Instruction Cache Locking Using Temporal Reuse Profile

Yun Liang, Tulika Mitra, and Lei Ju

***Abstract*—The performance of most embedded systems is critically dependent on the average memory access latency. Improving the cache hit rate can have significant positive impact on the performance of an application. Modern embedded processors often feature cache locking mechanisms that allow memory blocks to be locked in the cache under software control. Cache locking was primarily designed to offer timing predictability for hard real-time applications. Hence, prior techniques focus on employing cache locking to improve the worst-case execution time. However, cache locking can be quite effective in improving the average-case execution time of general embedded applications as well. In this paper, we explore static instruction cache locking to improve the average-case program performance. We introduce temporal reuse profile (TRP) to accurately and efficiently model the cost and benefit of locking memory blocks in the cache. We consider two locking mechanisms, line locking and way locking. For each locking mechanism, we propose a branch-and-bound algorithm and a heuristic approach that use the TRP to determine the most beneficial memory blocks to be locked in the cache. Experimental results show that the heuristic approach achieves close to the results of branch-and-bound algorithm and can improve the performance by 12% on average for 4 KB cache across a suite of real-world benchmarks. Moreover, our heuristic provides significant improvement compared to the state-of-the-art locking algorithm both in terms of performance and efficiency.**

***Index Terms*—Cache, cache locking, performance, temporal reuse profile (TRP).**

I. Introduction

CACHES have been employed by almost all embedded systems to mitigate the speed disparity between fast processors and slow memories. Caches can effectively reduce the number of accesses to main memory, which require more power consumption and longer delay per access. Hence, efficient use of caches is of paramount importance for embedded systems in terms of both performance and energy consumption. In this paper, we focus on instruction cache. Instruction cache is one of the foremost power consuming and performance determining microarchitectural features of modern embedded systems as instructions are fetched almost every clock cycle. For example, instruction cache consumes about 22% of the power in the Intel processor [1] and 27% of the power in the ARM processor [2]. In this paper, we focus on improving the average-case performance of embedded applications through instruction cache locking.

Most modern embedded processors (e.g., ARM Cortex series processors) feature with cache locking mechanisms whereby one or more cache blocks can be locked under software control using special lock instructions. Once a memory block is locked in the cache, it cannot be evicted from the cache under the replacement policy. Thus, all the subsequent accesses to the locked memory blocks will be cache hits. Only when the cache line is unlocked, the corresponding memory block can be replaced. Cache locking was initially designed to improve the timing predictability of hard real-time systems. As the cache content is known statically, the memory access time of each reference can be determined accurately leading to tighter worst-case execution time (WCET) analysis. Hence, most cache locking algorithms proposed in the literature target to improve the WCET of the application.

In this paper, we explore instruction cache locking to improve the average-case performance of general embedded applications. This can be achieved by systematically eliminating the cache conflict misses through cache locking. For example, consider two memory blocks m_0 and m_1 that are mapped to the same cache set and the sequence of memory block accesses is $(m_0m_1)^{10}$. Given a direct mapped cache, all the memory accesses will be cache misses (20 misses) as m_0 and m_1 replace each other from the cache alternatively. However, if either m_0 or m_1 is locked in the cache, then the total number of cache misses can be reduced to 10. Note that locking a memory block can negatively impact the performance of the remaining memory blocks mapped to the same set as the effective cache capacity gets reduced. Therefore, aggressive full locking does not always ensure good performance.

In this paper, we first introduce temporal reuse profile (TRP) to accurately capture the data reuses. TRP is significantly more compact compared to memory traces. Then, we develop cache locking modeling techniques that can accurately compute the cost and benefit of locking each memory block. We consider two locking mechanisms, line locking and way locking. For each locking mechanism, we propose locking algorithms that

Manuscript received August 18, 2014; revised November 13, 2014 and January 15, 2015; accepted February 16, 2015. Date of publication April 2, 2015; date of current version August 18, 2015. This work was supported in part by the National Natural Science Foundation of China under Grant 61300005 and Grant 61202015, and in part by the Singapore Ministry of Education Academic Research Fund Tier 1 under Grant T1-251RES1120. This paper was recommended by Associate Editor S. Kim. *(Corresponding author: Yun Liang.)*

Y. Liang is with the Center for Energy-Efficient Computing and Applications, School of Electronics Engineering and Computer Science, Peking University, Beijing 100871, China, and also with Collaborative Innovation Center of High Performance Computing, NUDT, Changsha 410073, China (e-mail: ericlyun@pku.edu.cn).

T. Mitra is with the Department of Computer Science, National University of Singapore, Singapore 117417.

L. Ju is with the School of Computer Science and Technology, Shandong University, Jinan 250101, China.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2015.2418320

judiciously select the beneficial memory blocks for locking. Specifically, we propose a branch-and-bound algorithm and an efficient heuristic approach.

Anand and Barua [3] have presented an instruction cache locking heuristic with the same objective. Their experiments confirm that locking is beneficial in improving the average-case performance. However, there are two major drawbacks in their work. First, they propose an iterative approach where detailed cache simulation is employed in every iteration to evaluate the cost/benefit of locking the memory blocks. Hence, the algorithm is quite inefficient specially for large applications. Moreover, they employ some approximations in the cost/benefit analysis to reduce the simulation cost leading to poor locking decisions. We show that our locking improves the prior work [3] in terms of both performance and efficiency.

We also compare cache locking with a complimentary technique called procedure placement [4]. The procedure placement techniques improve instruction cache performance through procedure reordering such that the conflict misses in the cache can be reduced. We show that procedure placement followed by cache locking can be an effective strategy in enhancing the instruction cache performance significantly.

This paper makes the following contributions to the state-of-the-art of cache optimizations for embedded system.

1) Cache locking modeling techniques based on TRP that accurately capture the cost and benefit of cache locking.
2) Cache locking algorithms that balance the cost and benefit of locking and judiciously select the beneficial memory blocks for locking.
3) Combined cache locking and procedure placement technique for cache performance improvement.

We demonstrate the advantages of our locking techniques using benchmarks from MiBench and MediaBench suites. Experiments indicate that for 4 KB cache our technique improves the performance and power consumption by 12% and 13% on average, respectively.

II. Related Work

Prior optimization techniques using cache locking mainly aim to improve the WCET for hard real-time systems. However, as we will demonstrate in this paper, cache locking can be quite effective in improving the average-case execution time for general embedded applications. In the following, we will summarize the techniques of using cache locking for improving the timing predictability of hard real-time systems and the average-case performance of general embedded systems, and other related cache optimization techniques.

A. Locking for Hard Real-Time Systems

In hard real-time systems, WCET is an important input to the schedulability analysis of multitasking real-time systems. Complex architecture features such as caches, are problematic for WCET estimation due to their timing unpredictability. Static analysis techniques have been widely used to bound the WCET [5]–[7] for hard real-time systems.

Cache locking improves the timing predictability as the contents of cache are statically known under locking. There exist two locking mechanisms, static cache locking [8]–[11] and dynamic cache locking [12]. However, all of the above techniques lock the entire cache. Recently, Ding *et al*. [13], [14] demonstrated that by partially locking the cache, WCET can be improved significantly. In the context of multitasking real-time systems, cache locking has been used to improve the processor utilization and tasks schedulability [15]–[18]. Data cache locking algorithms for WCET minimization are presented in [19]. Their techniques formulate cache miss equations to model the data reuses. Cache replacement policy is an important cache design parameter. Reineke *et al*. [20] analyzed and compared the timing predictability of different cache replacement policies.

B. Locking for General Embedded Systems

Cache locking can be effective for improving the average-case performance for general embedded applications too.

Data cache locking mechanism based on the length of the reference window for each data access instruction is proposed in [21]. However, they do not model the cost/benefit of locking and there is no guarantee of performance improvement. Anand and Barua [3] proposed an instruction cache locking algorithm for improving the average-case performance. However, there are mainly two disadvantages of their technique. First, Anand and Barau's approach relies on trace-driven simulation to evaluate the cost and benefit of cache locking. However, trace-driven simulation could be very slow, typically longer than the execution time of the program [22]. More importantly, in Anand and Barau's method, two detailed trace simulations are employed in each iteration where one iteration locks one memory block in the cache. Such extensive usage of simulation is not feasible for large applications. Second, in their method, cache locking benefit is approximated by locking dummy blocks to keep the number of simulations reasonable. Thus, the cost and benefit of cache locking are not precisely calculated in [3].

Liu *et al*. [23] used instruction cache locking for the same purpose. In their method, they represent the program using the probability execution flow tree and formulate optimization problems based on it. However, they do not consider the cache mapping function in their locking algorithm. They assume that any memory block can be locked in any cache set [as if the cache is a scratchpad memory (SPM)]. After the locking decisions are made, they have to use code placement techniques at instruction or basic block level to force the locked memory blocks to be mapped to the appropriate cache sets. However, this can lead to serious code size blowup, which has not been addressed.

In this paper, we introduce TRP to model cache behavior. Previously, reuse distance has been proposed for the same purpose [24], [25]. Reuse distance is defined as the number of distinct data accesses between two consecutive references to the same address and it accurately models the cache behavior of a fully associative cache. However, to precisely model the effect of cache locking, we need the content instead of the number (size) of the distinct data accesses between two consecutive references. Our TRP records both reuse content and their frequencies.

C. Other Cache and SPM Optimization Techniques

Caches play an important role for both average- and worst-case performance. The state-of-the-art average-case cache optimization techniques focus on design space exploration of cache parameters [26]–[28], code layout reorganization [4], cache reconfiguration [29], and cache partitioning [30]. In this paper, we focus on cache locking. Cache locking is complementary to the existing cache optimization techniques. For example, cache locking can work together with cache partitioning for multicores with shared cache [31]. In this paper, we focus on application specific embedded system, we will combine our cache locking with one of the state-of-the-art code layout reorganization technique [4] to further improve cache performance.

SPMs have been used as an alternative to caches for embedded systems [32], [33]. Both cache locking and SPM allocation try to improve the performance by carefully selecting memory blocks for either locking in the cache or allocation in the SPM. For SPM, the optimal data allocation always uses the SPM fully. However, for locking, partial cache locking is more appealing than full locking as shown [13]. In partial locking, locking a cache line with a single memory block will be compared with keeping it unlocked so that more than one memory block can benefit from it. If the latter wins, the cache line will not be locked. Our cache locking techniques partially lock the cache based on careful cost and benefit analysis.

III. Cache Locking Problem

In this section, we define the cache locking problem. In static cache locking, once a memory block is locked in a cache line, it can not be evicted from the cache. The instructions are locked in the cache at the beginning of program execution and remain locked throughout the program execution. Note that the mapping of instructions to the cache sets depend on the code memory layout. Inserting additional code for cache locking may tamper this layout. To avoid this problem, we use the trampolines [34] 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 some dummy null operation instructions at the entry point of the program that get replaced by a call to this trampoline after locking decisions are made. For static cache locking, the cost of executing the trampoline is very small as it is only executed once before the program starts.

A. Cache Terminology

A cache memory is defined in terms of four major parameters: block or line size L, number of sets K, associativity A (also known as cache way), and replacement policy. The block or line size determines the unit of transfer between the main memory and cache. A cache is divided into K sets. Each cache set, in turn, is divided into A cache blocks, where A is the associativity of the cache. For a direct-mapped cache $A = 1$, for a set-associative cache $A > 1$, and for a fully associative cache $K = 1$. In other words, a direct-mapped cache has only one cache block per set, whereas a fully-associative cache has only one cache set. Now the cache size is defined as $(K \times A \times L)$. A memory block m can be mapped to only one cache set given by (m modulo K).

For direct mapped caches, there is only one cache block per cache set. Locking it with a memory block means cache misses for all the remaining memory blocks mapped to the same cache set. For set associative caches, the remaining unlocked cache lines per cache set serve as a set associative cache with reduced associativity. For set-associative or fully-associative caches, the replacement policy (e.g., least recently used (LRU), first in first out (FIFO), etc.) defines the block to be evicted when a cache set is full. In this paper, we model the cache based on the LRU replacement policy where the block replaced is the one that has been unused for the longest time. But as we will demonstrate in the experiment section, our technique is effective for other replacement policies, too (e.g., FIFO).

Two locking mechanisms are commonly used in modern embedded processors—way locking and line locking. In way locking, particular ways of a set associative cache are selected for locking and these ways are locked for all the cache sets. Way-locking is employed by ARM processor series. Compared to way locking, line locking is a finer grained locking mechanism. In line locking, different number of cache lines can be locked for different cache sets. Line locking is employed by Intel's Xcale, ARM9 family, and Blackfin 5xx family processors. We consider both line locking and way locking mechanisms in this paper.

Cache misses can be broadly categorized into cold (compulsory) misses, capacity misses, and conflicts misses. Cold misses are caused by the first reference to a memory block. Cache locking eliminates the cold miss, but at the same time introduces additional overhead to fetch and lock the memory block at the beginning of program execution (through the trampoline). Capacity misses are incurred due to the limited cache size and cannot be mitigated through locking. Indeed, locking a memory block in the cache reduces the cache capacity available to the remaining memory blocks and may negatively impact the cache hit rate. So, cache locking primarily targets to eliminate conflict misses while minimizing the negative impact on the unlocked memory blocks.

B. Cache Locking Problem

The goal of our cache locking is to determine the set of memory blocks to be locked such that the conflict cache misses are reduced and the program execution latency is improved.

IV. Cache Locking Modeling

In this section, we describe our cache locking modeling techniques. We rely on TRP to compute the cost and benefit of cache locking. TRP captures the temporal conflicts of memory accesses. Thus, using TRP, we can accurately determine the cache hits and misses. More importantly, TRP is more compact compared to memory trace and thus enables efficient cache locking algorithms. In the following, we formally define TRP and other related terms.

Let $\mathcal{T}$ be the memory trace (sequence of memory block references) generated by executing a program on the target

architecture. We use M_i to denote the set of all the memory blocks that are mapped to ith cache set C_i. Also given a memory block m, it is only mapped to a single cache set (m modulo K). Thus, for any two cache sets C_i, C_j, we have $M_i \cap M_j = \emptyset$. Therefore, the trace $\mathcal{T}$ can be partitioned into K traces $\mathcal{T}_1, \ldots, \mathcal{T}_K$—one corresponding to each cache set. The trace $\mathcal{T}_i$ corresponding to cache set C_i only contains the memory blocks M_i from the original trace $\mathcal{T}$. Finally, given a memory block $m \in M_i$, let us define the jth reference of m in the trace $\mathcal{T}_i$ as $m[j]$.

A memory block m benefits from cache locking as all its references will be cache hits. It is straightforward to quantize this benefit of cache locking. Let access_m be the total number of accesses to memory block m. Then by locking m, we will get access_m cache hits. That is

$$\text{hit}_m = \text{access}_m \quad \text{if } m \text{ is locked.}$$

However, locking memory block $m \in M_i$ in the cache set C_i may have negative impact on the other memory blocks $M_i \backslash \{m\}$. Therefore, in order to accurately compute the overall gain, we have to characterize this negative impact.

Theorem 1: Given two memory blocks $m, m' \in M_i$, if $m[j]$ is a cache miss before locking m', then $m[j]$ will remain a cache miss after locking m' in cache set C_i.

Proof: The proof follows directly from the inclusion property for LRU replacement policy. The inclusion property states that after any series of references, a smaller store always contains a subset of the blocks in the larger store. After locking m', the number of available cache blocks in the cache set C_i reduces by one. Clearly, if $m[j]$ (jth reference of m in the trace) was a cache miss (i.e., not present in the cache set) originally with more cache blocks, it will be a cache miss with one less cache block. ■

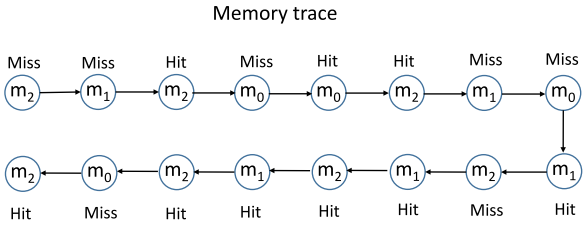
Definition 1 [Temporal Conflict Set (TCS)]: Given a memory reference $m[j]$ ($j > 1$) in the trace where $m \in M_i$, its $\text{TCS}_{m[j]}$ is defined as the set of unique memory blocks referenced between $m[j-1]$ and $m[j]$ in $\mathcal{T}_i$. If there is no such reference, then $\text{TCS}_{m[j]} = \emptyset$.

For example, in Fig. 1, the TCS of memory block m_2 is $\{m_1\}$ for its second reference and $\{m_0\}$ for its third reference. For a memory block reference, we determine its cache behavior (hit or miss) based on its TCS.

Theorem 2: If $|\text{TCS}_{m[j]}| \geq A$ for memory block $m \in M_i$, then the reference $m[j]$ will be a cache miss.

Proof: The proof follows directly from the definition of LRU replacement policy. As we bring in A or more unique memory blocks into the cache set, memory block m will be replaced from the cache and will incur a cache miss for its next reference. ■

Moreover, following Theorems 1 and 2, if $|\text{TCS}_{m[j]}| \geq A$, then $m[j]$ is guaranteed to be a cache miss irrespective of locking other memory blocks in the cache. Therefore, we can eliminate $\text{TCS}_{m[j]}$ from further consideration as far as cache locking decisions are concerned. Nevertheless, $m[j]$ will be a cache hit if m is locked. Hence, for each memory block m, we eliminate its $\text{TCS}_{m[j]}$ if $|\text{TCS}_{m[j]}| \geq A$, but keep its total number of access, access_m. This will guarantee that we can accurately compute the total number of cache hits and misses



Memory Blocks	Access	Temporal Reuse Profile
m_0	4	$\{<\{\emptyset\}, f(\{\emptyset\}) = 1>\}$
m_1	5	$\{<\{m_0\}, f(\{m_0\}) = 1>\}, \{<\{m_2\}, f(\{m_2\}) = 2>\}$
m_2	7	$\{<\{m_1\}, f(\{m_1\}) = 3>\}, \{<\{m_0\}, f(\{m_0\}) = 2>\}$

Fig. 1. TRP from a sequence of memory access for a 2-way set associative cache. Memory blocks m_0, m_1, and m_2 are mapped to the same cache set. Cache hits and misses are highlighted.

of the program. For example, in Fig. 1, for memory blocks m_0, m_1, and m_2, their number of memory accesses are recorded; the second reference to memory block m_1 is a cache miss and its TCS ($\{m_0, m_2\}$) can be removed.

Let Lock_i be the set of memory blocks locked in the cache set C_i. Clearly, $|\text{Lock}_i| \leq A$.

Theorem 3: If $|\text{TCS}_{m[j]}| < A$ for $m \in M_i \backslash \text{Lock}_i$, then $m[j]$ will be a cache miss only when $|\text{Lock}_i \cup \text{TCS}_{m[j]}| \geq A$.

Proof: As $|\text{TCS}_{m[j]}| < A$, the reference $m[j]$ is a cache hit before locking. Now as we lock memory blocks into the cache set C_i, the space available to accommodate the unlocked memory blocks will reduce. $m[j]$ will be a cache miss when the number of conflicting blocks and the locked memory blocks together exceeds the associativity of the cache. That is, $m[j]$ will be a cache miss when $|\text{Lock}_i \cup \text{TCS}_{m[j]}| \geq A$. ■

For example, in Fig. 1, the second reference of memory block m_2 will be a cache miss if m_0 is locked, because $|\{m_0, m_1\}| \geq 2$. However, it will remain as a cache hit if m_1 is locked.

Let $\mathcal{R}_m = \{\text{TCS}_{m[j]} : j > 1, |\text{TCS}_{m[j]}| < A\}$, i.e., $\mathcal{R}_m$ is the set of TCS for the references of m that result in cache hits in the original cache.

Definition 2 (Temporal Reuse Profile): The TRP_m of a memory block m is defined as a set of 2-tuples $\{\langle s, f(s)\rangle\}$ where $s \in \mathcal{R}_m$ and $f(s)$ denotes the frequency of the TCS s in the trace.

Fig. 1 shows an example of TRP given a memory trace. There are three memory blocks in the trace and the number of accesses for each of them is collected. These three memory blocks have different TCSs and thus different TRPs.

Given the TRP for a program execution and the locked memory blocks per cache set $\text{Lock}_i : i = 1 \ldots K$, we can now accurately compute the number of cache hits/misses for the entire program. For a memory block $m \in M_i$, is computed as follows:

$$\{\text{hit}_m | \text{Lock}_i\} = \begin{cases} \sum_{\substack{\forall \langle s, f(s)\rangle \in \text{TRP}_m \\ |s \cup \text{Lock}_i| < A}} f(s) & \text{if } m \notin \text{Lock}_i \\ \text{access}_m & \text{otherwise.} \end{cases} \quad (1)$$

In other words, if m is locked, then obviously all its accesses are cache hits; otherwise, we walk through all the possible

TCS scenarios in the TRP and determine cache hit or miss based on Theorem 3.

Then, the total number of cache hits for the cache set C_i is

$$\left\{\text{hit}_{\mathcal{T}_i}|\text{Lock}_i\right\} = \sum_{m\in M_i}\{\text{hit}_m|\text{Lock}_i\} \tag{2}$$

and the total number of cache hits for the entire program

$$\text{hit}_{\mathcal{T}} = \sum_{i=1}^{K}\left\{\text{hit}_{\mathcal{T}_i}|\text{Lock}_i\right\}. \tag{3}$$

V. Cache Locking Algorithm

Our cache locking algorithm consists of two phases: 1) profiling and 2) locking.

1) *Profiling Phase:* The profiling phase creates the TRP for each memory block in the program. This profiling can be achieved either by simulating the application or by executing the application on the target platform with a representative set of inputs. The simulation or execution creates the address trace. The TRP is built by a single pass through the address trace.
2) *Locking Phase:* The locking phase determines the set of memory blocks to be locked such that the program execution latency is minimized. Note that locking the entire cache, which is similar to the SPM allocation problem, does not ensure optimal performance. It is because locking a memory block may have negative impact on other unlocked memory blocks as the effective cache capacity is reduced. Our cache locking algorithms partially lock the cache based on cost-benefit analysis. More importantly, partial cache locking problem is more challenging 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 [13].

In the following, for each of the locking mechanism (line locking and way locking), we propose two algorithms to select the memory blocks to be locked. One is a branch-and-bound algorithm and the other one is a heuristic algorithm. In line locking, each cache set can be modeled independently. Thus, it is possible that different cache sets have different number of cache lines locked. In way locking, all the cache sets have to be considered together as the cache ways are locked for all the cache sets.

A. Line Locking

For line locking mechanism, both branch-and-bound and heuristic algorithms analyze each cache set individually.

1) Branch-and-Bound Algorithm: Our branch-and-bound algorithm systematically enumerates all the locking solutions that are organized to a tree form. Before evaluating the candidate solutions of a branch, branch-and-bound algorithm will compare the branch with the current best solution, and prune the branch if it can not produce a better solution than the best one found so far.

Algorithm 1: Branch-and-Bound Algorithm for Line Locking

```
1  foreach set C_i in the cache do
2      access_i = |T_i| ;
3      opt_sol := ∅; ;
4      h := {hit_{T_i}|∅} ;
5      min_lat := h × hit_lat + (access_i − h) × miss_lat ;
6      sort M_i based on the number of accesses.;
7      search(M_i, ∅);
8  Function (search(M, Lock))
9  if |Lock| = A or M = ∅ then
10     h := {hit_{T_i}|Lock};
11     lat := h×hit_lat+(access_i−h)×miss_lat+|Lock|×lock_lat
       ;
12     if lat < min_lat then
13         opt_sol := Lock;
14         min_lat := lat;
15     return;
16 Let m be the highest frequently accessed block in M;
17 M := M\m; ;
18 /* m is not locked */
19 cur_hit := Σ_{m'∈M_i\M} {hit_{m'}|Lock};
20 remain_hit := ComputeHitBound(M, Lock);
21 lat_bound := (cur_hit + remain_hit) × hit_lat + (access_i −
   cur_hit − remain_hit) × miss_lat + |Lock| × lock_lat ;
22 if lat_bound < min_lat then
23     search(M, Lock);
24 /* m is locked */
25 Lock := Lock ∪ m; ;
26 cur_hit := Σ_{m'∈M_i\M} {hit_{m'}|Lock};
27 remain_hit := ComputeHitBound(M, Lock);
28 lat_bound := (cur_hit + remain_hit) × hit_lat + (access_i −
   cur_hit − remain_hit) × miss_lat + |Lock| × lock_lat ;
29 if lat_bound < min_lat then
30     search(M, Lock);
```

a) Search algorithm: Our branch-and-bound algorithm is presented in Algorithm 1. Given a set of memory block M and the current locked list Lock (initially it is empty), the search function (line 8) in Algorithm 1 returns the set of locked memory blocks that gives the minimum execution latency. We sort the memory blocks in a cache set in the descending order of accesses (line 6). We use miss_lat, hit_lat and lock_lat to represent the latency of cache miss, cache hit, and locking one memory block, respectively.

The entire search space can be seen as a binary search tree as shown in Fig. 2. For each memory block m, we have to decide whether to lock it or not. Algorithm 1 covers the entire search space. Fig. 2 shows an example of binary search tree with five memory blocks. Each level of the tree corresponds to one memory block. The numbers on the edges represent locking decisions (i.e., 1 represents locked and 0 represents unlocked). We obtain a solution when a leaf node is reached or the entire cache set is locked (line 9). The number of cache hits is computed based on the cache modeling described in Section IV and the best solution (the lock list and minimum execution latency) is kept (lines 13 and 14). The locking overhead is taken into account when we compute the total execution latency (lines 11, 21, and 28).

The search function in Algorithm 1 is a recursive function. It explores the locking decision for one memory block at

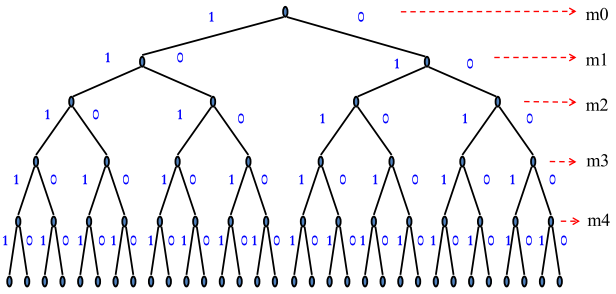


Fig. 2. Binary search tree of the branch-and-bound algorithm. There are five memory blocks in the example and each level corresponds to one memory block. The numbers on the edge represents locking decisions (i.e., 1 represents locked and 0 represents unlocked).

every recursion depth. During search, the recursion depth (D) divides the memory blocks into two categories—the current set of memory blocks (recursion depth $\leq D$) and the remaining memory blocks for future exploration (recursion depth $> D$). At any recursion depth (line 8), M denotes the remaining set of memory blocks for exploration. Thus, the current set of memory blocks is $M_i \backslash M$, where M_i is the set of memory blocks mapped to the cache set i. We use cur_hit to represent the cache hits of the current set of memory blocks given the current lock list. cur_hit is computed as follows:

$$\text{cur\_hit} = \sum_{m \in M_i \backslash M} \{\text{hit}_m | \text{Lock}\} \tag{4}$$

where Lock is the list of locked memory blocks for the current set of memory blocks (e.g., Lock $\subseteq M_i \backslash M$).

b) Pruning: To improve the search efficiency, search function computes the execution latency bound (lat_bound in lines 21 and 28) at every recursion depth (lines 21 and 28) and uses it to prune the search space if it is possible. More clearly, let min_lat be the minimum latency for all the solutions we have explored. If lat_bound $\geq$ min_lat, then it is impossible to find a better solution than the best solution found so far from the current branch and thus the branch can be safely pruned.

To compute the latency bound, we need to determine the upper bound of the cache hits for the current set of memory blocks ($M_i \backslash M$) and the remaining memory blocks (M), respectively. Let Lock$'$ be the list of locked memory blocks among the remaining memory blocks (Lock$' \subseteq M$), which gives the minimum execution latency. Clearly, $|\text{Lock} \cap \text{Lock}'| = \emptyset$ and $|\text{Lock} \cup \text{Lock}'| \leq A$.

Theorem 4: For a memory block $m \in M_i \backslash M$, $\{\text{hit}_m | \text{Lock}\} \geq \{\text{hit}_m | \text{Lock} \cup \text{Lock}'\}$.

Proof: $m \in M_i \backslash M$ and Lock$' \subseteq M$, so $m \notin$ Lock$'$. Given a memory reference $m[j]$ of memory block m, there are two cases.

1) $m[j]$ is a cache miss under the lock list Lock. Then, according to Theorem 1, $m[j]$ remains as a cache miss.
2) $m[j]$ is a cache hit under the lock list Lock. The inclusion property for LRU replacement policy states that after any series of references, a smaller store always contains a subset of the blocks in the larger store. After locking additional $|\text{Lock}'|$ memory blocks, the number of available cache blocks in the cache set reduces. Thus, $m[j]$ might be converted to a cache miss under the new lock list Lock $\cup$ Lock$'$ due to the smaller cache space. ■

Algorithm 2: Computation of remain_hit

```
1  Function(ComputeHitBound(M, Lock))
2  foreach memory block m ∈ M do
3  |  benefit_m := access_m − {hit_m|Lock} ;
4  Sort M into decreasing order according to benefit_m ;
5  remain_hit := Σ_{m∈M} {hit_m|Lock};
6  size := 0 ;
7  foreach memory block m ∈ M do
8  |  remain_hit := remain_hit + benefit_m ;
9  |  if size ≥ A − |Lock| then
10 |  |  break ;
11 |  size := size + 1;
12 return remain_hit ;
```

Therefore, for a memory block m in the current set ($m \in M_i \backslash M$), $\{\text{hit}_m | \text{Lock}\}$ is the upper bound of the cache hits of m when exploring the remaining memory blocks. Subsequently, cur_hit (4) is the upper bound of the total number of cache hits for the current set of memory blocks when exploring the remaining memory blocks M.

For the remaining memory blocks (M), we use remain_hit to represent its upper bound of the number of cache hits. This bound is returned by function ComputeHitBound (Algorithm 2 to be described later). We define Hit as the number of cache hits for the remaining memory blocks M for this case, then

$$\text{Hit} = \sum_{m \in \text{Lock}'} \text{access}_m + \sum_{m \in M \backslash \text{Lock}'} \{\text{hit}_m | \text{Lock} \cup \text{Lock}'\}. \tag{5}$$

Finding Lock$'$ is the exact problem solved by Algorithm 1. Here, we attempt to derive an upper bound for Hit without knowing Lock$'$. Following Theorem 4, we have:

$$\begin{aligned} \text{Hit} &\leq \sum_{m \in \text{Lock}'} \text{access}_m + \sum_{m \in M \backslash \text{Lock}'} \{\text{hit}_m | \text{Lock}\} \\ &= \sum_{m \in \text{Lock}'} \text{access}_m - \sum_{m \in \text{Lock}'} \{\text{hit}_m | \text{Lock}\} \\ &\quad + \sum_{m \in M} \{\text{hit}_m | \text{Lock}\}. \end{aligned} \tag{6}$$

For a memory block $m \in M$, we define benefit$_m$ as the additional number of cache hits achieved by locking m

$$\text{benefit}_m = \text{access}_m - \{\text{hit}_m | \text{Lock}\}. \tag{7}$$

By combining (6) and (7), we have

$$\text{Hit} \leq \sum_{m \in \text{Lock}'} \text{benefit}_m + \sum_{m \in M} \{\text{hit}_m | \text{Lock}\}. \tag{8}$$

The locked memory blocks list for the remaining memory blocks $|\text{Lock}'| \leq A - |\text{Lock}|$. Let $M' \in M$ be the set of top $|A - \text{Lock}|$ memory blocks with the maximum benefit values. Thus

$$\text{Hit} \leq \sum_{m \in M'} \text{benefit}_m + \sum_{m \in M} \{\text{hit}_m | \text{Lock}\}. \tag{9}$$

Algorithm 2 presents the computation details of remain_hit following (9). However, in the worst case, the complexity of

Algorithm 3: Heuristic Cache Line Locking Algorithm

```
1  foreach set C_i in the cache do
2      access_i = |T_i| ;
3      Lock_i := ∅; flag := TRUE;
4      cur_hit := {hit_{T_i}|Lock_i};
5      min_lat := cur_hit × hit_lat + (access_i − cur_hit) × miss_lat;
6      while flag do
7          benefit := 0;
8          foreach m ∈ M_i\Lock_i do
9              new_hit_m := {hit_{T_i}|Lock_i ∪ {m}};
10             lat_m := new_hit_m × hit_lat + (access_i −
               new_hit_m) × miss_lat + (|Lock_i| + 1) × lock_lat ;
11             if (min_lat − lat_m) > benefit then
12                 benefit := min_lat − lat_m;
13                 selected_block := m;
14         if benefit > 0 then
15             Lock_i := Lock_i ∪ selected_block;
16             min_lat := min_lat − benefit;
17         else
18             flag := FALSE;
19         if |Lock_i| = A then
20             flag := FALSE;
```

branch-and-bound algorithm is as high as that of exhaustive search. Next, we also propose an efficient heuristic approach.

2) Heuristic Approach: Our heuristic is iterative in nature and exploits the cache locking modeling techniques described in Section IV. As each cache set can be modeled independently, the iterative algorithm is applied for each cache set separately. So given a cache set C_i, our goal is to determine Lock_i such that the execution latency is minimized.

Initially, we set $\text{Lock}_i = \emptyset$ and compute the number of cache hits in the original cache

$$\text{cur\_hit} = \{\text{hit}_{\mathcal{T}_i}|\emptyset\}.$$

In each iteration, we go through all the unlocked memory blocks in the cache set $m \in M_i \backslash \text{Lock}_i$ and compute the number of cache hits and execution latency if m was locked in the cache

$$\begin{aligned}\text{new\_hit}_m &= \{\text{hit}_{\mathcal{T}_i}|\text{Lock}_i \cup \{m\}\}\\ \text{lat}_m &= \text{new\_hit}_m \times \text{hit\_lat} + (\text{access}_i - \text{new\_hit}_m)\\ &\quad \times \text{miss\_lat} + (|\text{Lock}_i| + 1) \times \text{Lock\_lat} \quad (10)\end{aligned}$$

where access_i is the number of references of cache set C_i.

Let

$$\text{benefit} = \max_{m \in M_i \backslash \text{Lock}_i} (\text{min\_lat} - \text{lat}_m)$$

where min_lat is the minimum execution latency for the solutions we have explored. If benefit ≤ 0, then locking any of the remaining memory blocks would worsen the performance and we will terminate our iterative algorithm. Otherwise, we choose the memory block m with the maximum benefit. We break ties arbitrarily. The algorithm also terminates when $|\text{Lock}_i| = A$, i.e., we have locked all the cache blocks in the cache set. Our cache locking algorithm is detailed in Algorithm 3.

Algorithm 4: Branch-and-Bound Algorithm for Way Locking

```
1  cur_sol = ∅  cur_lat = 0 ;
2  foreach set C_i in the cache do
3      cur_hit := {hit_{T_i}|∅} ;
4      opt_lat :=
       cur_lat + cur_hit × hit_lat + (access_i − cur_hit) × miss_lat ;
5  opt_sol = ∅ ;
6  for way ← 1 to |A| do
7      cur_lat = 0; cur_sol = ∅ ;
8      foreach set C_i in the cache do
9          sort M_i based on the number of accesses.;
10         min_lat = inf ;
11         < sol, lat > := search(M_i, ∅, way) ;
12         cur_lat := cur_lat + lat;
13         cur_sol := cur_sol ∪ sol;
14     if cur_lat < opt_lat then
15         opt_lat := cur_lat ;
16         opt_sol := cur_sol ;
17         opt_way := way ;
18 function(search(M, Lock, way))
19 if |Lock| = way or M = ∅ then
20     compute latency and update the best solution;
21     return;
22 M := M\m; ;
23 cur_hit := Σ_{m'∈M_i\M} {hit(m')|Lock};
24 remain_hit := ComputeHitBound(M, Lock, way);
25 compute lat_bound;
26 if lat_bound < min_lat then
27     search(M, Lock, way);
28 Lock := Lock ∪ m; ;
29 cur_hit := Σ_{m'∈M_i\M} {hit(m')|Lock};
30 remain_hit := ComputeHitBound(M, Lock, way);
31 compute lat_bound;
32 if lat_bound < min_lat then
33     search(M, Lock, way);
```

a) Complexity: Let w the average analysis time of computing the number of cache hits per cache set (2). w depends on the TRP of the application, cache parameters (e.g., cache size, associativity), and locked list. The complexity of the heuristic algorithm is $O(w \cdot M_s \cdot A \cdot K)$, where M_s is the average number of memory blocks per cache set of the application, A is the cache associativity, and K is the number of cache sets.

B. Way Locking

In way locking mechanism, a particular number of cache ways are locked for the entire cache. Thus, all the cache sets need to be considered together.

1) Branch-and-Bound Algorithm: The branch-and-bound algorithm for way locking is presented in Algorithm 4. Initially, it computes the latency without locking any cache ways (lines 3 and 4). Let way be the number of locked cache ways. Then, Algorithm 4 walks through all the possible values for way ($1 \leq \text{way} \leq A$) and compares them. For each possible value for way, it uses branch-and-bound search (search function at line 11) to find the optimal solution (e.g., way number of memory blocks that gives the minimal latency) for each cache set. The search function used in Algorithm 4 is similar to the search function in Algorithm 1 but with a few differences. First, the search function in Algorithm 4 is extended

Algorithm 5: Heuristic Way Locking Algorithm

```
1  cur_sol = ∅  cur_lat = 0 ;
2  foreach set C_i in the cache do
3  |   Lock_i := ∅ ;
4  |   cur_hit := {hit_{T_i}|∅} ;
5  |   opt_lat :=
   |   opt_lat + cur_hit × hit_lat + (access_i − cur_hit) × miss_lat ;
6  way := 1; flag := TRUE ;
7  while flag do
8  |   cur_lat := 0; cur_sol := ∅ ;
9  |   foreach set C_i in the cache do
10 |   |   min_lat = inf;
11 |   |   foreach m ∈ M_i\Lock_i do
12 |   |   |   new_hit_m := {hit_{T_i}|Lock_i ∪ {m}};
13 |   |   |   lat_m := new_hit_m × hit_lat + (access_i −
   |   |   |   new_hit_m) × miss_lat + (|Lock_i| + 1) × lock_lat ;
14 |   |   |   if (lat_m < min_lat) then
15 |   |   |   |   min_lat := lat_m ;
16 |   |   |   |   selected_block := m;
17 |   |   cur_lat := cur_lat + min_lat ;
18 |   |   Lock_i := Lock_i ∪ selected_block ;
19 |   |   cur_sol := cur_sol ∪ selected_block ;
20 |   benefit := opt_lat − cur_lat;
21 |   if benefit > 0 then
22 |   |   opt_lat := cur_lat;
23 |   |   opt_sol := cur_sol ;
24 |   else
25 |   |   flag := FALSE;
26 |   if way = A then
27 |   |   flag := FALSE;
28 |   way := way + 1;
```

with an extra argument way (line 11), the target number of locked cache ways. The search process will be terminated if the locked cache lines is equivalent to the target way value. Second, when the ComputeHitBound function (Algorithm 2) is called in the search function, only way number of memory blocks can be locked. Finally, Algorithm 4 sums the total latency for all the cache sets and determines the best way value (lines 6–17).

2) Heuristic Approach: We also propose an efficient heuristic algorithm for way locking. Our heuristic way locking algorithm detailed in Algorithm 5 is iterative in nature. For each iteration, Algorithm 5 selects the memory block with the minimum latency from the remaining unlocked memory blocks for each cache set. Then, it sums the total latency for all the cache sets. Algorithm 5 stops to increase the number of locked cache ways until either the total latency of the current iteration (way) is larger than the previous iteration (way $-$ 1) or the entire cache is locked. Similar to the line locking, the complexity of the heuristic algorithm is $O(w \cdot M_s \cdot A \cdot K)$.

C. Code Layout

The performance of the cache locking algorithm critically depends on the code memory layout. In the discussion so far, we have assumed that we start with the "natural" code layout. However, instruction cache performance can be improved throughout procedure placement—reordering procedures so that cache conflicts are reduced [4]. Clearly, procedure placement and cache locking are complementary approaches. In the experiments, we will evaluate the effects of cache locking, procedure placement, and a combined approach. For procedure placement, we choose temporal block profile (TBP) [4]—a state-of-the-art procedure placement technique. In TBP, the memory block conflicts among procedures are modeled using temporal relationship (i.e., which procedures and memory blocks are referenced between two consecutive accesses to another memory block). Then, the TBP is used along with the cache configuration and procedure sizes to estimate the cost/benefit of procedure placement and determine the procedure locations. Compared to previous procedure placement techniques, TBP achieves higher cache miss improvement thanks to its accurate modeling [4].

D. Discussion

In this paper, we focus on the static locking where the locking routine is executed only once at the beginning of the program and remains unchanged. In the experiments, we will demonstrate that our static locking achieves substantial improvement for the widely used embedded applications. However, static locking may not be effective for large programs where a large number of memory blocks compete for limited cache resource. For these applications, we can use dynamic cache locking to overcome the cache space limitation through time multiplexing. More clearly, we can partition the program into regions based on the program phase behavior [12], [23] and use our static locking algorithms for each region. However, for dynamic cache locking, the locking overhead and code layout change due to the insertions of locking routines have to be taken into account.

We use instruction cache to demonstrate the benefit of cache locking. Instruction cache is important for embedded systems as instructions are fetched every clock cycle. But our techniques are equally applicable to data caches. Given the instruction access trace, our algorithms determine the locked memory blocks. Similarly, our techniques can be used for data caches provided with the data access trace. In order to do this, we can execute the program on the target architecture and generate the entire memory trace and each memory access is associated with its type (instruction or data).

VI. Experimental Evaluation

A. Experimental Setup

We select benchmarks from MiBench and MediaBench for evaluation purpose. The benchmarks and their characteristics are shown in Table I. We conduct our experiments using SimpleScalar framework [35]. We evaluate our techniques with different cache parameters. We vary the cache size (2, 4, 8, and 16 KB) and cache associativity (1, 2, 4, 8), but keep the block size constant (32 bytes). The extra code to fetch and lock memory blocks are inserted at the end of the program as a trampoline. Thus, it will not affect the original program layout. As we are modeling the instruction cache, we assume a simple in-order processor with unit-latency for all the data memory references. The cache hit latency is 1 cycle and the cache miss penalty is 100 cycles. We assume 150 cycles

TABLE I
CHARACTERISTICS OF BENCHMARKS

Benchmark	Description	Trace (MB)		TRP (KB)								Code Size (bytes)
				4KB Cache				8KB Cache				
		input1	input2	1way	2way	4way	8way	1way	2way	4way	8way	
Adpcm	differential pulse-code modulation	220	118	3.3	3.6	4.5	6.2	4.3	4.1	5.2	7.5	15,264
Sha	secure hash algorithm	103	220	4.1	4.8	6.8	11	5.1	5.7	7.5	13	17,600
Rijndael	encryption standard algorithm	147	62	8.7	9.1	11	13	9.7	9.8	12	15	32,256
Blowfish	symmetric block cipher	318	2,000	5.6	7.2	11	16	6.6	8.3	13	24	21,024
Dijkstra	shortest path algorithm	293	742	5.5	7	11	17	6.6	8	12	23	19,744
Bitcnts	bit manipulation algorithm	170	32	2.7	2.5	2.8	4	3.7	3.1	3.1	4	10,176
Basicmath	mathematical calculations	1,400	1,400	15	20	30	56	16	22	35	64	45,728
Qsort	quick sort algorithm	360	2000	6.6	8.5	13	18	7.7	9.3	14	25	25,280
Susan	image recognition algorithm	220	2,000	5.8	7.5	11	16	6.8	8.1	12	19	30,400
Stringsearch	string search algorithm	39	78	4.7	5.9	8.9	18	5.7	7	13	31	16,864
FFT	fast fourier transform	790	736	13	17	27	68	14	17	23	38	40,128
Jpeg	image compression/decompression	235	869	17	27	52	133	19	28	50	103	79,104
Lame	mp3 encoder	965	2,000	45	67	129	314	47	65	105	283	181,472
Gsm	voice encoding/decoding algorithm	297	1100	16	18	22	31	17	19	22	28	60,896
Mpeg2dec	mpeg2 decoder algorithm	222	2,000	13	19	33	76	14	20	34	64	56,448

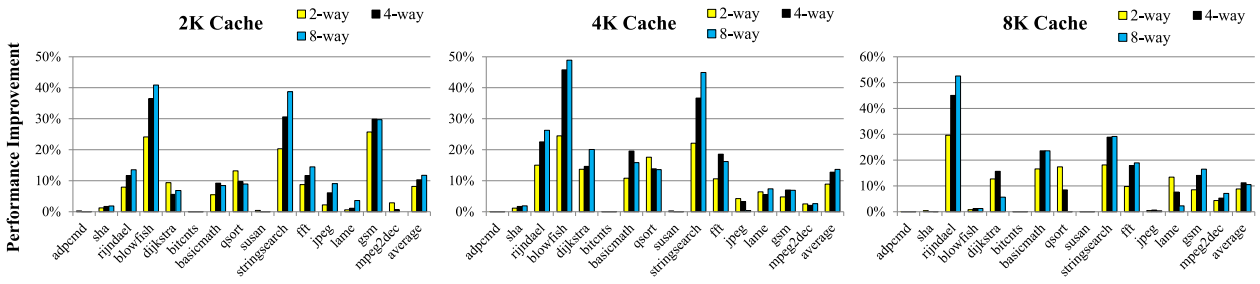


Fig. 3. Performance improvement over cache without locking for various cache configurations.

for locking and unlocking a memory block. The code size of each program is shown in the last column of Table I.

We generate the instruction trace of each benchmark using sim-profile, a functional simulator. Given the address trace and the cache configuration, we can easily create the TRP. We use multiple different inputs to evaluate our techniques. Table I gives the trace size of inputs 1 and 2. The TRP size of input 1 for 4 and 8 KB caches are also shown in Table I. For each cache configuration, its TRP size is significantly more compact compared to the address trace (KB versus MB or GB). The TRP size depends on the cache configuration because the number of cache hits varies for different cache configurations and TRP only records the cache hits as only cache hits are affected by locking. The TRP size also depends on the TCS, which might be different for different cache configurations. We observe that for the same size cache, most likely the size of TRP increases with the associativity. For the same size of cache, when the associativity (the number of cache ways) increases, the number of cache sets will reduce to keep the cache size constant. Thus, there are more memory blocks mapped to each cache set for the highly associative caches. Memory blocks mapped to the same cache set will conflict with each other. Hence, the TCS will be more complicated for the highly associative caches. This explains why TRP size increases with the associativity. Similar findings have been observed for 2 and 16 KB caches, which are shown due to space limitation.

We propose two locking mechanisms: 1) line locking and 2) way locking. For each locking mechanism, we propose two algorithms: a branch-and-bound algorithm and a heuristic algorithm. In the following, we perform two sets of experiments to evaluate our technique. First, we present the performance and power consumption improvement of our heuristic line locking technique in Section VI-B. Second, we compare different locking mechanisms, locking algorithms, code memory layouts, and cache replacement policies in Section VI-C.

B. Performance Results

We use heuristic line locking as our default locking technique. Here, we first evaluate its performance and energy consumption improvement. In Section VI-C, we compare line and way locking mechanisms. For each cache size, we vary the associativity from 1 to 8. However, for any cache size, the performance and energy improvement for direct mapped cache is minimal. This is expected as only one block is available per cache set and locking that block implies cache misses for all the remaining memory blocks mapped to the same cache set. Hence, in the following, we focus on the high associativity caches (2-, 4-, and 8-way).

1) Performance Improvement: Fig. 3 shows the performance (e.g., cycles) improvement for various cache sizes (2, 4, and 8 KB). For each cache size, we vary the associativity from 2 to 8. Some benchmarks (e.g., adpcmd, sha, etc.) do not gain considerable performance improvement because the absolute cache misses for them are small. Thus, the improvement in cache misses will not contribute much to the overall performance improvement. But for the benchmarks with high number of cache misses (e.g., blowfish, gsm, etc.), cache locking achieves substantial performance improvement.

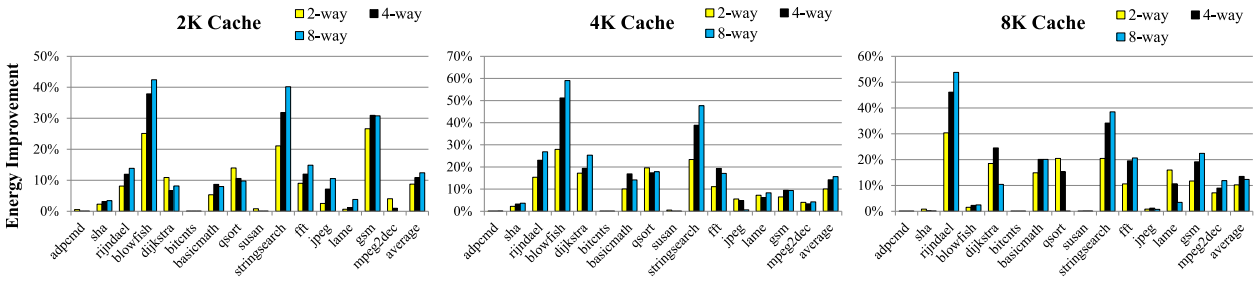


Fig. 4. Energy consumption improvement over cache without locking for various cache configurations.

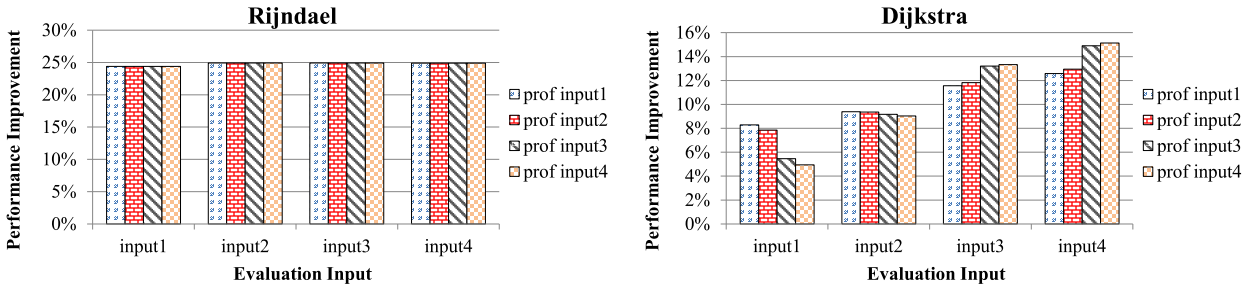


Fig. 5. Evaluations with more than two inputs for Rijndael and Dijkstra.

Overall, we obtain 10% improvement on average for 2 KB cache, 12% improvement on average for 4 KB cache, and 11% improvement on average for 8 KB cache.

2) Energy Consumption Improvement: Fig. 4 shows the memory hierarchy energy consumption improvement for various cache sizes (2, 4, and 8 KB). For each cache size, we vary the associativity from 2 to 8. For different cache configurations, the energy consumed per cache access is different. We model the energy consumption of different cache configurations using the CACTI [36] model for 0.13 μm technology. As for the energy consumption of one access to memory, it is assumed to be 200 times of energy consumption of cache hit [29]. We obtain 11% improvement on average for 2 KB cache, 13% improvement on average for 4 KB cache, and 12% improvement on average for 8 KB cache.

3) Large Caches: We also evaluate our techniques using 16 KB cache. On average, we obtain 7% and 9% improvement for performance and energy consumption, respectively. Compared to smaller caches (2, 4, and 8 KB), the improvement of large cache (16 KB) is smaller. This is because the number of cache misses decreases given a larger cache, which means there are less opportunities for cache locking to improve.

4) Evaluation With Two Inputs: Our locking algorithm has two phases: 1) profiling and 2) locking phases as described in Section V. In the profiling phase, it builds the TRP based on a set of representative inputs. However, in reality, it might be difficult to identify representative inputs for various programs. For the above experiments, we use the same input for profiling and evaluation (input 1 in Table I). Here, we evaluate the our technique (heuristic) when the profiling and evaluation inputs are different. More clearly, we build the TRP and determine the memory blocks to lock using input 1 and then evaluate the performance improvement using input 2. Overall, we still obtain high performance improvement. We obtain 10% improvement on average for 2 KB cache, 11% improvement on average for 4 KB cache, 10% improvement on average for 8 KB cache, and 6% improvement on average for 16 KB cache.

5) Evaluation With More Than Two Inputs: We also evaluated our technique (heuristic) using more than two input sets. We use benchmarks Rijndael and Dijkstra as examples. For each benchmark, we use four different inputs and the inputs are obtained from the Midatasets [37]. Then, for each input, we use it as the profiling input and evaluate across all the four inputs. The results are shown in Fig. 5. The performance improvement shown in Fig. 5 is the average performance improvement across all the cache configurations. For benchmarks with predictable behavior like Rijndael, there is very little variation across different inputs. Thus, for these applications, any input can be a representative input. On the other hand, for benchmarks with dynamic behavior like Dijkstra, different inputs can lead to different performance improvement. This is because different inputs may exercise different program paths. But we still get significant performance improvement compared to the cache without locking for all test inputs. Moreover, given a test input, the variation across different profile inputs is small (except for input 1). Nevertheless, for such dynamic applications, multiple different input sets should be chosen carefully to increase the likelihood of covering all the input-dependent branches and accurately create the representative inputs. We leave this as future work.

6) Partial Locking Results: Our cache locking is guided by careful cost-benefit analysis. On one hand, if it is beneficial to lock one memory block so that its cache misses are converted to cache hits, then our cache locking solution will lock it. On the other hand, if it is beneficial to keep a cache line unlocked so that multiple memory blocks can benefit from it, then our cache locking solution will not lock it. Through judicious cost-benefit analysis, our technique only partially locks the cache. Table II gives the percentage of cache lines locked in the cache for different cache configurations. As shown,

TABLE II
PERCENTAGE OF LOCKED CACHE LINES

Benchmark	2KB Cache (%)				4KB Cache (%)				8KB Cache (%)			
	1-way	2-way	4-way	8-way	1-way	2-way	4-way	8-way	1-way	2-way	4-way	8-way
Adpcm	0	32.81	32.81	35.94	0.78	22.66	28.13	41.41	0.39	10.94	12.50	14.83
Sha	3.13	43.75	68.75	73.44	4.69	33.60	50.00	61.72	2.34	16.80	19.14	19.92
Rijndael	0	50.00	75.00	87.50	0	50.00	75.00	87.50	5.86	51.56	75.00	87.50
Blowfish	3.13	50.00	75.00	87.50	7.03	42.19	74.22	87.50	4.69	24.22	36.33	40.23
Dijkstra	3.13	43.75	65.63	68.75	3.91	37.50	35.94	67.19	2.34	23.05	28.13	21.88
Bitcnts	0	23.44	28.13	28.13	7.81	18.75	40.63	47.66	0	6.64	5.47	1.56
Basicmath	0	35.94	50.00	32.81	0	40.63	57.81	47.66	3.13	46.88	62.89	70.70
Qsort	1.56	39.06	31.25	37.50	3.91	28.13	33.60	35.94	1.17	26.17	35.16	25.00
Susan	0	17.19	17.19	20.31	0	23.44	25.78	14.84	0.78	21.09	28.91	23.44
Stringsearch	7.81	50.00	75.00	87.50	8.59	51.56	75.00	87.50	5.86	33.20	50.00	64.45
FFT	0	31.25	45.31	56.25	1.56	29.69	37.50	29.69	4.69	32.81	52.34	56.25
Jpeg	0	18.75	17.19	23.44	0	20.31	21.09	25.00	1.17	24.61	26.17	21.88
Lame	0	3.13	6.25	15.63	0	15.63	15.63	14.06	0	17.97	28.52	35.16
Gsm	1.56	50.00	64.06	60.94	0	43.75	64.84	62.50	0	49.61	72.66	84.77
Mpeg2dec	0	17.19	10.94	0	0.78	28.13	37.50	39.84	1.17	40.63	59.77	67.67

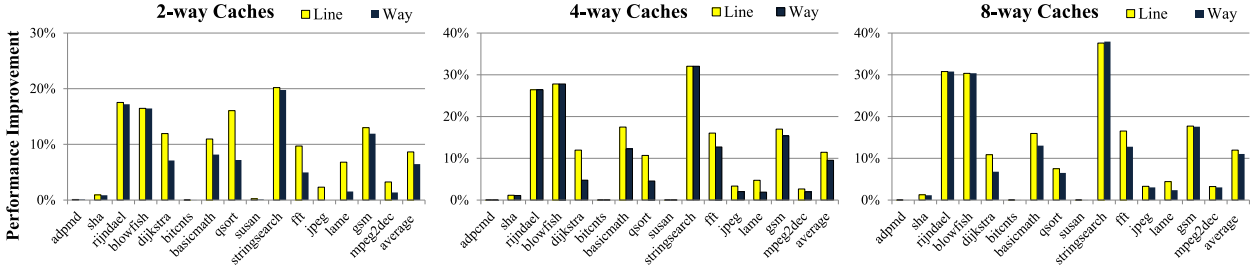


Fig. 6. Performance improvement comparison of heuristic line and way locking for different cache associativity.

for all the benchmarks, our cache locking algorithm locks only a fraction of the cache lines.

For most of the benchmarks and cache configurations, the percentage of the locked cache lines increases with the associativity for the same size cache. Higher associativity cache provides more opportunities for cache locking as more TCSs can fit into it. However, this is not always true (e.g., Bitcnts in 8 KB cache and Mpeg2dec in 2 KB cache). For higher associativity caches, their TCSs tend to be more complex as the number of cache sets is reduced and there are more memory blocks mapped to each cache set. When TCSs become complex, locking one memory block may result in cache misses for other memory blocks. For example, for Mpeg2dec in 2 KB and 8-way cache, it turns out locking any memory block will cause negative performance improvement. Thus, our locking algorithm chooses not to lock for this case.

C. Comparison

1) Line Locking Versus Way Locking: There exist two locking mechanisms: 1) line locking and 2) way locking. In line locking, different number of cache lines can be locked for different cache sets. In way locking, same number of cache lines are locked for all the cache sets. Fig. 6 compares the performance improvement of line and way locking for the heuristic algorithms. For different associativity (2-, 4-, and 8-way), Fig. 6 computes the average performance improvement across different size caches (2, 4, and 8 KB). We observe that for the benchmarks that have different memory behaviors across cache sets (e.g., Dijkstra, qsort), way locking that forces all the cache sets lock the same number of cache lines performs worse than line locking. For the benchmarks that have almost

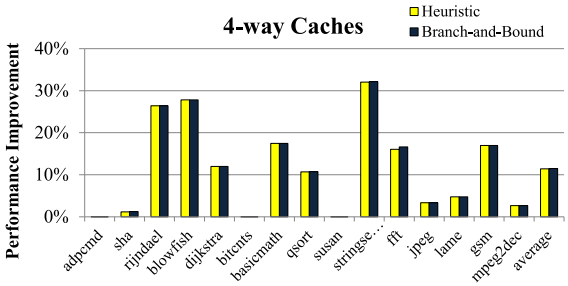


Fig. 7. Performance improvement comparison of heuristic and branch-and-bound line locking algorithms for different cache associativity.

unique memory behaviors across the cache sets (e.g., Rijndael, gsm), way locking is as good as line locking. For 2-way caches, line locking improves performance by 8.6% on average while way locking improves performance by 6.4% on average. For 4-way caches, line locking improves performance by 11.4% on average while way locking improves performance by 9.5% on average. For 8-way caches, line locking improves performance by 12% on average while way locking improves performance by 11% on average.

2) Heuristic Versus Branch-and-Bound: For both line locking and way locking, we propose two algorithms. One is a branch-and-bound and another is a heuristic. We first compare the performance improvement of the heuristic and branch-and-bound algorithms for line locking. Fig. 7 shows the average performance improvement of 4-way associative caches across different sizes (2, 4, and 8 KB). As shown, the heuristic results are very close to the results of branch-and-bound algorithm. For 2-way set associative caches, the

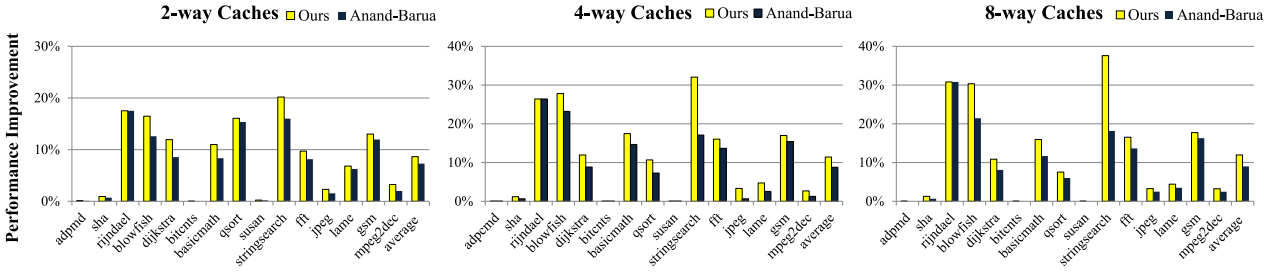


Fig. 8. Performance improvement comparison of ours and Anand–Barua techniques.

TABLE III
RUNTIME COMPARISON

Benchmark	Runtime (sec)		
	Heuristic line locking	Anand-Barua	Speedup
Adpcm	11.62	2178	187
Sha	5.412	1092	201
Rijndael	8.31	2289	275
Blowfish	17.38	3558	204
Dijkstra	15.93	3120	195
Bitcnts	9.37	1458	155
Basicmath	80.98	21312	263
Qsort	20.54	3970	193
Susan	11.79	2561	217
Stringsearch	2.16	425	196
FFT	47.66	11418	239
Jpeg	14.43	2505	173
Lame	73.96	12979	175
Gsm	16.78	4766	284
Mpeg2dec	12.64	2668	211

heuristic solution improves the performance by 8.6% on average, while the branch-and-bound algorithms improves it by 8.7% on average. For 4-way associative cache, the heuristic achieves 11.4% performance improvement on average, while the branch-and-bound algorithm returns 11.5% improvement. As for the runtime of the algorithms, the heuristic is 1–273 times faster than the branch-and-bound algorithm for low associativity caches ($\leq$4). For 8-way associative cache, the heuristic returns solutions quite fast (Table III), but the branch-and-bound algorithm fails to terminate within 10 hours for some big benchmarks.

Similar to line locking, our heuristic results are very close to the results of branch-and-bound for way locking. For 2-way set associative caches, the heuristic solution improves the performance by 6.4% on average, while the branch-and-bound algorithm improves it by 6.5% on average. For 4-way set associative caches, the heuristic solution improves the performance by 9.5% on average, while the branch-and-bound algorithm improves it by 9.9% on average.

3) Comparison With Anand–Barua Method: We compare our line locking heuristic with Anand–Barua method [3]—a state-of-the-art technique in the literature targeting cache locking for the average-case performance improvement. Anand–Barua method is based on line locking mechanism. Their proposal is an iterative simulation-based heuristic and needs feedback from trace-driven simulator in each iteration. We implement their algorithm and compare against our line locking heuristic both in terms of performance and efficiency.

In terms of performance improvement, our approach generally performs better or at least equal compared to Anand–Barua's method for every cache configuration. Fig. 8 shows the average performance improvement for 2-, 4-, and 8-way caches. For each way value, the improvement shown in Fig. 8 is the average performance improvement across all cache sizes (2, 4, and 8 KB) as both methods do not gain much for direct mapped caches. As evident from Fig. 8, our heuristic achieves higher performance improvement than Anand–Barua's method across all the benchmarks and cache configurations. For the benchmarks blowfish and stringsearch, the improvement over Anand–Barua's method are about 20% for some configurations. This is because our cache modeling is accurate whereas the cache hits/misses are approximated in Anand–Barua's work.

Anand–Barua method invokes cache simulation in each iteration. However, cache simulation can be very slow. In addition, in their technique, the number of simulations required grows linearly with the total number of locked memory blocks. When the number of memory blocks locked is not small, simulation based approach may not be feasible. In contrast, we only need one round of profiling and the subsequent analysis relies only on compact TRPs. The runtime comparison of our heuristic and Anana-Barua's method is detailed in Table III. The time presented is the average runtime across all the tested cache configurations (2, 4, and 8 KB cache). Our approach is 155–284 times faster compared to Anand–Barua's method.

4) Code Memory Layout: As discussed in Section V-C, procedure placement and instruction cache locking are complementary approaches and cache performance can benefit significantly through a combination of these two approaches. Here, we first compare locking and procedure placement [4] and then combine them together. For procedure placement, we choose TBP [4]—a state-of-the-art procedure placement technique. Procedure placement techniques are effective for applications with substantial number of procedures. Hence, we compare cache locking and procedure placement using large applications Jpeg, Lame, and Mpeg2. Fig. 9 compares the performance improvement of TBP and cache locking. We note that procedure placement performs very well for direct mapped caches, while cache locking achieves very small or no improvement at all. In general, procedure placement is a good choice for the low associativity caches (1 or 2), while locking is more suitable for the higher associativity caches (2, 4, and 8). This is because higher associativity leads to fewer cache sets leaving little opportunity for procedure reordering. In contrast, higher associativity provides more opportunities for cache locking. We also evaluated a combined locking and layout optimization. We first perform procedure placement for

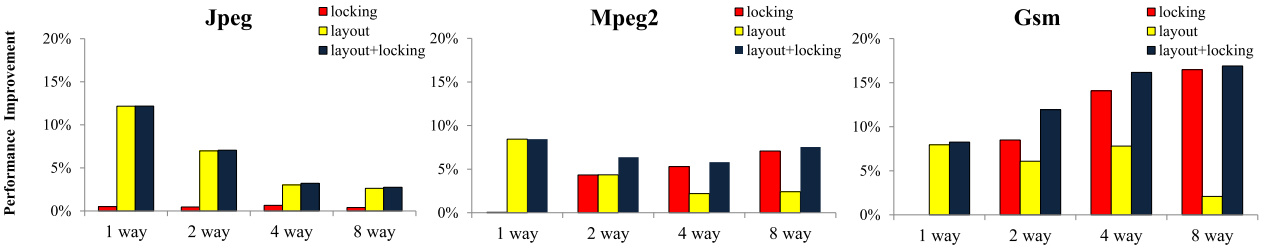


Fig. 9. Performance improvement comparison of procedure placement (TBP) and our cache locking. Cache size is 8 K.

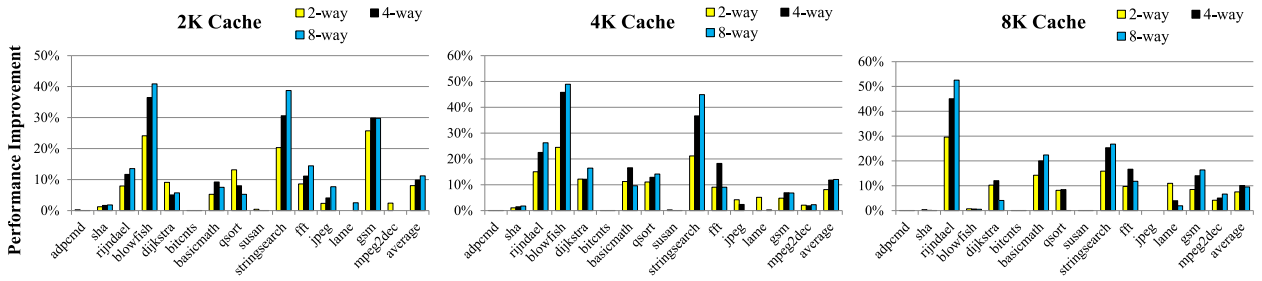


Fig. 10. Performance improvement with FIFO replacement policy for various cache configurations.

each benchmark. Then, we apply cache locking based on the new layout. Fig. 9 shows the performance improvement using this combined strategy (layout + locking). As shown, layout combined with locking is an effective technique to improve cache performance.

5) Impact of Replacement Policy: Our TRP based cache modeling is developed under the assumption that replacement policy is LRU. Berg and Hagersten [38] observed that different replacement policies may have little effect on the miss ratio for most of the applications, but small differences exist. We evaluate our techniques for other replacement policies as well. We try replacement policies FIFO. We first obtain the performance without cache locking using FIFO policy. Then, we use our techniques which employ LRU based TRP cache modeling to select the memory blocks to lock for each cache set. Finally, for the modified program (with locking), we obtain the performance using FIFO replacement policy. The performance improvement with locking (heuristic) over a cache without locking for FIFO replacement policy is shown in Fig. 10 for different cache sizes (2, 4, and 8 KB). For each cache size, we vary the associativity from 1 to 8. We observe that our cache locking technique is still quite effective for FIFO replacement policy. Overall, we obtain 9.7% improvement on average for 2 KB cache, 10.7% improvement on average for 4 KB cache, and 9.0% improvement on average for 8 KB cache.

VII. Conclusion

Cache locking was primarily used for improving timing predictability for hard real-time embedded systems. In this paper, we argue and demonstrate that cache locking is a quite effective technique to improve the average-case execution time of general embedded applications. We first propose TRP to model the cost and benefit of cache locking precisely and efficiently. Then, we propose two locking algorithms–a branch-and-bound algorithm and a heuristic algorithm for line and way locking mechanisms, respectively. Our locking algorithms partially lock the cache with beneficial memory blocks and leave the remaining space for other memory blocks to exploit their data localities. Experiment results indicate that our heuristic locking algorithm can improve the performance by 12% on average for 4 KB cache. In addition, compared to the state-of-the-art approach, our heuristic is better both in terms of performance and efficiency.

References

[1] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimizations," in *Proc. 27th Annu. Int. Symp. Comput. Archit.*, Vancouver, BC, Canada, 2000, pp. 83–94.

[2] J. Montanaro *et al.*, "A 160-Mhz, 32-b, 0.5-W CMOS RISC microprocessor," *Digit. Tech. J.*, vol. 9, no. 1, pp. 49–62, 1997.

[3] K. Anand and R. Barua, "Instruction cache locking inside a binary rewriter," in *Proc. Int. Conf. Compil. Archit. Syn. Embedded Syst.*, Grenoble, France, 2009, pp. 185–194.

[4] Y. Liang and T. Mitra, "Improved procedure placement for set associative caches," in *Proc. Int. Conf. Compil. Archit. Syn. Embedded Syst.*, Grenoble, France, 2010, pp. 147–156.

[5] Y.-T. S. Li, S. Malik, and A. Wolfe, "Cache modeling for real-time software: Beyond direct mapped instruction caches," in *Proc. 17th IEEE Real-Time Syst. Symp.*, Los Alamitos, CA, USA, 1996, pp. 254–263.

[6] H. Theiling, C. Ferdinand, and R. Wilhelm, "Fast and precise WCET prediction by separated cache and path analyses," *Real-Time Syst.*, vol. 18, nos. 2–3, pp. 157–179, 2000.

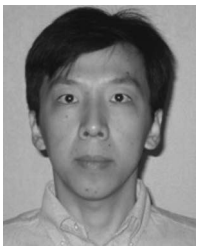
[7] X. Li, Y. Liang, T. Mitra, and A. Roychoudury, "Chronos: A timing analyzer for embedded software," *Sci. Comput. Program.*, vol. 69, nos. 1–3, pp. 56–67, 2007.

[8] H. Falk, S. Plazar, and H. Theiling, "Compile-time decided instruction cache locking using worst-case execution paths," in *Proc. 5th IEEE/ACM Int. Conf. Hardw./Softw. Codesign Syst. Syn.*, Salzburg, Austria, 2007, pp. 143–148.

[9] T. Liu, M. Li, and C. Xue, "Minimizing WCET for real-time embedded systems via static instruction cache locking," in *Proc. IEEE Real-Time Embedded Technol. Appl. Symp.*, San Francisco, CA, USA, 2009, pp. 35–44.

[10] S. Plazar, J. C. Kleinsorge, P. Marwedel, and H. Falk, "WCET-aware static locking of instruction caches," in *Proc. 10th Int. Symp. Code Gener. Optim.*, San Jose, CA, USA, 2012, pp. 44–52.

[11] Y. Liang *et al.*, "Timing analysis of concurrent programs running on shared cache multi-cores," *Real-Time Syst.*, vol. 48, no. 6, pp. 638–680, 2012.
[12] A. Arnaud and I. Puaut, "Dynamic instruction cache locking in hard real-time systems," in *Proc. 14th Int. Conf. Real-Time Netw. Syst.*, Poitiers, France, 2006, pp. 1–10.
[13] H. Ding, Y. Liang, and T. Mitra, "WCET-centric partial instruction cache locking," in *Proc. 49th Annu. Design Autom. Conf.*, San Francisco, CA, USA, 2012, pp. 412–420.
[14] H. Ding, Y. Liang, and T. Mitra, "WCET-centric dynamic instruction cache locking," in *Proc. Conf. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Dresden, Germany, 2014, pp. 1–6.
[15] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Proc. 23th IEEE Real-Time Syst. Symp.*, Austin, TX, USA, 2002, pp. 114–123.
[16] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Viñals, "Improving the WCET computation in the presence of a lockable instruction cache in multitasking real-time systems," *J. Syst. Archit.*, vol. 57, no. 7, pp. 695–706, 2011.
[17] T. Liu, M. Li, and C. Xue, "Instruction cache locking for multi-task real-time embedded systems," *Real-Time Syst.*, vol. 48, no. 2, pp. 166–197, 2012.
[18] H. Ding, Y. Liang, and T. Mitra, "Integrated instruction cache analysis and locking in multitasking real-time systems," in *Proc. 50th Annu. Design Autom. Conf.*, Austin, TX, USA, 2013, pp. 1–10.
[19] X. Vera, B. Lisper, and J. Xue, "Data cache locking for higher program predictability," in *Proc. ACM SIGMETRICS Int. Conf. Meas. Model. Comput. Syst.*, San Diego, CA, USA, 2003, pp. 272–282.
[20] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, "Timing predictability of cache replacement policies," *Real-Time Syst.*, vol. 37, no. 2, pp. 99–122, Nov. 2007.
[21] H. Yang *et al.*, "Improving power efficiency with compiler-assisted cache replacement," *J. Embedded Comput.*, vol. 1, no. 4, pp. 487–499, 2005.
[22] R. A. Uhlig and T. N. Mudge, "Trace-driven memory simulation: A survey," *ACM Comput. Surv.*, vol. 29, no. 2, pp. 128–170, 1997.
[23] T. Liu, M. Li, and C. J. Xue, "Instruction cache locking for embedded systems using probability profile," *J. Signal Process. Syst.*, vol. 69, pp. 173–188, Nov. 2012.
[24] K. Beyls and E. H. D'Hollander, "Reuse distance as a metric for cache behavior," in *Proc. IASTED Int. Conf. Parallel Distrib. Comput. Syst.*, Richardson, TX, USA, 2001, pp. 617–662.
[25] C. Ding and Y. Zhong, "Predicting whole-program locality through reuse distance analysis," *ACM SIGPLAN Not.*, vol. 38, no. 5, pp. 245–257, 2003.
[26] A. Gordon-Ross, F. Vahid, and N. Dutt, "Automatic tuning of two-level caches to embedded applications," in *Proc. Conf. Design Autom. Test Europe*, vol. 1. Paris, France, 2004, pp. 208–213.
[27] I. Nawinne, J. Schneider, H. Javaid, and S. Parameswaran, "Hardware-based fast exploration of cache hierarchies in application specific MPSoCs," in *Proc. Conf. Design Autom. Test Europe*, Dresden, Germany, 2014, pp. 283:1–283:6.
[28] Y. Liang and T. Mitra, "Static analysis for fast and accurate design space exploration of caches," in *Proc. 6th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Syn. (CODES+ISSS)*, Atlanta, GA, USA, 2008, pp. 103–108.
[29] C. Zhang, F. Vahid, and W. Najjar, "A highly configurable cache architecture for embedded systems," *SIGARCH Comput. Archit. News*, vol. 31, no. 2, pp. 136–146, 2003.
[30] H. Cook *et al.*, "A hardware evaluation of cache partitioning to improve utilization and energy-efficiency while preserving responsiveness," in *Proc. 40th Annu. Int. Symp. Comput. Archit. (ISCA)*, Tel Aviv, Israel, 2013, pp. 308–319.
[31] V. Suhendra and T. Mitra, "Exploring locking & partitioning for predictable shared caches on multi-cores," in *Proc. 45th Annu. Design Autom. Conf. (DAC)*, Anaheim, CA, USA, 2008, pp. 300–303.
[32] L. D. Bathen, N. D. Dutt, D. Shin, and S. Lim, "SPMVisor: Dynamic scratchpad memory virtualization for secure, low power, and high performance distributed on-chip memories," in *Proc. 7th IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Syn.*, Taipei, Taiwan, 2011, pp. 79–88.
[33] M. Verma, L. Wehmeyer, and P. Marwedel, "Dynamic overlay of scratchpad memory for energy minimization," in *Proc. 2nd IEEE/ACM/IFIP Int. Conf. Hardw./Softw. Codesign Syst. Syn.*, Stockholm, Sweden, 2004, pp. 104–109.
[34] B. Buck and J. K. Hollingsworth, "An API for runtime code patching," *Int. J. High Perform. Comput. Appl.*, vol. 14, no. 4, pp. 317–329, 2000.
[35] T. Austin, E. Larson, and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling," *IEEE Comput.*, vol. 35, no. 2, pp. 59–67, Feb. 2002.
[36] J. E. W. Steven and P. J. Norman, "CACTI: An enhanced cache access and cycle time model," *IEEE J. Solid-State Circuits*, vol. 31, no. 5, pp. 677–688, May 1996.
[37] G. Fursin, J. Cavazos, and O. Temam, "MiDataSets: Creating the conditions for a more realistic evaluation of iterative optimization," in *Proc. Int. Conf. High Perform. Embedded Archit. Compil. (HiPEAC)*, Ghent, Belgium, 2007, pp. 245–260.
[38] E. Berg and E. Hagersten, "StatCache: A probabilistic approach to efficient and accurate data locality analysis," in *Proc. IEEE Int. Symp. Perform. Anal. Syst. Softw. (ISPASS)*, Austin, TX, USA, 2004, pp. 20–27.



Yun Liang received the B.S. degree in software engineering from Tongji University, Shanghai, China, and the Ph.D. degree in computer science from the National University of Singapore, Singapore, in 2004 and 2010, respectively.

He was a Research Scientist with Advanced Digital Science Center, University of Illinois Urbana-Champaign, Urbana, IL, USA, from 2010 to 2012. He has been an Assistant Professor with the School of Electronics Engineering and Computer Science, Peking University, Beijing, China, since 2012. His current research interests include graphics processing unit architecture and optimization, heterogeneous computing, embedded system, and high level synthesis.

Dr. Liang was a recipient of the Best Paper Award in International Symposium on Field-Programmable Custom Computing Machines (FCCM)'11 and the best paper award nominations in International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)'08 and Design Automation Conference (DAC)'12. He serves a Technical Committee Member for Asia South Pacific Design Automation Conference (ASPDAC), Design Automation and Test in Europe (DATE), and International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES). He is the TPC Subcommittee Chair for ASPDAC'13.

Tulika Mitra received the Ph.D. degree in computer science from the State University of New York at Stony Brook, Stony Brook, NY, USA, in 2000.

She is a Professor of Computer Science with the School of Computing, National University of Singapore, Singapore. Her current research interests include design automation of embedded real-time systems with particular emphasis on application-specific processors, software timing analysis/optimizations, heterogeneous multicores, and energy-aware computing.

Lei Ju received the B.E. and Ph.D. degrees from the School of Computing, National University of Singapore, Singapore, in 2005 and 2010, respectively.

He has been an Associate Professor with the School of Computer Science and Technology, Shandong University, Jinan, China, since 2011. His current research interests include design, analysis, and optimization of real-time systems and embedded systems. He has authored a number of referred publications.

Prof. Ju was a recipient of the best paper award nominations in International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS)'08 and RTAS'11. He is currently a member of the CCF Computer Architecture Technical Committee, and serves as the Technical Program Committee Member of several international conferences.