Evaluation of Message Passing Synchronization Algorithms in Embedded Systems

Lazaros Papadopoulos[1], Ivan Walulya[2], Philippas Tsigas[2], Dimitrios Soudris[1] and Brendan Barry[3]

[1] School of Electrical and Computer Engineering, National Technical University of Athens, Greece.
{lpapadop, dsoudris}@microlab.ntua.gr

[2] Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden.
{ivanw, philippas.tsigas}@chalmers.se

[3] Movidius Ltd., Dublin, Ireland. brendan.barry@movidius.com

***Abstract*—The constantly increasing computational power of the embedded systems is based on the integration of a large number of cores on a single chip. In such complex platforms, the synchronization of the accesses of the shared memory data is becoming a major issue, since it affects the performance of the whole system. This problem, which is currently a challenge in the embedded systems, has been studied in the High Performance Computing domain, where several message passing algorithms have been designed to efficiently avoid the limitations coming from locking. In this work, inspired from the work on message passing synchronization algorithms in the High Performance Computing domain we design and evaluate a set of synchronization algorithms for multi-core embedded platforms. We compare them with the corresponding lock-based implementations and prove that message passing synchronization algorithms can be efficiently utilized in multi-core embedded systems. By using message passing synchronization instead of lock-based, we managed to reduce the execution time of our benchmark up to 29.6%.**

Keywords— message passing; multi-core embedded systems; lock-free;

I. INTRODUCTION

In the last years, we experience the constantly increasing computational power of the embedded systems. There exist a large number of multi-core embedded platforms running complex applications like 3D-games and databases and performing computational demanding tasks, such as video processing and image rendering. According to Bell's Law [22], roughly every decade evolves a new lower priced computer class (i.e. a category of computer systems) that replaces the existing one. This class creates a new market and establishes a new industry. Nowadays, the new class can be considered the embedded systems. Indeed, we experience the trend of porting computational demanding applications from general purpose computers and High Performance Computing (HPC) systems to embedded platforms. Such platforms are utilized in high-end consumer embedded devices, such as smartphones and game consoles.

The constantly increasing computational power of embedded systems is a result of the integration of more and more cores on a single chip. However, it has been proven that the integration of multiple cores does not necessarily increases the performance of real-world applications [19]. The shared memory data synchronization issue is critical in multithreading programming, since it can greatly affect the performance of the whole system. The synchronization techniques developed so far can be generally categorized as lock-based (e.g. locks, semaphores) and lock-free [1]. These techniques have been developed mainly for high performance computing platforms. Multi-core embedded systems rely so far mostly on mutual exclusion and interrupt handling to achieve synchronization.

Regardless the well-known disadvantages of the lock-based approaches (e.g. limited scalability, starvation and blocking), extensive research on the HPC domain shows that lock-based mechanisms are efficient in cases of low contention [2]. Therefore, they can provide adequate performance in the case of embedded systems with a relative small number of processing cores and low contention. Lock implementation primitives can be found in many single-core and multi-core embedded systems [3].

However, the number of cores integrated on a single embedded chip will increase further over the next years. The term High Performance Embedded Computing (HPEC) has been recently used to describe embedded devices with very large processing power, used mostly in aerospace and military applications [4]. For instance, Wandboard Quad multimedia board integrates 4 ARM Cortex-A9 cores [5], while the embedded processor AMD Opteron 6200 integrates 16 x86 cores [6]. The trend of integrating more and more cores on a single chip will continue over the next years, since embedded systems are expected to process large amounts of data in embedded servers or perform computational intensive operations, such as high resolution rendering, image processing, etc.

In embedded systems with large number of cores accessing shared data, the scalability can be a major issue. Scalability refers to how system throughput is affected by the increasing number of contending threads. Lock-based techniques, such as mutexes are not expected to be efficient in embedded systems with such a high number of cores, since they do not scale in case of high contention and become a bottleneck leading to

This work was supported by the EC through the FP7 IST project 611183, EXCESS (Execution Models for Energy-Efficient Computing Systems).

performance decline. This has been adequately proven in experiments on the HPC domain [2]. Additionally, locks can cause unpredictable blocking times, making them unattractive in real-time embedded systems. Therefore, other scalable synchronization algorithms should also be evaluated to overcome the limitations of locks.

It has been argued, during the last years, that the embedded system and HPC domains are gradually converging [7]. The shared data accessing synchronization issues that appeared on the HPC domain in the past are now a major issue on the multi-core embedded systems. Therefore, the adoption of techniques from the HPC to the embedded systems domain is likely to lead to efficient solutions. Lock-free mechanisms have been developed in the HPC domain as alternatives to locks, aiming to avoid the lock disadvantages and retain scaling under high contention. In this work we evaluate a number of lock-free solutions in the embedded systems domain.

To summarize our motivation, we argue that embedded platforms tend to integrate an ever-increasing number of cores on a single chip. For instance, Myriad1 embedded platform integrates 8 cores and newer versions of the platform are expected to integrate even more cores [18]. Therefore, synchronization of accessing shared data is becoming a critical issue in such platforms. The same issues apply to the HPC domain, where it is proven that lock-based synchronization is becoming inefficient, as the number of cores (and therefore the contention) increases. A proposed solution in the HPC domain is the message-passing synchronization. In this work we transfer this solution to the embedded systems domain.

More specifically, in this paper we evaluate four synchronization algorithms that can be used as alternatives to locks in embedded platforms with a large number of cores, where contention is relatively high and locks become inefficient. These algorithms are inspired by designs from the HPC domain and are based on the message passing, which seems to be an efficient solution under high contention [14]. The general idea is to dedicate cores that do not contribute to improving the performance of the application to handle the synchronization. The four algorithms we evaluated can be divided in two categories: Two of them use a Client-Server model, where a core plays the role of the server, synchronizing the access to shared data. The second category is an adaptation of the Remote Core Locking, (initially proposed in the HPC domain), to embedded systems: A core which is not utilized by the application, not only synchronizes the access requests to the critical sections, but also executes them [14][20]. We implemented the aforementioned algorithms in a multi-core embedded platform and tuned them to the platform specifications. As a shared data structure case study, we chose the queue, which is one of the most widely used data structures in embedded applications (used, for instance, in path-finding and work-stealing algorithms).

The rest of the paper is organized as follows: Section II discusses the related work. Next, we provide a summary of the technical details of the embedded platform used to evaluate our implementations. Section IV is the description of the proposed algorithms. The experimental results are presented and discussed on Section V. Finally, we draw our conclusions in Section VI.

II. Related Work

The main synchronization paradigm used in embedded systems is based on mutual exclusion and more specifically on locks. The instruction set of many modern embedded processors supports specific atomic instructions that can be used as primitives for implementing mutual exclusion. For instance, the latest ARM processors provide atomic load and store (load exclusive and store exclusive) instructions used to implement mutexes [3]. Another common way of achieving synchronization, used mainly in simple embedded platforms is based on the utilization of disabling interrupts for preventing task preemption inside a critical section [8].

Various lock-based synchronization techniques have been proposed in embedded systems. The C-Lock approach tries to combine the advantages of both locks and Transactional Memory, by detecting conflicts and avoiding a roll-back overhead [9]. Also, disabling the clock of blocked cores minimizes power consumption. Synchronization-operation Buffer is a hardware block targeting the minimization of polling operations [10]. Speculative Lock Elision allows the concurrent execution of non-conflicting critical sections [11]. All the aforementioned techniques try to improve the efficiency of locks. However, as previously stated, locks are expected to be a performance bottleneck in case of high contention, caused by a large number of cores trying to access shared data.

Another proposed approach is the Embedded Transactional Memory (Embedded TM) that tries to search a compromise between the simplicity and the energy efficiency required by embedded systems [12]. An evaluation of a lock-free synchronization approach in single processor dynamic real-time embedded systems can be found in [13].

Message passing techniques have been extensively researched in the HPC domain. For instance, Remote Core Locking (RCL) is based on the utilization of a dedicated server core which executes the critical sections [14][20]. Flat combining technique is an entirely software solution: the role of the server which serves the access requests to the critical sections is played by client threads in a periodical manner [21]. Intel SCC [15] experimental processor utilizes hardware message passing. Other works from the HPC domain focus on Hardware Transactional Memory [16] and in mutual exclusion techniques, like the token-based messaging [17].

In this paper we propose four message passing algorithms from the HPC domain, similar to the one proposed in [14] and evaluate them on a multi-core embedded platform.

III. Platform Description

Myriad1 is a 65nm heterogeneous Multi-Processor System-on-Chip (MPSoC) designed by Movidius Ltd [18] to provide high throughput coupled with large memory bandwidth. The design of the platform is tailored to satisfy the ever-increasing demand for high computational capabilities at a low energy footprint on mobile devices such as smartphones, tablets and wearable devices.

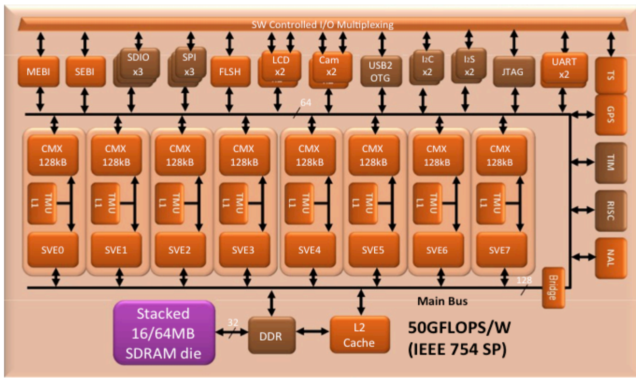

Fig. 1. Myriad1 architecture diagram: The RISC processor (LEON) and the 8 SHAVE cores. In each core a CMX memory slice is attached. (TMU is the Texture Management Unit). The 64MB SDRAM memory is depicted at the bottom of the diagram.

A. General Description of Myriad1 Platform

The recommended use case of the Myriad chip is as a co-processor connected between a sensor system such as a set of cameras and the host application processor. Myriad platform is designed to perform the heavy processing on the data stream coming from the sensor system and feed the application processor with metadata and processed content from the sensor system.

The heterogeneous multi-processor system integrates a 32-bit SPARC V8 RISC processor core (LEON) utilized for managing functions such as setting up process executions, controlling the data flow and interrupt handling. Computational processing is performed by the Movidius Streaming Hybrid Architecture Vector Engine (SHAVE) cores with an instruction set tailored for streaming multimedia applications. The Myriad1 SoC integrates 8 SHAVE processors as depicted in Fig. 1.

Regarding the memory specifications, the platform contains 1MB on-chip SRAM memory (named Connection Matrix – CMX) with 128KB directly linked to each SHAVE processor providing local storage for data and instruction code. Therefore, the CMX memory can be seen as a group of 8 memory "slices", with each slice being connected to each one of the 8 SHAVEs. The Stacked SDRAM memory of 64MB is accessible through the DDR interface. (Stacked SDRAM will be referred as DDR in the rest of the paper). Finally, LEON has 32KB dedicated RAM (LRAM).

Table I shows the access costs of LEON and SHAVES for accessing LRAM, CMX and DDR memories. Access cost refers to the cycles needed to access each memory. We notice that LEON has low access cost on CMX and potentially on DDR. The same applies to the SHAVEs. However, SHAVE access time to the DDR is much higher in comparison with the access time to CMX for random accesses. DDR is designed to be accessed by SHAVEs efficiently only through DMA. It is important to mention that each SHAVE accesses its own CMX slice at higher bandwidth and lower power consumption.

TABLE I. ACCESS COSTS FOR LEON AND SHAVES ACCESSING DIFFERENT MEMORIES

Memory	Size	LEON access cost	SHAVE access cost
LRAM	32KB	Low	High
CMX	1MB	Low	Low
DDR	64MB	• High • Low when data cache hit	• Low via DMA • Low for L1 cache hit • Moderate when L2 hit • High for random access

Myriad platform avails a set of registers that can be used for fast SHAVEs arbitration. Each SHAVE has its own copy of these registers and its size is 4x64 bit words. An important characteristic is that they are accessed in a FIFO pattern, so each one of them is called "SHAVE's FIFO". Each SHAVE can push data to the FIFO of any other SHAVE, but can read data only from its own FIFO. A SHAVE writes to the tail of another FIFO and the owner of the FIFO reads from the front. If a SHAVE attempts to write to a full FIFO, it stalls. Finally, LEON cannot access SHAVE FIFOs.

SHAVE FIFOs can be utilized for achieving efficient synchronization between the SHAVEs. Also, they provide an easy and fast way for exchanging data directly between the SHAVEs (up to 64 bits per message), without the need of using shared memory variables.

B. Mutexes on Myriad1Platform

The chip supports basic synchronization primitives implemented in hardware. It avails Test-and-Set registers that can be used to create spin locks, which are commonly referred as "mutexes". Spin-locks are used to create busy-waiting synchronization techniques: a thread spins to acquire the lock so as to have access to a shared resource. As mentioned in Section I, busy-waiting techniques can result in high contention on the interconnect buses and could result in starvation of threads trying to acquire the lock. Additionally the busy-waiting technique has very aggressive energy demands.

However, analysis of experiments on the Myriad1 platform shows that the mutex implementation is a fair lock with round-robin arbitration. Therefore, there is enough time available for each SHAVE to acquire the lock, so we observed that the locks have very low latency. Myriad1 platform provides 8 hardware implemented mutexes.

As mentioned before, synchronization techniques based on locks have been observed not to scale well with increasing thread count [2]. We anticipate that the thread count on the Myriad platform is bound to increase with increase demand and interest in multi-core embedded processors. To this effect, it is paramount that we develop synchronization techniques that can scale fairly on these embedded platforms with respect to both system throughput and energy consumption.

In contrast to synchronization under the HPC domain, the embedded systems pose very stringent constraints on energy, memory and resource management prerequisites. Considering the hardware restrictions, we apply synchronization techniques

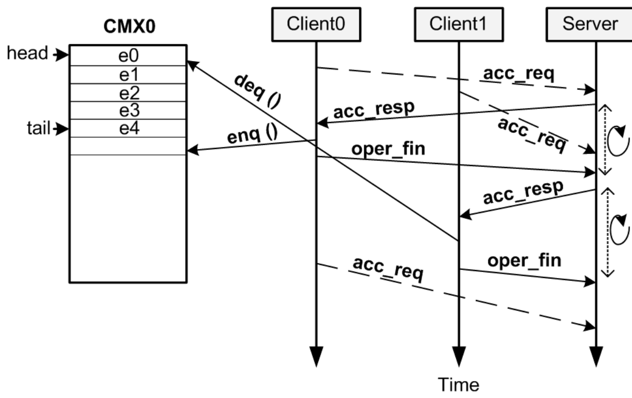

Fig. 2. Client-Server Implementations. Notice that the server remains idle (spinning on a local variable or on a mutex register value) while it is waiting for the client to exit the critical section.

evaluated so far only in the HPC domain, to the embedded systems domain.

IV. Description of Synchronization Algorithms

In this section, we describe the synchronization algorithms we implemented in the context of this work. All the algorithms were evaluated using a concurrent queue data structure, which is implemented as an array [23]. Elements are inserted to the tail of the queue and removed from the head, by decreasing or increasing the head and the tail pointer, respectively.

The concurrent queue is shared by all the SHAVE cores. In other words, all SHAVEs try to access the queue by enqueueing and / or dequeueing elements. Therefore, the critical sections of the concurrent queue are obviously inside the *enqueue()* and *dequeue()* functions. The queue array was placed in the CMX memory. Although the CMX is much smaller in comparison with the DDR memory, it provides much smaller access time for the SHAVEs, than the DDR.

The synchronization algorithms we implemented can be divided in three categories: The Lock-based, the Client-Server and the Remote Core Locking (RCL) implementations.

A. Lock-based Implementations

The lock-based implementations of the concurrent queue utilize the mutexes provided by the Myriad architecture. We designed two different lock-based implementations: In the first one, a single lock is used to protect the critical section of the *enqueue()* function and a second one to protect the critical section of the *dequeue()* [24]. Therefore, simultaneous access to both ends of the queue can be achieved. The second implementation utilizes only one lock to protect the whole data structure.

B. Client-Server Implementations

In the Client-Server implementations one of the cores is utilized as a server to arbitrate the clients' access to the shared data. We implemented two different versions of the algorithm: In the first one, the clients and the server communicate through shared variables, while in the second one they utilize the mutexes for achieving faster communication.

Both implementations are based on the idea of using LEON as an arbitrator (server), which will accept requests from SHAVEs (clients) for accessing the shared data structure. The communication is based on message passing. It is important to mention that the Myriad1 platform is ideal for such kind of synchronization algorithms. LEON is indeed designed to be utilized for controlling the data flow on the platform and SHAVEs for playing the role of the workers, responsible for performing computational intensive operations assigned to them by LEON. So, the utilization of LEON as an arbitrator is not expected to degrade the application's performance, since this is the kind of tasks LEON is expected to perform in Myriad1.

The Client-Server implementation is relatively simple. There are 3 kinds of messages exchanged between the server and the clients: *acc_req* (sent from a client to the server to request access to the shared data structure), *acc_resp* (sent from the server to a client to grant access) and *oper_fin* (sent from a client to the server to notify the server that the client has left the critical section).

Each client requests access to the shared queue by sending to the server an *acc_req* message. The client waits until it receives an *acc_resp* message from the server. When such a message is received, the client has exclusive access to the critical section and completes one operation (enqueue or dequeue). As soon as the operation is completed, the server is notified by receiving an *oper_fin* message from the client. Then, the server is ready to handle the next request, issued from another (or the same) client. Therefore, clients are served in a cyclical fashion.

To illustrate further the algorithm, Fig. 2 shows an example: *Client0* requests access to the queue by sending an *acc_req* message to the server. It spins on the buffer until it receives an *acc_resp* message. Then, it accesses the queue in the CMX memory and performs an enqueue. Upon the completion of the operation, *Client0* notifies the server with an *oper_fin* message. *Client1* requests access to perform a dequeue. Its request is handled by the server immediately after receiving the *oper_fin* from *Client0* and is served in a similar manner.

As previously stated, the first variation of the algorithm utilizes buffers implemented as shared variables between the LEON and the SHAVEs, which are placed in the CMX local slices of each SHAVE. The second variation utilizes mutexes in the following manner: when a client receives an *acc_resp* message and enters the critical section acquires one the 8 mutexes provided by the Myriad1 architecture. The client releases the mutex as soon as it exits the critical section. At the same time, LEON spins on the mutex status register and when the mutex get released, LEON is ready to serve another request. Thus, the *oper_fin* message is implemented though mutexes, instead of using a shared variable. It is important to underline again that mutexes in this algorithm are not used for protecting shared data (as happens with the lock-based implementations). Instead, they are utilized for achieving efficient communication.

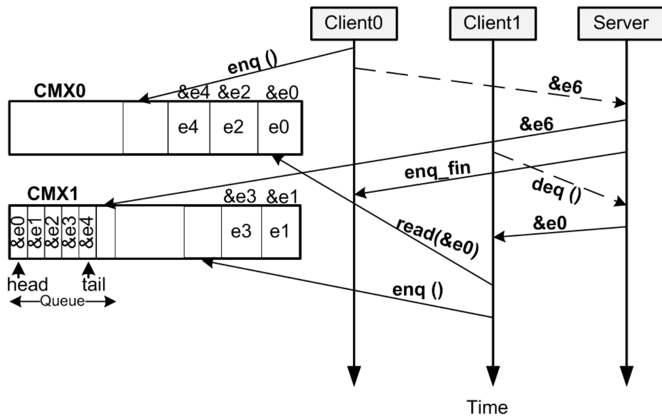

Fig. 3. Remote Core Locking (RCL) implementation. Each SHAVE stores its elements in its dedicated CMX memory. Notice the the queue contains the addresses of the elements. The server executes the critical sections, by enqueuing or dequeuing the addresses of the elements.

The Client-Server algorithms do not utilize much space in the CMX. Also, they allow the direct access of the data structure by the SHAVEs. LEON is responsible only for arbitrating the access requests to the queue, so it is actually replacing the lock. More specifically, it loops over the *acc_req* buffer slots to identify the clients that request access. Therefore, clients access the critical sections of the data structure in a cyclical fashion.

The motivation of using mutexes for communication lies in the assumption that the accessing of mutex registers by the LEON should be faster than spinning on shared variables. Mutexes are implemented on hardware and low-level assembly code and therefore are expected to provide increased communication performance.

The Client-Server implementations move the synchronization from the mutexes of the lock-based implementations, to a core that has the role of the arbitrator. They are relatively simple and the Client-Server implementation using mutexes for communication is expected to perform relatively fast. Client-Server designs can be considered as relatively platform independent, since they do not utilize any specific platform characteristics. Another advantage of this category of algorithms is that it provides fair utilization of the data structure by all clients, since they are served in a cyclical fashion. However, the main disadvantage is that the core having the role of the server is underutilized, since it performs only very limited amount of work. In this design, it is restricted in sending and receiving messages to the client cores and does not execute any part of the actual application workload.

C. *Remote Core Locking Implementations*

We optimized the Client-Server implementations by upgrading the role of the server. Instead of using the server only for arbitrating the access to the shared data structure, we moved the execution of the critical sections from the clients to the server. We implemented two different versions of the algorithm. In both of them we took advantage of the Myriad1 platform specifications. As mentioned before, Myriad1 platform provides 128KB of local CMX memory for each SHAVE. Taking advantage of the relatively large CMX memory, we optimized the queue by allowing the SHAVEs to access only their local CMX slice, instead of granting access directly to the queue. The first implementation uses shared variables for communication between the LEON and the SHAVEs, while the second one utilizes the SHAVEs' FIFO for achieving fast arbitration to the shared data.

The differences between the Client-Server and the Remote Core Locking (RCL) implementations can be summarized as follows:

- LEON (server) is responsible not only for arbitrating, but also for enqueuing / dequeuing elements to / from the queue. Therefore, the server executes the critical sections.
- Queue entries are not the allocated elements, but pointers to the elements that are physically allocated in local CMX memory slices. Therefore, RCL implementations occupy more memory space that the Client-Server ones, where the clients access the queue directly for storing and retrieving elements to / from the queue. This memory overhead of the RCL is obviously application specific.
- While in the Client-Server implementations the server remains idle while waiting for a client to exit the critical section, in the RCL implementations the server executes the critical section. At the same time, the client may remain idle or continue with the application execution, if possible, or even issue more requests to the server for accessing the shared data.

More specifically, when a client requests access to the data structure to enqueue an element allocated in its local CMX slice, it sends the address of the element to the server. Then, the server enqueues the address and replies with an *enq_fin* message to notify the client that the operation is completed. In the case of dequeueing, the client sends a *deq()* message to the server and waits for the address of the dequeued element.

Fig. 3 shows an example: *Client0* allocates the new element e6 to its local CMX slice. Then, it sends the address of element (*&e6*) to the server. The server enqueues the address to the queue (which in this example is placed in the *CMX1*) and it responds with an *enq_fin* message. *Client1* requests to dequeue, so it sends a *deq()* message and spins on the buffer waiting for an element address. The server responds by sending the address of e0 (*&e0*). *Client1* accesses the dequeued element in the *CMX0* memory slice, where it is allocated.

We implemented two versions of the RCL on the Myriad1 platform. The first one uses shared variables for the communication between the clients and the server. The second one utilizes the SHAVE FIFOs. More specifically, the element addresses, the *enq_fin* and the *deq()* messages are sent and received between the SHAVEs through the FIFO of each one. It is important to mention that, in contrast with all the other message passing implementations, in this case (i.e. the second version of the RCL implementations) the server is not the LEON, but one of the 8 SHAVE cores, since LEON does not have access to the SHAVE FIFOs.

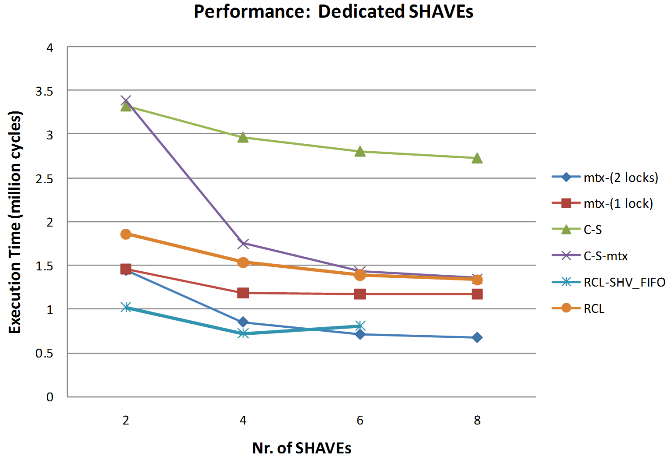

Fig. 4. Performance when half SHAVEs perform enqueue and half dequeue operations. In the *RCL-SHV_FIFO* implementation there are only 6 clients and one SHAVE is used as a server.

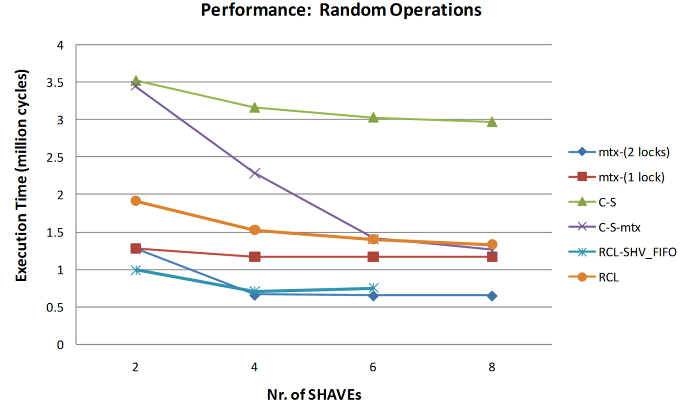

Fig. 5. Performance when the SHAVEs perform randomly enqueue and dequeue operations. In the *RCL-SHV_FIFO* implementation there are only 6 clients and one SHAVE is used as a server.

The implementations are an adaptation of the RCL algorithm presented in [20]. The main characteristic of the algorithm is that the arbitrator executes the critical section of the application (in the case of concurrent queue the enqueing and dequeing). The clients send requests to the server and the arbitrator is responsible for serving them. Therefore, in case of enqueueing there is no reason for SHAVEs to stall, while waiting for server to complete the enqueue. Instead, they can continue with other computations or even issue more enqueue requests. SHAVEs stall only in dequeue operations, while waiting for server to send the address of the dequeued element. The RCL implementations are more platform dependent in comparison with the Client-Server. They require a relatively large local memory (since they allocate new objects only in their local CMX slice) and in the case of the SHAVE's FIFO implementation the necessary hardware support.

V. EXPERIMENTAL RESULTS

The algorithms described in the previous section were evaluated using a concurrent array-based queue. The queue is shared between the 8 SHAVE cores of the Myriad1 platform. The synthetic benchmark we used is composed by a fixed workload of 20,000 operations and it is equally divided between the running SHAVEs. In other words, in an experiment with 4 SHAVEs each one completes 5,000 operations, while in an experiment with 8 SHAVEs, each one completes 2,500 operations.

All algorithms were evaluated in terms of time performance, for the given fixed workload, which is expressed in number of execution cycles. More specifically, in Myriad1 platform the data flow is controlled by LEON. SHAVEs start their execution when instructed so by LEON and then LEON waits for them to finish. The number of cycles measured is actually LEON cycles from the point that SHAVEs start their execution until they all finish. This number represents accurately the execution time.

A. Experimental Results

We performed two sets of experiments for evaluating the behavior of the designs: dedicated SHAVEs and random operations. In the "dedicated SHAVEs" experiment each SHAVE performs only one kind of operations. In other words, half of the SHAVEs enqueue and half dequeue elements to / from the data structure. "Random operations" means that each SHAVE has equal probability to perform either an enqueue or a dequeue each time it prepares its next operation.

In the rest of the subsection we present the performance experimental results. *mtx-(2-locks)* is the lock-based queue implementation with 2 locks, while *mtx-(1 lock)* is the same implementation with a single lock. *C-S* refers to the Client-Server implementation using shared variables, while *C-S-mtx* is the Client-Server implementation using mutexes for communication. *RCL* refers to the Remote Core Locking implementation using shared variables for passing messages between the clients and server. Finally, *RCL-SHV_FIFO* is the RCL implementation using SHAVEs' FIFOs for communication.

The execution time is displayed in Fig. 4 and Fig. 5 for dedicated SHAVEs and random operations respectively. *mtx-(2 locks)* performs better in both dedicated SHAVEs and random operations for 8 SHAVES, (up to 51% in comparison with the *RCL*), since it is the only implementation that provides the maximum possible concurrency for the queue data structure. All other implementations serialize the accesses to the shared data. *mtx-(1 lock)* doubles the execution time in comparison with the *mtx-(2 locks)*.

In respect with the message passing implementations, the *C-S* leads to poor performance, due to the extensive utilization of shared variables. However, *C-S-mtx* performs much better. We noticed that the bottleneck of *C-S* is the spinning of the server to the shared variable, while waiting for the *oper_fin* message from the client. In the *C-S-mtx* implementation the message is transferred through a mutex, which leads to much lower communication overhead in comparison with the shared memory variables. *RCL-SHV_FIFO* is another implementation with very low communication overhead, with only 12% more execution time than the *mtx-(2 locks)* in the experiment with 6

TABLE II. QUALITATIVE COMPARISON OF THE SYNCHRONIZATION ALGORITHMS

Algorithm	Pros	Cons
Lock-based using mutexes	• Hardware implemented, so it is quite fast. • Performs well in low contention.	• Becomes a bottleneck in high contention • Danger of deadlocks, in the case of complex synchronization.
C-S	• Platform independent • Low memory size.	• Low performance due to spinning in non-local shared variables. • Server is underutilized.
C-S-mtx	Communication using mutexes is fast and reliable.	Server is underutilized.
RCL-SHV_FIFO	FIFOs are hardware implemented, so communication has very low overhead.	Consumes more memory than the other implementations, since SHAVEs store elements in their dedicated memory. Pointers to the elements are stored in the concurrent queue.
RCL	SHAVEs spin only in variables stored in their dedicated memories, and LEON does not spin at all, so it is fast.	Same as RCL-SHV_FIFO

SHAVEs. Finally, the *RCL* implementation scales well and its performance is very close to the *mtx-(1 lock)* (it reaches 14.5% more execution cycles in comparison with *mtx-(1 locks)*). In this algorithm, we utilize shared variables in a very careful way: each SHAVE spins only on a variable that is stored in its local CMX slice (mostly when a SHAVE is waiting for the server to respond with the address of a dequeued element). In contrast with the C-S implementation, the spinning on a variable in a local CMX slice does not lead to performance degradation.

In both experiments, we notice in some implementations a high increase in performance from two to four SHAVEs (especially in *RCL*, *mtx-(2 locks)*). The reason is that in the case of two SHAVEs there are many small time intervals where no SHAVE accesses the data structure (because it prepares to the next operation to be performed). However, as the number of SHAVEs increases, there is always one SHAVE (or two in case of *mtx-(2 locks)*) that makes progress by accessing a critical section. On the other hand, in the case of the *mtx-(1 lock)* the performance remains almost flat in all experiments. This is because all operations are serialized, so even when there are only two SHAVEs, there is always one making progress. In *C-S* and *C-S-mtx*, however, even the small communication overhead between LEON and SHAVEs results in lower performance for two SHAVEs, in comparison with four.

B. *Discussion of the Experimental Results*

One important observation from the experimental results is that *RCL-SHV_FIFO* performs very close (or even better for small number of SHAVEs) in comparison with the *mtx-(2 locks)*, although it serializes the accesses to the critical section. Due to the very low communication overhead it provides, since no memory is utilized for synchronization, it outperforms the *mtx-(1 lock)* by up to 39%. Also, it provides very low execution time in the case of small number of cores (29.6% reduced execution time compared with the *mtx-(2 locks)* for 2 SHAVEs). This proves that message passing implementations can be a feasible solution, not only in the HPC, but also in the multi-core embedded systems domain. Platforms like Myriad1 that integrate a RISC core for controlling the chip data flow can favor the usage of synchronization algorithms based on the server-client model.

In all experiments, we stored the shared data structure in CMX memory. Although, CMX size limits the maximum memory size that the queue can occupy, it provides higher performance for all the evaluated implementations. However, if an application utilizes a larger shared queue, then it should be placed in DDR memory, with cost in execution time.

The utilization of shared variables for synchronization, although it is relatively platform independent, usually leads to poor performance. A core spinning in shared variables causes bus saturation and performance degradation. According to our experiments, the only case that shared variables do not lead to poor performance is when the spinning takes place to a core dedicated memory (as in the *RCL* case, which performs quite close to *mtx-(1 lock)* for 8 SHAVEs). However, this implies that the multi-core embedded platform avails dedicated memories, so the *RCL* cannot be considered entirely platform independent. On the other hand, the platform dependent implementations and especially the *RCL-SHV_FIFO* lead generally to high performance. Taking advantage of the hardware specifications for achieving fast communication is an important factor leading to efficient synchronization.

Table II summarizes the advantages and disadvantages of each algorithm we evaluated in this paper. Our goal was to transfer message passing synchronization algorithms from the HPC domain in multi-core embedded systems. All algorithms implemented and adapted to the hardware characteristics of the embedded platform. This tuning led to very satisfactory performance results, since some message passing algorithms perform close, or even better than the single lock queue implementation. Therefore, since in the near future the embedded systems will integrate even more cores and mutexes will become a bottleneck (as happened in HPC), message passing synchronization can be a feasible solution.

VI. CONCLUSION AND FUTURE WORK

In this work we evaluated a number of message-passing synchronization algorithms in an embedded platform and proved that message passing implementations are an efficient solution to the shared data synchronization issue. We intent to extend our work in three directions: First, we plan to evaluate the message passing implementations in terms of power consumption, which is a very important evaluation metric in the embedded systems domain. We can also evaluate more synchronization algorithms which are proven to perform well in the HPC domain (e.g. the Flat Combining [21]). Finally, in this work we tuned the synchronization algorithms to take advantage of the hardware characteristics of the Myriad1

platform. We consider very interesting the evaluation of the algorithms in other embedded platforms as well, with different specifications and identify the hardware metadata that affect the synchronization algorithms implementation.

ACKNOWLEDGMENT

The authors would like to thank Paul Renaud-Goud (Distributed Computing and Systems Research Group at Chalmers University) for his useful comments and suggestions on the paper and Anders Gidenstam (Distributed Computing and Systems Research Group at Chalmers University) for his help and support on running the synchronization algorithms on the Myriad1 platform.

REFERENCES

[1] D. Cederman, et al., "Lock-free concurrent data structures," in "Programming Multi-Core and Many-Core Computing Systems", Wiley Series on Parallel and Distributed, Wiley-Blackwell, ISBN: 978-0470936900.

[2] D. Cederman, et al., “A study of synchronization methods in commonly used languages and systems,” in Proc. IPDPS, 2013, pp. 1309-1320.

[3] ARM synchronization primitives:
http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.dht0008a/ch01s02s01.html.

[4] W. Wolf, “High-performance embedded computing: architectures, applications and methodologies,” Morgan Kaufmann, 2007.

[5] Wandboard ARM multi-core board: http://www.wandboard.org.

[6] AMD Opteron 6000 Series Embedded Platform:
http://www.amd.com/us/Documents/6000_Series_product_brief.pdf.

[7] D. Kaeli and D. Akodes, “The convergence of HPC and embedded systems in our heterogenous computing future,” in Proc. ICCD, 2011, pp. 9-11.

[8] F. Schon, et al., “On interrupt-transparent synchronization in an embedded object-oriented operating system,” in Proc. ISORC 2000, pp. 270-277.

[9] S. H. Kim, et al., “C-Lock: energy efficient synchronization for embedded multicore systems,” IEEE Transactions on Computers, Vol. 99, 2013.

[10] M. Monchiero, G. Palermo, C. Silvano, and O. Villa, “Power/performance hardware optimization for synchronization intensive applications in MPSOCs,” in Proc. Design, Automation and Test in Europe (DATE), vol.1, 2006.

[11] R. Rajwar and J. Goodman, “Speculative lock elision: enabling highly concurrent multithreaded execution,” in Proc. 34th Annual ACM/IEE Int’l. Symp. on Microarchitecture. IEEE Computer Sociaty, 2001, pp. 294-305.

[12] S. Sanyal, et al. “Clock gate on abort: towards energy-effient hardware transactional memory,” in Proc. IEEE Int’l Symp. on Parallel and Distributed Processing, 2009.

[13] H. Cho, B. Ravindran and E. D. Jensen, “Lock-Free synchronization for dynamic embedded real-time systems,” in Proc. Design, Automation and Test in Europe (DATE), vol.1, 2006.

[14] D. Petrovic, T. Ropars and A. Schiper, “Leveraging hardware message passing for efficient thread synchronization,” in Proc. 19th Symposium on Principles and Practice of Parallel Programming, 2014.

[15] J. Howard, et al., “A 48-core IA-32 message-passing processor with DVFS In 45nm CMOS,” In Proc. International IEEE Solid-State Circuits Conference Digest of Technical Papers, 2010.

[16] V. Gramoli, R. Guerraoui, and V. Trigonakis, “TM2C: A software transactional memory for many-cores,” In Proc. 7th ACM European Conference on Computer Systems, 2012.

[17] J. L. Abellan, J. Fernandez, and M. E. Acacio. “GLocks: efficient support for highly-contended locks in many-core CMPs,” In Proc. 2011 IEEE International Parallel and Distributed Processing Symposium, 2011.

[18] Movidius Ltd: http://www.movidius.com/.

[19] M. Hill and M. Marty, “Amdahl’s law in the multicore era,” Computer, vol. 41, no. 7, pp. 33-38, 2008.

[20] J. P. Lozi, F. David, G. Thomas, J. Lawall, and G. Muller, “Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications,” in Proc. USENIX Annual Technical Conference, 2012.

[21] D. Hendler, I. Incze, N. Shavit, and M. Tzafrir. “Flat combining and the synchronization-parallelism tradeoff,” in Proc. SPAA’10, pages 355-364, 2010.

[22] G. Bell, “Bell's law for the birth and death of computer classes,” Communications of the ACM, January 2008, Vol 51, No. 1, pp 86-94.

[23] P. Tsigas and Y. Zhang, “A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems,” in Proc. thirteenth annual ACM symposium on Parallel Algorithms and Architectures (SPAA), 2001, pp.134-143.

[24] M. Michael and M. Scott, “Simple, fast, and practical non-blocking and blocking concurrent queue algorithms,” in Proc fifteenth annual ACM symposium on Principles of Distributed Computing (PODC), 1996, pp.267-275.