Comput Sci Res Dev
DOI 10.1007/s00450-014-0261-0

SPECIAL ISSUE PAPER

Performance and power consumption evaluation of concurrent queue implementations in embedded systems

**Lazaros Papadopoulos · Ivan Walulya ·
Paul Renaud-Goud · Philippas Tsigas ·
Dimitrios Soudris · Brendan Barry**

© Springer-Verlag Berlin Heidelberg 2014

Abstract Embedded and high performance computing (HPC) systems face many common challenges. One of them is the synchronization of the memory accesses in shared data. Concurrent queues have been extensively studied in the HPC domain and they are used in a wide variety of HPC applications. In this work, we evaluate a set of concurrent queue implementations in an embedded platform, in terms of execution time and power consumption. Our results show that by taking advantage of the embedded platform specifications, we achieve up to 28.2 % lower execution time and 6.8 % less power dissipation in comparison with the conventional lock-based queue implementation. We show that HPC applications utilizing concurrent queues can be efficiently implemented in embedded systems and that synchronization algorithms from the HPC domain can lead to optimal resource utilization of embedded platforms.

L. Papadopoulos (✉) · D. Soudris
School of Electrical and Computer Engineering,
National Technical University of Athens, Athens, Greece
e-mail: lpapadop@microlab.ntua.gr

D. Soudris
e-mail: dsoudris@microlab.ntua.gr

I. Walulya · P. Renaud-Goud · P. Tsigas
Computer Science and Engineering, Chalmers University of Technology, Gothenburg, Sweden
e-mail: ivanw@chalmers.se

P. Renaud-Goud
e-mail: goud@chalmers.se

P. Tsigas
e-mail: philippas.tsigas@chalmers.se

B. Barry
Movidius Ltd., Dublin, Ireland
e-mail: brendan.barry@movidius.com

Keywords Multicore platforms · Concurrent data structures · Lock-free

1 Introduction

High performance computing (HPC) systems execute computationally demanding applications from a wide range of domains (engineering, bioinformatics, etc.). Until recently, the demand for high computational performance has been met with the increase of the number of general purpose processing cores, leading to the emergence of multicore homogeneous HPC platforms. The application developers try to extract high performance from parallelism by utilizing such architectures.

On the other hand, in the embedded systems domain the most common design paradigm is the integration of different kind of units on a single chip. CPUs, DSPs and even graphic accelerator units (GPUs) are integrated on a single die leading to the development of heterogeneous architectures. Such platforms dominate today the embedded systems market and they are utilized in smart-phones, tablets and other consumer devices.

Although HPC and embedded domains followed different architecture design paradigms in the past, it has been argued that in recent years they have started to converge. For example, the heterogeneous paradigm is recently adopted by the HPC systems. The integration of CPU and GPU in a single die has been utilized in general purpose mainstream platforms. Recent examples include Intel's Sandy Bridge [1] and AMD Kaveri architecture, which integrates 4 CPUs and 8 GPUs on a single chip [2]. Tegra K1 mobile processor by Nvidia, integrates an ARM Cortex CPU, along with a Kepler GPU, focusing on low power consumption [18]. Such heterogeneous architectures provide the opportunity to take advan-

Published online: 05 July 2014

tage of both the flexibility of execution of CPUs for irregular workloads and the high computational power of GPUs [3].

According to Bell's Law [4], 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 as the embedded systems. Indeed, we experience the trend of porting computational demanding applications from general purpose computers and HPC systems to embedded platforms. 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. Applications that until recently were executed only in general purpose computer systems, will be realized on high-end multicore embedded platforms. 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 [5]. Such platforms are utilized in high-end consumer embedded devices, such as smartphones and game consoles.

Another important point of the convergence between the two domains is related with the power consumption, which is a traditional design constraint in embedded systems. Although high performance has been so far the major issue in the HPC, power efficiency is now becoming a major concern, as well. It is argued that the total energy cost summed over a few years of operation of a large supercomputer facility can almost equal the cost of the hardware infrastructure [6]. Also, power is listed as one of the most important issues and constraint for future Exascale systems [7]. HPC can adopt low power solutions utilized in embedded systems.

To tackle with the low power issue, new HPC design paradigms have been proposed. A recently proposed solution for power efficient HPC is the development of HPC systems from low-power embedded solutions [6,8]. For instance, the nCore BrownDwarf supercomputer is composed of ARM A15 cores and C66x DSPs, focusing on low power computational performance [19]. Embedded systems follow the concept of System-on-Chip (SoC), where all the components are a part of the same module, reducing communication distances and hence power. Therefore, embedded platforms can be efficiently used as building blocks for HPC systems.

Towards this end, it is important to evaluate HPC algorithms and applications in modern embedded devices. Embedded systems impose constraints such as low CPU frequency, heterogeneous architectures and limited memory, which differ from the characteristics of the conventional HPC systems. Therefore, the evaluation of modern computational intensive applications in multicore embedded systems in terms of performance and power consumption is a step towards the utilization of low-power embedded platforms for the design of HPC systems.

In this work, we realize a set of concurrent queue implementations inspired from the HPC domain, on the embedded multicore Myriad platform [11]. The implementations are inspired from the Remote Core Locking (RCL) synchronization algorithm [9,10] implemented for HPC and they are evaluated on the Myriad platform in terms of performance and power consumption. We aim to examine how synchronization algorithms used in the HPC domain perform in embedded platforms. In other words, we evaluate how efficiently HPC applications utilizing concurrent data structures can take advantage of the embedded platform specifications and how they can be optimized taking into consideration the platform limitations. As stated earlier, due to the convergence of the two domains the shared memory synchronization issues exist in both, therefore the adoption of solutions from one domain to another can provide interesting results.

The rest of the paper is organized as follows: In Sect. 2 is presented the related work. Next, we provide a summary of the technical details of the embedded platform utilized in the context of this work. In Sect. 4 we describe the concurrent queue implementations and in Sect. 5 we present the experimental results in terms of execution time and power consumption. Finally, in Sect. 6 we draw our conclusions.

2 Related work

Since power consumption in HPC has become a major issue, some attempts have been recently made to utilize low power processors in an attempt to build more energy efficient systems. In BlueGene supercomputers the computational power comes from low-power embedded PowerPC cores and achieve satisfactory performance to energy ratio [8]. Tibidabo is an HPC cluster built from low power ARM multicore chips, with performance to energy ratio comparable to the BlueGene [6]. Such approaches have been proposed as a solution in reaching the implementation of an Exascale system, which is currently hindered by the power that such a system would consume [7]. Other approaches for achieving energy efficiency in HPC focus on the controlling of heating, cooling and power of supercomputer systems. For instance, Microsoft has developed Marlowe framework for data senders that migrates workloads during low utilization periods to turn off machines [12] and thus improve energy efficiency.

The application we evaluated in the context of this work is a concurrent queue. The synchronization algorithm used is inspired from the Remote Core Locking technique described in [9,10]. It is based on the utilization of a dedicated server that executes the critical sections of the application. Similar techniques are the Flat combining technique, which 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 [13]. Other related works from the HPC domain focus on Hardware Transactional Memory [14] and in mutual exclusion techniques, like the token-based messaging [15]. In respect with the concurrent queue implementations, several non-blocking queue designs have been proposed in the HPC domain. For instance, [16] describes a scalable non-blocking bounded queue for shared multiprocessor systems utilizing compare-and-swap primitive, dealing with the contention on shared variables and the ABA problem. Other concurrent queue implementations are the Michael-Scott's queue, where the queue is implemented as a singly-linked list and the head and tail pointers are modified by Compare-and-Swap operations [17].

3 Myriad platform description

Myriad is a 65 nm heterogeneous Multi-Processor System-on-Chip (MPSoC) designed by Movidius Ltd [11] 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. Myriad chip will be utilized in the context of Project Tango, which aims at the design of a mobile device capable of creating a 3D model of the environment around it [20].

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 (LEON3) 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) 128 bit VLIW cores with an instruction set tailored for streaming multimedia applications. The Myriad 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 CMX memory is accessible to all SHAVEs and to LEON processor, as well. The Stacked SDRAM memory of 64 MB is accessible through the DDR interface. (Stacked SDRAM will be referred as DDR in the rest of the paper). Finally, LEON has 32 KB dedicated RAM (LRAM).

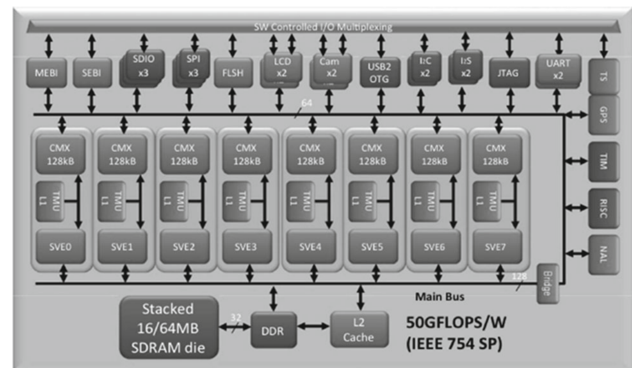

Fig. 1 Myriad architecture diagram: the RISC processor (LEON) and the 8 SHAVE cores. In each core a CMX memory slice is attached (TMU is the Texture Management Unit). The 64 MB SDRAM memory is depicted at the *bottom* of the diagram

Table 1 Memory access costs for LEON and SHAVEs

Memory	Size	LEON cost	SHAVE cost
LRAM	32 KB	Low	High
CMX	1 MB	Low	Low
DDR	64 MB	High, but low on data and cache hit	Low only for DMA cache hit

Table 1 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. 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, in comparison with the other CMX slices.

3.1 Synchronization and arbitration primitives

Myriad platform 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. Analysis of experiments on the Myriad platform shows that the mutex implementation is a fair lock with round-robin arbitration.

The Myriad platform avails a set of registers that can be used for fast SHAVE 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 can read from any entry. If a SHAVE attempts to write to a full FIFO, it stalls. Finally, LEON does not avail FIFOs. SHAVE FIFOs can be utilized to achieve 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 to use shared memory buffers.

4 Concurrent queue implementations description

In this section we describe the concurrent queue implementations we evaluated on Myriad platform in the context of this work. Concurrent queues are used in a wide range of application domains, especially in the implementation of pathfinding and work-stealing algorithms. The queue is implemented as a bounded cyclical array, accessed by all SHAVE cores. SHAVEs request concurrently access to the shared queue for inserting and removing elements.

Table 2 summarizes all different queue implementations we developed and evaluated on the platform. We used three different kinds of synchronization primitives: mutexes, message passing over shared variables and SHAVEs' FIFOs. Mutexes and SHAVEs' FIFOs were described in the previous section. In respect with the shared variables, we implemented communication buffers between the processors, used to exchange information for achieving synchronization. To reduce the cost of spinning on shared variables, we allocated these buffers in processor local memories, to avoid the congestion of the Myriad main buses.

The queue implementations can be divided in two basic categories: lock based and client-server. The lock-based implementations of the concurrent queue utilize the mutexes provided by the Myriad architecture. We implemented two different lock-based algorithms: 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()*. 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.

Table 2 Concurrent queue implementations: ("Y" indicates the ones that are evaluated in this work)

	Synchronization primitive		
	Mutex	Shared Var	SHAVE FIFO
No server	Y	–	–
Leon-srv-addr	–	Y	–
SHAVE-srv-addr	–	Y	Y
Leon-srv-h/t	–	–	–
SHAVE-srv-h/t	–	Y	Y

4.1 Client-server implementations

The client-server implementations are based on the idea that a server arbitrates the access requests to the critical sections of the application and executes them. Therefore, the clients do not have direct access to the critical section. Instead, they provide the server with the required information for executing the critical section. This set of implementations is an adaptation of the Remote Core Locking algorithm [9,10]. In Myriad platform the role of the server can be played either by LEON or a SHAVE core, as shown in Table 2. The SHAVEs are the clients, requesting access to the shared data from the server.

To maximize the efficiency of the client-server implementations, each SHAVE allocates the elements to be enqueued in its local CMX slice. Although the CMX is much smaller in comparison with the DDR memory, it provides much smaller access time for the SHAVEs than, for example, with the DDR. The server allocates the queue in a CMX slice, since LDRAM is much smaller.

We implemented two versions of the client-server synchronization algorithms. In the first one, the server maintains the FIFO order of the queue by storing the addresses of the allocated elements in a FIFO manner. In an enqueue, the client allocates the element in its local CMX slice and then sends the address of the element to the server, which pushes the address in the queue. In the dequeue case, the client sends a *dequeue* message to the server and waits for the server to respond with the address of the dequeued element.

Figure 2 illustrates this implementation with only two clients: *Client0* enqueues element *e5* in *CMX0* and sends the address to the server. The server pushes the address in the queue and notifies client that the enqueue has finished

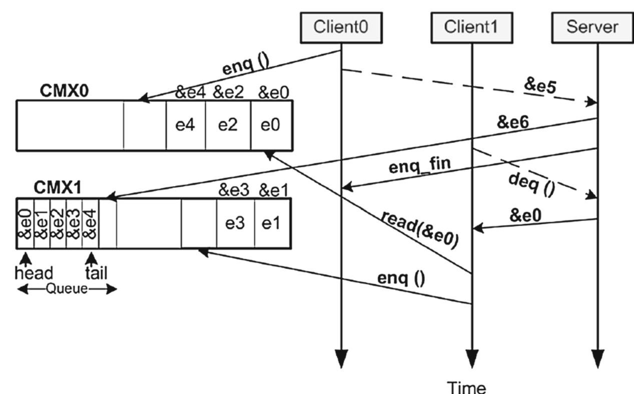

Fig. 2 Client-server implementation: the server maintains the order of the allocated elements by storing their addresses in a FIFO

with an *enq_fin* message. *Client1* requests a dequeue and the server responds with the address of the *e0* element.

We evaluated this algorithm by designing several alternatives: In the first one, the server is the LEON processor and in the second is a SHAVE. In Table 2 are displayed as *Leon–server–addr* and *SHAVE–server–addr* respectively. Also, we experimented with both shared variables and SHAVEs' FIFOs synchronization primitives.

The main advantage of this algorithm is that it reduces the stalling of the clients, especially during the enqueue operations. The client sends the address to the server and then can proceed with other calculations, without waiting for the server to respond. This applies especially in the case where the SHAVE FIFO synchronization primitive is used. The client stalls only when the server's FIFO is full. Additionally, the fact that the element allocation takes place only in local memories reduces both the execution time and the power consumption. Another parameter that affects the efficiency of the algorithm is the synchronization primitive used. We expect SHAVEs' FIFOs primitive to be efficient both in terms of performance and power consumption, since it avoids memory accesses during the exchange of information between the clients and the server. However, the main disadvantage in this case is that it can be only implemented using a SHAVE as a server.

In the second client-server implementation we altered the queue structure as follows: the server, instead of managing a queue to store object addresses, utilizes two pointers: *head* and *tail* that point at the first and the last element allocated respectively. Additionally, each element has a *next* pointer which points to the next element, keeping in this way the FIFO order. All these pointers are managed by the server, in order to improve the application parallelism by allowing the clients to perform tasks only outside the critical section.

When a client allocates an object in its local queue, it sends the address to the server, as in the first implementation. When the server receives the address, first updates the *next* pointer of the last element to point in the newly allocated element. Then, it updates the *tail* pointer to point to the new element. (This is the same that happens in a singly linked list FIFO data structure). In the dequeue, as soon as the server receives the request, sends to the client the address of the element pointed by the *head* and then updates the *head*, using the *next* pointer of the dequeued element.

To illustrate this algorithm, Fig. 3 shows an example. Initially, five elements exist in the queue. *e0* is the first allocated and *e4* is the last one. Therefore, head points to *e0* and *tail* to *e4*. *Client0* allocates element *e5* an element in *CMX0* and sends the address to the server. The server sets the *next* pointer of *e4* to point to *e5* and updates the *tail* pointer to *e5* as well. Then, sends an *enq_fin* message to *Client0*. *Client1* requests a dequeue. The server receives the message, updates the *head* pointer and sends the address of *e0* to the client.

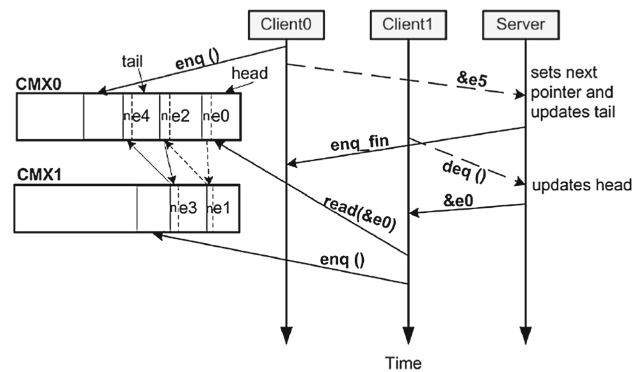

Fig. 3 Client-server implementation: the server maintains FIFO order of the allocated objects by pointing to the first and the last enqueued elements

In comparison with the previous one, this implementation consumes less memory space. Therefore, the space available in each local CMX slice is affected only by the number of allocated elements of the corresponding client, unlike the previous implementation where the queue of stored addresses reduced the available memory of the slice where it was allocated. It is important to mention that each client accesses only its local CMX slice during the enqueue and the dequeue operations. Only the server makes inter-slice accesses. The disadvantage of this implementation is that it cannot be efficiently implemented using LEON as a server, since it accesses the *next* pointers of each element with high cost. The implementation was designed using both shared variables and SHAVEs' FIFOs for communication between the server and the clients. In Table 2 is displayed as *SHAVE-srv_ ht*.

5 Experimental results

The concurrent queue implementations were evaluated using a synthetic benchmark, which 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. In the implementations where a SHAVE is utilized as a server, we run the experiments using up to six clients.

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 Myriad platform the data flow is controlled by LEON. SHAVEs start their execution when instructed to do 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. Power consumption was measured using a shunt resistor connected at the 5V

power supply cable. Using a voltmeter attached to the resistor's terminals we calculated the current feeding the board and therefore the power consumed by the Myriad platform.

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.

5.1 Execution time evaluation

In this 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. *leon-srv-addr* refers to the client-server implementation, where the server is LEON and stores the addresses of the objects in a queue, while *SHAVE-srv-addr* is the same implementation where a SHAVE is the server. *leon-srv-ht* and *shave-srv-ht* refers to the client-server implementation where the server (LEON and a SHAVE respectively) manages a head and a tail pointer to maintain the FIFO order. Finally, "shared-var" means that the communication is achieved using a shared buffer (i.e. shared variables) and "sf" means that the communication is made through the SHAVEs' FIFOs.

Figures 4 and 5 show the execution time of dedicated SHAVEs and random operations respectively. We notice that the *mtx-2-locks* implementation is the fastest one in the case of 8 SHAVEs and seems to scale well. This is expected, since it provides the maximum possible concurrency. It requires about half the number of execution cycles in comparison with the *mtx-1-lock*.

The SHAVEs' FIFOs implementations perform well, especially in the case of 4 SHAVEs in the random operations experiment (28.3 % in comparison to the mtx-2-locks). The utilization of SHAVEs' FIFOs for communication seems to be very efficient in terms of execution time. On the other hand, shared variables provide much lower execution time: For example, *shave-srv-addr-shared-var* leads to 53.3 % more execution cycles in comparison with the *shave-srv-addr-sf* in the dedicated SHAVEs experiment with 6 SHAVEs. Also, the implementations where the server maintains a head and tail pointer performs slightly better in most cases in comparison with the one where the server stores the addresses of the elements (up to 16.7 % in random operations for the 6 SHAVEs experiment).

In most implementations, we notice a very large drop in the execution time from 2 to 4 SHAVEs, due to the increase of concurrency. In other words, in the 2 SHAVEs experiments there are time intervals where no client requests access to the shared data. However, in case of four clients or more, there is always a SHAVE accessing the critical section. Since the workload is fixed, there is a large drop in the execution time compared to the 2 client experiment. However, since access to the critical section is serialized, the execution time drop for more than 4 SHAVEs is much smaller (e.g. in case of *mtx-2-locks*) or even non-existent (e.g. in *mtx-1-lock*).

Finally, in all experiments where a SHAVE is utilized as a server there is an increase in execution time from 4 to 6 SHAVEs. The reason for that is the overhead added by inter-slice accesses, which is larger than the decrease in execution time due to increased concurrency. The utilization of LEON as a server using shared variables for communication (i.e. *leon-srv-addr-shared-var*) is inefficient since two overheads are accumulated: LEON is accessing variables in the CMX memory (which is more costly in comparison with SHAVEs) and the spinning on shared variables for achieving communication.

5.2 Power consumption evaluation

As previously stated, power consumption was measured using a shunt resistor connected to the power supply of the platform. Figures 6 and 7 show the power consumption for dedicated SHAVEs and random operations. For the 8 SHAVEs experiment the most power efficient implementation is the lock-based with a single lock (6.25 % in dedicated SHAVEs in comparison with the *mtx-2-locks*). However, for a smaller number of clients, the SHAVEs' FIFOs implementations are the most power efficient. Indeed, power consumption drops up to 6.8 % for 6 SHAVEs in the dedicated SHAVEs experiment.

We notice that the lock-based implementation with a single lock consumes less power than the 2-lock implementation. This is due to the fact that the power consumption is affected by the number of SHAVEs accessing the memory concurrently. In the single lock implementation only one SHAVEs accesses the memory for performing operations. However, in the 2-locks implementation there are 2 SHAVEs which perform operations concurrently, while the rest are spinning on the locks. Therefore, the 2-lock algorithm consumes more power.

Spinning on a lock consumes very low power, because mutexes are hardware implemented. Microbenchmarking experiments show that 8 SHAVEs spinning on a lock concurrently, consume about 20 % less power in comparison with the case where 8 SHAVEs access the memory concurrently. In fact, this is the case with the shared variables synchronization primitive. All shared variable implementations consume a lot of power, because spinning on a memory location is energy inefficient, even if the spinning takes place in a local CMX slice.

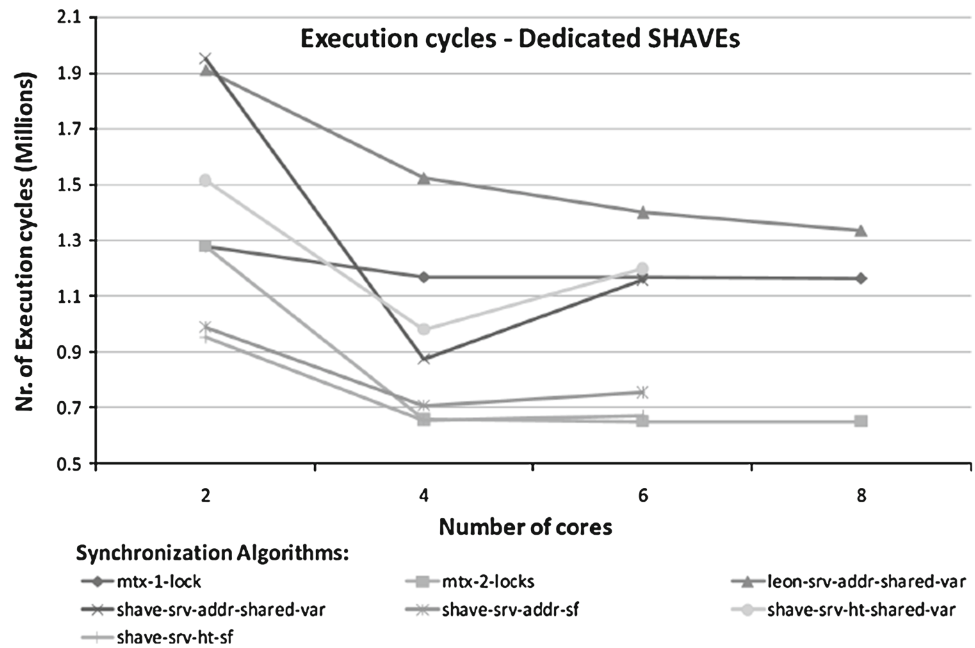

Fig. 4 Execution cycles for different synchronization algorithms, when half SHAVEs perform enqueue and half dequeue operations

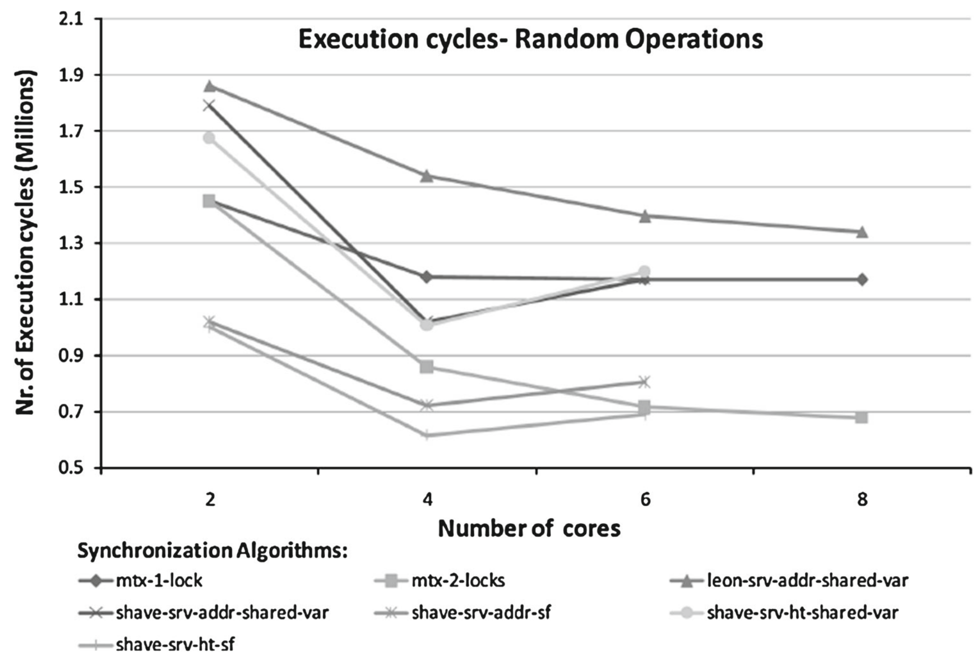

Fig. 5 Execution cycles for different synchronization algorithms, when the SHAVEs perform randomly enqueue and dequeue operations

SHAVEs' FIFOs communication method is the most energy efficient. When a SHAVE tries to write in a full FIFO or read from an empty FIFO stalls, until the FIFO gets non-full or non-empty respectively. Microbenchmarking experiments we performed show that 8 SHAVEs stalling in a FIFO consume about 28 % less power than spinning on a mutex. Indeed, stalling in FIFOs is common in our experiments, where the contention is high. The fact that this synchronization algorithm avoids spinning on memory locations and set SHAVEs to stall mode makes it very power efficient.

5.3 Energy per operation evaluation

To evaluate in more depth the synchronization algorithms, we present the energy per operation results in Figs. 8 and 9 for the dedicated and the random operations experiments respectively. The results are normalized to the *mtx-2-locks* calculated values. We notice that the RCL implementations that utilize the SHAVE's FIFOs for communication between the clients and the server consume in almost all cases less energy per operation than the *mtx-2-locks*. In the random operation experiment, *shave-srv-ht-sf* consumes 33 % less

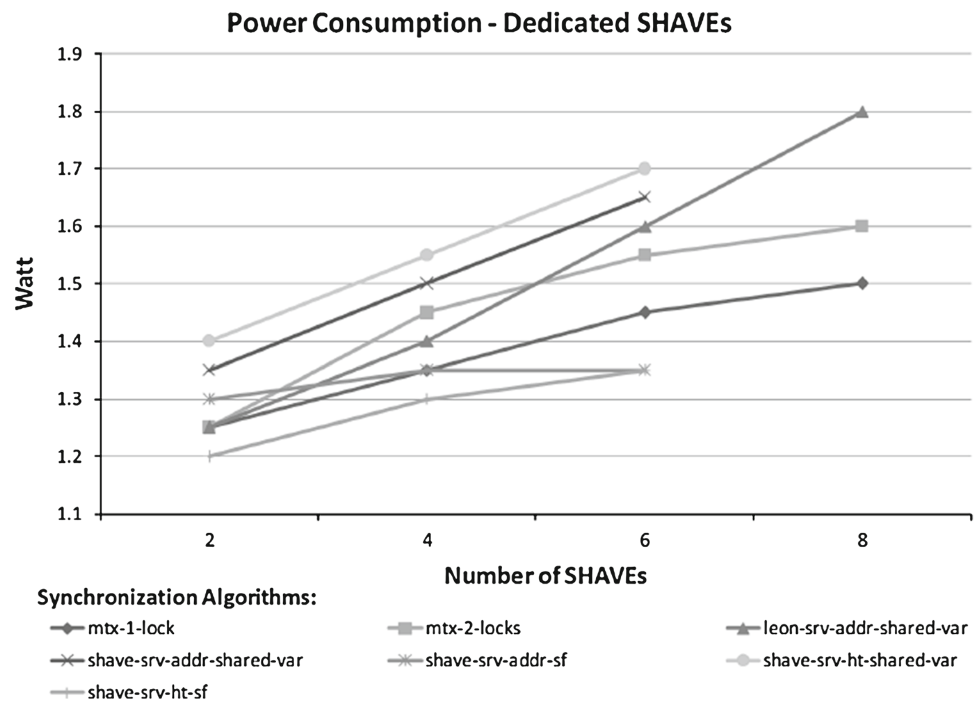

Fig. 6 Power consumption for different synchronization algorithms, when half SHAVEs perform enqueue and half dequeue operations

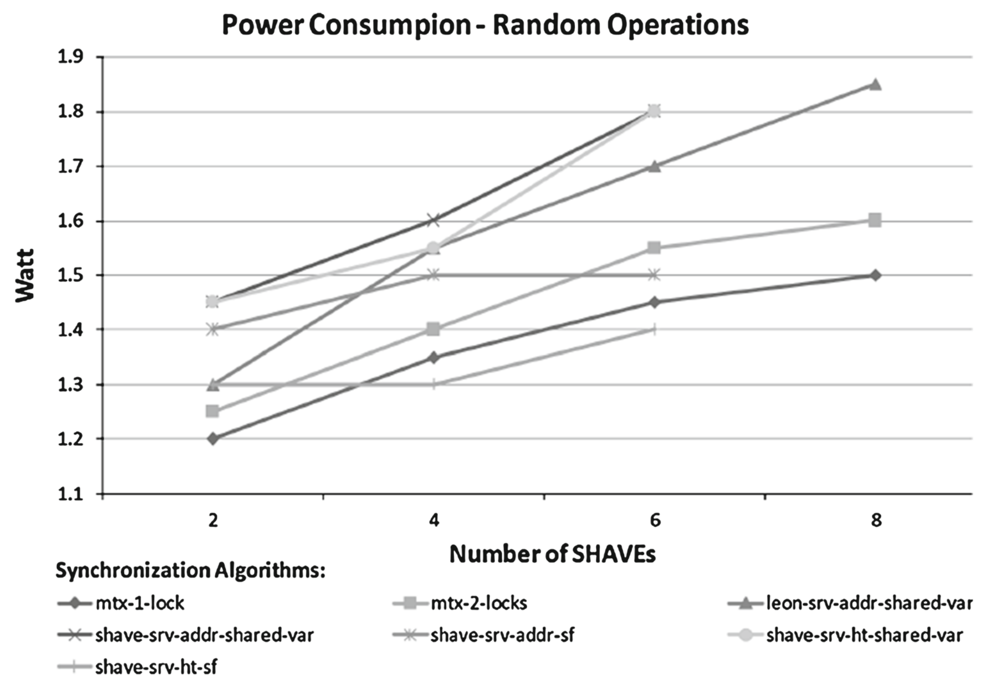

Fig. 7 Power consumption for different synchronization algorithms, when the SHAVEs perform randomly enqueue and dequeue operations

energy per operation in comparison with the *mtx-2-locks*. Indeed, when utilizing SHAVEs FIFOs instead of memory buffers for arbitration between the SHAVEs, the energy consumption is low. The shared buffer communication scheme is proven to be inefficient in terms of energy consumption. For instance, *leon-srv-addr-shared-var* consumes more than two times energy per operation in comparison with the *mtx-2-locks*.

5.4 Discussion

Mutex synchronization primitive is indeed efficient for the concurrent queue implementation in terms of both performance and power consumption (especially in the Myriad platform, where mutexes are hardware implemented and very optimized). However, our results show that RCL implementations provide very promising results for the queue imple-

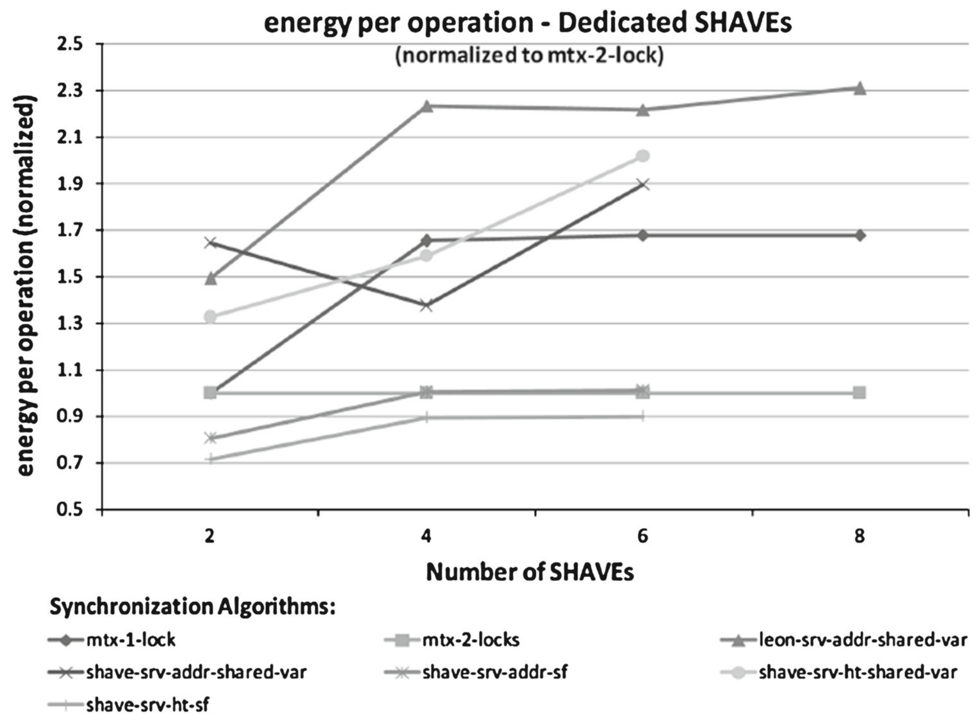

Fig. 8 Normalized energy per operation of the synchronization algorithms, when half SHAVEs perform enqueue and half dequeue operations

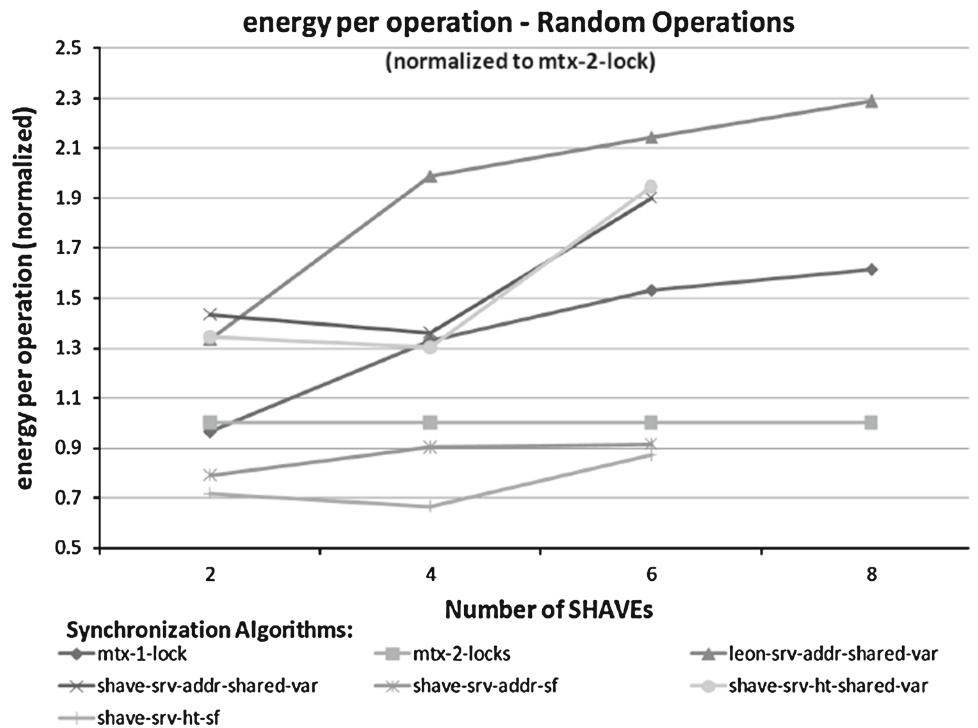

Fig. 9 Normalized energy per operation of the synchronization algorithms, when the SHAVEs perform randomly enqueue and dequeue operations

mentations and in most cases perform similar to the lock-based ones. We expect that in future MPSoCs, where the number of cores will increase even further, client-server implementations will become even more efficient.

The reason that the RCL implementations seem an attractive alternative to the lock-based ones is the fact that they transfer computational overhead of the critical sections from the cores, which are the queue workers (clients) to another dedicated core that plays the role of the server. The computational overhead of inserting and removing elements to/from the queue is transferred from the clients to the server. Therefore, while the server executes the critical section, the clients can proceed with other computations, thus increasing the parallelism and reducing the application execution time. Further-

more, by minimizing the communication overhead between the clients and the server (e.g. by utilizing the SHAVE's FIFOs), the results are very satisfactory. On the other hand, in the lock-based implementations, the computational overhead of accessing the queue elements is handled by the workers. However, in this case, simultaneous accesses to the data structure can be achieved, which is obviously not possible in the RCL algorithm. However, with these experiments we show that the RCL algorithm should be evaluated in embedded systems along with the lock-based solutions, especially in applications that use data structures which allow relatively low level of parallelism.

6 Conclusions

In this work we evaluated a set of concurrent queue implementations inspired from the HPC domain to an embedded platform. We showed that applications from the HPC domain utilizing concurrent queues can be efficiently ported to embedded systems. Also, this work proves that HPC synchronization algorithms can be utilized in the embedded systems and provide satisfactory results. We plan to extend our work by porting more synchronization algorithms, by taking advantage of other hardware based synchronization primitives, such as interrupts and DMA and evaluate our approach in real-world multithreaded applications.

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

References

1. Sandy Bridge (2013). http://ark.intel.com/products/64610/Intel-Xeon-Processor-E5-2450L-20M-Cache-100-GTs-Intel-QPI. Accessed 8 Apr 2013
2. AMD Kaveri (2013). http://www.amd.com/en-us/products/processors/desktop/a-series-apu. Accessed 1 Jun 2013
3. Yang Y, Xiang P, Mantor M, Zhou H (2012) CPU-assisted GPGPU on fused CPU–GPU architectures. In: Proceedings of the HPCA, pp 103–114
4. Bell G (2008) Bell's law for the birth and death of computer classes. Commun ACM 51(1):86–94
5. Wolf W (2007) High-performance embedded computing: architectures, applications and methodologies. Morgan Kaufmann, San Francisco
6. Rajovic N et al (2013) Tibidabo: making the case for an ARM-based HPC system. Future generation computer systems. Elsevier, Amsterdam
7. Shalf J, Dosanjh S, Morrison J (2011) Exascale computing technology challenges. In: Proceedings of the VECPAR'10, pp 1–25
8. Adiga NR et al (2002) An overview of the BlueGene/L supercomputer. In: Proceedings of the ACM/IEEE supercomputing conference, p 60
9. Petrovic D, Ropars T, Schiper A (2014) Leveraging hardware message passing for efficient thread synchronization. In: Proceedings of the symposium on principles and practice of parallel programming, pp 143–154
10. Lozi JP, David F, Thomas G, Lawall J, Muller G (2012) Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In: Proceedings of the USENIX annual technical conference
11. Moloney D (2011) 1TOPS/W software programmable media processor. HotChips HC23, Stanford
12. Guenter B, Jain N, Williams C (2011) Managing cost, performance, and reliability tradeoffs for energy-aware server provisioning. In: Proceedings of the INFOCOM, pp 1332–1340
13. Hendler D, Incze I, Shavit N, Tzafrir M (2010) Flat combining and the synchronization-parallelism tradeoff. In: Proceedings of the annual ACM symposium on parallel algorithms and architectures, pp 355–364
14. Gramoli V, Guerraoui R, Trigonakis V (2012) TM2C: a software transactional memory for many-cores. In: Proceedings of the ACM European conference on computer systems
15. Abellan JL, Fernandez J, Acacio ME (2011) GLocks: efficient support for highly-contended locks in many-core CMPs. In: Proceedings of the IEEE international parallel and distributed processing symposium
16. Tsigas P, Zhang Y (2001) A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. In: Proceedings of the annual ACM symposium on parallel algorithms and architectures, pp 134–143
17. Michael MM, Scott ML (1996) Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the PODC, pp 267–275
18. Nvidia Tegra K1 mobile processor (2014). http://www.nvidia.com/object/tegra-k1-processor.html. Accessed 6 Jan 2014
19. nCore supercomputer (2013). http://www.ncorehpc.com/browndwarf. Accessed 17 Jan 2013
20. Project Tango (2014). https://www.google.com/atap/projecttango. Accessed 21 Feb 2014

Lazaros Papadopoulos received his diploma in electrical and computer engineering at Democritus University of Xanthi, in Greece. He is now a Ph.D. student at National Technical University of Athens. His research interests include low power embedded systems design and optimization of embedded applications.

Ivan Walulya received his B.Sc. degree in electrical engineering in 2010 at Makerere University Kampala and an M.Sc. in Computer Systems and Networks in 2013 from Chalmers University of Technology. He is currently a Ph.D. student in the Networks and Systems division at the Department of Computer Science and Engineering at Chalmers. His main research interest is concurrent data structures, especially the energy aspects in multicore or many core processors.

Paul Renaud-Goud received his Msc degree in Artificial Intelligence in 2009 at the Université Paul Sabatier, France and his Ph.D. at the École Nationale Supérieure de Lyon, France, in 2012. After a post-doc in the Institut Mathématiques de Bordeaux, France, he is now a post-doc in the Department of Computing Science at Chalmers University of Technology, Sweden. His main research activities are around energy optimization, from scheduling in large-scale machines, to data-structures for multi- and many-core systems.

Philippas Tsigas received the BSc degree in mathematics and the Ph.D. degree in computer engineering and informatics from the University of Patras, Greece. He was at the National Research Institute for Mathematics and Computer Science, Amsterdam, The Netherlands (CWI), and at the Max-Planck Institute for Computer Science, Saarbrücken, Germany, before. At present, he is a professor in the Department of Computing Science at Chalmers University of Technology, Sweden. His research interests include concurrent data structures and algorithmic libraries for multiprocessor and many-core systems, communication and coordination in parallel systems, power aware computing, fault-tolerant computing, autonomic computing, and information visualization.

Dimitrios Soudris received his Diploma in Electrical Engineering from the University of Patras, Greece, in 1987. He received the Ph.D. Degree in Electrical Engineering, from the University of Patras in 1992. He was working as a Professor in Department of Electrical and Computer Engineering, Democritus University of Thrace for 13 years since 1995. He is currently working as Ass. Professor in School of Electrical and Computer Engineering, Department of Computer Science of National Technical University of Athens, Greece. His research interests include embedded systems design, reconfigurable architectures, reliability and low power VLSI design.

Brendan Barry received his B.Eng. degree in electrical and electronics engineering in the Dublin City University in 1993. He is now the VP of engineering in Movidius Ltd. in Dublin, Ireland.