WCET-Centric Partial Instruction Cache Locking

Huping Ding[1], Yun Liang[2] and Tulika Mitra[1]
[1]School of Computing, National University of Singapore
[2]Advanced Digital Sciences Center, Illinois at Singapore
{dinghuping,tulika}@comp.nus.edu.sg, eric.liang@adsc.com.sg

ABSTRACT

Caches play an important role in embedded systems by bridging the performance gap between high speed processors and slow memory. At the same time, caches introduce imprecision in Worst-case Execution Time (WCET) estimation due to unpredictable access latencies. Modern embedded processors often include cache locking mechanism for better timing predictability. As the cache contents are statically known, memory access latencies are predictable leading to precise WCET estimates. Moreover, by carefully selecting the memory blocks to be locked, WCET estimate can be reduced compared to cache modeling without locking. Existing static instruction cache locking techniques strive to lock the entire cache to minimize the WCET. We observe that such aggressive locking mechanisms may have negative impact on the overall WCET as some memory blocks with predictable access behavior get excluded from the cache. We introduce a partial cache locking mechanism that has the flexibility to lock only a fraction of the cache. We judiciously select the memory blocks for locking through accurate cache modeling that determines the impact of the decision on the program WCET. Our synergistic cache modeling and locking mechanism achieves up to 68% reduction in WCET for a large number of embedded benchmark applications.

Categories and Subject Descriptors

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

General Terms

Algorithm, Design, Performance

Keywords

WCET, Partial Cache Locking

1. INTRODUCTION

Cache memories are often employed in embedded systems to hide the main memory access latency. Caches are quite effective in improving the average-case performance due to the temporal and spatial locality of memory accesses in a program. For hard real-time systems, however, caches are problematic due to timing unpredictability — specially in the context of Worst-case Execution Time (WCET) estimation. WCET is an important metric for hard real-time systems. It is defined as the upper bound on the maximum execution time of the program on a particular hardware platform across all the possible inputs. In the presence of caches, it is challenging to determine the cache behavior for a memory access (hit or miss) through static program analysis for WCET estimation. If a memory access cannot be guaranteed as a cache hit, it is conservatively estimated to be a miss in WCET analysis. This leads to significant imprecision in WCET estimation.

In this paper, we focus on instruction caches, which are present in almost all embedded systems today. There exist many static program analysis techniques that model the instruction cache for tight WCET estimation [19, 10]. For example, Theiling et al. [19] model the cache states at each program point and classify the cache behavior (hit or miss) of a memory access based on the cache state. However, in the presence of complex control flow, cache modeling may fail to accurately determine the cache behavior for some memory accesses. Such unclassified accesses are conservatively assumed to be cache misses in WCET analysis due to the safety critical nature of hard real-time systems. This over-estimation can lead to serious over-dimensioning of the processor resources.

To improve timing predictability, modern embedded processors often feature cache locking mechanisms. Memory blocks can be locked in the cache using special lock instructions. Once a memory block is locked, it cannot be evicted from the cache under replacement policy. Thus, locking the entire cache resolves the problem of timing unpredictability. More importantly, by carefully choosing the memory blocks to be locked, WCET estimate can be reduced compared to cache modeling techniques without locking [6, 14].

Embedded processors also provide the option of partial cache locking through two different mechanisms: way locking and line locking. In way locking, particular ways are entirely locked for all the cache sets. Way-locking is employed in several ARM processor series. Line locking allows different number of lines to be locked in different cache sets and is employed in Intel Xscale, ARM9 family and Blackfin 5xx family. In partial cache locking, the unlocked portion of the cache behaves as a normal cache. For example, if in a 4-way set associative cache, 2 cache ways are locked, then the other two cache ways serve as a 2-way set associative cache. Clearly, line locking is more flexible than way locking, which in turn is more flexible than full cache locking.

Recently, a heuristic [6] and an optimal solution [14] have been proposed to minimize the WCET via static instruction cache locking. These existing techniques make an *implicit but important de-*

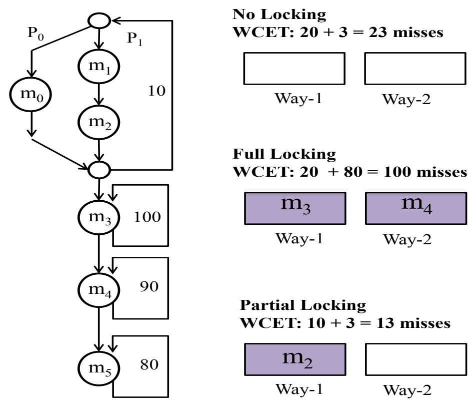

Figure 1: Advantage of partial cache locking over full cache locking and cache modeling with no locking. The program consists of four loops. The first loop contains two paths (P_0 and P_1) and the other three loops contain only one path. The loop iteration counts appear on the back edges.

cision of locking the entire cache. This crucial decision arises from the assumption that instruction cache modeling for WCET analysis is quite imprecise. By employing full cache locking, [6, 14] can completely bypass cache modeling in WCET analysis phase and thereby achieve tight WCET estimation. Indeed, as these techniques are oblivious to cache modeling, they assume the worst-case behavior with empty cache (where all the accesses are serviced from main memory) as the reference point and improve upon it through locking of memory blocks along the WCET path. In this context, it is guaranteed that locking the entire cache will provide maximum WCET reduction compared to the baseline empty cache. In other words, the cache locking problem becomes equivalent to the scratchpad memory allocation problem.

In this paper, we argue (and experimentally validate) that aggressive full cache locking as proposed in [6, 14] may have substantial negative impact on WCET reduction. State-of-the-art instruction cache modeling techniques for WCET analysis are quite mature. Most memory accesses can thus be successfully classified as hit/miss through WCET analysis techniques. Consider a memory block m originally classified as cache hit in a normal cache through static WCET analysis. But m is not selected for locking under full cache locking scenario. Thus m does not have any opportunity to reside in the cache and all its accesses incur cache misses. Now consider an alternative scenario where partial cache locking is employed. Again m is not selected for locking. However, as the cache has some unlocked lines, m may still be brought into the cache at runtime and the cache misses can be avoided.

In summary, full cache locking does not exploit the entire spectrum of opportunities presented by cache locking. Thus, in this paper we propose a partial cache locking technique that explores in conjunction with accurate cache modeling the entire spectrum of choices. Partial cache locking problem is more challenging compared to full cache locking as it requires careful cost-benefit analysis to decide between locking a cache line with a single memory block versus keeping it unlocked so that more than one memory blocks can benefit from it. This synergistic interaction between cache modeling and memory block selection for locking sets apart our technique from the state-of-the-art.

Motivating Example. We illustrate the benefit of partial cache locking over full cache locking with a concrete example shown in Figure 1. The program consists of four loops and we assume that all the memory blocks are mapped to the same cache set in a 2-way set associative cache.

Cache modeling with no locking: Let us first estimate the WCET via cache modeling with no locking. Theiling et al. [19] models the cache states at all program points. All the memory blocks in the first loop ($m0$, $m1$, $m2$) are cache misses in the worst case because alternate execution of the two program paths ($P0$ and $P1$) can lead to mutual eviction of the blocks. Thus, program path $P1$ with 2 cache miss is the worst case path in the first loop. For the other three loops, cache modeling techniques can easily determine that the first access is a cold miss and the subsequent accesses are cache hits via persistence analysis or virtual unrolling [15, 19]. Therefore, cache modeling estimates 23 cache misses in the worst case — 20 misses for the first loop and 3 misses for the other loops.

Full cache locking: Existing cache locking techniques [6, 14] first build the worst case path (e.g., $(m1m2)^{10}m3^{100}m4^{90}m5^{80}$) assuming that all accesses are serviced from the main memory (i.e., there is no cache). Now memory blocks are selected for locking along the worst-case path so as to improve the WCET until the cache is fully locked. Both cache locking techniques [6, 14] model the fact that the WCET path may change after locking some memory locks. For this example, the heuristic [6] and the optimal [14] approach return the same solution. $m3$ and $m4$ are chosen to be locked as they contribute most towards WCET reduction. After locking, we get 100 cache misses in total in the worst case — 20 misses in the first loop and 80 misses in the last loop. Thus, cache locking performs worse than cache modeling in this example.

Partial cache locking: Our partial locking technique can determine that it is beneficial to keep one cache line free so that accesses to $m3$, $m4$, and $m5$ can be cache hits after the first cold miss. It only chooses to lock $m1$ or $m2$ in the cache. Thus we get 13 cache misses in the worst case — 10 misses in the first loop and 3 cold misses for the other loops. Thus partial cache locking improves upon cache modeling and full locking.

From the example above, we first observe that full locking techniques [6, 14] are not guaranteed to perform better than cache modeling (with no locking) specially when some memory accesses can be easily classified as cache hits ($m3$, $m4$, $m5$ in our example). Locking these memory blocks with deterministic access pattern does not yield any benefit. On the other hand, if the cache is fully locked and these memory blocks with deterministic access pattern are not chosen for locking, it can have serious impact on the WCET.

Our partial locking mechanism integrates cache locking with cache modeling. We model the cache content at all program points and select the memory blocks for locking based on the cache state and their impact on the WCET. In particular, we use the concrete cache states or the abstract cache states to model the cache content. Concrete cache state captures the exact path behavior while abstract cache state is a compact representation that merges multiple concrete cache states together. For concrete cache state, we use integer linear programming (ILP) approach to optimally select the memory blocks for locking. As no cache locking and full cache locking are just two extreme instances of partial cache locking, partial locking is guaranteed to be equivalent to or better than them. To improve the efficiency, we also propose a heuristic partial locking strategy based on abstract cache state. Experimental results show that our partial cache locking technique reduces WCET by up to 68%.

2. RELATED WORK

Cache locking is used to improve timing predictability in real-time systems. Puaut and Decotigny [16] explore static cache locking in multitasking real-time systems. Two content selection algorithms have been proposed in their work to minimize the utilization and inter-task interferences. Campoy et al. [4] employ genetic algorithm to perform instruction cache locking. However, both [16] and [4] do not model the change in worst-case path after locking.

Falk et al. [6] perform cache locking by taking into account the change of worst-case path and achieve better WCET reduction. Their greedy algorithm computes the worst-case path and selects the procedure with maximum WCET reduction for locking. This process continues until the cache is fully locked. Liu et al. [14] present an optimal solution to minimize WCET via cache locking. However, their approach is optimal on the premise that the cache is fully locked. It may not be optimal towards minimizing WCET as shown in our motivating example. More importantly, they do not consider the cache mapping function at all in the locking algorithm. They simply assume that any memory block can be locked in any cache set (as if the cache is a scratchpad memory). After locking decisions are taken, they have to use code placement/layout technique [7, 12] that force the locked memory blocks to be mapped to the appropriate cache sets. This can lead to serious code size blowup, which has not been addressed.

Vera et al. [20] combine compile-time cache analysis and data cache locking in order to estimate a safe and tight worst-case memory performance. This work also assume full cache locking. Arnaud and Puaut [1] propose dynamic instruction cache locking for hard real-time systems. In their approach, the program is partitioned into regions, and static cache locking is performed for each region. In [17], cache locking is explored for predictable shared caches on multi-core systems. Cache locking are also shown to be quite effective for improving average-case execution time [13]. Finally, optimal on-chip scratchpad memory allocation to improve the WCET has been explored in [5, 18].

3. CACHE MODELING

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

Given a memory block m, it is mapped to only one cache set. Thus, the different cache sets are independent and can be modeled independently. In the following, we describe our modeling technique for one cache set. The same modeling techniques can be repeated for other cache sets. We use M to denote the set of memory blocks mapped to a cache set and use $\perp$ to indicate the absence of any memory block in a cache line.

3.1 Cache States

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

DEFINITION 2 (**Concrete Cache State Hit**). *Given a concrete cache state c and a memory access $m \in M$*

$$c\_hit(c, m) = \begin{cases} 1 & if\ \exists i\ (0 \leq i \leq A-1)\ \ s.t.\ c[i] = m \\ 0 & otherwise \end{cases}$$

We use Ω to denote the set of all possible concrete cache states of a program. Note that a program point can be reached via multiple paths and these paths may lead to different concrete cache states. We use $\mathcal{P}$ to denote the set of all possible concrete cache states at a program point, i.e., $\mathcal{P} \in 2^{\Omega}$. We can easily compute $\mathcal{P}$ at each program point through static program analysis as shown in [11].

Given the set of all possible cache states $\mathcal{P}$ at a program point and a memory access $m \in M$,

$$p\_hit(\mathcal{P}, m) = \begin{cases} 1 & if\ \forall c \in \mathcal{P}\ \ c\_hit(c, m) = 1 \\ 0 & otherwise \end{cases}$$

That is, an access m is a hit at a program point with the set of all possible concrete cache states $\mathcal{P}$ if and only if m is hit in all the concrete cache states of $\mathcal{P}$.

Maintaining the set of all possible cache states may not be feasible (and scalable) for large programs with complex control flows where a program point can potentially have hundreds or even thousands of cache states. Thus we also employ abstract interpretation to compute the abstract cache state at every program point [19]. An abstract cache state is derived by joining all possible concrete cache states at a program point.

DEFINITION 3 (**Abstract Cache State**). *An abstract cache state a is a vector $\langle a[0], ...a[A-1]\rangle$ of length A where $a[i] \in 2^M$.*

An abstract cache state maps a cache line to a set of memory blocks. Must analysis and may analysis [19] are usually employed to compute abstract cache states for WCET analysis. Given a program point, must analysis determines the set of memory blocks that are guaranteed to be present in the cache, while may analysis determines the set of memory blocks that are never in the cache. Must analysis uses abstract cache states where the position of a memory block is an upper bound of its age. In may analysis, the lower bound of the age of a memory block is used as its position in the abstract cache state, in order to capture the set of all memory blocks that may be in the cache. Figure 2 shows the relationship between a set of concrete cache states and the corresponding abstract cache states.

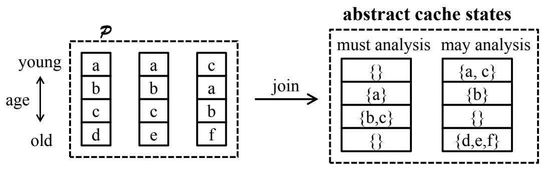

Figure 2: Concrete cache states and abstract cache states.

4. CACHE LOCKING

In this paper, we consider static cache locking, where the selected memory blocks are locked into the cache before the program starts execution and remain unchanged throughout the execution. Furthermore, we consider line locking mechanism, where different number of lines can be locked in different cache sets. As discussed before, for our purposes, we can treat each cache set independently because the memory blocks mapped to different cache sets do not interfere. Each cache set can be considered as a fully associative cache containing A lines, where A is the associativity. Once a memory block is locked in a cache line, it can not be evicted from the cache. The remaining unlocked lines in the cache set serve as a fully associative cache with reduced capacity.

Note that the mapping of instructions to the cache sets depends on the code memory layout. Inserting additional code for cache locking may tamper this layout. To avoid this problem, we use the trampolines [2] approach. The extra code to fetch and lock the memory blocks in the cache are inserted at the end of the program as a trampoline. We leave a dummy NOP instruction at the entry point of the program that gets replaced by a call to this trampoline.

The main challenge is in selecting the memory blocks for locking so as to minimize the WCET. In the following, we propose two solutions. The first one is an optimal solution employing Integer

Linear Programming (ILP) formulation based on concrete cache states and the second one is a heuristic approach based on abstract cache states.

4.1 Optimal solution with concrete cache states

The set of concrete cache states at any program point captures the exact set of cache states resulting from all possible program paths. Based on this accurate set of cache states, we formulate an ILP problem to optimally select the memory blocks for partial locking. In the following, we first show the ILP formulation for a loop and then extend it to the whole program.

4.1.1 ILP Formulation for Loop

We represent the loop body as a Directed Acyclic Graph (DAG). Each DAG is associated with a unique source and sink node. We compute the set of possible concrete cache states $\mathcal{P}$ at any point of the program through static program analysis [11]. Given the set of all possible cache states $\mathcal{P}$ and a memory block access m, $p\_hit(\mathcal{P}, m)$ determines whether the access is a cache hit or miss before locking. Next, we proceed to determine the cache access behavior of m after locking.

For each memory block m, we define a 0-1 decision variable L_m, which indicates whether m is locked in the cache. Thus,

$$0 \le L_m \le 1$$

There are only A (associativity) cache lines available for locking in each cache set. Thus for each cache set i

$$\sum_{m \in M_i} L_m \le A$$

where M_i is the set of memory blocks mapped to cache set i.

The accesses to the locked memory blocks are cache hits. Let $Lock_i$ denote the set of memory blocks locked in cache set i. For an unlocked memory block m mapped to cache set i ($m \in M_i, m \notin Lock_i$), its access can be classified as hit or miss depending on the concrete cache states $\mathcal{P}$ at that program point and $Lock_i$.

For a concrete cache state $c \in \mathcal{P}$, we define age^c_m as the age of the memory block m in c, where $age^c_m = 0$ ($age^c_m = A-1$) if m is the most (least) recently accessed memory block in c. If $m \notin c$ ($c\_hit(c, m) = 0$), then $age^c_m = A$. Thus, $0 \le age^c_m \le A$. If $m \in M_i$ and $m \notin Lock_i$, then given a concrete cache state c, the access to m is cache hit if

$$\left(age^c_m + \sum_{m' \in Lock_i \wedge age^c_{m'} > age^c_m} L_{m'} \right) < A \qquad (1)$$

In other words, if a locked memory block $m' \in Lock_i$ is younger than m in the cache state c, then locking m' does not change the hit classification of m. However, if $m' \in Lock_i$ is older than m in cache state c (i.e., $age^c_{m'} > age^c_m$), then locking m' essentially increases age of m by 1. If the number of such older memory blocks added to age^c_m exceeds the associativity, then m becomes a cache miss due to locking.

We define a 0-1 variable h^c_m, which specifies whether m is a cache hit in c after locking. Based on Equation 1,

$$h^c_m = \begin{cases} 1 & if \ \left(A - age^c_m - \sum_{m' \in Lock_i \wedge age^c_{m'} > age^c_m} L_{m'} \right) > 0 \\ 0 & otherwise \end{cases}$$

However, the above equation is not linear. We substitute it with the equivalent linear equations as follows.

$$A - \sum_{m' \in Lock_i \wedge age^c_{m'} > age^c_m} L_{m'} - age^c_m - U \times h^c_m \le 0$$

$$A - \sum_{m' \in Lock_i \wedge age^c_{m'} > age^c_m} L_{m'} - age^c_m + U - U \times h^c_m > 0$$

where U is a large constant ($U \ge A$).

The set of concrete cache states $\mathcal{P}$ at a program point usually contains more than one concrete cache states ($|\mathcal{P}| > 1$). Memory block access m is guaranteed as cache hit if and only if it is cache hit for every concrete cache state $c \in \mathcal{P}$. We define a 0-1 variable $h^{\mathcal{P}}_m$, which specifies whether m is a cache hit in $\mathcal{P}$ after locking.

$$h^{\mathcal{P}}_m = \begin{cases} 1 & if \ \sum_{c \in \mathcal{P}} h^c_m = |\mathcal{P}| \\ 0 & otherwise \end{cases}$$

We linearize the above equation as follows.

$$\sum_{c \in \mathcal{P}} h^c_m - h^{\mathcal{P}}_m \le |\mathcal{P}| - 1$$

$$\sum_{c \in \mathcal{P}} h^c_m - |\mathcal{P}| \times h^{\mathcal{P}}_m \ge 0$$

Finally, for each memory block access m, we define a 0-1 decision variable hit_m, which specifies whether m is cache hit or miss after locking. Locked memory blocks are guaranteed to be cache hits. On the other hand, for an unlocked memory block m, we rely on its corresponding cache state $\mathcal{P}$ to determine the cache behavior.

$$hit_m = \begin{cases} 1 & if \ L_m = 1 \\ h^{\mathcal{P}}_m & otherwise \end{cases}$$

We linearize the above equation as follows.

$$hit_m \ge L_m, hit_m \ge h^{\mathcal{P}}_m \text{ and } hit_m \le L_m + h^{\mathcal{P}}_m$$

Thus, the access latency of basic block B after cache locking is calculated as follows

$$T_B = \sum_{m \in B} (miss\_lat - (miss\_lat - hit\_lat) \times hit_m)$$

where $miss\_lat$ and hit_lat are the cache miss penalty and cache hit latency, respectively.

We also define a variable W_B for each basic block B in the loop, which represents the latency of the worst-case path rooted at basic block B in the DAG after cache locking. Then

$$W_B = \max_{B' \in imsucc(B)} \{W_{B'} + T_B\}$$

where $imsucc(B)$ is the set of immediate successors of B in DAG. Therefore, for any outgoing edge from node B to node B' ($B \rightarrow B'$) in the DAG, we have the following constraint

$$W_B \ge W_{B'} + T_B$$

Since there is no outgoing edge for the sink node of the loop, it is defined specially

$$W_{sink} = T_{sink}$$

Obviously, W_{src} will capture the latency of the worst-case acyclic path in the DAG (src is the source node of DAG). Let lb be the loop bound of this loop (maximum number of iterations of this loop). Then, $W_{src} \times lb$ is the WCET of this loop after cache locking. Thus, the optimal cache locking result for this loop can be obtained by minimizing $W_{src} \times lb$ (the objective function of ILP formulation).

4.1.2 Extension to the Whole Program

In the previous section, we present an ILP formulation to obtain the optimal cache locking for a loop. In order to obtain the ILP formulation for the whole program, we are required to start from the innermost loops of the program. We first generate the ILP formulation for the innermost loops, and then each innermost loop is treated as a dummy basic block of the outer loop. Therefore, we can construct the ILP formulation for the next level of loop. We continue this way until we reach the outmost loop in the program. Clearly, W_{entry} represents the WCET of the whole program under cache locking, where $entry$ denotes the entry node of program. Finally, the locking overhead (e.g., the execution of the locking instructions) are included in the WCET of the whole program.

4.2 Heuristic with abstract cache states

In the previous section, we develop an optimal ILP formulation using concrete cache states. However, programs with complex control flow may have hundreds or even thousands of cache states at a program point. For such programs, maintaining all possible concrete cache states may not be feasible. Also ILP formulation may take very long to reach a solution specially for larger programs and larger associativity. Thus, we propose a heuristic approach based on abstract cache states. Abstract cache state is a more compact representation compared to the set of concrete cache states.

We first perform WCET analysis with cache modeling based on abstract interpretation [19]. Then we can easily determine cache hit/miss classification for each memory access based on the abstract cache states. As a by-product of the WCET analysis, we obtain the abstract cache states under *must analysis* at all program points. Meanwhile, we also collect the execution frequency of each basic block along the worse-case path. Then we iteratively lock some memory blocks on the worse-case path to improve the WCET.

Suppose memory block m is on the worst-case path. Let lat_m be the access latency of m according to the hit/miss classification in WCET analysis, and f_m is its execution frequency along the worst-case path. By locking memory block m, all accesses to m will be cache hits. Therefore, we define the benefit of locking m as

$$benefit_m = (lat_m - hit\_lat) \times f_m$$

where hit_lat is the cache hit latency. Thus, locking a memory block guaranteed to be hit before locking does not give any benefit.

On the other hand, locking memory block m in cache may have negative impact for the memory blocks mapped to the same set as the associativity for this set is reduced by 1. Similar to concrete cache state, we define the age of a memory block m in abstract must cache state $\mathcal{C}$ as $age^{\mathcal{C}}_m$. When $m \in \mathcal{C}$, $0 \le age^{\mathcal{C}}_m \le A-1$, where A is the associativity. Otherwise, we set its age to A.

Suppose we choose to lock memory block m in the cache and its $benefit_m > 0$. In other words, m is not in the abstract must cache state before locking. Then, locking m will downgrade the memory block m' from cache hit to cache miss if $age^{\mathcal{C}}_{m'} = A-1$. Note that the associativity A here refers to the current associativity of the set. That is A refers to the original associativity of the cache minus the number of memory blocks locked in the set so far. Therefore, we define the cost of locking m as follows.

$$cost_m = \sum_{m' \in M_i \wedge age^{\mathcal{C}}_{m'} = A-1} (miss\_lat - hit\_lat) \times f_{m'}$$

where as before $m \in M_i$. Then, the overall gain of locking m is

$$gain_m = benefit_m - cost_m$$

We compare different memory blocks in terms of their gain and select the most beneficial memory block m to be locked. However, $gain_m$ may not be the actual WCET reduction because the worst-case path may change after locking m. Thus, we update cache state for instructions mapped to the affected cache set and perform WCET analysis again to obtain the exact WCET after locking m. If the WCET is actually reduced, we lock m in the cache. We continually select memory blocks for locking until either there is no actual WCET improvement after locking any memory block or there is no gain in the cost-benefit analysis for any memory block m (i.e., $gain_m \le 0$). Finally, the locking overhead is included. The detail algorithm is shown in Appendix A.

Table 1: Characteristic of benchmarks & analysis time

Benchmarks	Code Size (bytes)	Optimal (sec)	Heuristic (sec)	Speedup
adpcm	12,480	313.37	1.28	245
cnt	1,648	0.43	0.05	9
compress	4,864	145.61	0.33	441
crc	2,048	1.44	0.10	14
edn	7,296	1.07	0.16	7
fir	1,152	0.10	0.02	5
jfdctint	5,520	0.35	0.06	6
matmult	1,632	0.37	0.07	5
minver	6,256	114.20	0.35	326
qurt	2,048	1.20	0.13	9

5. EXPERIMENTAL EVALUATION

In this section, we present the experimental evaluation of partial cache locking. We compare both the optimal and the heuristic solutions with static cache analysis [19] and the full cache locking approach proposed by Falk et al. [6].

5.1 Experimental Setup

We use the benchmarks from MRTC benchmark suite [8] as shown in Table 1. We compile our benchmarks for SimpleScalar PISA (Portable ISA) instruction set [3] — a MIPS like instruction set architecture — with gcc cross-compiler. The control flow graphs of these benchmarks are extracted and provided as input to our cache locking analysis. Our framework is built on top of the open-source WCET analysis tool Chronos [9]. The binary code size of each program is shown in the second column of Table 1. We perform all the experiments on 2.53GHz Intel Xeon CPU with 24GB memory. IBM CPLEX is used as the ILP solver.

We assume only one level of instruction cache in the architecture. In other words, an instruction access is either cache hit or it has to be fetched from memory. The cache hit latency is 1 cycle, while a cache miss takes 30 cycles. As we are modeling the instruction cache, we assume a simple in-order processor with unit-latency for all data memory references.

5.2 Partial Cache Locking vs. Static Analysis

Figure 3 shows the WCET improvement of partial cache locking over static analysis with no locking based on abstract interpretation [19]. The instruction cache is 4-way set associative with block size of 32 bytes, and its capacity is varied from 512B to 1KB (see Appendix for other settings).

Our partial cache locking technique significantly improves the WCET over static analysis with no locking for many benchmarks (e.g., cnt, crc and $qurt$) for different cache sizes. However, some benchmarks show limited improvement of WCET via partial cache locking, especially when the cache size is small. This is mainly due to the fact that locking memory blocks destroys the deterministic access pattern for some unlocked blocks. Therefore, our partial locking technique decides not to lock these memory blocks and the result of partial locking is close to that of static analysis.

For most of the benchmarks, the improvement increases as the cache size increases, because there is more space for locking and more memory blocks can be locked into the cache. However, for

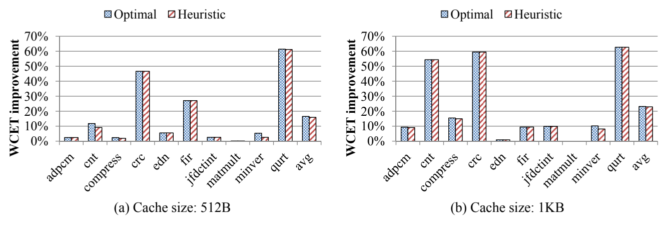

Figure 3: WCET improvement of partial cache locking (optimal and heuristic solution) over static cache analysis with no locking (cache: 4-way set associative, 32-byte block).

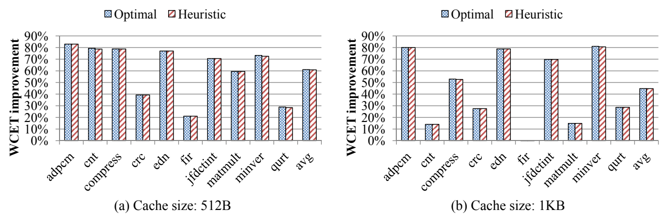

Figure 4: WCET improvement of partial cache locking (optimal and heuristic solution) over Falk et al.'s method (cache: 4-way set associative, 32-byte block).

some benchmarks, the improvement decreases as cache size increases, for example fir. For fir, when the cache size increases, more memory accesses become deterministic, which can be successfully identified by static cache analysis. Thus, cache locking may not help to improve the WCET much compared to static cache analysis. Overall, more WCET improvement is observed as the cache size increases. On an average, 16% and 23% improvement are achieved for 512B and 1KB size cache, respectively.

5.3 Partial versus Full Cache Locking

There exist two full cache locking techniques as mentioned in Section 2 [14, 6]. Even though Liu et al. [14] show that their approach can achieve better WCET reduction compared to [6], it has several limitations. Liu et al. do not consider the cache mapping function in the locking algorithm. They simply assume that any memory block can be locked in any cache set (as if the cache is a scratchpad memory). After locking decisions are made, they employ code placement techniques that force the locked memory blocks to be mapped to the appropriate cache sets. This can lead to code size blowup, which has not been addressed in their work.

Thus we decide to compare our partial locking results with that of Falk et al.'s method [6] as both approaches do not require any subsequent code placement/layout technique. We choose memory blocks as locking granularity instead of procedures originally used in Falk et al.'s method for a fair comparison. Note that this choice of granularity does not change the core greedy heuristic algorithm proposed in [6]. The instruction cache is 4-way set associative with block size of 32 bytes, and we vary its size from 512B to 1KB (see Appendix for other settings).

The WCET improvement of partial cache locking over Falk et al.'s method is shown in Figure 4. Both optimal and heuristic partial locking approaches outperform Falk et al.'s method for different cache sizes. Our partial cache locking techniques usually lock part of the cache. The number of locked cache lines vary for different cache sets (see Appendix for detailed results). Thus, after locking, there are still some cache lines left for the unlocked memory blocks to exploit their locality of accesses. However, in Falk et al.'s method, the cache is fully locked and all the accesses to the unlocked memory blocks are cache misses.

5.4 Optimal vs. Heuristic Approach

As shown in Figure 3 and 4, our heuristic approach obtains nearly the same results as the optimal solution. Table 1 presents the average analysis time of different algorithms for all the benchmarks. Clearly, our heuristic approach produces comparable results to the optimal solution while it is more efficient in analysis time.

6. CONCLUSION

In this paper, we propose partial cache locking for WCET reduction. We have proposed an optimal partial locking solution based on concrete cache states as well as a heuristic approach based on abstract cache states. Our partial cache locking significantly reduces the WCET compared to the static cache analysis and the state-of-the-art cache locking techniques that fully lock the cache. Our heuristic achieves comparable WCET reduction to the optimal solution but it is more efficient in terms of runtime. In the future, we will consider data cache locking and explore dynamic cache locking for further improvement.

7. ACKNOWLEDGMENTS

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

8. REFERENCES

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

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

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

[4] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *ECRTS*, 2005.

[5] H. Falk and J. C. Kleinsorge. Optimal static WCET-aware scratchpad allocation of program code. In *DAC*, 2009.

[6] H. Falk, S. Plazar, and H. Theiling. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS*, 2007.

[7] C. Guillon, F. Rastello, T. Bidault, and F. Bouchez. Procedure placement using temporal-ordering information: Dealing with code size expansion. *J. Embedded Comput.*, 1, 2005.

[8] J. Gustafsson, A. Betts, A. Ermedahl, and B. Lisper. The Mälardalen WCET benchmarks – past, present and future. In *WCET*, 2010.

[9] X. Li, Y. Liang, T. Mitra, and A. Roychoudury. Chronos: A timing analyzer for embedded software. *Science of Computer Programming*, 69(1-3), 2007.

[10] Y.-T. S. Li, S. Malik, and A. Wolfe. Cache modeling for real-time software: beyond direct mapped instruction caches. In *RTSS*, 1996.

[11] Y. Liang and T. Mitra. Cache modeling in probabilistic execution time analysis. In *DAC*, 2008.

[12] Y. Liang and T. Mitra. Improved procedure placement for set associative caches. In *CASES*, 2010.

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

[14] T. Liu, M. Li, and C. J. Xue. Minimizing WCET for real-time embedded systems via static instruction cache locking. In *RTAS*, 2009.

[15] F. Martin, M. Alt, R. Wilhelm, and C. Ferdinand. Analysis of loops. In *CC*, 1998.

[16] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS*, 2002.

[17] V. Suhendra and T. Mitra. Exploring locking & partitioning for predictable shared caches on multi-cores. In *DAC*, 2008.

[18] V. Suhendra, T. Mitra, A. Roychoudhury, and T. Chen. WCET centric data allocation to scratchpad memory. In *RTSS*, 2005.

[19] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Syst.*, 18, 2000.

[20] X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev.*, 31, 2003.

APPENDIX

A. HEURISTIC ALGORITHM

Algorithm 1 presents the details of our heuristic approach described in Section 4.2. This approach is based on abstract cache states. The algorithm iteratively selects the most beneficial memory block for locking based on the cost and benefit metrics defined in Section 4.2. This process continues until there is no WCET improvement with further locking.

Algorithm 1: Heuristic with abstract cache states

```
   Input: Cache configuration $cfg$ and binary executable $prog$
   Output: Set of locked memory blocks $lock\_set$ and WCET
           after locking $wcet$
1  begin
2      $stop\_locking := false$; $lock\_set = null$;
3      analyze_abstract_cache_states($prog$, $cfg$);
4      $wcet$ := analyze_wcet();
5      while $(!stop\_locking)$ do
6          /* select candidate memory block to lock */
           $cnd := null$; $gain_{cnd} := 0$;
7          foreach $m \in M$ do
8              Suppose $m$ is mapped to cache set $s$;
9              Let $assoc$ be the current associativity of $s$;
10             if $(m \notin lock\_set \wedge assoc > 0)$ then
11                 $benefit_m$ := calculate_benefit();
12                 $cost_m$ := calculate_cost($assoc$);
13                 $gain_m := benefit_m$ - $cost_m$;
14                 if $gain_m > gain_{cnd}$ then
15                     $cnd = m$; $gain_{cnd} = gain_m$;
16
17
18         if $cnd \neq null$ then
19             lock_to_cache($cnd$);
20             /* update cache states for affected cache set */
               update_cache_state($prog$, $cfg$, $cnd$);
21             $new\_wcet$ := analyze_wcet();
22             if $new\_wcet < wcet$ then
23                 $wcet := new\_wcet$;
24                 $lock\_set := lock\_set \cup cnd$;
25                 update associativity for the affected cache set;
26             else
27                 stop_locking := $true$;
28
29         else
30             stop_locking := $true$;
31
32
33 end
```

The input to the algorithm is the cache configuration cfg and the binary executable cfg. First, we perform cache modeling based on abstract interpretation [19] for this binary executable on line 3. The output of this analysis are the abstract cache states at each program point. Next we perform WCET analysis of the binary executable (line 4) where memory accesses are categorized into always hit, always miss, and unclassified based on abstract cache states. The $wcet$ obtained in this step is the original WCET obtained through static cache analysis and no cache locking.

Now, we iteratively select the most beneficial memory block for locking into the cache. Let M be the set of all memory blocks. We perform cost-benefit analysis for each memory block $m \in M$ where m is not yet locked ($m \notin lock\_set$) and the cache set m is mapped to still has some unlocked cache lines ($assoc > 0$). We gain benefit from locking m if m was not guaranteed to be a hit after static cache analysis (see Section 4.2). However, there is a cost associated with locking m. Let s be the cache set where m is mapped to. Then the other memory blocks mapped to cache set s but not yet locked will have one less cache block available in the cache set s. As discussed in Section 4.2, some of these blocks now may incur cache miss (even though their accesses were hits under static cache analysis) depending on their relative age with respect to the age of m in cache set s. The additional latency incurred due to these cache misses will contribute to the cost of locking m. If difference between benefit and cost of locking m is the gain. We identify the memory block cnd with maximum gain.

If we cannot identify any memory block with positive gain, then the locking algorithm terminates. Note that the cost-benefit analysis is approximate in nature because it depends on the frequency of memory accesses along the worst-case path before locking memory block cnd. After locking memory block cnd, the worst-case path may change. So we update the abstract cache states for the cache set where cnd is mapped to and repeat WCET analysis with this new abstract cache states. If the new WCET is indeed lower than the previous WCET, then we add the memory block cnd to $lock\_set$. We also need to decrease the associativity of the corresponding cache set. If the actual WCET after locking m is lower than the previous WCET, then we terminate the algorithm.

B. EVALUATION WITH DIFFERENT CACHE CONFIGURATIONS

In this section, we perform an extensive comparison of our partial cache locking solutions (optimal and heuristic) with static cache analysis (no locking) [19] and Falk et al.'s method (full cache locking) [6] for various cache configurations.

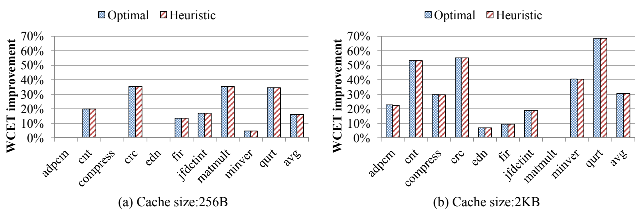

Figure 5: Comparison between partial cache locking and static cache analysis with no locking for cache size of 256B and 2KB (cache:4-way set associative, 32-byte block).

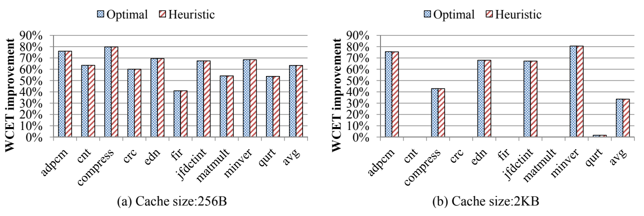

Figure 6: Comparison between partial cache locking and Falk et al.'s method for cache size of 256B and 2KB (cache:4-way set associative, 32-byte block).

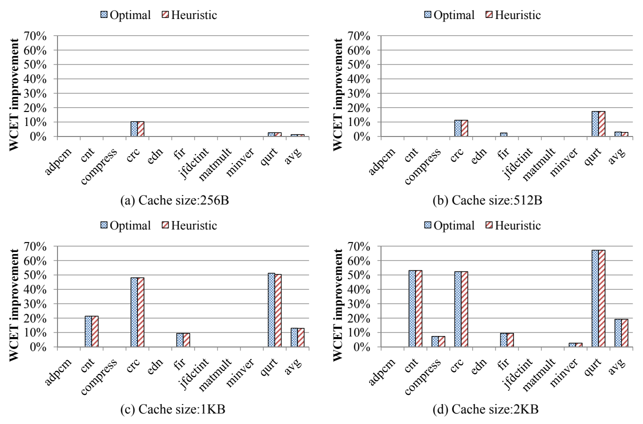

Figure 7: WCET improvement of partial cache locking over static cache analysis (no locking) for direct mapped cache, 32-byte block.

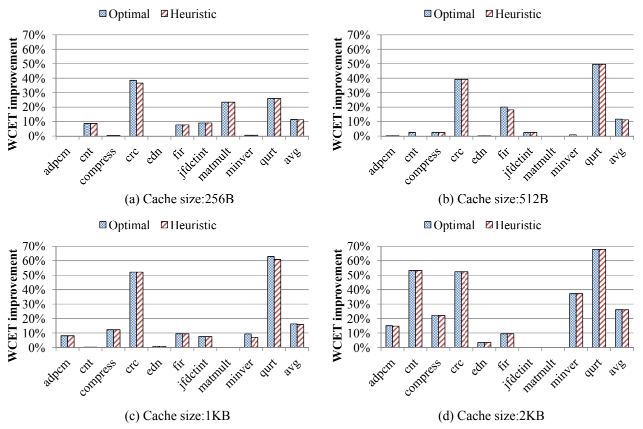

Figure 8: WCET improvement of partial cache locking over static cache analysis (no locking) for 2-way set-associative cache, 32-byte block.

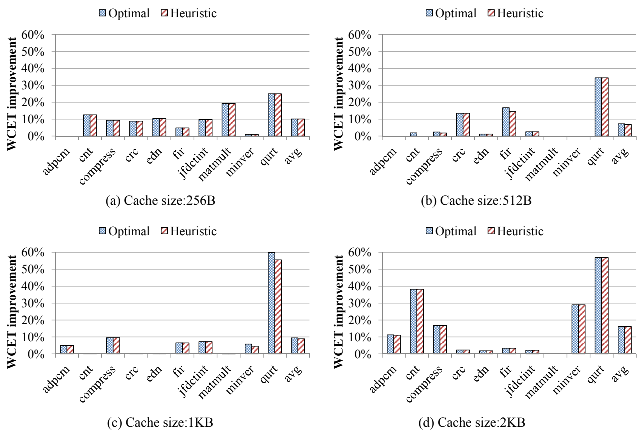

Figure 9: WCET improvement of partial cache locking over static cache analysis (no locking) for 2-way set-associative cache, 64-byte block.

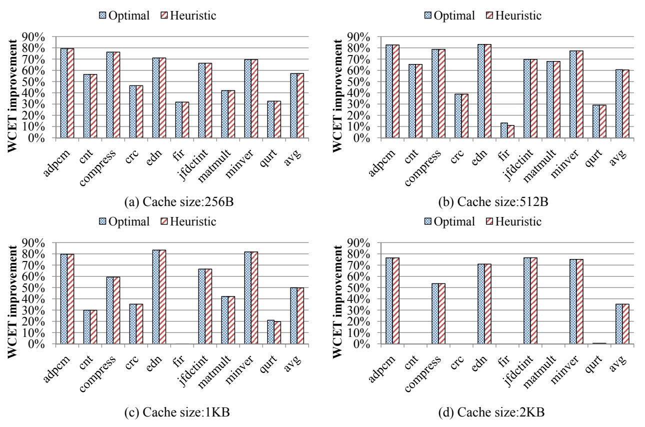

Figure 10: WCET improvement of partial cache locking over Falk et al.'s method (full locking) for direct mapped cache, 32-byte block.

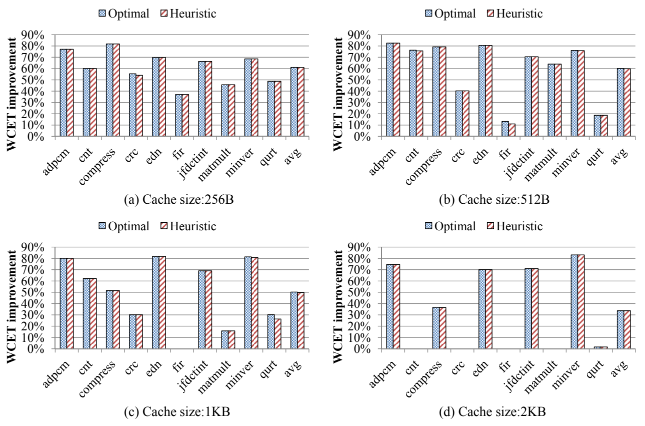

Figure 11: WCET improvement of partial cache locking over Falk et al.'s method (full locking) for 2-way set-associative cache, 32-byte block.

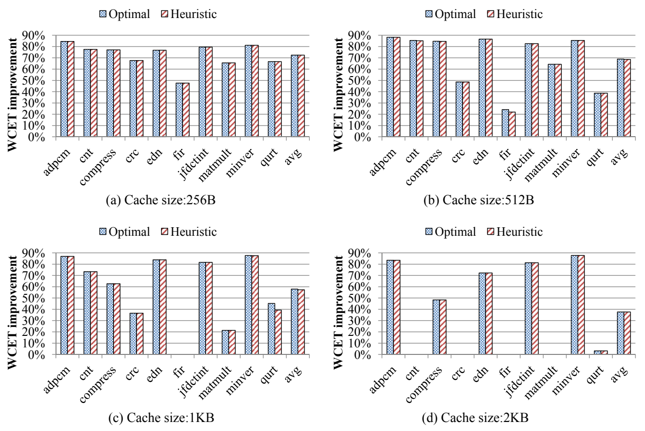

Figure 12: WCET improvement of partial cache locking over Falk et al.'s method (full locking) for 2-way set-associative cache, 64-byte block.

B.1 Different Cache Sizes

In this subsection, we present the evaluation results for 256B and 2KB size caches, respectively. The block size (32-byte) and associativity (4-way) are the same as the cache configuration presented in Section 5. Figure 5 shows the WCET improvement of partial cache locking over static cache analysis with no locking and Figure 6 shows the WCET improvement of partial cache locking over full locking (Falk et al.'s method). As expected, WCET improvement of partial cache locking over static analysis is much higher with bigger cache size as more space is available for locking memory blocks. As compared to Falk et al.'s method, when the cache is large enough to hold the entire program, all the memory blocks can be locked to achieve the minimum WCET. In that scenario, partial and full cache locking obtain identical solutions (e.g., *cnt*, *crc*, *fir*, and *matmult*).

B.2 Different Associativity

In this subsection, we evaluate our partial cache locking for different cache associativity values. In Section 5, 4-way cache associativity results are presented. Here we show the results of direct mapped and 2-way set associative caches, while the block size remains constant at 32 bytes. Figure 7 and 8 present the improvement of partial cache locking over static cache analysis with no locking for direct mapped cache and 2-way set associative cache, respectively. Figure 10 and 11 present the improvement over full locking (Falk et al.'s method) for different cache associativity.

It is observed that the WCET improvement of direct mapped cache is not as good as that of 2-way and 4-way set associative cache, especially when the cache size is small. For direct mapped cache, there is only one cache line available in each set. Locking a memory block in a cache set implies that all the accesses to the other memory blocks in the cache set will be cache miss. Thus our partial cache locking method decides not to lock any memory block for most of the benchmarks. Therefore, the partial cache locking results are similar to that of static cache analysis with no locking, especially when the cache size is small. 2-way set associative caches provide more opportunities for partial cache locking. Thus, more WCET improvement is achieved compared to direct mapped cache. Finally, partial cache locking outperforms static cache analysis with no locking and full locking (Falk et al.'s method) for different associativity.

B.3 Different Block Sizes

In this section, we evaluate our partial cache locking for different block size. In Section 5, we present results for 32 byte block size. Here we evaluate the benefits of partial cache locking for 64 bytes block size. Figure 9 and 12 present the WCET improvement with partial cache locking over static cache analysis with no locking and full locking (Falk et al.'s method), respectively. As shown, our partial locking still achieves significant improvement.

C. NUMBER OF LOCKED LINES

As mentioned before, the main strength of partial cache locking lies in the fact that cache lines are locked judiciously after performing careful cost-benefit analysis. If it is beneficial to keep a cache line unlocked so that multiple memory blocks can benefit from it, partial cache locking can identify such situations. In this subsection, we present the cache locking solutions derived by our partial cache locking mechanisms (optimal and heuristic) for a particular cache configuration. We choose a 512-byte cache with 32-byte block size and 4-way associativity. This cache has four cache sets and each set has four cache lines. Table 2 shows the number of locked cache line (i.e., locked memory blocks) per cache set.

Table 2: Number of lines locked in each set (cache: 512-byte capacity, 4-way set associative, 32-byte block).

Benchmarks	Algorithms	Set 0	Set 1	Set 2	Set 3
adpcm	optimal	1	1	1	1
	heuristic	1	1	1	1
cnt	optimal	2	0	1	1
	heuristic	2	1	0	1
compress	optimal	3	2	1	1
	heuristic	1	2	2	2
crc	optimal	3	3	2	3
	heuristic	3	3	2	3
edn	optimal	0	0	2	1
	heuristic	0	0	2	1
fir	optimal	3	3	3	3
	heuristic	3	3	3	3
jfdctint	optimal	1	3	2	2
	heuristic	3	3	3	3
matmult	optimal	1	2	0	0
	heuristic	1	2	0	0
minver	optimal	1	0	1	0
	heuristic	1	0	1	1
qurt	optimal	3	3	3	4
	heuristic	3	3	3	3

As can be observed, for all the benchmarks, our partial cache locking algorithms (optimal and heuristic) locks only a fraction of the cache lines. But the number of locked cache lines varies for different benchmarks and cache sets. For example, for *compress* benchmark, the number of locked cache lines vary from 1–3 per cache set across all the cache sets for a 4-way set associative cache. These results clearly confirm that partial cache locking is indeed important to minimize WCET compared to the two extreme ends of the spectrum of choices, namely, full cache locking and no cache locking.