LUT Saving in Embedded FPGAs for Cache Locking in Real-Time Systems

Antonio Martí Campoy, Francisco Rodríguez-Ballester, and Rafael Ors Carot
Departamento de Informática de Sistemas y Computadores
Universitat Politècnica de València
46022, València, Spain
e-mail: {amarti, prodrig, rors}@disca.upv.es

***Abstract*—In recent years, cache locking have appeared as a solution to ease the schedulability analysis of real-time systems using cache memories maintaining, at the same time, similar performance improvements than regular cache memories. New devices for the embedded market couple a processor and a programmable logic device designed to enhance system flexibility and increase the possibilities of customisation in the field. This arrangement may help to improve the use of cache locking in real-time systems. This work proposes the use of this embedded programmable logic device to implement a logic function that provides the cache controller the information it needs in order to determine if a referenced main memory block has to be loaded and locked into the cache; we have called this circuit a Locking State Generator. Experiments show the requirements in terms of number of hardware resources and a way to reduce them and the circuit complexity. This reduction ranges from 50% up to 80% of the number of hardware resources originally needed to build the Locking State Generator circuit.**

***Keywords*—Real-Time Systems; Cache Locking; FPGA; Memory Hierarchy**

I. INTRODUCTION

In a previous work [1], the authors proposed and evaluated the use of an embedded Field-Programmable Gate Array (FPGA) to implement a lockable cache. The FPGA was used to build a logic circuit that signals to the cache controller if a main memory block should be loaded and locked in cache. This paper extends previous work presenting a way to reduce hardware resources when implementing the logic circuit by means of an FPGA.

Cache memories are an important advance in computer architecture, offering a significant performance improvement. However, in the area of real-time systems, the use of cache memories introduces serious problems regarding predictability. The dynamic and adaptive behaviour of a cache memory reduces the average access time to main memory, but presents a non deterministic fetching time [2]. This way, estimating execution time of tasks is complicated. Furthermore, in preemptive multi-tasking systems, estimating the response time of each task in the system becomes a problem with a solution hard to find due to the interference on the cache contents produced among the tasks. Thus, schedulability analysis requires complicated procedures and/or produces overestimated results.

In recent years, cache locking have appeared as a solution to ease the schedulability analysis of real-time systems using cache memories maintaining, at the same time, similar performance improvements of systems populated with regular cache memories. Several works have been presented to apply cache locking in real-time, multi-task, preemptive systems, both for instructions [3][4][5][6] and data [7]. In this work, we focus on instruction caches only, because 75% of accesses to main memory are to fetch instructions [2].

A locked cache is a cache memory without replacement of contents, or with contents replacement in a priori and well known moments. When and how contents are replaced define different uses of the cache locking mechanism.

One of the ways to use cache locking in preemptive real-time systems is called dynamic use [3]. In the dynamic use cache contents change only when a task starts or resumes its execution. From that moment on cache contents remain unchanged until a new task switch happens. The goal is that every task may use the full size of the cache memory for its own instructions.

The other possible use of cache locking is called static use [8][9]. When a cache is locked in this way the cache contents are pre-loaded on system power up and remain constant while the system runs. For example, a simple software solution may be used to issue processor instructions to explicitly load and lock the cache contents. How the cache contents are pre-loaded is irrelevant; what is important is that the cache behaviour is now completely deterministic. The drawback of this approach is that the cache must be shared among the code of all tasks so the performance improvement is diminished.

This paper focuses on the dynamic use of locked cache and is organized as follows. Section II describes previous implementation proposals for dynamic use of cache locking in real-time systems, and the pursued goals of this proposal to improve previous works. Section III presents a detailed implementation of the Locking State Generator (LSG), a logic function that signals the cache controller whether to load a referenced main memory block in cache or not. Section IV presents some analysis about the complexity of the proposal, and then Section V shows results from experiments carried out to analyse resource requirements in the LSG implementation in terms of number of LUTs (Look-Up Tables) needed to build the circuit. Section VI presents a way to reduce the complexity of the LSG by means of reusing LUTs when implementing the mini-terms of the LSG logic function. Finally, this paper ends with the ongoing work and conclusions.

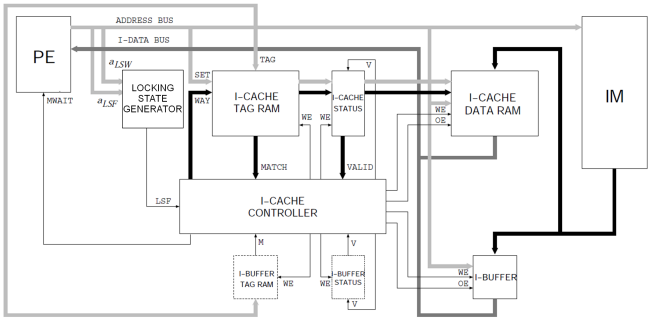

Fig. 1: The LSG architecture.

II. State of the Art

Two ways of implementing dynamic use of cache locking can be found in the bibliography. First of them, [3], uses a software solution, without hardware additions and using processor instructions to explicitly load and lock the cache contents. This way, every time a task switch happens, the operating system scheduler runs a loop to read, load and lock the selected set of main memory blocks into the cache memory for the next task to run. The list of main memory blocks selected to be loaded and locked in cache is stored in main memory.

The main drawback of this approach is the long time needed to execute the loop, which needs several main memory accesses for each block to be loaded and locked.

In order to improve the performance of the dynamic use of cache locking, a Locking State Memory (LSM) is introduced in [4]. This is a hardware solution where the locking of memory blocks in cache is controlled by a one-bit signal coming from a specialized memory added to the system. When a task switch happens, the scheduler simply flushes the cache contents and a new task starts execution, fetching instructions from main memory. But not all referenced blocks during task execution are loaded in cache, only those blocks selected to be loaded and locked are loaded in cache. In order to indicate whether a block has to be loaded or not the LSM stores one bit per main memory block. When the cache controller fetches a block of instructions from main memory, the LSM provides the corresponding bit to the cache controller. The bit is set to one to indicate that the block has to be loaded and locked in cache, and the cache controller stores this block in cache. If the bit is set to zero, indicates that the block was not selected to be loaded and locked in cache, so the cache controller will preclude the store of this block in cache, thus change in cache contents are under the control of the LSM contents and therefore under the control of system designer.

The main advantage of the LSM architecture is the reduction of the time needed to reload the cache contents after a preemption compared against the previous, software solution.

The main drawback of the LSM is its poor scalability. The size of the LSM is directly proportional to main memory and cache-line sizes (one bit per each main memory block, where the main memory block size is equal to the cache line size). This size is irrespective of the size of the tasks, or the number of memory blocks selected to be loaded and locked into the cache. Moreover, the LSM size is not related to the cache size. This way, if the system has a small cache and a very large main memory, a large LSM will be necessary to select only a tiny fraction of main memory blocks.

In this work, a new hardware solution is proposed, where novel devices found in the market are used. These devices couples a standard processor with an FPGA, a programmable logic device designed to enhance system flexibility and increase the possibilities of customisation in the field. A logic function implemented by means of this FPGA substitutes the work previously performed by the LSM. For the solution presented here hardware complexity is proportional to the size of system, both software-size and hardware-size. Not only the circuit required to dynamically lock the cache contents may be reduced but also those parts of the FPGA not used for the control of the locked cache may be used for other purposes. We have called this logic function a Locking State Generator (LSG) and think our proposal simplifies and adds flexibility to the implementation of a real-time system with cache locking.

III. The proposal: Locking State Generator

Recent devices for the embedded market [10][11] couple a processor and an FPGA, a programmable logic device designed to enhance system flexibility and increase the possibilities of customisation in the field. This FPGA is coupled to an embedded processor in a single package (like the Intel's Atom E6x5C series [10]) or even in a single die (like the Xilinx's Zynq-7000 series [11]) and may help to improve the use of cache locking in real-time systems.

Deciding whether a main memory block has to be loaded in cache is the result of a logic function with the memory address bits as its input. This work proposes the substitution of the Locking State Memory from previous works by a logic function implemented by means of this processor-coupled FPGA; we have called this element a Locking State Generator (LSG).

Two are the main advantages of using a logic function instead of the LSM. First, the LSG may adjust its complexity and circuit-related size to both the hardware and software characteristics of the system. While the LSM size depends only on the main memory and cache-line sizes, the number of circuit elements needed to implement the LSG depends on the number of tasks and their sizes, possibly helping to reduce hardware. Second, the LSM needs to add a new memory and data-bus lines to the computer structure. Although LSM bits could be added directly to main memory, voiding the requirement for a separate memory, in a similar way as extra bits are added to ECC DRAM, the LSM still requires modifications to main memory and its interface with the processor. In front of that the LSG uses a hardware that is now included in the processor package/die. Regarding modifications to the cache controller, both LSM and LSG present the same requirements as both require that the cache controller accepts an incoming bit to determine whether a referenced memory block has to be loaded and locked into the cache or not.

Figure 1 shows the proposed architecture, similar to the LSM architecture, with the LSG logic function replacing the work of the LSM memory.

A. Implementing logic functions with an FPGA

An FPGA implements a logic function combining a number of small blocks called logic cells. Each logic cell consists of a Look-Up Table (LUT) to create combinational functions, a carry-chain for arithmetic operations and a flip-flop for storage. The look-up table stores the value of the implemented logic function for each input combination, and a multiplexer inside the LUT is used to provide one of these values; the logic function is implemented simply connecting its inputs as the selection inputs of this multiplexer.

Several LUTs may be combined to create large logic functions, functions with input arity larger than the size of a single LUT. This is a classical way of implementing logic functions, but it is not a good option for the LSG: the total number of bits stored in the set of combined LUTs would be the same as the number of bits stored in the original LSM proposal, just distributing the storage among the LUTs.

1) Implementing mini-terms: In order to reduce the number of logic cells required to implement the LSG, instead of using the LUTs in a conventional way this work proposes to implement the LSG logic function as the sum of its mini-terms (the sum of the input combinations giving a result of 1).

This strategy is not used for regular logic functions because the number of logic cells required for the implementation heavily depends on the logic function itself, and may be even larger than with the classical implementation. However, the arity of the LSG is quite large (the number of inputs is the number of memory address bits) and the number of cases giving a result of one is very small compared with the total number of cases, so the LSG is a perfect candidate for this implementation strategy.

A mini-term is the logic conjunction (AND) of the input variables. As a logic function, this AND may be built using the LUTs of the FPGA. In this case, the look-up table will store a set of zero values and a unique one value. This one value is stored at position j in order to implement mini-term j. Figure 2 shows an example for mini-term 5 for a function of arity 3, with input variables called C, B and A, where A is the lowest significant input.

For the following discussion we will use 6-input LUTs, as this is the size of the LUTs found in [11]. Combining LUTs to create a large mini-term is quite easy; an example of a 32-input mini-term is depicted in Figure 3 using a two-level associative network of LUTs. Each LUT of the first level (on the left side) implements a 1/6 part of the mini-term (as described in the previous section). At the second level (on the right side), a LUT implements the AND function to complete the associative property.

2) Sum of mini-terms: For now, we have used 7 LUTs to implement one mini-term. To implement the LSG function we have to sum all mini-terms that belong to the function; a mini-term k belongs to a given logic function if the output of the function is one for the input case k. In this regard,

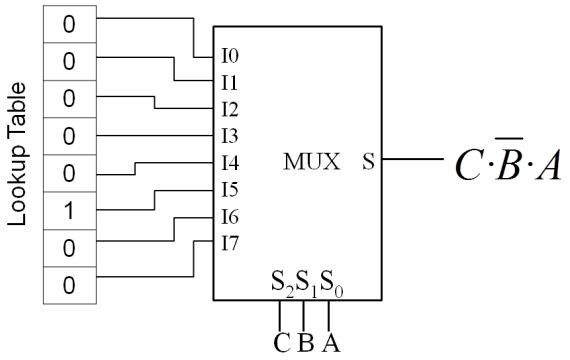

Fig. 2: Implementing mini-term 5 of arity 3 (C, B, A are the function inputs).

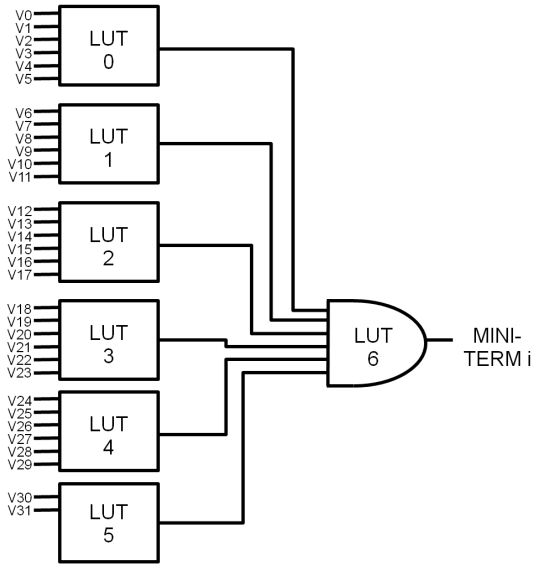

Fig. 3: Implementing a 32-input mini-term using 6-input LUTs.

two questions arise: first, how many mini-terms belong to the function, and second, how to obtain the logic sum of all of them.

The first question is related to the software parameters of the real-time system we are dealing with. If the real-time system comprises only one task, the maximum number of main memory blocks that can be selected to load and lock in cache is the number of cache lines (L). If the real-time system is comprised of N tasks this value is $L \times N$ because, in the dynamic use of cache locking, each task can use the whole cache for its own instructions.

A typical L1 instruction cache size in a modern processor is 32KB; assuming each cache line contains four instructions and that each instructions is 4B in size, we get $L = (32KB/4B)/4$ instructions $= 2K$ lines.

This means that, for every task in the system, the maximum number of main memory blocks that can be selected is around

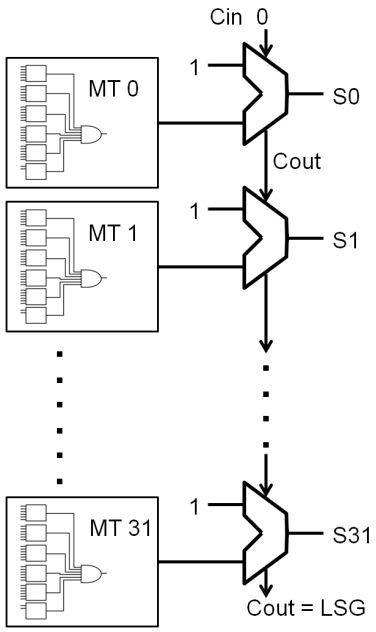

Fig. 4: Implementing the LSG function.

2000. Supposing a real-time system with ten tasks, we get a total maximum of 20 000 selectable main memory blocks. That is, the LSG function will have 20 000 mini-terms. Summing all these mini-terms by means of a network of LUTs to implement the logic OR function with 20 000 inputs would require around 4000 additional LUTs in an associative network of 6 levels.

The solution to reduce the complexity of this part of the LSG is to use the carry chain included in the logic cells for arithmetic operations. Instead of a logic sum of the mini-terms, an arithmetic sum is performed: if a binary number in which each bit position is the result of one of the mini-terms is added with the maximum possible value (a binary sequence consisting of ones), the result will be: i) the maximum possible value and the final carry will be set to zero (if the outputs of all mini-terms are zero for the memory address used as input to the LSG), or ii) the result will be $M-1$ and the final carry will be set to one (being $M > 0$ the number of mini-terms producing a one for the memory address). Strictly speaking, mini-terms are mutually exclusive, so one is the maximum value for M. In the end, the arithmetic output of the sum is of no use, but the final carry indicates if the referenced main memory block has to be loaded and locked in cache. Figure 4 shows a block diagram of this sum applied to an example of 32 mini-terms, each one nominated MTk.

Using the carry chain included into the LUTs which are already used to calculate the LSG function mini-terms produce a very compact design. However, a carry chain adder of 20 000 bits (one bit per mini-term) is impractical, both for performance and routing reasons. In order to maintain a compact design with a fast response time, a combination of LUTs and carry-chains are used, as described below.

First, the 20 000 bits adder is split into chunks of reasonable size; initial experiments carried out indicate this size to be between 40 and 60 bits in the worst case, resulting into a set of 500 to 330 chunks. All these chunk calculations are performed in parallel using the carry chains included into the same logic cells used to calculate the mini-terms, each one providing a carry out. These carries have to be logically or-ed together to obtain the final result. A set of 85 to 55 6-input LUTs working in parallel combine these 330 to 500 carries, whose outputs are arithmetically added with the maximum value using the same strategy again, in this case using a single carry chain. The carry out of this carry chain is the LSG function result.

IV. Evaluation of the LSG

The use of the LSG to lock a cache memory is a flexible mechanism to balance performance and predictability as it may have different modes of operation. For real-time systems, where predictability is of utmost importance, the LSG may work as described here; for those systems with no temporal restrictions, where performance is premium, the LSG may be easily forced to generate a fixed one value, obtaining the same cache behaviour with a locked cache than with a regular cache. It can even be used in those systems mixing real-time and non real-time tasks, as the LSG may select the proper memory blocks for the former in order to make the tasks execution predictable and provide a fixed one for the latter to improve their performance as with a regular cache memory.

Initial experiments show timing is not a problem for the LSG as its response time has to be on par with the relatively slow main memory: the locking information is not needed before the instructions from main memory. Total depth of the LSG function is three LUTs and two carry chains; register elements are included into the LSG design to split across several clock cycles the calculations in order to increase the circuit operating frequency and to accommodate the latency of main memory as the LSG has to provide the locking information no later the instructions from main memory arrive. Specifically, the carry out of all carry chains are registered in order to increase the operating frequency.

Regarding the circuit complexity, the following calculations apply: although the address bus is 32 bits wide, the LSG, like the cache memory, works with memory blocks. Usually a memory block contains four instructions and each instruction is 4B, so main memory blocks addresses are actually 28 bits wide.

Generating a mini-term with a number of inputs between 25 to 30 requires 6 LUTs in a two-level network. Supposing a typical cache memory with 2000 lines, 12 000 LUTs are required. But if the real-time system has ten tasks, the number of LUTs needed for the LSG grows up to 120 000. It is a large number, but more LUTs may be found on some devices currently available [11]. Calculating the logic OR function of all these mini-terms in a classical way adds 4000 more LUTs to the circuit, but the described strategy merging LUTs and carry chains reduce this number to no more than 500 LUTs in the worst case.

The estimated value of 120 000 LUTs required to build the LSG function is an upper bound, and there are some ways this

TABLE I: Cache sizes used in experiments

	Size (lines)	Size (instructions)	Size (bytes)
1	64	256	1K
2	128	512	2K
3	256	1K	4K
4	512	2K	8K
5	1024	4K	16K
6	2048	8K	32K
7	4096	16K	64K

number may be reduced. A real-time system with five tasks will need just half this value of LUTs. The same is true if the cache size is divided by two. Following sections show some experiments and a easy way to reduce the total number of LUTs.

V. EXPERIMENTS

Previous sections have detailed, in a theoretical way, an upper bound of the number of LUTs required to implement the LSG. Experiments conducted in this section provide more realistic values, and identify both hardware and software characteristics that affect the number of required LUTs in order to implement the LSG for a particular system.

Regarding hardware characteristics the size of cache memory, measured in lines, is the main parameter because this number of lines is the maximum number of blocks a task may select to load and lock in cache. And, in a first approach, every block selected to be locked needs a mini-term in the LSG implementation in order to identify it when it is fetched by the processor.

As described previously, it is not possible to build a mini-term with only one LUT, because the number of inputs of the latter, ranging from 4 up to 7 inputs [12] in today devices is not enough to accommodate the inputs of the former.

Mini-terms are then implemented combining several LUTs. Thus the number of inputs of LUTs is also a main characteristic, because the lower the number of LUT inputs, the higher the number of LUTs needed to build a mini-term. Finally, width of address bus (measured in bits) is also a parameter to be taken into account, because the number of variables in a mini-term is the number of lines in the address bus.

Regarding software parameters, the number of tasks in the system presents the larger impact in the number of needed LUTs. In dynamic use of cache locking, irrespective of the use of software reload, LSM or the here proposed LSG, every task in the system may select as many blocks to load and lock in cache as cache lines are available. So, the number of LUTs needed to build the LSG circuit will be a multiple of the number of system tasks.

Other software parameters like size, periods or structure of tasks do not affect the number of LUTs needed, or their effect is negligible.

In order to evaluate the effect of these characteristics, and to obtain realistic values about the number of required LUTs, experiments described below have been accomplished.

TABLE II: Main characteristics of systems used in experiments

System	Number of tasks	Task average size (blocks)
1	4	849
2	5	158
3	4	429
4	4	641
5	5	424
6	3	855
7	8	205
8	3	1226
9	5	617
10	3	1200
11	3	476
12	3	792

Hardware architecture and software systems are the same, or a subset of those described and used in [3][13].

The hardware architecture is based on the well-known MIPS R2000 architecture, added with a direct-mapping instruction cache memory (i-cache). The data size of this i-cache range from 64 up to 4096 lines. The size of one cache line is the same as one main memory block, and it is 16B (four instructions of four bytes each). Seven cache sizes have been used, as described in Table I. Although MIPS R2000 address bus is 32 bits wide, it has been reduced to 16 bits in the following experiments, giving a maximum size of 64KB of code.

Regarding the number of LUT inputs, four cases have been studied: LUTs with 4, 5, 6, and 7 input variables.

Regarding the software used in experiments, tasks are artificially created to stress the cache locking mechanism. Main parameters of tasks are defined, such as the number of loops and their nesting level, the size of the task, the size of its loops, the number of if-then-else structures and their respective sizes. A simple tool is used to create such tasks. The workload of any task may be a single loop, if-then-else structures, nested loops, streamline code, or any mix of these. The size of a task code may be large (close to the 64KB limit) or short (less than 1KB). 12 different sets of tasks are defined, and with these sets a total of 24 real-time systems have been created modifying the periods of the tasks. Task periods are hand-defined to make the system schedulable, and the task deadlines are set to be equal to the task period. Finally, the priority is assigned using a Rate Monotonic policy (the shorter the period the higher the priority). Table II shows the main characteristics of the systems used for this experimentation.

Using cache locking requires a careful selection of those instructions to be loaded and locked in cache. It is possible to make a random selection of instructions: that would provide predictability to the temporal behaviour of system, but there would be no warranty about system performance. Several algorithms have been proposed to select cache contents [14].

For this work, a genetic algorithm is used. The target of the genetic algorithm is to find the set of main memory blocks that, loaded and locked in cache, provides the lower utilisation for the whole system. In order to achieve this objective, the genetic algorithm gets as inputs the number of tasks, their periods and temporal expressions [15] that are needed to calculate Worst Case Execution Time and Worst Case Response Time.

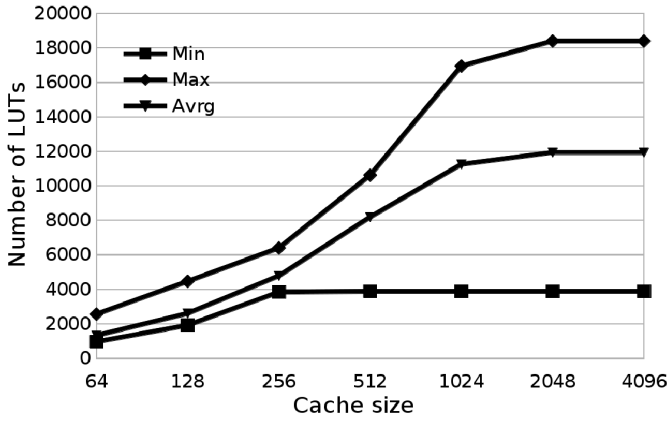

Fig. 5: Number of 4-inputs LUTs required.

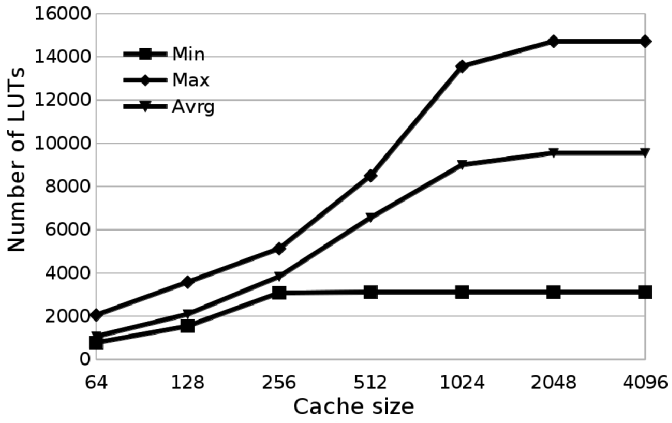

Fig. 6: Number of 5-inputs LUTs required.

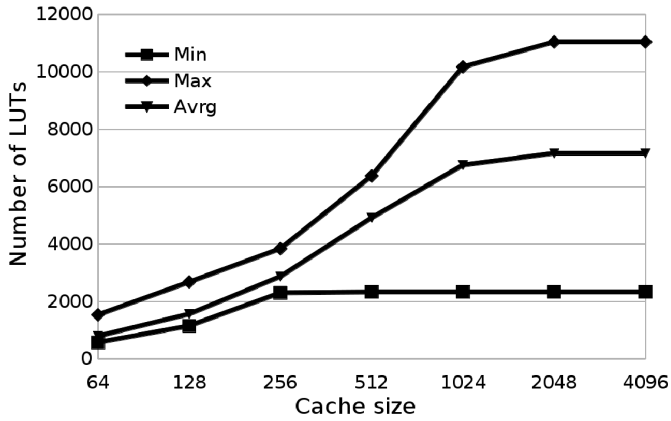

Fig. 7: Number of 6-inputs LUTs required.

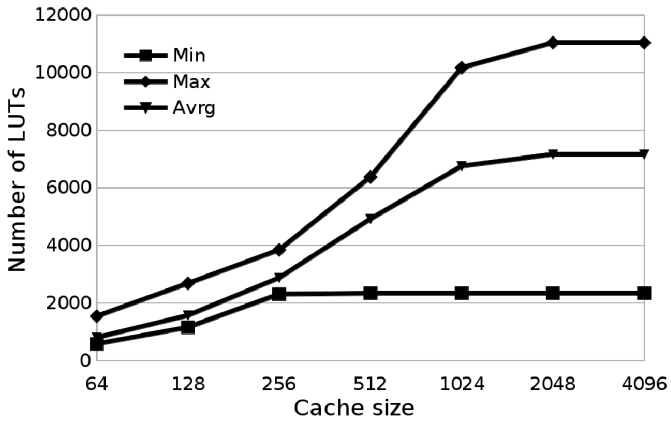

Fig. 8: Number of 7-inputs LUTs required.

Also the cache parameters like line size, total cache size, mapping policy and hit and miss times are inputs to the genetic algorithm.

The solution found by the genetic algorithm, that is, the set of main memory blocks selected for each task in the system, has to meet cache requirements like cache size and mapping policy. As output the genetic algorithm determines if the system is schedulable, the worst case execution time and worst response time for all the tasks, and the list of selected main memory blocks for each task to be loaded and locked in the cache. This list of blocks has been used in this work to calculate the number of required LUTs to implement the LSG.

Figure 5 shows the number of LUTs needed to build the mini-terms of each one of the 24 systems, as a function of the cache size using 4-input LUTs. Graph shows the maximum value, the minimum value, and the average number of LUTs for the 24 systems. Figures 6, 7, and 8 show the same information than Figure 5, using LUTs of 5, 6, and 7 inputs, respectively.

The four figures are identical in shape and tendency, but present some differences in their values. As expected the most noticeable is the effect of cache size. There is a clear and positive relationship between the cache size and the number of required LUTs. And regarding average values, this increment is very close to a lineal increase.

But there are two exceptions, both for the same reason. For the curve of minimum values, it presents a zero slope when cache size is larger than 256 lines. This is because the tasks in set 2 have a size lower than 256 main memory blocks (in average, size of tasks is 158; see Table II), but none of the tasks is larger than 256 blocks. This means that for each task, the genetic algorithm will select no more than 158 blocks, so, no matter the cache size, a maximum of 158 blocks multiplied by 5 (number of tasks in this system) will be selected and, thus, implemented as mini-terms.

Since the largest task in all systems is close to 2000 blocks, when the cache reaches a size of 2048 lines or larger, it

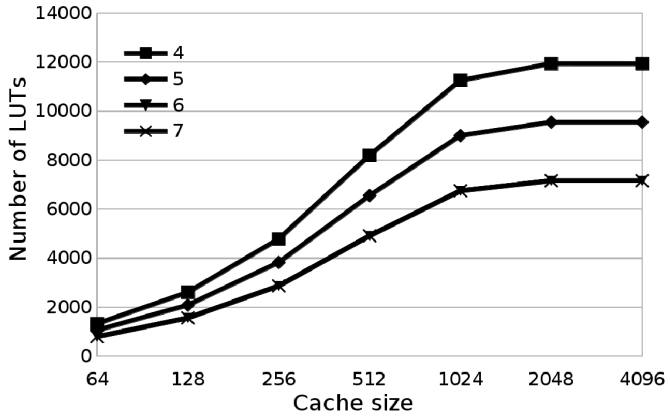

Fig. 9: Average LUTs required for LUTs of 4, 5, 6, and 7 inputs.

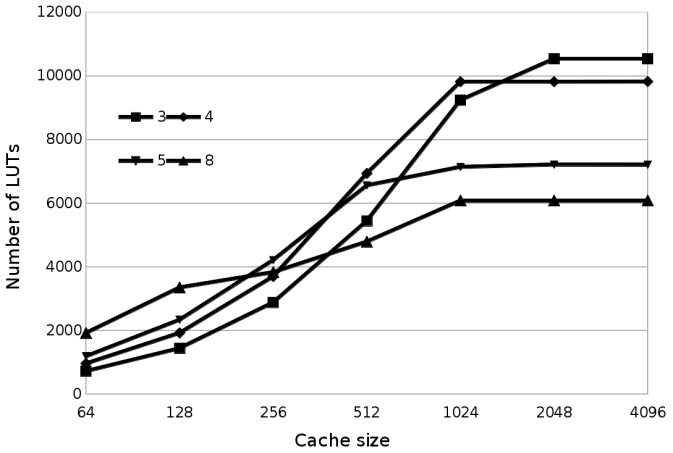

Fig. 10: Average LUTs required for number of system tasks (3, 4, 5 and 8 tasks).

does not affect the number of LUTs needed, because the number of blocks selected, and thus the number of mini-terms to be implemented, cannot be larger. Numerical differences between maximum and minimum values maybe explained by differences in tasks structures or genetic algorithm executions, but most probably differences come from the number of tasks in each system. However, the effect of cache size and the existence of tasks with sizes smaller than the largest caches prevent to clearly state this idea. Regarding the effect of the number of LUT inputs, there are significant differences in the number of needed LUTs to implement the LSG when using LUTs of 4, 5, 6, and 7 inputs.

This effect is more important as cache size increases. For small cache sizes, the difference in the number of LUTs related to the number of LUT inputs is about some hundreds. But for large cache sizes, this difference is around five thousand LUTs. This effect is better appreciated in Figure 9, where average of needed LUTs for all systems and total number of LUT sizes is shown.

Figure 10 shows the average number of LUTs needed to implement the LSG, in front of cache size and number of tasks, for the 24 systems analysed. This figure shows that both cache size and number of tasks are important characteristics regarding the number of LUTs needed, but no one is more important than the other. When the cache size is small, and thus individual task sizes are larger than the cache size, the number of tasks in the system becomes a significant parameter regarding the number of needed LUTs, as shown for cache sizes of 64, 128, and 256 lines. However, when the cache becomes larger, the effect of the number of tasks seems to be the inverse. This is not completely true. Curves arrange in inverse order for small cache sizes than for large cache sizes, but this is because all systems must fit into the limit of 64KB of code, so systems with more tasks have smaller tasks while systems with fewer tasks have larger tasks. The conclusion is that the most important factor is neither the cache size nor the number of tasks, but the relationship between cache size and size of the tasks in the system. This factor, called System Size Ratio (SSR), was identified as one of the main factors deciding cache locking performance in [16].

VI. Reducing complexity

There is a way of reducing LSG circuit complexity without affecting the number of tasks in the system or the cache size. This simplification comes from the way each mini-term is implemented. As explained before, the number of inputs of a LUT is not enough to implement a whole mini-term, so the associative property is used to decompose the mini-term in smaller parts, each implemented using a LUT that are then combined using again a LUT performing the function of an AND gate, as shown in Figure 3.

As an example, consider two mini-terms of six variables implemented with 3-input LUTs. In order to implement the two mini-terms each one is decomposed in two parts, and each part is implemented by a LUT, using four LUTs to build what may be called half-mini-terms. Finally, two LUTs implementing an independent AND logic function each are used to combine these parts to finally implement both mini-terms. Consider now that both mini-terms have one of its part equal. In this case, implementing the same half-mini-term twice it is not mandatory, because the output of a LUT may be routed to two different AND gates, so mini-terms with some parts equal may share the implementation of that part. Figure 11 shows an example with two mini-terms sharing one of their parts.

Profiting from the limited number of inputs of the LUTs previous experiments have been repeated, but in this case an exhaustive search have been carried out to count the number of mini-terms that share some of their parts. The number of parts a mini-term is divided into depends on the number of LUT inputs, and four sizes have been used like in previous experiments: 4, 5, 6, and 7 inputs. This way, and considering a 16 bits address bus, a mini-term may be divided in three or four parts. In some cases, some inputs of some LUTs will

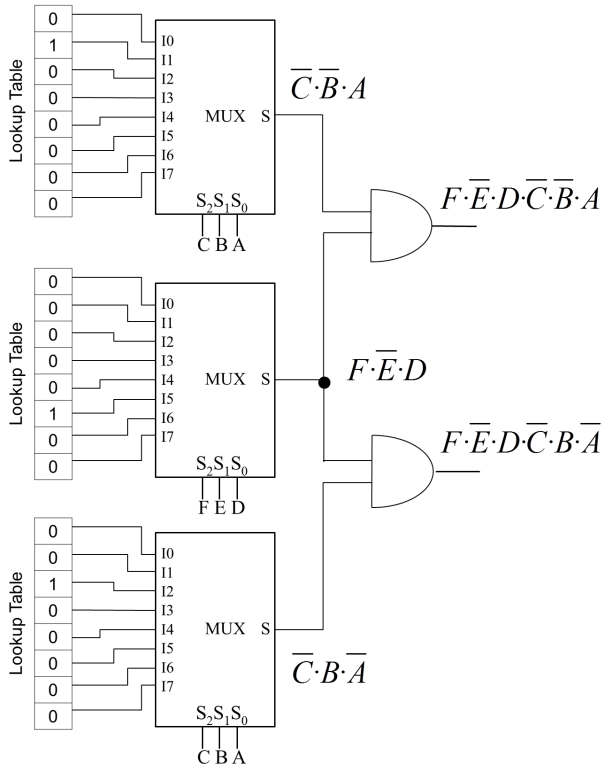

Fig. 11: Example of reducing LUTs needed to implement mini-terms.

not be used, being the worst case when using 7-inputs LUTs, because each mini-term requires 3 LUTs so there are 21 inputs available to implement mini-terms of arity 16.

A simple algorithm divides mini-terms in parts as a function of LUT size, and detects common parts between mini-terms. This exhaustive search is performed for the whole system, that is, it is not applied to mini-terms of the selected blocks of individual tasks but applied for all selected blocks of all tasks in the system.

Figure 12 shows the number of LUTs needed to build the mini-terms of each one of the 24 systems as a function of the cache size and using LUTs with 4 inputs, after applying the algorithm to search and reduce the LUTs needed due to the fact the implementation of common parts may be shared by the corresponding mini-terms. Graph shows the maximum value, the minimum value, and the average number of LUTs for the 24 systems. Figures 13, 14, and 15 show the same information than Figure 12 when the number of LUT inputs are 5, 6, and 7, respectively.

Figures 12, 13, 14, and 15 all present the same shape and the same values for minimum, average, and maximum curves, respectively. Numerical values show differences, but they are not significant, so it can be said that the number of LUT inputs does not affect the number of needed LUTs to implement the LSG when shared LUTs implementing common parts of mini-terms are used to reduce the total number of LUTs needed.

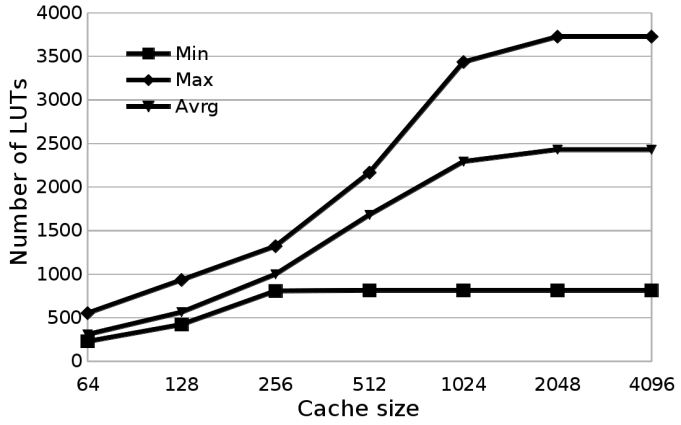

Fig. 12: Number of 4-inputs LUTs required after reduction.

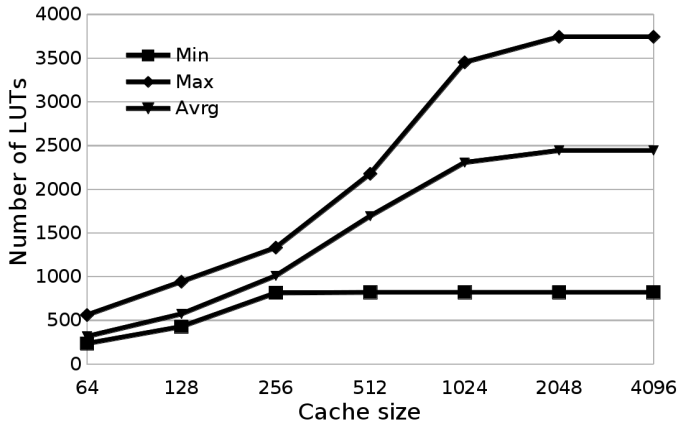

Fig. 13: Number of 5-inputs LUTs required after reduction.

This can be explained because when the number of LUT inputs is small the probability to find common parts among mini-terms increases. Figure 16 shows the average values of needed LUTs after reduction for the four LUT sizes considered. No significant differences appears in this graph. In front of average values for non-reduced implementation of the LSG, the number of required LUTs is between a 50% and a 80% when reducing the LSG implementation using common parts of mini-terms.

Regarding saving LUTs, shapes and tendencies of figures 12 to 15 are very similar to those in figures 5 to 8, so the effect of cache size, number of tasks in the system, and other parameters (except the LUT size) are similar for non-reduced and reduced implementation of the LSG.

Figure 17 shows the percentage of reduction in the number of LUTs regarding cache size and LUT size. The minimum reduction is 55% for a 64 lines cache size and 6-input LUTs, and a maximum reduction is close to 80% for 4-inputs LUTs and a cache of 512 lines or larger. The effect of cache size over reduction is more acute when using LUTs with six and

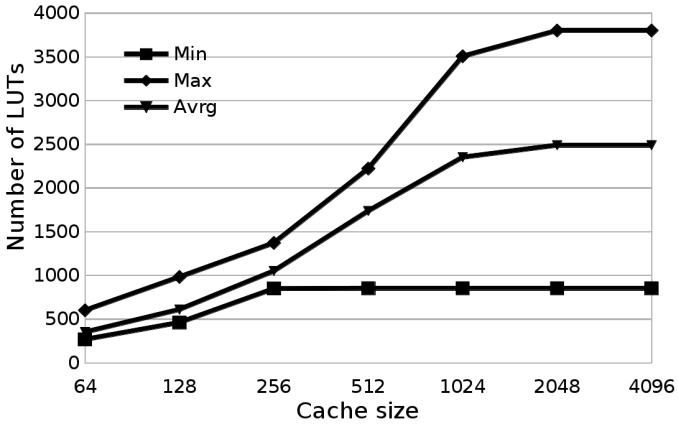

Fig. 14: Number of 6-inputs LUTs required after reduction.

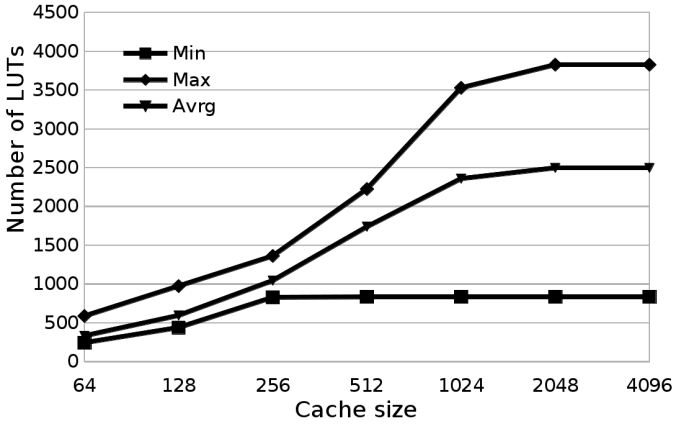

Fig. 15: Number of 7-inputs LUTs required after reduction.

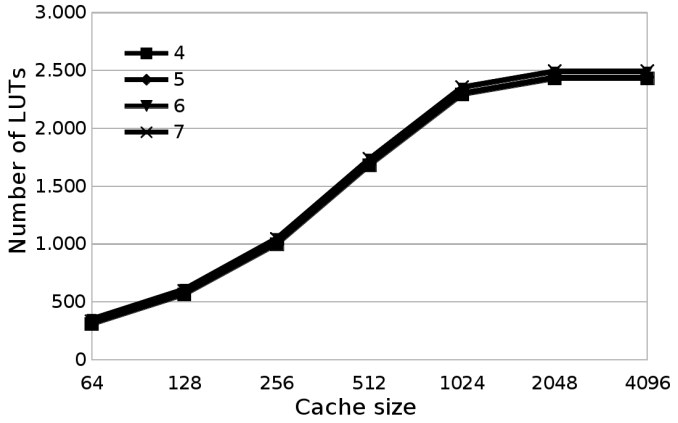

Fig. 16: Average LUTs required after reduction for LUTs of 4, 5, 6, and 7 inputs.

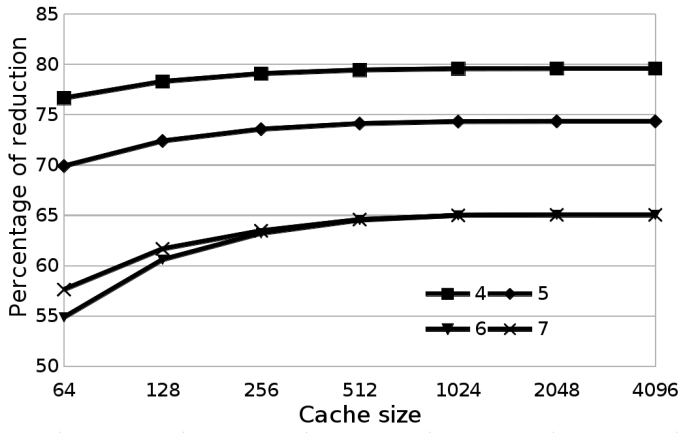

Fig. 17: Percentage of reduction as a function of cache size and number of LUT inputs.

seven inputs than when four and five inputs are used. However, the main effect over the percentage of reduction comes, as stated before, from the size of the LUTs. In absolute values, from the worst case of 14 000 LUTs needed to build the LSG (maximum for a cache size of 1024 lines in Figure 5), the reduced implementation of LSG lowers this number to 3500 (maximum for a cache size of 1024 lines in Figure 12).

VII. Ongoing work

The previous simplification may be improved by the selection algorithm, e.g., the genetic algorithm used to determine the main memory blocks that have to be loaded and locked into the cache. Usually, the goal of these algorithms is to provide predictable execution times and an overall performance improvement of the system and its schedulability, for example reducing global system utilisation or enlarging the slack of tasks to allow scheduling non-critical tasks. However, new algorithms may be developed that take into account not only this main goals, but that also that try to select blocks with common parts in their mini-terms, enhancing LUT reusing and reducing the complexity of the final LSG circuit. This is more than just wish or hope: for example, considering a loop with a sequence of forty machine instructions —10 main-memory blocks— the resulting performance is the same if the selected blocks are the five first ones or the last five ones, or even if the selected blocks are alternate blocks. Previous research show that genetic algorithms applied to this problem may produce different solutions, that is, different sets of selected main memory blocks, all with the same results regarding performance and predictability. So, next step in this research is the development of a selection algorithm that simultaneously tries to improve system performance and reduce the LSG circuit complexity.

What is performance and circuit complexity need to be carefully defined in order to include both goals in the selection algorithm. Once the algorithm works, the evaluation of

implementation complexity will be accomplished.

VIII. Conclusion

This work presents a new way of implementing the dynamic use of a dynamically locked cache for preemptive, real-time systems. The proposal benefits from recent devices coupling a processor with an FPGA, a programmable logic device, allowing the implementation of a logic function to signal the cache controller whether to load a main memory block in cache or not. This logic function is called a Locking State Generator (LSG) and replaces the work performed by the Locking State Memory (LSM) in previous proposals.

As the FPGA is already included in the same die or package of the processor, no additional hardware is needed as in the case of the LSM. Also, regarding circuit complexity, the LSG adapts better to the actual system as its complexity is related to both hardware and software characteristics of the system, an advantage in front of the LSM architecture, where the LSM size depends on the size of main memory exclusively. Results from experiments state than final LSG complexity is mainly related to cache size, and not main memory size as LSM is.

Implementation details described in this work show that it is possible to build the LSG logic function with commercial hardware actually found in the market.

Moreover, a way to reduce hardware requirements by means of reusing LUTs has been developed and experimented. Sharing LUTs among mini-terms allows a reduction in the number of LUTs needed to implement the LSG between 50% and 80%, and makes negligible the effect of LUT size over the number of LUTs needed.

Ongoing research steps about the selection algorithm of main memory blocks in order to reduce circuit complexity.

Acknowledgments

This work has been partially supported by PAID-06-11/2055 of Universitat Politècnica de València and TIN2011-28435-C03-01 of Ministerio de Ciencia e Innovación.

References

[1] A. M. Campoy, F. Rodríguez-Ballester, and R. Ors, "Using embedded fpga for cache locking in real-time systems," in *Proceedings of The Second International Conference on Advanced Communications and Computation*, INFOCOMP 2012, pp. 26–30, Oct 2012.

[2] J. L. Hennessy and D. A. Patterson, *Computer Architecture: A Quantitative Approach, 4th Edition*. Morgan Kaufmann, 4 ed., 2006.

[3] A. M. Campoy, A. P. Ivars, and J. V. B. Mataix, "Dynamic use of locking caches in multitask, preemptive real-time systems," in *Proceedings of the 15th World Congress of the International Federation of Automatic Control*, 2002.

[4] J. B.-M. E. Tamura and A. M. 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)*, pp. 75–84, 2007.

[5] J. C. K. Sascha Plazar and P. Marwedel, "Wcet-aware static locking of instruction caches," in *Proceedings of the 2012 International Symposium on Code Generation and Optimization*, pp. 44–52, 2012.

[6] L. C. Aparicio, J. Segarra, C. Rodríguez, and V. Vials, "Improving the wcet computation in the presence of a lockable instruction cache in multitasking real-time systems," *Journal of Systems Architecture*, vol. 57, no. 7, pp. 695 – 706, 2011. Special Issue on Worst-Case Execution-Time Analysis.

[7] X. Vera, B. Lisper, and J. Xue, "Data cache locking for tight timing calculations," *ACM Trans. Embed. Comput. Syst.*, vol. 7, pp. 4:1–4:38, Dec. 2007.

[8] M. Campoy, A. P. Ivars, and J. Busquets-Mataix, "Static use of locking caches in multitask preemptive real-time systems," in *Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium)*, IEEE, 2001.

[9] I. Puaut and D. Decotigny, "Low-complexity algorithms for static cache locking in multitasking hard real-time systems," in *Real-Time Systems Symposium, 2002. RTSS 2002. 23rd IEEE*, pp. 114–123, IEEE, 2002.

[10] I. Corp., "Intel atom processor e6x5c series-based platform for embedded computing." http://download.intel.com/embedded/processors/prodbrief/324535.pdf, 2013. [Online; accessed 15-March-2013].

[11] X. Inc., "Zynq-7000 extensible processing platform." http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm, 2012. [Online; accessed 15-March-2013].

[12] M. Kumm, K. Mller, and P. Zipf:, "Partial lut size analysis in distributed arithmetic fir filters on fpgas," in *Proceedings of the IEEE International Symposium on Circuits and Systems*, 2013.

[13] A. M. Campoy, F. Rodríguez-Ballester, R. Ors, and J. Serrano, "Saving cache memory using a locking cache in real-time systems," in *Proceedings of the 2009 International Conference on Computer Design*, pp. 184–189, jul 2009.

[14] A. M. Campoy, I. Puaut, A. P. Ivars, and J. V. B. Mataix, "Cache contents selection for statically-locked instruction caches: An algorithm comparison," in *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, (Washington, DC, USA), pp. 49–56, IEEE Computer Society, 2005.

[15] A. Shaw, "Reasoning about time in higher-level language software," *Software Engineering, IEEE Transactions on*, vol. 15, pp. 875–889, July 1989.

[16] 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*, vol. 2, pp. 1283–1286 vol.2, May.