An efficient Unbounded Lock-Free Queue for Multi-Core Systems

Authors:

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

[1] Computer Science Dept. - University of Turin – Italy

[2] Computer Science Dept. - University of Pisa - Italy

[3] Computer Science Dept. Queen's University Belfast – UK

[4] IBM Dublin Research Lab, Ireland

Speaker:

Massimo Torquati e-mail: torquati@di.unipi.it

Talk Outline

- Producer-Consumer queues
  - Lock-Free vs Wait-Free
  - Bounded SPSC queue
  - Why lock freedom and why SPSC queues ?
- Unbounded List-based Lock-Free queue
- Fast Unbounded Lock-Free queue
- Performance measurements
- Conclusions

Producer-Consumer queues

- Producer-Consumer queues are fundamental data structures in concurrent systems
- Widely used in many run-time supports and algorithms to implement:
  - data/message channels
  - synchronization mechanisms
  - task scheduling policies
- No single queue implementation suitable/efficient for all situations

Producer-Consumer queues

- Queues can be classified depending on many factors, the most significant are:
  - Concurrency level: SPSC, SPMC,MCSP,MPMC
  - Internal data structures: Array-based, List-based
  - Size: Bounded, Unbounded
  - Progress guarantees: No guarantee (blocking), Obstruction freedom, Lock freedom, Wait freedom
- We are interested in unbounded non-blocking **Lock/Wait-free SPSC** queues, either array-based or list-based

Blocking vs Non-Blocking

- **Non-blocking**: No thread is blocked waiting for other threads to complete some operations.

  ⇨ no locks or critical sections may be used

- **Progress guarantees**:
  - Obstruction freedom (weakest)
    - A thread, if executed in isolation, makes progress.
  - Lock freedom
    - With enough time, at least one thread makes progress
  - Wait freedom (strongest)
    - A thread always completes its ops in a bounded number of steps

Expected performance

- What are the performance implications of the progress properties ?
- In the general case:
  - For medium/coarse grain applications:

    Blocking faster than → **Non-Blocking**

  - For fine grain applications:

    Non-Blocking faster than → **Blocking**

    Obstruction-Free faster than → **Lock-Free** faster than → **Wait-Free**

  - Stronger properties are harder to maintain

Bounded SPSC queues

- Lamport's ring buffer was the cornerstone [Lamport'83]
  - Original works under *Sequential Consistency*
- With few modifications it works also under *weak ordered* memory models (P_1C_1-queue) [Higham&Kavalsh'97]
  - Combines control and data info using a special $\perp$ value
- fastforward lock-free queue optimized Lamport's queue for multi-core [Giacomoni at all.'08]
  - It uses the same algorithm of the P_1C_1-queue
- Various other works (e.g. MCRingBuffer, LibertyQueue) focused on further or different optimizations of the bounded SPSC for multi-core systems

Bounded SPSC queue

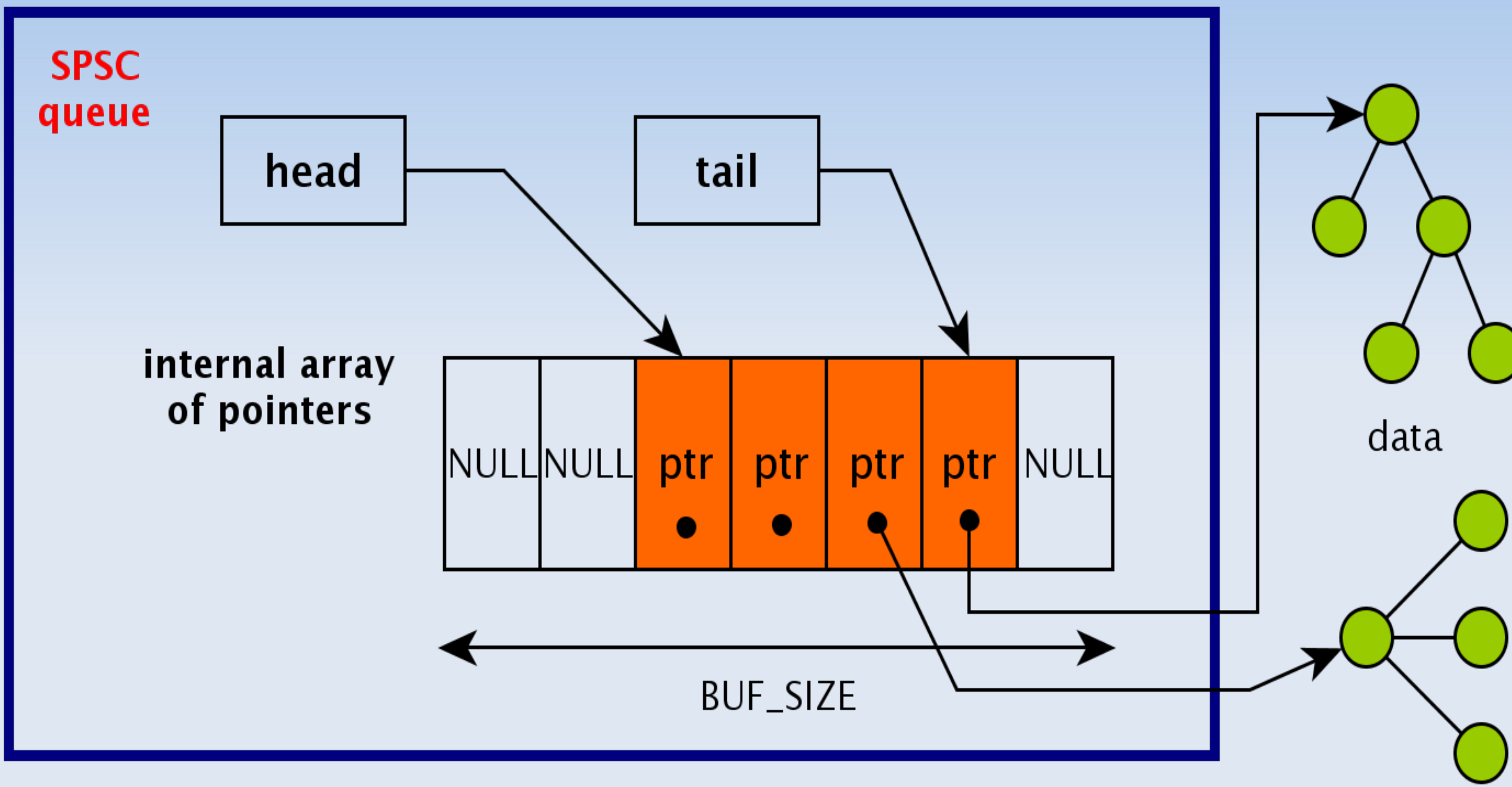
- We used the **fastforwad** queue (without the temporal sliping optimization) as a basic building block:
  - **<u>Req</u>**: effects of a store op seen completed or not at all
  - The queue carries **memory pointers**:
    - A Write-Memory-Barrier (WMB) is needed under WO
    - The special $\bot$ value is NULL

```
// Internal buffer having BUF_SIZE slots initialized to NULL
void* buf[BUF_SIZE];

bool push(void* data) {
  if (buf[tail]==NULL) {
    WBM();
    buf[tail]=data;
    tail = NEXT(tail,BUF_SIZE);
    return true;
  }
  return false;
}

bool pop(void** data) {
  if (buf[head]==NULL) return false;
  *data = buf[head];
  buf[head] = NULL;
  head= NEXT(head,BUF_SIZE);
  return true;
}
```

Bounded SPSC queue schema



- The ***head*** and ***tail*** indexes are mapped on different cache-lines (data padding is added to fill up a cache-line)

Progress guarantees of the SPSC queue

- Lamport's ring buffer is **Wait-Free**
  - An ”extra” WMB operation is needed under weak memory ordering (WO) to ensure correctness

- The fastforwad queue (i.e. the SPSC queue in our nomenclature) is Lock-Free and works under WO
  - If thread termination and restart is not allowed during the program execution, then it is Wait-Free

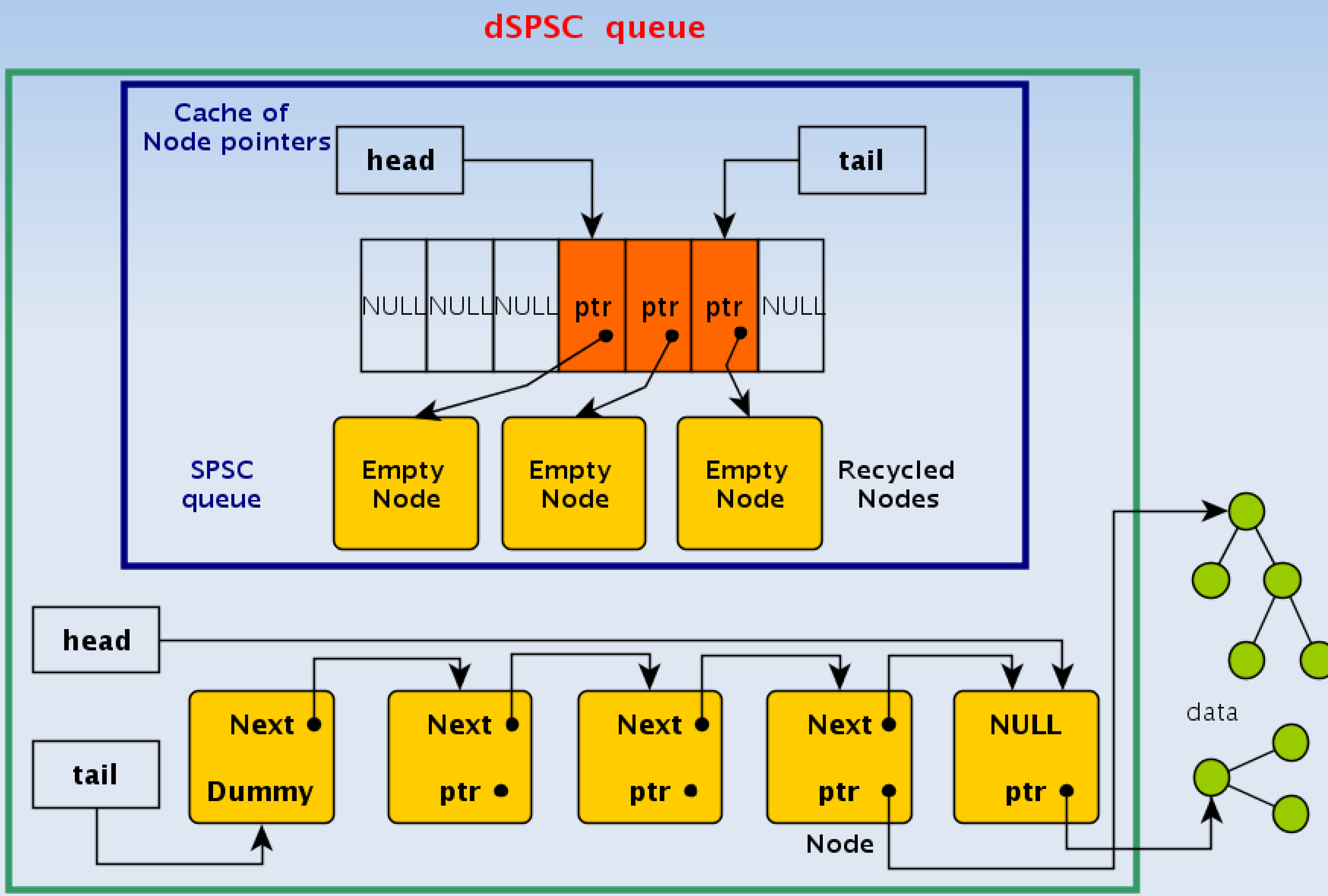
Bounded vs Unbounded SPSC queues

- Bounded SPSC queue is very **simple**, **elegant** and **performs very well on multi-cores** when the producer and the consumer work on different cache lines.
- So, why do we need unbounded SPSC queues ?
  - May not be easy to determine the correct BUF_SIZE for the internal array
    - Need to set it up for the worst case
  - Bounded queues may introduce deadlock issues
    - In case of complex process networks with cycles
  - They are more general than bounded queues
    - But sometimes dangerous to use
  - Unbounded SPSC queues have been less investigated

Unbounded List-Based SPSC queue

- Starting from the well-known two-lock MPMC queue by Micheal and Scott (MS_2-lock) [Michael&Scott'98]
  - A List-based unbounded SPSC queue is obtained simply removing the 2 locks
  - MS_2-lock links dynamically allocated *Node*(s) containing pointer to user data
  - *head* and *tail* pointers initially point to a dummy Node
- Our version (called dSPSC) uses an internal cache of Nodes implemented with a SPSC queue to reduce memory allocation/deallocation
  - The SPSC cache is used in the opposite direction w.r.t. the dSPSC queue, i.e. the dSPSC consumer is the producer of the SPSC cache.

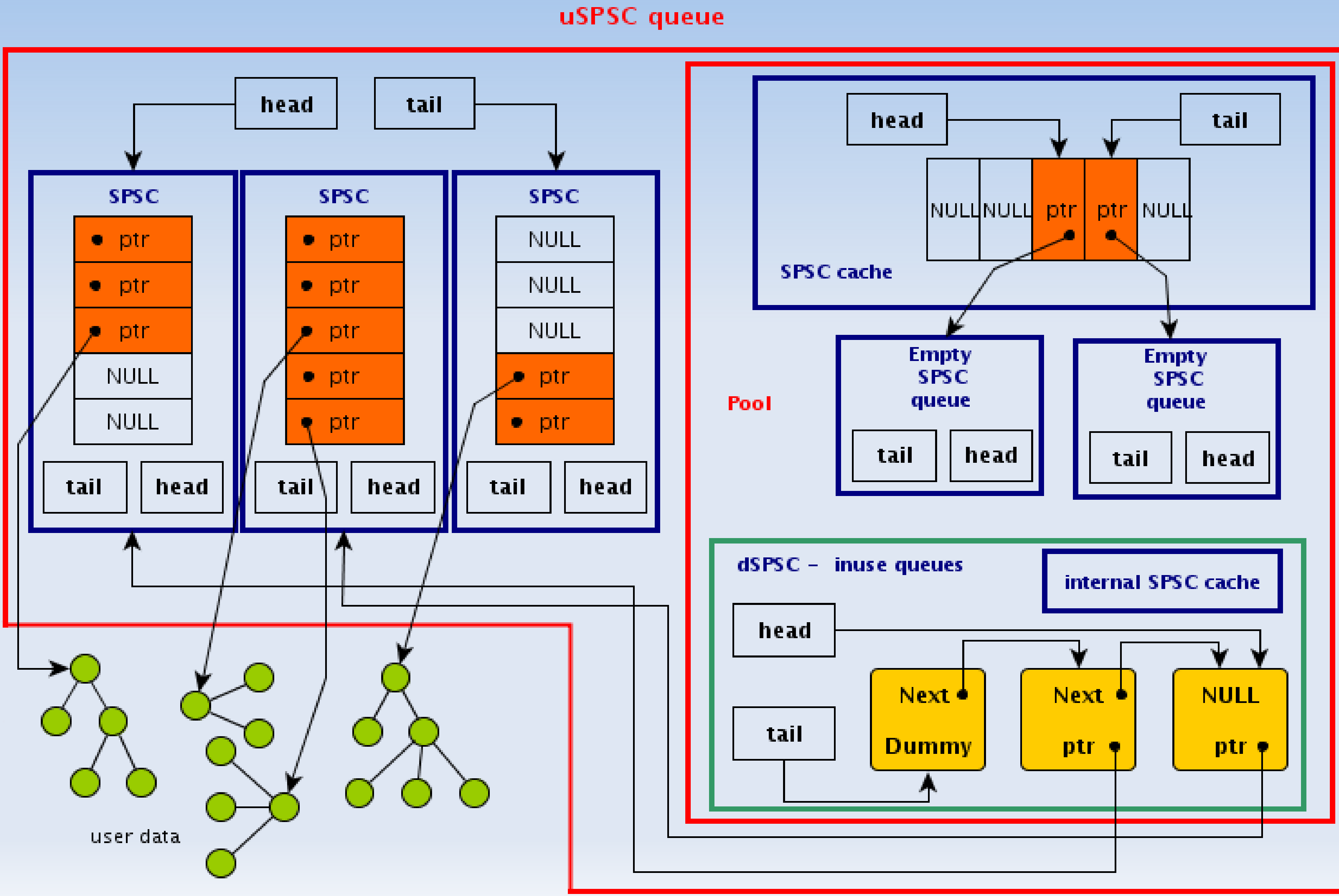
dSPSC queue schema



Unbounded array-based SPSC queue

- **Idea**: **using a ”pool” of SPSC queues**
  - FIFO ordering is guaranteed by an internal dSPSC queue which contains pointers to in-use SPSC queues
- **Advantages**: using array-based SPSC queues increases overall cache locality
- As for the dSPSC, dynamic memory management has to be minimized
  - The internal Pool leverages on a cache of empty SPSC queues
- The unbounded queue has the same interface as the SPSC and dSPSC queues.

uSPSC queue schema



uSPSC queue algorithm

- At the beginning a single bounded SPSC queue is used
  - Both P and C work on the same initial queue
- As soon as the queue fills up, the Pool provides a new (possibly recycled) SPSC queue to the producer
- When the tail queue has been emptied, the Pool provides the consumer with the next SPSC queue that is in-use in the dSPSC queue
- The algorithm is independent of the underlying SPSC queue implementation used

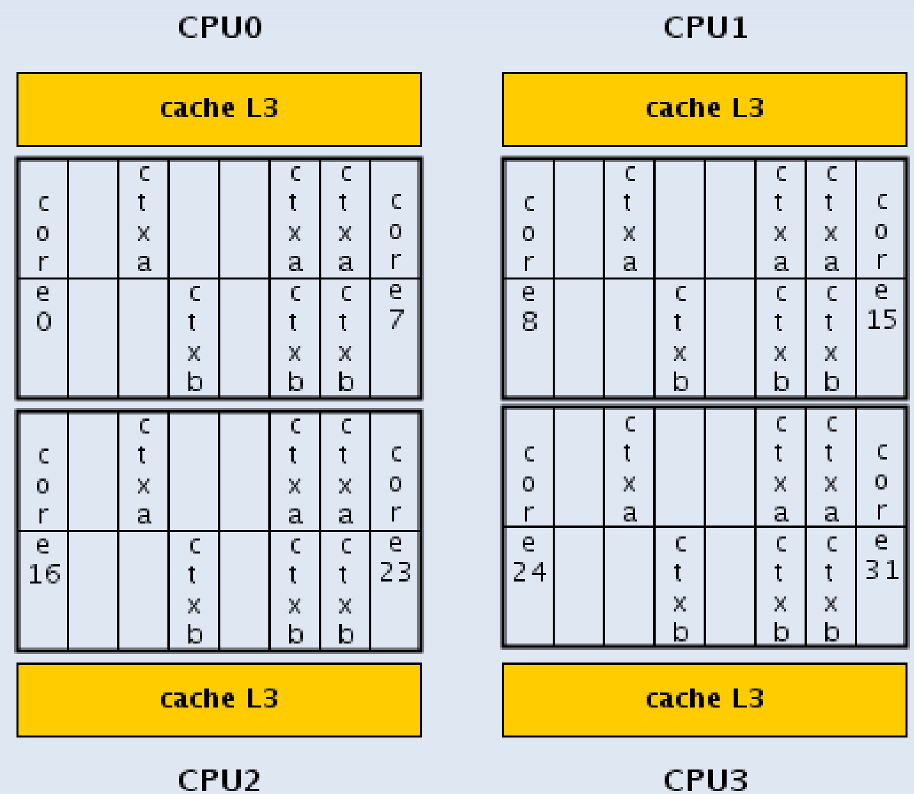
```
// buf_w is the tail pointer
bool push(void* data) {
  if (buf_w->full())
    buf_w = pool.next_w();
  buf_w->push(data);
  return true;
}
// buf_r is the head pointer
bool pop(void** data) {
  if (buf_r->empty()) {
    if (buf_r == buf_w) return false;
    if (buf_r->empty()) {
      SPSC* tmp = pool.next_r();
      pool.release(buf_r) ;
      buf_r = tmp;
    }
  }
  return buf_r->pop(data);
}
```

Progress guarantee of the dSPSC and uSPSC queues

- Both queues dynamically allocate memory, if needed
- Depending on the memory allocator used, the two queues are:
  - **”Almost” Lock-Free**
    - Lock-free only in the fast-path, when memory is neither allocated nor freed but recycled from the cache
  - **Lock-Free** if a lock-free memory allocator is used for Node or SPSC allocation outside the fast-path
  - **Wait-Free** if a wait-free memory allocator is used outside the fast-path, and no thread termination and restart is allowed.

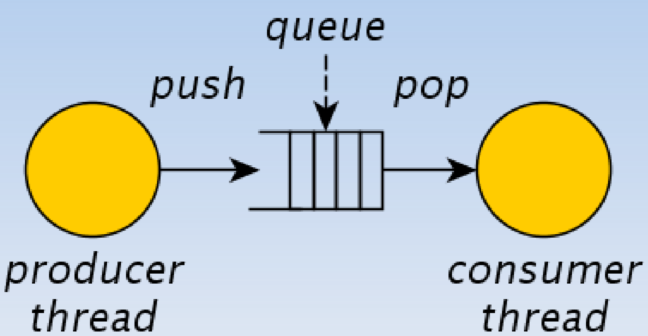
Performance evaluations

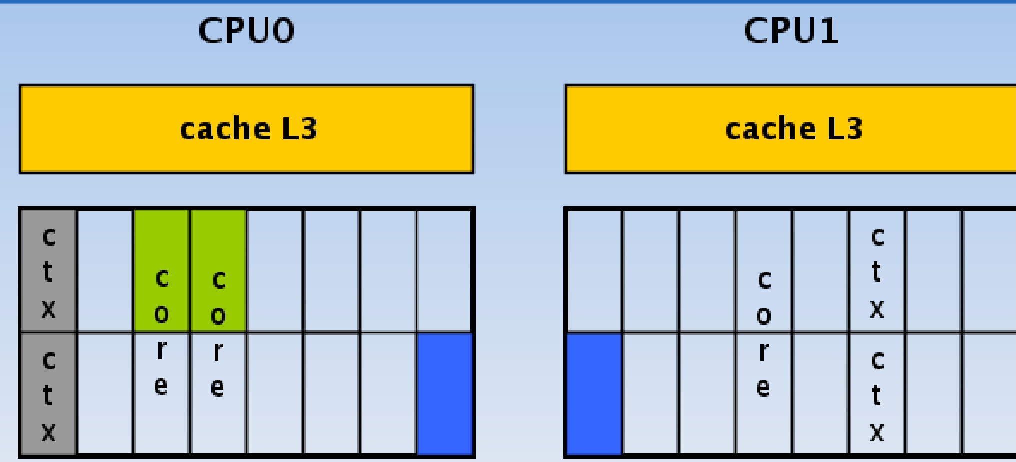
- Test environment: 4 eight-core double context Xeon E7-4820 @2.0GHz, 18MB shared L3 cache
  - L3 cache is shared among all eight cores
  - L2 cache (256KB) is shared between the 2 contexts of the single core



- 2 kinds of tests:
  - Latency of queue ops
  - Scalability when many queues are used

Latency of push/pop ops





Thread Mapping: different context of the same core | different core of the same CPU | different CPUs

```
queue_t Q(size);
```

Producer:

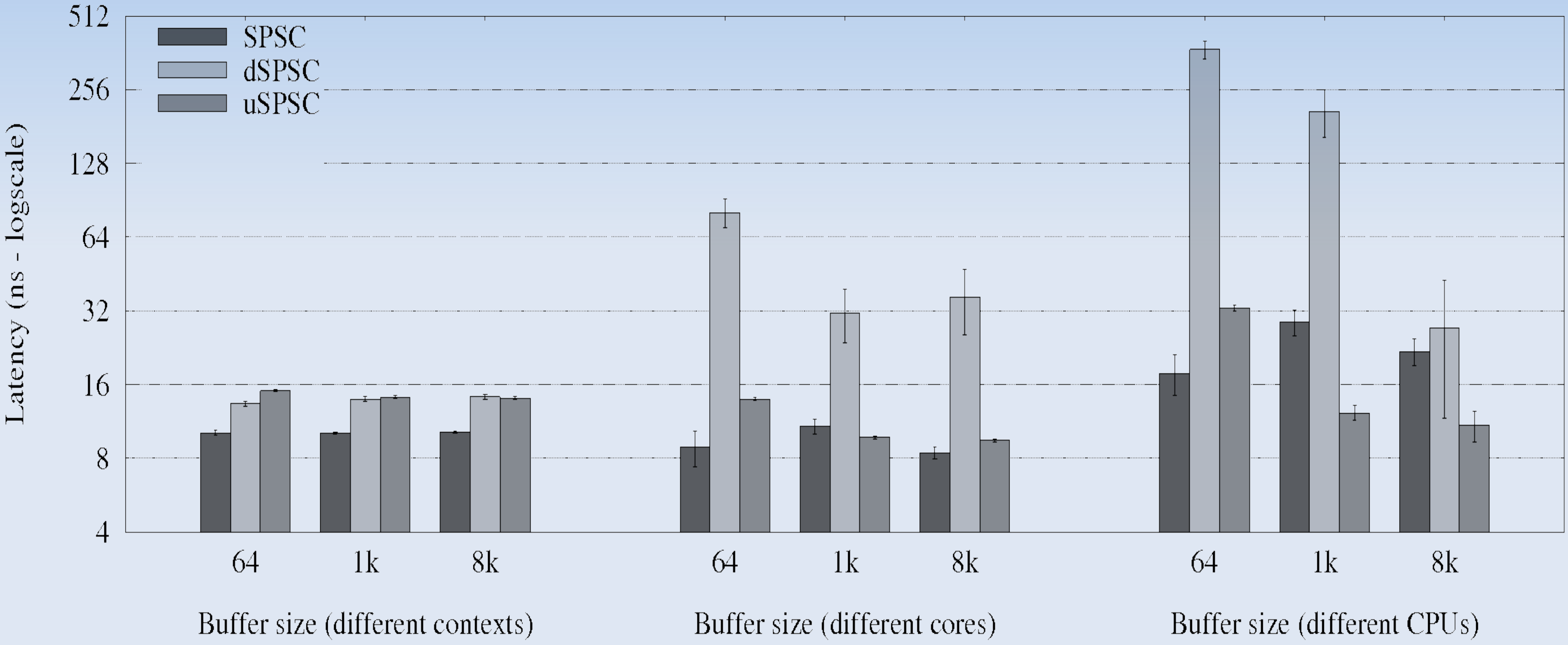
```
for(i=0;i<NUMTASK;++i) {
  long* p = 0x1234 + i;
  do ;  while(Q->push(p));
}
do; while(Q->push(EOS));
```

Consumer:

```
do  {
   Q.pop(task);
    If (task == EOS) end =true;
    else
         If (p != (0x1234 + i)) error();
} while (! end );
```

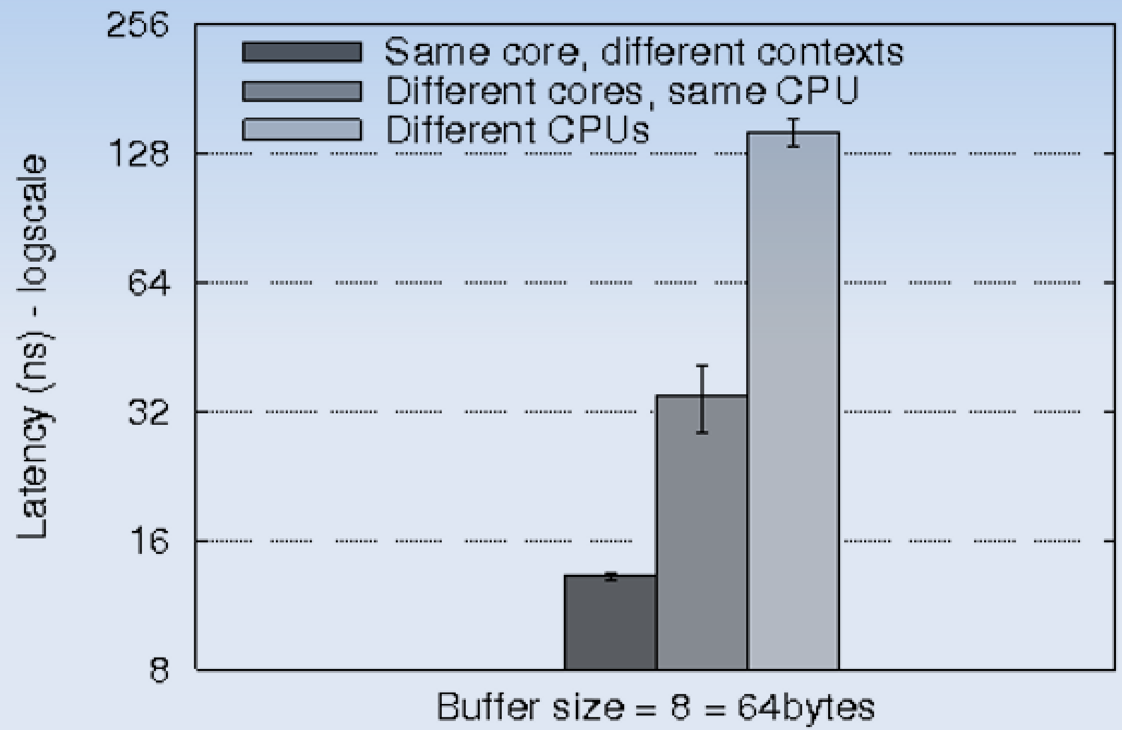
- In this test the producer is a bit faster than the consumer

Latency: cross-comparison



Latency with small buffer size

- What if the producer and the consumer work on the same cache-line ?

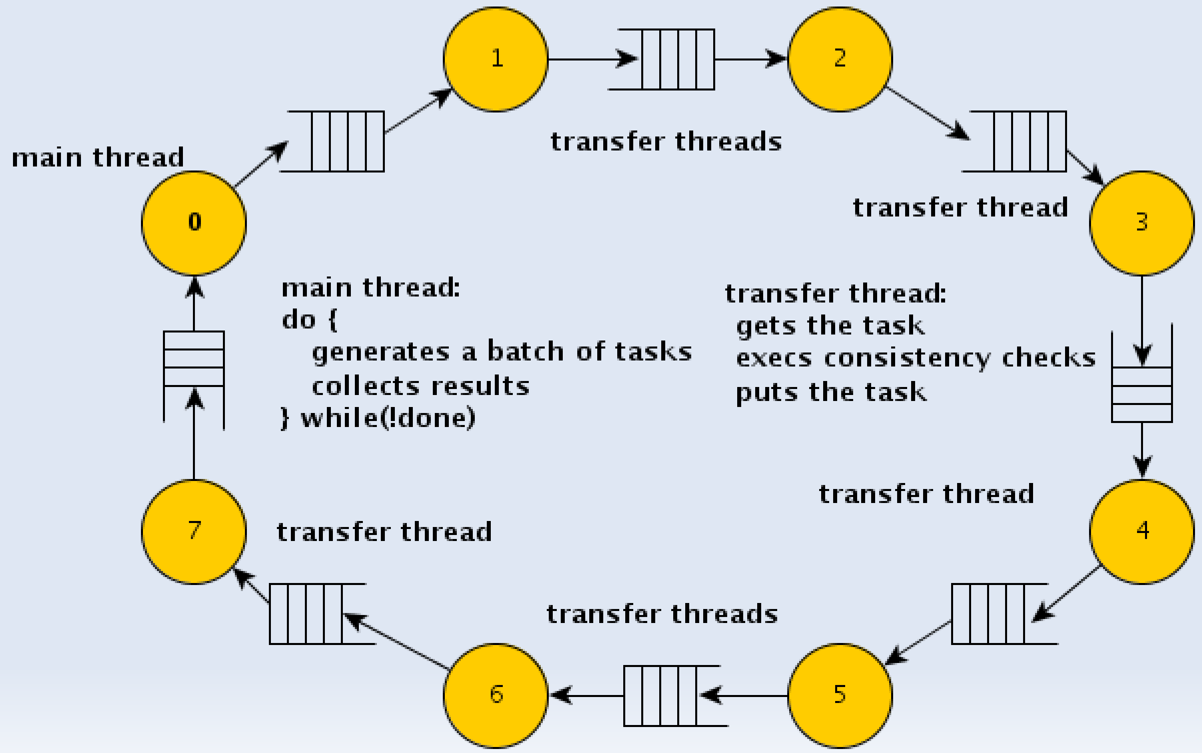


- Lots of cache invalidations due to false-sharing
- From 3 to 5 times slower
- This happens when the producer is (temporarily?) slower than the consumer

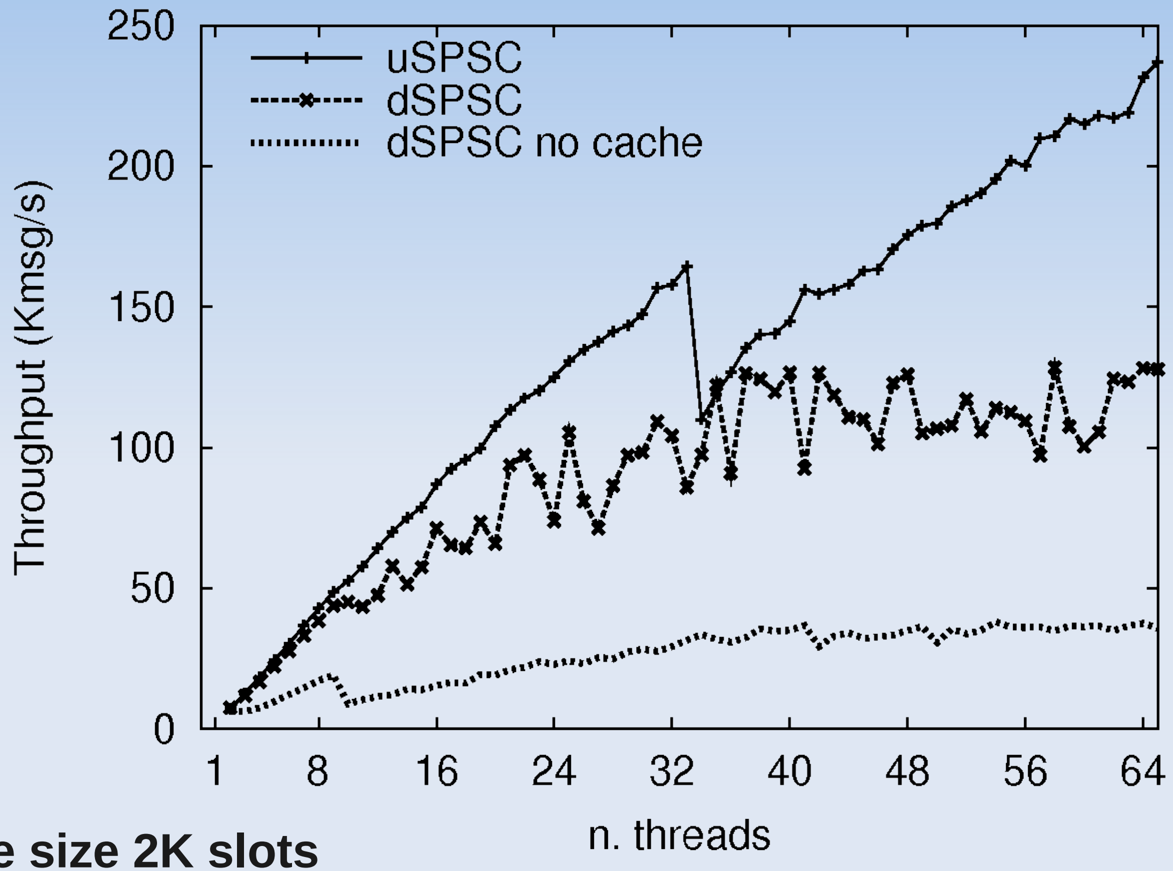
- There are several techniques which ”force” the producer and the consumer to work on separate cache-lines (temporal sliping, batch update of control variables, multi-push, etc.)
  - **Not easily usable**, may require fine tuning or non-standard interface to avoid deadlock

Measuring scalability

- The benchmark consists in a ring of N threads exchanging msgs using unbounded queues as inter-thread channel
  - The main thread produces K batches of 256 msgs
  - In this test bounded queues can also be used but in general the stages may be unbalanced
- msgs are just few bytes of dynamically allocated memory



Unbounded queues throughput



Settings:
- dSPSC cache size 2K slots
- uSPSC buffer size 2K slots, cache size 32 slots

- The dSPSC queue performs poorly without the internal cache
- The uSPSC queue scales quite well (~ 250K msgs/s) eventually obtaining ~32x scalability

SPSC queues in FastFlow

- Both the SPSC and the uSPSC queues are used as basic building blocks in the FastFlow parallel framework
- FastFlow provides a skeleton based parallel programming model on shared-cache multi-core
  - Using the unbounded queue it implements the *pipeline*, *farm* and *D&C* skeletons (all of them may be nested).
- Many parallel applications have been developed using the FastFlow framework
  - The low overhead of the SPSC queues results in good performance also for fine-grain parallel algorithms
- More info on FastFlow:
  - http://mc-fastflow.sourceforge.net

Conclusions

- Unbounded SPSC queues have been studied
- A new lock-free implementation called dSPSC of the widely used two-lock MS-queue algorithm has been proposed
- A novel unbounded array-based SPSC queue called uSPSC has been proposed and tested
  - The uSPSC queues performs very well on shared-cache multi-core
  - It is built in such a way that ”specialized” bounded SPSC queues can be used

Thanks !

Any questions?

Queues implementation can be found within the FastFlow source code:

http://mc-fastflow.sourceforge.net/

uSPSC complete algorithm

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

bool push(void* data) {
  if (buf_w->full())
    buf_w = pool.next_w();
  buf_w->push(data);
  return true;
}


struct Pool {
  dSPSC inuse;
  SPSC  cache;

  SPSC* next_w() {
    SPSC* buf;
    if (! cache.pop(&buf))
       buf = allocateSPSC(size);
    inuse.push(buf);
    return buf;
  }
  SPSC* next_r() {
    SPSC* buf;
    if (inuse.pop(&buf)) return buf;
    return NULL;
 }
 void release(SPSC* buf) {
   buf->reset(); // reset head and tail pointer of the SPSC queue
   if (! cache.push(buf)) deallocateSPSC(buf);
 }
};
```

```
bool pop(void** data) {
   if (buf_r->empty()) {
     if (buf_r == buf_w) return false;
     if (buf_r->empty()) {
       SPSC* tmp = pool.next_r();
       pool.release(buf_r) ;
       buf_r = tmp;
     }
   }
   return buf_r->pop(data);
}
```