Using Embedded FPGA for Cache Locking in Real-Time Systems

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

***Abstract*—In recent years, locking caches 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 locking caches in real-time systems. This work propose the use of this embedded programmable logic device to implement a logic function that provides the locking 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.**

***Keywords*-Real-Time Systems; Locking Caches; FPGA.**

I. Introduction

Cache memories are an important advance in computer architecture, giving 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 behavior of a cache memory reduces the average access time to main memory, but presents a non deterministic fetching time [5]. This way, estimating execution time of tasks is complicated. Furthermore in preemptive, multi-tasking systems, estimating the response time of every 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, locking caches 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 has been presented to apply locking caches in real-time, multi-task, preemptive systems, both for instructions [1][4][7] and data [8]. In this work, we focus on instruction caches only, because 75% of accesses to main memory are to fetch instructions [5].

A locking 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 locking cache memory.

One of the ways to use locking caches in preemptive real-time systems is called the dynamic use. In this way of using a locking cache, cache contents change only when a task starts or resumes its execution. Then, 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.

This paper is organized as follows. Section two describes previous implementation proposals for the dynamic use of a locking cache in real-time systems, and the pursued goals of this proposal to improve previous works. Section three presents a detailed implementation of the Locking State Generator (LSG), a logic function that signals to the cache controller whether to load a main memory block in cache. Section four presents some analysis about the complexity of the proposal, and Section five outlines some ideas about how to simplify the circuit complexity. Finally, this paper ends with the ongoing work and conclusions.

II. State of the Art

Two ways of implementing dynamic use of locking cache can be found in the bibliography. First of them, [1], 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 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 load and lock 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 locking cache, in [4] is introduced the Locking State Memory (LSM). This is a hardware solution where the loading of memory blocks in cache is controlled by a one-bit signal coming from a 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 are loaded in cache; only those blocks selected to be loaded and locked are loaded in cache. In order to

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-226-4

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 locking cache controller. If the bit is set to 1, indicates 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 0, 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 cache contents remain unchanged.

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 size and cache-line size (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. This way, if the system has a small locking cache and a very big 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 (Field-Programmable Gate Array), 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, however this time hardware complexity is proportional to the size of system, both software-size and hardware-size. Not only the circuit required to dynamically use the locking cache may be reduced but also those parts of the FPGA not used for the control of the locking 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 locking cache.

III. THE PROPOSAL: LOCKING STATE GENERATOR

Recent devices for the embedded market [3][6] couple a processor and an FPGA (Field-Programmable Gate Array), 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 [3]) or even in a single die (like the Xilinx's Zynq-7000 series [6]) and may help to improve the use of locking caches 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.

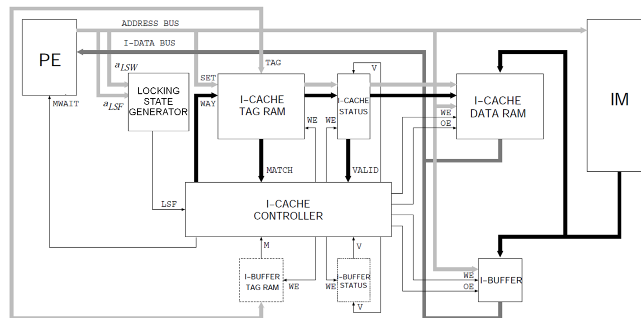

Figure 1. The LSG architecture.

This work proposes the substitution of the Locking State Memory 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 generator instead of the LSM. First, the LSG may adjust its complexity and circuit-related size to both the hardware and software characteristics. While the LSM size depends only on the main memory size and cache-line size, 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.

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 for 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,

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-226-4

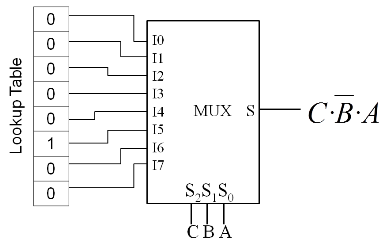

Figure 2. Implementing mini-term 5 of arity 3 (C, B, A are the function inputs).

but it is not a good option for the LSG: the total number of bits stored in the set of combined LUTs is 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 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 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 1 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 Lookup table will store a set of zero values and a unique one value. This one will be stored in the 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 [6]. 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, two questions arise: first, how many mini-terms belong to the function, and second, how to obtain the logic sum of all

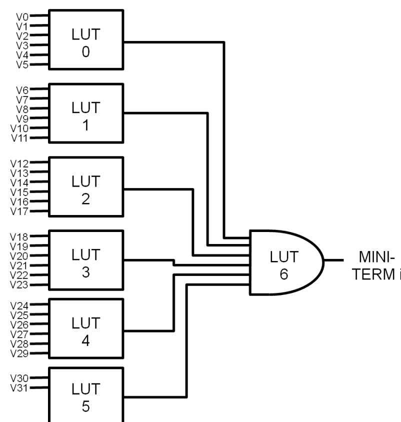

Figure 3. Implementing a 32-input mini-term using 6-input LUTs.

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 a locking cache, each task can use the whole cache for its own blocks.

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 $L = (32KB/4B)/4instructions = 2K$ lines.

This means that, for every task in the system, the maximum number of main-memory blocks that can be selected is around 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 all mini-terms are zero), or ii) the result will be $M-1$ and the final carry will be set to one (being M 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, and the final carry indicates if the referenced main memory block has to be loaded and

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-226-4

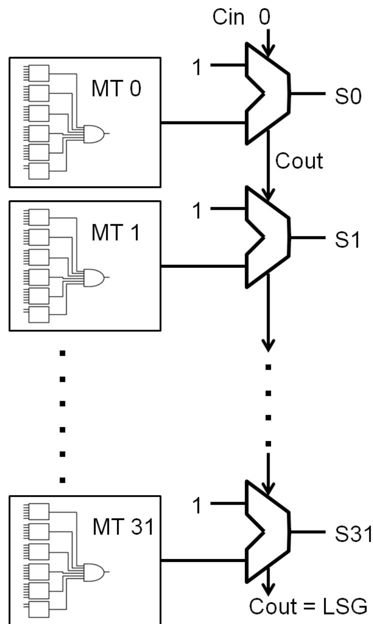

Figure 4. Implementing the LSG function.

locked in cache. Figure 4 shows a block diagram of this sum.

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 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 with a locking 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 forced to generate a fixed one value, obtaining a system with the same behavior as 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 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 32 bits, so main-memory blocks addresses are 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 [6]. 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.

V. Reducing complexity

The estimated value of 120 000 LUTs required to build the LSG function is an upper bound, and there are some ways this number may be reduced. A real-time system with five tasks will need just half this value of LUTs. Same if the cache size is divided by two.

In some cases, not all tasks will use the whole cache, that is, the number of selected blocks for a given task may be lower than the cache capacity, reducing the number of mini-terms in the LSG. In this aspect, the LSG improves the LSM because it better adapts to hardware and software characteristics of the system. Finally, as with any logic function implementation, there are well-known simplification algorithms that may be applied, reducing both the number of terms and their size (arity), which in turn reduce the number of LUTs required for the implementation.

This simplification may be improved by the selection algorithm. To use a locking cache, no matter the way it is

used and how locking information is stored or generated, an off-line algorithm has to select those main memory blocks that will be loaded and locked in cache [2]. Usually, the target of these algorithms is to provide predictable execution times and improve the overall performance of the system and its schedulability, for example reducing global utilisation or enlarging the slack of tasks to allow scheduling non-critical tasks. But, new algorithms may be designed that take into account not only this main target, but also trying to select blocks with adjacent addresses, enhancing simplification and reducing 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— selecting the five first blocks will give the same performance than selecting the last five, or selecting alternate blocks. Previous research show that genetic algorithms applied to this problem produce different solutions, that is, different sets of selected main memory blocks but with the same results regarding performance and predictability.

This is a first approach to a new architecture, and many experiments are needed to precisely evaluate the complexity and cost of the LSG implementation, and to state the scenarios where its use is more suitable than using LSM. Number of LUTs detailed in this work are for the worst case, that is, real-time systems with many tasks, large cache memory and many main memory blocks selected to lock in cache. Not in all cases the upper bound of LUTs will be reached.

VI. Ongoing work

Next step 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, evaluation of implementation complexity will be accomplished.

Also, some design strategies have to be explored in detail in order to reduce the number of LUTs required to implement the LSG. In particular initial experiments from a design strategy merging LUTs from pairs of mini-terms show promising results as the number of bits to be added by the carry chains may be cut in half without a serious impact on the circuit operating frequency.

VII. Conclusion

This work presented a new way of implementing the dynamic use of locking cache for preemptive real-time systems. The proposal benefits from recent devices coupling a processor with a 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. 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 with 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 exclusively on the size of main memory.

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, ongoing research steps about the selection algorithm of main memory blocks and the LSG hardware implementation are outlined.

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. Marti Campoy, A. Perles Ivars, and J. V. Busquets 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.

[2] Antonio Marti Campoy, Isabelle Puaut, Angel Perles Ivars, and Jose Vicente Busquets Mataix. Cache contents selection for statically-locked instruction caches: An algorithm comparison. In *Proceedings of the 17th Euromicro Conference on Real-Time Systems*, pages 49–56, Washington, DC, USA, 2005. IEEE Computer Society.

[3] Intel Corp. Intel atom processor e6x5c series. http://www.intel.com/p/en_US/embedded/hwsw/hardware/atom-e6x5c/overview, 2012. [Online; accessed 16-June-2012].

[4] J.V. Busquets-Mataix E. Tamura and A. Mart 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.

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

[6] Xilinx Inc. Zynq-7000 extensible processing platform. http://www.xilinx.com/products/silicon-devices/epp/zynq-7000/index.htm, 2012. [Online; accessed 16-June-2012].

[7] 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.

[8] Xavier Vera, Björn Lisper, and Jingling Xue. Data cache locking for tight timing calculations. *ACM Trans. Embed. Comput. Syst.*, 7(1):4:1–4:38, December 2007.

Copyright (c) IARIA, 2012. ISBN: 978-1-61208-226-4