International Journal of Electronics Communication and Computer Engineering
Volume 3, Issue 1, ISSN (Online): 2249–071X, ISSN (Print): 2278–4209

# Unavoidability Routine Enrichment for Real-Time Embedded Systems by Using Cache-Locking Technique

**M.Shankar[1], Dr.M.Sridar[2], Dr.M.Rajani[3]**

***Abstract*— In multitask, preemptive real-time systems, the use of cache memories make difficult the estimation of the response time of tasks, due to the dynamic, adaptive and non predictable behavior of cache memories. But many embedded and critical applications need the increase of performance provided by cache memories. Recent studies indicate that for application-specific embedded systems, static cache-locking helps determining the worst case execution time (WCET) and cache-related pre-emption delay. The determination of upper bounds on execution times, commonly called Worst-Case Execution Times (WCETs), is a necessary step in the development and validation process for hard real-time systems. This problem is hard if the underlying processor architecture has components such as caches, pipelines, branch prediction, and other speculative components. This article describes different approaches to this problem and surveys several commercially available tools and research prototypes**

***Key words* — Cache locking, real-time embedded system, Heptane, Miss Table, Multi-core architecture, Performance/power ratio, Timing predictability**

## I. INTRODUCTION

In hard real-time systems all task deadlines have to be met in all situations for safety reasons. For that reason, many schedulability analysis methods rely on the knowledge of an upper bound for the execution times of tasks (WCETs, for Worst-Case Execution Times). WCET estimates have to be safe (i.e. greater than any possible execution time) and as tight as possible (as close as possible to the execution time of the longest path). Safe bounds for task execution times may be computed using static WCET analysis methods that obtain WCETs through a static analysis of task source and/or object code. Cache locking shows promises in improving predictability as shown by Cache locking is defined as the ability to put off some or all of the data or instruction cache from being overwritten. Cache entries can be locked either for individual ways within the cache or for the entire cache. In entire cache locking, cache hits are treated in the same manner as hits to an unlocked cache. Cache misses are treated as a cache-inhibited access. Invalid cache entries at the time of the locking will remain invalid and inaccessible until the cache is unlocked. Entire cache locking is inefficient when the size of the data or the number of the instructions to be locked is small compared to the cache size. In way locking (a.k.a., set locking and partial locking), only a portion of the cache is locked by locking ways within the cache. Unlocked ways of the cache behave normally. Using way locking, the Xbox 360's Xenon processor achieves the performance of using local storage by the Synergistic Processing Elements (SPEs) in the IBM Cell processor. Way locking at the L1 cache is not permitted on some processors (like PowerPC 750GX), but way locking at the L2 cache is possible. Locking entries in the cache is a common and effective technique in reducing cache misses and therefore in helping to reduce execution time as well as execution time fluctuations (over back to back runs) which leads to better predictability and lower power consumption. The idea is that by locking important blocks in the cache, future accesses to these blocks in the cache are possible; thereby better predictability should be achieved with reduced memory access time and reduced total power consumption. In this paper, we explore instruction cache locking to improve the average-case execution time. Recently, Anand and Barua 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 the every iteration to evaluate the cost/benefit of locking the memory blocks. Thus the algorithm is more over, they employ some approximations in the cost/benefit analysis to cut down simulation cost leading to inaccuracy

## II. RELATED WORK

The performance and reliability of time-sensitive systems depends significantly on the execution environment (compilers, operating systems, processors, buses, I/O devices). It is often very expensive to rehost such systems when computing capacity is exceeded or the hardware becomes obsolete. Embedded real-time software is particularly difficult to rehost because of its tailoring and optimization to fit the limited resource footprint of the hardware [1] and the need to support specialized device interfaces. Avionics and flight control software adds to the complexity by requiring multilevel safety, fault tolerance, modular multiprocessor architectures, and very complex multi-mode system behavior. Because of the complexity of upgrading the software for a new processing environment, one of the most significant risks in system development of large real-time systems, especially avionics and flight control systems, is the problem of exceeding the computational resources during the software development process and during the operational lifetime of the system. Program after program has had to scale back system requirements to fit on the hardware. Integration, maintenance, and upgrade costs are driven up since software must be shoehorned into the available resources

for as long as possible. Caches raise predictability issues in hard real-time systems because they are designed to speed up the system average case performance rather than the system worst-case performance which is of prime importance in hard real-time systems. As a consequence, the designers of hard real-time systems may choose not to use cache memories at all, or may choose to use on-chip static RAM – scratchpad memories instead of caches. A second class of approaches [2] to deal with caches in real-time systems is to use them in a restricted or customized manner, so as to adapt them to the needs of real-time systems and schedulability analysis. Cache partitioning techniques assign reserved portions of the cache (partitions) to certain tasks in order to guarantee that their most recently used code or data will remain in the cache while the processor executes other tasks. The dynamic behavior of the cache is kept within partitions. These techniques eliminate the inter-task interferences, but need extra-support to tackle intra-task interference (e.g. static cache analysis) and reduce the amount of cache memory available for each task.

## III. CACHE-LOCKING ALGORITHMS

In this paper, we propose algorithms 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. It is performed in a non-blind manner by using memory access patterns obtained by profiling the task. The goal is to improve the worst-case performance as compared with a system with no cache; in such a way that this performance be comparable with results obtained from static analysis of the same cache whose replacement policy is the least recently used (LRU). Our algorithms determine a set of functions of a program that is locked into the cache at system start up time. During the whole execution time of the program exploiting cache locking, the locked cache's contents remains invariant (\static locking"). Cache contents selection is done such that the set of selected functions leads to the highest WCET reductions.

### *A. Low-Complexity Cache-Locking Algorithm*

In this paper, we propose two algorithms to select the contents of the instruction cache for periodic task sets. In contrast to [CIBM01], the proposed algorithms select the contents of the locked cache in a non blind manner, by using the tasks memory access patterns of the instruction flow. An important property of both algorithms is their low complexity (pseudo-polynomial). Both algorithms optimize the worst-case behavior of the task set: the first one aims at minimizing the CPU utilization of the task set, and does not prescribe any particular schedulability analysis method, whereas the second one, designed for fixed-priority schedulers, aims at reducing the interferences between tasks due to preemptions. The worst-case performances of benchmark systems using the proposed cache contents selection algorithms are analyzed. Another contribution of this work is a comparison of the performance of task sets using static cache locking with a state of the art static cache analysis technique.

### *B. Region Merging and In Lining (RMI) Algorithm*

RPC, however, does not translate well into distributed object systems where communication between program level objects residing in different address spaces is needed. In order to match the semantics of object invocation, distributed object systems require remote method invocation or RMI. In such systems, the programme has the illusion of invoking a method on an object, when in fact the invocation may act on a remote object (one not resident in the caller's address space). In this work the integration of sequential development and master-slave models to calculate Mat Mul by using RMI Java threads. In this proposed model, the server determines the distributed numbers of rows from the first matrix and the columns of the second matrix depending on the balance of workload on registered clients. For example, when n = 20, which mean the size of 2- matrix multiplication is 20 X 20.

### *C. Algorithm for Selective Cache Loading*

Our work is based in the IDT-79R4650 cache schema. This processor offers an 8KB, two-set associative instruction cache. Also, the processor offers the instruction "cache fill" to selective load cache contents. However, this processor allows locking only one set of cache, leaving unlocked the other. Since the main objective of this work is to reach a deterministic cache [3], we need to lock the entire cache. In the MPC7400 it is possible to lock the entire cache, using a one-cache-line size buffer to temporally store instructions not loaded in cache, improving sequential access. However, the selective load of cache contents is not available. This way, in this work, a merge of the two above processor is proposed, resulting in a cache system with the following characteristics.

### *D. Cache-Defect-Aware Code Placement Algorithm*

We propose a defect-aware code placement technique which reduces the performance degradation of a processor with a partially good cache memory. We used FT bits in our technique. Our 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. To the best of our knowledge, this is the first compiler technique which reduces the performance degradation of a partially good cache memory. The rest of the paper is organized as follows. In Section 2, we summarize previous work and our approach. The definition of the problem and our algorithm for solving it are presented in Section 3. Section 4 presents experimental results.

## IV. STATIC CACHE-LOCKING ALGORITHM

The static cache locking algorithm implemented [13] selects the contents of the statically locked cache according to the knowledge of the tasks memory accesses, obtained using simulation. We compare the worst-case performance of this task set with the one obtained through the use of a state of the art cache analysis technique based on F. Mueller's work on static cache simulation for details). The Heptane tree-based WCET analysis tool has been used to compute WCETs. No attempt is made here to bound the cache-related preemption delay precisely[4] (it

is assumed that all program lines of a given task have to be reloaded after a preemption, with a maximum of N reloads where N is the number of cache lines). They can be created from static analysis, dynamic characteristics, or other sources. Some practical systems that rely on Graph viz for software visualization are the Acacia Doxygen and Mono static analysis systems, the Syntacs toolkit for Java compiler generation, the Spin concurrent protocol analyzer and the Bugzilla bug tracking system originally created for the Mozilla (Netscape) open source project. Graphviz has also been applied to digital logic design, database schema design, knowledge representation, Bayesian networks and decision diagrams, to name a few other areas in related branches of engineering and technology.

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*

This allows the researcher to modify and extend the tool for his/her individual needs. Current state of the art WCET analyzers, such as tare commercial tools that will not provide the source code. Even most of the research prototypes, such as Cinderella do not make the source code available. The only notable exception in this aspect is HEPTANE, which is open source. However, HEPTANE does not support complex architectural features such as out-of-order pipeline and global branch prediction

### *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*

Cache partitioning techniques assign reserved portions of the cache (partitions) to certain tasks in order to guarantee that their most recently used code or data will remain in the cache while the processor executes other tasks [5]. The dynamic behavior of the cache is kept within partitions. These techniques eliminate the inter-task interferences, but need extra-support to tackle intra-task interference (e.g. static cache analysis) and reduce the amount of cache memory available for each task.

### *D. Static Cache-Locking*

The cache contents can be loaded and locked at system start for the whole system lifetime (static cache locking), or changed during the system execution, like for instance when a task is preempted by another one (dynamic cache locking). The key property of cache locking techniques is that the time required to access the memory is predictable Regarding static locking of instruction caches, two classes solutions have been proposed: one using a genetic algorithm for cache contents selection and a pragmatically algorithm, called hereafter reference-based algorithm, which uses the string of memory references issued by a task as an input of the cache contents selection [6,7] algorithm . These two classes of solution can be tailored to select the cache contents at the system level (global locking) or at the task-level (local locking) with then the necessity of reloading the cache contents at context switch points. In this paper, we concentrate on global locking.

### *E. CPU Utilization*

This comparison of one-client tests to corresponding two-client tests shows that the native tests scale almost perfectly: both throughput and CPU utilization double. VMware ESX Server does very well, too: the throughput [8,9] for two-client tests goes up 1.9-–2 times compared to the one-client tests. Xen is almost CPU saturated for the

one-client case, hence it does not get much scaling and even slows down for the send case.

### 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 | 235253 | 562315 | 65856 | 85695 |
| MI | 589647 | 859647 | 52361 | 45625 |
| DFT | 289635 | 568963 | 52614 | 74126 |

## 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 (15% to 35% 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 20% 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 Utility | Num of Block locked | CPU Utility |
| 6% | 1 | 0.700 | 6 | 0.700 |
| 20% | 3 | 0.699 | 12 | 0.752 |
| 30% | 4 | 0.702 | 15 | 0.746 |
| 40% | 6 | 0.685 | 17 | 0.785 |
| 50% | 8 | 0.684 | 19 | 0.785 |

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

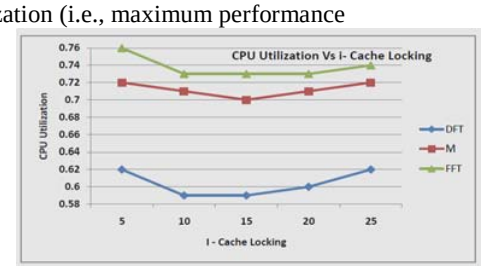


Fig.1 CPU Utilization for different instruction cache locking capacities. CPU utilization is the minimum at 15% locking for all three applications

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

### A. CPU Utilization Vs Associativity Level

In this work, we focus on the impact of various cache design parameters namely, cache size, line size, associativity, and cache levels on the performance of MPEG4 decoding algorithm running on a single processor system. Using Visual Sim, we measure the following two performance metrics – utilization and transactions Using Heptane tool, we obtain the [10, 11] 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.

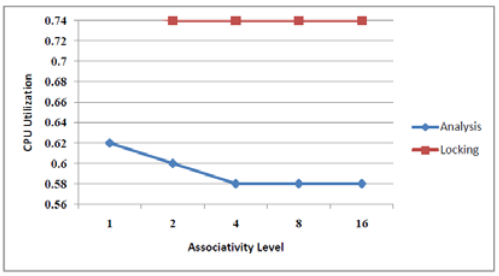


Fig.2 CPU Utilization for various levels of associativity. Cache-locking performs better than cache analysis for FFT

### B. CPU Utilization Vs Cache Line Size

MM transactions decrease with the increase of CL2 size. For a total of 10,000 tasks, 3,333 are data and 6,667 are instructions. All tasks are initiated at CPU and referred to CL1 (D1+I1). For D1 hit ratio 5.0% and I1 hit ratio 2.0%, 168 + 135 = 303 task requests go to CL2. For CL2 size 32 KB (miss ratio is 0.9%), only 3 requests go to MM via the shared bus. For CL2 size 2 MB and higher (miss ratio is 0%), no requests go to main memory. Figure 9 shows the impact of CL2 size variation on CPU utilization (without and with CL1). CPU utilization decreases with the increase of CL2 size.

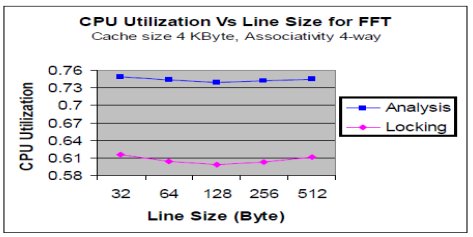


Fig.3. CPU utilization for various line sizes. Static cache locking performs better than static cache analysis for FFT.

### C. CPU Utilization Vs Cache Size

The tests include native and virtual experiments at 1, 2, 4, and 8 CPUs. For the native experiments, the BCD edit Windows utility was used to specify the number of processors to be booted. The size of the database, the

memory size, and SQL Server buffer cache were carefully scaled to ensure the workload profile was unchanged, while getting the best performance from the system [12,13] for each data point. All the scale up experiments ran at 100 percent CPU utilization in native and virtual configurations. For example, at 2 v CPUs, in the guest, the workload would saturate both virtual CPUs.

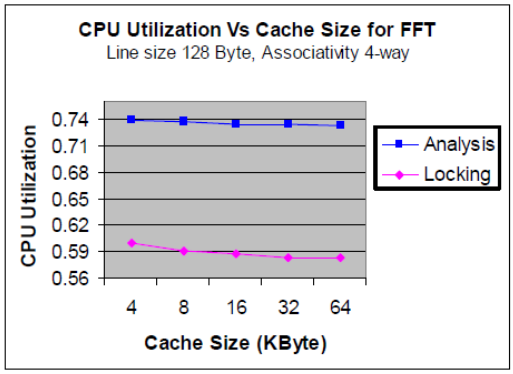


Fig.4. CPU utilization for various cache sizes. The performance increase of static cache-locking is higher than the one of static cache analysis

## 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. As a further work, it would be interesting to explore the transposition of the RMI algorithm (i.e. we keep the basic merging and in lining operations) from a greedy algorithm towards a genetic algorithm. The main reason is that a genetic algorithm exhibits a better exploration of a solution space and thus might find sets of locked cache states which would lead to better improvements on worst-case performances. Another direction would be to adapt this work in other situations. It could be easily achieved for multi-level instruction caches. Finally the adaptation to data caches should be investigated.

## REFERENCES


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

[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

[6] C. Lee, J. Hahn, Y. Seo, S. Min, R. Ha, S. Hong, C. Park, M. Lee, and C. S.Kim. Analysis of cache-related preemption delay in fixed-priority preemptive scheduling. IEEE Transactions on Computers, 47(6):700–713, June 1998

[7] T. Lundqvist and P. Stenström. An integrated path and timing analysis method based on cycle-level symbolic execution. Real-TimeSystems, 17(2-3):183–207, 1999.

[8] F. Mueller. Compiler support for software based cache partitioning. In LCTES ’95: Proceedingsof the ACM SIGPLAN 1995 workshop on Languages, compilers, & tools for real-time systems, pages 125–133, New York, NY, USA

[9] F. Mueller. Timing analysis for instruction caches. Real-time systems, 18(2):217–247, May 2000.

[10] I. Puaut, A. Arnaud, and D. Decotigny. Analyse de performance de methods de verrouillage statique de caches dans les systems temps-réel strict. In Proc. of the 12th International Conference on Real-Time Systems (RTS’04), March 2004.

[11] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multi taskinghard real-time systems. In Proceedings of the 23rd IEEE International Real-Time Systems Symposium (RTSS ’02), Austin, TX, USA.

[12] S. Steinke, N. Grunwald, L. Wehmeyer, R. Banakar, M. Balakrishnan, and P. Marwedel. Reducing energy consumption by dynamic copying of instructions onto onchip memory. In ISSS ’02: Proceedings of the15th international symposium on System Synthesis, pages 213–218, New York, NY, USA, 2002. ACM Press.

[13] X.Vera, B. Lisper, and J.Xue. Data caches in multitasking hard real-time systems. In 24th IEEE International Real-Time Systems Symposium (RTSS ’03), Cancun, Mexico, 2003.IEEE Computer Society Press.



**M.Shankar** received B.E degree in Electrical and Electronics Engineering from Madras University, Tamilnadu, India in 2003 and the M.Tech degree in embedded system Technology from Bharath University, Chennai, India in 2006.Currently doing research in Bharath University Chennai. He is a lie member of Computer Society of India (CSI) and published many national and international research articles. His researches interest is Timing analysis in embedded system software.

**Dr.M.Sridar** Currently working as Director - International relations, Bharath University. He has published many international research articles. His researches interests are timing analysis and security for embedded systems.

**Dr.M.Rajani** Currently working as Research director in Bharath University in Chennai, India. She has published many international research articles. Her research interests are error correcting codes addresses effectively decoding algorithm and VLSI Architecture.