A Message-Passing Microcoded Synchronization for Distributed Shared Memory Architectures

Zois-Gerasimos Tasoulas, Iraklis Anagnostopoulos, Lazaros Papadopoulos, and Dimitrios Soudris

***Abstract*—Implementation of concurrent data structures in architectures that provide limited synchronization primitives is a critical challenge. Typical lock-based implementations suffer from well-known problems such as poor scalability and unfairness. In this paper, we propose a client-server based synchronization model that can be applied in data structures with low level of parallelism for distributed shared memory many-core systems that support also message-passing communication. Additionally, we utilize a programmable hardware accelerator with appropriate application interfaces to overcome the performance-flexibility dilemma. Experimental results show that the proposed work performs 20× faster than the single lock model with 88× less idle cycles and 7× less power consumption.**

***Index Terms*—Distributed shared memory (DSM), message passing, microcoded synchronization, programmable hardware accelerator.**

I. Introduction

Modern computing and embedded systems are moving away from superscalar and multicore architectures and follow the many-core paradigm, which is characterized by the constant increase in the number of integrated processors (e.g., 48 [1] and 57–72 [2] cores). In order to exploit the presence of multiple computing components, efficient intercore communication and data management techniques are required. Network-on-chip has been established as the communication architecture for overcoming bottlenecks and provide efficient intercore data exchange. Additionally, the amount of integrated memory is increased as well occupying in some cases up to 85% of the whole chip [3] making distributed shared memory (DSM) systems a necessity in order to avoid performance bottlenecks [4]. In a many-core DSM platform, local memories consist of a private and a shared part, which are accessible under shared address space.

From the application perspective, many-core systems are not fully utilized. Modern applications are highly parallel, dynamic, compute intensive, and intercore synchronization significantly impacts performance. Regardless of the degree of parallelism and the underlying hardware, applications can suffer severe performance degradation without efficient synchronization mechanisms. Specifically, the throughput of Linux operating system may fall up to 50% on many-core platforms [5] while the per core throughput also decreases more than 10× [6] due to inefficient synchronization. This happens because current operating systems were designed for single-core architectures and still serialize many operations.

Manuscript received December 12, 2017; revised March 12, 2018; accepted April 15, 2018. Date of publication May 8, 2018; date of current version April 19, 2019. This paper was recommended by Associate Editor J. Cortadella. *(Corresponding author: Zois-Gerasimos Tasoulas.)*

Z.-G. Tasoulas and I. Anagnostopoulos are with the Department of Electrical and Computer Engineering, Southern Illinois University at Carbondale, Carbondale, IL 62901 USA (e-mail: zoisgerasimos.tasoulas@siu.edu).

L. Papadopoulos and D. Soudris are with the School of Electrical and Computer Engineering, National Technical University of Athens, 15780 Zografou, Greece.

Color versions of one or more of the figures in this paper are available online at http://ieeexplore.ieee.org.

Digital Object Identifier 10.1109/TCAD.2018.2834423

In this paper, we propose a synchronization model that can be applied in data structures with low level of parallelism (e.g., queues, stacks, and heaps) for DSM many-core systems that support also message-passing communication. The fundamental idea is that a simple well-designed protocol that synchronizes the accesses based on message-based communication between cores combined with a hardware dual-microcoded controller (DMC) [4] offering message-passing services over a DSM architecture can provide comparable or even better results than the corresponding typical lock-based synchronization and overcome the performance-flexibility dilemma. The presence of the hardware accelerator allows programmers to lighten the workload of the main processor and execute parts of programs and applications directly on the accelerator. The proposed model significantly reduces performance degradation while achieving fairness, high utilization and high power gains.

II. Related Work

There is high interest in the evaluation of different synchronization techniques and their impact on system's performance, especially in the case of many-core platforms. As presented in [5] and [7], nonscalable locks can severely degrade the performance of widely used operating systems and software applications. In [8], a remote core locking algorithm is presented, introducing the concept of client-server synchronization model as an alternative to the traditional single lock implementation. A detailed presentation of different aspects of synchronization techniques and a theoretical approach to multicore synchronization scalability is depicted in [9]. The author presents different ideas and approaches toward synchronization and analyzes why some techniques fail to scale after a certain number of cores. Unfortunately, many of the presented synchronization concepts are theoretical and their implementation is limited by the underlying hardware. Dice *et al.* [10] presented a flat-combining technique to construct hierarchical locks. This idea can be used to build many groups of clients (clusters) that publish their requests in local queues and these queues are later merged in a global request queue. In [11], a thorough comparison of synchronization techniques is presented. Specifically, the authors evaluate the performance of various synchronization models on different types of computer platforms and state the type of system that is appropriate to achieve better performance for the various models. Papadopoulos *et al.* [12] explored and evaluated the performance of message passing algorithms in embedded systems. Message passing can be an alternative to shared memory techniques and can provide solutions for certain architectures and configurations. Additionally, Morrison [9] showed that message passing synchronization solutions can scale for a large number of cores. Koutras *et al.* [4] and Anagnostopoulos *et al.* [13] utilized a programmable hardware accelerator by developing a microcode-based dynamic memory allocator on DSM systems. Specifically, the allocator presented in [13] was based on priority tables stored in the local memory of each node, as a single linked list, in order to reduce the cost of simultaneous single locks in the heap. Koutras *et al.* [4] utilized a lazily heap selection scheme based on try-locks to overcome the complexity of

generating priority tables for large number of cores. However, in this way more memory fragmentation was created.

Min and Eom [14] evaluated and compared many synchronization techniques using queue type data structures. However, they do not provide any results for the fairness of the synchronization techniques and their solution implies the existence of compare-and-swap (CAS) supporting hardware. In addition, Wu *et al.* [15] compared state-of-the-art synchronization algorithms but they are evaluated in a limited number of different data structures and the evaluation platform provides cache coherency. Also, they mainly focus on interthread synchronization. Finally, Gangwani *et al.* [16] although provided a lock free synchronization technique for multicore systems, their evaluation is based on a general purpose chip, their solution implies that the hardware supports CAS operations and requires extra hardware components to be present in the processor or in the directory module. The differentiators of the proposed model are manifold.

1) Exploit the presence of the DMC, with the usage of customized microcoded functions, for accelerating data management functions and hiding synchronization details.
2) Provide functionality in a distributed way over a DSM environment integrating message-passing services, transparent to the developer.
3) Core-to-core communication and operations of the server are implemented in the microcode level aiming for hardware performance but maintaining flexibility by keeping the standard interfaces.
4) Provide software flexibility (due to the transparency of the services with the hardware accelerator).

III. Microcode-Based Synchronization Model

The development of concurrent data structures provides various challenges, especially in systems with limited synchronization primitives. Assume that we have a DSM platform which is composed of processor-memory nodes interconnected via a packet-switched mesh network, as depicted in Fig. 1. Additionally, each node employs a DMC, a programmable hardware accelerator [3] which allows the programmer to implement custom microcoded functions and trigger them by corresponding C-level APIs. Specifically, DMC consists of two mini-processors. Mini-processor A which is responsible for inter-core tasks and memory accesses of the local core, and mini-processor B, responsible for accessing remote cores and serving shared memory requests by remote cores. The utilized platform follows the principle of industry-driven architectures [1], [17], [18] that adopt the DSM architecture with limited synchronization primitives (e.g., no CAS support).

The proposed synchronization model is based on the idea that a single core plays the role of the "server" and it is the only one that accesses the data structure directly. The rest of the cores that need to access the concurrent data structure are "clients." Instead of accessing the data structure they send requests to the server and wait for its response, if necessary. The server, as soon as it receives a request, performs the operation on behalf of the client and then sends a response to the client from which the request came from. In the proposed method, the data structure is initially stored in the local private memory of the server in order to save time as there is no need to translate memory addresses. If the server needs more space, then it utilizes the shared memory of the system (starting from its local shared) defined during the initialization of the system.

The steps of the proposed synchronization model are displayed in Fig. 1. The application developer calls in the C level the appropriate function to either add [insert()/push()] or remove [delete()/pop()] an element from the structure (❶). This call

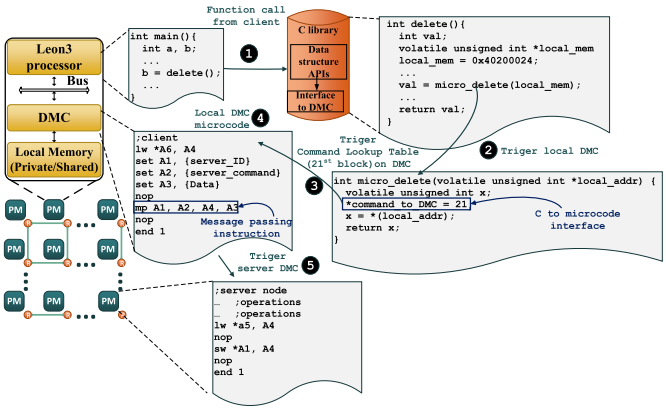

Fig. 1. DSM platform and proposed synchronization.

triggers our C library that contains the functions and the appropriate Processor-to-DMC interfaces for the used data structures. This function in its body contains a call to the microcoded function. At that point (❷), the microprocessor, attached to the core requesting a transaction (client), is notified and gets the control of the transaction. It is important to mention here that the microcoded functions are stored as command blocks in the control store of the accelerator and they are dynamically loaded by using a specific load command and the id of that block (❸). When the microcoded block has been loaded (❹), the mini-processor A fetches the microinstructions. Then, through the mini-processor B of the client, the communication with the server is initiated. Communication is based on message-passing, which means that the client sends a message to the server with its request and any additional required arguments. Specifically, message-passing communication was implemented with the usage of the **mp** `reg1, reg2, reg3, reg4` command. This message passing command allows to send a message directly from the mini-processor of a client core to the mini-processor of the server. The arguments of the command in order of presence are as follows.

1) The destination node (server) of the message.
2) The number of the microcoded block, stored in server's control store, to be executed.
3) The address where the returned data is expected.
4) The data of the client.

At that point (❺), the DMC of the server is triggered and starts executing the corresponding microcoded function (insert or delete). It is important to mention here, that all the actions of the server are performed by the hardware accelerator, at the microcode level, and not by the processor (C level). Particularly, according to the received message, the DMC of the server triggers the appropriate command block in the control store and performs the requested operation. Finally, when the client receives the extracted data, in case of `delete()/pop()`, it finalizes the execution of the microcode function and the control returns to the main processor.

Apparently, the server serializes all operations. However, the presented model has a number of advantages that compensate for the decreased parallelism that it provides. As depicted in Fig. 1, the concurrent data structure is initially allocated in the local private memory of the server, making the access to the data structure faster. Therefore, the number of memory accesses in remote memories is limited for both the server and the clients. Apart from the memory allocation issues, the primary reasons for performance improvement of the proposed synchronization method are as follows.

1) In the case of an `insert()/push()` operation, when the client sends its request to the server, there is no need to wait

TABLE I
TOTAL TIME PER REQUEST EXECUTION FOR ALL SYNCHRONIZATION MODELS

Synch. model	Total time
Single lock model	$T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rem} + T_{poll} * n_l + T_{com}}_{\text{acquiring lock}} + \underbrace{T_{LEON3} + T_{v2p} + \beta * T_{rsm} + (1-\beta) * T_{lsm}}_{\text{inserting/extracting element}}$
Client-server model [9]	$T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{checking if a request exists}} + \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{reading/writting requested element}} + \underbrace{T_{LEON3} + \gamma * T_{lpm} + (1-\gamma) * (\beta * T_{rsm} + (1-\beta) * T_{lsm})}_{\text{accessing the data structure}}$
Clustered client-server model [10]	$T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{checking if a request exists}} + \underbrace{T_{LEON3} + T_{v2p} + T_{rem} + T_{poll} * n_l + T_{com}}_{\text{acquiring lock}} + \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{reading/writting requested element}}$ $+ \underbrace{T_{LEON3} + T_{v2p} + \beta * T_{rsm} + (1-\beta) * T_{lsm}}_{\text{accessing the data structure}}$
DSM-sync model [11]	$T_{total} = \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{checking if a request exists}} + \underbrace{T_{LEON3} + T_{v2p} + T_{rsm}}_{\text{reading/writting requested element}} + \underbrace{T_{LEON3} + T_{v2p} + \beta * T_{rsm} + (1-\beta) * T_{lsm}}_{\text{accessing the data structure}}$
Proposed model	$T_{total} = \underbrace{T_{LEON3}}_{\text{initiating the microcoded model}} + \underbrace{T_{v2p} + T_{rem}}_{\text{accessing remote core}} + \underbrace{\gamma * T_{lpm} + (1-\gamma) * (\beta * T_{rsm} + (1-\beta) * T_{lsm}) + T_{csd} + a * T_{cds}}_{\text{accessing data structure}}$

for a response. Thus, in contrast with the typical lock-based implementations, an insert operation is getting blocked by an insert from another client, only if the server's mini-processor buffer is full and the message passing command blocks.

2) The reduction in the instruction overhead (the synchronization details are hidden from the high C-level).
3) The exploitation of the DMC for performing memory operations, thus alleviating the main processor's workload.

The proposed method can be applied to data structures with low level of parallelism (e.g., queues, stacks, and heaps), which are widely used and found in applications and operating systems, but it cannot be straight-forward applied effectively [11] to data structures that allow multiple-write and/or multiple-read operations at the same time.

IV. EVALUATION

The hardware platform used to implement the synchronization models is described in [3] and [4]. Each node consists of a LEON3 processor, a hardware accelerator DMC and memory, shared between the nodes. The nodes are interconnected by nostrum [19], a packet-switched mesh network, and see a continuous logical address space for the shared memory. To access the shared memory, nodes perform an address translation, accessing a lookup table to obtain the physical address and the number of the node holding this part of the memory.

To evaluate the proposed synchronization model, we compare it with four synchronization models that focus on pure or DSM systems and work at the C-level.

1) A coarse grain model of a single-lock.
2) A client–server model, called delegation model presented in [9].
3) A clustered client–server model, inspired by the idea of flat-combining presented in [10].
4) A modified DSM-sync model, presented in [11], with two h-factor values, 1 and 10, where each core acts as a server, in a round robin way, for h requests.

Each of these methods was implemented on the same hardware platform and to be fair, the DMC was used to accelerate memory operations in all cases. Additionally, we chose node (0,0) (first node in a mesh topology) as the server for our implementation and the one initializing the structure.

Table I presents the actions for each synchronization model. T_{LEON3} is the command execution time on LEON3 processor, including cache lookup time or time spent at the bus until reaching the microprocessor, while T_{v2p} is the time for virtual-to-physical address translation. T_{lsm} stands for the time to access local shared memory and T_{rsm} is the time to access remote shared memory. T_{rem} is the time to launch a remote read request, T_{poll} is the polling time for the lock, and n_l are the times that a core polls for it. T_{lpm} is the time to access local private memory. $T_{\text{com}} = T_{\text{csd}} + T_{\text{cds}}$ is the communication latency, where T_{csd} is the latency from source to destination and T_{cds} is the latency from destination to source. Parameter $\beta = 0$ means the structure is located in local shared memory, while $\beta = 1$ corresponds to remote shared memory and $\alpha = 1$ for memory read and 0 for a memory write. Last, $\gamma = 0$ means that the structure is in, local or remote, shared memory whereas for $\gamma = 1$ it is located in local private memory.

To evaluate and compare the proposed microcoded synchronization model, we utilized the data structures of stack, queue, deque [20], and binary max heap. The metrics we used are as follows.

1) The total execution time.
2) Progress, defined as the average number of cycles per request.
3) Core utilization, defined as the number of idle cycles of a core while waiting for its request to be completed.
4) Power gain, defined as the power consumption gain achieved over the single-lock.

As input, we utilized the benchmarks presented in [11], which consist of sequences of pairs of insertion and extraction operations.

Figs. 2(a)–5(a) depict total execution time. Beyond four cores, *the lock model experiences a breakdown in performance due to lock congestion* validating previous approaches [9]. We observe that the clustered client–server model performs better than the client–server model and many times it outperforms DSM-sync implementations. The clustered client–server model combines principles from the client-server and lock model and its efficiency is explained by the fact that clients are grouped under two servers and only these compete for the lock, leading to low congestion. The DSM-sync models have lower performance which is a result of server transitions, leading to wasted cycles. The microcoded model performs better than the rest of the synchronization techniques, for the queue with 14 cores, performs 3.7× faster than the DSM-sync model with h-factor = 10, and for 22 cores 2.2× faster than the clustered client–server model. This performance is achieved by utilizing the hardware accelerator for longer periods, bypassing main cores and using message passing to achieve communication between remote nodes.

Regarding progress, the microcoded model achieves gains compared to the other models. In Figs. 2(b)–5(b), we measured the

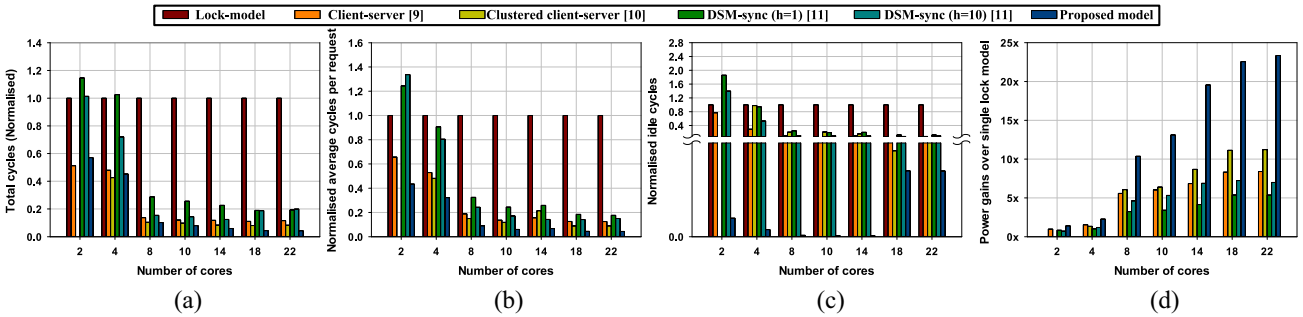

Fig. 2. Stack. (a) Normalised execution time. (b) Progress. (c) Utilization. (d) Power gains over single lock model.

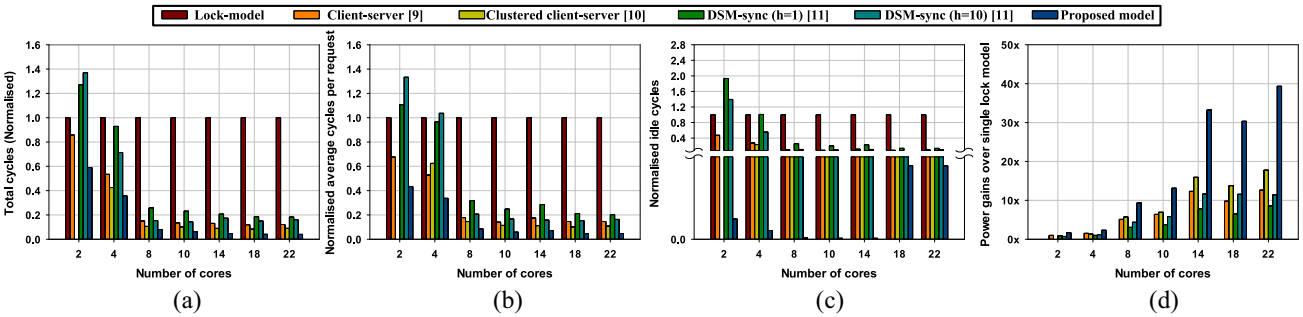

Fig. 3. Queue. (a) Normalised execution time. (b) Progress. (c) Utilization. (d) Power gains over single lock model.

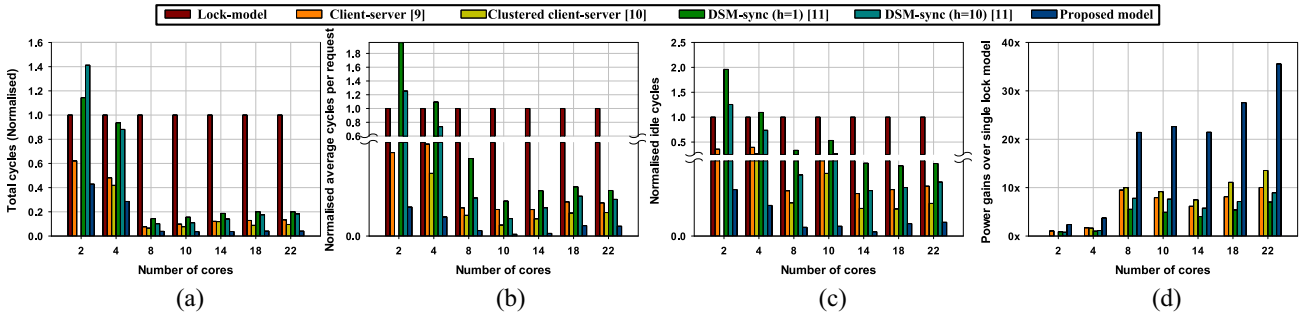

Fig. 4. Deque. (a) Normalised execution time. (b) Progress. (c) Utilization. (d) Power gains over single lock model.

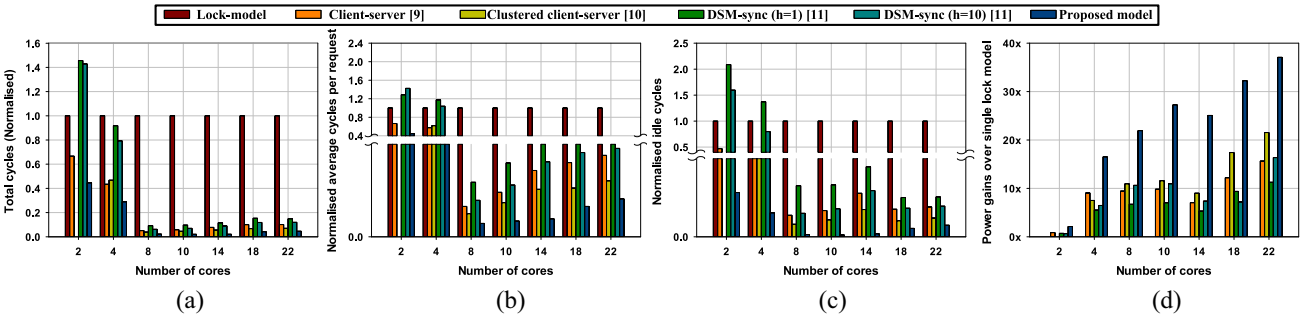

Fig. 5. Binary max heap. (a) Normalised execution time. (b) Progress. (c) Utilization. (d) Power gains over single lock model.

average number of cycles for each request. The baseline is the number of cycles for the single lock model, which is used in many conventional systems. The proposed method performs a request 1.39× faster than the DSM-sync model with h-factor = 10 for the stack and demonstrates even better results compared to the lock model.

The core utilization is presented in Figs. 2(c)–5(c), where the baseline is the number of cycles for the single lock implementation. The proposed microcoded model uses the message passing command, which saves cycles by bypassing the upper execution level and allows a direct communication with the node hosting the data structure. Specifically, for the deque structure and 14 cores, the microcoded model has 88× less idle cycles compared with the lock model, 9.8× compared with DSM-sync model and 5.6× less idle cycles than the clustered client–server model. Figs. 2(d)–5(d) present the power gains over the single-lock model. The proposed model demonstrates greater power gains due to the significant performance improvement and due to the fact that it utilizes the hardware accelerator for longer periods (processors remain idle). Specifically, the proposed model achieves

an average gain of 5× for stack, 5× for queue, 8× for deque, and 10× for binary max heap.

Overall, the proposed method achieves better results for the list-type structures and the tree-type structure, and provides promising performance, fair progress, and greater power gain. According to [11] developing fully nonblocking techniques for tree-type structures is a cumbersome task and requires support of advanced synchronization primitives. The proposed method offers an efficient alternative for these structures, does not require advanced synchronization support and imposes an area overhead of up to $351k$ NAND gates per node [3].

V. Conclusion

This paper presented an efficient, hardware-accelerated, scalable synchronization model for DSM systems. Specifically, this paper is a contribution toward the implementation of concurrent data structures in architectures that provide limited synchronization primitives support. Experimental results show that the proposed message-passing based client–server model provides increased performance, better throughput, better core utilization, and greater power gains even in cases of high contention.

References

[1] T. G. Mattson *et al.*, "The 48-core SCC processor: The programmer's view," in *Proc. ACM/IEEE Int. Conf. High Perform. Comput. Netw. Storage Anal.*, New Orleans, LA, USA, 2010, pp. 1–11.

[2] J. Jeffers, J. Reinders, and A. Sodani, *Intel Xeon Phi Processor High Performance Programming: Knights Landing Edition*. Amsterdam, The Netherlands: Morgan Kaufmann, 2016.

[3] X. Chen, Z. Lu, A. Jantsch, and S. Chen, "Supporting distributed shared memory on multi-core network-on-chips using a dual microcoded controller," in *Proc. DATE Conf.*, 2010, pp. 39–44.

[4] I. Koutras, I. Anagnostopoulos, A. Bartzas, and D. Soudris, "Improving dynamic memory allocation on many-core embedded systems with distributed shared memory," *IEEE Embedded Syst. Lett.*, vol. 8, no. 3, pp. 57–60, Sep. 2016.

[5] Y. Cui, Y. Wang, Y. Chen, and Y. Shi, "Experience on comparison of operating systems scalability on the multi-core architecture," in *Proc. IEEE Int. Conf. Cluster Comput. (CLUSTER)*, Austin, TX, USA, 2011, pp. 205–215.

[6] S. Boyd-Wickizer *et al.*, "An analysis of Linux scalability to many cores," in *Proc. OSDI*, vol. 10. Vancouver, BC, Canada, 2010, pp. 86–93.

[7] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich, "Non-scalable locks are dangerous," in *Proc. Linux Symp.*, 2012, pp. 119–130.

[8] J.-P. Lozi *et al.*, "Remote core locking: Migrating critical-section execution to improve the performance of multithreaded applications," in *Proc. USENIX Annu. Tech. Conf.*, 2012, pp. 65–76.

[9] A. Morrison, "Scaling synchronization in multicore programs," *Queue*, vol. 14, no. 4, p. 20, 2016.

[10] D. Dice, V. J. Marathe, and N. Shavit, "Flat-combining NUMA locks," in *Proc. 23rd Annu. ACM Symp. Parallelism Algorithms Archit.*, 2011, pp. 65–74.

[11] P. Fatourou and N. D. Kallimanis, "Revisiting the combining synchronization technique," *ACM SIGPLAN Notices*, vol. 47, no. 8, pp. 257–266, 2012.

[12] L. Papadopoulos, I. Walulya, P. Tsigas, D. Soudris, and B. Barry, "Evaluation of message passing synchronization algorithms in embedded systems," in *Proc. Int. Conf. Embedded Comput. Syst. Archit. Model. Simulat. (SAMOS XIV)*, Agios Konstantinos, Greece, 2014, pp. 282–289.

[13] I. Anagnostopoulos *et al.*, "Custom microcoded dynamic memory management for distributed on-chip memory organizations," *IEEE Embedded Syst. Lett.*, vol. 3, no. 2, pp. 66–69, Jun. 2011.

[14] C. Min and Y. I. Eom, "Integrating lock-free and combining techniques for a practical and scalable FIFO queue," *IEEE Trans. Parallel Distrib. Syst.*, vol. 26, no. 7, pp. 1910–1922, Jul. 2015.

[15] S. Wu, J. Zhang, Y. Peng, H. Jin, and W. Jiang, "Optimization strategies for inter-thread synchronization overhead on NUMA machine," in *Proc. IEEE 34th Int. Perform. Comput. Commun. Conf. (IPCCC)*, Nanjing, China, 2015, pp. 1–8.

[16] T. Gangwani, A. Morrison, and J. Torrellas, "CASPAR: Breaking serialization in lock-free multicore synchronization," *ACM SIGOPS Oper. Syst. Rev.*, vol. 44, no. 2, pp. 789–804, 2016.

[17] D. Moloney, "1TOPS/W software programmable media processor," in *Proc. IEEE Hot Chips 23 Symp. (HCS)*, Stanford, CA, USA, 2011, pp. 1–24.

[18] L. Benini, E. Flamand, D. Fuin, and D. Melpignano, "P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator," in *Proc. Design Autom. Test Europe Conf. Exhibit. (DATE)*, Dresden, Germany, 2012, pp. 983–987.

[19] M. Millberg, E. Nilsson, R. Thid, S. Kumar, and A. Jantsch, "The nostrum backbone-a communication protocol stack for networks on chip," in *Proc. 17th Int. Conf. VLSI Design*, 2004, pp. 693–696.

[20] H. Sundell and P. Tsigas, "Lock-free and practical deques using single-word compare-and-swap," Dept. Comput. Sci., Chalmers Univ. Technol., Rep. 2004-02, Mar. 2004.