Hardware Synchronization for Embedded Multi-Core Processors

Christian Stoif
Institute of Computer Technology
Vienna University of Technology
stoif@ict.tuwien.ac.at

Martin Schoeberl
Department of Informatics and
Mathematical Modeling
Technical University of Denmark
masca@imm.dtu.dk

Benito Liccardi
BMW Group
Munich, Germany
benito.liccardi@bmw.de

Jan Haase
Institute of Computer Technology
Vienna University of Technology
Vienna, Austria
haase@ict.tuwien.ac.at

***Abstract*— Multi-core processors are about to conquer embedded systems — it is not the question of whether they are coming but how the architectures of the microcontrollers should look with respect to the strict requirements in the field. We present the step from one to multiple cores in this paper, establishing coherence and consistency for different types of shared memory by hardware means. Also support for point-to-point synchronization between the processor cores is realized implementing different hardware barriers. The practical examinations focus on the logical first step from single- to dual-core systems, using an FPGA-development board with two hard PowerPC processor cores. Best- and worst-case results, together with intensive benchmarking of all synchronization primitives implemented, show the expected superiority of the hardware solutions. It is also shown that dual-ported memory outperforms single-ported memory if the multiple cores use inherent parallelism by locking shared memory more intelligently using an address-sensitive method.**

I. Introduction

The following quote from [1] indicates that dealing with parallelism is mandatory for any field of information technology:

If researchers meet the parallel challenge, the future of IT is rosy. If they don't it's not.

Despite the fact that exploiting parallelism has a long history in computer science it is still uncommon in the field of embedded systems. Electronic embedded architectures face a continuous increase in functionality which requires additional memory and computational power. Due to the stringent environmental conditions to be fulfilled, increasing the core frequency of the single-core embedded processor cores is much more limited than in other fields. As the road maps of leading semiconductor companies denote, multi-core alternatives for embedded applications are about to be introduced [2].

The advent of parallelism is a renowned topic in the field of computer science and there are enough examples that show how parallelism — in all its undeniable benefit — introduces new kinds of problems which, unfortunately, are nontrivial in the majority. However, with ever increasing miniaturization, introducing parallelism is a natural next step in the evolution of any microprocessor architecture (e.g. the UltraSPARCI and the dual-core 64b UltraSPARC [3]).

Conventional parallel architectures have to be judged with respect to their applicability in the field of embedded systems. Since the issue here concerns also safety-critical systems (especially automotive and aeronautics), cost will or should not be the sole criterion in evaluating promising solutions.

Changing from a single to multiple processor cores is not without pitfalls and requires prudence. *Synchronization* is the main topic that must be addressed. *Data synchronization* prevents data from being invalidated by parallel access whereas *event synchronization* coordinates concurrent execution. One common mechanism to achieve data synchronization is a *lock*.

Event synchronization forces processes to join at a certain point of execution. *Barriers* can be used to separate distinct phases of computation and are normally implemented without special hardware using locks and shared memory [4]. An involved process *enters* the barrier, *waits* for the other processes and then all processes *leave* the barrier together.

Waiting can be of type *busy-waiting* or *blocking*, whereas locking by busy-waiting is not a preferred locking technique. This common belief is challenged recently [5], but not regarding embedded but database systems. In [6] synchronization primitives are analyzed regarding the amount of energy consumption of busy-waiting vs. blocking methods.

In this paper blocking hardware solutions ensuring synchronization for a multiple number of processor cores are presented and compared to pure spinning software solutions.

II. Related Work

Locks are implemented in hardware in the CRAY X-MP [7]: a limited set of *lock registers* is shared by the processors and are assigned to certain processes by the operating system.

In [8] the architecture of an early RISC-based multiprocessor is described. Each processor has a fixed number of channels to send data to the other processors, some bytes can be sent on a channel without blocking the sending processor. Here support of the compiler is needed to coordinate the execution of the processes on the different processors.

The synchronization primitives locks, barriers and lock-free data structures are the focus of attention in [9]. The classical implementations of those primitives are compared against hybrid synchronization primitives that use hardware support and the caches to improve efficiency and scalability, yielding promising results that seem to justify hardware acceleration.

In the specialized multi-core architecture described in [10] a DSP-, RISC- and VLIW-core are connected by a 64-bit AMBA AHB bus. For fast synchronization each pair of the three cores share dual-ported memory on-chip. Caching is not done for the on-chip but for the off-chip memory (SD-RAM). The work in [10] shows some relevance regarding the hardware configurations used and described in this paper.

In [11] an analysis of how to provide an efficient synchronization by barriers on a shared memory multiprocessor with a shared multi-access bus interconnection is described. An innovative, perhaps unorthodox, alternative to ordinary barriers is given in [12]: the waiting of a thread is forced by continuous invalidation of the respective instruction cache.

An example of global event synchronization across parallel processors using a barrier support library is given in [13], a compiler is needed to produce the parallelized binary code. Besides the performance overhead due to waiting, barriers also have a significant power consumption [14] as disadvantage.

Different barrier implementations for many-core architectures are analyzed in terms of efficiency and scalability in [15], proving that the scaling behavior of actual hardware implementations can differ to the expected scaling behavior.

Directly related to the topic of this paper is the work done in [16], the on-chip global synchronization unit presented there shares ideas with the work presented in this paper. In [16] this synchronization unit is only simulated, in our paper we actually implemented such an on-chip synchronization aid in hardware too. On the other hand in [17] hardware implementations of basic synchronization mechanisms are described. This was done here for comparison reasons as well, previously to designing and implementing the synchronization unit which is the main part of this work. The mechanisms realized in [17] are building on vendor-specific bus systems, lacking the advantage of direct on-chip synchronization realized here.

Similar system architectures based on FPGAs are discussed in [18] with respect to accelerating data processing.

III. Hardware Synchronization

A hardware-environment based on hard-wired processor cores and on-chip shared memory is the fundament for the implementation and practical verification of the concepts presented in this paper. The main focus is on how to achieve reliable communication between the processor cores using the on-chip shared memory.

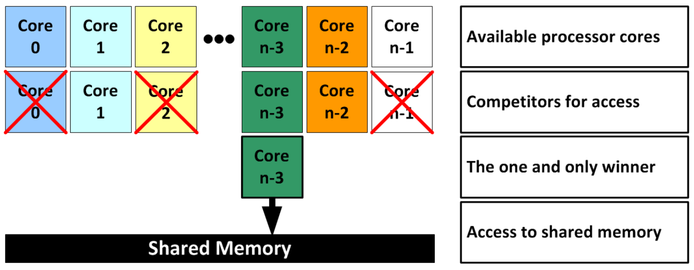

Fig. 1. Scheme of an efficient race for access

A. Problem

Synchronization between the arbitrary number of cores in their access to the shared memory is necessary. An efficient mechanism to resolve arbitrary concurrent requests for our critical resource should fulfill the following demands:

- Efficiency:
  - only cores actually competing for access attend the race and can become its winner
  - the race itself must not consume much time
    ⇒ ideally the race elects one winner per cycle
- Fairness:
  - no competitor waits indefinitely to get access
  - each competitor is served in a finite time
    ⇒ ideally the worst-case waiting time is bounded only by the number of processor cores

All requirements are met with the synchronization mechanism developed and described in the following.

B. Concept

In order to fulfill the efficiency and fairness demands a synchronization mechanism has been developed. A simple round robin scheme cycling all available processor cores would require minimal resources for implementation but would be very inefficient when only a few cores want to access the shared memory. As shown in Fig. 1 only the cores which really need access are considered for it by our mechanism. Processor cores that have to wait are blocked until it is their turn. The worst case occurs in the situation when all the available cores want access to the shared memory simultaneously, resulting in different waiting times. In order to avoid any processor core to be favored or discriminated a *dynamic priority scheme* is used to choose the access-order.

A *global locking* scheme making not just arbitrary *single* but also *multiple* accesses to the shared memory atomic is present as well. Global locking is offered by a global locking bit that is shared by all processor cores in the system.

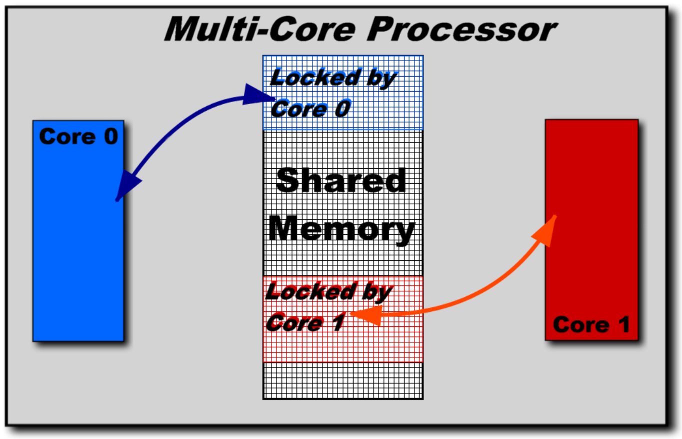

Fig. 2. Parallel access to different memory-regions by address-sensitivity

Locking the whole shared memory when accessing essentially only a small area of the memory is not very efficient.

Therefore a special form of *address-sensitive locking* that allows the locking of just blocks instead of the whole shared memory has been developed and implemented. This enables concurrent read- and write-access to regions of shared memory, as is demonstrated for two cores in Fig. 2.

For event synchronization *simple barriers* using fixed configuration bit patterns and more flexible *complex barriers* have been developed, the latter ones allow to specify multiple barriers for different subsets of the available processor cores.

C. Realization

The developed Multi-Access Controller (MACtrl) consists of *core-side* and *inter-core logic* as shown in Figure 3.

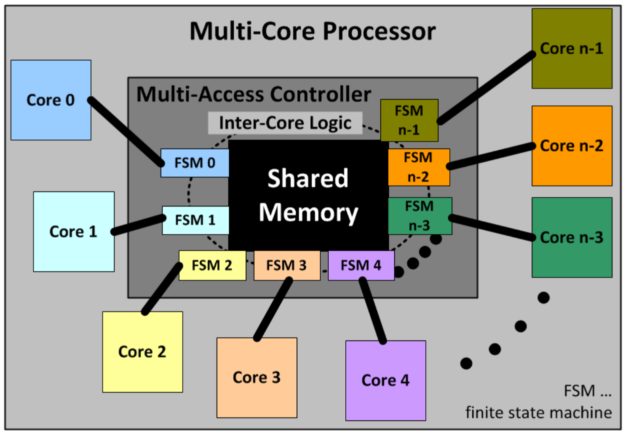

Fig. 3. Memory-access controller (MACtrl), abstraction

A fully generic design of the MACtrl has been developed in the hardware description language VHDL in order to allow easy scaling in terms of the processor cores. The goal to keep the design as compact as possible is achieved by a code-optimized algorithm that selects the next core that is allowed to access the shared memory in case of concurrent requests. The algorithm continuously cycles the highest priority among all available cores. Only cores requesting access are used and the other cores are masked out in the process.

In order to execute *multiple* accesses atomically also global locking is implemented using a variation of the algorithm.

Address-sensitive locking is activated as soon as an upper and a lower address of a memory block is loaded into the respective registers of the MACtrl. Then the block of memory is tried to be locked by the controller. Due to their very nature, global locking and address-sensitive locking are implemented to be mutual exclusive - the two access methods cannot be used simultaneously to access the shared memory.

Simple barriers are the easiest method to achieve efficient point-to-point synchronization: each core that wants to meet at a given point of execution writes an arbitrary value to a dedicated barrier register of the MACtrl. Then the respective processor core is blocked until at least one other core writes to its corresponding counterpart-register.

With more than two processor cores in the system, *extended simple barriers* offer the possibility to define exactly for what other cores to wait for. Each bit in the register corresponds to the fixed number of a processor core in the system. The drawback is that the number of the cores must be known at compile-time.

A more flexible replacement for the extended simple barriers are *complex barriers*. They allow a more hardware-remote — abstracted — level of programming: the numbers of the processor cores must not be known in advance, only the number of other cores to wait for, making it irrelevant on what processor core the program will eventually be executed.

IV. Implementation

All the hardware synchronization methods described in the previous Section were successfully simulated for four, some of them also for eight processor cores. From the beginning on there was no assumption limiting the number of cores.

A. System Configuration

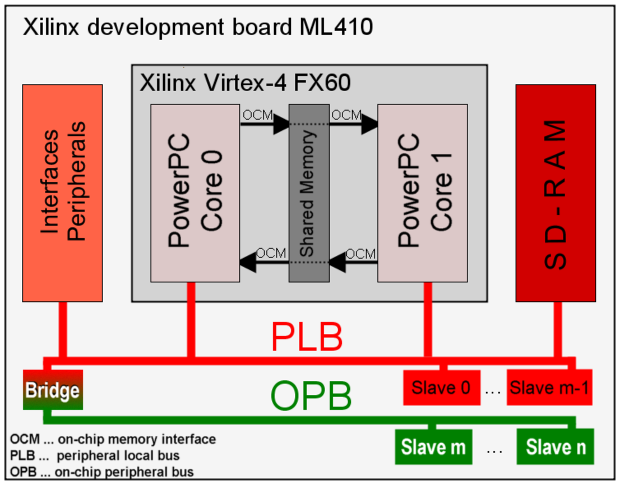

Fig. 4. Configuration of dual-core test system

For the actual hardware implementation the dual-core case could be examined thoroughly using a Xilinx development board ML410 with a mounted FPGA Virtex-4 FX60 containing two hard PowerPC processor cores and a dual-ported block RAM which is used as on-chip shared memory. The generic design has been specialized to fit as on-chip logic to the system at hand, using the on-chip memory interface (OCM) for accessing the shared memory and taking advantage of its two ports (concurrent reads possible). Also external shared SD-RAM was used as shared memory. The basic configuration of the system is drafted in Figure 4, freed from irrelevant vendor-specific details which can be found in [19], [20].

All implemented types of synchronization mechanisms were tested. As most important case the worst case of incessant access to the shared memory was simulated with the help of assembler routines. The core frequency of the processor cores was 100 MHz when conducting the following test data sets.

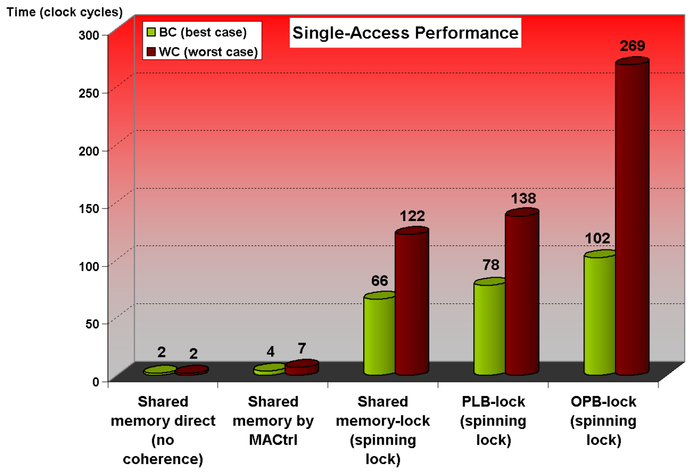

Fig. 5. Performance of single accesses to shared memory over all methods

B. Results

Figure 5 and Table I clearly show the superiority of the MACtrl with *implicit* and *explicit locking* to the badly performing spinning software locking methods. A total of 10 million successive single or paired accesses (read/write-pairs) to the same word in shared memory are executed in sum by both processor cores, resulting in countless conflicts between them.

In order to test *address-sensitive locking* an artificial scenario with overlapping memory blocks was constructed. Each processor core reserves a memory block, accesses it and then relocates it by an arbitrary offset. Relocation is done in opposite directions, hence conflicts between the processor cores occur. The promising results are presented in Table II.

Synchronization Details ([access-target], [locking details])	No	Massive
	Contention (ms)	
MACtrl	1030	1054
Spinning lock in on-chip memory	2734	4426
Spinning PLB-lock	2890	5512
Spinning OPB-lock	3681	7071
Shared SDRAM, spinning PLB-lock	4058	8116
Shared SDRAM, spinning OPB-lock	4874	8627

TABLE I

Paired accesses, exact values with lock-details

Synchronization Details ([access-target], [locking details])	No	Massive
	Contention (ms)	
MACtrl address lock	12144	12767
MACtrl global	13628	27496
Spinning lock in on-chip memory	12186	24608
Spinning PLB-lock	12707	24654
Spinning OPB-lock	12722	24679
Shared SDRAM, spinning PLB-lock	22215	46001
Shared SDRAM, spinning OPB-lock	22232	47241

TABLE II

Block accesses to different regions of memory

V. Conclusion

The problem of synchronization in multi-core systems with shared memory demands for efficient and reliable solutions, in particular for embedded systems. An approach is to integrate the synchronization mechanisms, which are normally based on locks, into the on-chip hardware. This guarantees fairness and stability, avoiding poor performance under load, starvation and even deadlock. A specialization of the multi-core approach to a dual-core PowerPC system proved the clear superiority of the hardware over software solutions that were implemented. Ongoing and future research focuses on testing efficient partitions of real embedded software on multi-core systems with hardware synchronization like the one presented here.

References

[1] K. Asanovic *et al.*, “A view of the parallel computing landscape,” in *Communications of the ACM, Vol. 52, No. 10.* ACM Press, October 2009, pp. 56–67.
[2] D. McGrath, “Intel rolls quad-core CPUs for embedded computing.” EE Times, April 2007. [Online]. Available: http://www.eetimes.com
[3] T. Takayanagi *et al.*, “A dual-core 64b UltraSPARC Microprocessor for Dense Server Applications,” Sun Microsystems, Sunnyvale, USA, 2004.
[4] D. E. Culler and J. P. Singh, “Parallel Computer Architecture, a hw/sw approach.” Morgan Kaufmann Publishers, Inc., Editorial and Sales Office, San Francisco, U. S. A., 1999.
[5] R. Johnson *et al.*, “A New Look at the Roles of Spinning and Blocking,” in *Proceedings of the Fifth International Workshop on Data Management on New Hardware, Providence, Rhode Island.* ACM Press, June 2009.
[6] C. Ferri, I. Bahar, M. Loghi, and M. Poncino, “Energy-optimal synchronization primitives for single-chip multi-processors,” in *GLSVLSI’09, Boston, Massachusetts.* ACM Press, May 2009, pp. 141–144.
[7] M. C. August *et al.*, “Cray X-MP: The Birth of a Supercomputer,” Cray Research, 1989.
[8] R. Gupta *et al.*, “The Design of a RISC based Multiprocessor Chip,” University of Pittsburgh, Philips Laboratories New York, 1990.
[9] Nikolopoulos and Papatheodorou, “Fast synchronization on scalable cache-coherent multiprocessors using hybrid primitives,” University of Patras, Greece, 2000.
[10] H.-J. Stolberg *et al.*, “HiBRID-SoC: A multi-core System-on-Chip architecture for multimedia signal processing applications,” Universitaet Hannover, Germany, 2003.
[11] S. Y. Cheung and V. S. Sunderam, “Performance of Barrier Synchronization Methods in a Multi-Access Network,” Emory University, Atlanta, Georgia, 1993.
[12] J. Sampson *et al.*, “Fast synchronization for chip multiprocessors,” in *ACM SIGARCH Computer Architecture News, Vol. 33*, UCSD, UPC Barcelona, Palo Alto, California, 2005.
[13] A. Marongiu, L. Benini, and M. Kandemir, “Lightweight barrier-based parallelization support for non-cache-coherent MPSoC platforms,” in *CASES’07, Salzburg, Austria.* ACM Press, Sept. 2007, pp. 145–149.
[14] C. Liu, A. Sivasubramaniam, M. Kandemir, and M. J. Irwin, “Exploiting barriers to optimize power consumption of CMPs,” in *Proceedings of IPDPS*, 2005.
[15] O. Vila, G. Palermo, and C. Silvano, “Efficiency and scalability of barrier synchronization on NoC based many-core architectures,” in *CASES’08, Atlanta, Georgia, USA.* ACM Press, October 2007, pp. 81–89.
[16] E. W. Lynch and G. F. Riley, “Hardware supported time synchronization in multi-core architectures,” in *ACM/IEEE/SCS 23rd workshop on principles of advanced and distributed simulation.* IEEE Press, 2009, pp. 88–94.
[17] A. Tumeo *et al.*, “HW/SW methodologies for synchronization in FPGA multiprocessors,” in *FPGA’09, Monterey, California, USA.* IEEE Press, 2009, pp. 265–268.
[18] R. Mueller, J. Teubner, and G. Alonso, “Data processing on FPGAs,” in *VLDB’09.* ACM Press, August 2009, pp. 910–921.
[19] *ML410 Embedded Development Platform User Guide*, v1.6.1 UG085, Xilinx, 2007.
[20] *Virtex-4 Family Overview*, v3.0 DS112, Xilinx, 2007.