Concurrent Data Structures in Architectures with Limited Shared Memory Support

Ivan Walulya, Yiannis Nikolakopoulos,
Marina Papatriantafilou, and Philippas Tsigas

Computer Science and Engineering
Chalmers University of Technology, Sweden
{ivanw,ioaniko,ptrianta,tsigas}@chalmers.se

Abstract. The Single-chip Cloud Computer (SCC) is an experimental multicore processor created by Intel Labs for the many-core research community, to study many-core processors, their programmability and scalability in connection to communication models. It is based on a distributed memory architecture that combines fast-access, small on-chip memory with large off-chip private and shared memory. Additionally, its design is meant to favour message-passing over the traditional shared-memory programming. To this effect, the platform deliberately does not provide hardware supported cache-coherence or atomic memory read/write operations across cores. Because of these limitations of the hardware support, algorithmic designs of concurrent data structures in the literature are not suitable.

In this paper, we delve into the problem of designing concurrent data structures on such systems. By utilising their very efficient message-passing together with the limited shared memory available, we provide two techniques that use the concept of a coordinator and one that combines local locks with message passing. All three achieve high concurrency and resiliency. These techniques allow us to design three efficient algorithms for concurrent FIFO queues. Our techniques are general and can be used to implement other concurrent abstract data types. We also provide an experimental study of the proposed queues on the SCC platform, analysing the behaviour of the throughput of our algorithms based on different memory placement policies.

1 Introduction

Shared concurrent data structures are necessary tools for inter-thread communication and synchronization. They provide multiple threads with access to shared data, guaranteeing the integrity of the latter. The commonly assumed model for providing algorithmic constructions of such data structures is the shared memory programming model, with a uniform address space and all threads having a uniform view of memory. In the literature there is a plethora of data structure implementations that guarantee high concurrency and resiliency (commonly through non-blocking guarantees, that avoid the use of locks [1, 2]). However, hardware that supports shared memory models does not scale very well with

L. Lopes et al. (Eds.): Euro-Par 2014 Workshops, Part I, LNCS 8805, pp. 189–200, 2014.

highly increasing numbers of cores. Furthermore, traditionally assumed cache-coherence is hard to achieve on many-core architectures. Consequently, from a hardware perspective, it is preferable to have a message-passing model, despite shared-memory programming being more desirable at the software level. Hence the high motivation for studying algorithmic implementations of concurrent shared data structures in alternative architectures and system models.

The Single-chip Cloud Computer (SCC) is a 48-core homogeneous experimental processor created by Intel Labs for the many-core research community [3]. The SCC is a hybrid system providing both shared and private memory, but favors message-passing over shared-memory programming, as the cores communicate using a low latency message passing architecture through the on-die mesh network. Neither hardware supported cache-coherence, nor atomic memory read/write operations across cores are provided. Shared memory access is manually instrumented while each of the 48 P54C cores is strongly memory ordered. Message-passing is ideal for transfer of small amounts of data due to the limited size of the message passing buffers, but not very efficient for large data transfers. In addition, replicating all datasets and exchanging them as messages is less efficient and more wasteful than using the data directly in shared memory.

Challenges in providing algorithmic data structure implementations in such a system model are summarized as follows: (i) the absence of universal atomic primitives (e.g. Compare&Swap) makes it hard or impossible to have fully non-blocking implementations [4] (ii) we need to construct an abstraction to the application layer that guarantees safety and provides liveness properties achievable within the bounds of the architecture.

We study the possibilities enabled by the configurable SCC architecture and propose a set of methods that address the above challenges for architectures that combine message-passing with shared-memory We propose that the message-passing operations not only can provide data-transfer, but also can be exploited for coordinating shared-memory access and synchronization of concurrent operations. We provide two techniques that build on this; one that uses the concept of a coordinator and one that combines spin locks with message passing. Our methods are inspired by ideas from shared-memory programming and distributed data structures; all three achieve high concurrency and resiliency properties. These techniques allow us to design three efficient algorithms for concurrent FIFO queues, while they can be also used to implement other concurrent abstract data types. The first, based on the blocking two-lock queue by Michael and Scott [2], provides fine-grained synchronization by using a distinct lock for the head and tail pointers of the queue, while uses the message passing to coordinate access to the data. Two more queue implementations are presented, that are based on leader based coordination by using the message passing communication.

The structure of the paper is the following. In section 2 we describe the model of the system and the studied problem. Section 3 presents the proposed implementations, while in section 4 correctness and progress guarantees are discussed. Section 5 presents an experimental evaluation and related work is discussed in section 6. Section 7 concludes the paper.

2 System Model and Problem Description

The SCC consists of a 6x4 mesh of dual-core tiles. The die has four on-chip memory controllers, making it capable of addressing up to a maximum of 64GB off-chip memory, in addition to 16Kb SRAM located on each tile. The off-chip memory is divided into segments private to each core and segments accessible to all cores. Non-uniform memory access (NUMA) applies to both on and off-chip memory. The chip features a 2D mesh network used to carry inter-tile communications by memory read/write operations. Packet routing on the mesh employs Virtual Cut-Throughswitching with X-Y routing [5]. Processing threads executing on the platform communicate by reading and writing to the on-chip shared SRAM, commonly referred to as the *message passing buffers (MPB)*. Data read from these buffers is only cached in L1 cache and tagged as MPB type (MPBT). MPBT cache lines can be explicitly invalidated. The platform deliberately provides neither hardware supported cache-coherence, thus shared memory accesses are manually instrumented while each of the cores is strongly memory ordered. Each core on the chip is assigned a Test&Set register, which can be used to build spin locks instrumental in constructing synchronization primitives or communication protocols. We refer the reader to [6] for more details on the platform.

The problem studied in this work is how to develop linearizable concurrent data structures for many-core systems that combine the shared memory and message-passing model, as the one in the SCC architecture outlined above — i.e. without universal atomic primitives — and provide to as high extent as possible liveness properties, throughput and scalability. The access methods to the shared data structures are invoked by units of execution (processes/threads). Implementations of such methods need to rely on the system's communication methods described above. The correctness of such data structures is commonly evaluated with safety properties that are provided by their method specifications. Linearizability [1] is a commonly used and desirable target: an implementation of a concurrent object is *linearizable* if every concurrent execution can be mapped to a sequential equivalent that respects the real time ordering of operations. Liveness properties of concurrent data structure implementations, such as non-blocking properties, are also desirable since they ensure system-wide progress as long as there are non-halted processes. In the stystem model under consideration, the absence of universal atomic primitives (e.g. Compare&Swap) makes it hard or impossible to have fully non-blocking implementations [4].

More specifically in the paper we present linearizable concurrent implementations of FIFO queues that also achieve liveness properties to the extent possible. Besides throughput, fairness is an important property of concurrent implementations. We follow the definition in [7], i.e. to study each core's i number of operations *ops* in relation to the total operations achieved. More formally, $fairness = \min\left\{\frac{\min(ops_i)}{average_i(ops_i)}, \frac{average_i(ops_i)}{\max(ops_i)}\right\}$. Values close to 1 indicate fair behavior, while lower values imply the existence of a set of cores being treated differently from the rest.

3 New Methods and Implied FIFO Queues with Limited Shared Memory

The main ideas in the methods we propose are the following: Our approach in ensuring mutual exclusion and thread coordination starts with implementing fine-grain locking mechanisms by exploiting local per core Test&Set primitives along with message passing based communication. The latter is used in order to enable the concurrent copying or accessing of data to the off-chip memory. We then present a client server model that further exploits the low latency communication and reduces blocking conditions. Moving towards a non-blocking algorithmic construction, we improve the last design by using acknowledgments that help reducing dependencies within concurrent operations.

3.1 Critical Sections over Local Spin Locks

We start by proposing the utilization of the local per core Test&Set in order to derive a method for spin locking over a mesh network and its use for an implementation of a concurrent queue. The queue design is based on the two-lock queue by Michael & Scott [2], implemented as a static memory array in shared off-chip memory. The two locks utilized, one for each of the head and tail pointers, allow an enqueue operation to proceed in parallel with a dequeue one. These pointers are stored in the fast access message-passing buffer (MPB).

Due to the NUMA architecture of the SCC, cores experience different latencies while accessing MPB or off-chip memory. With this in mind, we sought to increase parallelism by limiting the work inside the critical sections to updating the head or tail pointer; i.e. a thread holds a lock *only* for the duration of updating the respective pointer. This is achievable because the data structure is stored in statically allocated memory.

The general procedure for the two-lock queue methods is as described below. To enqueue an item to the list:

1. A core acquires the *enqueue_lock* or *tail_lock* and reads the current tail pointer. The pointer holds the offset to the next empty index in the array and not the last added item; thus it refers to a place holder in the queue.
2. The enqueuer increments the tail pointer and writes back the new value of the pointer to the MPB.
3. Then it releases the lock, allowing other processes to proceed and acquire the enqueue_lock, while enqueuer proceeds to add the data item to the address location read to the shared off-chip memory .

An item is logically added to the list as soon as the tail pointer position is incremented and written back, albeit the data item not having been added to the shared off-chip memory. There is a possibility of a dequeuer reading an item that is yet to be added to the queue. To prevent a dequeuer from reading a null item value, a flag bit is added to the data item to indicate that the latter has absolutely been enqueued to the list. To dequeue a data item:

1. A process acquires the *dequeue_lock*, reads the queue_t pointer, and checks if the head_offset is equal to the tail_offset.
2. If true, the queue is momentarily empty and the dequeuer releases the lock returning an appropriate value to the caller indicating an empty queue.
3. If the queue is not empty, the dequeuer reads the current queue's head offset. Then increments the head offset pointer, writes the new value back to MPB and releases the *dequeue_lock*.
4. After releasing the lock, it proceeds to check the node's dirty flag to confirm that the dequeued data item has been enqueued previously. If the flag is not set, the dequeueing processes spins on the flag until it is set. When this happens, the dequeuer can proceed and return the value of the node.

The enqueue/dequeue thread releases the lock before writing/reading data to the high latency off-chip memory. Early release of the lock implies that the critical section is reduced to incrementing the head or tail pointer indexes, however physically adding an item to the data structure is not included in the critical section.

Despite the blocking nature of the algorithm, we allow for multiple concurrent memory reads/writes to the off-chip shared memory, resulting in an optimized implementation that suits the SCC NUMA architecture. We also observe that since the platform does not support cache coherence, every request for the lock takes a cache miss thus generating network traffic. Network congestion has a significant impact on system performance, therefore as a result of the X-Y routing used , lock placement on the mesh is very imperative.

3.2 Critical Sections over Message Passing and Hybrid Memory

Memory contention and network congestion when many cores simultaneously try to acquire a lock are bound to degrade system performance and increase execution time within critical sections. To overcome problems associated with lock synchronization, we created a shared queue to which concurrent access is synchronized by message-passing. In this abstraction, one node (the server) owns the data structure, and all other nodes (the clients) request exclusive ownership to elements of the data structure at a given time.

Hybrid Memory Queue. The queue data structure resides in off-chip shared-memory, with the server holding pointers to the head and tail of the queue in its private memory. For a core to add an item to the queue, it must request for the position of the tail from the server. A node removing an item from the list also acquires the head position in a similar procedure. All this communication takes place in the MPB. The server handles enqueue/dequeue requests in a round-robin fashion polling over dedicated communication slots. The server process loops over the slots, reading requests, writing responses to the response buffers, and correspondingly exclusively updates the queue pointers. The clients spin over receive slots in their local MPB buffers until a response is received from the server.

We implement the message-passing mechanism utilizing receiver side placement i.e. the sender writes the data to the receiver's local MPBs. This way, both

the server and the clients spin on local memory while polling for incoming messages, thus reducing network traffic. To manage the message-passing buffers, the server allocates slots in its local MPB memory, one for each of the participating cores. Each client core can have only one outstanding request at a time. The server also requires a slot on the client core's local MPB memory for writing responses for the requesting core. After each client request is made, the client core spins on this receive buffer for a response from the server.

Particularly for an *enqueue*, the client sends a request for the queue tail-offset to the server and spins on a slot in its local MPB afterwards, waiting for a response. On receiving the enqueue request message, the server updates the queue tail pointer and responds with the previous offset to the requesting core. The enqueueing client then proceeds to add the data to the shared memory location pointed to by the offset and completes the operation by setting the flag on the memory slot to true. Similarly a *dequeue* operation starts with a request for the head offset to the server. The latter handles the request returning an appropriate offset. The client spins until the dirty flag on the memory slot is true before removing the data from the list, so that it does not try to dequeue an item from a stalled operation.

Hybrid Memory Queue with Acknowledgments. In the previous solution, the use of flags to indicate completion of an enqueue process may result in blocking of the dequeue method. This blocking could be indefinite if the enqueueing process fails before adding a node to the allocated memory location. To eliminate this and improve the concurrency and liveness properties of the method, we add an acknowledgement step to the enqueue procedure, where the enqueuer notifies the server after writing the data item to shared memory.

The server maintains a private queue of pointers to locations in the shared memory where data nodes have been enqueued successfully and the server has been acknowledged. Similarly to the previous design, the client requests for a pointer in the shared memory from the server via the MPB. The server replies as before and when acknowledged it adds the appropriate memory offset to its private queue of memory locations. Respectively a dequeuer is assigned with pointers only from this private queue, thus making sure it can proceed and dequeue the data item from shared memory without blocking. This implementation adds an extra message-exchange phase, nevertheless it solves problems arising from failed client processes during an enqueue method call.

Memory Management. The algorithms presented in this paper utilize statically allocated arrays in off-chip shared memory. Queues built with underlying arrays are bounded by the size of the array and algorithms have to address memory reclamation/reuse. One popular way of addressing the memory reuse problem is to deploy the array as a circular buffer. However, naive implementations of circular buffers run into an issue of array indices growing indefinitely. In our algorithms we use the approach of a linked list of arrays. In this approach, when the head pointer gets to the end of the current array, a new block of memory is allocated, and a pointer to this new block is added to the current block [8].

In our designs, head and tail pointers hold references to memory addresses and not indices, therefore we do not have to deal with indefinitely increasing array indices. On the grounds that shared memory is managed by the application and not the operating system, we can combine both mechanisms, that is circular buffer and linked list of arrays re-use the allocated static memory and resize the data structure respectively.

4 Correctness and Progress Guarantees

Followingly, limited to space constraints, we sketch the steps for proving the correctness of the implemented algorithms and also study their liveness properties.

Claim. All the three algorithms can be proved to maintain the following safety properties, by using induction, similar to the proof sketch of [2]. (i) The queue is a list of nodes that are always connected. (ii) New items are only added to the end of the list. (iii) Items are only removed from the beginning of the list (iv) Tail always points to the next empty slot in the list. (v) Head always points to the first element in the list.

Based on these properties now the following lemmas can be proved:

Lemma 1. *If a dequeue method call returns an item x, then it was previously enqueued, thus Enqueue(x) precedes Dequeue(x).*

Lemma 2. *If an item x is enqueued at a node, then there is only one dequeue method call that returns x and removes the node from the queue.*

The latter lemmas, along with the fact that all the data structure operations are serialized either by locks or by the server's request handling, imply the following:

Theorem 1. *The presented FIFO queues are linearizable to a sequential implementation.*

Moving to the liveness properties of the proposed implementations, the lock based queue is a blocking but deadlock and livelock-free implementation, since there exists no dependency when owning shared resources and the locks used are livelock-free. The message-passing based queue algorithms utilize a *Server-Client* model of interaction, in which all clients' progress depends on the server due to the request-response interactions to ascertain the values of the head and tail pointers. This dependence of all clients on the server forces us to present liveness properties assuming a fault-free server. Under this assumption and considering that the server handles requests in a round-robin fashion we can see that:

Lemma 3. *The server responds to every client request in bounded time.*

Message-passing based algorithm. Lemma 3 guarantees that an enqueue method's poll for memory allocation will be arbitrated eventually. Similarly for the dequeue operation's polling for the position of the head. However, the dequeue method

loops again, after receiving the pointer to the queue head and before reading the nodes value, in order to check that an item was successfully enqueued to the assigned slot. It fails to terminate this loop, only if the enqueue fails to successfully add data items to the allocated slot. Consequently, the dequeue operation would be blocked with respect to the corresponding enqueue operation. Other operations in the system will succeed though, despite this blocked dequeue operation. Thus this blocking is only at an enqueue-dequeue pair level and does not deter progress of other processes i.e. system progress. We propose the term "pairwise-blocking" to describe this behavior.

Definition 1 (Pairwise-blocking). *Pairwise blocking is the condition under which the progress failure of a process affects at most one other process in the system (its pair).*

In the context of the latter queue data structure, a dequeue operation can be pairwise-blocked to the respective enqueue operation. Consequently we can show:

Theorem 2. *Message-passing based algorithm is pairwise-blocking.*

Message-passing based algorithm with acknowledgments. To eliminate the possibility of blocked operations, the enqueue method call notifies the server when successfully adding a node to the data structure, which is the point when the node is logically added. In this way, the server maintains record of added nodes, and only allocates successfully enqueued memory slots to a dequeuer. The above along with lemma 3 assist in showing that the restrictions of pairwise-blocking have been eliminated.

5 Experimental Study

In this section, we comprehensively evaluate the throughput, fairness and scalability of the algorithms described in sections 3.1 and 3.2. We begin by giving a detailed description of our experimental setup. Then we present experimental results and an evaluation of the different concurrent queue implementations.

Setup and Methodolody. We implemented the algorithms on the Intel SCC platform running cores at 533MHz, mesh routers at 800MHz and 800MHz DDR3 memory. All algorithms and testing code are written in C and compiled with *icc* targeted for the P54C architecture. The version of icc used was only validated for GCC version 3.4.0, and an experimental Linux version 3.1.4scc Image loaded on each core. In the experiments, we treat each core as a single unit of execution and assume only one application thread/process running per core. We run the algorithms for 600ms per execution, with each core choosing randomly an enqueue or a dequeue operation with equal probability. We varied contention levels by adding "other work" or "dummy executions" to the experiments. The dummy executions consist of 1000 or 2000 integer increment operations. We also randomized the execution and duration of the dummy work on the different cores, as we compared the algorithms under different contention levels. *High*

contention is the execution in which no dummy work is done in between queue method calls, while as under *low contention* conditions we execute dummy work every after calling a queue method. We measured system throughput as the number of successful enqueue and dequeue operations per millisecond. Fairness, as introduced in Sec. 2, is used to analyze how well each core performs in relation to the other cores.

5.1 Experimental Results

Figure 1 presents the results of the study, showing the system throughput and the fairness of our algorithms. In this experiment, the locks were located on tile (x=5,y=0) and we used the core 0 on tile (x=0,y=0) as the server for the message-passing algorithms. Using one core as a dedicated server, we only run experiments up to 47 contending cores in the message-passing algorithms.

To evaluate the benefit of latency hiding in the two-lock data structure, we examined the performance of the data structure without the early release of the lock (henceforth MSW-Queue). From Fig. 1(a), we observe that the optimized implementation of the two-lock concurrent data structure with early release of the lock achieves higher system throughput than the MSW-Queue implementation under high contention. The figure also shows that message-passing based algorithms achieve less system throughput, however, with very high level of fairness to all participating cores.

Figure 1(b) shows that the fairness of the lock-based algorithms drops as we increase the number of cores due to increasing contention for the head and tail pointers. Because of the NUMA architecture and mesh network , this observation led us to investigate the throughput and fairness of the system as we vary the placement of the locks on the chip. We present the results of this experimentation later in this section. Under low contention system configuration, all algorithms give a linear improvement in system throughput with increasing number cores. One key observation was that the lock-based algorithms give almost identical throughput and fairness values.

Performance in Relation to Lock Location. In this experiment, we selected 5 cores scattered around the chip and measured the throughput of these cores as we increased the number of participating cores. The additional cores were also scattered uniformly around the chip so as to spread the traffic almost evenly on the mesh network. When running the experiments, we activated a single core on each tile until all the tiles hosted an active core, after which, we started activating the second core on every tile.

Additionally, we changed the positions of the lock and repeated the experiment, so as to investigate how the performance of the system varies with lock placement in the grid. XY routing [5] used on the SCC motivated the decision to perform these experiments because it does not evenly spread traffic over the whole network, resulting in a highest load in the center of the chip. This creates the need to find an optimal location for the locks so that the system performance does not degrade tremendously as we increase the number of executing cores.

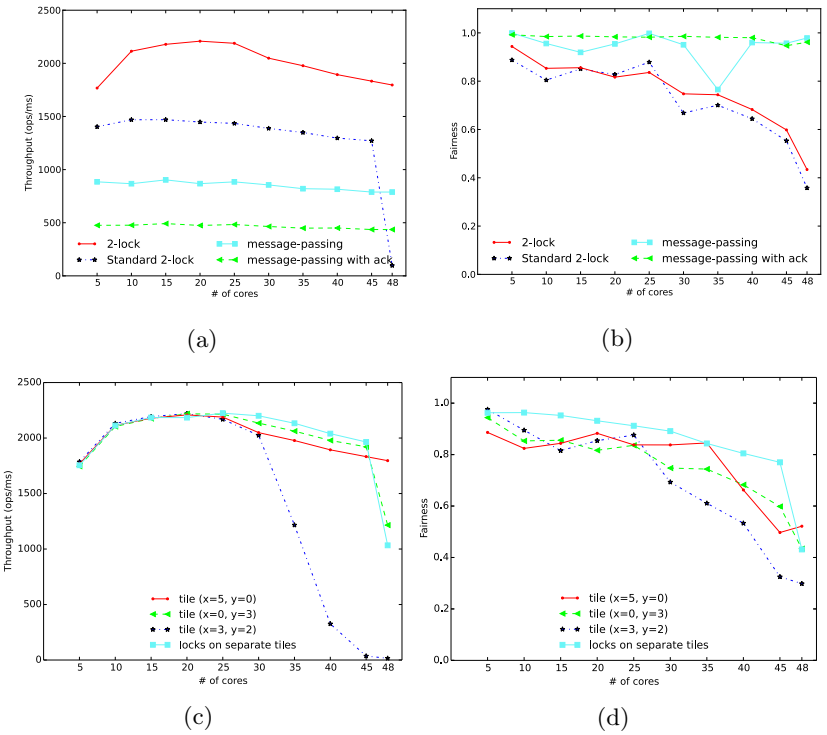
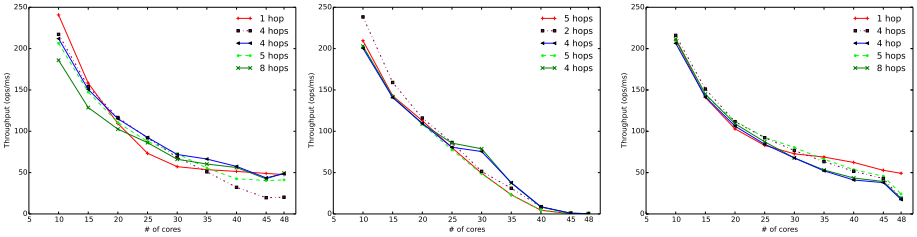


Fig. 1. Throughput and fairness at high contention of the different queue implementations (a, b) and the different lock positions for the lock based implementation (c, d)

We plot the results in Fig. 1(c) and 1(d). The degradation of performance varies with the location of the locks, with the highest performance degradation experienced when we place the lock at tile (x=3,y=2).

When the lock is placed in the middle of the chip, tile (x=3,y=2), the high contention for the lock creates very high traffic load in the middle of the chip, which degrades performance to almost a halt. With the network overloaded due to contention for the lock, the core holding the lock delays to release the lock. Contending cores cannot acquire it, continuing to spin. This creates a situation where the lock is not used, and thus no progress is achieved by the system. This deterioration in system throughput is evident in Fig. 1(c). Figure 1(d) presents the fairness values for the different lock positions, generally fairness decreases as we increase contention, but it is more paramount when the locks are positioned in the middle of the mesh.

We also performed the experiments under low contention configuration. In this case, congestion does not impact lock acquisition and release and there is a linear throughput improvement with the increase in participating cores. We further observed that placing the locks in the middle of the chip, we achieved very good throughput figures. This confirms the earlier intuition that network congestion delays release of the lock thus deteriorating system performance. We further analyzed the throughput of each core relative to its distance from the



(a) Tile (x=5,y=0) (b) (x=3,y=2) (c) (x=0,y=3)&(x=5,y=0)

Fig. 2. Core throughput with changes in lock location and an increasing number of executing cores at high contention

locks. For this analysis, we plot the throughput of each core against the number of executing cores. This allows us to observe how increasing contention affects the core's throughput with regards to its location and distance from the locks. Figure 2 shows the results from 5 cores and different lock placements as discussed previously. The performance of each core degrades with an increasing number of executing cores. It is also clear that for a low number of executing cores, the cores closer to the locks have a higher throughput. However, as we increase the core count, the throughput of these cores deteriorates, and in some cases (Fig 2(a)), it is worse than for cores further from the locks. To reduce the congestion on a single area of the chip for the 2-lock algorithm, we placed the two locks in different locations on the chip. In this case, the cores achieved almost identical performance as no single core has a much better latency advantage with regards to the distance from both enqueue and dequeue locks (Fig. 2(c)).

6 Related Work

Petrovic et al. [9] provide an implementation that takes advantage of hardware message-passing and cache-coherent shared memory on hybrid processors. They propose HYBCOMB, a combining algorithm in which threads elect a combiner that handles requests from other threads for operations to be executed in mutual exclusion. The identity of the combiner is managed using a shared variable that is updated using a CAS, thus not viable without atomic read/write instructions;

Delegation of critical sections to a dedicated server has also been considered in the design of a NUMA friendly stack [10]. Similarly migration of critical-sections was used to implement remote core locking [11]. In this work, a thread that wants to execute a critical section, migrates the critical section to a dedicated server thread. The general idea is to replace lock acquisitions with software remote procedure calls. All the above solutions require a shared memory model and cache-coherent hardware architecture. The emergence of non-uniform memory access many-core architectures with or without cache coherence is limiting the portability of concurrent data structures based on a shared memory model. Our algorithms can be implemented on new multi-core platforms without support for cache coherent shared memory model or universal atomic primitives.

7 Conclusion

The proposed implementations illustrate that constructing concurrent data structures on many-core processors without atomic read-modify-write primitives or uniform address space is a possible but non-trivial task. We further make a proposition that, for architectures that combine message-passing with shared-memory, the message-passing is not necessarily only essential for data-transfer but can also be exploited to coordinate shared-memory access and synchronization of operations on the different cores.

Acknowledgements. We extend our gratitude to Intel for access to the SCC. The research leading to these results has received funding from the European Union Seventh Framework Programme (FP7/2007-2013) under grant agreement No: 611183, EXCESS Project, www.excess-project.eu.

References

[1] Herlihy, M., Shavit, N.: The Art of Multiprocessor Programming. Morgan Kaufmann Publishers Inc., San Francisco (2008)
[2] Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing, PODC 1996, pp. 267–275. ACM (1996)
[3] J., Dighe, Howard, o.: A 48-core ia-32 message-passing processor with dvfs in 45nm cmos. In: 2010 IEEE International Solid-State Circuits Conference Digest of Technical Papers (ISSCC), pp. 108–109 (2010)
[4] Herlihy, M.P.: Impossibility and universality results for wait-free synchronization. In: Proceedings of the Seventh Annual ACM Symposium on Principles of Distributed Computing, PODC 1988, pp. 276–290. ACM, New York (1988)
[5] Zhang, W., Hou, L., others: Comparison research between xy and odd-even routing algorithm of a 2-dimension 3x3 mesh topology network-on-chip. In: WRI Global Congress on Intelligent Systems, GCIS 2009, vol. 3, pp. 329–333 (2009)
[6] Intel Cooporation: SCC External Architecture Specification (November 2010)
[7] Cederman, D., Chatterjee, B., et al.: et al.: A study of the behavior of synchronization methods in commonly used languages and systems. In: Proceedings of the 27th IEEE International Parallel & Distributed Processing Symposium (2013)
[8] Gidenstam, A., Sundell, H., Tsigas, P.: Cache-aware lock-free queues for multiple producers/Consumers and weak memory consistency. In: Lu, C., Masuzawa, T., Mosbah, M. (eds.) OPODIS 2010. LNCS, vol. 6490, pp. 302–317. Springer, Heidelberg (2010)
[9] Petrovic, D., André, Schiper, o.: Leveraging hardware message passing for efficient thread synchronization. In: 19th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming. Number EPFL-CONF-190495 (2014)
[10] Calciu, I., Gottschlich, J.E., Herlihy, M.: Using elimination and delegation to implement a scalable numa-friendly stack. In: Proc. Usenix Workshop on Hot Topics in Parallelism, HotPar (2013)
[11] Ozi, J.P., David, F., et al.: Remote core locking: migrating critical-section execution to improve the performance of multithreaded applications. In: Proc. Usenix Annual Technical Conf., pp. 65–76 (2012)