Abstract
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 WCET (Worst Case Execution Time) of a task. To alleviate this problem, cache locking has been proposed. We investigate the I-cache locking problem, and propose a WCET-aware, min-cut based dynamic instruction cache locking approach for reducing the WCET of a single task. We have implemented our approach and compared it with the two state-of-the-art cache locking approaches by using a set of benchmarks from the MRTC benchmark suite. The experimental results show that our approach achieves the average improvements of 41%, 15% and 7% over the partial locking approach for the 256B, 512B and 1KB caches, respectively, and 7%, 18% and 17% over the longest path based dynamic locking approach for the 256B, 512B and 1KB caches, respectively.

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

Keywords Worst-case execution time; dynamic instruction cache locking; minimum cut

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. Caches are managed by hardware. For each memory access, it is difficult to know at compile time if the corresponding data or instruction is already in the cache. Therefore, caches make it significantly harder to compute the WCET of a task. Cache locking is an effective technique to remedy the unpredictability of caches. If an instruction or data is locked into the cache, it will not be replaced. The access time of each locked instruction or data is always one cycle. As a result, cache locking makes it significantly easier to compute the WCET of a task. Furthermore, cache locking can reduce the WCET of a task as we can lock the contents on the longest paths to reduce the access times of those contents.

There are two types of cache locking, static cache locking and dynamic cache locking. Static cache locking assumes that the live ranges of the locked contents span the entire task. Therefore, the locked contents of a task remain locked during the entire execution of the task. As a result, the contents mapped to the same location in a cache cannot be locked simultaneously, resulting in low cache utilization. Dynamic cache locking considers the live ranges of the locked contents. Different contents can be locked into the same location of a cache as long as their live ranges do not overlap, leading to more efficient cache utilization than static cache locking.

For static cache locking, the major task is to select the contents to be locked into a cache. For dynamic cache locking, there are two additional tasks, determining when to load and lock the selected contents, and when to unlock them. Typically, the locked contents can be unlocked when their live ranges end. To reduce the WCET of a task, we need to select the contents on the longest path to be locked into the cache. The longest path may change after some contents on the longest path are locked into the cache. For this reason, the previous cache locking approaches iteratively select the contents on the longest path to be locked into the cache. However, selecting contents on a single longest path may not minimize the WCET of a task.

In this paper, we investigate the WCET-aware, dynamic I-cache locking problem for a single task. The unit of contents for locking is a memory block entirely mapped to a cache line. Our objective is to select a set of memory blocks of a task and find a loading point for each memory block selected such that the task’s WCET is minimized after the memory blocks selected are loaded and locked at the loading points. We make the following major contributions.

1. We propose a min-cut based, dynamic cache locking approach. Unlike all the previous cache locking approaches that consider the longest path only, our approach considers a subgraph that contains not only the longest path, but also all the paths whose lengths are close to that of the longest path, and select a minimum set of memory blocks of instructions to be locked into the cache.

2. We propose an efficient technique for determining a good loading point for each memory block selected as the locked cache contents, further reducing the WCET of a task.

3. We have implemented our approach and compared it with the two state-of-the-art cache locking approaches, the partial locking approach proposed in [6] and the longest path based dynamic locking approach proposed in [15], by using a set of benchmarks from the MRTC benchmark suite [8]. The experimental results show that our approach achieves the average improvements of 41%, 15% and 7% over the partial locking approach for the 256B, 512B and 1KB caches, respectively, and 7%, 18% and 17% over the longest path based dynamic locking approach for the 256B, 512B and 1KB caches, respectively.

The rest of this paper is organized as follow. Section 2 gives a brief survey of the related work. Section 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 single loop as locked I-cache contents. Section 6 presents our approach to loop nests. Section 7 shows the experimental results and analyses, and Section 8 concludes this paper.

2. Related Work

Cache locking problems have been extensively studied, and various approaches have been proposed. Many cache locking approaches have been proposed to reduce the WCET or the ACET (Average Case Execution Time) of a single task. [13] proposes instruction cache locking algorithms that aim at minimizing the ACET of a task. It uses a PEFT (Probability Execution Flow Tree) to model a task. [1] proposes an approach to reduce the ACET of a task. A cost-benefit model is constructed to determine which memory addresses should be locked in order to reduce the ACET. [10] uses the temporal reuse profile to construct the cost-benefit model, and locks the memory blocks with the largest miss rate to reduce the ACET.

[19] studies the D-cache locking problem. It combines D-cache locking with the static cache analysis to estimate the precise worst cache memory performance. [7] proposes a static I-cache locking approach that aims at minimizing the WCET by iteratively reducing the longest path length. It uses an EFG (Execution Flow Graph) to model the possible execution paths, and selects the instructions on the current longest path with the largest benefit to be locked in the cache. [11] studies the static I-cache locking problem. It formulates the problem using an EFT (Execution Flow Tree) and a linear programming model. For a subset of the problems with certain properties, it proposes polynomial time optimal algorithms. Furthermore, it proves that the general problem is NP-Hard. [14] proposes an ILP (Integer Linear Programming) based, static I-cache locking algorithm for minimizing the WCET of a single task. [6] points out that full cache locking may cause more cache misses which would have a negative effect on the WCET reduction. It proposes a partial I-cache locking mechanism to lock a part of I-cache. [16] proposes a dynamic I-cache locking approach. It partitions a program into a set of regions. For each regions, there is a loading point. [15] 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 pre-header of loops to determine the whether this loop pre-header should be selected as a loading point. However, the cost function is not clearly defined. [5] compares static and dynamic cache locking using genetic algorithm. It proves that the static cache locking is more predictable, and dynamic cache locking shows better improvement in most cases.

A number of approaches have been proposed to integrate task scheduling and cache locking. [17] studies static I-cache locking for multitask real-time systems. It proposes two algorithms, one aiming at minimizing the CPU utilization and the other attempting to minimize the interferences between tasks. [3] proposes a genetic algorithm for the problem of selecting instructions to be locked into the I-cache to reduce the response time of multitasks. [12] combines cache locking with task assignment to reduce the WCETs of a set of tasks on a multiprocessor system with two levels of caches. It applies cache locking to both I-cache and D-cache by using the algorithms proposed in [11] and [19] to reduce the WCET for each task. Then, it optimizes the task assignment considering the locked cache size. [4] proposes a dynamic I-cache locking approach for multitask systems. It uses the response time analysis approach proposed in [18] 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.

3. A Motivational Example

In this section, we use an example to compare the state-of-the-art approaches to the WCET-aware I-cache locking problem and illustrate the key ideas of our approach. Consider a task whose CFG is shown in Figure 1. The task consists of three loops, loop A and loop B are nested in loop C. The numbers of iterations of loop A, loop B and loop C are 4, 3, 10, respectively. For simplicity, we make the following assumptions.

1. The I-cache is fully associative and has 4 cache lines which can store 4 memory blocks.
2. Each basic block is exactly one memory block.
3. If a basic block is in the I-cache, its execution time is 1. Otherwise, it is 30.

3.1 Static Full Cache Locking

Consider the approach proposed in [7] which iteratively finds the longest path and selects the memory block with the largest benefit on the longest path as the locked cache contents. Before cache locking, there are four longest paths. Obviously, the nodes of loop A have larger benefits because of its larger number of iterations. The four basic blocks $m_0$, $m_1$, $m_3$, $m_5$ may be subsequently selected as the locked cache contents, resulting in a WCET of $120 + (62 * 4 + 90 * 3) * 10 = 5300$.

3.2 Longest Path Based Dynamic Cache Locking

Using the longest path based dynamic cache locking approach proposed in [15], the selected contents are loaded and locked at the pre-headers of loop A and loop B. For loop A, $m_0$, $m_1$, $m_3$, $m_5$ may be subsequently selected as the locked cache contents as they have the largest frequencies on the current longest paths. When loop A terminates, all the cache lines occupied by its memory blocks can be released. So, the entire loop B can be locked. The memory blocks $m_0$, $m_1$, $m_3$, $m_5$ will be fetched and locked 10 times at the pre-header of loop A, and the three memory blocks of loop B will also be fetched and locked 10 times at the pre-header of loop B. The times taken to fetch and lock the selected memory blocks of loop A and loop B are 30*4+10=120 and 30*3+10=900, respectively. Therefore the WCET of the task is $(62*4+30*4)*10+3*3+3*30)*10 = 4670$.

3.3 Partial Cache Locking

Using the partial cache locking approach proposed in [6], $m_0$ and $m_5$ may be selected as the locked cache contents. The other two cache lines are kept free so that the memory accesses to $m_2$ and $m_8$ will be hit after the first cache misses. For loop A, each iteration will cause 3 cache misses. For loop B, there are only two cache
misses during the execution of the task. Since only two memory blocks are locked, the time spent on fetching and locking these two memory blocks is $2 \times 30 = 60$. Hence, the WCET of the task is $60 + (91 \times 4 + (3 \times 2 + (1 + 30 + 30)) \times 10) = 4370$.

3.4 Optimal Solution

In order to find the optimal solution, we need to use a dynamic approach so that after loop A terminates, all the cache lines occupied by it can be released. This is done by loading and locking all the selected memory blocks at the preheaders of loops A and B. Furthermore, we need to select a minimum number of basic blocks as the locked cache contents. In this case, the minimum number of basic blocks can be computed by finding three minimum node cuts of the control flow graphs of loop A and loop B without back edges. For loop A, the three minimum node cuts are either $(m_0, \{m_1, m_2\}, \{m_3\})$ or $(m_0, \{m_3, m_4\}, \{m_5\})$. For loop B, all the three basic blocks can be locked. Therefore, the WCET of the optimal solution is $(33 \times 4 + 30 \times 4 + 3 \times 3 + 30 \times 3) = 3510$.

4. System Model and Definitions

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 the following parameters, the cache line size l, the associativity n, and the cache size s. Therefore, the number of sets is equal to s/l. Each set has a set number between 0 and k − 1, where k is equal to s/l. We do not consider the D-cache locking problem. If an instruction is in the I-cache, it takes one cycle to fetch the instruction. Otherwise, it takes w cycles. The unit for locking is a cache line.

The main memory is partitioned into memory blocks such that each memory block is mapped to exactly one cache line. Thus, each basic block of a task is mapped to one or more sets of the I-cache.

A task is represented by the weighted CFG (Control Flow Graph) where each node denotes a basic block, each node weight is the execution time of the corresponding basic block, and each edge represents the control dependency between the two nodes. We assume that each path of a CFG is feasible.

Given a loop, a weighted DAG is constructed by removing all the back edges from the CFG of the loop. Given a weighted DAG and a path P, the length of P is the sum of the weights of all the nodes on P.

Given a weighted DAG G and an integer x, the x-spanning graph G(x) is a subgraph of G where the length of each path is larger than x. The x-spanning graph of a weighted DAG G can be computed in $O(e)$ time as shown in Algorithm 1, where e is the number of edges in G.

Given an x-spanning graph G(x), the y-projection graph G(x, y) is a subgraph of G(x) satisfying the following constraint:

- For each node of G(x, y), its corresponding basic block has a memory block mapped to the set y and the memory block has not been selected as the 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).

For each loop nest, we define a loop nest tree as follows. A loop nest is a tree where each node denotes a loop, and each edge $(v_i, v_j)$ denotes that $v_j$ is immediately nested in $v_i$.

5. I-Cache Locking for A Single Loop

Before presenting our approach to dynamic I-cache locking for a whole task, we consider a single loop and propose a min-cut based...
Given a single loop $L$, our algorithm aims at finding a minimum set of memory blocks of the loop such that the WCET of the loop is minimized after the set of memory blocks are locked into the I-cache. Our algorithm works as follows.

1. Create a weighted DAG $G'$ by removing the back edges of the CFG of the loop.
2. For each set $y$, find a minimum set of memory blocks of the loop that are mapped to the set $y$, and select them as the locked cache contents.

Let $c$ be the time taken to fetch one memory block from the off-chip memory. To find the minimum set of memory blocks for each set $y$, our algorithm repeats the following steps until the set $y$ has no enough free space for the selected memory blocks.

1. Find the longest path length $l_{\text{max}}$ of $G'$
2. Construct the $x$-spanning graph of $G'$, where $x$ is $l_{\text{max}} - c$.
3. Construct the $y$-projection graph of the $x$-spanning graph.
4. Find a minimum node cut $C$ of the $y$-projection graph.
5. If the size of $C$ is not less than the number of iterations of the loop, discard $C$. Otherwise, do the following:
   - If the number of free cache lines of set $y$ is not less than the size of $C$, for each basic block $B_i$ in $C$, select a memory block $M_i$ with the largest execution time among all the memory blocks of $B_i$ that have not been selected as the locked cache contents, recalculate the execution times of $B_i$, and all the basic blocks affected, and update their corresponding node weights in the $y$-projection graph.

The details of our algorithm are shown in pseudo code in Algorithm 3. Next, we use an example to show how our algorithm works. Consider a DAG $G$ shown in Figure 3 that represents a single iteration of a loop. For ease of descriptions, we assume that there are two sets in the I-cache, each set has two cache lines, and each node (basic block) is exactly one memory block of instructions. The execution time of each basic block is 1 if it is locked into the I-cache. Otherwise, it is 30. All the circle nodes are mapped to set 0, and all the square nodes are mapped to set 1. Now, we show how our algorithm selects locked cache contents for set 0. Based on the assumptions, the longest path length of the DAG $G$ is 180, and $x$ is set to 150. In the first iteration, $G(150)$, the 150-spanning graph, is constructed as shown in Figure 4. Based on $G(150)$, the $0$-projection graph $G'(150,0)$ of $G(150)$ is constructed as shown in Figure 5. In $G(150,0), \{v_0\}$ is a minimum node cut. Thus, $v_0$ is selected as the locked cache contents and its weight is changed from 30 to 1. Now, the longest path length of $G$ becomes 151. In the second iteration, the 121-spanning graph, $G(121)$, is constructed as shown in Figure 6. The $0$-projection graph of $G(121)$ is constructed as shown in Figure 7. A minimum node cut of $G(121,0)$ is $\{v_1\}$. So, $v_1$ is selected as the locked cache contents and its weight is changed from 30 to 1 as shown in Figure 8. Now, the longest path length of the DAG $G$ is reduced to 122. The same selection process is applied on square nodes to find the locked cache contents for set 1.

### 6. Cache Locking for Loop Nests

In this section, we present our approach to the WCET aware, dynamic I-cache locking problem for a whole task. Our approach ignores all the basic blocks that are not inside a loop, and considers individual loop nests. For each loop nest, all its memory blocks selected as the cache contents are loaded and locked into the I-cache at the preheaders of the loops of the loop nest. Hence, the live range of each basic block of a loop nest does not go beyond the loop nest. After the live range of a memory block ends, the cache line storing this memory block can be reused.

We introduce a special instruction `fetch` for loading contiguous memory blocks into the I-cache and locking them. When a cache line is already locked, `fetch` can still load a memory block into it. Therefore, no `unlock` instruction is needed. In order not to disrupt 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 which is placed at the end of the task. The loading procedure contains a sequence of `fetch` instructions for loading the memory blocks selected into the I-cache and locking them. The program point of a call to a loading procedure is called the loading point of each memory block to be loaded and locked into the I-cache by the loading procedure. Init-
Initially, each loading procedure is empty. After applying our I-cache locking algorithm, a sequence of fetch instructions will be added to each loading procedure. If no fetch instruction is added to a loading procedure, the corresponding call will be replaced by a nop instruction.

Our approach processes each loop nest sequentially. For each loop nest, all its loops are processed in reverse topological order of the loop nest tree, i.e., starting with an inner-most loop and working toward the out-most loop. For a leaf loop in the loop nest tree, our approach applies Algorithm 3 to select a set of memory blocks of the loop as the locked cache contents. Then, the entire loop reduces into two blocks, a preheader block and a loop block. The preheader block contains a call to the loading procedure for this loop, and its weight is the execution time of the loading procedure plus the execution time of the call. The loop block represents the entire loop and its weight is the worst case execution time of the loop after locking all the selected memory blocks into the cache. For a non-leaf loop in the loop nest tree, our approach finds a good loading point for each loop to reduce the WCET as much as possible. Initially, the loading point of all the selected memory blocks of each loop is at its preheader. Our approach may move the loading points of some selected memory blocks of a loop to the preheader of one of its outer loops in order to decrease the WCET of the loop nest. Notice that after moving the loading point of a selected memory block to the preheader of an outer loop, its live range will increase. Therefore, a loading point motion typically occurs when the I-cache has enough space to store the selected memory blocks of a loop and its inner loops. After moving the loading points for a set of selected memory blocks, our approach applies Algorithm 3 to select a set of memory blocks of the loop as the locked cache contents providing that the cache still has free space. Then, the entire loop reduces into two blocks, a preheader block and a loop block as before.

To facilitate descriptions, we introduce the following notation.

- $S(L_i)$: a set of the selected memory blocks which are fetched and locked at the preheader of $L_i$.

Given a loop nest $L$, our approach works as follows.

1. Construct the loop nest tree of $L$.
2. For each loop $L_i$ in reverse topological order of the loop nest tree, do the following.
   - If $L_i$ is a leaf node in the loop nest tree, perform the following tasks.
ALGORITHM 4: loopNestLocking($L, k, s$)

\textbf{input :} $L$: a loop nest  
\hspace{1cm} $k$: the number of sets in the I-cache  
\hspace{1cm} $s$: the set size  

\textbf{output:} A set of memory blocks selected as the locked cache contents and a set of loading points  

\textbf{var} $n[i]:$ an array where $n[i](i = 0, 1, \cdots, k - 1)$ stores the number of cache lines available in the set $i$  

\begin{itemize}
  \item[construct a loop nest tree $T$ of $L_i$]
  \item[\textbf{for each loop }$L_i$, in reverse topological order of $T$ \textbf{do}]
    \begin{itemize}
      \item[\textbf{if } $L_i$ \textbf{is a leaf node in } $T$ \textbf{then}]
        \begin{itemize}
          \item[\textbf{for } $j = 0, 1, \cdots, k - 1$ \textbf{do}]
            \begin{itemize}
              \item[$n[j] = s$;]
              \item[$A = \text{SingleLoopLocking}(L_i, k, n)$;]
              \item[$S(L_i) = \bigcup_{i=0}^{k-1} A[j]$;]
              \item[shrink the subgraph 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;]
            \end{itemize}
        \end{itemize}
      \item[else]
        \begin{itemize}
          \item[construct the DAG $G(L_i)$;]
          \item[$S(L_i) = \emptyset$;]
          \item[\textbf{for each set }$y(y = 0, 1, \cdots, k - 1)$ \textbf{do}]
            \begin{itemize}
              \item[construct the $y$-projection graph $G_y(L_i)$;]
              \item[find the maximum size $s_{max}$ of the space in the set $y$ used by the selected memory blocks of any child of $L_i$ in $T$;]
              \item[$s_y = s = s_{max}$;]
              \item[\text{* Move loading points from the preheaders of the inner loops of $L_i$ into the preheader of $L_i$ to further reduce the WCET of $L_i$ */}
            \end{itemize}
          \end{itemize}
        \end{itemize}
      \end{itemize}
    \end{itemize}
  \item[\textbf{while } $s_y \geq 0$ \textbf{do}]
    \begin{itemize}
      \item[find the longest path in $G_y(L_i)$;]
      \item[on the longest path, find a selected memory block $B$ with the maximum accumulated execution time that is mapped to the set $y$ and in $\{S(L_i): L_i$ is a child of $L_i$ in $T\}$;]
      \item[assume $B$ is in $S(L_i)$;]
      \item[$S(L_i) = S(L_i) - \{B\}$;]
      \item[$S(L_i) = S(L_i) \cup \{B\}$;]
      \item[update the weights of the nodes affected;]
      \item[find the new maximum size $s_{max}$ of the space in the set $y$ used by any child of $L_i$ in $T$;]
      \item[\textbf{if } $s_{max} = s_{max}$ \textbf{then}]
        \begin{itemize}
          \item[$s_y = s_y - 1$;]
          \item[$s_{max} = s_{max}$;]
        \end{itemize}
      \item[\textbf{for } $j = 0, 1, \cdots, k - 1$ \textbf{do}]
        \begin{itemize}
          \item[$n[j] = s_y$;]
          \item[$A = \text{SingleLoopLocking}(L_i, k, n)$;]
          \item[$S(L_i) = S(L_i) \cup \bigcup_{j=0}^{k-1} A[j]$;]
          \item[shrink the subgraph 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;]
        \end{itemize}
    \end{itemize}
  \end{itemize}
\end{itemize}

\textbf{for each }$L_i$ \textbf{in the loop nest do}
  \begin{itemize}
    \item[add fetch instructions to the loading procedure of $L_i$ to load and lock the selected memory blocks into the I-cache;]
  \end{itemize}

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{loopNestLocking}
\caption{A loop nest}
\end{figure}

\begin{figure}[h]
\centering
\includegraphics[width=0.2\textwidth]{loopNestTree}
\caption{The loop nest tree of the loop nest in Figure 9}
\end{figure}

(a) Select a set of the memory blocks of $L_i$ as the locked cache contents by using Algorithm 3.

(b) Shrink the subgraph representing $L_i$ in the CFG of the loop nest 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.

Otherwise, do the following.

(a) Construct the weighted DAG $G(L_i)$ of the loop $L_i$.

(b) For each set $y(y = 0, 1, \cdots, k - 1)$, do the following.
  \begin{itemize}
    \item[i.] Construct the $y$-projection graph $G_y(L_i)$ of $G(L_i)$.
    \item[ii.] Keep finding a selected memory block with the highest impact on the WCET of $L_i$ and move its loading point to the preheader of $L_i$ until the set $y$ has no more free space as follows.
    \item[A.] Find the longest path of $G_y(L_i)$.
    \item[B.] On the longest path, find a selected memory block $B$ with the maximum accumulated execution time in $L_i$ among all the selected memory blocks that are mapped to the set $y$ and loaded at the preheaders of the children of $L_i$ in the loop nest tree. The accumulated execution time of a memory block is its execution time multiplied by its total number of executions in $L_i$.
    \item[C.] Move the loading point of $B$ to the preheader of $L_i$, update the weights of the nodes affected in $G_y(L_i)$, and adjust the size of the free space of the set $y$.
    \item[(c)] Select a set of the memory blocks of $L_i$ as the locked cache contents by using Algorithm 3.
    \item[(d)] Shrink the subgraph 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.
The details of our algorithm are shown in Algorithm 4. Next we use an example to show our approach works. Consider the loop nest shown in Figure 9. Its loop nest tree is shown in Figure 10. For ease of descriptions, we make the following assumptions. 1) The number of sets of the I-cache is one. 2) The associativity n is 8. 3) Each basic block is mapped to exactly one memory block. 4) If a basic block is in the I-cache, its execution time is 1. Otherwise, it is 30. Our approach starts with L4. It will select all the 4 basic blocks m7, m8, m9, m10 as the locked cache contents and put their loading points at the preheader of L4. When processing L3, our approach will move the loading points of m7, m8, m9, m10 to the preheader of L3. Then, it will select m6 and m11 as locked contents of L3. Similarly, our approach will select m2, m3, m4 and m5 of L2 and put their loading points at the preheader of L2. The selection results of locked contents for L2 and L3 are shown in Figure 13. When processing L1, our approach will move the loading points of m4, m5, m7, m8, m9, and m10 to the preheader of L1 as shown in Figure 14. All the three locking points are shown in Figure 12, where each locking point is a call to its corresponding locking procedure that contains the instructions for fetching and locking all the memory blocks selected. Notice that after the locking points of m4, m5, m7, m8, m9, and m10 are moved to the preheader of L1, their live ranges extend to entire L1, overlapping with the live ranges of m2, m3, m6 and m11.

7. Experimental Results

We have implemented our approach and compared it with the two state-of-the-art approaches, the partial locking approach proposed by Ding et al. [6] and the longest path based dynamic cache locking approach proposed by Puaut [15].

7.1 Setup

The benchmarks, as shown in Table 1, are taken from the MRTC benchmark suite [8]. SimpleScalar PISA instruction set [2] is used to compile benchmarks. We use Chronos [9], a WCET analysis tool, to compute the WCETs. Since we focus on the instruction cache, we switch off the pipeline model and branch prediction. We use two different associativities 2 and 4, two different cache line sizes 32B and 64B, 3 different cache sizes 256B, 512B and 1KB. The miss latency is 30 cycles and the hit latency is 1 cycle. For each benchmark, we use different cache configurations to compute its WCETs by using our approach and its WCETs by using the partial locking approach and the longest path based dynamic cache locking approach proposed by Puaut [15].

7.2 Our Approach vs. Partial Locking

The improvements of our approach over the partial locking approach are shown in Figures 15-17 where the horizontal axis and the vertical axis denote the benchmarks and the improvements of our approach over the partial locking approach, respectively.
For the 4-way set associative cache with a cache line size of 32B, the average improvements of our approach are 41%, 15% and 7% for the 256B cache, the 512B cache and the 1KB cache, respectively. For the 2-way set associative cache with a cache line size of 32B, the improvements are 37%, 9% and 7% for the 256B cache, the 512B cache and the 1KB cache, respectively. The improvement for the 2-way set associative cache with a cache line size of 64B are 39%, 17% and 7% for the 256B cache, the 512B cache and the 1KB cache, respectively.

Compared with the partial locking approach, our approach performs much better for the small caches. The reason is as follows. In our approach, many memory blocks have disjoint live ranges and thus can be locked to the same cache lines. In contrast, in the partial locking approach, all the memory blocks have overlapping live ranges and consequently cannot share cache lines.

On average, the improvement of our approach over the partial locking approach decreases as the cache size increases, especially for some benchmarks with a small code size, such as cnt. The key reason is that more cache lines are available so that more memory blocks can be loaded and locked into the same cache lines. Furthermore, with a fixed associativity, the improvement of our approach increases as the cache line size increases for most benchmarks.

Our approach achieves dramatic improvements for the benchmarks which contain many sequentially executed, non-nested loops such as edn. Obviously, the live ranges of those loops do not overlap. Therefore, the memory blocks of those loops can share cache lines.

For the benchmarks contained many deeply nested loops such as malmult, our approach makes less improvement. The key reason is that in our approach, a lot of time is spent on loading and locking the selected memory blocks.

The experimental results also 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.

Furthermore, with a fixed associativity, the improvement of our approach increases as the cache line size increases for most benchmarks.

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 disjoint live ranges can be loaded and locked into the same cache line, resulting in more efficient utilization of the I-cache. Secondly, our approach selects a minimum number of memory blocks by using the min-cut algorithm.

7.3 Our Approach vs. Longest Path Based Dynamic Locking

The improvements of our approach over the longest path based dynamic locking approach are shown in Figures 18-20. Our approach achieves the average improvements of 7%, 18% and 17% for the 4 way set associative cache with a size of 256B, 512B, and 1KB, respectively. For the 2-way set associative cache with a cache line size of 32B, the improvements are 7%, 16% and 16% for 256B, 512B and 1KB, respectively. The improvements for the 2-way set associative cache with a cache line size of 64B are 7%, 15% and 16% for 256B, 512B and 1KB, respectively.

Our approach performs much better for the benchmarks containing many deeply nested loops such as malmult. The reason is that our approach moves some loading points to the preheaders of outer loops to reduce the the execution frequency of the fetch instructions. For such benchmarks, the improvement increases as the cache size increases. The reason is that more selected memory blocks can be loaded and locked at the preheaders of outer loops.

For the benchmarks such as edn which contain many sequentially executed small non-nested loops, our approach achieves less improvements. The key reason is that each fetch instruction is ex-
executed only once and the costs on loading and locking memory blocks are nearly the same as the longest path based dynamic locking approach.

For the benchmarks containing large sequential non-nested loops, the improvement depends on the structure of the CFG of the benchmark. Our approach has a dramatic improvement in some benchmarks such as fir, or a slight improvement in some benchmarks such as adpcm.

With a fixed cache line size, the improvement of our approach slightly decreases as the associativity of the I-cache decreases due to the reason that fewer memory blocks can be loaded and locked into a set. With a fixed associativity of the I-cache, the improvement of our approach remains similar as the cache line size increases.

Overall, our approach has two major advantages compared with the longest path based dynamic locking approach. Firstly, our approach uses a more effective algorithm for determining the loading points of the selected memory blocks to further reduce the WCET of a loop nest. Secondly, our approach makes more effective selections of locked cache contents by using the min-cut based algorithm.

8. Conclusion

We investigate the problem of selecting memory blocks of the instructions of a single task as the locked contents and determining their loading points to minimize the WCET of the task. We propose a dynamic cache locking approach. Our approach considers the live range of each memory block. When the live range of a memory block ends, the cache line it occupies can be reused. Our approach selects a minimum set of memory blocks of each loop as the locked cache contents to reduce the WCET of the loop as much as possible by finding minimum node cuts of the weighted DAG of the 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, our approach can find good loading points for the selected memory blocks to further reduce the WCET of a task. We have implemented our approach and compared it with the two state-of-the-art cache locking approaches, the partial locking approach proposed in [6] and the dynamic locking approach proposed in [15], by using a set of benchmarks from the MRTC benchmark suite. The experimental results show that our approach achieves the average improvements of 41%, 15% and 7% over the partial locking approach proposed in [6] for the 256B, 512B and 1KB caches, respectively, and 7%, 18% and 17% over the longest path based dynamic locking for the 256B, 512B and 1KB caches, respectively.

To conclude this paper, we propose two open problems. One open problem is to extend our approach to the D-cache locking problem. As each variable has its own live range, it is very challenging to propose a fast, efficient dynamic D-cache locking heuristic for selecting variables as the D-cache contents to effectively reduce the WCET of a task. The other one is to integrate our approach with task scheduling. Tasks may preempt each other. If a task \( T_i \) is preempted by a task \( T_j \), the cache lines locked by \( T_i \) cannot be used by \( T_j \). Therefore, pre-emptions have a significant impact on the performance of cache locking. A good cache locking approach needs to take both task scheduling and cache locking into account simultaneously.

References


Figure 19. WCET improvements of our approach over the longest path based dynamic locking approach (2-way set associative cache, 32B cache lines)

(a) cache size : 256B

(b) cache size : 512B

(c) cache size : 1kB

Figure 20. WCET improvements of our approach over the longest path based dynamic locking approach (2-way set associative cache, 64B cache lines)


