Proceedings of the 16th
Euromicro Conference on
Real-Time Systems Work in
Progress. Pages 1-4. Catania,
Italy, June 2004.

High Performance Memory Architectures with Dynamic Locking Cache for Real-Time Systems

E. Tamura[1], F. Rodríguez[2], J.V. Busquets-Mataix[2], A. Martí Campoy[2]
[1] *Grupo de Automática y Robótica, Pontificia Universidad Javeriana - Cali, Cali, Colombia*
eutamo@doctor.upv.es
[2] *Departamento de Informática de Sistemas y Computadores, Universidad Politécnica de Valencia, Camino de Vera s/n, 46022 Valencia, España*
{prodrig, vbusque,amarti}@disca.upv.es

Abstract

In modern computers, memory hierarchies play a paramount role in improving the average execution time. However, this fact is not so important in real-time systems, where the worst-case execution time is what matters the most. System designers must use complex analyses to guarantee that all tasks meet their deadlines. As an alternative to making those complex analyses, it is proposed to build a memory hierarchy such that it provides high performance coalesced with high predictability. At the same time, the memory assist should imply small-scale modifications in the hardware. The solution is to be centred on instruction fetching since it represents the highest number of memory accesses.

1. Introduction

Cache memories are extensively used to reduce the increasing speed gap between processor and main memory because they provide lower average execution times by minimising the number of accesses to main memory. However, their use in real-time systems arises hard problems in calculating the Worst Case Execution Time (WCET) because cache memories present a dynamic, adaptive and non-predictable behaviour. This lack of determinism precludes the use of well-known schedulability analysis techniques in order to validate the temporal correctness of the system. Several methods and algorithms have been proposed to model the cache behaviour and include it in the WCET [1], and to take into account cache effects in the schedulability analysis [2, 3]. As an alternative to cache modelling, other cache schemas have been proposed to simplify the analysis, like the use of locking caches [4]. Locking cache presents several advantages:

- Implementation of locking cache is feasible and presents no construction complexity
- Locking cache offers a fully predictable behaviour, allowing the use of simple and well-known techniques for schedulability analyses
- Performance of locking cache is similar to that of conventional, non-predictable caches

Two ways of using locking caches have been proposed: static use of locking caches [5] and dynamic use of locking caches [6].

In static use of locking caches, cache contents, determined via the use of a genetic algorithm, are loaded and locked during system start-up, and those contents will never change during system operation. Since the static use of locking cache is fully predictable, it is easy to analyse its behaviour.

On the other hand, in the dynamic use of locking caches, cache contents are modified during system operation in a controlled way. Every time a task begins or resumes execution, cache is flushed and then reloaded with a set of instructions belonging to the new scheduled task. This set of instructions is always the same for each task, so dynamic use of locking cache presents not just a high degree of predictability, but at the same time, it improves the performance of static use of locking cache [7] because each task may use all of the space available in the whole cache for its own instructions, while in static use the cache space is shared by all of the tasks.

Furthermore, in more than 210 experiments with assorted caches –varying parameters like cache size, line size, degree of associativity, miss and hit times-- and several sets of tasks, the dynamic use of locking caches provided the same or better performance than conventional caches in about 60% of the tests [6]; in some cases, however, performance falls significantly. This loss of performance is due to the cost of loading and locking the cache contents every time a task begins or resumes its execution. The cost of loading and locking one memory block in cache is about five times

the time needed to transfer a block from main memory to cache in a conventional cache, because the load is accomplished trough the execution of a small routine, which is included in the scheduler.

This paper explores several memory architectures in order to load and lock the cache via hardware mechanisms, reducing so the time needed to load each block. To load and lock memory blocks automatically demands modifications both in main memory organization as well as in cache memory.

2. Rationale

The system operation of dynamic use of locking cache as proposed in [6] is the following:

Every time the operating system schedules a new task, a small routine is executed. This routine reads from main memory the predefined set of addresses whose contents must be loaded in cache, and then loads the blocks associated to them; each block has the same size as the cache line size. After all of the blocks have been loaded, the locking cache is locked and the task begins/resumes execution. Both load and lock operations are accomplished through cache-management instructions.

This way of operation is inefficient mainly due to two reasons: first, several accesses to main memory are required for each block of main memory that must be transferred to cache, imposing a significant overhead. Second, the new task can not be dispatched immediately after it is ready to execute, suffering a significant latency because all of its to-be-locked blocks must be loaded before it begins/resumes execution.

In order to improve the performance of dynamically locking cache, two basic requirements must be satisfied:

a) Main memory blocks must be transferred to cache when fetched by the processor. That is, instructions will be loaded and locked in cache, as the control flow of the program requires them without invoking any piece of extra code.
b) The latter implies that the cache memory controller must be able to identify the instructions to be locked in an automatic way. Blocks –belonging to any task- to be locked in cache must be marked before the system begins execution, during system design or system start-up.

Should the previous requirements hold, task execution will begin as soon as possible since there is no penalty involved by load and lock instructions, and there are no delays involved in identifying the blocks to be locked.

3. Cache memory requirements

Several commercial processors include locking caches in their memory hierarchies, but to the authors knowledge there is no one adequate enough to achieve the characteristics needed to get a predictable and high performance cache schema. The most interesting is the IDT 79RC64574 family of standalone processors, that include a two-way set associative cache with a locking mechanism which can be enabled/disabled on a per-line basis. This is all that is needed for static use of locking cache, where the instructions locked in cache will never change. In the dynamic use of locking cache however, every time a new task is scheduled to run, cache contents must be reloaded from main memory. With a locking cache like the one provided in the IDT, it would be necessary to execute a small loop that writes the tag of each cache line, hence delaying task execution. So, in order to get the maximum performance, the following characteristics are proposed for a locking cache:

- Cache should be locked in a per-line basis. That is, the system designer should be able to select the main memory block to be loaded in each cache line. Albeit, the whole cache must be locked to guarantee predictability.
- Every cache line includes an extra bit named *Lock State Flag* (henceforth, *LSF*), to signal whether the cache line is locked or not. If any cache line is locked, its content will not be replaced; otherwise, its behaviour is the same as that of a non-locking cache.
- The LSF is automatically adjusted when the processor fetches the block since the value of the LSF is somehow embedded in the instruction stream read from main memory.
- By executing a processor instruction the LSF is cleared for all the cache lines simultaneously, unlocking the entire cache in one operation. This is the only mechanism available to a programmer to clear the LSFs.
- There exists a temporal buffer –or prefetch queue-- with size equal to one cache line and same behaviour than a conventional cache. This buffer improves execution time of non-locked instructions by taking into account the spatial locality of non-locked memory blocks.

When the scheduler dispatches a new task or resumes the execution of a pre-empted task, it only needs to clear the LSF of every cache line. After this, the cache is loaded and locked as instructions are fetched; the only penalty experienced is one miss in cache for every sequence of L instructions that were selected to be locked, where L is the cache line size.

The complexity of the described behaviour for the proposed locking cache is simpler than that provided by current, high-performance, commercial processors, so implementation issues are not expected. Nonetheless, in order to have the desired behaviour without compromising the operation time (which should remain

identical to the one provided by a conventional cache) the design of the locking cache memory must be performed very carefully.

4. Main memory requirements

The success of the proposed use of dynamic locking cache relies also on the main memory ability to embed information related to whether lock or not its contents. The system designer must add the value of the LSFs somehow in the tasks instructions. Embedding this flag is the major issue of the proposed schema. Following, several alternatives are described:

1. Embed the LSF in the instruction op code. This proposal poses many problems, since the designer has to face the need to modify the instruction set repertory, and hence, the processor decoding stage. Furthermore, sometimes it is not possible to accomplish this in a simple manner, i.e. by just rearranging the bits for example; in those cases where every combination of bits is already used, it is mandatory to increase the processor instruction word size, which might lead in turn, to wider data buses. Additionally, it also requires modifying the compiler back-end.
2. The next approach is more software-oriented. Since the blocks to lock are known beforehand, it is feasible to embed some sort of header at the beginning of the binary image with the corresponding map of blocks that should be loaded and locked for every task: if a task occupies m blocks of main memory, it requires a map of size m bits, plus some delimiter to mark the beginning and end of each map; if a bit is set, it means that the block must be locked into the cache and not loaded into cache otherwise. The main advantage of this proposal is that it requires a minimum amount of storage. Its main disadvantage is that every time the processor fetches an instruction, the cache memory controller needs to access the map, which represents a delay. This delay, however, would be shorter if the map is stored in some sort of look-aside memory; in case the whole memory map does not fit into the look-aside memory, this schema may introduce significant lack of determinism and increase the complexity of schedulability analysis.
   In addition, the implementation of this alternative would require modifications in the compiler back end, the linker, and the loader.
3. Increase the memory size word by one bit, which will store the *Instruction Lock State Flag*, *ILSF*. Each ILSF is adjusted when the program is loaded into main memory; if it is set, it means that the corresponding instruction must be locked. Whenever the processor fetches an instruction, the ILSF is also copied to the instruction cache, so there is no penalty involved in execution time.
   The main disadvantage is that the data memory bus between the main memory and cache memory has to be one bit wider; also, the compiler, linker, and loader require modifications. Besides, notice that space is wasted since in this schema, each instruction has a corresponding bit, but just one bit is required by every memory block; hence $m\ (\ L - 1\ )$ bits are wasted, where L is the number of instructions per main memory block, and m is the number of blocks.
4. Use a memory organization following the layout proposed in the Stanford's TORCH [8]. In this architecture, groups of eight instructions are preceded by eight extension bytes that provide information about dynamic instruction scheduling.
   In a similar vein, it is possible to group together some instructions into a *parcel*. There are two possibilities in terms of gathering together the instructions: a) the number of memory blocks in a parcel is constant and b) it is feasible to have parcels of varying size. In any case, each parcel is preceded by one *Locking State Word*, *LSW*, which contains the locking state information for every memory block in the parcel. Each bit in the LSW determines the state of one memory block. The number of blocks per parcel cannot exceed the instruction word size, w.
   In this approach the main disadvantage is that every time that there is a cache miss, the worst-case penalty will be equal to two main memory accesses, one to read the instructions and one more to read the concerning LSW. Furthermore, in case b), calculating the LSW address requires an extra access to a table to know the size of the current parcel. In both cases, the drawback could be alleviated by caching the LSW into the cache controller but doing so would require a more detailed WCET analysis to get tighter results.
   On the processor side, every datum located at an address corresponding to an LSW should be considered as a no-operation instruction. In addition, the compiler back end and maybe the linker, have to suffer considerable changes to patch the resulting binary image.
   Last, but not least, whenever the amount of extra information required per instruction is substantial, the TORCH approach has its merits; yet in this case, for each main memory block just one bit is required.
5. This approach stems from a combination of the previous two and provides an easy to model architecture, space efficiency, and no delays. Furthermore, it does not require any modification in the processing element, just in the memory system.
   In this approach, an extra, dedicated memory, the *Locking State Memory (LSM)*, is added to the memory subsystem. Its depth should be the same as

the number of memory blocks in the main memory and its width may be one. However, in order to use off-the-shelf 8-bit wide memories, the information for eight blocks (comprised in 8 LSFs) will be stored in one LSW. Hence, given a main memory of depth $d_{MM} = mL$, where m is the number of blocks, the required LSM has depth, d_{LSM}, equal to $m / 8$.

Let b_I be the number of bytes per instruction, and L the cache line size (then each memory block has L instructions). Each parcel has eight memory blocks, so an LSW has information for eight memory blocks.

Then the number of instructions, I, that corresponds to each LSW is given by $I = 8L$. The address of any LSW, a_{LSW}, is such that $a_{LSW} \bmod I = 0$.

Now, every time that an instruction, Ir, at address a_r is referenced, the cache memory controller, at the same time, has to access the memory system and the LSM, to check the LSW at address a_{LSWr}, which is the address of the LSW that corresponds to m_r, the memory block that stores Ir. a_{LSWr} is obtained by stripping off the $Log_2\ Ib_I$ least significant bits of a_r.

Finally, it is necessary to extract the corresponding LSF within the LSW to determine whether to load and lock the memory block or not; it is given by the *3* bits to the left of the $Log_2\ Lb_I$ bits of a_r.

In order to keep the same operation time provided by a conventional cache system, it is necessary to add an extra line to the cache memory data bus to carry the locking state information. On the other hand, only one 8-way multiplexer and some decoding stage is needed to address the LSM.

5. Conclusions and Future Work

Automating the locking process in the dynamic use of the locking cache offers, in an intuitive way, faster execution times. Unfortunately it is not so easy to provide the necessary mechanisms to embed the required information about locking states into the program code since many components, both on the hardware and on the software areas might be involved: on one side, the locking cache structure, the main memory organization, the data bus width, and even the processor itself; on the other hand, the compiler, the linker and the loader.

The goal to pursue in the design of the memory system is then to incorporate this information without increasing the memory requirements in terms of storage efficiency, or slowing down the general operation of the memory hierarchy. Furthermore, the processor itself should not have significant modifications in its architecture.

The last approach illustrated seems very promising in trying to adhere to the previous requirements. The resulting memory system has to be diligently tested and verified by means of thorough analyses and simulations and at the end, by its implementation on an FPGA. Besides, it is necessary to evaluate the improvements and the amount of resources involved in order to ponder the cost-benefit of the proposed solution.

6. References

[1] F. Mueller, “Timing Analysis for Instruction Caches”, *Real-Time Systems Journal*, 18(2-3), Kluwer Academic Publishers, Boston, USA, May 2000, pp. 217-247.

[2] 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 *Proceedings of the 1996 Real-Time Technology and Applications Symposium*, IEEE Computer Society, Boston, USA, June 1996, pp. 204-213.

[3] C.G. Lee, J. Hahn, Y.M. Seo, S.L. Min, R. Ha, S. Hong, C. Y. Park, M. Lee, and C. S. Kim, “Analysis of Cache-Related Preemption Delay in Fixed-Priority Preemptive Scheduling”, *IEEE Transactions on Computers*, 47(6), IEEE Computer Society, Los Alamitos, USA, May 2000, pp. 217-247.

[4] A. Martí, A. Perles, and J.V. Busquets-Mataix, “Using Locking Caches in Preemptive Real-Time Systems”, In *Proceedings of the 12th Real-Time Congress on Nuclear and Plasma Sciences*, IEEE Computer Society, Valencia, Spain. June 2001, pp. 157-159.

[5] A. Martí, A. Perles, and J. V. Busquets-Mataix. "Static Use of Locking Caches in Multitask Preemptive Real-Time Systems" In *IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the 22nd IEEE Real-Time Systems Symposium)*, London, UK, December 2001.

[6] A. Martí, A. Perles, 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*, Elsevier Science, Barcelona, Spain. July 2002.

[7] A. Martí, S. Sáez, A. Perles, and J.V. Busquets-Mataix, "Performance Comparison of Locking Caches under Static and Dynamic Schedulers", In *Proceedings of the 27th IFAC/IFIP/IEEE Workshop on Real-Time Programming, IFAC/IFIP/IEEE*, Lagow, Poland, May 2003.

[8] M. Smith, M. Horowitz, and M. Lam, "Efficient Superscalar Performance Through Boosting", In *Proceedings of the 5th International Conference on Architectural Support for Programming languages and Operating Systems*, ACM/IEEE, Boston, USA, October 1992, pp. 248-259.

This work is supported by the *Comisión Interministerial de Ciencia y Tecnología* under project CICYT DPI 2003-08320-C02-01