Instruction Cache Locking Using Temporal Reuse Profile

Yun Liang, Tulika Mitra
School of Computing, National University of Singapore
{liangyun,tulika}@comp.nus.edu.sg

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, the compiler optimization techniques focus on employing cache locking to improve 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 average-case program performance. We introduce temporal reuse profile to accurately and efficiently model the cost and benefit of locking memory blocks in the cache. We propose an optimal algorithm and a heuristic approach that use the temporal reuse profile to determine the most beneficial memory blocks to be locked in the cache. Experimental results show that locking heuristic achieves close to optimal results and can improve the cache miss rate by up to 24% 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.

Categories and Subject Descriptors

B.3.2 [**Memory Structures**]: Design Styles—*Cache memories*

General Terms

Algorithms, Performance, Design

Keywords

Cache, Cache Locking, Temporal Reuse Profile

1. INTRODUCTION

Cache memory help bridge the performance gap between the fast processor and the slow main memory. In particular, instruction cache plays a critical role in embedded systems in terms of both performance and energy consumption as instructions are fetched almost every clock cycle. Thus instruction cache optimizations are of paramount importance in meeting both the performance and the energy constraints of embedded system.

Most modern embedded processor (e.g., ARM Cortex series processors) feature 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 replacement policy. Thus, all the subsequent accesses to the locked memory blocks will be cache hit. 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 embedded 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.

However, cache locking has the potential to significantly improve the average-case performance of a general embedded application. This can be achieved by systematically eliminating conflict misses in the cache through 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 $(m_0 m_1)^{10}$. Given a direct mapped cache, all the accesses will be cache miss (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 block in a cache set can negatively impact the performance of the remaining memory blocks mapped to the same set as the effective cache capacity gets reduced. Therefore, any cache locking algorithm should carefully balance the cost and benefit of locking.

In this paper, we explore instruction cache locking to improve the average-case execution time. Recently, Anand and Barua [5] have presented an instruction cache locking heuristic with the same objective. Their experiments confirm that locking is beneficial in improving 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. Thus the algorithm is quite inefficient specially for large benchmarks. Moreover, they employ some approximations in the cost/benefit analysis to cut down simulation cost leading to inaccuracy.

We introduce temporal reuse profile (TRP) to accurately and compactly capture the cost/benefit of locking each memory block. TRP is significantly more compact compared to memory traces. We propose two cache locking algorithms based on TRP: an op-

DAC 2010, June 13-18, 2010, Anaheim, California, USA.

timal algorithm based on branch-and-bound search and a heuristic approach. We show that our cache locking heuristic improves the state of the art [5] in terms of both performance and efficiency and achieves close to the optimal result.

We also compare cache locking with a complementary technique called procedure placement. 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.

2. RELATED WORK

Instruction cache locking has been primarily employed in hard real time systems for better timing predictability [12, 14, 10]. In hard real time systems, worst case execution time (WCET) is an essential input to the schedulability analysis of mutli-tasking real time systems. It is difficult to estimate a safe but tight WCET in the presence of complex micro-architectural features such as caches. By statically locking instructions in the cache, WCET becomes more predictable. Locking has also been applied to shared caches in multi-cores in [13].

In this paper, our aim is to improve average-case execution time for general embedded systems through locking. Data cache locking mechanism based on the length of the reference window for each data-access instruction is proposed in [15]. However, they do not model the cost/benefit of locking and there is no guarantee of performance improvement. Recently, Anand and Barua proposed an instruction cache locking algorithm for improving average-case execution time in [5]. This work is most related to ours. However, our approach differs in two important aspects. First, Anand and Barua's approach relies on simulation, while we use an accurate cache modeling. More concretely, in Anand and Barua's method, two detailed trace simulations are employed in each iteration where an iteration locks one memory block in the cache. In contrast, we require only one profiling step. Secondly, in our approach, the cost/benefit of cache locking is modeled precisely using temporal reuse profiles of memory blocks. However, in their method, cache locking benefit is approximated by locking dummy blocks to keep the number of simulations reasonable. Thus our work improves over [5] both in terms of performance and efficiency.

In this work, we introduce temporal reuse profile to model cache behavior. Previously, reuse distance has been proposed for the same purpose [7, 9]. Reuse distance is defined as the number of distinct data accesses between two consecutive references to the same address and it accurately models 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. TRP records both the reuse content and their frequencies.

3. CACHE LOCKING PROBLEM

In this section, we formally define the cache locking problem. We only consider static instruction cache locking in this work where 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* [8] approach. The extra code to fetch and lock the memory blocks in cache are inserted at the end of the program as a trampoline. We leave some dummy NOP instructions at the entry point of the program that get replaced by a call to this trampoline after locking decision are made. As we are considering static cache locking, the cost of executing the trampoline is negligible and we will ignore this overhead in the rest of the discussion.

Cache Terminology. A cache memory is defined in terms of four major parameters: *block or line size* L, *number of sets* K, *associativity* A, and *replacement policy*. The block or line size determines the unit of transfer between the main memory and the 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 a set-associative or fully-associative cache, the replacement policy (e.g., LRU, FIFO, etc.) defines the block to be evicted when a cache set is full. In this work, we consider Least Recently Used (LRU) replacement policy where the block replaced is the one that has been unused for the longest time.

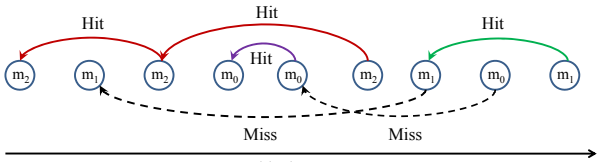
Two locking mechanisms are commonly used in modern embedded processors — way locking and line locking. In way locking, particular ways of a set associative cache are selected for locking and these ways are locked for all the cache sets. Way-locking is employed by ARM processor series [3, 4]. Compared to way locking, line locking is a fine grained locking mechanism. In line locking, different number of lines can be locked for different cache sets. Line locking is employed by Intel's Xcale [1], ARM9 family and Blackfin 5xx family processors [2]. We assume the presence of line locking mechanism in this work.

Modeling Cache Locking. 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). As discussed before, we ignore the cost of execution of the trampoline, which is negligible. 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 their hit rates. So cache locking primarily targets to eliminate conflict misses while minimizing the negative impact on the unlocked memory blocks.

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 i^{th} cache set C_i. Also given a memory block m, it is only mapped to set $(m \ \ modulo \ \ K)$. Thus, for any two cache sets C_i, C_j, we have $M_i \cap M_j = \phi$. That is, there is no interference between the cache sets and they can be modeled independently to arrive at locking decisions. 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 j^{th} 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

$$num\_hit(m) = access(m) \text{ if } m \text{ is locked}$$



Memory blocks	Access	Temporal Reuse Profiles
m_0	3	$\{<\{\phi\}, f(\{\phi\})=1>\}$
m_1	3	$\{<\{m_0\}, f(\{m_0\})=1>\}$
m_2	3	$\{<\{m_0\}, f(\{m_0\})=1>, <\{m_1\}, f(\{m_1\})=1>\}$

Figure 1: Temporal reuse profiles 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 set. Cache hits and misses are highlighted.

However, locking memory block $m \in M_i$ in cache set C_i will have negative impact on the memory blocks $M_i \backslash \{m\}$. We will now proceed to characterize this impact accurately.

THEOREM 3.1. *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 set C_i reduces by one. Clearly, if $m[j]$ (j^{th} reference of m in the trace) was a miss (i.e., not present in the cache set) originally with more cache blocks, it will be a miss with one fewer 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 temporal conflict $TCS_{m[j]}$ 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 $TCS_{m[j]} = \varnothing$.*

For example, in Figure 1, the temporal conflict set of memory block m_2 is $\{m_1\}$ for its second reference and $\{m_0\}$ for its third reference. The temporal conflict set determines whether the memory block reference will be a cache hit or a cache miss.

THEOREM 3.2. *If $|TCS_{m[j]}| \geq A$ 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 miss in its next reference. □

Moreover, following Theorem 3.1, if $|TCS_{m[j]}| \geq A$, then the $m[j]$ will be cache miss irrespective of locking other memory blocks in the cache. Therefore, we can eliminate $TCS_{m[j]}$ from further consideration as far as cache locking decisions are concerned. For example, in Figure 1, the second reference to memory block m_1 is cache miss and its temporal conflict set can be removed.

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

THEOREM 3.3. *If $|TCS_{m[j]}| < A$ for $m \in M_i \backslash Lock_i$, then $m[j]$ will be cache miss only when $|Lock_i \cup TCS_{m[j]}| \geq A$.*

PROOF. As $|TCS_{m[j]}| < A$, the reference $m[j]$ will be cache hit in the original cache. Now as we lock memory blocks into the cache set C_i, the space available to accommodate the unlocked cache blocks will reduce. $m[j]$ will be cache miss when the number of conflicting blocks and the locked blocks together exceeds the associativity of the cache. That is, $m[j]$ will be cache miss when $|Lock_i \cup TCS_{m[j]}| \geq A$. □

For example, in Figure 1, the second reference of memory block m_2 will be 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 $\mathcal{R}_m = \{TCS_{m[j]} : j > 1, |TCS_{m[j]}| < A\}$, i.e., $\mathcal{R}_m$ is the set of TCS for reference of m that result in hits in the original cache.

DEFINITION 2 (**Temporal Reuse Profile**). *The temporal reuse profile 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 temporal conflict set s in the trace.*

Figure 1 shows an example of temporal reuse profiles given a trace of memory block access. There are three memory blocks in the trace and the number of access for each of them is collected. More importantly, for each memory block, only the TCS which results in cache hits is kept in the profiles. The TCS of second reference of memory block m_1 is not kept, because it is a cache miss in a 2-way set associative cache (both m_0 and m_2 are in between).

Given the temporal reuse profile for a program execution and the locked memory blocks per cache set $Lock_i : i = 1 \ldots K$, we can now accurately compute the number of cache hit/miss for the entire trace. For a memory block $m \in M_i$

$$\{num\_hit(m)|Lock_i\} = \begin{cases} \sum_{\forall \langle s, f(s)\rangle \in TRP_m \atop |s \cup Lock_i| < A} f(s) & \text{if } m \notin Lock_i \\ access(m) & \text{otherwise} \end{cases}$$

The total number of cache hits for a cache set C_i is

$$\{num\_hit(\mathcal{T}_i)|Lock_i\} = \sum_{m \in M_i} \{num\_hit(m)|Lock_i\}$$

and the total number of cache hits for the entire program

$$num\_hit(\mathcal{T}) = \sum_{i=1}^{K} \{num\_hit(\mathcal{T}_i)|Lock_i\}$$

Problem Statement. The goal of static instruction cache locking is to determine the set of memory blocks to be locked per cache set $Lock_i : i = 1 \ldots K$ such that $num\_hit(\mathcal{T})$ is maximized.

4. CACHE LOCKING ALGORITHMS

Our cache locking algorithm consists of two phases: *profiling* and *locking*.

Profiling Phase. The profiling phase creates the temporal reuse profile (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 instruction address trace. The temporal reuse profile can be generated by a single pass through the instruction address trace.

Locking Phase. The locking phase determines the set of memory blocks to be locked in each cache set such that the number of cache hits is maximized. We propose two algorithms to select the memory blocks to be locked. One is an optimal solution based on branch and bound search and the other one is an iterative heuristic.

4.1 Optimal Algorithm

Our optimal cache locking algorithm is presented in Algorithm 1. Each cache set is analyzed individually. For each memory block m, we have to decide whether to lock it or not. Thus, the entire search space can be seen as a binary decision tree. Algorithm 1 covers the entire search space and is guaranteed to find the optimal solution. However, in the worst case, its complexity is as high as that of exhaustive search.

Algorithm 1: Optimal cache locking algorithm

```
1  foreach set C_i in the cache do
2  |   optimalSoln := ∅; Maxhit := {num_hit(T_i)|∅} ;
3  |   search(M_i, ∅);
4  Function (search(M, Lock))
5  if |Lock| = A or M = ∅ then
6  |   Newhit := {num_hit(T_i)|Lock};
7  |   if Newhit > Maxhit then
8  |   |   optimalSoln := Lock;
9  |   |   Maxhit := Newhit;
10 |   return;
11 Let m be any memory block from M;
12 Lock := Lock ∪ m; M := M\m; /* impact of locking m */
   Curhit := Σ_{m'∈M_i\M} {num_hit(m')|Lock};
13 Let M' be set of A − |Lock| memory blocks from M with maximum access ;
14 Bound := Σ_{m'∈M'} access(m');
15 M'' := ∅;
16 while |M'| + |M''| < |M| do
17 |   find m' s.t {num_hit(m')|Lock} =
   |   max_{m∈M\M''} {num_hit(m)|Lock};
18 |   Bound := Bound + {num_hit(m')|Lock};
19 |   M'' := M'' ∪ m';
20 Bound := min(Bound, Σ_{m'∈M} access(m'));
21 if Curhit + Bound > Maxhit then
22 |   search(M, Lock); /* branching decision for m */
23 search(M, Lock\m);
```

Each level in the binary search tree corresponds to locking decision for one memory block in the set. We obtain a solution when a leaf node is reached or the entire cache set is locked (line 5). The number of cache hits is computed based on the cache modeling described in Section 3 and only the best solution (the lock list) is kept (line 6-9). At each level i, we use $Curhit$ to represent the cache hits from level 1 to i given the current lock list $Lock$. $Curhit$ is an upper bound of the number of cache hits from level 1 to i as some of the hits might become miss as more blocks are locked while exploring the lower levels. We define $Bound$ as the maximum possible number of cache hits from the remaining levels. This bound is estimated by adding the accesses of $A - |Lock|$ memory blocks with the maximum number of accesses (corresponds to locking the remaining cache lines with most profitable memory blocks) and the hits of $|M| - (A - |Lock|)$ memory blocks with the maximum number of hits given the current lock list (corresponds to not locking remaining cache lines). If $Curhit + Bound \leq Maxhit$, then the search space rooted at current node will be pruned.

4.2 Heuristic Approach

Our heuristic is iterative in nature and exploits the modeling of cache locking described in Section 3. 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 $num\_hit(\mathcal{T}_i)$ is maximized.

Initially, we set $Lock_i = \emptyset$ and compute the number of cache hits in the original cache

$$current\_hit = \{num\_hit(\mathcal{T}_i)|\emptyset\}$$

In each iteration, we go through all the unlocked memory blocks in the cache set $m \in M_i \backslash Lock_i$ and compute the number of cache hits if m was locked in the cache.

$$new\_hit_m = \{num\_hit(\mathcal{T}_i)|Lock_i \cup \{m\}\}$$

Let

$$benefit = \max_{m \in M_i \backslash Lock_i} (new\_hit_m - current\_hit)$$

If $benefit \leq 0$, then locking any of the cache blocks would worsen the memory performance and we should terminate our iterative algorithm. Otherwise, we choose the memory block m with the maximum benefit, i.e., $benefit = new\_hit_m - current\_hit$. We break ties arbitrarily. The algorithm also terminates when $|Lock_i| = A$, i.e., we have locked all the blocks in the cache set. Our cache locking algorithm is detailed in Algorithm 2.

Algorithm 2: Heuristic cache locking algorithm

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

Benchmark	Trace (MB)	TRP (KB)	Runtime (sec)	
			Our-Heuristic	Anand-Barua
Adpcm	220	4.43	11.68	1649
Sha	103	6.43	5.41	803
Rijndael	147	10.58	8.25	1936
Blowfish	318	9.68	17.44	2666
Dijkstra	293	10.01	15.82	2304
Bitcnts	170	3.08	9.45	1146
Basicmath	1400	28.92	80.75	13590
Qsort	360	11.78	20.52	2722
Susan	220	9.98	11.80	1645
Stringsearch	39	9.76	2.16	327
FFT	790	25.58	48.63	7436
Jpeg	235	50.58	14.89	1583
Lame	965	124.08	80.14	7255
Gsm	297	21.00	16.99	3293
Mpeg2dec	222	39.42	12.88	1883

Table 1: Characteristics of Benchmarks.

5. EXPERIMENTAL EVALUATION

Experimental Setup. We select benchmarks from MiBench and MediaBench for evaluation purposes. The benchmarks and their characteristics are shown in Table 1. We conduct our experiments using SimpleScalar framework [6]. 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 temporal reuse profile (TRP). Both the trace size and TRP size are shown in Table 1. Note that the TRP size depends on the cache configuration. The table shows the average TRP size across all evaluated cache configurations. The temporal reuse profile is significantly more compact compared to the address trace (KB vs. MB).

We evaluate the effectiveness of our algorithms with different cache parameters. We vary the cache size (2KB, 4KB, 8KB) and 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. Cache hit latency is assumed to be 1 cycle and cache miss penalty is assumed to be 100 cycles. We perform all the experiments on a 3GHz Pentium 4 CPU with 2GB memory. We propose two algorithms for cache locking: an optimal solution

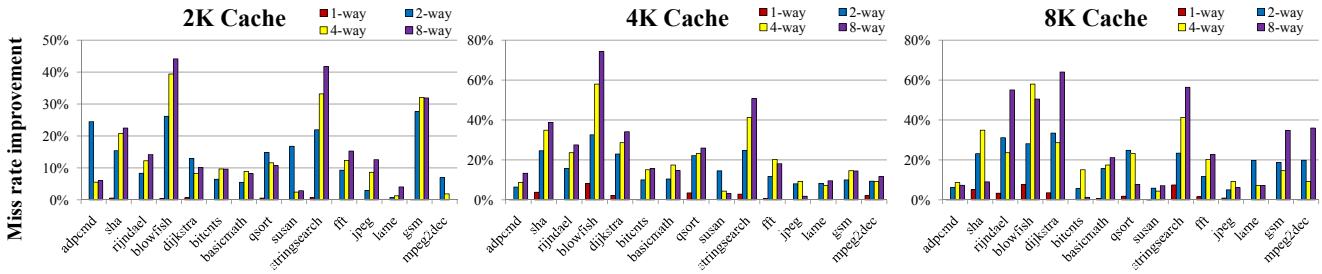


Figure 2: Miss rate improvement (percentage) over cache without locking for various cache configurations.

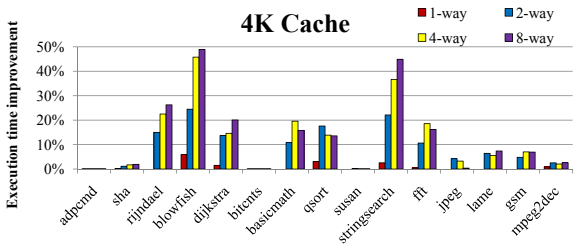


Figure 3: Execution time improvement (percentage) over cache without locking for 4K cache configurations.

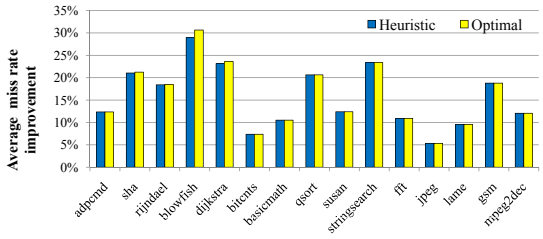


Figure 4: Cache miss rate improvement comparison of heuristic and optimal algorithm for 2-way set associative cache.

and a heuristic approach. We present the results for the heuristic first, followed by a comparison of the two algorithms.

Miss Rate Improvement. The instruction cache miss rate improvement with locking (heuristic) over a cache without locking is shown in Figure 2 for different cache size (2KB, 4KB and 8KB). For each cache size, we vary the associativity from 1 to 8. For any cache size, the miss rate improvement for direct mapped cache is minimal. This is expected as only one block is available per cache set and locking that block implies miss for all the remaining memory blocks mapped to the cache set. However, for set associative caches, our locking algorithm achieves significant performance improvement across all the benchmarks. We obtain 14% improvement on an average for 2KB cache, 20% improvement on an average for 4KB cache, and 24% improvement on an average for 8KB cache.

Execution Time Improvement. Figure 3 shows the execution time improvement due to locking for 4KB cache. Similar trend can be seen for other cache sizes. Some benchmarks do not gain considerable execution time improvement even though cache miss rate is improved. This is because for these benchmarks the absolute cache miss number without cache locking is very small. Thus, improvement in cache miss rate will not contribute much to the overall execution time reduction. We obtain 10% improvement on an average for 2KB cache, 12% improvement on an average for 4KB cache, and 10% improvement on an average for 8KB cache.

Heuristic versus Optimal. The cache miss improvement comparison of heuristic and optimal solution for 2-way set associative caches is shown in Figure 4. The cache miss improvement is an average value across all the cache sizes. As shown, the heuristic returns close to optimal solutions. For 2-way set associative caches, our heuristic improves instruction cache miss rate by 15.6% on an average, while the optimal solution improves it by 15.8%. For direct mapped caches, both heuristic and optimal solution achieve marginal improvement. For 4-way associative cache, the heuristic achieves 21.1% miss rate improvement on an average, while the optimal solution returns 21.4% improvement. As for runtime of the algorithms, the heuristic is $1-273$ times faster than optimal for low associativity caches (≤ 4). For 8-way associative cache, the heuristic returns solutions quite fast (Table 1), but optimal algorithm fails to terminate within 10 hours for some big benchmarks.

Comparison with Anand-Barua Method. We compare our heuristic with Anand-Barua method [5] — the only other approach in the literature targeting cache locking for average-case program performance improvement. Their proposal is an iterative simulation-based heuristics and needs feedback (cache performance) from trace driven simulator in each iteration. We implement their algorithm and compare against our heuristic both in terms of performance (cache miss rate improvement) and efficiency (algorithm runtime).

In terms of cache miss rate improvement, our approach generally performs better or at least equal compared to Anand-Barua's method for every cache configuration. Figure 5 shows the average instruction cache miss rate improvement for 2KB, 4KB and 8KB cache sizes. The miss rate improvement shown in Figure 5 is the average across all set-associative caches (2, 4 and 8 way) as both the methods do not gain much for direct mapped caches. As evident from Figure 5, our heuristic achieves higher cache miss rate improvement than Anand-Barua's method across all the benchmarks and cache configurations. For benchmarks *blowfish* and *stringsearch*, the improvement over Anand-Barua's method are more than 20% for some configurations. This is because our cache modeling is accurate whereas it is approximated in Anand-Barua's work.

Anand-Barua method invokes cache simulation in each iteration. However, cache simulation can be very slow for large traces. In addition, the number of simulations required grows linearly with the total number of locked memory blocks. When, the number of memory blocks to be locked is not that small, simulation based approach may not be feasible. In contrast, we only need one round of profiling and the subsequent analysis relies only on those compact profiles. The runtime comparison of our heuristic and Anana-Barua's method is detailed in Table 1 under column *Runtime*. The time presented is the average runtime across 12 cache configurations. Our approach is $91-234$ times faster compared to Anand-Barua's method.

Code Memory 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 significantly throughout procedure placement — reordering procedures so that cache conflicts are reduced [11]. Clearly, procedure placement and cache locking are complementary approaches.

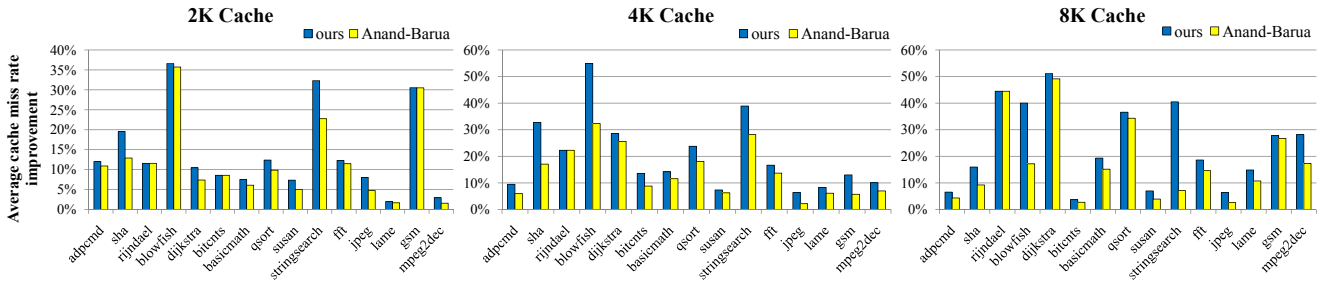


Figure 5: Average cache miss rate improvement comparison.

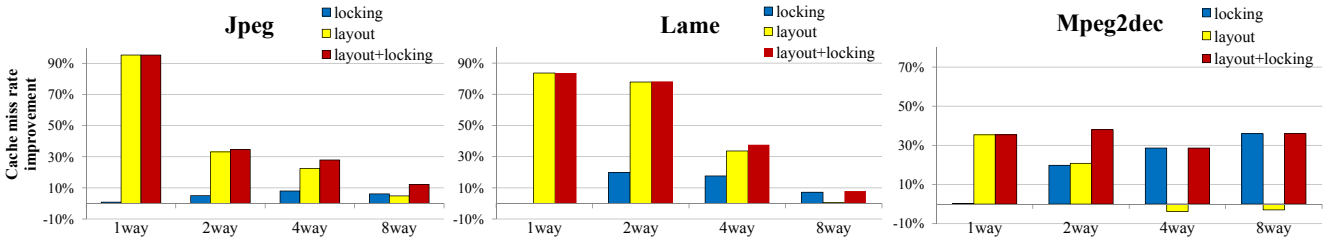


Figure 6: Procedure placement (TPCM) vs Cache locking. Cache size is 8K.

In the following, we evaluate the effects of these two techniques on cache performance. For procedure placement, we choose TPCM [11] — a state of the art procedure placement technique. TPCM does not guarantee improvement of instruction cache performance after procedure reordering. In contrast, our cache locking algorithm, by design, is guaranteed to either improve the performance or keep it the same. Moreover, procedure placement techniques are effective for applications with substantial number of procedures such as *Jpeg*, *Lame*, and *Mpeg2dec*. Figure 6 shows the comparison of TPCM with cache locking in terms of cache miss rate improvement.

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 low associativity caches (1 or 2), while locking is more suitable for higher associativity caches (2, 4 and 8). Procedure placement may not be good choice for higher associativity caches due to the following reasons. First, higher associativity leads to fewer cache sets leaving little opportunity for procedure reordering. In contrast, higher associativity provides more opportunity for cache locking. Moreover, procedure placement may incur performance loss (see *Mpeg2dec*) due to coarse grained performance modeling, while our cache locking heuristic is guaranteed not to degrade the performance.

Clearly, procedure placement and locking are complementary approaches and the cache performance can benefit significantly through a combination of these two approaches. To validate our hypothesis, we first performed procedure placement for each benchmark. If the newly generated layout degrades the performance, we eliminate the layout and revert back to the original layout for cache locking. Otherwise, we perform cache locking based on the new layout. Figure 6 shows the miss rate improvement using this combined strategy. As evident from the figure, layout combined with locking can achieve significant improvement for some benchmarks.

6. CONCLUSION

In this paper, we propose two cache locking algorithms — an optimal algorithm and a heuristic approach — to improve the average-case instruction cache performance. We introduce temporal reuse profiles (TRP) to model the cost and benefit of cache locking precisely and efficiently and exploit TRP in both the algorithms. Experiment results indicate that our heuristic can improve cache miss rate by as much as 24% and achieves close to the optimal results. In addition, compared to the state of the art approach, our heuristic is better both in terms of performance and efficiency.

7. ACKNOWLEDGMENTS

This work was partially supported by research grants R-252-000-387-112 & MOE academic research fund Tier 2 MOE2009-T2-1-033.

8. REFERENCES

[1] 3rd Generation Intel Xscale Microarchitecture Developers's Manual. Intel, May 2007. `http://www.intel.com/design/intelxscale`.

[2] ADSP-BF533 Processor Hardware Reference. Analog Devices, April 2009. `http://www.analog.com/static/imported-files/processor_manuals/bf533_hwr_Rev3.4.pdf`.

[3] ARM Cortex A-8 Technical Reference Manual. ARM, Revised March 2004. `http://www.arm.com/products/CPUs/families/ARMCortexFamily.html`.

[4] ARM1156T2-S Technical Reference Manual. ARM, Revised July 2007. `http://www.arm.com/products/CPUs/families/ARM11Family.html`.

[5] K. Anand and R. Barua. Instruction cache locking inside a binary rewriter. In *CASES*, 2009.

[6] T. Austin et al. Simplescalar: An infrastructure for computer system modeling. *Computer*, 35(2):59–67, 2002.

[7] K. Beyls and E.H. D'Hollander. Reuse distance as a metric for cache behavior. In *Proceedings of the IASTED International Conference on Parallel and Distributed Computing and Systems*, 2001.

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

[9] C. Ding and Y. T. Zhong. Predicting whole-program locality through reuse distance analysis. *SIGPLAN Not.*, 38(5), 2003.

[10] H. Falk et al. Compile-time decided instruction cache locking using worst-case execution paths. In *CODES+ISSS*, 2007.

[11] N. Gloy and M. D. Smith. Procedure placement using temporal-ordering information. *ACM Trans. Program. Lang. Syst.*, 21(5):977–1027, 1999.

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

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

[14] X. Vera et al. Data cache locking for higher program predictability. In *SIGMETRICS*, 2003.

[15] H. Yang et al. Improving power efficiency with compiler-assisted cache replacement. *J. Embedded Comput.*, 1(4):487–499, 2005.