WCET-Centric Dynamic Instruction Cache Locking

Huping Ding
School of Computing
National University of Singapore
Email: d-huping@comp.nus.edu.sg

Yun Liang
Center for Energy-efficient Computing and Applications
School of EECS, Peking University
Email: ericlyun@pku.edu.cn

Tulika Mitra
School of Computing
National University of Singapore
Email: tulika@comp.nus.edu.sg

***Abstract*—Cache locking is an effective technique to improve timing predictability in real-time systems. In static cache locking, the locked memory blocks remain unchanged throughout the program execution. Thus static locking may not be effective for large programs where multiple memory blocks are competing for few cache lines available for locking. In comparison, dynamic cache locking overcomes cache space limitation through time-multiplexing of locked memory blocks. Prior dynamic locking technique partitions the program into regions and takes independent locking decisions for each region. We propose a flexible loop-based dynamic cache locking approach. We not only select the memory blocks to be locked but also the locking points (e.g., loop level). We judiciously allow memory blocks from the same loop to be locked at different program points for WCET improvement. We design a constraint-based approach that incorporates a global view to decide on the number of locking slots at each loop entry point and then select the memory blocks to be locked for each loop. Experimental evaluation shows that our dynamic cache locking approach achieves substantial improvement of WCET compared to prior techniques.**

I. INTRODUCTION

Caches greatly improve the performance of modern processors by exploiting the temporal and spatial localities. However, caches introduce timing unpredictability in hard real-time systems. In such systems, the worst-case execution time (WCET) is an important metric for schedulability analysis. The WCET of a program is the maximum execution time across all possible inputs for a particular architecture. In the presence of caches, the timing of real-time applications is unpredictable as the behavior of cache access can not be determined statically [21].

In this paper, we focus on the instruction cache. In the past, static cache analysis has been used to model the cache and estimate the WCET [17], [20]. However, static analysis may lead to high WCET overestimation in the presence of complex control flows. For example, when a memory block is not guaranteed to be cache hit under static analysis, it is conservatively assumed to be cache miss. Cache locking is an alternative approach. Memory blocks are locked in the cache via special locking routines, and they cannot be evicted from the cache at runtime by the replacement policies. Many modern processors support cache locking mechanism, e.g., Intel Xscale and ARM 9 series [1]. Cache locking improves timing predictability because all the memory accesses to the locked memory blocks are guaranteed to be cache hits. Moreover, by carefully selecting the memory blocks to lock, cache locking can greatly improve performance [19], [9], [14], [7].

Related Work. Most cache locking techniques aimed at improving the WCET employ static cache locking [9], [15], [18], [7]. Static locking loads and locks the memory blocks at program startup, and the locked content remains unchanged throughout the program execution. Most static locking techniques use full cache locking [9], [15], [18] where the entire cache is locked. However, full locking does not allow the unlocked memory blocks to use the cache and exploit their locality, and thus may introduce negative impact on the overall WCET. Recently, we have introduced partial cache locking to optimize the WCET [7], where only a portion of the cache is locked. Our partial locking mechanism integrates cache locking with cache modeling, which allows us to estimate the WCET of predictable accesses through cache modeling and optimizes the WCET of unpredictable accesses through cache locking. Compared to full cache locking and static analysis, our partial locking technique achieves better results [7], [8].

The drawback of static cache locking can manifest for large programs executing on small caches. As the locked content remains unchanged throughout execution, there is limited scope for optimization. In this context, dynamic instruction cache locking techniques that adjust the locked contents at runtime can further improve the WCET [3], [19], [16], [22]. For these approaches, the basic idea is to partition the program into appropriate regions and use full locking for each region. As the program execution moves from one region to another, the memory blocks for the new region are locked. However, due to rigid partitioning, it does not allow selective locking of different memory blocks from the same region at different program points (see Figure 1). Liu et al. [16] extended the region-based approach and proposed a swapping-based method. However, their technique is applied only to the regions with branching nodes but not the regions with nested loops. Moreover, with nested loops, frequent swapping operations will render it infeasible to lock memory blocks from the outer loop. They also do not consider cache mapping function, which makes cache locking similar to scratchpad memory allocation.

Overview. In this paper, we propose a loop-based dynamic instruction cache locking approach to optimize the WCET. We focus on the loops, in particular nested loops. As the locking routines are usually stored in the non-cacheable memory [1], the locking cost is quite high and needs to be offset through repeated access to the locked memory blocks in the program. This leads to memory blocks within loops as natural candidates for locking. We also lock a memory block at the entry point of a loop and unlock it at the corresponding exit point of the loop. This policy ensures that locking and unlocking costs are incurred before and after the execution of the loop.

Our approach differs from prior techniques along two important dimensions. First, [3], [19], [16] and [22] assume

978-3-9815370-2-4/DATE14/©2014 EDAA

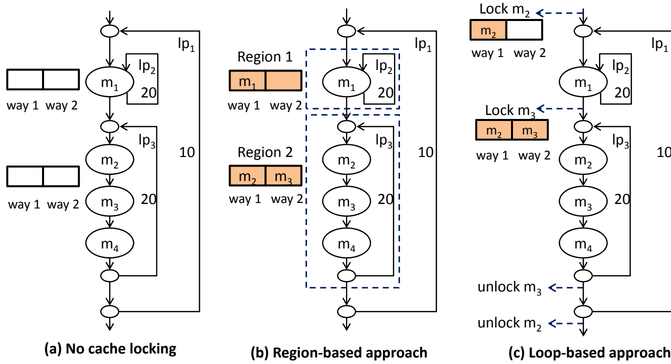

Fig. 1. Motivating example for dynamic cache locking.

full cache locking for each region, while we adopt partial cache locking. More importantly, we carefully select not only the memory blocks that can be locked but the program points where they should be locked. In particular, a memory block m from an inner loop L may be locked either at L or any of its enclosing outer loops. Moreover, memory blocks m and m' from L can potentially be locked at different loop levels. This selective promotion of memory blocks to different loop levels is a key contribution of our approach (see Figure 1(c)).

The challenge is to select the memory blocks and their locking points. We develop a constraint-based approach to first determine the number of memory blocks to be locked for each loop to minimize the WCET. In this process, we exploit the concept of resilience sets to quickly and accurately estimate cost-benefit tradeoff for cache locking. This is followed by a memory block selection phase that identifies the actual memory blocks to be locked for each available locking slot.

Motivating Example. We illustrate the benefit of our loop-based dynamic cache locking approach in Figure 1 and Table I, by comparing with static analysis approach without locking and the region-based approach [3]. For simplicity, we use single-path program here, but our loop-based approach can be applied to the general cases where loops are on different branches. Figure 1(a) shows the original control flow. There are three loops: lp_1, lp_2, lp_3 where lp_2 and lp_3 are nested in lp_1. The numbers on the loop back edges are the corresponding loop bounds. We assume a 2-way set associative cache with LRU replacement policy. The latency of cache hit is 1 cycle and cache miss penalty is 30 cycles. We assume 150 cycles overhead to lock and unlock a memory block because the locking/unlocking routines involve multiple instructions to lock/unlock each memory block and they are kept in the uncacheable region of the program memory [1] (In [1], Xscale uses 4 instructions to lock a memory block, and we assume there is another instruction to unlock it). We also assume that all the memory blocks are mapped to the same cache set.

No Cache Locking: As shown in Figure 1(a), there is no conflict for m_1 in lp_2, while all the other memory blocks conflict with m_1 in lp_1. Thus, m_1 is cache hit inside lp_2, while it is classified as cache miss in lp_1. For m_2, m_3 and m_4, they always conflict inside lp_3. So, each of them incurs 200 misses.

Region-based Approach: The program is partitioned into two regions, as shown in Figure 1(b). The memory blocks with highest execution frequencies are chosen to be locked (m_1 in region 1; m_2 and m_3 in region 2). When the flow enters

TABLE I. WCET ANALYSIS FOR THE MOTIVATING EXAMPLE.

Approaches	Blocks	# of hit	# of miss	Locking frequency	WCET (cycles)	Total WCET (cycles)
No cache locking	m_1	190	10	0	490	18,490
	m_2	0	200	0	6,000	
	m_3	0	200	0	6,000	
	m_4	0	200	0	6,000	
Region -based approach	m_1	200	0	10	1,700	11,100
	m_2	200	0	10	1,700	
	m_3	200	0	10	1,700	
	m_4	0	200	0	6,000	
Loop -based approach	m_1	190	10	0	490	8,540
	m_2	200	0	1	350	
	m_3	200	0	10	1,700	
	m_4	0	200	0	6,000	

a region, the region must load its locked memory blocks. Thus, each locked memory block is loaded and locked 10 times.

Loop-based Approach: We lock m_3 at the entry of lp_3, while m_2 is promoted to be locked at the entry of lp_1, as shown in Figure 1(c). In lp_2, as only m_2 is locked, m_1 can still use the remaining cache line and only suffers 10 cache misses. In lp_3, both m_2 and m_3 are locked. Meanwhile, m_2 is locked only once, while m_3 needs to be locked 10 times.

As shown in Table I, the loop-based approach achieves better WCET compared to both static cache analysis and region-based approach. As the loop-based approach locks m_2 at the outermost loop, its locking cost is substantially reduced. This gain in locking cost could have been offset by the fact that m_1 is locked in region-based approach but not in loop-based approach. Partial cache locking comes to rescue here as m_1 can still benefit from caching and incurs only 10 misses.

II. CACHE MODELING AND LOCKING

The design of a set associative cache involves several parameters: cache line (block) size L, which defines the unit of data transfer between the cache and main memory; number of cache sets K that the cache is divided into; cache associativity A, which determines the number of cache lines in a set. Thus, the capacity of the cache is $L \times K \times A$. We assume LRU (Least Recently Used) cache replacement policy and an uni-processor with only one level of cache.

A. Cache Modeling

Given a memory block m, it can be mapped to only one cache set (m *modulo* K) and will not interfere with the blocks mapped to other cache sets. Thus the cache sets are independent and can be modeled separately. To simplify the discussion and explanation, we will restrict our cache modeling to one cache set. We use M to denote the set of memory blocks mapped to cache set s. In addition, we use $\perp$ to indicate the absence of any memory block in a cache line.

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

Abstract cache state maps cache lines (blocks) to sets of memory blocks. Must analysis, may analysis and persistence analysis [20] are usually employed to compute the abstract cache states. At each program point, must analysis captures the memory blocks that are guaranteed to be present in the cache, may analysis determines the memory blocks that are never in the cache, while persistence analysis identifies the

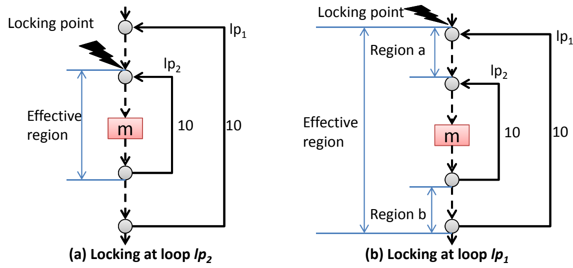

Fig. 2. Effect of different locking positions.

memory blocks that may not be in the cache at the first time but are guaranteed to be in the cache for the later accesses.

*Definition 2 (***Age in Abstract Cache State***):* The age of memory block m in an abstract cache state a is defined as

$$age_m^a = \begin{cases} i & if\ \exists i\ (0 \le i \le A-1)\ \ s.t.\ m \in a[i] \\ A & otherwise \end{cases}$$

*Definition 3 (***Younger/Older Memory Block***):* For two memory blocks m and m' in abstract cache state a, we define m as younger (older) than m' if $age_m^a \le age_{m'}^a$ ($age_m^a > age_{m'}^a$).

B. Cache Locking

In this paper, we consider dynamic instruction cache locking based on partial cache locking. There are two options for partial cache locking: way locking and line locking. Way locking locks all the cache lines in particular cache ways while line locking allows different number of lines to be locked in different ways. We adopt line locking mechanism as it is more flexible and fine-grained. Our approach locks a memory block at the entry of a loop and unlocks it at the corresponding exit of the loop. For simplicity, in the rest of the paper, a memory block m is locked at a loop L implies that m is locked at the entry of L and it is unlocked at the exit of L. We also define L as the effective locking region of m. A memory block can be locked at any loop that contains it. Thus, a memory block in the nested loops may have multiple candidate locking points.

We adopt the trampolines approach proposed in [5] to lock/unlock memory blocks. For each loop, we first leave a dummy NOP instruction at the entry point and at the exit point before we decide on cache locking. If we decide to lock memory blocks at this loop, the NOP instruction at the entry (exit) gets replaced by a call to the locking (unlocking) routine. As a loop may have multiple exits, all these loop exits are handled similarly. For an exit whose target is not the following basic block, a jump instruction is also required to return from unlocking routine. All locking/unlocking routines are placed at the end of the program and stored in non-cacheable memory. Thus, they do not affect the cache contents of the program during execution.

III. Dynamic Cache Locking

Our loop-based approach requires global optimization to select the memory blocks and the corresponding locking points, making the problem challenging. In the example of Figure 2, lp_1 is the outer loop while lp_2 is the inner loop, and their loop bounds are both 10. In Figure 2(a), the locking point is at lp_2, while memory block is locked at lp_1 in Figure 2(b). When we try to lock a memory block m, the locking benefit is the same at both locking points, while the locking costs may

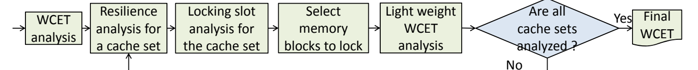

Fig. 3. Framework of dynamic cache locking.

be different. In Figure 2(a), locking m only affects the memory blocks in lp_2 (effective region), but the locking/unlocking routines execute 10 times. In Figure 2(b), effective region is enlarged to lp_1, and the memory blocks in *region a* and *region b* are also affected. However, execution frequency of the locking/unlocking routines is only 1. That is, the locking point affects the locking cost. Thus, apart from the cost-benefit analysis to select the memory blocks for locking, we introduce additional complexity of identifying appropriate locking point for each memory block.

Figure 3 illustrates the flow of our dynamic cache locking approach. First, we perform WCET analysis with abstract interpretation [20] for the entire cache and obtain an initial WCET. Our locking content selection algorithm proceeds for each cache set independently. For each cache set, we perform resilience analysis for the memory blocks mapped to this cache set. The resilience of a memory block m captures the number of memory blocks (excluding itself) that can be locked before the access to m changes from cache hit to cache miss. Based on the resilience analysis, we perform a locking slot analysis for this cache set. That is, we figure out the number of memory blocks that should be locked at each loop, in order to improve the WCET. Then, we select the most profitable memory blocks to fill the locking slots. Later, the abstract cache states for this cache set are re-computed, and the new WCET after locking the cache set is calculated. We call this light weight WCET analysis as the abstract cache states analysis is restricted to the particular cache set being analyzed. When all the cache sets are analyzed, we obtain the final WCET after locking. We detail the dynamic cache locking approach in the following sections.

A. WCET Analysis

First, we perform abstract cache state analysis via abstract interpretation [20]. Three types of analysis are carried out: must analysis, may analysis and persistence analysis. We adopt the multi-level persistence analysis technique [4] to improve the accuracy of WCET estimation. Meanwhile, we use the *Younger Set* approach in [11] to fix the safety issue that may underestimate the WCET in the traditional persistence analysis [20]. Based on the abstract cache states, memory accesses are classified into four categories: Always Hit, Always Miss, Persistent, and Non-Classified. The program WCET is calculated via the implicit path enumeration method [13] with ILP (Integer Linear Programming) formulation. As a by-product of the WCET analysis, we collect the memory blocks along the worst-case path and their corresponding execution frequencies, as well as the abstract cache states at each program point.

B. Resilience Analysis

We define resilience for each memory block [2] as follows.

*Definition 4 (***Resilience***):* The resilience of a memory block m in cache set s at a program point p is the maximum number of older memory blocks that can be locked in s before m is evicted out.

For a memory block m that can be classified as Always Hit or Persistent, we calculate its resilience at the program point p

as follows based on the abstract cache states of must analysis and persistence analysis.

$$res^p_m = \begin{cases} A-1-age^a_m & if\ 0 \le age^a_m \le A-1 \\ -1 & otherwise \end{cases}$$

where a is the abstract cache state at program point p, age^a_m is the corresponding age of m in a, and A is the cache associativity. We use the resilience value of -1 to indicate that a memory block is not in the cache. For a memory block m with non-negative resilience, we also define the set of younger memory blocks of m as ys_m.

$$ys_m = \{m' | res^p_m \ge 0 \wedge age^a_{m'} \le age^a_m\}$$

We collect the younger memory blocks of m, as locking a memory block $m' \in ys_m$ will not affect the age of m.

C. Locking Slot Analysis

The locking slot analysis determines the number of memory blocks that should be locked at each loop level in order to minimize the overall WCET.

We assume there are N loops in the program, $LP = \{lp_1, lp_2, ..., lp_N\}$. For a cache set s, we define the number of locking slots for the loop $lp_i \in LP$ as n_i. We use a function $gain(lp_i, n_i)$ to represent the locking gain on WCET by having n_i locking slots at lp_i. Thus, the total locking gain on WCET for the program in dynamic locking can be defined as follows.

$$\sum_{lp_i \in LP} gain(lp_i, n_i)$$

For the global optimization, we propose an ILP formulation approach to obtain n_i for each $lp_i \in LP$. We first derive the constraints on n_i. Then we approximate $gain(lp_i, n_i)$, the locking gain by having n_i slots at loop lp_i. The objective of the ILP formulation is to maximize $\sum_{lp_i \in LP} gain(lp_i, n_i)$.

1) Constraints in Local Loop: Suppose N_i is the number of memory blocks locked at loop $lp_i \in LP$, when static cache locking is independently applied to lp_i. That is, locking more memory blocks may have negative impact on the locking gain at lp_i. So, in our case, we should make n_i bounded by N_i.

$$n_i \le N_i \tag{1}$$

To obtain N_i for lp_i, we perform static locking cost-benefit analysis in lp_i and iteratively select the most profitable memory blocks to lock, as discussed below.

Based on the resilience analysis, the memory blocks of loop lp_i can be classified into $A+1$ sets: $\{S^{-1}_i, S^0_i, ..., S^{A-1}_i\}$. When $0 \le x \le A-1$, S^x_i indicates the set of memory blocks whose resilience is x in loop lp_i. While S^{-1}_i contains all memory blocks that are classified as Always Miss or Non-Classified in loop lp_i. Clearly, there is locking benefit only when we lock the memory blocks in S^{-1}_i. Thus, for a memory block $m \in S^{-1}_i$, its locking benefit can be defined as follows.

$$benefit_m = (LAT_{miss} - LAT_{hit}) \times freq_m$$

where LAT_{miss} is the access latency for a cache miss, LAT_{hit} is the cache hit latency, and $freq_m$ is the execution frequency of m on the worst-case path. However, locking m also incurs penalty, as the number of free cache lines in cache set s is reduced by 1, which may result in the eviction of memory blocks in S^0_i with resilience value 0.

$$cost'_m = \sum_{m' \in S^0_i \wedge m \notin ys_{m'}} ((LAT_{miss} - LAT_{hit}) \times freq_{m'})$$

where $cost'_m$ represents the locking cost due to free cache space reduction, m' is a memory block with resilience of 0, and $ys_{m'}$ is the set of younger memory blocks of m'. As we have mentioned, when $m \in ys_{m'}$, there is no impact on m' by locking m. Locking m also requires the execution of locking/unlocking routines. We use $cost''_m$ to represent this type of locking cost.

$$cost''_m = PENALTY \times freq_r$$

where $PENALTY$ is a constant indicates the penalty to execute the locking/unlocking routines for one memory block, and $freq_r$ is the total locking frequency of memory block m at lp_i on the worst-case path. Therefore, we obtain the total cost in locking m as follows.

$$cost_m = cost'_m + cost''_m$$

In this case, we can easily obtain the gain of locking m.

$$gain_m = benefit_m - cost_m$$

We perform the analysis for all the memory blocks in S^{-1}_i, and select the memory block with the maximum locking gain to lock. Each time we lock a memory block, we update the resilience sets, as the number of free cache line is reduced by 1 and the resilience values of memory blocks changes.

Let m_k be the kth $(1 \le k \le N_i)$ memory block selected to be locked at lp_i. We record the *individual* locking gain for this memory block at loop lp_i in static locking analysis as $gain^k_i$.

$$gain^k_i = gain_{m_k}$$

This value will be used to approximate the locking gain at lp_i in dynamic cache locking. Note that $gain(lp_i, k) = \sum^k_{t=1} gain^t_i$.

We continue to lock memory blocks with the updated resilience sets until there is no locking gain for all memory blocks or the cache set s is fully locked. In this case, we obtain the value of N_i, the maximum number of memory blocks that can be locked at lp_i when static locking is applied.

2) Accumulated Constraints: In our technique, the memory blocks locked in the outer loops will be brought into their corresponding inner loops. Thus, we also need to bound the accumulated locking slots at each loop in the program.

For $lp_i \in LP$, we use OL_i to indicate the set of outer loops of lp_i, while IL_i represents the set of inner loops of lp_i. Suppose loop $lp_j \in LP$ is an outer loop of lp_i, that is, $lp_j \in OL_i$. We define a loop set $LP_{i,j}$ as the set of loops between outer loop lp_j and inner loop lp_i (inclusive).

$$LP_{i,j} = \{lp_y | y = i \vee y = j \vee (lp_y \in OL_i \wedge lp_y \in IL_j)\}$$

Therefore, the accumulated number of locked memory blocks starting from lp_j to lp_i is as follows.

$$acc_{i,j} = \sum_{lp_y \in LP_{i,j}} n_y$$

We can also bound $acc_{i,j}$ with N_i. However, N_i is too restrictive for $acc_{i,j}$, as N_i only considers the locking benefit of memory blocks in lp_i, while the accumulated memory blocks locked at lp_i can potentially come from the outer loops $LP_{i,j} \setminus \{lp_i\}$. Therefore, an appropriate bound on $acc_{i,j}$ should consider maximum locking benefit and minimum locking cost. That is, we should consider the locking benefit of memory blocks in lp_j while only the locking cost of memory blocks in lp_i is taken into account. We define such bound as $N_{i,j}$, the number of memory blocks that can be locked at lp_j from the

perspective of lp_i in static locking analysis. The computation of $N_{i,j}$ is similar to that of N_i. The main difference is that we choose the memory blocks from the effective region lp_j (S_j^{-1}) for locking, while only the negative impact on lp_i is considered. Thus, we have

$$acc_{i,j} \leq N_{i,j} \tag{2}$$

3) Locking Gain Approximation: Recall that N_i is a bound for n_i. We define a 0-1 binary variable B_i^k to indicate whether the kth ($1 \leq k \leq N_i$) slot is allocated at lp_i in dynamic cache locking. Clearly, until the kth slot is allocated, its subsequent slot cannot be allocated. Thus, we have

$$B_i^k \geq B_i^{k+1}, where \quad 1 \leq k \leq N_i - 1 \tag{3}$$

Obviously, n_i is the summation of B_i^k over N_i.

$$n_i = \sum_{1 \leq k \leq N_i} B_i^k \tag{4}$$

When we lock a memory block in the kth slot at lp_i, we use the locking gain of the kth memory block in computing N_i to approximate its locking gain. When the kth slot is allocated, the locking gain is $gain_i^k$, otherwise we have no locking gain. So the locking gain at the kth slot in dynamic locking is

$$gain_i^k \times B_i^k$$

The total gain function $gain(lp_i, n_i)$ can be approximated as

$$\sum_{1 \leq k \leq N_i} gain_i^k \times B_i^k \tag{5}$$

Equations 1-5 are the constraints in the ILP formulation, and the objective is to maximize $\sum_{lp_i \in LP} gain(lp_i, n_i)$. The outcome of the ILP formulation is the n_i for each $lp_i \in LP$.

D. Memory Block Selection

Obviously, the locking point of a memory block affects the locking point of the other memory blocks. Let us take the program in Figure 2 for example. Suppose there is one locking slot at both lp_1 and lp_2. If we lock m at lp_2, the rest of the memory blocks in lp_1 compete for the locking slot at lp_1. On the other hand, if we lock m at lp_1, the rest of memory blocks in lp_2 compete for the locking slot at lp_2, and the memory blocks in *region a* and *region b* can no longer be locked.

We choose to fill the locking slots from the innermost loop to the outermost loop in the program. For each loop, we try to use up all the locking slots in order to maximize the WCET reduction. Without loss of generality, suppose $lp_i \in LP$ is the innermost loop with available locking slots. With the cost-benefit analysis in section III-C, we select the memory block $m \in S_i^{-1}$ with the maximum locking gain to fill a slot at lp_i. Later, we update the resilience sets for lp_i and its outer loops. We continue to fill the slots until all the slots at lp_i are filled. Then, we move to its outer loops with available locking slots. This process terminates when all the locking slots are filled. In the end, we perform abstract cache states analysis for the cache set, and update the WCET and the worst-case path information.

When all the cache sets are analyzed and there is no improvement of WCET compared to static analysis, no cache locking will be applied. In that case, no instruction is inserted before loop entry and after loop exit. Thus, in the worst case, our approach produces the same results as static analysis.

TABLE II. CHARACTERISTICS OF BENCHMARKS

Tasks	Original code size (bytes)	Code size after locking (bytes)	Code size increment (%)	# of loops (nested)
adpcm	11,000	11,248	2.25	15 (10)
cnt	1,648	1,712	3.88	4 (4)
crc	2,048	2,096	2.34	3 (0)
edn	7,296	7,472	2.41	11 (7)
fdct	5,176	5,208	0.62	2 (0)
jfdctint	5,520	5,568	0.87	3 (0)
matmult	1,632	1,712	4.90	5 (5)
minver	6,256	6,536	4.48	17 (16)
ndes	6,352	6,544	3.02	12 (8)
st	2,248	2,312	2.85	4 (0)

IV. EXPERIMENTAL EVALUATION

We evaluate our loop-based dynamic cache locking by comparing it with static cache locking, static cache analysis and region-based dynamic cache locking approach.

A. Experimental Setup

We use the benchmarks from MRTC benchmark suite [10] as shown in Table II. Our framework is built on top of the open-source WCET analysis tool Chronos [12]. We compile the benchmarks with gcc cross-compiler for the Simplescalar PISA (Portable ISA) instruction set [6]. The experiments are performed on 2.53GHz Intel Xeon CPU with 24GB memory. IBM CPLEX is used as the ILP solver to obtain both the WCET and the locking slots.

Our approach increases the code size due to the insertion of the call instructions to locking/unlocking routines, as shown in the 3rd column of Table II. However, the code size increment is very small. The number of loops and nested loops for each benchmark is presented in the last column of Table II. As we are modeling the instruction cache, we assume a simple in-order processor with unit-latency for all data memory references. Also, we consider architectures without timing anomalies caused by interactions between caches and other architecture features. We assume a 4-way set-associative cache with block size of 32-byte, and two different cache sizes are considered. The cache hit latency is 1 cycle and the cache miss penalty is 30 cycles. We assume 150 cycles for locking and unlocking a memory block.

B. Comparison with Static Approaches

We first compare our approach with static cache analysis and static cache locking. We perform static analysis with abstract interpretation [20], [4], while we adopt the heuristic approach for partial cache locking in [7] to obtain the static locking results. We use the static analysis results as the baseline, and normalize the other results, as shown in Figure 4.

In Figure 4(a), our loop-based dynamic locking outperforms static analysis for all the benchmarks, and the improvement is up to 40% (*cnt* and *matmult*). When compared with static locking, our approach wins in most of the cases except for *crc* and *fdct*. For *crc*, static locking performs well as memory blocks that mostly affect the WCET are locked, while cache locking does not help much in *fdct*. Furthermore, we usually pay extra cost due to code size increase. On average, static locking and our dynamic locking improve the WCET by 13% and 23%, respectively. In Figure 4(b), as the cache size increases, more memory accesses can be classified as cache hits, and the improvement via static locking decreases to

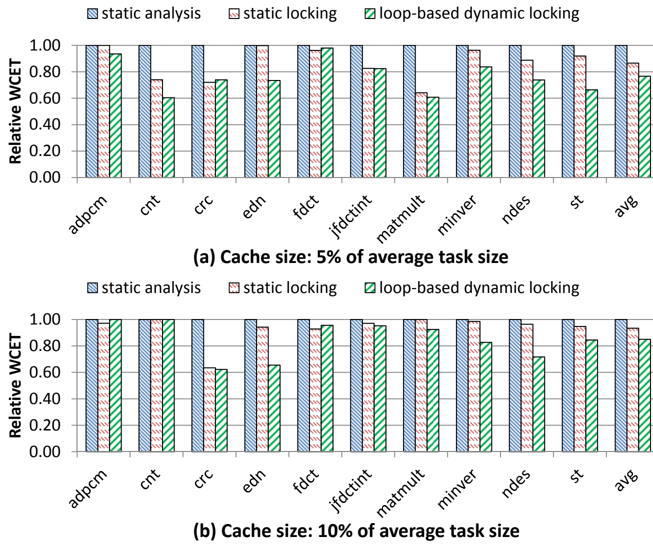

Fig. 4. Comparison between loop-based dynamic locking and static approaches.

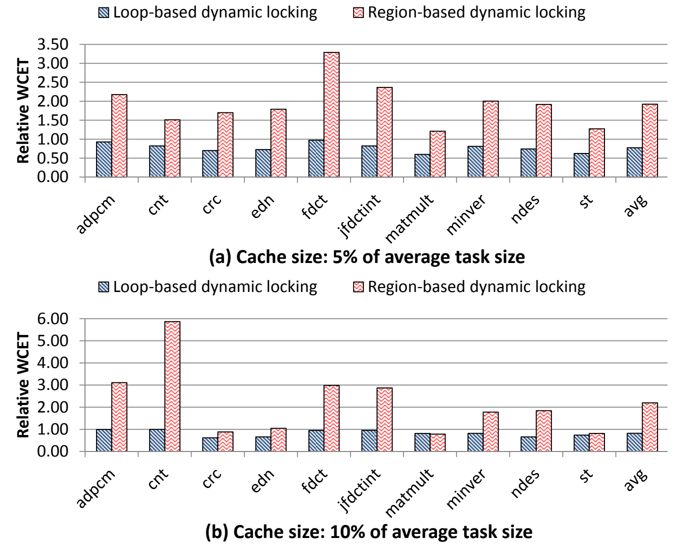

Fig. 5. Comparison between loop-based and region-based dynamic locking.

7% on average. However, our dynamic locking approach still has 15% improvement on average. For both cache sizes, the advantages of our flexible dynamic cache locking are clearly demonstrated by the benchmarks with many nested loops such as *edn*, *minver* and *ndes*. For *adpcm*, our approach achieves small improvement as most of its loops are small and most of its memory accesses are classified as cache hits.

C. Comparison with Region-based Approach

We implement the region-based dynamic cache locking approach [3] for comparison. In [3], the locking and unlocking of a memory block is handled by raising an exception, and there is no modification to the program. Thus, for a fair comparison, we assume no code change due to locking for both loop-based and region-based approaches.

Figure 5 shows the comparison results, where static analysis results are used as the baseline. For most of the benchmarks, our approach performs better than the region-based dynamic locking approach. For the region-based approach, memory blocks can only be locked at the beginning of a region, which does not provide fine-grained flexibility of selectively locking memory blocks from the same region at different program points. The region-based approach also uses full cache locking that may prevent the unlocked memory blocks to exploit their locality. However, there are a few exceptions, e.g., *matmult* in Figure 5(b). When the cache size is 10% of the average task size, most of the frequently executed memory blocks in *matmult* can fit into the cache in the region-based approach, but our approach does not allocate all the cache lines for locking due to the approximation analysis.

D. Runtime

Our loop-based approach runs efficiently for all the benchmarks. On average, it takes less than 1 second to complete.

V. Conclusion

In this paper, we propose a loop-based dynamic locking approach for instruction caches to minimize the WCET. We accurately capture the locking cost and benefit through resilience analysis. A global optimization method allocates the locking slots across loop levels and the most profitable memory blocks are selected to fill the slots. Our approach substantially reduces the WCET compared to static analysis, static locking and region-based dynamic cache locking.

Acknowledgments This work was supported by Singapore Ministry of Education Academic Research Fund Tier 2 MOE2012-T2-1-115.

References

[1] Intel XScale core developers manual. http://intel.com/design/intelxscale.

[2] S. Altmeyer et al. Resilience analysis: Tightening the CRPD bound for set-associative caches. In *LCTES*, 2010.

[3] A. Arnaud and I. Puaut. Dynamic instruction cache locking in hard real-time systems. In *RTNS*, 2006.

[4] C. Ballabriga and H. Casse. Improving the first-miss computation in set-associative instruction caches. In *ECRTS*, 2008.

[5] B. Buck and J. K. Hollingsworth. An API for runtime code patching. *Int. J. High Perform. Comput. Appl.*, 14(4), 2000.

[6] D. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *SIGARCH Comput. Archit. News*, 25(3), 1997.

[7] H. Ding et al. WCET-centric partial instruction cache locking. In *DAC*, 2012.

[8] H. Ding et al. Integrated instruction cache analysis and locking in multitasking real-time systems. In *DAC*, 2013.

[9] H. Falk et al. 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] B. K. Huynh et al. Scope-aware data cache analysis for WCET estimation. In *RTAS*, 2011.

[12] X. Li et al. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3), 2007.

[13] Y.-T. S. Li and S. Malik. Performance analysis of embedded software using implicit path enumeration. In *DAC*, 1995.

[14] Y. Liang and T. Mitra. Instruction cache locking using temporal reuse profile. In *DAC*, 2010.

[15] T. Liu et al. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *RTAS*, 2009.

[16] T. Liu et al. Instruction cache locking for multi-task real-time embedded systems. *Real-Time Syst.*, 48(2), 2012.

[17] F. Mueller. Timing analysis for instruction caches. *Real-Time Syst.*, 18(2/3), 2000.

[18] S. Plazar et al. WCET-aware static locking of instruction caches. In *CGO*, 2012.

[19] I. Puaut. WCET-centric software-controlled instruction caches for hard real-time systems. In *ECRTS*, 2006.

[20] H. Theiling et al. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18(2/3), 2000.

[21] L. Thiele and R. Wilhelm. Design for timing predictability. *Real-Time Syst.*, 28(2-3), 2004.

[22] X. Vera et al. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1), 2007.