Predictability and Performance Enhancement for Real-Time Embedded Systems by Cache-Locking

Abu Asaduzzaman, Niranjan Limbachiya, and Imad Mahgoub

Florida Atlantic University, Boca Raton, Florida, 33431, USA

Abstract — **In real-time systems, cache memory poses challenge to improve both predictability and performance because of its adaptive and dynamic behavior. Recent studies indicate that for application-specific embedded systems, static cache-locking helps determining the worst case execution time (WCET) and cache-related preemption delay. In this work, we propose a static instruction cache-locking algorithm that makes the real-time embedded system more predictable by locking the blocks that might cause more cache misses. We obtain CPU utilization for both static cache analysis (no cache-locking) and static cache-locking using Heptane. Experimental results show that our cache-locking algorithm may improve both predictability and performance of real-time systems.**

Index Terms — **Predictability, performance, cache-locking, real-time embedded system, Heptane.**

I. Introduction

The demands for running real-time multimedia applications on embedded systems are growing. Processing video applications on embedded systems is a significant challenge for memory subsystem, which are the primary performance bottleneck. Multimedia applications seriously suffer due to the memory inefficiency from dropped frames, blocking, or other annoying artifacts [1], [2]. Memory hierarchy is changing in order to support multimedia applications in embedded systems. Cache memory is introduced to improve performance by reducing the speed-gap between the processor and the main memory. However, cache is an important source of unpredictability due to its adaptive and dynamic characteristics; as a result programs may behave in an unexpected way. A lot of work has been done to predict the worst-case behavior of embedded applications in order to determine the safe and precise bounds on tasks worst-case execution time (WCET) and cache-related preemption delay [3], [4]. Cache-locking mechanism adapts caches to the needs of real-time systems. Recent studies show that the time required to perform a memory access is predictable with static/dynamic data/instruction cache-locking [5], [6], [7]. It is also observed that cache-locking improves predictability by removing both intra-task and inter-task interferences [8], [9], [10], [11]. Locking the cache is a solution that may trade performance for predictability - at a cost of lower performance, the time of accessing the memory becomes predictable. Special attention is needed in real-time systems with caches so that the performance and predictability remain reasonable.

In this work, we present a static instruction cache-locking algorithm that may improve the predictability and performance of real-time embedded systems. In Section II, a few relevant articles from our review are summarized. Section III discusses some cache-locking algorithms. In Section IV, proposed cache-locking algorithm is explained. Experimental setup is described in Section V. In Section VI, the experimental results are discussed. Finally, we conclude our work in Section VII.

II. Related Work

A lot of progress has been made in the past few years to address the predictability and performance issues in real-time embedded systems. A few of them, related to our work, are discussed in this section.

In [3], a new approach is presented that copes with caches in real-time systems that statically locks the contents so as to make memory access times and cache-related preemption delays more predictable. In [4], two low-complexity algorithms are proposed for selecting the contents of statically-locked caches. The performance is measured and compared with that of a state of the art static cache analysis method. Experimental results show that both techniques improve predictability but may reduced performance.

When the cache miss-ratio is higher, the program will lead to longer execution time and higher power consumption. By knowing the cache miss-ratio, performance can be estimated in advance and can be used as input for compilers and system developers. Article [6] presents a static method to limit the worst-case instruction cache miss-ratio of a program. The static method needs no manual annotations in the code and is safe in the meaning that no under-estimation is possible.

A dynamic algorithm is proposed in [8] that partitions the task into a set of regions. Each region owns statically a locked cache content that is determined offline. A set of tasks is used to experimentally analyze the effects of the algorithm on the worst-case cache miss rate (WCCMR). Experiments show significant improvement, when compared with a system without any cache.

Work [9] and [10] discuss data cache-locking related issues. In [9], a new technique is proposed to obtain predictability in preemptive multitasking systems in the presence of data caches. The complexity introduced by the data caches makes it difficult to bound execution times tightly. Cache partitioning, dynamic cache-locking, and static cache analysis are done to provide worst-case performance estimates in a safe and tight way. In this work, two strategies are employed in order to minimize the performance degradation due to cache partitioning and locking. First, the cache is loaded with data likely to be accessed so that their cache utilization is maximized. Second, compiler optimizations such as tiling and padding are applied in order to reduce cache replacement misses. Experimental results indicate that this scheme is very predictable, without compromising the performance of the transformed programs.

In [11], a genetic algorithm is developed and applied in real-time systems to improve predictability. The set of instructions to be locked in cache is selected using the genetic algorithm. This algorithm estimates a tight upper bound of the response time of tasks. Experimental results show that this scheme is highly predictable, and the performance loss is negligible for most tasks.

In our previous work [1], we focus on cache modeling and optimization for portable communication devices running MPEG-4 video decoder. Simulation results show that MPEG-4 decoding performance in embedded systems can be enhanced by optimizing cache parameters. In this work, we investigate the impact of our proposed static instruction cache-locking algorithm on the predictability and performance of real-time embedded systems.

III. Cache-Locking Algorithms

A lot of cache-locking algorithms have been proposed to improve predictability in real-time and hard real-time embedded systems. Some of those algorithms are discussed in the following subsections.

A. Low-Complexity Cache-Locking Algorithm

Two algorithms are proposed in [4] to select the contents of the locked instruction cache. Both algorithms are greedy (they do not reconsider the assignment of a cache block once it has been decided) and have a pseudo polynomial complexity. They use the knowledge of the memory accesses made by the tasks along their worst-case execution paths. The two algorithms are based on different metrics to select the cache contents in order to optimize the task set schedule ability. The first one, Lock-MU (minimizing utilization) aims at minimizing the worst-case CPU utilization, while the second one Lock-MI (minimizing interferences) aims at minimizing the interferences between tasks. It is important to note that the metrics used for the selection of the locked cache do not prescribe the use of any particular schedule ability analysis method, although CUA (Cache-aware Utilization-based Analysis) and CRTA (Cache-aware Response Time Analysis) are used to evaluate the performance of the algorithms, any scheduling policy taking into account the cache-related preemption delay can be used for schedule ability analysis. These algorithms are simple to implement and show predictability and performance improvement when evaluated on a small real task set.

B. Region Merging and Inlining (RMI) Algorithm

This algorithm is used for finding a partition of the machine code of a given task into regions, and to determine a locked state of the instruction cache for each such region [8]. It is performed in a non-blind manner by using memory access patterns obtained by profiling the task. In this algorithm the worst-case performance of the tasks is compared in two situations – the cache is dynamically locked and the cache is dynamic with a LRU policy. The goal of this algorithm is to improve the worst-case performance as compared with a system with no cache.

C. Algorithm for Selective Cache Loading

Compile-time cache analysis is combined with data cache-locking in this algorithm to estimate the worst-case memory performance (WCMP) in a safe, tight and fast way [10]. In order to get predictable cache behavior, the cache for those parts of the code where the static analysis fails is locked first. To minimize the performance degradation, this method loads the cache, if necessary, with data likely to be accessed. According to their experimental results this scheme is very predictable without compromising the performance of the transformed program. If compared to an algorithm that assumes compulsory misses when the state of the cache is unknown, this approach eliminates all overestimation for the set of benchmarks, giving an exact WCMP of the transformed program without any significant decrease in performance.

D. Cache-Defect-Aware Code Placement Algorithm

In this algorithm, a defect-aware code placement technique is introduced which reduces the performance degradation of a processor with a partially good cache memory [12]. This approach is to modify the placement of basic blocks or functions in the address space so that the number of cache misses is minimized for a given defective cache. This is the first known compiler technique which reduces the performance degradation of a partially good cache memory. Three benchmark programs, namely Compress version 4.0, JPEG encoder version 6b, and MPEG2 encoder version 1.2 are used. The Best Case, Worst Case and Average Case results are compared for the benchmark programs. Experiments demonstrate that the technique can compensate the performance degradation from 5% to 25% of the cache lines are faulty.

IV. Cache-Locking Algorithm Used

Our proposed static instruction cache-locking algorithm is based on the static tree-graph generated by Heptane tool [13]. The main objective of this simple-scheme is to lock the blocks that cause more misses. Heptane generates a tree-graph for C source file. The syntax tree is a tree whose nodes represent the structure of programs in the high-level language and whose leaves represent basic blocks. Leaves in the syntax tree coincide with the nodes in the control-flow graph. From tree graph, in static analysis we collect the following information,

- Name of the node
- Number of instructions
- Total number of cycles
- Cache miss (also, cache hit) for each node

From off-line analysis we determine which code section of the source file causes more misses. We divide the analysis in several parts including root node of main C source file, calling function for C source file, all leaf node analysis for root node and top loop node level analysis. When we perform cache analysis using Heptane WCET analyzer, Heptane generates tree graph of the C program used. We collect instruction block (IB) address cache miss information based on tree graph and we generate instruction cache-locking XML file. In order to implement the static instruction cache-locking scheme, a small routine is required to be executed at the system start-up to load the content of the cache with the selected IB-address values and lock the cache so that its contents remain available during the whole system execution.

First, our algorithm collects all the blocks that cause cache-misses by doing off-line analysis. Then the list of the blocks is sorted in a way so that the block that causes the most misses becomes the number one candidate to be locked, and so on. Major steps involved in our proposed algorithm are shown below,

```
Description: Determine the blocks to be locked
Input: IB-address-miss info based on the tree-graph
Output: Instruction cache-locking XML file
START:
▪ Read the Input File
▪ Create IB-Address Miss Block List
▪ Sort IB-Address Miss Block List
▪ List the Candidate Blocks
▪ Create Instruction Cache-Locking XML File
END
```

Determining the right amount of correct blocks to lock is the key to gain both predictability and performance improvement using this algorithm. This algorithm may also be used for pre-fetching and pre-loading.

V. Experimental Setup

In this work, we develop the simulation environment by configuring Heptane tool along with all required software components in Linux Red Hat 9. We use a simplified Pentium I processor as the target architecture. We obtain CPU utilization by varying the cache parameters for FFT, MI, and DFT applications.

A. Heptane

Heptane (Hades Embedded Processor Timing ANalyzEr) is a WCET analysis tool for embedded system. Before running Heptane, it must be configured. Once the configuration file is created, Heptane can be run by typing "heptane-run.sh"" in a command shell provided that the PATH variable contains the directory where Heptane is installed. The results will be placed in the directory as specified in the configuration file and can be viewed through a Web browser by opening the file "HTML/index.html" [13].

B. Target Architecture

The target architecture considered in this experiment is the simplified Pentium I (like the P54C) processor from Intel Corporation. The architecture model consists in a BTB (branch target buffer) and a CACHE system and a MEMORY description. No data cache is modeled, only one-level instruction cache is considered, one of the two integer pipelines is simulated, and branch prediction module is kept disabled. In this study, we consider an instruction cache with cache size ranges from 4 to 64 KB, line size from 32 to 512 Bytes, and the associativity level from 1 (direct-mapped cache) to 16 (set associative cache). An instruction is assumed to execute in 1 clock cycle in the case of a cache hit, and 10 clock cycles otherwise.

C. WCET and Static Cache Analysis

In Heptane, F. Mueller's static cache simulation technique is used to estimate the instruction cache behavior. WCET is obtained using the Heptane tree-based WCET analysis tool. Heptane computes WCETs through a bottom-up traversal of the syntax tree of the subject programs. Heptane includes hardware modeling capabilities so as to estimate the WCETs on architectures with instruction caches, pipelines, and simple branch predictors.

D. Static Cache-Locking

According to our algorithm, the block that causes more misses have higher chances to be locked. This algorithm aims at optimizing the task set schedule ability by minimizing the CPU utilization. On the considered architecture, when static cache-locking is used, the cache-related preemption delay is constant and equal to the delay required to refill the processor pre-fetch buffer (10 clock cycles in this case).

E. CPU Utilization

In Heptane, the CPU utilization (U) is measured using the WCET, periods of tasks, and the cache-related preemption delay as shown in Equation 1.

$$U = \sum_{i=1}^{n} \frac{C_i + \gamma_i}{P_i} \quad (1)$$

Here, γ_i is the upper bound on the cache-related preemption delay, n is the number of tasks, C_i and P_i are the WCET and period of task number i respectively.

F. Applications

In this work, we use three applications to run our simulation program, namely Fast Fourier Transform (FFT), Matrix Inversion (MI), and Discrete Fourier Transform (DFT). Table I shows computing time and WCET in terms of processor cycles for both cache analysis (non-locking) and instruction cache-locking. Here, locking decreases computing time, but increases WCET. So, the performance improvement depends on the right selection of the cache blocks to be locked.

TABLE I
APPLICATION STATISTICS

App.	Computing Time (Kilo Cycles)		WCET (Kilo Cycles)	
	No Locking	I-Cache Locking	No Locking	I-Cache Locking
FFT	121235	117813	58378	63123
MI	186519	145668	65673	71880
DFT	258456	186519	62673	65674

VI. RESULTS AND DISCUSSION

In this work, we implement a static instruction cache-locking algorithm and obtain CPU utilizations for FFT, MI, and DFT applications. CPU utilization obtained for 4 KB (and 8 KB) cache by varying cache-locking capacities (5% to 25% of the cache size) using FFT is shown in Table II. Results indicate that CPU utilization is the minimum (i.e., performance is the maximum) at 15% locking.

TABLE II
CACHE-LOCKING AND CPU UTILIZATION FOR FFT
LINE SIZE 128 BYTES, ASSOCIATIVITY 4-WAY

% Lock	Cache Size 4K		Cache Size 8K	
	Num of Block Locked	CPU Util.	Num of Block Locked	CPU Util.
5%	1	0.629	3	0.621
10%	3	0.606	6	0.618
15%	4	0.599	9	0.616
20%	6	0.605	12	0.617
25%	8	0.622	16	0.619

We also obtain CPU utilization for MI and DFT by varying cache-locking capacity. As shown in Fig. 1, at 15% cache-locking, all applications show minimum CPU utilization (i.e., maximum performance).

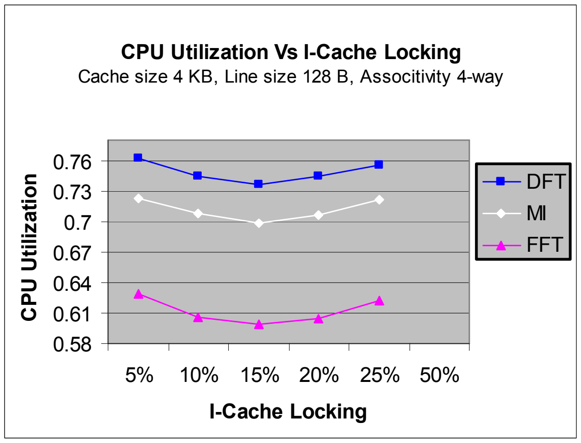

Fig. 1. CPU utilization for different instruction cache-locking capacities. CPU utilization is the minimum at 15% locking for all three applications.

Following subsections discuss the impacts of associativity level, line size, and cache size on performance for 15% cache-locking using FFT application.

CPU Utilization Vs Associativity Level

Using Heptane tool, we obtain the CPU utilization for both static cache analysis and cache-locking as shown in Fig. 2. Results show that for cache size 4 KB and line size 128 Bytes, the performance of static cache-locking scales better than the one of static cache analysis with an increasing level of associativity. Static cache-locking takes benefit of the increasing associativity level to eliminate both intra-task and inter-task interference.

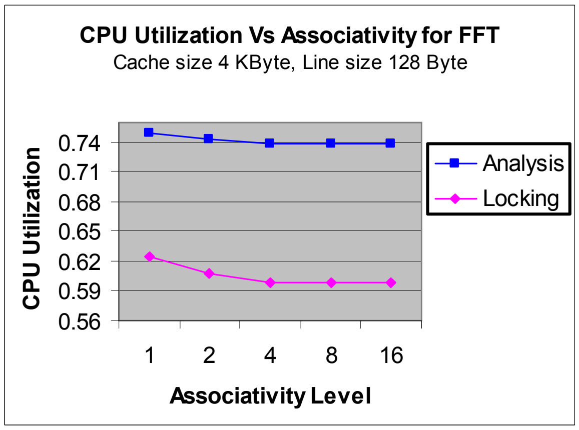

Fig. 2. CPU utilization for various levels of associativity. Cache-locking performs better than cache analysis for FFT.

CPU Utilization Vs Cache Line Size

CPU utilization obtained for both static cache analysis and static cache-locking for FFT application is shown in Fig. 3. For the cache size fixed at 4 KB and associativity level fixed at 4-way, the CPU utilization decreases (i.e., performance increases) with the increase of line size between 32 and 128 Bytes. For line size higher than 128 Bytes, CPU utilization increases (i.e., performance decreases).

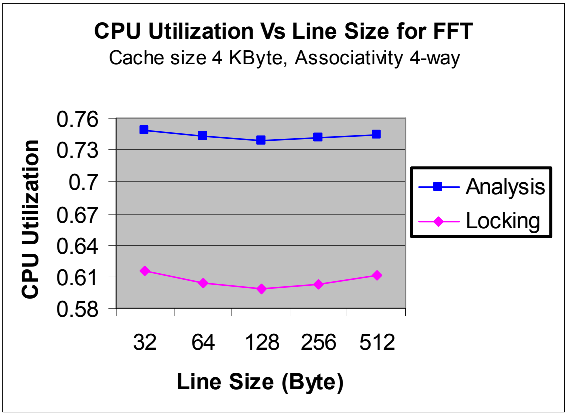

Fig. 3. CPU utilization for various line sizes. Static cache-locking performs better than static cache analysis for FFT.

CPU Utilization Vs Cache Size

Using Heptane, we investigate the impact of cache size on CPU utilization for FFT application. We keep line size fixed at 128 Bytes and associativity level at 4-way. Experimental results are shown in Fig. 4. For the given line size and associativity level, the CPU utilization of both static cache analysis and static cache-locking decreases (i.e., performance increases) with the increase of the cache size.

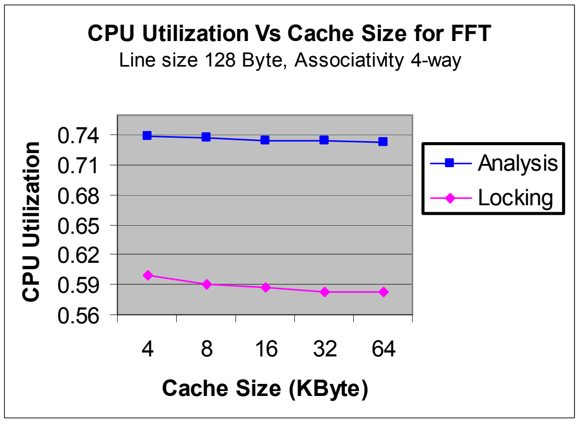

Fig. 4. CPU utilization for various cache sizes. The performance increase of static cache-locking is higher than the one of static cache analysis.

VII. Conclusion

Cache memory in real-time embedded systems is a great challenge to improve both predictability and performance at the same time. Studies show that for embedded systems where workload is almost known, static cache-locking helps to determine the worst case execution time (WCET) and cache-related preemption delay. In this work, we implement a static instruction cache-locking algorithm that makes the real-time embedded system more predictable. We obtain CPU utilization for both static cache analysis (no cache-locking) and instruction-cache locking using Heptane tool. Experimental results show that our cache-locking algorithm improves both predictability and performance when the right amounts of cache blocks are locked and appropriate cache parameters are used.

We plan to study the impact of cache-locking techniques on multi-level caches in our next endeavor.

References

[1] A. Asaduzzaman, I. Mahgoub, “Cache Modeling and Optimization for Portable Devices Running MPEG-4 Video Decoder,” *MTAP-06,* pp. 239-256, 2006.

[2] Z. Xu, S. Sohoni, R. Min, and Y. Hu, "An Analysis of Cache Performance of Multimedia Applications," *ACM SIGMETRICS Proceedings,* USA, 2001.

[3] I. Puaut, “Cache analysis vs static cache locking for schedulability analysis in multitasking real-time systems,” *23rd RT System Symposium,* INSA/IRISA, France, 2004.

[4] I. Puaut and D. Decotigny, "Low-Complexity Algorithm for Static-Cache Locking in Multitasking Hard Real time Systems," *Real-Time Systems Symposium,* 23rd IEEE Volume, pp. 114-123, 2002.

[5] J. Robertson and K. Gala, "Instruction and Data Cache Locking on the e300 Processor Core," *Freescale Semiconductor, Inc.,* 2006. freescale.com/files/netcomm/doc/app_note/AN2129.pdf.

[6] F. Sebek and J. Gustafsson, “Determining the Worst-Case Instruction Cache Miss-Ratio”, Sweden, 2002.

[7] C. Hong, K. Park, and Y. Song, “Hardware Support: A Cache Lock Mechanism without Retry,” *IEEE SNPD/SAWN’05,* 2005.

[8] A. Arnaud, et al, “Dynamic Instruction Cache Locking in Hard Real-Time Systems”, *INSA/IRISA,* France, 2005.

[9] X. Vera, B. Lisper, J. Xue, "Data caches in multitasking hard real-time systems," *Real-Time Systems Symposium,* 24th IEEE Volume, pp. 154-165, 2003.

[10] X. Vera, B. Lisper, J. Xue, "Data Cache locking for Higher Program Predictability,” *SIGMETRICS’03,* CA, USA, June 2003.

[11] M. Campoy, A.P. Ivars, J.V. Busquets-Mataix, "Static Use of Locking Caches in Multitask Preemptive Real-Time Systems", *IEEE Real-Time Embedded System Workshop,* London, UK, 2001.

[12] T. Ishihara, and F. Fallah, "A cache-defect-aware code placement algorithm for improving the performance of processors," *IEEE/ACM International Conference on Computer-Aided Design ICCAD'05,* pp. 995-1001, 2005.

[13] Heptane (Hades Embedded Processor Timing ANalyzEr), *A Static WCET Analyzer.* www.irisa.fr/aces/work/heptane-demo/heptane.html