = \frac{period}{reponse\ time}$$

It is calculated for the lowest priority task and indicates the slack available in the schedule. Thus, a speedup greater than or equal to 1 implies that the task set is schedulable. Figure 5 shows the response time speed-up for the task sets with varying cache size. Clearly, with our approach, the tasks with lowest priority are always schedulable. However, the lowest priority task in medium task set with *ASRV-Locking* and *PD-Locking* are not schedulable in most of the cases.

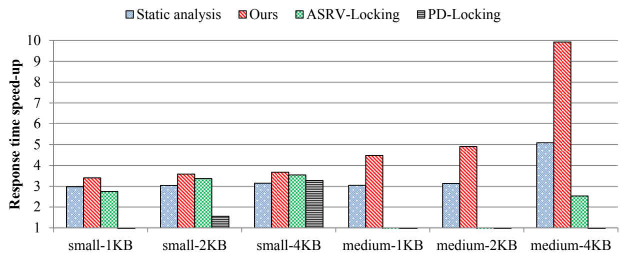


Figure 5: Response time speed-up.

B.2 Utilization Breakdown

Figure 6 details the contribution to the utilization by WCET, CRPD and re-locking overhead, respectively for the medium task set with 2KB cache size under RMS scheduling policy. Compared to static analysis, our approach either significantly reduces the WCET (*qurt*) or nearly eliminates the CRPD (*minver*, *jfdctint* and *fdct*). While for *ASRV-Locking*, we observe a great contribution to utilization due to re-locking overhead (*jfdctint* and *fdct*). Finally, the WCET using *ASRV-Locking* and *PD-Locking* are usually large, because the unlocked memory blocks are all serviced by main memory instead of cache.

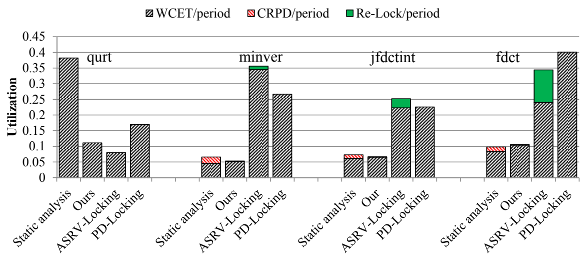
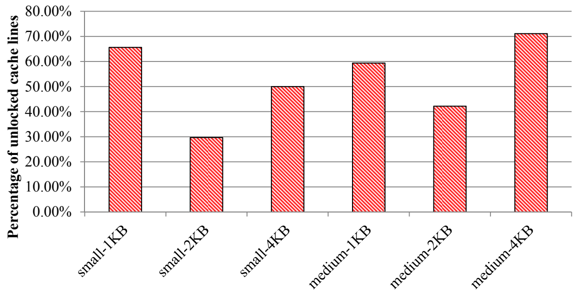


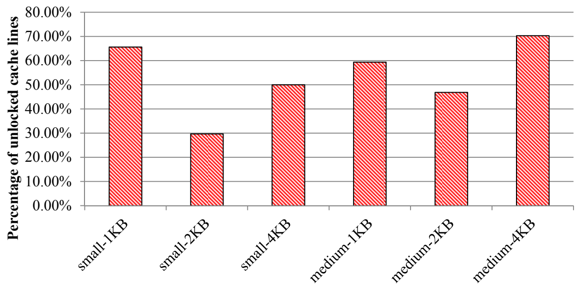
Figure 6: Utilization breakdown for medium-2KB.

B.3 Unlocked Cache Space

Figure 7 (a) and (b) show the percentage of the unlocked cache lines of our approach under EDF and RMS, respectively. The percentage of the unlocked cache space depends on the cache size and the scheduling policy. As shown, with our approach, there is a portion of cache space left unlocked for all the settings. The unlocked cache space can be used by all the tasks in the task set. We also notice that the percentage of unlocked cache lines of 2KB cache is smaller than that of 1KB and 4KB cache. When the cache is small, our approach decides to lock only a small portion of the cache. It is because locking more memory blocks may have significant negative impact on the WCET of the tasks. On the other hand, when the cache is big, more memory blocks can be classified as cache hits and locking those memory has no benefit. Thus, our approach decides to lock only a small portion of a big cache.



(a) Percentage of unlocked space with EDF



(b) Percentage of unlocked space with RMS

Figure 7: Percentage of unlocked cache lines with our approach.

B.4 Runtime of Our Approach

Table 2 presents the runtime of our approach under both EDF and RMS scheduling policy with different cache sizes. We perform all the experiments on 2.53GHz Intel Xeon CPU with 24GB memory. Clearly, our locking algorithm runs efficiently. The overall runtime depends on the number of locked memory blocks, WCET analysis and CRPD analysis. We notice that for the small task set, the runtime of 2KB cache is higher than that of 4KB cache. In small task set, the code size of *crc* is about 2KB. When the cache size is set to 2KB, *crc* has complicated RCS and LCS analysis that leads to long CRPD computation time.

Table 2: Runtime of our approach

Task set	Cache size	EDF (seconds)	RMS (seconds)
small	1KB	4.65	4.68
	2KB	22.72	22.82
	4KB	4.32	4.29
medium	1KB	1.67	1.67
	2KB	33.94	37.19
	4KB	133.44	137.05

C. INTEGRATED LOCKING + ANALYSIS ALGORITHMS

In this section, we present the detailed locking + analysis Algorithms used in our approach, including cost-benefit analysis algorithm, utilization optimization algorithm and schedulability improvement algorithm. The details are shown as follows.

C.1 Cost-benefit Analysis Algorithm

Algorithm 1 presents the detailed cost-benefit analysis by locking a memory block m. For each task T_i in the task set $\mathscr{T}$, if m belongs to T_i, then there may be WCET benefit for T_i by locking m as all the accesses to m are cache hits after locking (line 5). On the other hand, locking m also impacts the other memory blocks mapped to the same cache set in T_i as the effective cache size is reduced after locking m, which may leads to more cache misses. Therefore we compute the cost by locking m for T_i (line 6). If m does not belong to T_i, obviously, there is no benefit for T_i by locking m. Thus we only calculate the cost for T_i by locking m (line 8-9). The WCET gain for T_i should consider both WCET benefit and WCET cost (line 10). Locking memory block m also impacts the CRPD as the effective cache size is reduced after locking. Thus, we also compute the corresponding cost and benefit of CRPD for task T_i by locking m (line 11). Finally, The overall execution time gain for task T_i includes both the WCET gain and CRPD gain (line 12).

Algorithm 1: Cost-benefit analysis on WCET and CRPD

```
Input: Task set $\mathscr{T} = \{T_1, T_2 ... T_N\}$, cache configuration
       $config$ and candidate memory block $m$
Output: WCET gain $wcet\_gain_m^{T_i}$ and CRPD gain $crpd\_gain_m^{T_i}$
        by locking memory block $m$ for each task $T_i \in \mathscr{T}$
1  begin
2      foreach $T_i \in \mathscr{T}$ do
3          Suppose $M_i$ is the set of memory blocks of $T_i$;
4          if $m \in M_i$ then
5              $wcet\_benefit_m^{T_i}$ = wcet_benefit_self();
6              $wcet\_cost_m^{T_i}$ = wcet_cost_self();
7          else
8              $wcet\_benefit_m^{T_i} = 0$;
9              $wcet\_cost_m^{T_i}$ = wcet_cost_others();
10         $wcet\_gain_m^{T_i} = wcet\_benefit_m^{T_i}$ - $wcet\_cost_m^{T_i}$;
11         $crpd\_gain_m^{T_i}$ = crpd_cost_benefit_analysis();
12         $time\_gain_m^{T_i} = wcet\_gain_m^{T_i} + crpd\_gain_m^{T_i}$;
13
14 end
```

C.2 Utilization Optimization Algorithm

Algorithm 2 shows the details of utilization optimization. We first perform one round of WCET and CRPD analysis for each task $T_i \in \mathscr{T}$ (line 3-9). For each task T_i, we perform abstract cache state analysis and compute the WCET (line 4-5). We also perform RCS and LCS analysis for each task T_i (line 6). Based on the RCS and LCS analysis results, we calculate the UCB and ECB, as well as the resilience for each useful cache block (line 7-8). Then we do the CRPD analysis for the task set (line 9). With the CRPD and WCET, the initial utilization of the task set is then carried out (line 10). Later, we iteratively select memory blocks with the maximum utilization gain to lock. For each candidate memory block m, we first check whether it is locked or not (line 16). Meanwhile, we check whether the corresponding cache set is fully locked or not (line 16). If m has been locked or there is no free space in the corresponding cache set that *m* mapped to, we skip *m* and try other candidates. When we find a memory block *m* that can be locked, we first perform the cost-benefit analysis on WCET and CRPD by using Algorithm 1 (line 17). Then, we calculate the utilization gain for each task in the task set (line 18). The total utilization gain for the entire task set by locking *m* is the summation of utilization for all the tasks in the task set (line 19). We compare *m* with the candidate memory block *mblk* that currently has the most utilization gain (line 20). If *m* has more utilization gain than *mblk*, we update *mblk* with *m* (line 21-22). We continue to do this until all candidate memory blocks are considered. If we find no memory block with positive utilization gain, this algorithm will terminate (line 42-43). Otherwise, we will end up with a memory block *mblk* that has the maximum utilization gain. We lock *mblk* into the cache (line 27). For each task, we update the abstract cache states in the cache set that *mblk* mapped to, and recompute the WCET (line 29-30). We also update the RCS and LCS for this particular cache set, and recompute the UCB, ECB and resilience (line 31-33). After the resilience for each useful cache block is updated, we perform the CRPD analysis again to get the new CRPD (line 34). Based on the new WCET and CRPD, we obtain the new utilization of the task set (line 35). If there is improvement on utilization of the task set, we update the utilization and add *mblk* to the set of locked memory blocks, and continue to lock other memory blocks (line 37-18). Otherwise we stop locking and obtain the final results (line 40).

C.3 Schedulability Improvement Algorithm

Algorithm 3 presents the detailed approach to improve schedulability for RMS. For a task set $\mathscr{T}$ in RMS, since Equation 1 is not a sufficient condition for feasible schedule in RMS, we should first check the schedulability of $\mathscr{T}$ based on the response time of each task. Therefore, We also need to perform one round of WCET and CRPD analysis first for the task set as we did in Algorithm 2 (line 3-9). Then, apart from computing the utilization for the task set (line 10), we also need to calculate the corresponding response time for each task in the task set (line 11). We check the schedulability by comparing response time of each task with its deadline. If all tasks meet their deadline, we set the boolean variable *is_sch* to *true* (line 13). In this case, we stop locking for improving schedulability, and continue to optimize utilization with Algorithm 2 (line 14-15). Otherwise, we choose the highest priority task T among the tasks that do not meet their deadline, and try to improve its response time first (line 16). Based on Equation 3, the response time of T is mainly determined by T and the tasks that can preempt T. Thus, we only consider locking memory blocks belong to T or tasks that can preempt T (line 19-20). For such a memory block *m* belongs to T or tasks that can preempt T, if it is not locked and there is free space in the corresponding cache set, we carry out its cost and benefit analysis (line 23-24). After that, we perform utilization gain analysis by locking *m* as we did in Algorithm 2 (line 25-26). Apart from the utilization gain, we also compute the response time gain by locking *m* (line 27). We compare the response time gain between *m* and *mblk* that currently has the most response time gain. If *m* has higher response time gain than *mblk* and the utilization gain of *m* is not negative, we update *mblk* with *m* (line 28-30). We continue to do this until all candidate memory blocks are considered. If there is no suitable memory block to lock, we stop locking (line 51). Otherwise, we select the memory block *mblk* with the maximum response gain on T to lock (line 35). Then, we recompute the new utilization as we do in Algorithm 2, as well as the new response time (line 36-44). If utilization of the task set does not become worse and there is response time improvement on T, we add *mblk* to the set of locked memory blocks and continue to check the schedulability for the task set (line 46) Otherwise, we stop locking (line 51).

Algorithm 2: Utilization Optimization for EDF and RMS

```
Input: Task set $\mathscr{T} = \{T_1, T_2 ... T_N\}$ and cache configuration
       $config$
Output: Set of locked memory blocks $lock\_set$ and utilization
        after locking $util$
1  begin
2      $stop\_locking := false$; $lock\_set := null$;
3      foreach $T_i \in \mathscr{T}$ do
4          abstract_cache_states_analysis($T_i$, $config$);
5          wcet_analysis();
6          rcs_lcs_analysis($T_i$, $config$);
7          ucb_ecb_computation();
8          resilience_computation();
9      crpd_analysis($\mathscr{T}$);
10     $util$ = utilization_computation($\mathscr{T}$);
11     while $(!stop\_locking)$ do
12         $mblk := null$; $util\_gain_{mblk} := 0$;
13         foreach $T_i \in \mathscr{T}$ do
14             foreach $m \in M_i$ do
15                 Suppose $m$ is mapped to cache set $s$;
16                 if $m \notin lock\_set \wedge !is\_fully\_locked(s)$ then
17                     cost_benefit_analysis($\mathscr{T}$, $config$, $m$);
                       foreach $T_i \in \mathscr{T}$ do
18                         $util\_gain_m^{T_i} = time\_gain_m^{T_i} / P_i$;
19                     $util\_gain_m = \sum_{T_i \in \mathscr{T}} util\_gain_m^{T_i}$;
20                     if $util\_gain_m > util\_gain_{mblk}$ then
21                         $util\_gain_{mblk} = util\_gain_m$;
22                         $mblk = m$;
23
24
25
26         if $mblk \neq null$ then
27             lock_to_cache($mblk$);
28             foreach $T_i \in \mathscr{T}$ do
29                 update_abstract_cache_state($T_i$, $mblk$,
                   $config$);
30                 wcet_analysis();
31                 update_rcs_lcs($T_i$, $mblk$, $config$);
32                 ucb_ecb_computation();
33                 resilience_computation();
34             crpd_analysis($\mathscr{T}$);
35             $new\_util$ = utilization_computation($\mathscr{T}$);
36             if $new\_util < util$ then
37                 $util = new\_util$;
38                 $lock\_set := lock\_set \cup \{mblk\}$;
39             else
40                 $stop\_locking := true$;
41
42         else
43             $stop\_locking := true$;
44
45
46 end
```

Algorithm 3: Schedulability Improvement for RMS

```
Input: Task set $\mathscr{T} = \{T_1, T_2 ... T_N\}$ and cache configuration $config$
Output: Set of locked memory blocks $lock\_set$ and utilization after locking $util$
begin
    $stop\_locking := false$; $lock\_set := null$;
    foreach $T_i \in \mathscr{T}$ do
        abstract_cache_states_analysis($T_i$, $config$);
        wcet_analysis();
        rcs_lcs_analysis($T_i$, $config$);
        ucb_ecb_computation();
        resilience_computation();
    crpd_analysis($\mathscr{T}$);
    $util$ = utilization_computation($\mathscr{T}$);
    response_time_computation($\mathscr{T}$);
    while $(!stop\_locking)$ do
        $is\_sch$ = check_schedulability($\mathscr{T}$);
        if $is\_sch == true$ then
            break;
        Suppose $T$ is the task with highest priority that cannot be scheduled;
        $rsp\_gain^{T}_{mblk} := 0$;
        $mblk := null$;
        suppose $hp(T)$ is the set of task with higher priority than $T$;
        foreach $T_i \in T \cup hp(T)$ do
            foreach $m \in M_i$ do
                Suppose $m$ is mapped to cache set $s$;
                if $m \notin lock\_set \wedge !is\_fully\_locked(s)$ then
                    cost_benefit_analysis($\mathscr{T}$, $config$, $m$);
                    foreach $T_i \in \mathscr{T}$ do
                        $util\_gain^{T_i}_{m} = time\_gain^{T_i}_{m} / P_i$;
                    $util\_gain_m = \sum_{T_i \in \mathscr{T}} util\_gain^{T_i}_{m}$;
                    $rsp\_gain^{T}_{m}$ = response_time_gain();
                    if $rsp\_gain^{T}_{m} > rsp\_gain^{T}_{mblk} \wedge util\_gain_m >= 0$ then
                        $rsp\_gain^{T}_{mblk} = rsp\_gain^{T}_{m}$;
                        $mblk = m$;



        if $mblk \neq null$ then
            lock_to_cache($mblk$);
            foreach $T_i \in \mathscr{T}$ do
                update_abstract_cache_state($T_i$, $mblk$, $config$);
                wcet_analysis();
                update_rcs_lcs($T_i$, $mblk$, $config$);
                ucb_ecb_computation();
                resilience_computation();
            crpd_analysis($\mathscr{T}$);
            $new\_util$ = utilization_computation($\mathscr{T}$);
            response_time_computation($\mathscr{T}$);
            if $new\_rsp_T > rsp_T \wedge new\_util <= util$ then
                $lock\_set := lock\_set \cup \{mblk\}$;
            else
                $stop\_locking := true$;

        else
            $stop\_locking := true$;


end
```