24th IEEE International Conference on Emerging Technologies And Factory Automation (ETFA 2019)
Zaragoza, Spain. ISBN 978-1-7281-0303-7

Combining watchdog processor with instruction cache locking for a fault-tolerant, predictable architecture applied to fixed-priority, preemptive, multitasking real-time systems.

Antonio Martí-Campoy, Francisco Rodríguez-Ballester
Institute of Information and Communication Technologies (ITACA)
Universitat Politècnica de València
València, Spain
amarti@disca.upv.es, prodrig@disca.upv.es

***Abstract*—Control flow monitoring using a watchdog processor is a well-known technique to increase the dependability of a microprocessor system. Most approaches embed reference signatures for the watchdog processor into the processor instruction stream. These signatures contain the information required to detect control flow errors during program execution by the main processor. This paper proposes an architecture that offers both fault-tolerance and dynamic cache locking combined. This combination is achieved taking advantage of the fact that watchdog processor signatures are inserted along the program code. Then cache locking information is incorporated into these signatures. And also the required circuitry to inform the cache controller whether to lock or not the instructions fetched by the main processor is added into the watchdog processor. With this approach both fault-tolerant and real-time features are supported by the same hardware, therefore saving room on the silicon die or FPGA size. Results from experiments show that in most cases this approach reaches the same performance than previous, hardware-costly proposals.**

***Index Terms*—Fault-Tolerant Systems; Real-Time Systems; Watchdog Processor; Cache Locking; Multitasking; Embedded Systems.**

I. INTRODUCTION

A fault-tolerant system may be implemented using well-known techniques: replication, redundancy, or diversity. Each technique presents different degrees of reliability and responses in front of a failure, but all the three uses replication of hardware elements as a way to achieve fault-tolerance.

A watchdog processor [8], [5] is an alternative mechanism for error detection in order to add fault-tolerance to a system using less hardware than replication, redundancy, or diversity.

Instead of replicating the whole processor of a system, a watchdog processor is a relatively small coprocessor added to the system with the purpose of monitoring the activity of the main processor and detect deviations from the expected execution flow. This watchdog processor can detect *control flow errors*, errors in which the main processor deviates from the expected execution flow.

The information used by the watchdog processor to determine what is the correct, expected execution flow of the main processor is introduced into the system through the use of *signatures* or specialized instructions. In the most common case the signatures are fetched by the main processor and interpreted as no-operation instructions (NOP); other proposals [5] allow signatures to be interspersed with the instructions of the main processor but to remain unseen (neither fetched nor executed) by the main processor.

In the finest-grained control architectures a signature is inserted per *basic block* of instructions. A basic block is a sequence of instructions with only one possible execution flow that contains, at most, one *branch* instruction. In this context a generic *branch* instruction is any instruction that can break the sequential flow of execution: procedure call or return, conditional branch, or unconditional jump. A *branch-in* instruction is an instruction used as the destination of a generic branch instruction.

If the program instructions are divided into generic branch instructions, branch-in instructions, and blocks of branch-free instructions (with neither generic branches nor branch-in instructions), a basic block can be formally defined as a sequence of branch-free instructions started by the preceding branch-in (if any) and ended by the following generic branch instruction (if any). A basic block may not be started by a branch-in instruction if it follows a conditional branch instruction; a basic block may not be ended by a generic branch instruction if a branch-in is encountered (that starts a new basic block).

Cache memories are an important advance in computer architecture, improving system performance. The dynamic and adaptive behaviour of a cache memory reduces the average access time to main memory, but presents a non deterministic fetching time [6]. This lack of determinism is a serious threat in real-time systems, where execution and response time of tasks must be known or at least bounded. The use of cache memories poses two problems. The first one is to estimate the worst case execution time (WCET) of a task taking into account the intra-task or intrinsic interference, that is, the misses produced because the tasks remove from cache their own instructions [20]. The second one is to estimate the worst case response time (WCRT) taking into account the inter-task or extrinsic interference in preemptive, multitasking systems, where the preempting task removes from cache instructions belonging to the preempted task [7] [19]. This way, when the preempted task resumes execution, a burst of cache misses can happen that has not been accounted in the task's WCET.

Cache locking is a well-known technique to gain predictability when estimating the response time of a task when the architecture of the system includes a cache memory. By predictability we mean the ability to obtain a bounded, safe, accurate, and feasible estimation of the WCET of a task and the WCRT for multitasking systems. This way, it is possible to carry out an schedulability analysis in order to check if all tasks will meet their deadlines.

Regarding the use of cache locking there are *static* and *dynamic* techniques and also *full* or *partial* techniques.

When static cache locking is used, the cache contents are loaded and locked during the system start-up and cache contents remain unchanged during system execution. The maximum number of main memory blocks that can be selected is equal to the size of the cache in terms of cache lines. The main goal of this technique is to get full predictability [9] [16].

In dynamic cache locking, cache contents may change during system execution. During system design, the memory blocks to be loaded and locked in cache are selected, but this number can be larger than the size of the cache. During system run, cache contents may be updated with a subset of the selected blocks. This cache update can be at inter-task level [9] [2] or at intra-task level [15]. In dynamic cache locking an improvement in performance is pursued although the predictability is reduced and then safe upper bounds of execution and response times can not be as accurate as in static cache locking, although performance is still improved. Also, schedulability analysis becomes more complex.

In both static and dynamic cache locking, the whole cache can be locked; this is known as full cache locking. Or only some part of the cache is locked (partial locking). Full cache locking provides a large degree of predictability. On the other hand, partial cache locking may allow the existence in the system of non-critical tasks running at full speed by using part of the cache with no restriction. In [11] is presented a comprehensive and accurate review of the use of cache locking.

One of the challenges of cache locking techniques is to design the hardware to support it. Following section discuss this issue and states our proposal. Next sections describe how to implement cache locking with the help of the watchdog processor, and how to select cache contents. The paper finishes with some experimental results and conclusions.

II. Rationale

The use of cache locking requires to select the memory blocks that will be loaded and locked in cache memory. Once the memory blocks have been selected, they have to be loaded in cache and then locked. Depending on the technique of cache locking employed, there are several proposals to carry out the load and lock operations that may rely on software or hardware solutions.

In software-based works cache locking is implemented using the available processor instructions to load and lock main memory blocks. In [9] the load and lock instructions are executed by the system scheduler when a task switch happens, allowing inter-task replacements but not intra-task. In [15] [12] [1] the instructions to lock/unlock the cache are spread across the code, locating them just when they are needed. This allows to improve performance thanks to a fine-grain control of cache contents, and also allowing intra-task replacements. However, this method presents two disadvantages. First, the number of instructions added can be significant, so memory requirements may increase and the overload to execute them may degrade performance. Second, the instructions to control the cache have to be inserted into the code, shifting the task instructions and modifying the application memory mapping [11], thus probably requiring a new analysis.

To authors knowledge, there are two proposals that add hardware resources to implement cache locking. [18] proposes the addition of a side memory, Locking State Memory (LSM), and [4] proposes the addition of a combinational circuit, known as Locking State Generator (LSG), inside of an FPGA. In both cases the hardware tells the cache controller when to lock or not to lock the main memory block that is being fetched. This way any of the techniques of cache locking can be implemented. Despite the cost associated to add a new piece of hardware, the main advantage of the hardware support is that cache locking is transparent to the programmer and to the compiler because no modification of compiled code is needed.

Focusing on the hardware approach, its main drawback is the associated costs in terms of silicon die fabric or FPGA size. This can be minimized if an already existent hardware can be slightly modified to incorporate the circuitry to signal the cache controller when a main memory block has to be locked in cache. This is the case of the watchdog processor in this proposal; its main purpose is to detect control flow errors, but it can also be used to implement cache locking without a significant increment of hardware costs.

III. Adding cache locking

The proposal of this work is to add one bit into the watchdog signatures to tell the cache controller if the cache has to be locked or not. If the cache is locked then no content can be displaced by new fetched instructions; if the cache is not locked then newly arrived instructions will displace those already stored into the cache. The cache will remain in this state, locked or not locked, until a new signature tells the cache controller if the cache must remain in the same state (locked or not) or swap its state. The watchdog processor is in charge of reading this bit from the signatures and translate it to the cache controller.

As the signature precedes the instructions the processor is going to execute, if the bit compels the cache controller to lock the cache current contents this means not to load into

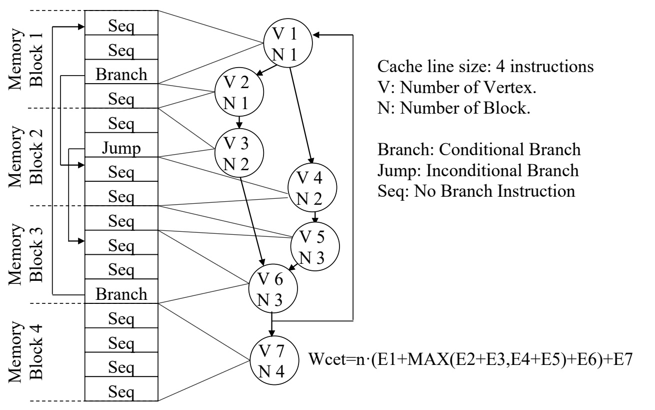

Fig. 1: Example of C-CFG and related expression to compute the WCET.

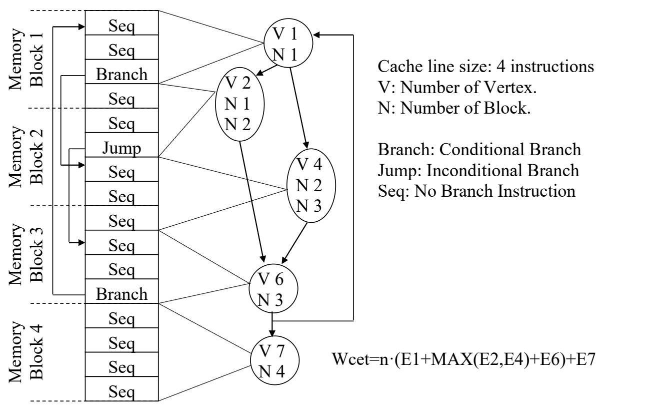

Fig. 2: Example of CFG and related expression to compute the WCET. Vertexes 3 and 5 are removed on purpose to ease the comparison with C-CFG.

the cache the following fetched main memory blocks. On the other hand, if the bit compels the cache controller to unlock the cache current contents, the following main memory blocks will be loaded in cache as they are fetched by the processor.

There are some differences in the described behaviour in front of previous proposals as [18] and [4] where ad-hoc hardware is added to lock and unlock the cache.

The main difference resides in the way main memory blocks are selected to be locked in cache.

The WCET of a task can be calculated using the Control Flow Graph [17] that leads to a set of expressions that can be computed to obtain the WCET of a task. In the CFG, there is a vertex for each basic block, that is, for each sequence of instructions with no flow break. But, in order to improve system performance in presence of cache locking, previous proposals use the Cached-Control Flow Graph, C-CFG, where there is a vertex for each cache basic block. A cache basic block is a sequence of instructions with no flow break and all the instructions belonging to the same main memory block. Figure 1 shows an example of a C-CFG for a piece of code with one branch inside a loop. The figure also shows the expression to compute the WCET, where E_i is the execution time of the vertex V_i, whose instructions belongs to main memory block N_i. The time of executing a vertex E_i changes if the main memory block N_i is selected or not selected to be locked in cache memory. If the block is selected to be loaded and locked in cache, its execution time is $T_{hit} * I_i$. If the block is not selected to be loaded in cache its execution time is $T_{miss} + (T_{hit} * I_i)$. Finally, the time to load into cache the selected main memory blocks, $T_{miss} * B$, is added to the final WCET of the task. T_{miss} is the penalty for a miss access in cache, T_{hit} is the time of executing an instruction from cache memory, I_i is the number of instructions of vertex i, B is the number of main memory blocks selected to load and lock in cache, and n is the number of iterations of the loop.

Dividing the basic blocks of instructions into cache basic blocks allows a more fine-grained selection of the contents to be locked in cache, irrespective of the tool or method used to select them. Regarding main memory blocks 1 and 2 in figure 1, there are four options to select them to be locked in cache: block 1 only, block 2 only, both blocks 1 and 2, or none of them. This is possible because each vertex in the C-CFG has its own cache control instruction to load it (software approach), or each main memory block is marked in the LSM or LSG to be loaded or not (hardware approaches).

But in the proposal presented here there isn't one bit per cache basic block as there is only a watchdog signature per basic block of instructions (a vertex of the CFG). Therefore, this proposal has to work with basic blocks of instructions, because it is built over the watchdog schema. This means that it is not possible to individually select each main memory block due to the fact several different main memory blocks can belong to the same vertex of the CFG.

For the same example code than figure 1, figure 2 shows the CFG, that is, using only basic blocks of instructions. Now vertex 3 is joined with vertex 2 (same basic block of instructions), that includes instructions belonging to main memory blocks 1 and 2, and vertex 4 is joined with vertex 5, including instructions belonging to main memory blocks 2 and 3. Since the technique of the watchdog processor inserts a signature only at the beginning of each basic block of instructions, both memory blocks 1 and 2 have to be selected together or not selected at all. Even more, if memory block 3 is to be selected, blocks 1 and 2 must be also selected in order to guarantee that blocks are loaded in cache independently of the execution path followed when the task runs. This way, the algorithm to select which blocks will be loaded and locked in cache has to be aware of the interference between vertexes, as described in the following section.

It would be possible to use the C-CFG by adding a new signature at the beginning of each cache basic block (interpreted as a NOP instruction by the main processor). Using these new signatures the watchdog processor would be able to inform the cache controller whether to lock or not main memory blocks in a one per one basis. However adding these signatures would enlarge the code size and reduce performance because the processor would have to execute more signatures. Moreover, these signatures do not improve reliability (control flow error detection), so in this paper basic blocks of instructions are used, and the impact of this decision over performance is evaluated in the section that discusses the experiments carried out.

Regarding the estimation of tasks' WCET and WCRT, and thus solving the schedulability analysis, the use of the C-CFG or the CFG does not matter, because to estimate execution and response times it is only needed to know if a main memory block is in cache. And a block will be always in cache if it is selected to be loaded and locked. On the other hand, if the block is not selected, the safe estimation is to consider all the accesses to this block as a cache miss. So, regarding schedulability analysis, this can be accomplished using the CRTA (Cache Response Time Analyse) [3] in the way described in [18].

However, the use of the C-CFG or the CFG matters when selecting the blocks to be loaded and locked in cache. Next section presents the details of how to carry out this selection.

IV. Selecting contents to be locked

There are several ways to select the contents to load and lock into the cache: integer linear programming, greedy, iterative, and genetic algorithms [11]. Without regard of the algorithm selected, the main goal is to make a coherent selection of blocks. This means to fulfil the mapping schema of the cache and the behaviour of the underlying hardware, while at the same time optimizing some system parameter.

In this work a modified version of the genetic algorithm presented in [10] is used. There are two main modifications to this algorithm. The first one is changing the fitness function in such a way that the lower system utilisation, the better solution. The second modification comes from the use of the CFG instead of the C-CFG to model the flow graph of tasks. In the original algorithm the codification of the problem is, basically, an array of mb elements, where mb is the number of main memory blocks. The value of each element determines if the related main memory block is selected to be loaded and locked in cache. The original algorithm has to meet some restrictions:

- The number of selected blocks for a task has to be equal or smaller than the number of lines of the cache. That is, no intra-task replacements are allowed, making easier the schedulability analysis.
- The number of selected blocks that map in a cache set has to be equal or smaller that the number of ways of the cache. As the previous requirement, this eliminates the intra-task interference.
- When the genetic algorithm performs crossover and mutation operations, the result of these operators has to meet the two previous restrictions.

The use of CFG imposes news restrictions when selecting blocks:

- If a memory block is selected to be loaded and locked in cache, all the blocks belonging to the same vertex in the CFG have to be selected. Since the signature that lock or unlock the cache contents is inserted at the beginning of the vertex, the cache will remain in the same state for the execution of the whole vertex. This is the case, for example, of vertex 2 in figure 2, where blocks 1 and 2 have to be locked or unlocked together.
- If a memory block B_i is selected, and this block is shared between two or more vertexes in the CFG, all the blocks belonging to all these vertexes sharing the block B_i have to be selected. This is a consequence of the previous restriction and helps to get predictability, keeping the same cache contents irrespective of the execution path. For example, in figure 2 if memory block 3 is selected, block 2 has also to be selected, and therefore block 1 has also to be selected.
- The two previous restrictions apply in the same way for non selected blocks.
- If the selected blocks for a given vertex create an access conflict (two or more blocks map onto the same cache line), then all the blocks will be marked as not selected. This requirement precludes the existence of intra-task replacements.

Listing 1: Genetic algorithm pseudo-code

```
function selectCacheContents(SystemParams, CacheParams)

  // Initialization phase
  population = initializePopulation(SystemParams, CacheParams)

  // Evolution phase
  for each value 1 to GENERATION

    // Evaluate fitness of population
    evaluateFitness(population)

    // Elitist selection added to next generation
    elite = selectBestIndividuals(population, 2)

    // Next generation's population
    for each value 1 to (POPULATION−2)/2
      parent_1 = selectParent(population, BinaryTournament)
      parent_2 = selectParent(population, BinaryTournament)
      cross_point = randomPoint(SystemParams.num_vertexes)
      children = crossParents(parent_1, parent_2, cross_point)
      addIndividuals(new_population, children, 2)
    end for

    // Mutation phase
    mutate(new_population, MUTATION_PROB)

    // Elite from previous generation added
    addIndividuals(new_population, elite, 2)

    population = new_population
  end for

  return selectBestIndividual(population)

end function
```

It is worth noting that previous restrictions are devoted to reach predictability and allow an easier schedulability analysis.

In order to implement the new restrictions, the way the genetic algorithm encodes the individuals is changed. An array of nv elements is created with nv being the number of vertexes of all tasks in the CFG. Each element of this array determines whether the vertex is selected to load and lock all its main memory blocks or not. If the vertex is selected, the bit in the signature at the beginning of its corresponding basic block is set to 1, meaning to unlock the cache contents and allowing fetched blocks to enter the cache. If the vertex is not selected to load and lock its main memory blocks, the bit in the signature is set to 0 so the cache contents are locked and thus preventing fetched blocks to enter the cache and produce cache replacements.

Genetic algorithm operators of initialisation, crossover and mutation are carried out guaranteeing that the previous stated restrictions are met. Listing 1 shows the pseudocode of the genetic algorithm, and below some detail of the operators is given.

- SystemParam and CacheParam: the CFG of tasks including the signatures for the watchdog processor; Real-time system parameters, like task periods and priorities; Cache parameters, like size and number of ways, hit a miss times.
- Initialisation: all individuals are created with a pseudo-random set of vertexes and thus, main memory blocks, marked as locked.
- Fitness function: fitness of an individual is the system utilisation in the range]0, 1[. The system utilisation is computed using CRTA and considering the vertexes, and thus, the main memory blocks, selected to be locked in cache. Non schedulable systems presents an utilisation of 1.
- Selection policy: binary tournament is used in order to reduce the execution time of the GA [14]. Two individulas are pseudo-randomly choosen and the better, the one with lower utilisation, will became one of the parents. The other parent is choosen repeating the same procedure.
- Crossover: one single point is randomly choosen and the two parents are merged, creating two new individuals.
- Mutation: two vertexes are pseudo-randomly choosen and their states are exchanged. The result can be a larger, equal or smaller number of vertexes selected to be locked.
- Elitist selection: the two best individuals from the previous generation are copied to the new generation with no crossover nor mutation.

The input to the genetic algorithm is all the information related to the real-time system and the cache memory. The result of the genetic algorithm is the WCET and WCRT of each task, the answer to the schedulability test, and the value of the cache-control bit that has to be set in each signature.

V. Experiments

The objective of the experiments is to estimate the cost of getting predictability. This cost is measured in terms of loss of performance between the previous hardware-assisted proposals, LSM and LSG, in front of the proposal presented in this paper that uses signatures for the watchdog processor to lock and unlock the cache. Performance is assessed by means of system utilisation. The higher the utilisation, the lower the performance.

Experiments are carried out using the real-time systems presented in [10]. In this set of systems there are 14 different task sets. Table I shows the main characteristics of tasks and task sets. Tasks are artificially created to stress the cache locking scheme. Main parameters of each task are defined, like its size, the number and size of loops and their nesting level, number of if-then-else structures and their respective sizes. These parameters are fixed or randomly selected. Then, a simple software tool creates the code using MIPS R2000 [13] compatible assembly language.

TABLE I: Main characteristics of task sets.

Characteristic	Minimum	Maximum
Number of tasks in the set	3	8
Task size	1.6KB	27.6KB
Task set size	12.5KB	57.6KB
Instructions executed per task (approx.)	50,000	8,000,000
Instructions executed per set (approx.)	200,000	10,000,000

Task periods are manually adjusted to force different number of preemptions among the tasks and therefore setting the system utilisation in different values. For each task set, two different sets of periods are assigned, where system utilisation is higher as the periods become shorter. For all task sets, tasks' deadlines are equal to tasks' periods and priority is assigned by the Rate Monotonic policy (the shorter the period the higher the priority).

A total of 28 systems = 14 (task sets) x 2 (periods sets) are evaluated using seven cache sizes (in cache lines, where cache line size is four instructions): 64, 128, 256, 512, 1024, 2048, and 4096 lines. The total number of experiments is 196. Regarding the cache mapping policy, direct mapping has been used since it is the most restrictive mapping policy for cache locking.

Using the same task set but different task periods and cache sizes allows assessing the behaviour of the architectures and algorithms in front of different scenarios.

The main parameters of the genetic algorithm are:

- Number of individuals: 240
- Number of generations: 5.000
- Crossover probability: 100 %
- Mutation probability: 8 %
- Number of repetitions to avoid the effect of seed for pseudo-random numbers generator: 25

The execution of the 196 experiments finished in less than 10 hours running on a desktop personal computer with an Intel I5 microprocessor.

From here after, U_w refers to system utilisation when the system runs using dynamic cache locking and the main memory blocks are locked by means of signatures for the watchdog processor. In the other hand, U_l refers to system utilisation when the system runs using dynamic cache locking and the main memory blocks are locked by means of LSM/LSG added hardware. Both utilisations have been computed from the genetic algorithm, using the presented WCET and CRTA analysis and the list of selected blocks. Therefore, the values presented are estimated, and represent an upper, safe bound of the actual utilisation values.

From the 196 experiments, none shows a performance gain ($U_w - U_l < 0$) using signatures to encode the cache locking information. This is due to the fact that using the CFG reduce the freedom degree when selecting main blocks to be loaded and locked into the cache. Using the LSM/LSG approach, however, the genetic algorithm can select each block individually.

In 153 of them (78%) the loss of performance ($U_w - U_l$) is less than one percent as can be shown in Figure 3. This means that for a large set of systems the same performance can be obtained implementing cache locking by means of a modified watchdog processor than by means of the LSM/LSG hardware.

For the remaining 43 experiments, where ($U_w - U_l$) is greater than one per cent, table II shows the main statistics. Figure 4 shows the frequency histogram of $U_w - U_l$ for the same 43 experiments. Around 60% of these 43 experiments present a loss of performance below 10%. However in some cases this loss of performance is close to 50%.

This significant loss of performance is due to the new restrictions imposed in the selection of main memory blocks when using basic blocks of instructions instead of cache basic blocks.

TABLE II: Summary statistics for $U_w - U_l$.

Count	43
Average	13,19
Standard deviation	13,69
Minimum	1,31
Maximum	44,83
Stnd. skewness	3,49
Stnd. kurtosis	2,59

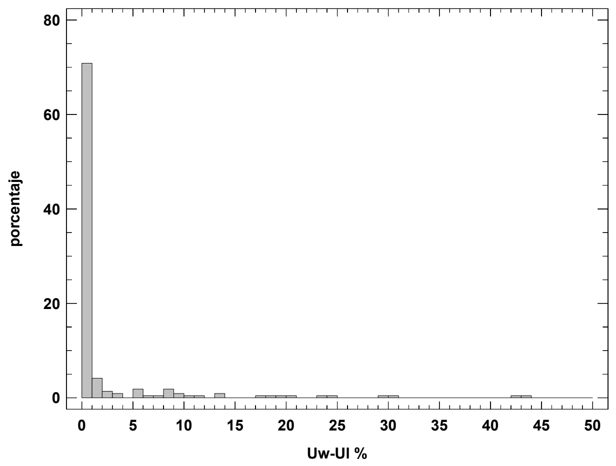

Fig. 3: Frequency histogram for $U_w - U_l$ for all 196 experiments.

Figure 5 shows the box and whisker plot of $U_w - U_l$ in front of the cache size for the 196 experiments. For small cache sizes the loss of performance is greater, and also presents a larger

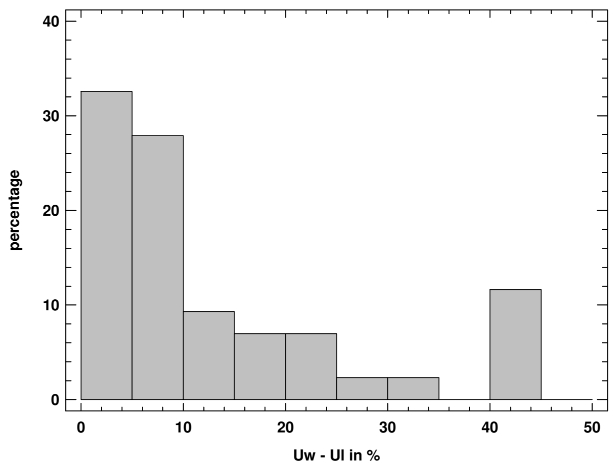

Fig. 4: Frequency histogram for $U_w - U_l$ for experiments with $(U_w - U_l) > 1$.

dispersion. For larger cache sizes the loss of performance is much smaller and presents less dispersion. These results are due to the fact that when the cache size becomes similar or greater than the average size of system tasks, more main memory blocks can be loaded, so the new restrictions described before for the use of CFG are applied less frequently or has lower impact in the selection of blocks. Although the effect of the cahe size is evident, the existence of outliers suggests that other parameters may affect the performance of cache locking when signatures for the watchdog processor are used to control the cache.

These parameters can be related to the size or structure of tasks, or the degree of interference between tasks. Because the data does not follow a normal distribution (see stnd. skewness and stnd. kurtosis values in table II) it is not advisable to carry out an analysys of variance to identify the significant parameters. Anyway, the next target of this research is to develop an algorithm to select which blocks to load and lock in cache, inserting new watchdog signatures when this addition helps to improve performance.

VI. Conclusion

This work presents a new way of implementing the dynamic use of cache locking for fixed-priority, preemptive multitasking real-time systems that combines with a watchdog processor used to gain reliability. The same hardware, the watchdog processor, is used to obtain both reliability and predictability. The proposal is feasible since devices coupling a processor with an FPGA, or FPGAs that incorporate processors inside have been available in the market for a long time. As the FPGA is already included in the same die or package with the processor, no additional hardware is needed.

The main advantage of this proposal is that previous, well known techniques for cache locking can be used. In this work, full dynamic cache locking has been implemented.

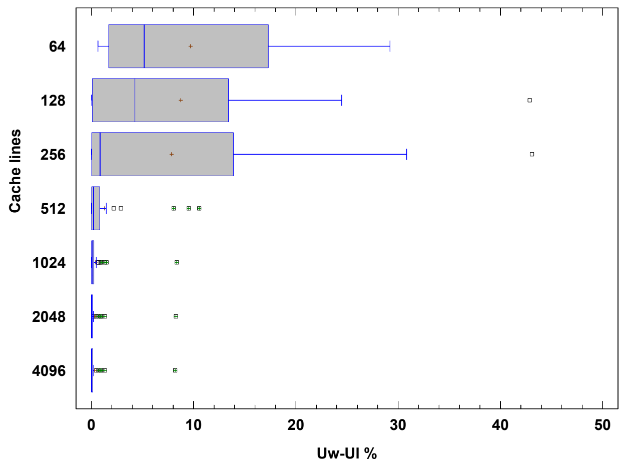

Fig. 5: Scatter plot for $U_w - U_l$ versus cache size in lines (all experiments).

Experiments show that close to 80% of the cases the obtained performance is the same than the performance resulted by implementing cache locking adding new hardware, like a Locking State Memory or a Locking State Generator. This way, predictability can be added to a reliable system without additional hardware cost.

In front of software solutions, that is, using a binary rewriter to insert instructions to lock an unlock the cache, this proposal presents as advantage that the signatures are inserted before blocks are selected and schedulability analysis is accomplished, so no repetition of the analysis has to be done after inserting the instructions to control the cache locking, as must be done with binary rewriters.

Future work is focused on the development of an algorithm to select the main memory blocks to be loaded and locked in cache, evaluating at the same time if the insertion of extra signatures can improve performance.

Acknowledgments

This work was partially funded by the Plan Nacional de I+D, Comisión Interministerial de Ciencia y Tecnología (FEDER-CICYT) under the project HAR2017-85557-P and Agencia Estatal de Investigación under the project DPI2016-80303-C2-1-P.

References

[1] Kapil Anand and Rajeev Barua. Instruction-cache locking for improving embedded systems performance. *ACM Trans. Embed. Comput. Syst.*, 14(3):53:1–53:25, April 2015.

[2] Luis C. Aparicio, Juan Segarra, Clemente Rodríguez, and Víctor Viñals. Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems. *Journal of Systems Architecture*, 57(7):695 – 706, 2011. Special Issue on Worst-Case Execution-Time Analysis.

[3] J.V. Busquets-Mataix, J.J. Serrano, R. Ors, P. Gil, and A. Wellings. Adding instruction cache effect to schedulability analysis of preemptive real-time systems. In *Real-Time Technology and Applications Symposium, 1996. Proceedings., 1996 IEEE*, pages 204–212, jun 1996.

[4] Antonio Martí Campoy, Francisco Rodríguez-Ballester, and Rafael Ors Carot. Lut saving in embedded fpgas for cache locking in real-time systems. *International Journal On Advances in Systems and Measurements*, 6(1&2):190–199, 2013.

[5] Rodríguez F., Campelo J.C., and Serrano J.J. A watchdog processor architecture with minimal performance overhead. In Anderson S., Felici M., , and Bologna S., editors, *Proceedings of the International Conference on Computer Safety, Reliability, and Security SAFECOMP 2002*, volume 2434 of *Lecture Notes in Computer Science*, pages 261–272. Springer, Berlin, Heidelberg, Sept 2002.

[6] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann, 4th edition, 2006.

[7] Chang-Gun Lee, Kwangpo Lee, Joosun Hahn, Yang-Min Seo, Sang Lyul Min, Rhan Ha, Seongsoo Hong, Chang Yun Park, Minsuk Lee, and Chong Sang Kim. Bounding cache-related preemption delay for real-time systems. *IEEE Transactions on Software Engineering*, 27(9):805–826, 2001.

[8] A. Mahmood and E. J. McCluskey. Concurrent error detection using watchdog processors-a survey. *IEEE Transactions on Computers*, 37(2):160–174, Feb 1988.

[9] A. Martí Campoy, A. Perles, F. Rodríguez, and J. V. Busquets-Mataix. Static use of locking caches vs. dynamic use of locking caches for real-time systems. In *Electrical and Computer Engineering, 2003. IEEE CCECE 2003. Canadian Conference on*, volume 2, pages 1283–1286 vol.2, May.

[10] Antonio Martí-Campoy, Francisco Rodríguez-Ballester, Eugenio Tamura Morimitsu, and Rafael Ors. An algorithm for deciding minimal cache sizes in real-time systems. In *Proceedings of the 13th annual conference on Genetic and evolutionary computation*, GECCO '11, pages 1163–1170, New York, NY, USA, 2011. ACM.

[11] Sparsh Mittal. A survey of techniques for cache locking. *ACM Transactions on Design Automation of Electronic Systems*, 21, 05 2016.

[12] Fan Ni, Xiang Long, Han Wan, and Xiaopeng Gao. Combining instruction prefetching with partial cache locking to improve wcet in real-time systems. *PLOS ONE*, 8(12):1–19, 12 2013.

[13] D.A. Patterson and J.L. Hennessy. *Computer Organization and Design: The Hardware/software Interface*. Morgan Kaufmann, 3rd edition, 2005.

[14] Adam Prügel-Bennett. Finite population effects for ranking and tournament selection. *Complex Systems*, 12(2):183–205, 2000.

[15] Isabelle Puaut. Wcet-centric software-controlled instruction caches for hard real-time systems. In *Proceedings of the 18th Euromicro Conference on Real-Time Systems*, pages 217–226, Washington, DC, USA, 2006. IEEE Computer Society.

[16] Jan C. Kleinsorge Sascha Plazar and Peter Marwedel. Wcet-aware static locking of instruction caches. In *Proceedings of the 2012 International Symposium on Code Generation and Optimization*, pages 44–52, 2012.

[17] A.C. Shaw. Reasoning about time in higher-level language software. *Software Engineering, IEEE Transactions on*, 15(7):875–889, July 1989.

[18] E. Tamura, J.V. Busquets-Mataix, and A. Martí Campoy. Towards predictable, high-performance memory hierarchies in fixed-priority preemptive multitasking real-time systems. In *Proceedings of the 15th International Conference on Real-Time and Network Systems (RTNS-2007)*, pages 75–84, 2007.

[19] Yudong Tan and Vincent Mooney. Timing analysis for preemptive multitasking real-time systems with caches. *ACM Trans. Embed. Comput. Syst.*, 6(1), feb 2007.

[20] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, 7(3):36:1–36:53, May 2008.