An Efficient and Flexible Hardware Support for Accelerating Synchronization Operations on the STHORM Many-Core Architecture

Farhat Thabet, Yves Lhuillier, Caaliph Andriamisaina, Jean-Marc Philippe and Raphaël David
CEA LIST, Embedded Computing Lab,
91191 Gif sur Yvette, France
Email: {farhat.thabet, yves.lhuillier, caaliph.andriamisaina, jean-marc.philippe, raphael.david}@cea.fr

***Abstract*—The current trend in embedded computing consists in increasing the number of processing resources on a chip. Following this paradigm, the STMicroelectronics/CEA Platform 2012 (P2012) project designed an area- and power-efficient many-core accelerator as an answer to the needs of computing power of next-generation data-intensive embedded applications. Synchronization handling on this architecture was critical since speed-ups of parallel implementations of embedded applications strongly depend on the ability to exploit the largest possible number of cores while limiting task management overhead. This paper presents the HardWare Synchronizer (HWS), a flexible hardware accelerator for synchronization operations in the P2012 architecture. Experiments on a multi-core test chip showed that the HWS has less than 1% area overhead while reducing synchronization latencies (up to 2.8 times) and contentions.**

I. Introduction

Embedded systems need ever increasing levels of performance to handle higher data rate processing, new multimedia services and features such as security using a rich user interface. Moving to many-core architectures is one way to address peak performance needs since they deliver high performance and low power consumption while offering design flexibility. Exploiting this huge amount of computing power requires to face one major challenge: efficient exploitation of parallelism.

To tackle the challenges related to the design of a many-core architecture, STMicroelectronics and CEA (French Alternative Energies and Atomic Energy Commission) associated their expertises in the Platform 2012 (P2012) project [1]. P2012 is a join initiative aiming at designing an area- and power-efficient many-core computing fabric for next-generation data-intensive embedded applications (e.g. augmented reality) while providing flexibility. The cluster-based resulting architecture, named STHORM (ST(microelectronics) Heterogeneous lOw poweR Many-core) is described in [2].

In an embedded computing fabric like STHORM, synchronization handling is critical since synchronization events are frequent, especially when using fine-grain parallelism [3], [4]. Traditional synchronisation primitives implementations use memory accesses relying on atomic operations such as *test-and-set*. These processor instructions rely on dedicated hardware to ensure the atomicity of operations (test then set). Unfortunately, to be generic enough to be integrated into future STMicroelectronics products, no assumption had to be made on the system. An additional challenge was to design a hardware accelerator that is sufficiently flexible to efficiently implement different synchronization primitives such as mutual exclusions (locks or mutexes), semaphores and barriers.

Besides the challenge of efficiently implementing synchronization, executing threads are continuously polling shared variables (i.e. spin-lock) in traditional implementations at the expense of interconnect bandwidth and energy efficiency.

This paper introduces an area-efficient hardware module, the HardWare Synchronizer (HWS), which is dedicated to accelerate synchronization primitives on massively-parallel embedded architectures. Next section focuses on hardware supports for synchronization handling. Section 3 describes the HWS. Section 4 evaluates the HWS integration in a physical multi-processor system. Section 5 presents simple synchronization primitives implementations using the HWS and more complex use cases currently integrated into a lightweight runtime software. Finally, Section 6 concludes the paper.

II. Related work

In parallel programs, synchronization is a main concern for correct execution and efficient synchronization mechanisms drive their performance. Parallel architectures handle synchronization and more generally atomicity in different ways: leveraging their memory model (atomic operations) or leveraging synchronization constructs of dedicated programming models.

In the first category, parallel shared-memory architectures provide hardware support for atomicity through a combination of shared-memory subsystems. At processor level, *test-and-set*-like instructions [5] allow to exchange values of a memory location with the one of a register or a hard-coded value. This hardware support does not make any assumption on the programming model and efficiently supports a wide range of synchronization constructs. Nevertheless, it shows scalability issues since it is distributed over the whole memory subsystem while needing to behave as a centralized entity.

Evolutions of these techniques were proposed, such as the SoC Lock Cache [6] which implements the *test-and-set* instruction as a memory-mapped load instruction. However,

978-3-9815370-0-0/DATE13/©2013 EDAA

this approach is optimized for locks (one physical bit) that prevents it to efficiently implement semaphores or barriers.

Other techniques use coherence mechanisms available in busses (i.e. bus snooping) to control lock variables status, such as the Lock Table approach [7]. This method is neither scalable nor applicable to architectures relying on network-on-chips.

The Lock Control Unit (LCU) approach [8] adds a LCU to each core for implementing distributed lock queues and provides each memory controller with a Lock Reservation Table (LRT) managing the locking status of each memory location. This approach requires new ISA primitives (*Acquire*, *Release*) which is an issue when using existing cores.

Other parallel architectures support synchronizations through programming model dedicated hardware support. To address performance of synchronizations, some approaches [9] have implemented hardware networks for barrier synchronizations and collective communication operations that are associated with data level parallelism. Programming models based on thread-level parallelism [10] allow to confine synchronizations at task transitions steps.

A general analysis of the related work shows that most of them rely on hardware mechanisms so as to ensure atomicity for the synchronization primitives. These approaches are not compatible with systems which do not support this property. Additionally, most of these approaches belong to the general purpose computing domain, not the embedded one. Another approach had to be investigated for STHORM.

III. Description of the HardWare Synchronizer

The HardWare Synchronizer (HWS) was introduced to address challenges such as:

- acceleration of common synchronization constructs,
- flexibility and composability to adapt to evolving execution models,
- ease of integration in most systems and area efficiency,
- reducing bandwidth, power consumption and polling,
- ability to be replicated in many-core systems (scalability),
- providing support for atomic operations in absence of atomic support in the memory sub-system

A. HWS main features

Fig. 1 presents the HWS architecture. To allow its seamless integration in systems based on load/store interconnects, its interface is a standard target/slave port: the HWS is viewed as a shared peripheral (a set of registers). A simple wrapper can be implemented to expose the HWS registers to the system. The computing cores are linked to the HWS by wires implementing interrupts.

The HWS is modular. Different blocks implement dedicated functions to support synchronization and communication primitives: the Atomic Counters, the Programmable Notifier and the Interrupt/Event Trigger (a memory-mapped module which generates an interrupt/event to the target cores). The fourth block named Advanced Features proposes hardware-assisted mechanisms for handling dynamic task-scheduling, resource allocation acceleration and thread migration which are not in the scope of this paper.

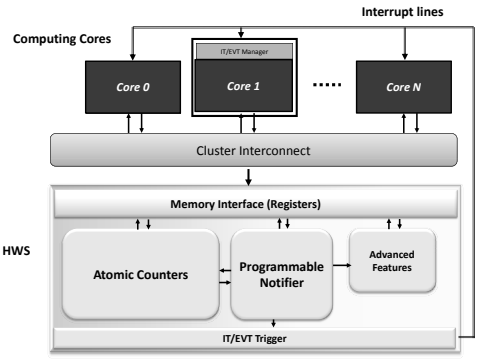


Fig. 1. HWS architecture overview.

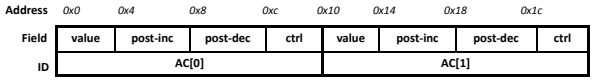


Fig. 2. Atomic Counters memory-mapped structure.

B. Atomic Counters module

The Atomic Counters (AC) module provides memory-mapped scalar values that can be atomically read-modify-written using load operations. This module deports the in-place modification in memory to remove the need for atomic operation support in the core and the memory interconnect sub-system. The modifications performed by the AC module support a wide range of atomic operations and synchronization mechanisms such as mutexes, semaphores and barriers.

In the current implementation, each physical AC has four 32-bit memory-mapped interfaces (Fig. 2). Store/load accesses to an AC have different properties depending on the address. The ”value” field allows to read/write the AC. When read, ”post-inc” and ”post-dec” fields return the value with an atomic side-effect of post-incrementing or post-decrementing the value. The ”ctrl” field configures the AC: it can be bound by writing a value to the ”post-inc” field (e.g. when reaching the upper bound, incrementing the AC has no effect).

An AC can be used to implement a hardware binary mutex using ”trylock” mechanisms (like in a *test-and-set* implementations). Unfortunately, the resulting extra-traffic between the processing cores and the HWS leads to *software polling*. To avoid polling, the HWS provides a runtime software support to implement various blocking synchronization functions that do not rely on the caller to actively poll hardware resources. The HWS is able to automatically notify the processing cores about an awaited event using the Programmable Notifier module.

C. Programmable Notifier module

The Programmable Notifier (PN) enables each processing core able to access the HWS to schedule event notifications according to specific ACs values. It allows a software code to program events notification whenever the value of an AC becomes equal or different (greater than, less than, not equal, less or equal than and greater or equal than) to zero and to

avoid software polling on the ACs while waiting for triggering conditions to occur. Thus, the traffic contention on the local interconnect is reduced. This hardware support ensures a high reactivity for synchronization barriers.

The PN consists of a set of memory-mapped condition storage registers, a control register and a condition checker. There is one condition storage register per AC which is read when the corresponding AC is modified: its value is used by the condition checker to check if the registered condition is matched by the new value of the AC. Each condition storage register contains three fields, *core_set* (set of cores to be notified), *evt_id* (interrupt line to be used for notification) and *pn_condition* (condition to be satisfied at every change on the target AC value). The condition storage registers are set or updated using a unique memory-mapped control register.

The first step for checking a condition consists in analyzing every access to an AC to verify if its new value satisfies the condition defined by the *pn_condition* field. In the second step, if the condition is met, then the corresponding events are generated. Finally, the set of fields stored in the corresponding condition storage register are reinitialized.

The PN module is useful for implementing barriers without polling (all threads are notified by event at barrier completion). Each thread entering the barrier updates the PN *core_set* for notification, decrements the AC value and enters into a waiting (i.e. low power) state until receiving an event notification.

IV. Integration Environment and Performance Evaluation

The HWS was chosen to be the infrastructure for synchronization handling in the STHORM architecture [2].

A. Evaluation of hardware-related characteristics of the HWS

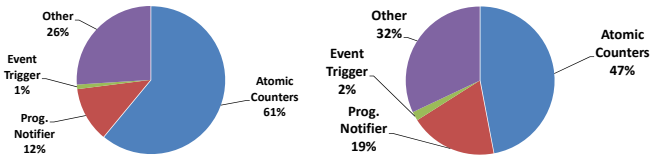
Scalability and area cost of the proposed approach are investigated using two environments. The first one is the RTL model of the HWS which was synthesized using different parameters of the STHORM cluster. The second one is a 32-nm test-chip of an instance of the STHORM platform, called Locomotiv, composed of one cluster of 4 cores.

1) Area and Frequency evaluation: synthesis experiments were conducted using first the Locomotiv configuration (4 PEs) and then STHORM cluster configurations (8 and 16 PEs) with the complete HWS configuration. The results were obtained from RTL models synthesized using a STMicroelectronics 32nm CMOS low-power library on Synopsys Design Compiler E-2010.12. The HWS achieved an operating frequency of 600MHz.

The area evaluation results are given for a 500 Mhz operating frequency. The HWS area and scalability evaluations were done using two different experiments. The first one evaluated the HWS area with respect to the number of processing elements with a fixed number (i.e. 64) of 32-bit ACs. Table I shows that for example the area of the Locomotiv HWS is about 0.049 mm^2 ($\simeq$ 70 KGates) which represents 1% of the chip area. It also shows that the overhead of the HWS compared to the total size of the cluster decreases when the number of PEs increases.

TABLE I
AREA, CLUSTER OVERHEAD AND POWER CONSUMPTION OF THE HWS WITH RESPECT TO THE NUMBER OF PES OF THE CLUSTER.

PEs	Area (mm^2)	Overhead (%)	Power Cons.(mW)
4	0.049	1.00	18.7
8	0.061	0.62	22.9
16	0.085	0.43	31.9



(a) Locomotiv Configuration - 4 PEs and 64 32-bit ACs

(b) STHORM Typical Configuration - 16 PEs and 128 32-bit ACs

Fig. 3. Contributions of different HWS parts to the global area in a 32nm technology.

TABLE II
AREA AND POWER CONSUMPTION OF THE HWS WITH RESPECT TO THE NUMBER OF ACS.

32-bit Atomic Counters	Area (mm^2)	Power Cons. (mW)
32	0.042	12.1
64	0.061	18.7
128	0.098	31.9

Fig. 3 presents the contribution of differents HWS parts to total area budget for Locomotiv and a typical STHORM configuration. This figure illustrates that for typical configurations, the ACs are one of the main contributors to HWS area budget. The second experiment evaluated the impact of the number of ACs on HWS area (the number of PEs was kept constant (i.e. 8)). Table II shows that doubling the number of ACs has a non-negligible area impact on the HWS. This increase is mainly due to the duplication of AC registers and the related consequences on internal multiplexer and demultiplexer logics.

2) Power Consumption Evaluation: the power consumption evaluation used a VC-1 decoder application [11]. This application was parallelized and implemented on a TLM simulation platform of STHORM and a runtime software using HWS acceleration features. The different accesses to the HWS TLM model were recorded to have traces of real execution of the application. These traces were transformed into VHDL testbench accesses to obtain the peak power consumption using Synopsys PrimeTime E-2010.12 in a STMicroelectronics 32-nm technology. Results are presented in Tables I and II. The peak dynamic power consumption of the HWS included in Locomotiv (i.e. for 4 PEs) is about 18.7 mW.

V. HWS use cases for Runtime Software Acceleration

In STHORM, the HWS is used to accelerate synchronization mechanisms, runtime-software-related tasks and powerful execution engines. Together with the HWS, a lightweight runtime

TABLE III
MEMORY ACCESSES FOR STANDARD SYNCHRONIZATION CONSTRUCTS.

Class	Op.	Description	Behavior
Binary mutexes	lock	exclusively acquires a mutex	$1 \times LD_{ac} + 1 \times ST_{pn}$
	unlock	releases an owned mutex	$1 \times ST_{ac}$
Semaphores	wait	waits for a free semaphore token	$1 \times LD_{ac} + 1 \times ST_{pn}$
	post	releases a semaphore token	$1 \times LD_{ac}$
Barriers	wait	synchronizes concurrent jobs	$1 \times LD_{ac} + 1 \times ST_{pn}$
Joiners	join	merges concurrent jobs	$1 \times LD_{ac}$
Queues	reserve	reserves a writable queue slot	$1 \times LD_{ac} + tries \times LD_{ac}$
	send	sends a writable queue slot	$1 \times LD_{ac} + tries \times LD_{ac}$
	peek	reserves a readable queue slot	$1 \times LD_{ac} + tries \times LD_{ac}$
	release	recycles a readable queue slot	$1 \times LD_{ac} + tries \times LD_{ac}$

software called HARS (Hardware-Assisted Runtime Software) was introduced to manage the platform.

A. Implementation of simple synchronization mechanisms

Table III shows the behavior of some synchronizations implemented in HARS using the HWS. Synchronization classes cover a wide range of synchronization mechanisms from binary mutexes to circular buffer queues. The last column shows the analytical behavior with respect to needed memory accesses: LD/ST_{pn} is for PN accesses (read/load or write/store), LD/ST_{ac} is for AC accesses, and $tries$ is for conditional trials (i.e. polling).

Table III shows that active polling is almost never required since it is replaced by a single PN access ($1 \times ST_{pn}$). Active polling is still required in HWS-based implementation of circular buffer queues as the PN is not able to perform side-by-side atomic-counters and non-zero comparisons. If the HWS is accessible in 3 cycles and the cluster-level shared-memory access costs 1 cycle, "reserve/send" and "peek/release" operations take an average of 26 cycles (HWS) instead of 40 cycles (*test-and-set*-based implementation).

Finally, most synchronization constructs (mutexes, barriers, etc.) are reduced to only two memory accesses (one AC access plus one PN access). For these constructs, *test-and-set*-based implementation often requires several accesses (1 per failed trial) for a mutex acquisition, and at least two accesses for *read-modify-write* operation on the synchronization structure meta-data. These memory access reductions allow to decrease synchronization primitives latency, to limit memory sub-sytem contention and to save bandwidth for computations.

B. RTM execution engine implementation using HWS features

On top of basic synchronization and allocation primitives building the core set of HARS APIs, some execution engines were implemented. Especially, the HWS has been used to optimize the Reactive Tasks Management (RTM) [3] execution engine. RTM is a fast lightweight fork/join API which leverages the HWS to accelerate its software scheduler. The RTM uses two ACs of the HWS per parallel fork operation. Using the HWS, this software scheduler is able to schedule duplicated tasks (i.e. *dup*) in 23 cycles and different tasks (i.e. *fork*) in 26 cycles, allowing to efficiently manage fine-grain tasks. A *test-and-set*-based implementation of this scheduler takes an average of 66 cycles per scheduled task. A more powerful asynchronous version is described in [4].

VI. CONCLUSION

This paper presented the HWS Atomic Counters and Programmable Notifier hardware supports that handle synchronization primitives in a flexible/scalable dedicated module. Thanks to the HWS, multi- and many-core system developers can write optimized dynamic task-scheduling and allocation algorithms, barrier synchronizations, communicating queues and atomic operations. The HWS has been integrated and validated in the Locomotiv chip and is the synchronization support of the STHORM many-core architecture. Its low area and power budget makes it suitable for many-core embedded architectures.

ACKNOWLEDGMENT

This work was partly supported by the European cooperative CATRENE project CA104 COBRA.

REFERENCES

[1] STMicroelectronics and CEA, "Platform 2012: A many-core programmable accelerator for ultra-efficient embedded computing in nanometer technology," *Research Workshop on STMicroelectronics Platform 2012, Toronto*, 2010.

[2] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Design, Automation & Test in Europe Conference & Exhibition (DATE12)*, 2012, pp. 983–987.

[3] M. Ojail, R. David, K. Ben Chehida, Y. Lhuillier, and L. Benini, "Synchronous reactive fine grain tasks management for homogeneous many-core architectures," in *ARCS '11*, February 2011.

[4] M. Ojail, R. David, Y. Lhuillier, and A. Guerre, "Artm: A lightweight fork-join framework for many-core embedded systems," in *Design, Automation Test in Europe Conference Exhibition (DATE)*, march 2013.

[5] J. M. Mellor-Crummey and M. L. Scott, "Algorithms for scalable synchronization on shared-memory multiprocessors," *ACM Trans. Comput. Syst.*, vol. 9, pp. 21–65, February 1991.

[6] B. E. S. Akgul and V. J. Mooney III, "The system-on-a-chip lock cache," *Design Automation for Embedded Systems*, vol. 7, pp. 139–174, 2002. [Online]. Available: http://dx.doi.org/10.1023/A:1019751632622

[7] M.-C. Chiang, "Memory system design for bus-based multiprocessors," Ph.D. dissertation, Madison, WI, USA, 1991.

[8] E. Vallejo, R. Beivide, A. Cristal, T. Harris, F. Vallejo, O. Unsal, and M. Valero, "Architectural support for fair reader-writer locking," in *Microarchitecture (MICRO), 2010 43rd Annual IEEE/ACM International Symposium on*, dec. 2010, pp. 275 –286.

[9] S. F. Lundstorm and G. H. Barnes, "A controllable mimd architecture," in *Proceedings of the International Conference on Parallel Processing*, 1980, pp. 19–27.

[10] Apple Inc., "Grand Central Dispatch (GCD) Reference," Tech. Rep., May 2010.

[11] M. Bariani, P. Lambruschini, and M. Raggio, "Vc-1 decoder on stmicroelectronics p2012 architecture," in *In 8th Intl. Workshop STreaming Day*, September 2010.