An Efficient Unbounded Lock-Free Queue for Multi-core Systems

Marco Aldinucci[1], Marco Danelutto[2], Peter Kilpatrick[3], Massimiliano Meneghin[4], and Massimo Torquati[2]

[1] Computer Science Department, University of Torino, Italy
aldinuc@di.unito.it
[2] Computer Science Department, University of Pisa, Italy
{marcod,torquati}@di.unipi.it
[3] Computer Science Department, Queen's University Belfast, UK
p.kilpatrick@qub.ac.uk
[4] IBM Dublin Research Lab, Ireland
massimiliano_meneghin@ie.ibm.com

Abstract. The use of efficient synchronization mechanisms is crucial for implementing fine grained parallel programs on modern shared cache multi-core architectures. In this paper we study this problem by considering Single-Producer/Single-Consumer (SPSC) coordination using unbounded queues. A novel unbounded SPSC algorithm capable of reducing the row synchronization latency and speeding up Producer-Consumer coordination is presented. The algorithm has been extensively tested on a shared-cache multi-core platform and a sketch proof of correctness is presented. The queues proposed have been used as basic building blocks to implement the FastFlow parallel framework, which has been demonstrated to offer very good performance for fine-grain parallel applications.

Keywords: Lock-free algorithms, wait-free algorithms, bounded and unbounded SPSC queues, cache-coherent multi-cores.

1 Introduction

In modern shared cache multi-core architectures the efficiency of synchronization mechanisms is the cornerstone of performance and speedup of fine-grained parallel applications. For example, concurrent data structures in multi-threaded applications require synchronization mechanisms which enforce the correctness of concurrent updates. They typically involve various sources of overhead which have an increasingly significant effect on performance with increasing parallelism degree and decreasing synchronization granularity.

In this respect, mutual exclusion using lock/unlock, is widely considered excessively demanding for high-frequency synchronisations [1]. Among other methods, *lock-free* algorithms for concurrent data structures are the most frequently targeted. These algorithms have been devised by way of a hardware-implemented class of atomic operations — so-called CAS, because of its paradigmatic member *Compare-and-Swap* — in order to avoid an explicit consensus that would increase the overhead for data accesses [2,3,4,5,6]. Unfortunately, CAS operations are not inexpensive since they might fail to swap operands when executed and may be re-executed many times, thus introducing other sources of potential overhead, especially under high contention [1]. Furthermore, without explicit consensus among parallel entities, the problem of correct memory management arises for dynamic concurrent data structures because of the complexity in tracking which chunk of memory is really in use at a given time. In general, lock-free dynamic concurrent data structures that use CAS operations should be supported by safe memory reclamation techniques in programming environments without automatic garbage collection [7].

In this work we study the synchronization problem for the simplest concurrent data structure: the Single-Producer/Single-Consumer (SPSC) queue. SPSC queues are widely used in many application scenarios: their efficiency can boost performance in terms of both latency and scalability to a non-negligible degree. In particular, SPSC-based synchronisations are used both in the implementation of high-level and completely general models of computation based on streams of tasks [8], and in a number of parallel frameworks as basic building blocks [9,10,11].

SPSC queues can be classified in two main families: *bounded* and *unbounded.* Bounded SPSC queues, typically implemented on top of a circular buffer, are used to limit memory usage and avoid the overhead of dynamic memory allocation. Unbounded queues are mostly preferred to avoid deadlock issues without introducing heavy communication protocols in the case of complex streaming networks, i.e. graph with multiple nested cycles. Bounded SPSC queues have been studied extensively since the emergence of the first wait-free algorithm presented by Lamport in the late 1970s [12]. More recently, some research work [13,14] revisited the Lamport queue, introducing a number of cache optimizations. On the other hand, unbounded SPSC queues, which are not any less relevant, have attracted less attention, resulting in quite a gap between the two SPSC families.

With the aim of filling this gap, we introduce and analyze here a novel algorithm for unbounded lock-free SPSC FIFO queues which minimizes the use of dynamic memory allocation. Furthermore, we provide a new implementation for the widely used dynamic list-based SPSC queue, along with proof sketches of correctness for both algorithms. Their performance is evaluated on synthetic benchmarks and on a simple yet relevant microkernel. The performance and the benefits deriving from the use of our SPSC queue when programming complete and complex application have already been assessed in [15,16].

The paper is organized as follows: Section. 2 provides the relevant background and related work discussing the reference implementations of the SPSC queue for shared-cache multi-cores. Section 3 introduces the list-based unbounded algorithm (dSPSC), while in Sec. 4 a novel algorithm for the unbounded queue (uSPSC) is presented, together with a proof sketch of its correctness. Performance results are discussed in Sec. 5. Section 6 summarizes the contribution of the work.

2 Producer-Consumer Coordination Using SPSC Queues: Background and Related Work

Producer-Consumer coordination is typically implemented by means of a FIFO queue, often realized with a circular buffer. Lamport proved that, under the *Sequential Consistency* (SC) memory model [12], a SPSC buffer can be implemented using only read and write operations [17]. Lamport's circular buffer is a *wait-free* algorithm, i.e. it is guaranteed to complete after a finite number of steps, regardless of the timing behavior of other operations. Another important class of algorithms are the *lock-free* algorithms, which enforce a weaker property than wait-free: they guarantee that at any time at least one process will make progress, although fairness cannot be assumed.

Lamport's circular buffer algorithm is no longer correct if the SC requirement is relaxed. This happens, for example, in all architectures where write-to-write memory ordering ($W \rightarrow W$ using the notation of [18]) is relaxed, i.e. two distinct writes at different memory locations may be executed out of program order (as in the *Weak Ordering* memory model [18]). A few modifications to the basic Lamport algorithm allow correct execution even under weakly ordered memory consistency models; they have been presented first and proved formally correct by Higham and Kavalsh [19]. The idea behind the Higham and Kavalsh queue basically consists in tightly coupling control and data information into a single buffer operation by extending the data domain with a new value called BOTTOM, which cannot be inserted into the queue. The BOTTOM value can be used to denote an empty cell, and then used to check if the queue is empty or full without directly comparing the indexes of the queue's *head* and *tail*.

Ten years later Giacomoni et al. [13] followed a similar line by proposing the same basic algorithm and studying its behavior in cache-coherent multiprocessor systems. As a matter of fact, Lamport's queue results in heavy cache invalidation/update traffic because both producer and consumer share both head and tail indexes[1]. This can be avoided, as already noted in [19], by using a BOTTOM value that makes it possible for the producer to write and read only the tail and for the consumer to write and read only the head indexes. Since this technique applies nicely to data pointers where NULL is the BOTTOM value, Giacomoni et al. proved that on weakly ordered memory model, a Write Memory Barrier (WMB) is actually required to enforce completion of the data write by the producer before the data pointer is passed to the consumer[2]. Figure 1 presents an implementation of the SPSC algorithm proposed in [13] which may be regarded as the reference algorithm for bounded SPSC queues.

Avoiding cache-line thrashing due to false-sharing is a critical aspect in shared-cache multiprocessors and thus much research has been focused on trying to minimize this effect. In [13] the authors present a *cache slipping* technique suitable for avoiding false sharing on true dependencies (i.e. pointers stored within

[1] The producer updates the tail index, the consumer updates the head index, and both the producer and the consumer read both head and tail indexes.

[2] WMB is also referred to as store-fence.

```
bool push(void* data) {
  if (buf[pwrite]==NULL) {
    WMB(); // write−memory−barrier
    buf[pwrite] = data;
    pwrite+=(pwrite+1>=size)?(1−size):1;
    return true;
  }
  return false;
}

bool pop(void** data) {
  if (buf[pread]==NULL)
    return false;
  *data = buf[pread];
  buf[pread]=NULL;
  pread+=(pread+1>=size)?(1−size):1;
  return true;
}
```

Fig. 1. SPSC circular buffer implementation as proposed in [13], where `buf` is an array of size `size` initialized to `NULL` values

queue cells) and for enforcing partial filling of the queues in such a way that producer and consumer operate on different cache lines.

A different approach for optimizing cache usage, named *cache line protection*, has been proposed in MCRingBuffer [14]. The producer and consumer thread update private copies of the head and tail indexes for several iterations before updating a shared copy. Furthermore, MCRingBuffer performs batch update of control variables, thus reducing the frequency of writing the shared control variables to main memory. A variation of the MCRingBuffer approach is used in the Liberty Queue [20]. The Liberty Queue shifts most of the overhead to the consumer end of the queue. Such customization is useful in situations where the producer is expected to be slower than the consumer.

Unbounded SPSC queues have not benefited from a similar optimization effort and, to the best of our knowledge, have been approached only through the more general and more demanding CAS-based Multiple-Producer/Multiple-Consumer (MPMC) queues.

3 Basic Unbounded List-Based Wait-Free SPSC Queue

A way to design a SPSC queue is to use as a starting point the well-known two-lock Multi-Producer/Multi-Consumer (MPMC) queue described by Michael and Scott (MS) [6]. The MS queue is based on a dynamically linked list of `Node`(s) data structure, using head and tail pointers which (both) initially point to a dummy `Node` (i.e. containing NULL values). The `Node` structure contains the actual user value and a *next* pointer. Concurrency between multiple producers is managed by a lock for enqueue operations and symmetrically consumers use a different lock for dequeue operations.

Inspired by the MS queue, we propose a new unbounded SPSC queue whose algorithm is sketched in Fig. 2 (where lines §2.10 and §2.23 can be safely ignored here at the moment as they introduce a further optimization that is described later in this section)[3]. The `push` method allocates a new `Node` data structure, fills it and then adjusts the tail pointer to point to the current `Node`. The `pop` method gets the current head `Node`, places the data values into the application buffer, adjusts the head pointer and, before exiting, deallocates the head `Node`.

[3] We use the §M.N notation to reference line N from the pseudo-code in Fig. M.

```
 1 struct Node {
 2   void*       data;
 3   struct Node* next;
 4 };
 5 Node* head,* tail;
 6 SPSC cache;

 8 bool push(void* data) {
 9   Node* n;
10   if (!cache.pop(&n))
11      n = (Node*)malloc(sizeof(Node));
12   n->data = data; n->next = NULL;
13   WMB(); // write-memory-barrier
14   tail->next = n; tail = n;
15   return true;
16 }

18 bool pop(void** data) {
19   if (!head->next) return false;
20   Node* n = head;
21   *data = (head->next)->data;
22   head = head->next;
23   if (!cache.push(n)) free(n);
24   return true;
25 }
```

Fig. 2. Unbounded list-based dSPSC queue implementation with Node(s) caching. The list is initialized with a dummy Node.

In general, one of the main problems with the list-based implementation of queues is the overhead associated with dynamic memory allocation/deallocation of `Node` structures. To mitigate the overhead, it is common to use a data structure as cache, where elements are kept for future fast reuse, instead of being deallocated [21].

For a more tailored optimization, the specific allocation pattern can be taken into account: the producer only allocates while the consumer only frees nodes. To take advantage of this pattern, we add a bounded wait-free SPSC queue implementing a `Node` cache, which is used to sustain a "return" path from consumer to producer of `Node` structures that can be reused.

The introduced optimization clearly moves allocation overhead outside the critical path at the steady state. The resulting algorithm, called *dSPSC*, is shown in Fig. 2; line §2.10 and line §2.23 introduce the proposed cache optimization.

Along with some standard definitions we now provide, for the presented dSPSC, a sketch proof of FIFO queue correctness and the lock-freedom property.

Definition 1 (Correctness). *Assuming that simple memory read and write operations are atomic, a SPSC queue is defined as* correct *if it always exhibits FIFO behavior for any interleaving of push and pop operations.*

Note that the condition that simple memory reads and writes are atomic is typically satisfied in any modern general-purpose processor for aligned memory word loads and stores.

Theorem 1 (dSPSC). *Under a weak consistency memory model, the dSPSC queue defines a correct lock-free SPSC FIFO queue if a lock-free allocator is used.*

Proof (Sketch). In a sequentially consistent model, correctness of the dSPSC derives trivially from correctness of the two-lock MS queue (where the two locks have been removed as there is no concurrency between producers or between consumers) and of the bounded SPSC queue used for the `Node` cache [13].

Moving to a weak memory model, the bounded SPSC queue is still correct ([13]) while the memory barrier at line §2.13 guarantees correctness regarding the dynamic linked list management. Indeed, all changes to the structure of `Node n` as well as to the memory pointed by `data` have to be committed to memory before the node itself can be visible to the consumer (i.e. before the tail is set to point at the new node). It is trivial to see that, similar to what happens with the SPSC queue in [13], no other store fence is required inside the push and pop methods under weakly ordered memory model.

Concerning the lock-free property, the strategy that is used by dSPSC for the memory management is lock-free because the allocator is lock-free by hypothesis and the SPSC used as a cache is lock-free by construction. As the rest of the dSPSC algorithm does not present any statement where producer or consumer can block, progress is guaranteed. □

4 Fast Unbounded Lock-Free SPSC Queue

The SPSC algorithm [19,13], described in Sec. 2, is extremely fast (see Sec. 5) but implements a bounded queue. The dSPSC algorithm, presented in Sec 3, is lock-free and realizes an unbounded queue, but pays for the flexibility achieved via a list-based implementation with decreased spatial locality in cache behavior. However, the two approaches can be combined in a new lock-free algorithm for the unbounded SPSC (called *uSPSC*) inheriting the best features of both. The new algorithm is sketched in Fig. 3. The basic idea underpinning uSPSC is the nesting of the two queues. A pool of SPSC bounded queues (called *buffers* from now on) is linked together into a list as a dSPSC queue. The implementation of the pool of buffers aims to minimize the impact of dynamic memory allocation/deallocation by using a fixed-size SPSC queue as a freelist as in the list-based dSPSC queue.

The unbounded queue uses two pointers: `buf_w` which points to the writer's buffer (i.e. the "tail" pointer), and `buf_r` which points to the reader's buffer (i.e. the "head" pointer). Initially both `buf_w` and `buf_r` point to the same buffer.

The push method works as follows: the producer first checks whether the current buffer is not full (line §3.4), and then pushes the data. If the current buffer is full, it asks the pool for a new buffer (line §3.5), adjusts the `buf_w` pointer and pushes the data into the new buffer.

The pop method, called by the consumer, first checks whether the current buffer is not empty and if so pops data from the queue. If the current buffer is empty, there are two possibilities: a) there are no items to consume, i.e. the unbounded queue is really empty; b) the current buffer is empty (i.e. the one pointed by `buf_r`), but there may be some items in the next buffer.

If the buffer is empty for the consumer, it switches to a new buffer releasing the current one to be recycled by the buffer pool (lines §3.14–§3.16). From the consumer viewpoint, the queue is really empty when the current buffer is empty and both the read and write pointers (`buf_r` and `buf_w`, respectively) point to the same buffer. If the read and writer queue pointers differ, the consumer has to re-check the current queue emptiness because in the meantime (i.e. between the execution of instructions §3.11 and §3.12) the producer could have written

```
 1 int size = N; //SPSC size

 3 bool push(void* data) {
 4   if (buf_w->full())
 5     buf_w = pool.next_w();
 6   buf_w->push(data);
 7   return true;
 8 }

10 bool pop(void** data) {
11   if (buf_r->empty()) {
12     if (buf_r == buf_w) return false;
13     if (buf_r->empty()) {
14       SPSC* tmp = pool.next_r();
15       pool.release(buf_r);
16       buf_r = tmp;
17     }
18   }
19   return buf_r->pop(data);
20 }

22 struct Pool {
23   dSPSC inuse;
24   SPSC cache;

26   SPSC* next_w() {
27     SPSC* buf;
28     if (!cache.pop(&buf))
29       buf = allocateSPSC(size);
30     inuse.push(buf);
31     return buf;
32   }
33   SPSC* next_r() {
34     SPSC* buf;
35     return (inuse.pop(&buf)? buf : NULL);
36   }
37   void release(SPSC* buf) {
38     buf->reset(); // reset pread and pwrite
39     if (!cache.push(buf)) deallocateSPSC(buf);
40   }
41 }
```

Fig. 3. Unbounded lock-free uSPSC queue implementation

some new elements into the current buffer before switching to a new one. This is the most subtle condition whose occurrence must be proved to be impossible since, if the consumer switches to the next buffer while the previous one is not really empty, a data loss will occur. In the next section we prove that the `if` condition at line §3.13 is sufficient to ensure correct execution.

Theorem 2 (uSPSC). *The uSPSC unbound queue given in Fig. 3 is correct (according to Def. 1) on architectures with Weak Ordering consistency model (and therefore with any stricter ordering).*

Proof (Sketch). The SPSC and the dSPSC queues used as building blocks have been proven to be correct under Weak Ordering (WO) consistency and stricter models (e.g. Total Store Ordering) in [19,13] and Theorem 1, respectively.

We distinguish four cases with respect to the values of `buf_r` and `buf_w` used by producer and consumer (respectively): `buf_r` and `buf_w` are 1) equal or 2) different throughout execution of a push/pop pair; 3) they are different and become equal; or 4) they are equal and become different. In case 1) uSPSC is correct because of the correctness of the underlying SPSC. In case 2) uSPSC is correct because the producer and consumer work on different SPSC buffers and correctness follows from the correctness of the underlying dSPSC. In case 3 the consumer catches up with the producer: correctness follows because, when `buf_w` was assigned (line §3.5), that assignment will have been preceded by the issue of a WMB within the dSPSC push that commits all values of the previous buffer (allowing them all to be read by the consumer before it advances `buf_r`).

Case 4 is more subtle: here the two buffers are equal and become different when the producer observes that `buf_w` is full and prompts a move to a new write buffer. The concern is that, because of WO, in the case where the SPSC buffer size = 1 the consumer may see the buffer as empty and release it *before* a write to it has been committed, thus causing data loss. We prove that this cannot happen and the FIFO ordering is preserved.

Under WO model the consumer may be aware that the producer has changed the write buffer only after a synchronization point that enforces program order between two store operations has been traversed. In our algorithm the synchronization point is the WMB. In fact, the new value of `buf_w` (line §3.5) and the value that is written to the buffer might appear in memory in any order or not at all. Thus it might be thought possible that the reading buffer `buf_r` could still be perceived as empty while a new writing buffer has already been started (thus `buf_r` $\neq$ `buf_w`); the condition at line §3.13 could therefore evaluate to `true` even if the previous buffer is not actually empty. This condition could lead to data loss because the consumer might overtake and abandon a buffer still holding a valid value. In the uSPSC this subtle case can never arise however, because, in order to change the write buffer, a push operation in the dSPSC queue is called (§3.30) thus enforcing a WMB, which commits all previous writes to memory, and so the if condition at line §3.13 is evaluated to `true` only if the consumer buffer is really empty. FIFO ordering is trivially enforced by the FIFO ordering of both nested queues dSPSC and SPSC queues. □

It is worth noticing that, regardless of the implementation of the pool used in the uSPSC queue, if $size > 1$ (line §3.1) the two conditions `buf_r` incorrectly perceived empty and `buf_r` $\neq$ `buf_w`, cannot hold together as at least two push and one WMB must occur to make an empty queue become a full queue.

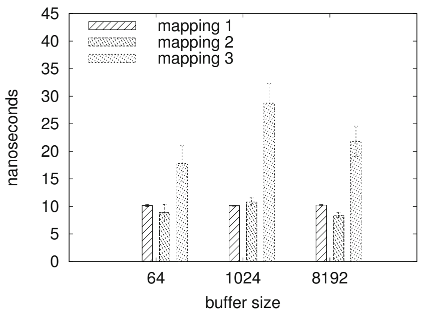
Corollary 1 (lock-free). *The uSPSC queue is lock-free provided a lock-free allocator is used.*

Proof (Sketch). The SPSC queue, and the dSPSC queue coupled with a lock-free allocator, are lock-free. Suppose we use a lock-free allocator in the `allocateSPSC` and in `deallocateSPSC`. As the `push` and `pop` methods contain no cycle nor can they block on any non lock-free function, progress is assured. □

Enhancing the queues to wait-freedom property. It can be demonstrated that the SPSC queue proposed in [13] as well as the dSPSC when a wait-free allocator is used, are both wait-free. The uSPSC is wait-free if a wait-free dSPSC queue is used and if a wait-free allocator is used in the pool: in fact both the push and pop methods complete in a bounded number of steps.

5 Experiments

All experiments reported in this section have been conducted on an Intel workstation with 4 eight-core double context Xeon E7-4820 @2.0GHz with 18MB L3 shared cache, 256K L2, and 24 GBytes of main memory with Linux x86_64. Some of the tests presented have been executed also on a different architecture and results can be found in [22]. Similar results to those presented in this paper have been obtained on the AMD Opteron platform.



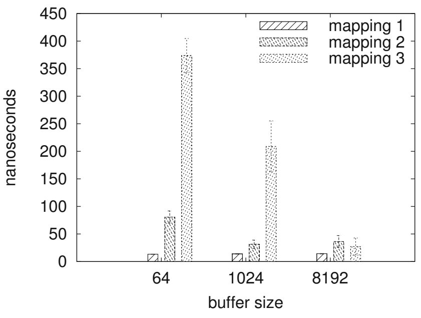
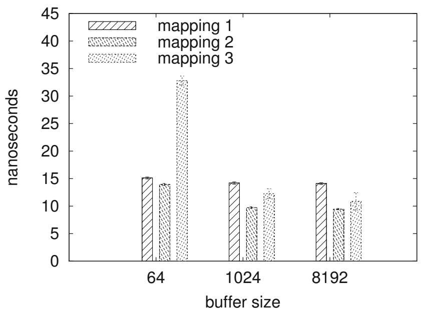


Fig. 4. Bounded SPSC (left) and unbounded dSPSC (right) average latency time in nanoseconds varying the internal buffer size and cache size respectively

The first test is a two-stage pipeline in which the first stage (P) pushes 1M tasks (a task is just a memory pointer) into a FIFO queue and the second stage (C) pops tasks from the queue and checks for correct values. Neither additional memory operations nor additional computation is executed. With this simple test we are able to measure the raw performance of a single push/pop operation by computing the average value of 100 runs, varying the buffer size for the bounded SPSC queue and the cache size for the dSPSC queue. We tested three distinct cases that differ in terms of the physical mapping of the two threads corresponding to the two stages of the pipeline. The first and the second stage of the pipeline are pinned: i) on the same physical core but on different HW contexts (`mapping1`); on the same CPU but on different physical cores (`mapping2`); on two cores of two distinct CPUs (`mapping3`).

Figure 4 reports the values obtained by running the first benchmark for the SPSC queue and the dynamic list-based dSPSC queue, varying the buffer size and the internal cache size, respectively. Fig. 5 (left) reports the values obtained by running the same benchmark using the unbounded uSPSC queue.

The bounded SPSC queue is almost insensitive to buffer size in all cases. It takes on average 8–12 ns corresponding to almost 16–24 cycles per push/pop operation with standard deviation less than 1.5 ns when the producer and the consumer are on the same CPU, and takes on average 16–36 ns if the producer and the consumer are on separate CPUs. The dSPSC queue is instead quite sensitive to the internal cache size on the tested architecture. The best values for the dSPSC queue range from 14 to 36 ns with a standard deviation that ranges from 0.5 to 11 ns. Such values are obtained with sufficiently large cache size (8192 slots). As expected, the bigger the internal cache, the better the performance obtained. As a reference, the MS queue implementation is one order of magnitude slower, going from 110–190 ns on sibling cores to 430–490 ns on non-sibling cores. The uSPSC queue (Fig. 5 left) is more sensitive to the internal buffer size in the case where the producer and the consumer are pinned to separate CPUs and when the internal buffer is small. The values obtained for the uSPSC



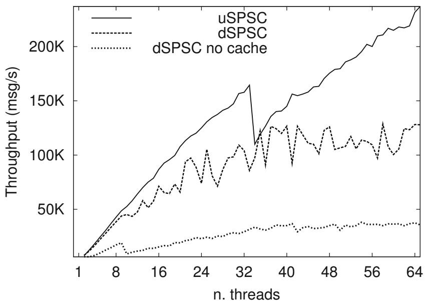


Fig. 5. Unbounded uSPSC average latency time in nanoseconds (left) varying the buffer size of the internal SPSC queues (pool cache size set to 32). Throughput in msgs/s (right) running the ring microkernel when using the dSPSC queue without any cache, the dSPSC with a cache size of 2048 and the uSPSC queue with an internal buffer size of 2048 elements (pool cache size set to 32).

are very good if compared with those obtained for the dSPSC queue, and are almost the same (or better) if compared with the bounded SPSC queue when using sufficiently large buffer size. It takes on average 9.7–14 ns per push/pop operation with standard deviation less than 1.5 ns when the internal buffer size is greater than or equal to 1024. The dSPSC queue is slower than the uSPSC version in all cases. If the producer and the consumer for the dSPSC queue are not pinned on the same core the dSPSC queue is more than 10 times slower than the uSPSC queue. Instead, when the producer and the consumer are pinned on the same core the performance is much better for the dSPSC queue (although always worse than the uSPSC one) because they work in lock step as they share the same ALUs and so dynamic memory management is reduced.

It is worth noting that caching strategies for the dSPSC queue implementation significantly improve performance but are not sufficient to obtain optimal figures like those obtained in the uSPSC implementations.

To test scalability of the queues we used a simple synthetic microkernel. We consider N threads linked into a ring using an unbounded queue (dSPSC and uSPSC). The first thread emits a number of messages which flow around the ring. The message is just a pointer obtained from dynamic allocation of a small segment of memory. The other threads accept messages, perform basic integrity verification, copy the input message into a new dynamically allocated buffer, free the input message and pass the new pointer to the next thread. When all messages return to the first thread, the program terminates. Each thread is statically pinned to a core whose id is the same as the thread id. For the architectures considered, the core ids are linear so core 0 and 32 as well as core 0 and 1 are on the same physical core and on the same CPU, respectively, whereas core 0 and 8 are on different CPUs. In Fig. 5 (right), we present the performance in messages per second (msgs/s) obtained while varying the number of threads of the ring. Three queue implementations were tested: the dSPSC queue without using any internal cache (the basic algorithm); the dSPSC queue with a cache size of 32K elements (i.e. with a SPSC queue of size 32K); and the uSPSC queue using a 32K internal SPSC queue and a cache size of 32 elements.

The uSPSC queue implementation obtains the best performance reaching a maximum throughput of ∼250K msgs/s, whereas the dSPSC reaches a maximum throughput of ∼128K msgs/s when using an internal cache of Node(s), and ∼37K msgs/s when no cache is used. For this test, the MS queue implementations (not shown in the graph) obtain almost the same speedup as the dSPSC queue without internal cache. The uSPSC queue scales almost linearly up to 32 cores, then the performance drops due to the fact that on core 0 we have 2 threads in separate contexts (the first and the last one of the ring) producing a bottleneck. Adding more threads in the ring, the bottleneck is slowly absorbed by the increasing throughput thus reaching an optimal final $32X$ improvement.

In this section we have shown only synthetic benchmarks in order to present evidence of the distinctive performance of the uSPSC implementations. The simple tests shown here prove the effectiveness of the uSPSC queue with respect to the dSPSC implementation, and prove also how a fast implementation of a cache of references inside the dSPSC queue leads to much higher throughput. Since the uSPSC queue is used in the FastFlow framework, more performance figures on real-world applications can be found in [15,16].

6 Conclusions

In this paper we studied several possible implementations of fast lock-free Single-Producer/Single-Consumer (SPSC) queues for shared cache multi-core platforms, starting from the well-known Lamport circular buffer algorithm. A new implementation, called dSPSC, of the widely used dynamic list-based algorithm has been proposed. Moreover, a novel unbounded lock-free SPSC queue algorithm called uSPSC has been introduced together with a sketch proof of its correctness and several performance assessments.

The uSPSC queue algorithm and implementation are able to minimize dynamic memory allocation/deallocation and increase cache locality thus obtaining very good performance figures on modern shared cache multi-core platforms. Our uSPSC implementation has been used as a foundation for a skeleton based parallel programming framework (FastFlow [9]) that has been demonstrated to be more efficient than other state-of-the-art programming environments, including OpenMP and Cilk, on significant fine-grain parallel applications.

References

1. Orozco, D.A., Garcia, E., Khan, R., Livingston, K., Gao, G.R.: Toward high-throughput algorithms on many-core architectures. TACO 8(4), 49 (2012)
2. Moir, M., Nussbaum, D., Shalev, O., Shavit, N.: Using elimination to implement scalable and lock-free FIFO queues. In: Proc. of the 7th ACM Symposium on Parallelism in Algorithms and Architectures, pp. 253–262 (2005)

3. Ladan-Mozes, E., Shavit, N.: An optimistic approach to lock-free FIFO queues. Distributed Computing 20(5), 323–341 (2008)
4. Prakash, S., Lee, Y.H., Johnson, T.: A nonblocking algorithm for shared queues using compare-and-swap. IEEE Trans. Comput. 43(5), 548–559 (1994)
5. Tsigas, P., Zhang, Y.: A simple, fast and scalable non-blocking concurrent fifo queue for shared memory multiprocessor systems. In: Proc. of the 13th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 134–143 (2001)
6. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multiprogrammed shared memory multiprocessors. Journal of Parallel and Distributed Computing 51(1), 1–26 (1998)
7. Michael, M.M.: Hazard pointers: Safe memory reclamation for lock-free objects. IEEE Trans. Parallel Distrib. Syst. 15(6), 491–504 (2004)
8. Kahn, G.: The semantics of simple language for parallel programming. In: IFIP Congress, pp. 471–475 (1974)
9. FastFlow framework: website (2009), `http://mc-fastflow.sourceforge.net/`
10. Thies, W., Karczmarek, M., Amarasinghe, S.: StreamIt: A Language for Streaming Applications. In: Horspool, R.N. (ed.) CC 2002. LNCS, vol. 2304, pp. 179–196. Springer, Heidelberg (2002)
11. Reinders, J.: Intel Threading Building Blocks: Outfitting C++ for Multi-core Processor Parallelism. O'Reilly (2007)
12. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. 28(9), 690–691 (1979)
13. Giacomoni, J., Moseley, T., Vachharajani, M.: Fastforward for efficient pipeline parallelism: a cache-optimized concurrent lock-free queue. In: Proc. of the 13th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP), pp. 43–52 (2008)
14. Lee, P.P.C., Bu, T., Chandranmenon, G.P.: A lock-free, cache-efficient multi-core synchronization mechanism for line-rate network traffic monitoring. In: Proc. of the 24th Intl. Parallel and Distributed Processing Symposium, IPDPS (2010)
15. Aldinucci, M., Ruggieri, S., Torquati, M.: Porting Decision Tree Algorithms to Multicore Using FastFlow. In: Balcázar, J.L., Bonchi, F., Gionis, A., Sebag, M. (eds.) ECML PKDD 2010, Part I. LNCS, vol. 6321, pp. 7–23. Springer, Heidelberg (2010)
16. Aldinucci, M., Danelutto, M., Meneghin, M., Kilpatrick, P., Torquati, M.: Efficient streaming applications on multi-core with FastFlow: the biosequence alignment test-bed. In: Parallel Computing: From Multicores and GPU's to Petascale. Advances in Parallel Computing, vol. 19, pp. 273–280. IOS Press (2009)
17. Lamport, L.: Concurrent reading and writing. CACM 20(11), 806–811 (1977)
18. Adve, S.V., Gharachorloo, K.: Shared memory consistency models: A tutorial. IEEE Computer 29, 66–76 (1995)
19. Higham, L., Kawash, J.: Critical sections and producer/consumer queues in weak memory systems. In: Proc of the Intl. Symposium on Parallel Architectures, Algorithms and Networks (ISPAN), pp. 56–63. IEEE (1997)
20. Jablin, T.B., Zhang, Y., Jablin, J.A., Huang, J., Kim, H., August, D.I.: Liberty queues for epic architectures. In: Proc. of the 8th Workshop on Explicitly Parallel Instruction Computer Architectures and Compiler Technology, EPIC (2010)
21. Hendler, D., Shavit, N.: Work dealing. In: Proc. of the 4th ACM Symposium on Parallel Algorithms and Architectures (SPAA), pp. 164–172 (2002)
22. Torquati, M.: Single-producer/single-consumer queues on shared cache multi-core systems. Technical Report TR-10-20, Computer Science Dept., University of Pisa, Italy (2010), `http://compass2.di.unipi.it/TR/Files/TR-10-20.pdf.gz`