WCET-Aware Dynamic I-Cache Locking for a Single Task

WENGUANG ZHENG, Tianjin University of Technology

HUI WU, The University of New South Wales

QING YANG, The University of Rhode Island

Caches are widely used in embedded systems to bridge the increasing speed gap between processors and off-chip memory. However, caches make it significantly harder to compute the worst-case execution time (WCET) of a task. To alleviate this problem, cache locking has been proposed. We investigate the WCET-aware I-cache locking problem and propose a novel dynamic I-cache locking heuristic approach for reducing the WCET of a task. For a nonnested loop, our approach aims at selecting a minimum set of memory blocks of the loop as locked cache contents by using the min-cut algorithm. For a loop nest, our approach not only aims at selecting a minimum set of memory blocks of the loop nest as locked cache contents but also finds a good loading point for each selected memory block. We propose two algorithms for finding a good loading point for each selected memory block, a polynomial-time heuristic algorithm and an integer linear programming (ILP)-based algorithm, further reducing the WCET of each loop nest. We have implemented our approach and compared it to two state-of-the-art I-cache locking approaches by using a set of benchmarks from the MRTC benchmark suite. The experimental results show that the polynomial-time heuristic algorithm for finding a good loading point for each selected memory block performs almost equally as well as the ILP-based algorithm. Compared to the partial locking approach proposed in Ding et al. [2012], our approach using the heuristic algorithm achieves the average improvements of 33%, 15%, 9%, 3%, 8%, and 11% for the 256B, 512B, 1KB, 4KB, 8KB, and 16KB caches, respectively. Compared to the dynamic locking approach proposed in Puaut [2006], it achieves the average improvements of 9%, 19%, 18%, 5%, 11%, and 16% for the 256B, 512B, 1KB, 4KB, 8KB, and 16KB caches, respectively.

Categories and Subject Descriptors: B.3.3 [Performance Analysis and Design Aids]: Worst-case Analysis

General Terms: Algorithms, Performance

Additional Key Words and Phrases: Worst-case execution time, dynamic I-cache locking, minimum cut

ACM Reference Format:

Wenguang Zheng, Hui Wu, and Qing Yang. 2017. WCET-aware dynamic I-cache locking for a single task. *ACM Trans. Archit. Code Optim.* 14, 1, Article 4 (March 2017), 26 pages.

DOI: <http://dx.doi.org/10.1145/3046683>

1. INTRODUCTION

Caches (I-cache and D-cache) are effective to bridge the speed gap between processors and off-chip memory. With the utilization of caches, the execution time of a task can be significantly reduced. Since caches are managed by hardware, it is difficult to know at

Authors' addresses: W. Zheng, School of Computer and Communication Engineering, Tianjin University of Technology, 391 Bin Shui Road, Xi Qing District, Tianjing 300384, China; email: wenguangz@tjut.edu.cn; H. Wu, School of Computer Science and Engineering, The University of New South Wales, Sydney, NSW 2052, Australia; email: huiw@cse.unsw.edu.au; Q. Yang, Department of Electrical, Computer & Biomedical Engineering, The University of Rhode Island, Kelley Hall, 4 East Alumni Ave., Kingston, RI 02881, USA; email: qyang@ele.uri.edu.

Permission to make digital or hard copies of part or all of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies show this notice on the first page or initial screen of a display along with the full citation. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, to republish, to post on servers, to redistribute to lists, or to use any component of this work in other works requires prior specific permission and/or a fee. Permissions may be requested from Publications Dept., ACM, Inc., 2 Penn Plaza, Suite 701, New York, NY 10121-0701 USA, fax +1 (212) 869-0481, or permissions@acm.org.

© 2017 ACM 1544-3566/2017/03-ART4 \$15.00

DOI: <http://dx.doi.org/10.1145/3046683>

compile time whether a memory access is a cache hit or not. Therefore, caches make it significantly harder to compute the worst-case execution time (WCET) of a task.

Cache locking is an effective technique for alleviating the unpredictability problem of memory accesses. If an instruction or a data item is locked into the I-cache or the D-cache, it will not be replaced. As a result, each access to a locked instruction or data item is a cache hit, taking one cycle. Furthermore, cache locking can reduce the WCET of a task, as we can lock the memory blocks on the longest paths of the task to reduce the total access time of those memory blocks.

Cache locking approaches can be classified into two categories: static cache locking and dynamic cache locking. For static cache locking, a locked memory block of a task remains locked during the entire execution of the task. Therefore, the memory blocks mapped to the same location in a cache cannot be locked simultaneously. For dynamic cache locking, a locked memory block is unlocked after its live range expires. Thus, different memory blocks can be locked into the same location of a cache if their live ranges do not overlap, leading to more efficient utilization of a cache.

For static cache locking, the major task is to select a set of memory blocks to be locked into a cache. For dynamic cache locking, there is another major task: determining a loading point for each selected memory block. A selected memory block is loaded and locked into the cache at its loading point. To reduce the WCET of a task, we need to select the memory blocks on the worst-case execution path as locked cache contents. The longest path may change after some memory blocks on the longest path are locked into the cache. For this reason, all previous cache locking approaches iteratively select the memory blocks on the longest path to be locked into the cache. However, selecting memory blocks on a single longest path does not necessarily minimize the WCET of a task.

In this article, we investigate the WCET-aware I-cache locking problem for a single task. Given a task, our objective is to select a set of memory blocks of the code of the task and find a loading point for each selected memory block such that the WCET of the task is minimized after loading and locking the selected memory blocks into the I-cache at their loading points. We make the following major contributions:

- (1) We propose a novel, min-cut-based, dynamic I-cache locking heuristic approach and two algorithms for finding a good loading point for each selected memory block: a polynomial-time heuristic algorithm and an integer linear programming (ILP)-based algorithm. Unlike all previous I-cache locking approaches that consider only the longest path, our approach considers a subgraph that contains not only the longest path but also all other paths whose lengths are close to the longest path length, and it selects a minimum set of memory blocks of the code of the loop as locked cache contents to reduce the maximum length of the whole subgraph.
- (2) We have implemented our dynamic I-cache locking approach and compared it to the two state-of-the-art approaches, namely the partial I-cache locking approach proposed in Ding et al. [2012] and the longest path-based dynamic I-cache locking approach proposed in Puaut [2006], by using a set of benchmarks from the MRTC benchmark suite [Burger and Austin 1997]. The experimental results show that the polynomial-time heuristic algorithm for finding loading points performs almost equally as well as the ILP-based algorithm. Compared to the partial locking approach proposed in Ding et al. [2012], our approach using the polynomial-time heuristic for finding a good loading point for each selected memory block achieves the average improvements of 33%, 15%, 9%, 3%, 8%, and 11% for the 256B, 512B, 1KB, 4KB, 8KB, and 16KB caches, respectively. Compared to dynamic locking proposed in Puaut [2006], it achieves the average improvements of 9%, 19%, 18%, 5%, 11%, and 16% for the 256B, 512B, 1KB, 4KB, 8KB, and 16KB caches, respectively.

This article is a major extension of previously published work [Zheng and Wu 2014]. Compared to Zheng and Wu [2014], this article proposes a new approach to loop nest locking, two new algorithms for finding a good loading point of each selected memory block, one polynomial-time heuristic algorithm, and one ILP-based algorithm, and provides new experimental results.

The rest of this article is organized as follows. Section 2 gives a survey of the related work. Section 6.3 presents a motivational example. Section 4 describes the system model and key definitions. Section 5 proposes our algorithm for selecting memory blocks of a nonnested loop as locked I-cache contents. Section 6 presents our approach to loop nests and two algorithms for finding a good loading point for each selected memory block. Section 7 shows the experimental results and analysis, and Section 8 concludes the article.

2. RELATED WORK

Many cache locking approaches have been proposed to reduce the WCET or the average-case execution time (ACET) of a task. Anand and Barua [2009] propose a static I-cache locking approach to reduce the ACET of a task. The approach introduces a cost-benefit model to approximate the net benefit of locking a memory block into the I-cache and iteratively selects a memory block with the highest benefit as locked cache contents until the I-cache is full or no memory block has a net benefit. Liang and Mitra [2010] explore the static I-cache locking problem and propose an optimal algorithm and a heuristic approach that use the temporal reuse profile to determine the most beneficial memory blocks to be locked into the I-cache to reduce the ACET of a task. Liu et al. [2012] propose two static I-cache locking approaches and one dynamic I-cache locking approach that aim at minimizing the ACET of a task by exploring the probability profile information, proving that the ACET-aware I-cache locking problem is NP-hard. Qiu et al. [2014] present a branch prediction-directed dynamic I-cache locking approach to improve the ACET of a task through cache conflict miss reduction. The approach partitions the control flow graph (CFG) of a program into disjoint execution regions and selects the most profitable memory blocks as locked cache contents by calculating the locking profit of each region. At runtime, locking routines are prefetched into a small high-speed buffer. The predetermined cache locking contents are loaded and locked at specific execution points during program execution. Anand and Barua [2015] investigate the problem of reducing the ACET of a task by using instruction cache locking and propose two heuristic algorithms: one using static cache locking and the other using dynamic cache locking.

Campoy et al. [2003] compare static cache locking and dynamic cache locking using a genetic algorithm, and they point out that static cache locking is more predictable and that dynamic cache locking shows better improvements in most cases. Puaut and Arnaud [2006] propose a dynamic I-cache locking approach that partitions a program into a set of regions. For each region, there is a loading point. Puaut [2006] presents two dynamic I-cache locking algorithms. One is a greedy algorithm, and the other is a genetic algorithm. A cost function is used for each loop to determine whether its preheader should be selected as a loading point. However, the cost function is not well defined.

Falk et al. [2007] propose a static I-cache locking approach that aims at minimizing the WCET of a task by iteratively reducing the longest path length of the task. It uses an execution flow graph (EFG) to model the possible execution paths and selects the basic block on the current longest path with the largest benefit to be locked into the cache. Liu et al. [2009] investigate the I-cache locking problem and formulate the problem using an execution flow tree (EFT) and a linear programming model. For a subset of the problems with certain properties, they propose polynomial-time optimal algorithms. Furthermore, they prove that the general problem is NP-hard. Plazar

et al. [2012] propose an ILP-based, static I-cache locking algorithm for minimizing the WCET of a task. Ding et al. [2012] point out that full cache locking may cause more cache misses that would have a negative effect on WCET reduction and propose a partial I-cache locking mechanism to lock parts of the I-cache.

Ding et al. [2014] propose a WCET-aware, dynamic I-cache locking approach for a single task. The approach uses ILP to determine the locking slots for each loop and selects the most profitable memory blocks to fill these slots. The locking benefit of a memory block m is defined as $benefit_m = (LAT_{miss} - LAT_{hit}) \times freq_m$, where LAT_{miss} and LAT_{hit} are the cache miss latency and the cache hit latency, respectively, and $freq_m$ is the execution frequency of the memory block m on the worst-case path. However, the approach is incomplete and seriously flawed. First, it assumes that the execution frequency of a memory block m on the worst-case path is known. However, this assumption is not valid because the longest path may change after a set of memory blocks are locked. Second, it does not address the problem of determining a good locking point for each selected memory block.

Several approaches to the D-cache locking problem have been proposed. Vera et al. [2003] propose an approach that combines compile-time cache analysis with data cache locking to estimate the worst-case memory performance (WCMP) in a safe, tight, and fast way. To obtain predictable cache behavior, the approach first locks the cache for those parts of the code where the static analysis fails. To minimize the performance degradation, the approach loads the cache with the data likely to be accessed. Zheng and Wu [2015] propose two D-cache locking approaches that aim at minimizing the WCET of a task. The first approach formulates the problem as a global ILP problem that simultaneously selects a near-optimal set of variables as the locked cache contents and allocates them to the D-cache. The second one iteratively constructs a subgraph of the CFG of the task where the lengths of all paths are close to the longest path length and uses an ILP formulation to select a near-optimal set of variables in the subgraph as the locked cache contents and allocate them to the D-cache. Zheng and Wu [2017] also propose two improved approaches that consider the individual sections of nonscalar variables in different cache sets as locked cache contents and a new D-cache allocation algorithm.

Several approaches have been proposed to integrate task scheduling and cache locking. Campoy et al. [2001] propose a genetic algorithm for the problem of selecting instructions to be locked into the I-cache to reduce the response time of multitasks. Puaut and Decotigny [2002] investigate static I-cache locking for multitask real-time systems. They propose two algorithms: one aiming at minimizing the CPU utilization and the other attempting to minimize the interferences between tasks. Campoy et al. [2002] propose a dynamic I-cache locking approach for multitask systems. The approach uses the response time analysis approach proposed in Busquets-Mataix et al. [1996] for the schedulability test and combines the schedulability analysis with cache locking using a genetic algorithm to improve the performance of the I-cache on a multitasking, preemptive real-time system. Liu et al. [2010] propose an approach that combines cache locking with task assignment to reduce the WCETs of a set of tasks on a multiprocessor system with two levels of caches. The approach applies cache locking to both the I-cache and D-cache by using the algorithms proposed in Liu et al. [2009] and Vera et al. [2003] to reduce the WCET of each task, and it optimizes the task assignment considering the cache size.

3. A MOTIVATIONAL EXAMPLE

In this section, we use an example to illustrate the key ideas of our approaches and compare it to three state-of-the-art approaches to the WCET-aware I-cache locking problem for a single task. The three approaches are the static full cache locking approach proposed in Falk et al. [2007], the longest path-based dynamic cache locking approach proposed in Puaut [2006], and the partial cache locking approach proposed in

Fig. 1. CFG of the motivation example.

Fig. 2. System model.

Ding et al. [2012]. Consider a task with the CFG shown in Figure 1. The task consists of three loops where the loop A and the loop B are nested in the loop C. The numbers of iterations of the loop A, the loop B, and the loop C are 4, 3, and 10, respectively. For ease of descriptions, we make the following assumptions:

- (1) The preheader of each loop is empty.
- (2) All of the basic blocks are mapped to the same set of the I-cache.
- (3) Each basic block is exactly one memory block.
- (4) The associativity of the I-cache is 4.
- (5) If a basic block is in the I-cache, its execution time is 1 cycle. Otherwise, its execution time is 30 cycles.
- (6) The time for fetching and locking one memory block is 30 cycles.

Figure 3 illustrates the locked basic blocks and their loading points computed by the three approaches and our approach, where each small circle denotes a preheader containing instructions for loading and locking the relevant memory blocks. For example, Figure 3(a) shows that the basic blocks m_0 , m_1 , m_3 , and m_5 are selected as the locked contents by the static full cache locking approach, and these four basic blocks are loaded and locked at the preheader of the loop C.

3.1. Static Full Cache Locking

Consider the approach proposed in Falk et al. [2007], which iteratively finds the longest path and selects the basic block with the largest benefit on the longest path as locked cache contents. Obviously, the basic blocks of the loop A have larger benefits due to its larger number of iterations. The four basic blocks m_0 , m_1 , m_3 , and m_5 may be subsequently selected as locked cache contents at the preheader of the loop C as shown in Figure 3(a). Since this approach uses static cache locking, the selected basic blocks are fetched and locked only once. Therefore, the time taken to fetch and lock the selected basic blocks is $30 * 4 = 120$. As a result, the WCET of the task is 5,300.

3.2. Longest Path-Based Dynamic Cache Locking

Using the longest path-based dynamic cache locking algorithm proposed in Puaut [2006], the selected basic blocks are loaded and locked at the preheaders of the loop A and the loop B as shown in Figure 3(b). For the loop A, m_0 , m_1 , m_4 , and m_5 may be subsequently selected as locked cache contents, as they have the largest frequencies on the current longest paths. When the loop A terminates, all cache lines occupied by its memory blocks are released. Therefore, the entire loop B will be locked. The memory blocks m_0 , m_1 , m_4 , and m_5 will be fetched and locked 10 times at the preheader of the loop A, and the three memory blocks of the loop B will also be fetched and locked

Fig. 3. Locked basic blocks and loading points by different approaches.

10 times at the preheader of the loop B. Thus, the time taken to fetch and lock the selected memory blocks is 2, 100, and the WCET of the task is 4, 670.

3.3. Partial Cache Locking

Using the partial cache locking algorithm proposed in Ding et al. [2012], m_0 and m_6 may be selected as locked cache contents at the preheader of the loop C as shown in Figure 3(c). The other two cache lines are kept free so that the memory accesses to m_7 and m_8 will be hit after the first cache misses during the execution of the loop B. For the loop A, each iteration will cause three cache misses. Since only two memory blocks are locked, the time for fetching and locking these two memory blocks is 60, and the WCET of the task is 4, 370.

3.4. Our Approach

To find a better solution, our dynamic cache locking approach aims at finding a minimum set of memory blocks as locked cache contents and finds a good loading point for each selected memory block. First, our approach finds the minimum node cuts of the CFG of each loop of a loop nest without back edges from the innermost loops to the outermost loop and selects the memory blocks in the minimum node cuts as locked cache contents considering the cache capacity constraints. For the loop A, the minimum node cuts $\{m_0\}$, $\{m_1, m_2\}$, and $\{m_5\}$ are selected. For the loop B, all three basic blocks are selected. Second, our approach moves the loading points of the selected memory blocks to the preheaders of the outer loops to further reduce the WCET of the loop nest. Initially, the loading point of a selected memory block is at the preheader of the loop that contains it directly. Later, a loading point may be moved to the preheader of an outer loop if the motion of the loading point results in a smaller WCET of the whole loop nest. In this example, our approach moves the loading point of m_5 from the preheader of the loop A to the preheader of the loop C to further reduce the WCET of

the entire loop nest as shown in Figure 3(d). Notice that after the loading point of a selected memory block is moved to the preheader of an outer loop, the live range of the memory block is extended to the whole outer loop. By our approach, the WCET of the task is 3, 240.

4. SYSTEM MODEL AND DEFINITIONS

We investigate the WCET-aware dynamic I-cache locking problem for a single task. The target processor has an I-cache and a D-cache as shown in Figure 2. The I-cache is an n -way set associative cache with k sets. Each set has a set number between 0 and $k - 1$, and all sets have the same size. We do not consider the D-cache locking problem.

A basic block is a sequence of instructions with only one entry and one exit. Each basic block is mapped into one or more sets of the I-cache. The main memory is partitioned into memory blocks such that each memory block is mapped to exactly one cache line. The basic locking unit is a cache line. If an instruction is locked in the I-cache, it takes one cycle to fetch the instruction. Otherwise, it takes c cycles.

Since it is not beneficial to lock a memory block that is not in any loop, we consider individual loop nests only. Notice that after a loop nest finishes, all of its locked memory blocks can be unlocked. Therefore, each loop nest can use the entire I-cache.

For each loop nest, we define a loop nest tree as follows.

Definition 4.1 (Loop Nest Tree). Given a loop nest, its loop nest tree is weighted tree $T = \langle L, W, E \rangle$, where, $L = \{L_i : L_i \text{ is a loop in the loop nest}\}$, $W = \{w_i : w_i \text{ is the WCET of the loop } L_i\}$, and $E = \{(L_i, L_j) : L_j \text{ is immediately nested in } L_i\}$.

We use CFG to represent a task where each node has a weight, denoting the execution time of the corresponding basic block.

Given a loop nest L , we construct a weighted DAG G_i for each loop L_i of L recursively as follows:

- (1) Construct the loop nest tree of L .
- (2) Construct the control flow graph $CFG(L)$ of L , where the weight of each node is the execution time of the corresponding basic block.
- (3) For each loop L_i of L in reverse topological order of the loop nest tree, do the following:
 - (a) Let G_i be the whole subgraph of $CFG(L)$ that represents L_i .
 - (b) Remove all back edges of G_i .
 - (c) Shrink the whole subgraph of $CFG(L)$ that represents L_i into two nodes: a preheader node and a loop node.
 - (d) Set the weight of the loop node to the WCET of L_i as follows:
 - If L_i is a loop where the loop exit condition is checked at the beginning of each iteration, the WCET of L_i is equal to $l_{max}(G_i) * n(L_i) + w_h(L_i)$, where $l_{max}(G_i)$, $n(L_i)$, and $w_h(L_i)$ are the length of the longest path of G_i without the preheader node, the maximum number of iterations of L_i , and the execution time of the header of L_i , respectively. Otherwise, it is $l_{max}(G_i) * n(L_i)$.

The weighted DAG of the loop nest L is the weighted DAG of the root of the loop nest tree of L .

Definition 4.2 (x -Spanning Graph). Given a weighted DAG G and a positive constant x , the x -spanning graph $G(x)$ is a subgraph of G where the length of each path is larger than x .

Given a weighted DAG G , its x -spanning graph can be computed in $O(e)$ time as shown in Algorithm 1, where e is the number of edges in G .

ALGORITHM 1: $x - \text{SpanningGraph}(G, x)$ **Input:** G : a weighted DAG x : a positive constant**Output:** The x -spanning graph $G(x)$ of G in which each path length is greater than x **begin** $G(x) = G$;/*compute the length $a(v)$ of the longest path to each node v */**for each node v in topological order do****if $\text{indegree}(v) = 0$ then** $a(v) = w(v)$;/* $w(v)$ is the weight of node v */**end****else** $a(v) = w(v) + \max\{a(u) : u \text{ is a parent of } v \text{ in } G\}$;**end****end**/*compute $b(v)$ for each node v where $b(v)$ is x minus the longest path length from v to a sink node*/**for each node in G in reverse topological order do****if $\text{outdegree}(v) = 0$ then** $b(v) = x$;**end****else** $b(v) = \min\{b(u) - w(u) : u \text{ is a child of } v\}$;**end****if $b(v) > a(v)$ then**delete v and all edges incident on v ;**end****end****return $G(x)$;****end**

Definition 4.3 (y -Projection Graph). Given an x -spanning graph $G(x)$ and a set y of the I-cache, the y -projection graph $G(x, y)$ is a subgraph of $G(x)$ where the corresponding basic block or loop of each node has a memory block mapped into the set y and the memory block has not been selected as locked cache contents.

The algorithm for constructing the y -projection graph $G(x, y)$ is shown in Algorithm 2. Its time complexity is $O(e)$, where e is the number of edges in $G(x)$.

The WCET-aware, dynamic cache locking problem is formulated as follows. Given a task, select a set of memory blocks of code as locked cache contents, and for each selected memory block, find its loading point and address (cache line number) in the cache set to which it is mapped such that the WCET of the task is minimized. We propose a heuristic approach to this problem. After selecting a set of memory blocks to be locked into the I-cache, our approach stores the memory addresses of all selected memory blocks at the end of code section so that they can be locked into the I-cache efficiently. To lock the selected memory blocks into the corresponding cache lines and reduce the size of extra code for cache locking, we introduce a special instruction *lock* with the following format:

$$\text{lock offset, start-addr, entries,}$$

where *start-addr* is the start address of a sequence of memory blocks that are contiguously locked into the same set of the I-cache, *entries* is the number of memory blocks of the sequence, and *offset* is the offset in the I-cache of the first memory block of the sequence. For each memory block $M_i (i = 0, 1, \dots, \text{entries} - 1)$ specified by the three

ALGORITHM 2: $y - \text{ProjectionGraph}(G(x), y)$ **Input:** $G(x)$: an x -spanning graph y : a cache set number**Output:** The y -projection graph $G(x, y)$ of $G(x)$ **begin** $G(x, y) = G(x);$ **for** each node v in $G(x, y)$ **do****if** all memory blocks of v have been selected as locked cache contents or v has no memory block mapped into the set y **then****for** each parent v_i and each child v_j of v **do****if** the edge (v_i, v_j) does not exist in $G(x, y)$ **then**Add (v_i, v_j) to $G(x, y);$ **end****end**delete v and all edges incident on $v;$ **end****end****return** $G(x, y);$ **end**

operands, the *lock* instruction locks M_i into the $(\text{offset} + i)$ -th cache line of the corresponding cache set, updates the corresponding entry in the tag store, and sets the lock bit of the cache line. The *lock* instruction takes c cycles for loading and locking each memory block, where c is the cache miss penalty.

When a cache line is already locked, *lock* can still load a memory block into it. To prevent disrupting the mapping between basic blocks and memory blocks, we insert a procedure call instruction at the preheader of each loop. The instruction calls the corresponding loading procedure that is placed at the end of the task. The loading procedure contains a sequence of *lock* instructions for loading the selected memory blocks and locking them. The program point of a call to a loading procedure is called a *loading point*. Initially, each loading procedure is empty. After a set of memory blocks of a task is selected as locked cache contents, a sequence of zero or more *lock* instructions will be added to each loading procedure. If no *lock* instruction is added to a loading procedure, the corresponding call will be replaced by a *nop* instruction.

Give a set S , $|S|$ denotes the number of elements of S .

5. I-CACHE LOCKING FOR A NON-NESTED LOOP

Before presenting our approach to dynamic I-cache locking for a loop nest, we propose a min-cut-based algorithm for selecting the memory blocks of a nonnested loop as locked cache contents. Given a nonnested loop L_i , our algorithm aims at selecting a minimum set of memory blocks of the loop as locked cache contents such that the WCET of the loop is minimized. Our algorithm works as follows:

- (1) Construct the weighted DAG G_i of the loop as described in Section 4.
- (2) Repeat the following steps for each set y of the I-cache until the set y is full or no more memory block can be selected as locked cache contents:
 - (a) Find the longest path length $l_{\max}(G_i)$ of G_i .
 - (b) Construct the x -spanning graph of G_i , where x is $l_{\max}(G_i) - c$, where c is the off-chip memory access latency.
 - (c) Construct the y -projection graph of the x -spanning graph.
 - (d) Find a minimum node cut N of the y -projection graph.

Fig. 4. A DAG G . Fig. 5. $G(150)$. Fig. 6. $G(150,0)$.

- (e) If the size of N is not less than the number of iterations of the loop, discard N and the algorithm terminates. Otherwise, if the number of free cache lines of the set y is not less than the size of N , do the following:
- (i) For each node v_i in N , select a memory block m_i of its corresponding basic block with the largest execution time among all memory blocks of the basic block that has not been selected as locked cache contents.
 - (ii) Recalculate the execution times of v_i and all basic blocks affected.
 - (iii) Update the weight of each node of N .

When the size of a node cut is not less than the number of iterations of the loop, locking the memory blocks of the corresponding basic blocks of the node cut does not reduce the WCET of the loop. Furthermore, the size of a node cut does not exceed the size of the previous node cut during the execution of our algorithm. Therefore, our algorithm terminates when the size of a node cut is not less than the number of iterations of the loop.

The details of our algorithm for a single loop are given in Algorithm 3. This algorithm returns $C_y(L_i)$ and $size[]$, where $C_y(L_i)$ stores a set of the minimum cuts of the memory blocks of the loop L_i and $size[i]$ stores the number of the memory blocks of the loop L_i that have been selected as locked cache contents in each cache set i . If L_i does not contain a nested loop, all memory blocks in $C_y(L_i)$ will be locked. Otherwise, our approach to a loop nest will determine which minimum cuts in $C_y(L_i)$ are locked by comparing the benefit of locking a minimum cut and the benefit of moving a loading point from the preheader of an inner loop to the preheader of L_i .

We use an example to show how our algorithm works. Consider a weighted DAG G shown in Figure 4. Assume that the I-cache has two sets, and each node (basic block) is exactly one memory block. The execution time of each basic block is 1 if it is locked into the I-cache. Otherwise, it is 30. All circle nodes are mapped to set 0, and all square nodes are mapped to set 1. Next, we show how our algorithm selects the locked cache contents for set 0. Based on the assumptions, the longest path length of the weighted DAG G is 180 and x is 150. In the first iteration, $G(150)$, the 150-spanning graph, is constructed as shown in Figure 5. Based on $G(150)$, the 0-projection graph $G(150, 0)$ of $G(150)$ is constructed as shown in Figure 6. In $G(150, 0)$, all nodes belong to set 0; $\{v_6\}$ is a minimum node cut of $G(150, 0)$. Therefore, v_6 is selected as the locked contents, and its weight is changed from 30 to 1. Now, the longest path length of G becomes 151. In the second iteration, $G(121)$ is constructed as shown in Figure 7, and the 0-projection graph of $G(121)$ is constructed as shown in Figure 8. A minimum node cut of $G(121, 0)$ is $\{v_1\}$. Hence, our algorithm selects v_1 as locked cache contents as shown in Figure 9.

ALGORITHM 3: *SingleLoopLocking*(L_i, k, n)**Input:** L_i : a loop k : the total number of sets in the I-cache $n[]$: a one-dimensional array where $n[y](y = 0, 1, \dots, k-1)$ stores the number of the free cache lines in the set y of the I-cache**Output:** $C(L_i)$: the set of $C_0(L_i), \dots, C_{k-1}(L_i)$, where $C_y(L_i)$ ($y = 0, 1, \dots, k-1$) is the set of the memory blocks of the loop L_i to be locked into the set y of the I-cache
 $size[]$: a one-dimensional array where $size[y](y = 0, 1, \dots, k-1)$ stores the number of the memory blocks of the loop that have been selected as locked cache contents in the set y **begin**construct the weighted DAG G_i of the loop L_i ;**for** each cache set $y(y = 0, 1, \dots, k-1)$ **do** $size[y] = 0$; $n'[y] = n[y]$; $C_y(L_i) = \emptyset$; $\lambda = 1$; /* λ stores the number of nodes cuts for each cache set */ **while** $n'[y] \geq 0$ **do** $B_\lambda^y(L_i) = \emptyset$; /* $B_\lambda^y(L_i)$ stores a cut of the selected memory blocks */ compute the longest path length $l_{max}(G_i)$ of G_i ; $x = l_{max}(G_i) - c$; $G_i(x) = x - \text{SpanningGraph}(G_i, x)$; $G_i(x, y) = y - \text{ProjectionGraph}(G_i(x), y)$; **if** $G_i(x, y) = \emptyset$ **then** **break**; **end** **else** compute the minimum node cut N of $G_i(x, y)$; **if** $|N|$ is less than the number of iterations of L_i **then** **if** $|N| \leq n'[y]$ **then** **for** each node $v_i \in N$ **do** select a memory block m_i with the largest execution time among all memory blocks of v_i that have not been selected as locked cache contents; $B_\lambda^y(L_i) = B_\lambda^y(L_i) \cup \{m_i\}$; recalculate the execution times of v_i and all other nodes affected; updates their corresponding node weights in $G_i(x, y)$; **end** $size[y] = size[y] + |B_\lambda^y(L_i)|$; **end** **else** **break**; **end** **end** **end** $n'[y] = n'[y] - size[y]$; $\lambda = \lambda + 1$; **end** $C_y(L_i) = \cup_{v=1, \dots, \lambda} \{B_v^y(L_i)\}$;**end****return** ($C(L_i), size[]$);**end**

Fig. 7. $G(121)$ with v_6 locked. Fig. 8. $G(121,0)$ with v_6 removed. Fig. 9. The DAG G after locking v_6 and v_1 .

Now, the longest path length of DAG G is reduced to 122. The same selection process is applied to square nodes to find locked cache contents for set 1.

Next, we analyze the time complexity of Algorithm 3. Clearly, the time complexity of one iteration of the outermost *for* loop is dominated by finding a minimum node cut C of the y -projection graph. The maximum number of iteration of the *while* loop is $O(m)$, where m is the number of memory blocks of the loop. Therefore, the time complexity of Algorithm 3 is $O(mt)$, where t is the time complexity of finding a minimum node cut of a y -projection graph. The minimum node cut problem can be converted into the minimum edge cut problem in $O(e)$ time [Skiena 1998], where e is the number of edges in the graph. Given a weighted DAG, the minimum edge cut can be found in $O(ne + n^2 \log n)$ time [Stoer and Wagner 1997], where n and e are the number of nodes and the number of edges, respectively, of the DAG G_i . As a result, the time complexity of Algorithm 3 is $O(mne + mn^2 \log n)$.

6. CACHE LOCKING FOR A LOOP NEST

In this section, we propose an approach to the problem of the WCET-aware dynamic I-cache locking for a loop nest. The objective of cache locking for a loop nest is to select a set of memory blocks of the loop nest and determine both the loading point and the start address in the corresponding cache set for each selected memory block such that the WCET of the loop nest is minimized and the cache capacity constraints are satisfied. To facilitate descriptions, we introduce the notations shown in Table I.

Consider a loop nest L . For each cache set y ($y = 0, 1, \dots, k-1$), our approach starts with leaf loops and work toward the root of the loop nest tree of L . Each time, it selects an unprocessed loop L_i with the maximum total number of iterations on the longest path of the weighted DAG of the loop nest L . If L_i is an innermost loop, our approach uses Algorithm 3 to select a set $C_y(L_i)$ of the memory blocks of L_i as locked cache contents and set their loading points to the preheader of L_i . Otherwise, our approach first evokes Algorithm 3 to select a set $C_y(L_i)$ of the memory blocks of L_i as the candidates of locked cache contents and then calls our heuristic algorithm or our ILP-based algorithm to select a subset of the memory blocks in $C_y(L_i)$ as locked cache contents and a set $M_y(L_i)$ of previously memory blocks whose loading points are in the preheaders of the children of L_i , sets the loading points of all memory blocks in the subset of $C_y(L_i)$ to the preheader of L_i , and moves the loading points of all memory blocks in $M_y(L_i)$ from the preheaders of the children of L_i to the preheader of L_i .

The objective of both the heuristic algorithm and the ILP-based algorithm is to select a subset of the memory blocks in $C_y(L_i)$ and a set $M_y(L_i)$ of the previously selected memory blocks whose loading points are at the preheaders of the children of L_i . The

Table I. Notations

Notation	Definition
s	Number of cache lines in each set of the I-cache
c	Time for loading and locking one memory block into the I-cache
I_i	Number of iterations of the loading points at the preheader of L_i
$c_y(L_i)$	Total number of cache lines needed to store all selected memory blocks of the loop nest tree rooted at L_i in the cache set y
$A_y(L_i)$	Set of memory blocks that are loaded and locked at the preheader of a loop L_i in the set y of the I-cache
$CH(L_i)$	Set of children of a loop L_i in the loop nest tree
$B_v^y(L_i)$ ($y = 0, 1, \dots, k-1$ and $v = 1, \dots, \lambda$)	Minimum cut of the memory blocks of L_i selected as the candidates of locked cache contents by using Algorithm 3 for each cache set y

memory blocks of the subset and $M_y(L_i)$ will be loaded and locked into the I-cache at the preheader of the loop L_i .

Our approach loads and locks the selected memory blocks into the I-cache at the preheaders of loops. The live range of a selected memory block spans the whole loop that loads and locks the memory block into the I-cache at its preheader. If a loading point of a memory block is moved from the preheader of a child of L_i in the loop nest tree to the preheader of L_i , the live range of the memory block will be extended to the whole loop of L_i . For each pair of the memory blocks whose loading points are in the preheaders of the two children of L_i in the loop nest tree, their live ranges are disjoint. Furthermore, for each memory block loaded at the preheader of L_i , its live range overlaps with that of each memory block loaded at the preheader of any child of L_i in the loop nest tree.

To determine the number of the free cache lines in the cache set y ($y = 0, 1, \dots, k-1$), for each loop L_i , we introduce a set of variables as shown in Table I.

After our approach calls our heuristic algorithm or our ILP-based algorithm, $c_y(L_i)$ needs to be updated as follows:

$$c_y(L_i) = |A_y(L_i)| + \max\{c_y(L_j) : L_j \in CH(L_i)\}. \quad (1)$$

After our approach selects a set of memory blocks as locked cache contents and finds a loading point for each selected memory block, it inserts *lock* instructions to the loading procedure of each loop L_i to load and lock the selected memory blocks of L_i into the I-cache. In each cache set y ($y = 0, 1, \dots, k-1$), all memory blocks locked at the preheader of each loop L_i are stored contiguously in any order. Let $addr_y(L_i)$ be the start address (cache line number) of the first memory block locked into the set y at the preheader of a loop L_i . For each loop L_i , $addr_y(L_i)$ is recursively calculated as follows:

$$addr_y(L_i) = \begin{cases} 0 & L_i \text{ is an innermost loop} \\ \max\{addr_y(L_j) + |A_y(L_j)| : L_j \in CH(L_i)\} & \text{otherwise.} \end{cases} \quad (2)$$

The details of our approach to the WCET-aware, dynamic cache locking problem for a loop nest are shown in Algorithm 4. Our heuristic algorithm and ILP-based algorithm will be described in Sections 6.1 and 6.3, respectively.

6.1. Heuristic Algorithm

The objective of our heuristic algorithm is to select a subset of the memory blocks in $C_y(L_i)$ as locked cache contents and a set $M_y(L_i)$ of previously selected memory blocks whose loading points are in the preheaders of the children of L_i , set the loading points of all memory blocks in the subset of $C_y(L_i)$ to the preheader of L_i , and move the loading

ALGORITHM 4: *LoopNestLocking*(L, k, s)**Input:** L : a loop nest k : the number of sets in the I-cache s : the number of the cache lines in each cache set**Output:** A set of memory blocks selected as locked cache contents and their loading points**begin** construct a loop nest tree T of L ; construct the weighted DAG G of L ; $T' = T$; **while** $T' \neq \emptyset$ **do** among all leaf nodes in T' find a leaf node L_i on the longest path of G with the maximum total number of iterations; **if** L_i does not exist **then**

break;

end **if** L_i is a leaf node in T **then** **for each** cache set $y(y = 0, 1, \dots, k - 1)$ **do** $n[y] = s$; /* $n[y]$ stores the number of the free cache lines in the set y of the I-cache*/ **end** $(C(L_i), \text{size}[]) = \text{SingleLoopLocking}(L_i, k, n)$; **for each** cache set $y(y = 0, 1, \dots, k - 1)$ **do** $A_y(L_i) = \{m_t : m_t \in B_y^v(L_i) \text{ and } B_y^v(L_i) \in C_y(L_i)\}$; /* $A_y(L_i)$ stores all selected memory blocks of L_i in the set y of the I-cache*/ $c_y(L_i) = \text{size}[y]$; **end** **end** **else** **for each** cache set $y(y = 0, 1, \dots, k - 1)$ **do** $n[y] = s - \max\{c_y(L_j) : L_j \in CH(L_i)\}$; **end** **if** no free cache line is left **then**

break;

end **else** $(C(L_i), \text{size}[]) = \text{SingleLoopLocking}(L_i, k, n)$; **for each** cache set $y(y = 0, 1, \dots, k - 1)$ **do** call our heuristic algorithm or our ILP-based algorithm to select a subset of the memory blocks in $C_y(L_i)$ and a set $M_y(L_i)$ of the previously selected memory blocks whose loading points are at the preheaders of the children of L_i and lock the selected memory blocks at the preheader of L_i ; update the weights of the preheader of L_i and the preheaders of its child loops in G ; **end** **end** **end** **end** shrink the subgraph of G representing L_i into a preheader block node and a loop block node, and set the node weights of the preheader block node and the loop block node to the execution time of the loading procedure and the WCET of L_i , respectively; remove the subtree rooted at L_i and the edge between L_i and its parent from T' ; **end** **for each** L_i in the loop nest **do** add *lock* instructions to the loading procedure of L_i to load and lock the selected memory blocks into the I-cache; **end****end**

points of all memory blocks in $M_y(L_i)$ from the preheaders of the children of L_i to the preheader of L_i , such that the WCET of L_i is minimized.

For each cut $B_v^y(L_i) \in C_y(L_i)$ ($v = 1, \dots, \lambda$), let $R(B_v^y(L_i))$ be the WCET reduction of L_i by locking all memory blocks in $B_v^y(L_i)$ at the preheader of L_i . We define a benefit, denoted $benefit(B_v^y(L_i))$, as follows:

$$benefit(B_v^y(L_i)) = \frac{R(B_v^y(L_i))}{|B_v^y(L_i)|}. \quad (3)$$

Our heuristic algorithm works iteratively. Each time, it finds a cut $B_v^y(L_i)$ from $C_y(L_i)$ with the maximum benefit and a set S of the memory blocks in $\cup_{L_j \in CH(L_i)} A_y(L_j)$ with the maximum impact on the WCET reduction of the loop nest such that the size of the cache set y used by the currently selected locked cache contents after moving the loading points of all memory blocks of S to the preheader of L_i does not exceed the size of the cache set y used by the currently selected locked cache contents after locking the memory blocks of $B_v^y(L_i)$ at the preheader of L_i . If the memory blocks in $B_v^y(L_i)$ reduce the WCET of the loop nest more than the memory blocks in S do, our heuristic algorithm selects all memory blocks in $B_v^y(L_i)$ as locked cache contents and locks them into the I-cache at the preheader of L_i . Otherwise, it moves the loading points of all memory blocks in S to the preheader of L_i . When no loading point can be moved, our heuristic keeps selecting a cut with the maximum impact on the WCET reduction of the loop nest as locked cache contents. When no memory cut in $C_y(L_i)$ is left, our heuristic keeps moving a loading point with the maximum WCET reduction from the preheader of a child of L_i in the loop nest tree to the preheader of L_i . The details of our heuristic algorithm are shown in Algorithm 5. This algorithm calls *NoComparison()* to handle the two special cases where either no loading point from the children of L_i can be moved or no cut in $C_y(L_i)$ is left.

6.2. Time Complexity Analysis

First, we analyze the time complexity of our heuristic algorithm, Algorithm 5. Given a loop L_i , this algorithm either selects a cut from $C_y(L_i)$ or selects a set of memory blocks from $\cup_{L_j \in CH(L_i)} A_y(L_j)$ at each iteration. Therefore, the total number of iterations of the *while* loop is $O(n_i)$, where n_i is the number of memory blocks of the loop L_i . The time complexity of computing the WCET of L_i is $O(e_i)$, where e_i is the number of edges in the weighted graph G_i of L_i . Therefore, the time complexity of each iteration of the *while* loop is $O(n_i e_i)$. As a result, the time complexity of Algorithm 5 is $O(n_i^2 e_i)$.

Next, we analyze the time complexity of Algorithm 4 using our heuristic algorithm. Given a loop nest L , in the worst case, this algorithm processes all loops of L . For each loop L_i of L , Algorithm 4 calls Algorithm 3 to find the candidates of locked cache contents of L_i , and each call takes $O(m_i n_i e_i + m_i n_i^2 \log n_i)$ time as discussed in Section 5, where m_i , n_i , and e_i are the number of memory blocks of L_i , the number of nodes, and the number of edges in the weighted DAG G_i of L_i , respectively. In addition, for each cache set y ($y = 0, 1, \dots, k-1$), it calls Algorithm 5 to select a subset of the memory blocks in $C_y(L_i)$ and a set $M_y(L_i)$ of the previously selected memory blocks whose loading points are at the preheaders of the children of L_i and lock the selected memory blocks at the preheader of L_i . Therefore, the time complexity of Algorithm 4 is $\sum_{L_i \in L} (O(m_i n_i e_i + m_i n_i^2 \log n_i) + O(n_i^2 e_i)) = O(d(mne + mn^2 \log n))$, where d , m , n , and e are the height of the loop nest tree of L , the number of memory blocks of L , the number of nodes, and the number of edges of the weighted graph of L , respectively. The height of the loop nest tree of L is a very small number and can be considered as a constant. As a result, the time complexity of Algorithm 4 is $O(mne + mn^2 \log n)$.

ALGORITHM 5: *HeuristicAlgorithm*

Input: $C_y(L_i)$: a set of the memory blocks of L_i selected as candidates of locked cache contents for the set y
 $A_y(L_j) (L_j \in CH(L_i))$: a set of the memory blocks loaded and locked into the set y at the preheader of each child L_j of L_i in the loop nest tree

Output: $A_y(L_i)$: a set of memory blocks loaded and locked into the set y at the preheader of L_i

begin

$A_y(L_j) = \emptyset$;

$c_y(L_i) = |A_y(L_j)| + \max\{c_y(L_j) : L_j \in CH(L_i)\}$;

while $c_y(L_i) \leq s$ **do**

for each cut $B_v^y(L_i) \in C_y(L_i)$ **do**

$benefit(B_v^y(L_i)) = R(B_v^y(L_i)) / |B_v^y(L_i)|$;

end

find a cut $B_{max}^y(L_i) \in C_y(L_i)$ with the largest benefit;

$c'_y(L_i) = c_y(L_i) + |B_{max}^y(L_i)|$;

compute the WCET $W_s(L_i)$ of L_i assuming that all memory blocks in $B_{max}^y(L_i)$ are locked at the preheader of L_i ;

for each $L_j \in CH(L_i)$ **do**

save the current state of $c_y(L_j)$ and $A_y(L_j)$;

end

save the current state of $A_y(L_i)$;

$M_y(L_i) = \emptyset$;

$c_y(L_i)'' = c_y(L_i)$;

while $c'_y(L_i) \leq c'_y(L_i)$ **do**

If $\cup_{L_j \in CH(L_i)} A_y(L_j) = \emptyset$ break;

on the longest path of weighted DAG G_i of L_i , find a memory block m_t in $\cup_{L_j \in CH(L_i)} A_y(L_j)$ with the maximum accumulated execution time in L_i .

/*The accumulated execution time of a memory block is the time for loading and locking the memory block multiplied by its total number of iterations in L_i */

assume that the loading point of m_t is at the preheader of a child L_r of L_i ;

/* tentatively move the loading point of m_t to the preheader of L_i */

$A_y(L_r) = A_y(L_r) - \{m_t\}$;

$A_y(L_i) = A_y(L_i) \cup \{m_t\}$;

$c_y(L_r) = c_y(L_r) - 1$;

$c_y(L_i)'' = |A_y(L_i)| + \max\{c_y(L_j) : L_j \in CH(L_i)\}$;

$M_y(L_i) = M_y(L_i) \cup \{m_t\}$;

compute the WCET $W_m(L_i)$ of L_i ;

end

if $W_s(L_i) > W_m(L_i)$ **then**

/* move the loading points of the memory blocks in $M_y(L_i)$ to the preheader of L_i */

$A_y(L_i) = A_y(L_i) \cup M_y(L_i)$;

$c_y(L_i) = |A_y(L_i)| + \max_{L_j \in CH(L_i)} c_y(L_j)$;

end

else

/* load and lock the memory blocks in $B_{max}^y(L_i)$ at the preheader of L_i */

for each $L_j \in CH(L_i)$ **do**

restore the state of $c_y(L_j)$ and $A_y(L_j)$ saved before;

end

restore the state of $A_y(L_i)$ saved before;

$C_y(L_i) = C_y(L_i) - B_{max}^y(L_i)$;

$A_y(L_i) = A_y(L_i) \cup B_{max}^y(L_i)$;

$c_y(L_i) = |A_y(L_i)| + \max_{L_j \in CH(L_i)} c_y(L_j)$;

end

$NoComparison()$;

end

ALGORITHM 6: *NoComparison()*

```

begin
  if  $\cup_{L_j \in CH(L_i)} A_y(L_j) = \emptyset$  and  $C_y(L_i) \neq \emptyset$  then
    while  $C_y(L_i) \neq \emptyset$  and  $c_y(L_i) \leq s$  do
      select a cut  $B_{max}^y(L_i) \in C_y(L_i)$  with the largest benefit and lock it at the preheader of
       $L_i$ .
       $C_y(L_i) = C_y(L_i) - B_{max}^y(L_i)$ ;
       $A_y(L_i) = A_y(L_i) \cup B_{max}^y(L_i)$ ;
       $c_y(L_i) = |A_y(L_i)| + \max_{L_j \in CH(L_i)} c_y(L_j)$ ;
    end
  end
  if  $\cup_{L_j \in CH(L_i)} A_y(L_j) \neq \emptyset$  and  $C_y(L_i) = \emptyset$  then
    while  $\cup_{L_j \in CH(L_i)} A_y(L_j) \neq \emptyset$  and  $c_y(L_i) \leq s$  do
      on the longest path of weighted DAG  $G_i$  of  $L_i$ , find a memory block  $m_t$  in
       $\cup_{L_j \in CH(L_i)} A_y(L_j)$  with the maximum accumulated execution time in  $L_i$ .
      /* move the loading point of  $m_t$  to the preheader of  $L_i$  */
      assume that the loading point of  $m_t$  is at the preheader of a child  $L_r$  of  $L_i$ ;
      /* move the loading point of  $m_t$  to the preheader of  $L_i$  */
       $A_y(L_r) = A_y(L_r) - \{m_t\}$ ;
       $A_y(L_i) = A_y(L_i) \cup \{m_t\}$ ;
       $c_y(L_r) = c_y(L_r) - 1$ ;
       $c_y(L_i) = |A_y(L_i)| + \max\{c_y(L_j) : L_j \in CH(L_i)\}$ ;
       $M_y(L_i) = M_y(L_i) \cup \{m_t\}$ ;
    end
  end
end

```

6.3. ILP-Based Algorithm

The objective of our ILP-based algorithm is the same as that of our heuristic algorithm. To minimize the WCET of L_i , we minimize the longest path length of the weighted graph G_i of L_i .

For each memory block m_k currently locked at the preheader of a child loop L_r of the loop nest L_i in the loop nest tree, we introduce a binary decision variable $X_k(L_r)$ to determine if m_k is locked at the preheader of L_r or the preheader of L_i as follows:

$$X_k(L_r) = \begin{cases} 1 & \text{if } m_k \text{ is locked at the preheader of } L_i \\ 0 & \text{if } m_k \text{ is locked at the preheader of } L_r \end{cases} \quad (4)$$

For each cut $B_v^y(L_i)$ of the loop L_i , we define a binary decision variable $Y_v^y(L_i)$ to determine whether the memory blocks in $B_v^y(L_i)$ are locked at the preheader of L_i or not:

$$Y_v^y(L_i) = \begin{cases} 1 & \text{if all the memory blocks in } B_v^y(L_i) \text{ are locked} \\ 0 & \text{if no memory block in } B_v^y(L_i) \text{ is locked} \end{cases} \quad (5)$$

For each memory block m_t of L_i , we introduce a binary variable to indicate if m_t is locked at the preheader of L_i as follows:

$$Z(m_t) = \begin{cases} 1 & \text{if } m_t \text{ is locked at the preheader of } L_i \\ 0 & \text{if } m_t \text{ is not locked} \end{cases} \quad (6)$$

If m_t is not in any cut of $C_y(L_i)$, we have the following constraint:

$$Z(m_t) = 0. \quad (7)$$

If m_t is in a cut $B_v^y(L_i)$ of $C_y(L_i)$, we have the following constraint:

$$Z(m_t) = Y_v^y(L_i). \quad (8)$$

Next, we derive all other constraints.

6.3.1. Execution Time Constraints. For each node v_s of L_i , v_s is either a preheader node, a loop node, or a basic block node. Next, we derive the WCET of v_s , denoted by $w(v_i)$, for each of the three cases.

If v_s is the preheader node of a child L_r of L_i in the loop nest tree, its weight $w(v_s)$ in G_i is calculated as follows:

$$w(v_s) = \sum_{m_k \in A_y(L_r)} (1 - X_k(L_r)) * c * I_i. \quad (9)$$

If v_s is the preheader node of L_i , its weight in G_i is formulated as follows:

$$w(v_s) = \sum_{L_r \in CH(L_i)} \sum_{m_k \in A_y(L_r)} X_k(L_r) * c + \sum_{v=1}^{\lambda} (Y_v^y(L_i) * |B_v^y(L_i)|) * c, \quad (10)$$

where $|B_v^y(L_i)|$ is the number of memory blocks in $B_v^y(L_i)$ and λ is the number of minimum node cuts of the weighted graph G_i of L_i found by Algorithm 3 for the cache set y .

If v_s is a loop node, its weight is a fixed value. If v_s is a basic block node, its weight is computed as follows:

$$w(v_s) = w_s - \sum_{m_t \in B(v_s)} Z(m_t) * \alpha(m_t), \quad (11)$$

where w_s is the execution time of the corresponding basic block of the node v_s before locking the memory blocks of v_s , $B(v_s)$ is a set of the memory blocks the corresponding basic block of v_s contains, and $\alpha(m_t)$ is the execution time reduction of the corresponding basic block of v_s after locking m_t into the I-cache.

6.3.2. Control Flow Constraints. The control flow constraints are constructed by using the weighted DAG G_i of the loop L_i . We model the upper bound of the WCET of L_i as follows:

- (1) $G'(L_i) = G(L_i)$.
- (2) Add a dummy sink node v_e and a directed edge from each of the previous sink nodes to the dummy sink node in $G'(L_i)$.
- (3) Set the weight of the dummy sink node $w(v_e)$ to 0.
- (4) Let $t(v_i)$ be the length of the longest path from the node v_i to the dummy sink node v_e in $G'(L_i)$.
- (5) For the dummy sink node v_e , $t(v_e) = 0$.
- (6) For each node $v_i \neq v_e$, we have the following constraint:

$$t(v_i) \geq \max\{t(v_j) : v_j \text{ is a child of } v_i \text{ in the DAG}\} + w(v_i). \quad (12)$$

6.3.3. Set Capacity Constraints. Given a loop L_i , we build the set capacity constraints for each set y as follows:

- (1) For each child loop L_r of L_i in the loop nest tree, the number of memory blocks that are loaded and locked at the preheader of L_r is formulated as follows:

$$|A_y(L_r)| = \sum_{m_k \in A_y(L_r)} (1 - X_k(L_r)). \quad (13)$$

Table II. The Benchmarks

Benchmark	Code Size (Bytes)	Description
cnt	2,880	Counting nonnegative numbers in the matrix
crc	5,168	Cyclic redundancy check computation on 40 bytes of data
edn	10,563	Finite impulse response filter calculations
matmult	3,737	Matrix multiplication of two 20×20 matrices
qurt	4,898	Root computation of quadratic equations
adpcm	26,852	Adaptive pulse code modulation algorithm
fir	13,881	Finite impulse response filter over a 700-item-long sample
ud	6,380	Loop nesting of three levels
select	4,494	Function to select the n -th largest number in a floating point array
qsort-exam	4,535	Nonrecursive version of the quick sort algorithm
nsichneu	118,351	Simulating an extended Petri Net
statemate	52,618	Generated by the STAtechart Real-Time-Code generator STARC

- (2) The total number of the cache lines in each set y needed to store all selected memory blocks of loops rooted at L_r is formulated as follows:

$$c_y(L_r) = |A_y(L_r)| + \max_{L_t \in CH(L_r)} c_y(L_t), \quad (14)$$

where $c_y(L_t)$ denotes the total number of the cache lines needed to store the selected memory blocks of the loop nest rooted at a child loop L_t of L_r . Note that $c_y(L_t)$ is a fixed value.

- (3) Similarly, we have the following constraints for the memory blocks that are loaded and locked at the preheader of L_i :

$$|A_y(L_i)| = \sum_{L_r \in CH(L_i)} \sum_{m_k \in A_y(L_r)} X_k(L_r) + \sum_{v=1}^{\lambda} (Y_v^y(L_i) * |B_v^y(L_i)|), \quad (15)$$

$$c_y(L_i) = |A_y(L_i)| + \max_{L_r \in CH(L_i)} c_y(L_r). \quad (16)$$

- (4) Last, we have the following set capacity constraint for the loop L_i :

$$c_y(L_i) \leq s. \quad (17)$$

6.3.4. Objective Function. Our ILP-based algorithm aims at minimizing the longest path length of G_i . Therefore, the objective function is formulated as follows:

$$\min t(v_r), \quad (18)$$

where v_r is the preheader node of the loop L_i .

7. EXPERIMENTAL RESULTS

We have implemented our approach and compared it to the partial locking approach proposed by Ding et al. [2012] and the longest path-based dynamic cache locking approach proposed by Puaut [2006].

7.1. Setup

The benchmarks, as shown in Table II, are taken from the MRTC benchmark suite [Gustafsson et al. 2010]. The SimpleScalar PISA instruction set [Burger and Austin 1997] is used to compile benchmarks. We use Chronos [Li et al. 2007] to calculate the WCET of a task. Chronos implements the approach in Li et al. [2006] to estimate the WCET of a task. It models the WCET of a task using an ILP formulation. For instruction

cache modeling, it employs a categorization-based technique proposed in Theiling et al. [2000] to classify instruction accesses. We switch off the branch prediction model of Chronos.

We use two sets of benchmarks. The first set contains 10 benchmarks with relatively smaller code sizes and loop nest sizes. For the benchmarks of the first set, we use two different associativities 2 and 4; two different cache line sizes 32B and 64B; and three different cache sizes 256B, 512B, and 1KB. The miss latency is 30 cycles, and the hit latency is 1 cycle. Since the code sizes of the benchmarks of the first set are small, we choose small cache sizes. The second set contains three benchmarks with large code sizes and loop nest sizes. For the benchmarks of the second set, the cache structure used in our experiments is similar to that of the processor PowerPC 604e [Motorola 1996]. The cache size, the associativity, the cache line size, and the miss latency are 16KB, 4, 32B, and 38 cycles, respectively, as in PowerPC 604e. In addition, we use another two different cache sizes, 4KB and 8KB, with the same cache associativity and cache line size. The loop nests of *adpcm* are much smaller than 16KB. Therefore, we artificially add an outer loop to the benchmark *adpcm*, which contains the whole *adpcm*, and set the number of iterations of the outer loop to 10.

We use MATLAB linprog as the ILP solver and run the programs for performing all experiments on an Intel i5-2430M CPU with 2.4GHz and 4GB memory. The maximum running time of our approach is 46.6 seconds for the benchmark *nsichneu*, which has the largest code size. We also evaluate the impact of our approach on the code size. The largest percentage of the increased code size is 2.7% for all benchmarks. Notice that for a benchmark, the percentage of the increased code size is nearly linearly proportional to the ratio of the number of the selected memory blocks to the total number of the memory blocks of the benchmark. All experimental results are shown in Figures 10 through 17, where each horizontal axis denotes benchmarks and each vertical axis shows the improvements of our approach.

7.2. Our Approach Versus Partial Locking

For the first benchmark set, the experimental results show that our approach achieves the average improvements of 35%, 16%, and 9% for the 256B, 512B, and 1KB 4-way set-associative caches with a cache line size of 32B, respectively. For the 256B, 512B, and 1KB 2-way set-associative caches with a cache line size of 32B, the average improvements are 31%, 12%, and 8%, respectively. For the 256B, 512B, and 1KB 2-way set-associative caches with a cache line size of 64B, the average improvements are 32%, 18%, and 9%, respectively.

For the second benchmark set, our approach achieves the average WCET improvements of 3%, 8%, and 11% for the 4KB, 8KB, and 16KB caches, respectively.

Compared to the partial locking approach, our approach performs much better for the cache with a relatively small size when the code sizes of the benchmarks are small. The reason is that in our approach, many memory blocks have disjoint live ranges and can be mapped into the same cache lines, whereas in the partial locking approach, all memory blocks have overlapping live ranges and cannot share cache lines. However, if the cache size is much smaller than the code size of a benchmark, our approach makes much lower improvements. The reason is that much fewer memory blocks can be selected to be locked into the I-cache when the cache size is much smaller than the code size of a benchmark. Thus, the execution time reduction of a benchmark is very small by using both our approach and the partial locking approach.

Our approach shows dramatic improvements for the benchmarks that contain sequentially executed nonnested loops such as *crc*. Obviously, the live ranges of those loops do not overlap. Therefore, memory blocks of different loops can share the cache in our approach. For the benchmarks containing many deeply nested loops such as

(a) cache size : 256B

(a) cache size : 256B

(b) cache size : 512B

(b) cache size : 512B

(c) cache size : 1KB

(c) cache size : 1KB

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

Fig. 11. WCET improvements of our approach over partial locking (2-way set-associative caches with 32B cache lines).

matmult, our approach shows lower improvements. The key reason is that in our approach, a lot of time is spent loading and locking the selected memory blocks.

The experimental results show that with a fixed cache line size, the improvement of our approach decreases as the associativity of the I-cache decreases. The key reason is that in our approach, fewer memory blocks can be loaded and locked into a set due to its decreased size. With a fixed associativity of the I-cache, the improvement of our approach increases as the cache line increases for the reason that our approach can lock more memory blocks into the cache due to the disjoint live ranges of locked memory blocks.

(a) cache size : 256B

(b) cache size : 512B

(c) cache size : 1KB

Fig. 12. WCET improvements of our approach over partial locking (2-way set-associative caches with 64B cache lines).

(a) cache size : 256B

(b) cache size : 512B

(c) cache size : 1KB

Fig. 13. WCET improvements of our approach over longest path-based dynamic locking (4-way set-associative caches with 32B cache lines).

Overall, there are two major reasons that our approach performs significantly better than the partial locking approach. First, our approach uses dynamic cache locking so that memory blocks with different live ranges can be loaded and locked into the same set, resulting in more efficient utilization of the cache. Second, our approach selects a better set of memory blocks as locked cache contents by using the min-cut algorithm.

7.3. Our Approach Versus Longest Path-Based Dynamic Locking

Compared to the longest path-based dynamic locking approach, the experimental results of the first benchmark set show the average improvements of 9%, 20%, and 19% for the 256B, 512B, and 1KB 4-way caches with a cache line size of 32B, respectively. For the 256B, 512B, and 1KB 2-way set-associative caches with a cache line size of 32B,

(a) cache size : 256B

(a) cache size : 256B

(b) cache size : 512B

(b) cache size : 512B

(c) cache size : 1KB

(c) cache size : 1KB

Fig. 14. WCET improvements of our approach over longest path-based dynamic locking (2-way set-associative caches with 32B cache lines).

Fig. 15. WCET improvements of our approach over longest path-based dynamic locking (2-way set-associative caches with 64B cache lines).

the improvements are 9%, 19%, and 18%, respectively. For the 256B, 512B, and 1KB 2-way set-associative caches with a cache line size of 64B, the improvements are 9%, 19%, and 17%, respectively.

For the second benchmark set, our approach achieves the average WCET improvements of 5%, 11%, and 16% for the 4KB, 8KB, and 16KB caches, respectively.

Our approach performs much better for the benchmarks that contain many deeply nested loops, such as *matmult*. The reason is that our approach moves the loading points of many memory blocks to the preheaders of outer loops to further reduce the execution frequencies of those *lock* instructions. For such benchmarks, the improvement increases as the cache size increases. The reason is that more selected memory

Fig. 16. WCET improvements of our approach over partial locking (4-way set-associative caches with sizes of 4KB, 8KB, 16KB, and 32B cache lines).

Fig. 17. WCET improvements of our approach over longest path-based dynamic locking (4-way set-associative caches with sizes of 4KB, 8KB, 16KB, and 32B cache lines).

blocks can be locked at the preheaders of outer loops. For the benchmarks that contain many sequentially executed nonnest loops such as *crc*, our approach shows lower improvements. The key reason is that each *lock* instruction is executed only once and the time on loading and locking instruction is nearly the same as the longest path-based locking approach.

When the cache size is much smaller than the code size of a benchmark, our approach makes lower improvements. The reason is that the WCET reduction is much smaller by using both our approach and the longest path-based approach when much fewer memory blocks can be locked into the cache.

On average, the results show similar improvement for different associativities and cache line sizes. With a fixed cache line size, the improvement of our approach slightly decreases as the associativity of the I-cache decreases for the reason that fewer memory blocks can be loaded and locked into a set in our approach. With a fixed associativity of the I-cache, the improvements remain similar.

Overall, our approach has two major advantages over the longest path-based dynamic locking approach. First, our approach selects a better set of memory blocks by using the min-cut algorithm. Second, our approach finds better loading points for the selected memory blocks.

7.4. ILP-Based Algorithm Versus Heuristic Algorithm

We have compared our heuristic algorithm and the ILP-based algorithm for finding a good loading point for each selected memory block. As the results show in Figures 10 through 15, our heuristic algorithm achieves nearly the same performance as the ILP-based algorithm. For the benchmarks that contain large nested loops such as *qsort - exam*, our ILP-based algorithm shows slightly better performance. For the benchmarks that only contain nonnested loops such as *qurt*, both algorithms show nearly the same performance.

8. 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 a min-cut-based dynamic cache locking approach. Our approach considers the live ranges of memory blocks. Two memory blocks with disjoint live ranges can share the cache space, resulting in higher cache utilization. Our approach finds minimum cuts of the weighted DAG of each loop that contains not only the longest path but also the paths whose lengths are close to the longest path length. Therefore, our approach makes more efficient use of the I-cache

than the previous approaches. Furthermore, we propose a heuristic algorithm and an ILP-based algorithm to find a good loading point for each selected memory block. The experimental results show that our approach achieves the average improvements of 33%, 15%, 9%, 3%, 8%, and 11% over the partial locking approach proposed in Ding et al. [2012] for the 256B, 512B, 1KB, 4KB, 8KB, and 16KB caches, respectively, and 9%, 19%, 18%, 5%, 11%, and 16% over the dynamic locking approach proposed in Puaut [2006] for the 256B, 512B, 1KB, 4KB, 8KB, and 16KB caches, respectively.

A typical embedded system consists of a set of tasks with various constraints, such as timing constraints. Constructing a feasible schedule satisfying all constraints is a primary goal at the design stage. Our future work is to integrate task scheduling and cache locking for multiprocessor-based embedded systems with two levels of caches. The major challenge is that the problem of selecting locked cache contents and the task scheduling problem are intertwined. On the one hand, we need to know the WCET of each task to construct a feasible schedule. On the other hand, we need to know how much cache space is allocated to each task to calculate its WCET. Given a schedule, for any two tasks, if their lifetimes overlap, they cannot share any space in both L1 caches and the L2 cache. Otherwise, they can share any space in both L1 caches and the L2 cache. 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. Given two tasks assigned to one processor, either the lifetime of one task contains the lifetime of the other task or the lifetimes of the two tasks are disjoint. This property leads to a polynomial-time optimal cache partitioning algorithm for all tasks sharing L1 caches. However, this property may not hold for two tasks that are assigned to two different processors. Since the L2 cache is shared by all tasks, the L2 cache partitioning problem becomes more complicated.

ACKNOWLEDGMENTS

We thank all reviewers for their insightful comments, which helped improve the quality of this article significantly.

REFERENCES

- Kapil Anand and Rajeev Barua. 2009. Instruction cache locking inside a binary rewriter. In *Proceedings of the Conference on Compilers, Architecture, and Synthesis for Embedded Systems*. ACM, New York, NY, 185–194.
- Kapil Anand and Rajeev Barua. 2015. Instruction-cache locking for improving embedded systems performance. *ACM Transactions on Embedded Computing Systems* 14, 3, Article No. 53.
- Doug Burger and Todd M. Austin. 1997. The SimpleScalar tool set, version 2.0. *ACM SIGARCH Computer Architecture News* 25, 3, 13–25.
- J. V. Busquets-Mataix, J. J. Serrano-Martin, R. Ors-Carot, P. Gil, and A. Wellings. 1996. Adding instruction cache effect to an exact schedulability analysis of preemptive real-time systems. In *Proceedings of the 8th Euromicro Workshop on Real-Time Systems*. IEEE, Los Alamitos, CA, 271–276.
- A. M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. 2001. Using genetic algorithms in content selection for locking-caches. In *Proceedings of the IASTED International Symposium on Applied Informatics*. 271–276.
- A. M. Campoy, A. P. Ivars, and J. V. Busquets-Mataix. 2002. Dynamic use of locking caches in multitask, preemptive real-time systems. *IFAC Proceedings Volumes* 35, 1, 411–416.
- A. M. Campoy, A. Perles, F. Rodriguez, and J. V. Busquets-Mataix. 2003. Static use of locking caches vs. dynamic use of locking caches for real-time systems. In *Proceedings of the Canadian Conference on Electrical and Computer Engineering (CCECE'03)*, Vol. 2. 1283–1286.
- Huping Ding, Yun Liang, and Tulika Mitra. 2012. WCET-centric partial instruction cache locking. In *Proceedings of the Design Automation Conference (DAC'12)*. IEEE, Los Alamitos, CA, 412–420.
- Huping Ding, Yun Liang, and Tulika Mitra. 2014. WCET-centric dynamic instruction cache locking. In *Proceedings of the 2014 Design, Automation, and Test in Europe Conference and Exhibition (DATE'14)*. 1–6.

- Heiko Falk, Sascha Plazar, and Henrik Theiling. 2007. Compile-time decided instruction cache locking using worst-case execution paths. In *Proceedings of the 5th IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS'07)*. ACM, New York, NY, 143–148.
- Jan Gustafsson, Adam Betts, Andreas Ermedahl, and Björn Lisper. 2010. The Mälardalen WCET benchmarks: Past, present and future. In *Proceedings of the 10th International Workshop on Worst-Case Execution Time Analysis (WCET'10)*. 136–146.
- Xianfeng Li, Yun Liang, Tulika Mitra, and Abhik Roychoudhury. 2007. Chronos: A timing analyzer for embedded software. *Science of Computer Programming* 69, 1–3, 56–67.
- Xianfeng Li, Abhik Roychoudhury, and Tulika Mitra. 2006. Modeling out-of-order processors for WCET analysis. *Real-Time Systems* 34, 3, 195–227.
- Yun Liang and Tulika Mitra. 2010. Instruction cache locking using temporal reuse profile. In *Proceedings of the 47th Design Automation Conference (DAC'10)*. ACM, New York, NY, 344–349.
- Tiantian Liu, Minming Li, and Chun Jason Xue. 2009. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *Proceedings of the 15th IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS'09)*. IEEE, Los Alamitos, CA, 35–44.
- Tiantian Liu, Minming Li, and Chun Jason Xue. 2012. Instruction cache locking for embedded systems using probability profile. *Journal of Signal Processing Systems* 69, 2, 173–188.
- Tiantian Liu, Yingchao Zhao, Minming Li, and Chun Jason Xue. 2010. Task assignment with cache partitioning and locking for WCET minimization on MPSoC. In *Proceedings of the 2010 39th International Conference on Parallel Processing (ICPP'10)*. IEEE, Los Alamitos, CA, 573–582.
- Motorola. 1996. PowerPC 604e RISC Microprocessor Technical Summary. Retrieved February 10, 2017, from <http://www.nxp.com/assets/documents/data/en/data-sheets/MPC604E.pdf>.
- Sascha Plazar, Jan C. Kleinsorge, Peter Marwedel, and Heiko Falk. 2012. WCET-aware static locking of instruction caches. In *Proceedings of the Symposium on Code Generation and Optimization*. 44–52.
- Isabelle Puaut. 2006. WCET-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the 2006 18th Euromicro Conference on Real-Time Systems*. IEEE, Los Alamitos, CA, 217–226.
- Isabelle Puaut and Alexis Arnaud. 2006. Dynamic instruction cache locking in hard real-time systems. In *Proceedings of the 14th International Conference on Real-Time and Network Systems*.
- Isabelle Puaut and David Decotigny. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *Proceedings of the Real-Time Systems Symposium*. IEEE, Los Alamitos, CA, 114–123.
- Keni Qiu, Mengying Zhao, Chun Jason Xue, and Alex Orailoglu. 2014. Branch prediction-directed dynamic instruction cache locking for embedded systems. *ACM Transactions on Embedded Computing Systems* 13, 5s, Article No. 156.
- Steven S. Skiena. 1998. *The Algorithm Design Manual*. Springer.
- Mechthild Stoer and Frank Wagner. 1997. A simple min-cut algorithm. *Journal of the ACM* 44, 4, 585–591.
- Henrik Theiling, Christian Ferdinand, and Reinhard Wilhelm. 2000. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems* 18, 2–3, 157–179.
- Xavier Vera, Björn Lisper, and Jingling Xue. 2003. Data cache locking for higher program predictability. *ACM SIGMETRICS Performance Evaluation Review* 31, 1, 272–282.
- Wenguang Zheng and Hui Wu. 2014. WCET-aware dynamic instruction cache locking. In *Proceedings of the Conference on Languages, Compilers, and Tools for Embedded Systems*. 53–62.
- Wenguang Zheng and Hui Wu. 2015. WCET-aware dynamic D-cache locking for a single task. In *Proceedings of the Conference on Languages, Compilers and Tools for Embedded Systems*. Article No. 8.
- Wenguang Zheng and Hui Wu. 2017. Dynamic data-cache locking for minimizing the WCET of a single task. *ACM Transactions on Embedded Computing Systems* 16, 2, Article No. 31.

Received May 2016; revised November 2016; accepted December 2016