IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received October 25, 2020, accepted November 10, 2020, date of publication November 16, 2020,
date of current version November 30, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3038170

A Dynamic Instruction Cache Locking Approach for Minimizing Worst Case Execution Time of a Single Task

TINGXU ZHANG, (Member, IEEE), WENGUANG ZHENG, YINGYUAN XIAO, AND GUANGPING XU

School of Computer Science and Engineering, Tianjin University of Technology, Tianjin 300384, China

Corresponding author: Wenguang Zheng (wenguangz@tjut.edu.cn)

This work was supported in part by the National Nature Science Foundation of China under Grant 61702368, in part by the Natural Science Foundation of Tianjin, China, under Grant 18JCQNJC00700 and Grant 17JCYBJC15200, and in part by the Major Research Project of National Nature Science Foundation of China under Grant 61971309.

ABSTRACT In real time embedded system, the embedded software often consists of a set of concurrent tasks, and such tasks are generally subject to timing constraints. In order to satisfy all the timing constraints, precisely predicting the WCET of a task is essential for the task scheduler to construct a feasible schedule for a task set. Caches have been widely used to bridge the gap between high speed processors and relatively slower off-chip memory. However, caches make it extremely harder to predict precise WCET (Worst Case Execution Time) of a task for the simple reason that it is difficult to predict if each cache access is a cache hit or miss. Cache locking is a mechanism which disables the replacement policy of caches and locks some contents (instruction or data) in the caches, such that the accesses to those contents become fully predictable and the WCET of a task is easier to predict. Furthermore, cache locking is also an effective technique to reduce the WCET of a task by locking appropriate contents in the caches. In this paper, we investigate the WCET-aware I-cache (Instruction cache) locking problem and propose an ILP-based (Integer Linear Programming) dynamic I-cache locking approach for reducing the WCET of a task. Our approach not only select locking contents which have largest benefit for reducing WCET of a task, but also finds a good locking point for each locked instruction such that extra execution time spend on locking instructions is also minimized. We have implemented this approach and compared it with two state-of-the-art I-cache locking approaches, the longest path based dynamic cache locking approach proposed in and the min-cut based dynamic locking approach proposed in by using MRTC benchmark suite. The experimental results show that our approach performs better for each benchmark. Compared to the longest path based dynamic approach, our approach achieves the average improvements of 5.8%, 13.8%, 16.1% and 12.6% for the 256B, 512B, 1KB and 2KB caches, respectively. Compared to the min-cut based dynamic approach, our approach achieves the average improvements of 2.2%, 7.6%, 8.2% and 6.4% for the 256B, 512B, 1KB and 2KB caches, respectively.

INDEX TERMS Cache locking, real-time systems, worst-case-executing-time.

I. INTRODUCTION

Modern real-time embedded systems are often suffered from hard timing constrains. Therefore, the WCET of each task needs to be predicted firstly such that a feasible schedule of the tasks which satisfies all the timing constrains can be constructed. However, with the utilization of caches, it is significantly harder to predict precise WCET of a task because of the unpredictability of memory access latency. Since caches are automatically managed by the hardware, it is difficult to know at compile time whether the contents needed to be accessed is in the cache (hit) or not (miss), and a cache hit and a cache miss obviously have different access latencies. Cache locking is an effective way to improve cache predictability. When a content is locked into a cache, it will not be evicted until it gets unlocked. Each access

The associate editor coordinating the review of this manuscript and approving it for publication was Laxmisha Rai.

to this locked content will be a cache hit and generally takes one clock cycle. Obviously, for a fully locked cache, accesses to those unlocked contents will be always cache misses. As a result, the WCET of a task is much easier to be predicted. For real-time embedded systems, tasks running on them are stationary, barely change after compiled and burned. Therefore, the process of cache locking can be done in the compiler. Most modern embedded systems support hardware level cache locking, including line locking and set locking.

Typically, there are two cache locking strategies described as follows, static cache locking and dynamic cache locking.

Static Cache Locking : This strategy does not analyse the life ranges of the locked contents and simply assumes that all the locked contents share the same life range which spans the entire execution time of the task. Thus, the static locking strategy selects locked contents only once, and the locked contents will not be replaced during the entire execution of the task.

Dynamic Cache Locking : This strategy considers the life ranges of the locked contents which means locked contents can be replaced after their life ranges expires. As a result, different locked contents can be stored in a cache as long as their life ranges do not overlap, leading to more efficient utilization of cache space than static strategy.

Furthermore, cache locking technique can also reduce the WCET of a task by selecting appropriate contents to be locked into the caches. Therefore, the main objective of WCET-aware static cache locking approaches is to select a set of locked contents according to the limitation of the cache space such that the WCET of a task can be minimized. For WCET-aware dynamic cache locking approaches, there is an additional task which is the determination of locking points for locked contents according to their life ranges. A selected cache content is loaded and locked into the cache at its locking point.

In this paper, we investigate the WCET-aware dynamic I-cache locking for a single task. Given a task, our objective is to select a set of memory blocks of the code of the task as locked cache contents and determine the locking point of each locked memory block such that the WCET of a task is minimized meanwhile the utilization of cache space is maximized. We make the following major contributions.

- We propose an ILP-based dynamic I-cache locking approach which constructs a unified model for the problems of both locking contents selection and locking points determination. Software Lingo is employed to obtains optimal solution of this ILP model.
- We have implemented our dynamic instruction locking approach and compared it to two state-of-the-art approaches, namely the longest path based dynamic instruction cache locking approach proposed in [24] and min-cut based dynamic instruction cache locking approach proposed in [17], by using a set of benchmarks from the MRTC benchmark suite. The experimental result show that compared to the longest path based dynamic approach, our approach achieves the average improvements of 5.8%, 13.8%, 16.1% and 12.6% for the 256B, 512B, 1KB and 2KB caches, respectively. Compared to the min-cut based dynamic approach, our approach achieves the average improvements of 2.2%, 7.6%, 8.2% and 6.4% 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.

II. RELATED WORK

Many cache locking approaches have been proposed to reduce the WCET of tasks or improve the schedulability of embedded systems. We summarized the previous work from the following perspectives.

A. STATIC CACHE LOCKING

Anand and Barua [1] propose an approach for instruction-cache locking that is able to reduce the average-case run-time of the program. They use a cost-benefit model to determine contents to lock, and iteratively lock the memory block with most benefit into the cache, until the cache is full or the benefit of the memory block with the most benefit is negative. Liang and Mitra [2] introduce temporal reuse profile to accurately and efficiently model the cost and benefit of locking memory blocks in the cache. Ding *et al*. [3] observe that such aggressive locking mechanisms may have negative impact on the overall WCET as some memory blocks with predictable access behavior get excluded from the cache. They introduce a partial cache locking mechanism that has the flexibility to lock only a fraction of the cache. They propose an algorithm to select memory blocks to lock by evaluating the impact of locking. They developed an ILP based algorithm to select lock contents. The WCET of a loop is represent by the memory blocks' time consumption multiply the iteration time. Campoy *et al*. [16] compare the performance of two algorithms for static locking of instruction caches: one using agenetic algorithm for cache contents selection [15] and a pragmatical algorithm, called her-after reference-based algorithm proposed in [4], which uses the string of memory references issued by a task on its worst-case execution path as an input of the cache contents selection algorithm. The genetic algorithm behaves slightly better than the reference-based algorithm with respect to the average slack of tasks but has more execution time spent. Liu *et al*. pointed out in [26] that it is NP difficult to reduce the average case executing time of the system through static cache locking. Adegbija and Gordon-Ross [18] present a phase-based locking technique for data caches. Their technique divides the execution into intervals and groups the intervals with similar characteristics into phases showing data reuse. At least 50% of cache lines are always kept unlocked to avoid extra misses for unlocked data. They show that their technique leads to significant reduction in miss rate and energy saving. They extended their

work in [12] and proposed a new cache locking method to achieve energy saving with minimal performance loss. The method can be used for both instruction and data cache. The selection, loading and retaining are dynamically executed at runtime. According to the experiments, their method has a good effect in reducing the energy consumption of data cache. Puaut and Decotigny [4] explore the use of static cache locking of instruction caches in multitasking real-time systems, addressing both intra-task and inter-task interferences and leading better performance. Dugo *et al*. proposed a method in [10]to mitigate interferences that occur in the memory hierarchy levels. Considering that predictability is a key concept of ARINC-653-compliant systems, their approach uses static cache locking. Two locking content selection approaches are proposed. An average improvement of 27.93% is demonstrated in their experiment.

B. DYNAMIC CACHE LOCKING

Dynamic locking has its own advantages since it shows better performance and flexibility than static locking in most cases, pointed out by Campoy *et al*. in [19]. Qiu *et al*. [5] presents an approach, Branch Prediction-directed Dynamic Cache Locking (BPDCL), to improve system performance through cache conflict miss reduction. Ding *et al*. [6] propose a flexible loop-based dynamic cache locking approach. They not only select the memory blocks to be locked but also the locking points. Isabelle Puaut present an dynamic cache locking approach in [24]. The code of each task is divided into regions and every region has its corresponding cache contents. They use greedy algorithm and genetic algorithm to select the reload point of contents to be locked. At runtime, at region-transition boundaries, statically computed cache contents are loaded into the cache and then the cache is locked. Their experiments show that the worst-case performance of locking contents is comparable with that of unlocked contents. Also, the greedy algorithm can find acceptable solutions quickly, while genetic algorithm is much slower when the initial population is randomly chosen. Zheng and Wu elaborate a min-cut based dynamic cache locking approach in [7]. 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 pathes are reduced simultaneously. Zheng *et al*. [17] propose two algorithms to find appropriate locking point for selected memory blocks. The ILP algorithm achieves optimal solution and the heuristic algorithm obtains approximate optimal solution faster.

C. INSTRUCTION AND DATA CACHE LOCKING

Most of the cache locking approaches [1], [2], [5]–[7], [15], [17] focus on instruction cache locking. And for data cache locking, Vera *et al*. [8] propose an approach for data cache analyzing and locking. Due to the complex situation brought by indirect array accesses (X[Y[j]]) and dynamic memory allocations, the locking procedure cannot be easily determined, even the corresponding memory blocks are locked into cache, misses still happen at run time. They introduce a method that combines static cache analysis and cache locking in order to achieve both predictability and good performance. Furthermore, it allows computing a WCMP(Worst Case Memory Performance) estimate of tasks in a fast and tight way. Xue *et al*. [13] pointed out that the conventional instruction cache locking approach has little effect on data cache. They proposed a data re-allocation approach to minimize the memory block interference frequency by reorganizing data objects in the memory. Data interference graph and memory block interference graph are used to perform data re-allocation and data cache locking. The experimental results show that the miss rate and energy savings are obtained by a set of benchmarks. Wan *et al*. [14] propose a novel data cache locking approach. SPM(Scratch Pad Memory) is used as cache so they don't need to consider the memory mapping problem. They use k-longest-path instead of longest-path to select memory blocks to be locked, making better locking decisions. Zheng and Wu propose two full cache-locking approaches to the D-cache locking problem in [22]. An effective technique for reducing the number of false dependencies between variables is propose and better D-cache utilization is observed. They are the first ones on the WCET-aware dynamic D-cache locking problem considering the live ranges of variables. In addition, there have been some researches on L2 cache locking. Since L2 is usually larger than L1 and shared by multiple cores, function-level contents can be locked and significant improvement can be achieved. Asaduzzaman *et al*. [21] present a cache locking technique for L2 cache. Blocks with the most miss rate in cache are locked in cache in turn. Their experiments show that for the tasks that working set fits in L1 cache, locking provides nearly no benefit. For those tasks with significant L2 accesses, large energy saving and performance improvement are observed.

III. SYSTEM MODEL AND DEFINITIONS

We investigate the WCET-aware dynamic I-cache locking for a single task. A typical processor models of an embedded system is shown in Fig.1. The target processor is consisted of I-cache and D-cache (data cache). In this paper, we focus on I-cache locking problem only. The I-cache is an n-way set associative cache with m sets. Every set has a set number from 0 to $m-1$, and all sets have the same size. Every instruction is mapped into a certain set determined by its memory address. The locking unit of our approach is a cache line.

Given a task, we firstly construct CFG (Control Flow Graph) of the task. The definition of CFG is shown in *Definition* 1.

Definition 1 (Control Flow Graph): The CFG is a classical structure for representing the code of a task, which can be formatted as $D =< N, T, E >$, where $N = \{n_i$: n_i represents a basic block$\}$, $T = \{t_i$: t_i is the execution time of basic block $n_i\}$ and $E = \{(n_i, n_j)$: a control flow from n_i to n_j $\}$.

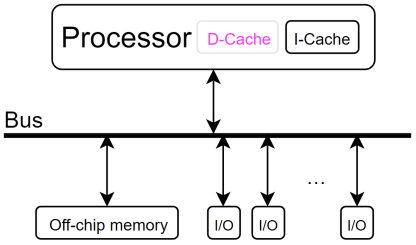


FIGURE 1. System model.

In a CFG, each node denotes a basic block which is a sequence of code that has only one entry and only one exit. Typically, the CFG contains two types of edges: forward edges and back edges. For ease of constructing ILP model, we construct a DAG (Directed Acyclic Graph) by removing all the back edges of the CFG and multiply the weight of each basic block by its maximum iteration times. The definition of a DAG of a task is shown in *Definition* 2.

Definition 2 (Directed Acyclic Graph): Given a CFG of a task, the DAG of the task is a weighted graph $G =< N, W, E >$, where $N = \{n_i$: n_i represents a basic block$\}$, $W = \{w_i$: w_i is the maximum iteration times of n_i multiply the execution time of n_i without using cache$\}$ and $E = \{(n_i, n_j)$: an edge from n_i to n_j $\}$.

Since the locking unit is a cache line, we split each basic blocks into one or more memory blocks such that each memory block is mapped to exactly one cache line. Given a DAG of a task, we transform it into a memory block graph (MBG) defined in *Definition* 3.

Definition 3 (Memory Block Graph): Given a DAG of a task, its memory block graph is a weighted directed acyclic graph $G' =< M, W', E' >$ where $M = \{m_{ij}$: m_{ij} denotes the $j-th$ memory block of the basic block n_i in the DAG of the task$\}$, $W' = \{w'_{ij}$: w'_{ij} is the total execution time of memory block m_{ij}without using cache$\}$ and E' represents the set of edges between memory blocks.

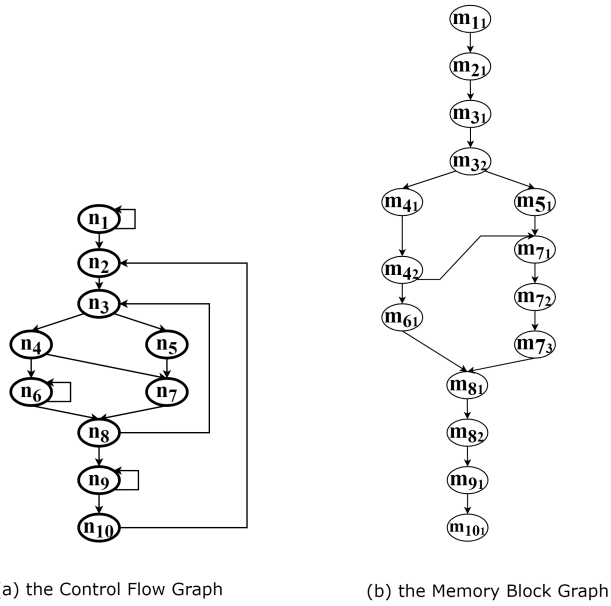
Given a DAG, the MBG can be constructed by the following steps.

1) For each node n_i of the DAG G of a task, create new nodes $m_{ij(j=1,2,...,k)}$ in MBG G'. k is the total number of cache lines that need to lock the instructions of the basic block n_i.
2) For each pair of nodes m_{ij} and $m_{ij+1(j=1,2,...,k-1)}$ in G', create an edge from m_{ij} to m_{ij+1} in G'.
3) For each edge (n_i, n_j) in DAG G, construct an edge (m_{ik}, m_{j1}) in MBG G'.

Fig.2 shows a simple example of the MBG from the CFG.

For the process of generating MBG (Memory Block Graph), the time complexity is O$(n + e)$ where n denotes the number of memory blocks and e denotes the number of edges in DAG.

Given an MBG of a task, the memory blocks of MBG can be regarded as locking candidates. When a memory block of MBG is locked into cache, accesses to this memory block would be a cache hit, and each fetching costs one clock cycle.



(a) the Control Flow Graph (b) the Memory Block Graph

FIGURE 2. Constructing DAG and MBG.

Otherwise, a cache miss occurs and each fetching costs c clock cycles. Obviously, locking a memory block which executes only once provides no benefit for reducing the WCET of a task. Thus, we only consider the memory blocks within a certain loop nest as locking candidates.

The objective of our approach is to not only select memory blocks as locked cache contents but also determine a good locking point for each selected memory blocks such that the WCET of a task is minimized. In this paper, we define that the life range of a particular memory block is the life range of the loop it belongs to. Different memory blocks can share the same cache space as long as their life ranges do not overlap. For a single loop, the locking point of a selected memory block is placed at the preheader of the loop which it belongs to. However, for a loop nest, the memory blocks of inner loop have different choices of locking points. Such memory blocks can be locked at the preheader of its own loop or the preheader of outer loops. Locking at the preheader of outer loops has benefit for reducing the WCET of a task because the iteration times of locking selected memory blocks are reduced and leads to less time cost on loading and locking selected memory blocks. However, if a memory block is locked at the preheader of its outer loop, its life range is extended to span the entire outer loop and cause more overlaps. Thus, the utilization of cache space is reduced and less memory blocks can be locked, which brings increase of WCET.

Two simple examples are shown in Fig.3 and Fig.4 to illustrate the affect of different locking points choices. Given a loop nest, we assume that there are only four memory blocks and the cache size is extremely small which can store only one memory block. $loop_2$ and $loop_3$ are direct inner loop of $loop_1$, and we use I to denote the iteration time of $loop_1$. For memory block m_1, it can be locked at the preheader of $loop_2$ or $loop_1$. The examples compare these two cases.

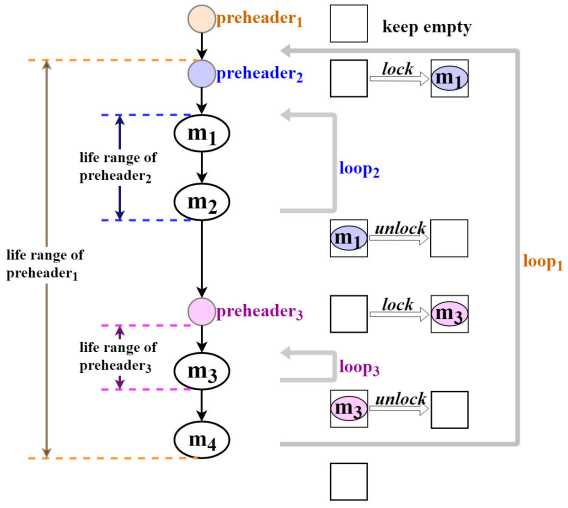


FIGURE 3. Locking at inner preheader.

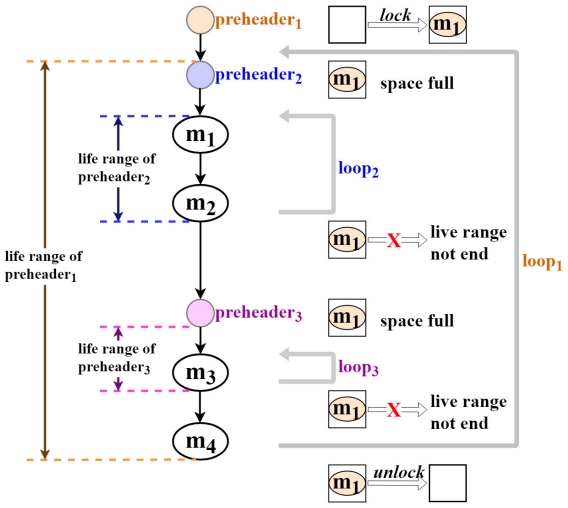


FIGURE 4. Locking at outer preheader.

- *Case* 1; If m_1 is locked at the preheader of $loop_2$ as shown in Fig.3, it can be unlocked when its life range expires. Thus, m_3 also can be locked into the cache because the life ranges of m_1 and m_3 do not overlap. However, m_1 and m_3 need to be locked I times because such actions occurs in $loop_1$. For *Case*1, more memory blocks can be locked into the cache, but more time cost on locking selected memory blocks.
- *Case* 2; If m_1 is locked at the preheader of $loop_1$ as shown in Fig.4, m_1 is locked only once. However, the life range of m_1 extend to the life range of $loop_1$. As a result, the life ranges of m_1 and m_3 overlap and cannot share the same space of the cache. For *Case*2, fewer memory blocks can be locked into the cache, and less time cost on locking selected memory blocks.

In order to analyse the life ranges of memory blocks and record the usage of cache space, we define the Loop Nested Tree (LNT) of a loop nest in *Definition* 4.

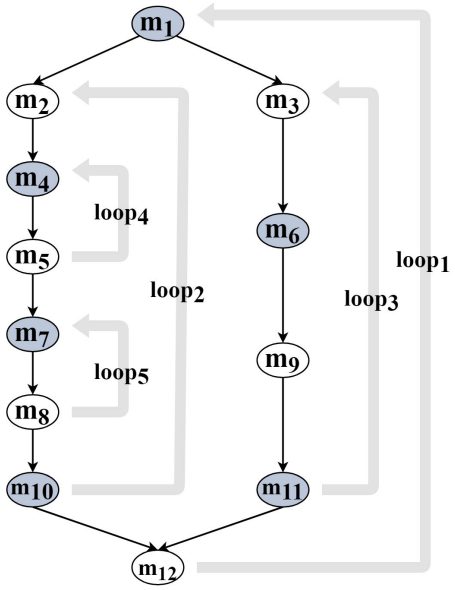


FIGURE 5. A memory block graph.

Definition 4 (Loop Nested Tree): For each loop nest of a task, its Loop Nested Tree is weighted tree $T =< L, S, R >$ where $L = \{l_i$: l_i denotes a loop in the loop nest of the task$\}$, $S = \{S_i$: S_i is the weight of l_i $\}$ and $R = \{(l_i, l_j)$: l_j is immediately nested in $l_i\}$.

Specifically, the weight S_i of a loop l_i in loop nest is an *m-th* tuple formed as $[s_{i_0}, s_{i_1}, \ldots, s_{i_{m-1}}]$, where m is the number of sets of the cache. Each element s_{i_j} of S_i denotes the number memory blocks that are locked at the preheader of l_i and stored in the set j of the cache.

For generating LNT (Loop Nested Tree), we first traverse all the loops of a task to construct a tree structure, and then traverse all the memory blocks of MBG to determine the weight of each node of the constructed tree. Thus, then complexity is O(n+I) of time complexity where n denotes the number of memory blocks in MBG and I denotes the number of loops in a task.

Fig.5 to Fig.6 show an example of a loop nested tree. Consider an MBG in Fig.5, there are 12 memory blocks. Notice that, the MBG is a directed acyclic graph and has no back edges. The gray arrows in Fig.5 are just used to illustrate which loop is the memory blocks belong to. The loops are labeled from $loop_1$ to $loop_5$ respectively. For ease of descriptions, we make the following assumptions:

- The I-cache considered in this example has only 2 sets and each set contains 2 cache lines.
- $loop_2$ and $loop_3$ are the inner loops of $loop_1$.
- $loop_4$ and $loop_5$ are the inner loops of $loop_2$.
- Each memory block is mapped to exactly one cache line and the memory blocks with the same color means they are mapped to the same cache set.

Take the locking decisions shown in the charts near the nodes of the LNT in Fig.6 for example where each column of the chart represents a cache set and each row represents

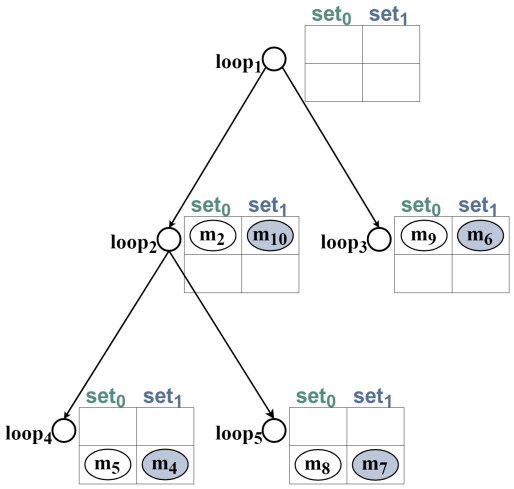


FIGURE 6. The loop nested tree.

a cache line. As can be seen from the figure, the weight of node $loop_2$ of the LNT is [1, 1] which means 1 memory block is locked into set_0 of the I-cache at the preheader of $loop_2$ and 1 memory block is locked into set_1 of the I-cache at the preheader of $loop_2$. Since the life ranges of $loop_4$ and $loop_5$ do not overlap, m_4, m_5 in $loop_4$ and m_7, m_8 in $loop_5$ can share the same space of the I-cache. Thus, the weight of $loop_4$ and $loop_5$ are both [1, 1]. By using loop nested tree, we can calculate the total number of cache lines needed to store the locked memory blocks for each set of the I-cache.

To achieve the dynamic cache locking, we insert a *call_lock* instruction at the preheader of each loop for loading and locking selected memory blocks of corresponding loops. After the life range of a loop ends, the locked memory blocks of this loop can be unlocked and the cache lines storing these memory blocks are able to reuse. In order not to disrupt the memory address mapping of the instructions, the *call_lock* instruction is designed as a procedure-call instruction which calls the corresponding loading and locking procedures placed at the end of the code of the task. For each loop, if there are memory blocks to be locked, we insert a single *call_lock* instruction at its preheader, otherwise, we insert a *nop* instruction. For loading and locking procedures, the original cache management policy is disabled and we introduce a special instruction *lock*. When a cache line is already locked, *lock* instruction replaces the old content with the new one. Thus, no *unlock* instruction is needed.

Given a task, our approach is performed to determine the locked cache contents of the task before running it. We first use static analysis tool, named Chronos [28], to analyze the code of a task and construct corresponding CFG. Then, our cache locking approach can be performed to determine locked cache contents of the task.

IV. OUR APPROACH

In this section, we describe our ILP-based dynamic I-cache locking approach. The objective of our approach is to select a set of memory blocks as locked cache contents and determine locking point for each selected memory block such that the WCET of a single task is minimized. The details are shown as follows.

Our approach selects the appropriate content lock for a task to minimize the WCET, and the framework is listed as follows:

- Get the CFG and the number of cycles of the basic blocks by using Chronos [28];
- Generate the DAG from the CFG by removing the back edges and modifying the weight of the nodes. Then, generate the MBG from the DAG by decomposing the nodes of the DAG into serval memory blocks;
- Generate the loop nodes by analysing the back edges of the CFG and the number of cycles of the basic blocks. Then, construct the LNT with the loop nodes.
- Construct the ILP model. Construct the set capacity constraints by summing the weight of the loop nodes of the LNT by layer. Construct the execution time constraints by summing the weight of the memory blocks of the MBG by layer. Set minimizing the WCET as the objective function.
- Get the locking decision of each memory block by solving the ILP model;
- Calculate the WCET of the task.

To facilitate descriptions, we introduce the following notations in Table-1.

TABLE 1. Notations.

Notation	Definition
S_{cache}	the entire size of the I-cache
m	the number of cache set of the I-cache
l_{cache}	the cache line size of the I-cache
c	time for loading and locking one memory block into the I-cache
X_i	the locking state tuple of memory block m_i
EX_i	the execution time of memory block m_i
LP_i	the execution time from memory block m_{start} to m_i
I_j	the iteration times of preheader P_j

A. ILP MODELLING

Give an MBG of a task, our approach only consider the memory blocks within a certain loop as locked candidates for the simple reason that a memory block which is not in any loop executes only once and has no benefit to be locked.

As we mentioned in Section III, we insert a *call_lock* instruction at each preheader of a loop. Thus, a memory block of a loop nest can be locked at the preheader of its own loop or the preheader of outer loops of the loop nest. Therefore, for each memory block m_i in a loop nest, we design a *k-th* tuple $X_i = [x_{i,1}, x_{i,2}, \ldots, x_{i,k}]$ to determine the locking state and locking point of the memory block m_i. We create a set K_i which stores all the potential locking points of m_i, and $k = |K_i|$ is the number of the potential locking points of memory block m_i. Consider the example shown in Fig.3, m_3 can be locked at $preheader_3$ or $preheader_1$. Thus $K_3 = \{perheader_3, perheader_1\}$.

Each element $x_{i,j}$ of X_i is a binary decision variable to determine whether memory block m_i is locked at preheader

p_j or not. $x_{i,j}$ is formulated as follow.

$$x_{i,j} = \begin{cases} 1, & \text{if } m_i \text{ is decided to be locked at } p_j \\ 0, & \text{otherwise} \end{cases} \quad (1)$$

1) MEMORY BLOCK LOCKING STATE CONSTRAINTS

For a candidate memory block in loops, it can either be locked at only one preheader or not be locked at any preheader. Therefore, the k-th tuple $X_i = [x_{i,1}, x_{i,2}, \ldots, x_{i,k}]$ of a memory block m_i has the following constraints.

$$\sum_{j=1}^{k} x_{i,j} \leq 1 \quad (2)$$

2) SET CAPACITY CONSTRAINTS

The cache considered in this paper is an n-way set associative cache with m sets. The locking decisions of memory blocks in different sets are independent of each other. Therefore, we consider the capacity of different sets separately.

Given a loop nested tree T_q, for each loop l_j in T_q, we create a set $NL_j^y = \{m_i :$ memory block m_i is mapped to set y of the cache and the potential locking point set K_i of m_i contains preheader p_j of loop $l_j\}$. The weight $S_j = [s_{j_0}, s_{j_1}, \ldots, s_{j_{m-1}}]$ of each loop l_j in T_q is calculated as follows:

$$s_{j_y} = \sum_{m_q \in NL_j^y} x_{q,j} (y = 0, 1, \ldots, m-1) \quad (3)$$

For each loop nested tree T_q of the task, we construct the following constraints considering each set y ($y = 0, 1, \ldots, m-1$) of the I-cache separately.

- If a loop L_j is a leaf node in the T_q:

$$C_j^y = s_{j_y} \quad (4)$$

- Otherwise, we have the following constraints:

$$C_j^y = s_{j_y} + \max\{C_g^y : l_g \text{ is a child of } l_j \text{ in } T_q\} \quad (5)$$

For the root node l_{root} of the T_q, C_{root}^y denotes the entire number of cache lines of set y of the I-cache needed to store the locked memory blocks of T_i. Thus, there is an additional constraint for the root node l_{root} of T_q as follow.

$$C_{root}^y \leq \frac{S_{cache}}{l_{cache} * m} \quad (6)$$

Actually, $\frac{S_{cache}}{l_{cache} * m}$ is the associativity of the I-cache.

The time complexity of generating set capacity constraints is $O(n+e)$ where n denotes the number of nodes in LNT and e denotes the number of connections between nodes.

3) EXECUTION TIME CONSTRAINTS

Given the constructed memory block graph G' of a task, the weight w'_i of memory block m_i of G' is the total execution time of m_i without using the I-cache. If m_i is a memory block which does not belong to any loops, its wight remains w'_i, which is a constant. Otherwise, we create the following constraint to calculate the execution time EX_i of each memory block m_i.

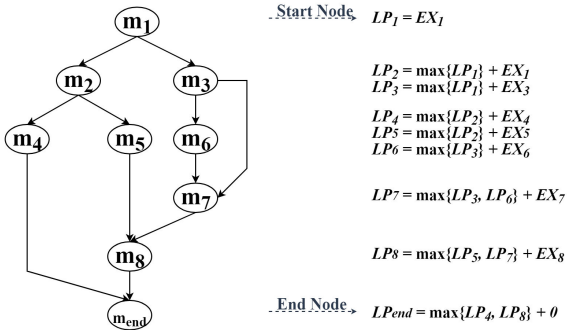


FIGURE 7. Construction of execution time constraints.

$$EX_i = w'_i - \sum_{j=1}^{k} x_{i,j} * (w'_i - w'_{i_{lock}}) \quad (7)$$

$w'_{i_{lock}}$ is the total execution time of the memory block m_i when it is locked into the I-cache. If m_i is locked into the I-cache, $\sum_{j=1}^{k} x_{i,j} = 1$, and its execution time is reduced to $w'_{i_{lock}}$. Otherwise, EX_i remains w'_i.

In the memory block graph G', There is only one start node m_{start} and may be multiple end nodes. For the convenience of the calculation, we add a dummy end node m_{end} and directed edges from previous end nodes to m_{end}. The weight of m_{end} is 0. Based on the memory block graph G', we model the WCET of the task as follows.

- We define a variable LP_i as the longest path length from m_{start} to m_i. Obviously, $LP_{start} = EX'_{start}$.
- For each memory block m_i, we construct constraints as follow:

$$LP_i = EX_i + \max\{LP_j : m_j \text{ is a parent of } m_i\} \quad (8)$$

Fig.7 shows an example of construction of execution time constraints. Firstly, execution time of each memory block is calculated by (8) denoted by EX_i. Then, we model the longest path length of the given memory block graph from start node m_1 to end node m_{end}. Finally, LP_{end} represents the longest path length of the memory block graph.

The time complexity of generating execution time constraints is $O(n + e)$ where n denotes the number of nodes in MBG and e denotes the number of edges.

4) OBJECTIVE FUNCTION

We top-down construct a set of execution time constraints to model the longest path length of G' which denotes the WCET of the task. Thus, LP_{end} of the end node of G' represents the WCET of the task. Loading and locking a memory block into the I-cache will cause extra time cost. We create a set B to store all the locking candidates of memory blocks. The objective of our approach is to minimize the WCET of a task. Thus, we have the following objective function.

$$\min \quad LP_{end} + \sum_{m_i \in B} \sum_{j=1}^{k} (x_{i,j} * I_j * c) \quad (9)$$

TABLE 2. Benchmarks.

Benchmark	Code Size (Bytes)	Description
adpcm	26852	Adaptive pulse code modulation algorithm.
bs	4248	Binary search for the array of 15 integer elements.
cnt	2880	Counts non-negative numbers in a matrix.
crc	5168	Cyclic redundancy check computation on 40 bytes of data.
expint	4288	Series expansion for computing an exponential integral function.
fibcall	3499	Simple iterative Fibonacci calculation, used to calculate fib (30).
lcdnum	1678	Read ten values, output half to LCD.
matmult	3737	Matrix multiplication of two 20x20 matrices.
nsichneu	118351	Simulate an extended Petri Net.
sqrt	3567	Square root function implemented by Taylor.
ud	6000	Calculation of matrixes.

The second term of (9) denotes the time cost on loading and locking all the selected memory blocks, where I_j is the total number of iterations of preheader P_j and c is the time cost by loading and locking one memory block into the I-cache.

The time complexity of solving ILP can't be calculated directly. The typical ILP solve method, named Interior Point Method, has the complexity of $O(n^4 * L)$ where n is the number of variables and L is the number of constraints.

V. EXPERIMENTS AND RESULTS

We have implemented our approaches and compared them with the longest path based dynamic instruction cache locking approach proposed in [24] and the min-cut based dynamic cache-locking approach proposed in [17]. We use Chronos [28] to estimate the WCET of each task shown in the table [27] by using compared approaches and our approach.

A. SETUP AND PREPROCESS

The benchmarks are taken from the MRTC benchmark suite [27]. Each benchmark is source code written in C language. All the benchmarks are consisted of basic programming structures including branching and iterations, don't include third party libraries. For example, the benchmark *bs* is the implementation of binary search among 15 integer elements. The detailed descriptions are included in Table-2.

We use four different type of cache sizes 256B, 512B, 1KB and 2KB; two different associativities 2 and 4; two different cache line size 32B and 64B. Although the CPU of modern embedded systems usually contains bigger cache sizes, we use relatively smaller cache size to reflect the effect of our approach because of the small size of the benchmarks.

By using cache locking algorithm, the worst case execution time (WCET) of a task will be reduced. Chronos provide the number of clock cycles of WCET of each task. Comparing with other approaches, our approach achieves more WCET reduction. For each task in the benchmark suit, the improvement is calculated by the equation in (10).

$$Improvement = \frac{W' - W}{W'} \tag{10}$$

W is the WCET of a task by using our approach, and W' denotes the WCET of a task by using another approach. Notice that, in our experiments, WCET is represented by the number of clock cycles.

We make some processes to the benchmarks before the experiments. First, because there are too few memory blocks in loops in the tasks of 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. This operation increases the number of memory blocks that can be locked to simulate tasks with more complex structures. Then, we use static analysis tool, named Chronos [28], to analyze the code of a task and construct corresponding CFG. Chronos divides the task into several independent basic blocks and compiles the code in each basic block into fixed length assembly instructions. Chronos also estimates the number of cycle times of each basic blocks. The loop information can be obtained by analyzing the cycle times and reverse edges of the CFG. The fixed length assembly instructions are used to generate the memory block graph.

After the memory block graph and loop nested tree are generated, we construct the ILP model described in IV. We use Lingo global solver as the ILP solver and run the programs for preforming all experiments on an Intel i7-6700 CPU with 3.4GHz and 16GB memory. The maximum running time of our approach is 17.54 seconds for the benchmark *nsichneu*, which has largest code size. Notice that most of the running time is spend on constructing constraints of the ILP model, such as generating and traversing the MBG. Solving the ILP model takes up approximately 20% of the total running time. Finally, the locked cache contents can be determined by solving ILP model and Chronos is used to estimate the WCET of the benchmark after locking the selected cache contents. All experimental results are shown from Fig.8 through Fig.11, where each horizontal axis denotes benchmarks and each vertical axis shows the improvements of our approach.

B. OUR APPROACH VERSUS OTHER DYNAMIC CACHE LOCKING APPROACHES

Compared to the longest path based approach, the experimental results show that our approach achieves the average improvements of 6.2%, 13.9%, 16.5% and 12.9% for the 256B, 512B, 1KB and 2KB 2-way set-associative caches with a cache line size of 32B, respectively. For the 256B, 512B, 1KB and 2KB 4-way set-associative caches with a cache line size of 32B, the average improvements are 6.6%, 14.9%, 17.2% and 13.4% respectively. For the 256B, 512B, 1KB and 2KB 2-way set-associative caches with a cache line size of 64B, the average improvements are 5.1%, 12.7%, 14.7% and 11.9% respectively.

The longest path based approach [24] divides the task into serval regions. Then, for each region, the approach iteratively selects locking contents with biggest benefit in the longest path of the control flow graph, locks the content to the nearest loop lock point until all locked content has a negative locking benefit or the cache is full. The locking instructions are placed outside of the outermost loop of the region. Because the life

TABLE 3. WCET improvements of our approach versus longest path based approach (%).

Cache Scheme			Tasks											
Capacity	Line Size	Sets	adpcm	bs	cnt	crc	expint	fibcall	lcdnum	matmult	nsichneu	sqrt	ud	average
256B	32B	2	9.66	7.32	10.34	22.47	3.88	0.08	6.05	0.16	0.01	0.22	7.45	6.15
512B	32B	2	25.37	7.62	20.36	31.42	12.13	9.57	10.19	15.32	0.02	5.57	15.07	13.88
1KB	32B	2	35.02	14.51	23.76	31.68	14.72	2.56	4.40	19.67	0.05	10.07	24.86	16.48
2KB	32B	2	17.52	15.12	12.86	31.6	12.56	3.97	2.78	13.82	0.10	8.96	22.7	12.91
256B	64B	2	9.66	7.32	10.38	22.45	4.88	2.48	7.38	0.16	0.01	0.22	7.76	6.61
512B	64B	2	25.37	7.62	25.69	31.53	14.07	12.46	10.93	15.32	0.02	5.57	15.66	14.93
1KB	64B	2	35.02	15.13	25.77	31.66	14.72	5.73	4.91	21.01	0.05	10.07	25.35	17.22
2KB	64B	2	18.11	15.12	13.02	31.6	12.56	4.57	4.33	13.82	0.10	9.56	24.67	13.4
256B	32B	4	8.79	5.24	10.25	20.85	1.66	0.07	3.15	0.16	0.01	0.14	5.97	5.12
512B	32B	4	22.29	4.42	19.93	29.39	12.05	7.63	10.07	13.62	0.01	5.16	14.58	12.65
1KB	32B	4	33.8	11.1	22.17	31.91	12.75	1.73	2.00	15.93	0.03	8.37	22.12	14.72
2KB	32B	4	17.04	14.51	11.00	30.95	12.75	2.34	1.69	12.06	0.05	8.00	20.12	11.87
256B	64B	4	8.79	5.29	10.24	22.08	1.66	0.08	5.25	0.16	0.01	0.14	5.97	5.42
512B	64B	4	23.22	4.87	23.59	31.08	14.07	11.2	10.17	14.77	0.01	5.16	14.71	13.9
1KB	64B	4	33.8	11.33	24.6	31.9	12.93	4.21	2.04	20.53	0.03	9.45	25.35	16.02
2KB	64B	4	17.26	14.51	12.16	30.95	10.05	2.44	1.69	12.06	0.05	9.22	25.00	12.31

TABLE 4. WCET improvements of our approach versus min-cut based approach (%).

Cache Scheme			Tasks											
Capacity	Line Size	Sets	adpcm	bs	cnt	crc	expint	fibcall	lcdnum	matmult	nsichneu	sqrt	ud	average
256B	32B	2	6.59	3.55	10.08	0.60	2.95	0.00	0.04	0.00	0.00	0.00	2.57	2.40
512B	32B	2	15.49	3.70	8.71	20.45	10.25	6.90	4.26	0.16	0.02	0.48	12.41	7.53
1KB	32B	2	23.36	4.14	12.63	18.67	10.87	1.78	0.78	0.17	0.01	7.94	15.17	8.68
2KB	32B	2	13.57	4.02	7.29	12.22	9.40	1.66	0.06	0.01	0.00	7.59	14.82	6.42
256B	64B	2	7.25	3.55	10.29	0.60	2.92	0.00	0.05	0.00	0.00	0.00	2.22	2.44
512B	64B	2	19.76	3.70	11.49	22.67	10.25	7.69	5.65	0.16	0.01	0.50	12.90	8.62
1KB	64B	2	23.36	4.41	12.63	20.45	11.64	4.67	1.79	0.17	0.03	7.94	15.66	9.34
2KB	64B	2	15.71	4.02	9.93	17.92	9.44	3.47	0.57	0.01	0.02	7.59	14.72	7.58
256B	32B	4	5.75	2.30	9.98	0.19	0.26	0.00	0.00	0.00	0.00	0.00	1.90	1.85
512B	32B	4	13.87	3.54	8.71	15.52	9.23	5.11	2.67	0.16	0.02	0.12	12.41	6.49
1KB	32B	4	21.23	3.43	12.06	11.28	9.23	0.94	0.78	0.16	0.01	6.14	13.09	7.12
2KB	32B	4	12.66	2.03	6.73	10.74	9.20	0.71	0.00	0.02	0.00	7.41	12.14	5.60
256B	64B	4	4.83	3.45	9.98	0.00	0.26	0.00	0.03	0.00	0.00	0.00	1.90	1.86
512B	64B	4	18.94	3.64	10.34	17.95	9.23	6.88	4.09	0.16	0.01	0.12	12.28	7.60
1KB	64B	4	21.23	4.06	12.06	11.28	9.59	1.74	1.18	0.16	0.02	6.14	14.22	7.42
2KB	64B	4	13.38	2.03	5.29	11.33	9.20	1.56	0.03	0.02	0.01	7.41	13.15	5.76

ranges of the regions do not overlap, the cache space can be reused.

The min-cut based approach [17] reduces the WCET of the task by locking the content in the min cut point set of the control flow graph. For a non-nested loop, the approach iteratively find the memory blocks that located on the min cut points and lock them outside of the loop sequentially, until the benefit brought by locking is negative or the cache is full. For a nested loop, the approach first selects a set of memory blocks as the locking candidates using min cut algorithm. Then, the approach selects the best locking points for each candidate.

Compared to the min-cut based approach, the experimental results show that our approach achieves the average improvements of 2.4%, 7.5%, 8.7% and 6.4% for the 256B, 512B, 1KB and 2KB 2-way set-associative caches with a cache line size of 32B, respectively. For the 256B, 512B, 1KB and 2KB 4-way set-associative caches with a cache line size of 32B, the average improvements are 2.5%, 8.6%, 9.3% and 7.6% respectively. For the 256B, 512B, 1KB and 2KB 2-way set-associative caches with a cache line size of 64B, the average improvements are 1.9%, 6.5%, 7.1% and 5.6% respectively.

The approach in [17] and [24] both select the locking content first, and select the locking point for each determined locking content. In this case, when a locked content is locked, the longest path of the task could no longer be the most valuable choose, which making the overall locking decision can not reach the optimal. As for our approach, for each lockable memory block, it considers if the memory block is worthy to be locked and where to be locked concurrently, resulting in more efficient lock decisions and higher cache utilization.

Our approach has different WCET improvement under different setups. In terms of cache capacity, our approach achieves less improvements when the cache capacity is relatively small such as 256*B*. The main reason is that, only several memory blocks can be selected as locked contents and these three approaches provide similar locking strategies. When the cache size increases, the improvements of our approach increase compared with the other two approaches for the simple reason that more memory blocks can be locked

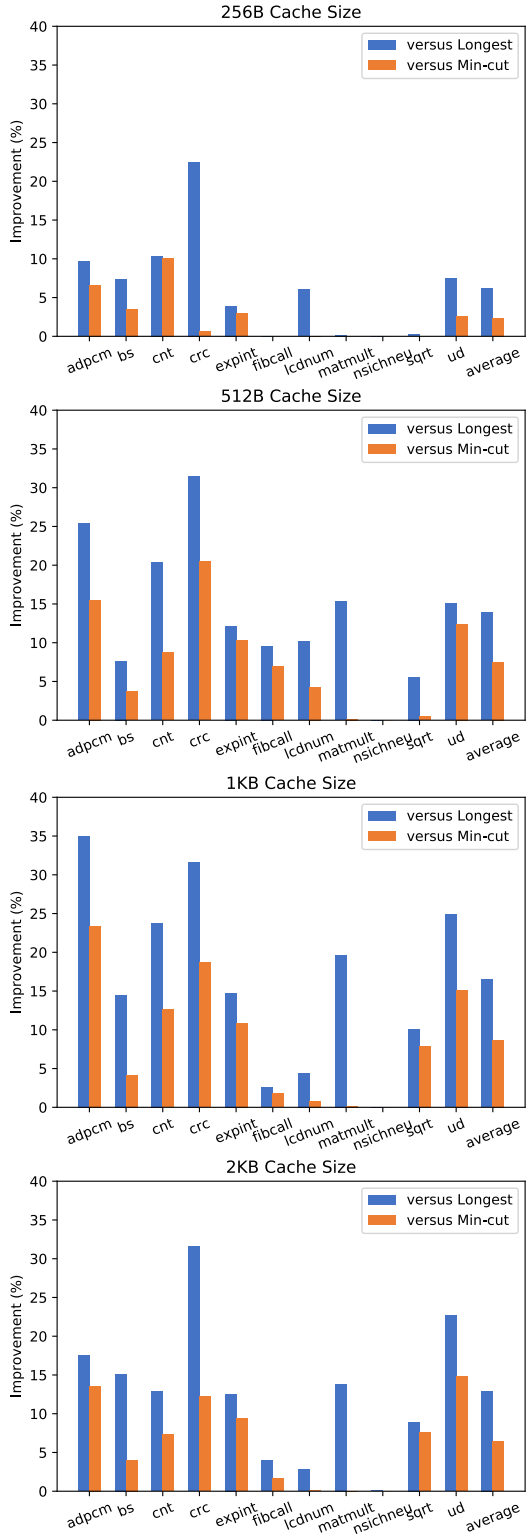


FIGURE 8. WCET improvements of our approach (2-way set associative caches with 32B cache lines).

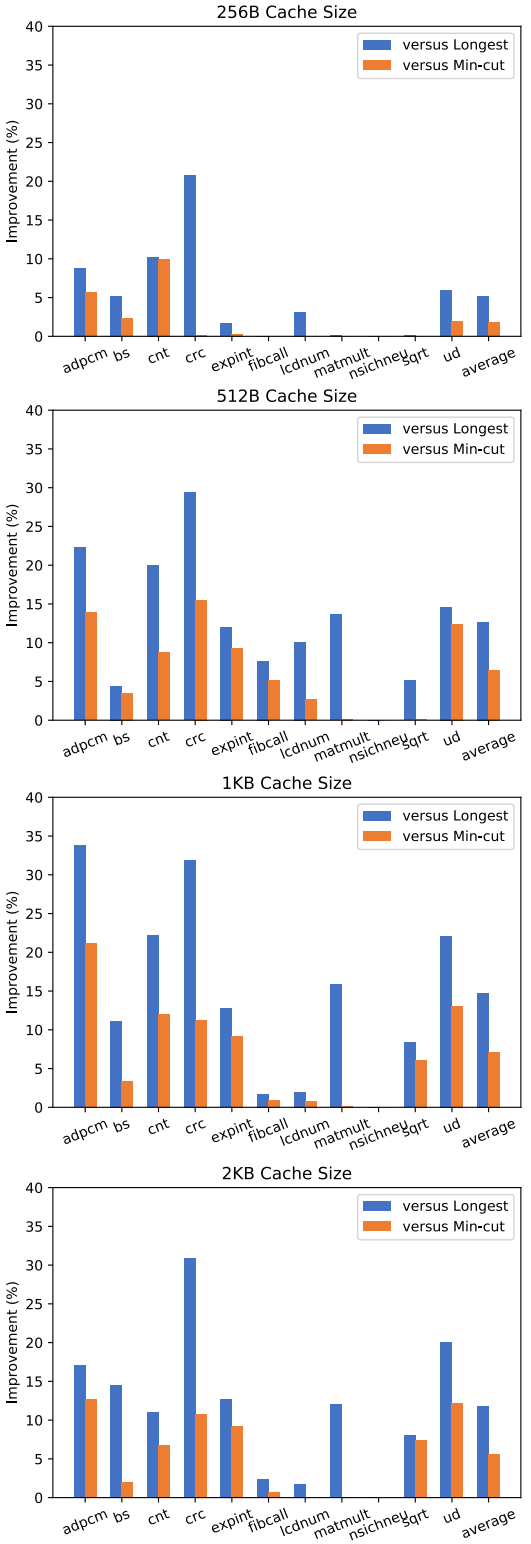


FIGURE 9. WCET improvements of our approach (2-way set associative caches with 64B cache lines).

and our approach provides a better solution of the selection of the locked contents. Besides, our approach also provides the most appropriate determination of the locking points for selected memory blocks, leading to further reduction of the WCET. However, when cache space continues growing, the improvement of our approach tends to dwindle. This is because most of the memory blocks which are valuable for locking have been chosen and continue to lock new memory

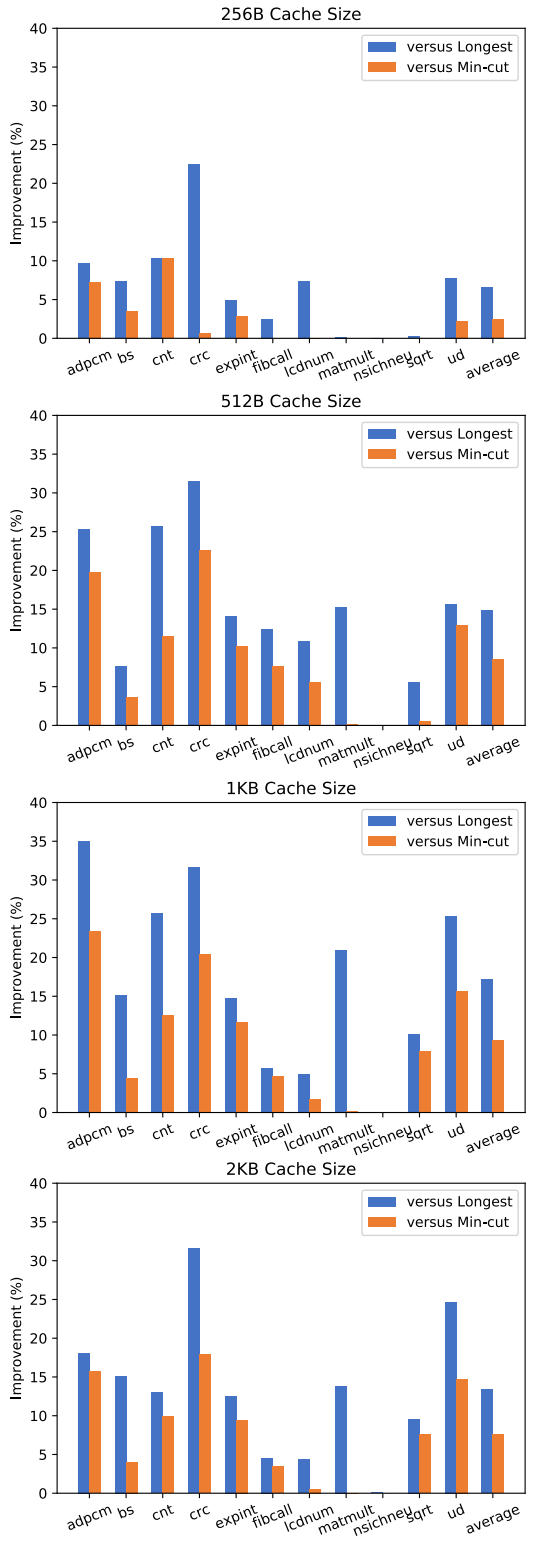


FIGURE 10. WCET improvements of our approach (4-way set associative caches with 32B cache lines).

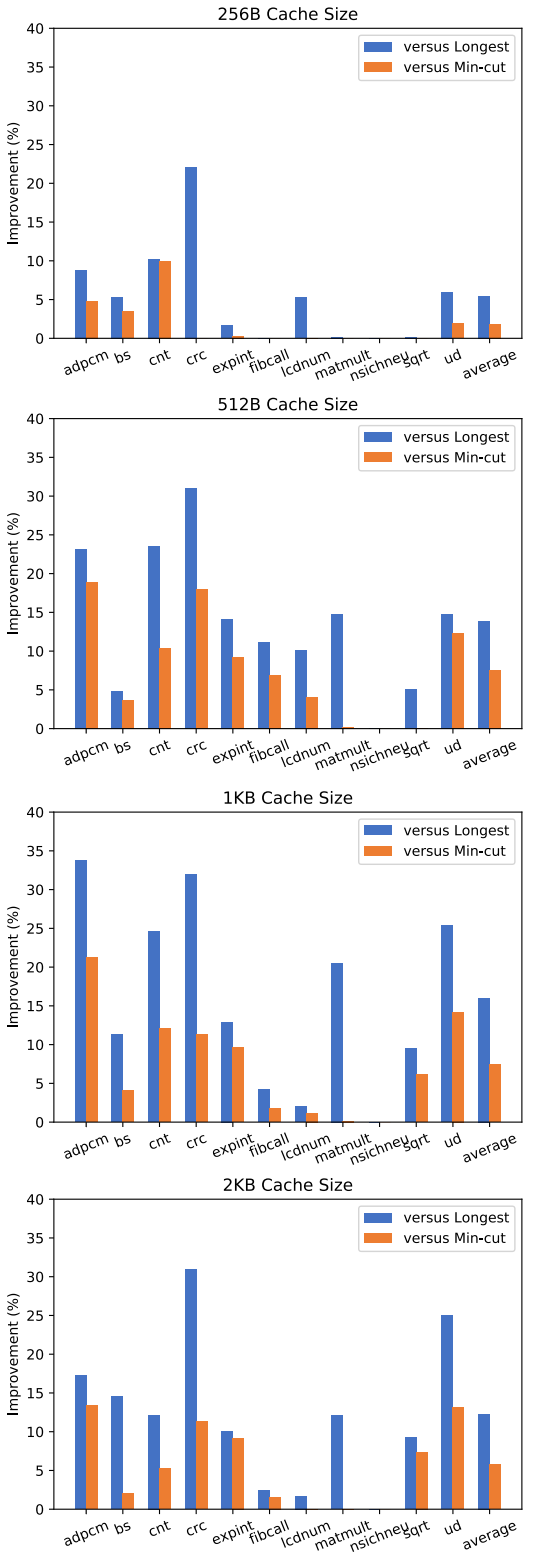


FIGURE 11. WCET improvements of our approach (4-way set associative caches with 64B cache lines).

blocks brings nearly no reduction the WCET. Still, our approach have improvement compared with the other two approaches because the locking point decisions are always better.

For caches with fixed cache size and set associativities, caches with 32B line size performs slightly better than the caches with 64B line size. Since the locking unit of these three approaches is a cache line, smaller cache line size means more

locked candidates, Our approach can achieve a better selection of locked candidates. However, more memory blocks also means more variables in ILP model, leading higher time cost on solving the ILP model.

With the fixed cache size and cache line size, 4-way set associative caches performs slightly better than 2-way set associative cache. For the caches with the same line size and total size, associativity increase means the number of sets reduce. A smaller number of sets can avoid the situation like one set is short of space while the other set is not full.

Compared with the other two approaches, our approach performs much better for the benchmarks which contain deeply nested loops or complex branches, such as *crc*, *expint* and *ud*. The longest path based approach do not optimize the selection of the locking points, leading to more time cost on loading and locking selected memory blocks. Especially for the benchmarks with deeply nested loop, the longest path based approach shows the worst performance. The min-cut based approach iteratively selects a minimum set of cut of the CFG as locked contents, so it cannot efficiently handle the benchmarks with complex branches.

For benchmark *nsichneu*, our approach has basically no improvement compared with the other two approaches. From the source file and generated CFG of this task, we can see that this task has only one cycle, and the structures in the loop are basically sequential structure and single branch structure. Because there is only one cycle, the locking points selected by the three approaches are the same, and there are only some differences in the selection of memory block to be locked, which leads to a slight difference in the final results of the three approaches. For benchmark *matmult*, our approach obtained slight improvement compared with min-cut based approach. This is because the task has a simple nested loop structure, and there are no branches in the loop. When the cache size is small, the locking contents selected by the two approaches are the same, that is, the memory blocks in the innermost loop. With the increase of cache size, our approach can select the memory blocks which can bring the largest WCET improvement, while locking the cut point selected by min-cut based approach may not bring the maximum WCET improvement. However, the min-cut based approach can select the most appropriate locking point of the memory blocks by the ILP algorithm, so our approach has limited improvement.

No promotion in some benchmarks compared with other approaches does not mean that our algorithm has limitations. In the real-time embedded system or mobile edge computing server, the tasks often have complex nested loops and branches,and our approach will perform better.

VI. CONCLUSION

In this article, we investigate the problem of selecting instructions of a task as locked cache contents to minimize the WCET of the task and propose an ILP-based dynamic cache locking approach. For each memory block, our approach considers if the memory block is worthy to be locked and where to be locked concurrently, resulting in more efficient lock decisions and higher cache utilization. The experimental result show that compared with the longest path based dynamic approach, our approach achieves the average improvements of 5.8%, 13.8%, 16.1% and 12.6% for the 256B, 512B, 1KB and 2KB caches, respectively. Compared to the min-cut based dynamic approach, our approach achieves the average improvements of 2.2%, 7.6%, 8.2% and 6.4% for the 256B, 512B, 1KB and 2KB caches, respectively.

A typical embedded system generally consists of a set of tasks. Our cache locking approach can be easily applied to multi-task embedded systems if no timing constraints exist. Timing constrains means that each task has its own release time and deadline and a task must execute after release time and finish before deadline. For a set of tasks without timing constraints, we first select locked cache contents of each single task by using our cache locking approach, such that the WCET of each task is minimized. Then, all the tasks are sequentially executed. Finally, the total worst-case execution time of the multi-task system is minimized. Unfortunately, tasks are often subject to timing constraints, and a feasible schedule is needed to be constructed for satisfying all the timing constraints. Our cache locking approach cannot easily extend to multi-task embedded systems.

In the future, we will investigate the problem of integrating task scheduling and cache locking for multiprocessor-based embedded systems with two levels of caches. This problem is more challenging. For WCET calculation of a task, a basic requirement is to know how much cache space is assigned to the task for locking its selected contents. The cache space assignment problem needs to consider the lifetimes of tasks. Given a schedule, the lifetime of a task is an interval where the start point is the start time of the task and the end point is the finish time of the task in the schedule. For any two tasks, if their lifetimes overlap, they are executed in parallel and cannot share the same space in caches. Otherwise, they can share any space in caches. In order to improve the unitization of the cache space, a good cache space assignment approach is needed to determine the size of cache space assigned to each task. For two tasks with non-overlapping lifetimes, named t_1 and t_2, the cache space assigned to t_1 can be re-assigned to t_2 when the lifetime of t_1 expire. After determining the size of cache space assigned to each task, the WCET of the tasks can be calculated. Finally, a feasible schedule can be constructed.

REFERENCES

[1] K. Anand and R. Barua, "Instruction cache locking inside a binary rewriter," in *Proc. Int. Conf. Compil., Archit., Synth. Embedded Syst. (CASES)*, 2009, pp. 185–194.

[2] Y. Liang and T. Mitra, "Instruction cache locking using temporal reuse profile," in *Proc. 47th Design Autom. Conf. (DAC)*, 2010, pp. 344–349.

[3] H. Ding, Y. Liang, and T. Mitra, "WCET-centric partial instruction cache locking," in *Proc. 49th Annu. Design Autom. Conf. (DAC)*, 2012, pp. 412–420.

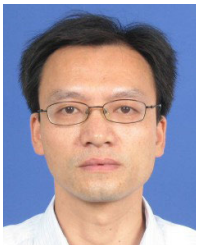
[4] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proc. 23rd IEEE Real-Time Syst. Symp. (RTSS)*, Dec. 2002, pp. 114–123.

[5] K. Qiu, M. Zhao, C. J. Xue, and A. Orailoglu, "Branch prediction-directed dynamic instruction cache locking for embedded systems," *ACM Trans. Embedded Comput. Syst.*, vol. 13, no. 5s, p. 156, 2014.
[6] H. Ding, Y. Liang, and T. Mitra, "WCET-centric dynamic instruction cache locking," in *Proc. Design, Autom. Test Eur. Conf. Exhib. (DATE)*, 2014, p. 27.
[7] W. Zheng and H. Wu, "WCET: Aware dynamic instruction cache locking," in *Proc. Conf. Lang., Compil. Tools Embedded Syst.*, 2014, pp. 53–62.
[8] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst. (SIGMETRICS)*, vol. 31, 2003, pp. 272–282.
[9] M. Dohan and M. O. Agyeman, "A study of cache management mechanisms for real-time embedded systems," in *Proc. 2nd Int. Symp. Comput. Sci. Intell. Control (ISCSIC)*, 2018, pp. 1–5.
[10] A. T. A. Dugo, J.-B. Lefoul, F. G. De Magalhaes, D. Assal, and G. Nicolescu, "Cache locking content selection algorithms for ARINC-653 compliant RTOS," *ACM Trans. Embedded Comput. Syst.*, vol. 18, no. 5s, pp. 1–20, Oct. 2019.
[11] V. Kern, "Optimizing cache performance and predictability," Friedrich-Alexander Univ. Erlangen, Erlangen, Germany, Tech. Rep. [Online]. Available: https://www4.cs.fau.de/Lehre/SS20/PS_KVBK/arbeiten/Cache_Awareness.pdf
[12] T. Adegbija and A. Gordon-Ross, "PhLock: A cache energy saving technique using phase-based cache locking," *IEEE Trans. Very Large Scale Integr. (VLSI) Syst.*, vol. 26, no. 1, pp. 110–121, Jan. 2018.
[13] C. Xue, K. Qiu, W. Zhang, J. Wang, Y. Xu, and M. Zhao, "Data reallocation enabled cache locking for embedded systems," *J. Syst. Archit.*, vol. 77, pp. 3–13, Jun. 2017.
[14] Wan, Qing, Hui Wu, and Jingling Xue, "WCET-aware data selection and allocation for scratchpad memory," *ACM SIGPLAN Notices*, vol. 47, no. 5, pp. 41–50, 2012.
[15] A. Mart Campoy, A. P. Ivars, and J. V. Busquets-Mataix, "Using genetic algorithms in content selection for locking-caches," in *Proc. Int. Symp. Appl. Informat.*, 2001, pp. 271–276.
[16] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix, "Cache contents selection for statically-locked instruction caches: An algorithm comparison," in *Proc. Euro Micro Conf. Real-Time Syst. (ECRTS)*, 2005, pp. 49–56.
[17] W. Zheng, H. Wu, and Q. Yang, "WCET-aware dynamic I-Cache locking for a single task," *ACM Trans. Archit. Code Optim.*, vol. 14, no. 1, pp. 1–26, Apr. 2017.
[18] T. Adegbija and A. Gordon-Ross, "Phase-based cache locking for embedded systems," in *Proc. 25th Ed. Great Lakes Symp. (VLSI GLSVLSI)*, 2015, pp. 115–120.
[19] A. M. Campoy, A. Perles, F. Rodriguez, and J. V. Busquets-Mataix, "Static use of locking caches vs. Dynamic use of locking caches for real-time systems," in *Proc. Can. Conf. Electr. Comput. Eng. Toward Caring Humane Technol. (CCECE)*, vol. 2, 2003, pp. 1283–1286.
[20] C. Lin, N. Gu, and S. Cai, "Cache locking optimization in Java virtual machine," in *Proc. IEEE Conf. Anthology*, Jan. 2013, pp. 1–4.
[21] A. Asaduzzaman, F. N. Sibai, and M. Rani, "Improving cache locking performance of modern embedded systems via the addition of a miss table at the L2 cache level," *J. Syst. Archit.*, vol. 56, nos. 4–6, pp. 151–162, Apr. 2010.
[22] W. Zheng and H. Wu, "Dynamic data-cache locking for minimizing the WCET of a single task," *ACM Trans. Embedded Comput. Syst.*, vol. 16, no. 2, pp. 1–29, Apr. 2017.
[23] K. Anand and R. Barua, "Instruction-cache locking for improving embedded systems performance," *ACM Trans. Embedded Comput. Syst.*, vol. 14, no. 3, pp. 1–25, May 2015.
[24] I. Puaut, "WCET-centric software-controlled instruction caches for hard real-time systems," in *Proc. 18th Euromicro Conf. Real-Time Syst. (ECRTS)*, 2006, p. 10.
[25] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proc. 5th IEEE/ACM Int. Conf. Hardw./Softw. Codesign Syst. Synth. (CODES+ISSS)*, 2007, pp. 143–148.
[26] T. Liu, M. Li, and C. J. Xue, "Instruction cache locking for embedded systems using probability profile," *J. Signal Process. Syst.*, vol. 69, no. 2, pp. 173–188, Nov. 2012.
[27] M. Research Group. *WCET Benchmark Programs*. [Online]. Available: http://www.mrtc.mdh.se/projects/wcet/benchmarks.html
[28] X. Li, Y. Liang, T. Mitra, and A. Roychoudhury, "Chronos: A timing analyzer for embedded software," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 56–67, Dec. 2007. [Online]. Available: http://www.comp.nus.edu.sg/ rpembed/chronos

TINGXU ZHANG (Member, IEEE) received the B.S. degree in information management from Xinjiang Medical University, Urumqi, China, in 2018. He is currently pursuing the M.E. degree in computer science technology with the Tianjin University of Technology, Tianjin, China.

His research interests include the optimization and application of embedded systems and mobile edge computing. Compared with the most advanced theoretical algorithms, he is more willing to optimize and extend the existing algorithms to the application field. During his undergraduate period, he participated in the research on the topic of research and development and signal transmission of small ECG monitoring devices, explored the application of small embedded devices in medical treatment, and won the provincial outstanding student innovation topic award.

WENGUANG ZHENG received the B.S. degree from the University of Electronic Science and Technology of China, Chengdu, Sichuan, China, in 2010, the M.S. degree from the National University of Singapore, in 2011, and the Ph.D. degree from the University of New South Wales, Sydney, NSW, Australia, in 2016. He is currently an Associate Professor with the School of Computer Science and Engineering, Tianjin University of Technology, Tianjin, China. His research interests include computer architecture, embedded systems, and wireless sensor networks. He has published journal and conference papers in these areas, including LCTES, TACO, TECS, and so on.



YINGYUAN XIAO received the Ph.D. degree in computer science from the Huazhong University of Science and Technology, China, in 2005. He was a Visiting Scholar with the School of Computing, National University of Singapore, from 2009 to 2010. He is currently a Professor with the School of Computer Science and Engineering, Tianjin University of Technology, China. His research interests include personalized recommender systems, advanced databases, and data mining. He has published more than 100 journal and conference papers in these areas, including IEEE TRANSACTIONS ON COMPUTERS (TC), IEEE TRANSACTIONS ON PARALLEL AND DISTRIBUTED SYSTEMS (TPDS), *Future Generation Computer Systems*, *Information Processing Letters*, *Journal of Classification*, *Soft Computing*, WWW, DASFAA, DEXA, and so on. He has served as a Program Chair for WAIM 2013 International Workshop (International Workshop on Location-based Query Processing in Mobile Environments) and a Publicity Chair for FSKD2009. He also served as a program committee member for a number of international conferences, including APWeb2011, APSCC2011, IEEE CloudCom2012, and ISI2013.

GUANGPING XU received the B.S. degree in computer science from the Tianjin University of Technology, in 2000, and the M.Sc. and Ph.D. degrees in computer science from Nankai University, China, in 2005 and 2009, respectively. He is currently an Associate Professor with the Tianjin University of Technology. His research interests include distributed storage systems and algorithm optimization.