Non-Blocking Synchronization of Concurrent Shared Data Structures

TDDD56 Lecture 4

Christoph Kessler

PELAB / IDA
Linköping university
Sweden

2014

Outline

Motivation

Lock-based synchronization of critical sections has drawbacks:

- Mutual exclusion leads to sequentialization
  - Overhead of acquiring + releasing the lock
  - Shared memory access traffic (cache update) for polling the lock status
  - Convoying effect, if critical section is long and/or frequently accessed
  - Reduced parallelism as waiting processors go idle
- Deadlock risk
- Priority inversion problem
  - Priority inheritance only works for single-processor systems
  - Requires much more pessimistic WCET (worst-case execution time) prediction
    - Either low utilization or risk of missing deadlines in real-time computing

Desired: Non-Blocking Operations

- **Non-Blocking**
  = No thread can be blocked by the **in-action** of other threads such as preemption, page fault, or even termination

Idea: Lock-Free Synchronization

- No locks, no mutual exclusion
  → skip all the problems above ☺
- **Principle**: Use hardware atomic operations to ensure correct update of shared data structures.
  - Speculatively prepare an operation (e.g., inserting an element in a shared data structure)
  - Then check if no conflicts have occurred, and atomically commit the new situation at the same time
  - If a conflict was detected, repeat the operation
    - One operation will always succeed → **lock-free**
    - If upper bound on #retrys guaranteed → **wait-free**
- **Specific solutions required** for each kind of shared data structure and each kind of HW atomic operation available!

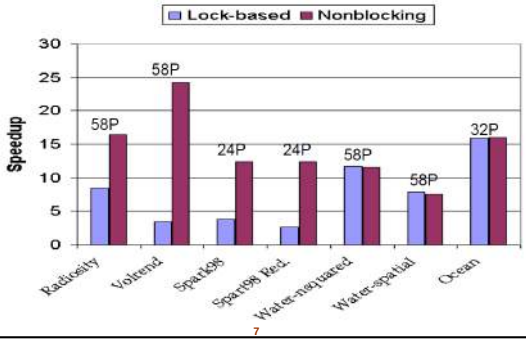
Case Study: Tsigas and Zhang 2002

Benchmark programs

- Ocean
  - simulates eddy currents in an ocean basin.
- Radiosity
  - computes the equilibrium distribution of light in a scene using the radiosity method.
- Volrend
  - renders 3D volume data into an image using a ray-casting method.
- Water
  - Evaluates forces and potentials that occur over time between water molecules.
- Spark98
  - a collection of sparse matrix kernels.
- Run on a SUN Enterprise 10000 server with 64 processors

Case Study (cont.)

Ph. Tsigas and Y. Chang: Integrating Non-blocking Synchronizsation in Parallel Applications: Performance Advantages and Methodologies. Proc. 3rd ACM Workshop on Software and Performance (WOSP'02), 2002



Recall: Hardware Atomic Operations

- Modern multiprocessor machines provide special ***atomic instructions***
  - **TestAndSet**: test memory word and set value atomically
    - Atomic = non-interruptable
    - If multiple TestAndSet instructions are executed *simultaneously* (each on a different CPU in a multiprocessor), then they take effect sequentially in some arbitrary order.
  - **Fetch-and-Op**: read memory word and apply op to it atomically
    - Fetch-and-increment is most common
    - Implementation in HW: lock the memory bus while being performed so no other processor can access the memory location simultaneously
  - **AtomicSwap**: swap contents of two memory words atomically
  - **CompareAndSwap**: swap if old value still equals current value
  - **Load-linked / Store-conditional:**
    - LL: check out a memory word with version number
    - SC: check in a new value only if version number still as of my last LL

TestAndSet Instruction

- Definition in pseudocode:

```
boolean TestAndSet (boolean *target)
{
    boolean rv = *target;
    *target = TRUE;
    return rv;      // return the OLD value
}
```

atomic

Mutual Exclusion using TestAndSet

- Shared boolean variable lock, initialized to FALSE (= unlocked)
- **do** {

```
    while ( TestAndSet (&lock ))
        ;   // do nothing but spinning on the lock (busy waiting)

    // …  critical section

    lock = FALSE;

    // …    remainder section

} while ( TRUE);
```

→EXAMPLE

Fine-Grained Locking (1)

A concurrent hashtable with fine-grained locking

1 mutex lock per hash value / list, as operations on different lists are independent

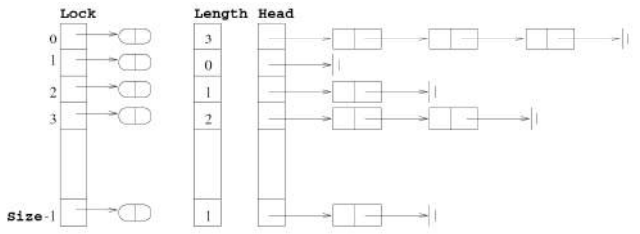


Image Source: J. Keller, C. Kessler, J. Träff, *Practical PRAM Programming*, Wiley Interscience, New York © 2001.

FetchAndIncr Instruction

- Definition in pseudocode:

```
boolean FetchAndIncr ( int *target, int k )
{
    int  old = *target;
    *target = old + k;
    return old;
}
```

atomic

Example: Fair Lock with Fetch&Incr

- Implementation of a **Fair lock** (FIFO admission) with busy waiting
  - 2 shared counters, initialized to 0: ticket, active
  - Acquire: myticket = **FetchAndIncr**(&ticket, 1);
    while (myticket != active) ; // busy waiting
  - Release: **Fetch&Incr**(&active, 1);
    (or: active ++; if store is atomic)

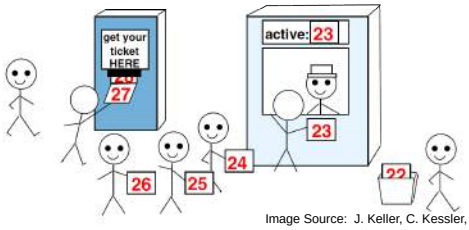
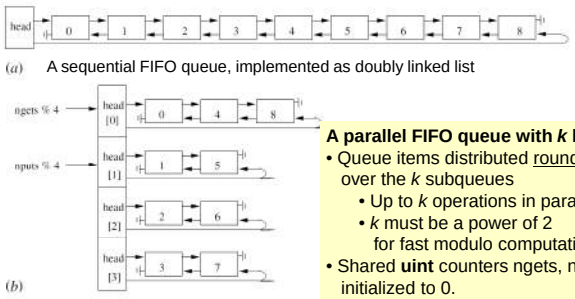


Image Source: J. Keller, C. Kessler, J. Träff, *Practical PRAM Programming*, Wiley Interscience, New York © 2001.

Fine-Grained Locking (2)



A sequential FIFO queue, implemented as doubly linked list

A parallel FIFO queue with *k* heads.
- Queue items distributed round-robin over the *k* subqueues
  - Up to *k* operations in parallel
  - *k* must be a power of 2 for fast modulo computation.
- Shared **uint** counters ngets, nputs initialized to 0.
- Use **fetch&incr**(&nputs, 1) to determine subqueue for next insert
- Use **fetch&incr**(&ngets, 1) to determine subqueue for next extract
- Each subqueue protected by a lock.

Image Source: J. Keller, C. Kessler, J. Träff, *Practical PRAM Programming*, Wiley Interscience, New York © 2001.

Fine-Grained Locking (2) – cont.

Remarks on the Parallel FIFO Queue:

- The *i*-th put / get access is routed to sublist *i* % k, for i > 0.
  - The counters nputs and ngets overflow at 2^{32}-1 (no problem in practice as long as *k* is a power of 2 and there are not that many pending put or get operations)
- The locks for the sublists must be fair locks (see above) to guarantee FIFO order of operations of the same kind (put or get) working on the same sublist
- Also, a mutex lock is required to avoid concurrent modification by both put and get operations in the same list
- Test for isempty() makes the implementation more tricky (not discussed here for simplification)
  - For details see A. Gottlieb, B. Lubachevsky, L. Rudolph: Basic techniques for the efficient coordination of large numbers of cooperating sequential processes. *ACM Transactions on Programming Languages and Systems* **5**(2):164-189, Apr. 1983.

Load-linked / Store-conditional

2 new instructions for memory access:

- **LoadLinked** *address, register*
  - records the version number of the value read (cf. a svn update)
- **StoreConditional** *register, address*
  - will only succeed if no other operations were executed on the accessed memory location since my last LoadLinked instruction to *address*, (cf. a svn commit)
  - and set the register operand of Store-conditional to 0, otherwise.

Mutual Exclusion using LL/SC

- Shared int variable lock, initialized to 0 (= unlocked)
- **do** {
  register = 0;
  while (register == 0) {
  dummy = **LoadLinked** (&lock);
  if (dummy == 0) { // read a 0 – found unlocked
  register = 1;
  register = **StoreConditional** (register, &lock);
  }
  // if register is 0, StoreConditional failed, retry…
  }
  // … critical section
  lock = 0; // ordinary store
  // … remainder section
  } **while** (TRUE);

Compare-And-Swap (CAS)

- **CAS** (adr_memcell, value, register)
  - Atomically compares a value in a memory cell to a supplied value and, if these are equal, swaps the contents of the memory cell with the value stored in a register.
- **Example: Mutual Exclusion using CAS:**

```
register = 1;
CAS ( &lock, 0, register );
while (register != 0)
        CAS ( &lock, 0, register );
// … critical section
lock = 0;
//
```

Atomic Counter Increment with CAS

- Shared int counter initialized to 0;
- **do** {
  oldval = counter;
  newval = oldval + 1;
  CAS (&counter, oldval, &newval);
  } **while** (newval != oldval); // repeat until CAS succeeds

Example:

Time	counter val.	Thread 1	Thread 2
t	0	read counter: 0	
t + 1		newval = 1	read counter: 0
t + 2	1	**CAS** (&counter, 0, 1) succeeds	newval = 1
t + 3	1		**CAS** (&counter, 0, 1) **fails**
t + 4			read counter: 1
t + 5			newval = 2
t + 6	2		**CAS** (&counter, 1, 2) succ.

Availability of Atomic Instructions (1)

- **CAS instruction**
  - Intel/AMD x86/Itanium:
    CMPXCHG (Compare-And-Exchange)
    - Needs a **LOCK** prefix to make it really atomic
  - Oracle/Sun SPARC: **CAS**
  - Take 3 arguments (addr, exp_value, new_value) and return boolean (successful / unsuccessful)
- **DCAS (Double-CAS, CAS2) instruction**
  - (adr1, expval1, newval1, adr2, expval2, newval2)
  - Motorola MC68K **CAS2** (for some time)
- **Double-Width CAS**
  - Intel/ARM x86: **CMPXCHG8B**, **CMPXCHG16B**
  - Works on two pointers adjacent in memory
- **Single-Compare Double-Swap**: Intel x86 CMP8XCHG16B

Availability of Atomic Instructions (2)

- **LL/SC instructions**
  - Alpha AXP (ldl_l, stl_c)
  - IBM PowerPC (lwarx, stwcx)
  - MIPS (ll, sc)
  - ARM (ldrex, strex)
- Some restrictions though
  - Context switch or a Load, LL, SC instruction between LL and SC may cause SC to fail

Availability of Atomic Instructions (3)

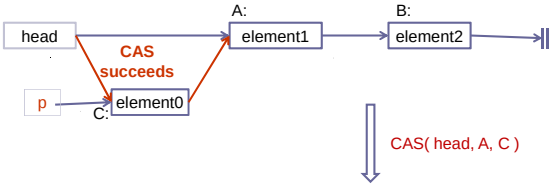
- **Cost**: widely varying across architectures and situations
  - CAS / LL / SC *can* be an order of magnitude slower than Load / Store accesses!
    - Includes a memory barrier (fence)
    - Prevents out-of-order execution and certain compiler optimizations (instruction reordering)

Idea: Lock-Free Synchronization

- No locks, no mutual exclusion
  → skip all the problems above ☺
- **Principle**: Use hardware atomic operations to ensure correct update of shared data structures.
  - Speculatively prepare an operation (e.g., inserting an element in a shared data structure)
  - Then check if no conflicts have occurred, and atomically commit the new situation at the same time
  - If a conflict was detected, repeat the operation
    - One operation will always succeed → **lock-free**
    - If upper bound on #retrys guaranteed → **wait-free**
- **Specific solutions required** for each kind of shared data structure and each kind of HW atomic operation available!

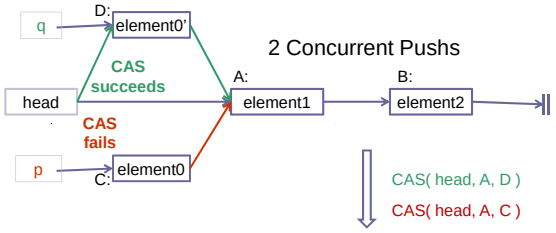
Example: Lock-Free Concurrent Stack (1)

- Stack implemented as linearly linked list
  - Push (element): insert element at the head
  - Pop (element): splice first element out of the list



Example: Lock-Free Concurrent Stack (2)

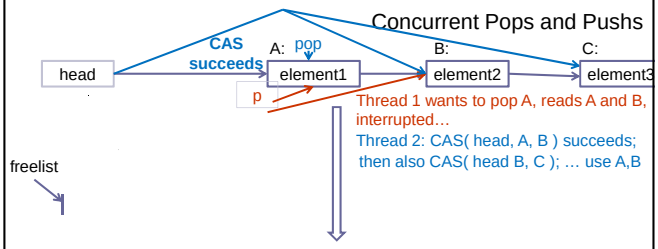
- Stack implemented as linearly linked list
  - Push (element): insert element at the head
  - Pop (element): splice first element out of the list



At first glance, seems to work with CAS, as only 1 memory word (head pointer) needs to be changed atomically. But…

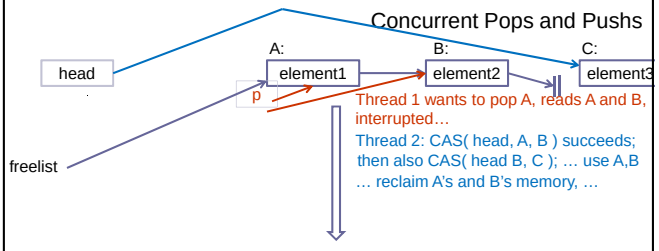
Example: Lock-Free Concurrent Stack (3)

- Stack implemented as linearly linked list
  - Push (element): insert element at the head
  - Pop (element): splice first element out of the list



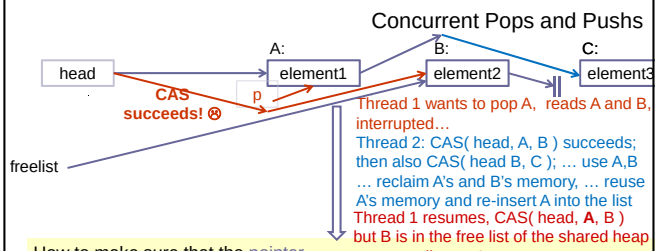
Example: Lock-Free Concurrent Stack (3)

- Stack implemented as linearly linked list
  - Push (element): insert element at the head
  - Pop (element): splice first element out of the list



Example: Lock-Free Concurrent Stack (3)

- Stack implemented as linearly linked list
  - Push (element): insert element at the head
  - Pop (element): splice first element out of the list



How to make sure that the pointer to element1 is not altered before CAS?
→ **the ABA-problem!**

The ABA Problem with Compare&Swap

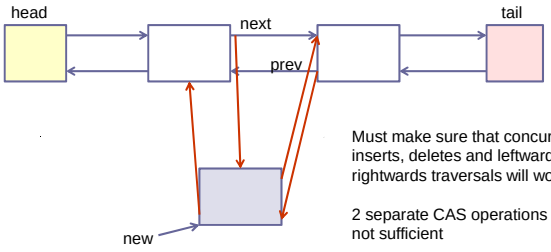
- **ABA-Problem:**
  CAS cannot detect if a memory location has changed value from a value *A* to a value *B* and then back to *A* again.
- Can be fixed e.g. by adding a time stamp (version counter) to the memory location
  - E.g., in some (unused?) most-significant bits of the memory word
  - Increment the version counter every time the value changes
  - Lowers the probability that the ABA problem occurs
  - Restricts the amount of data (#bits) that can be stored in the variable, and adds some bit-masking overhead
- Or use double-CAS (DCAS), if available…

The relative "synchronization power" of 1-word Hardware Atomic Operations

- Level 0: Weak synchronization support even with sequential consistency
  - **Load, store**
    (e.g. with Dekker/Peterson software *mutual exclusion* algorithms)
- Level 1: Sufficient to synchronize simple problems, e.g. shared counter increment, or for atomic acquire/release of mutex locks
  - **TestAndSet**: test memory word and set value atomically
  - **Fetch-and-Op**: read memory word and apply op to it atomically
  - **AtomicSwap**: swap contents of two memory words atomically
- Level 2: Suitable for a number of non-blocking concurrent data structures
  - **CompareAndSwap**: swap if expected value equals current value
  - **Load-linked / Store-conditional**
- For some non-blocking concurrent data structures, multi-word compare-and swap (e.g. DCAS, double CAS) is necessary

Example: Doubly Linked List

- How to atomically update both prev and next pointers?



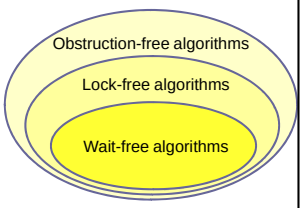
Must make sure that concurrent inserts, deletes and leftwards / rightwards traversals will work

2 separate CAS operations are not sufficient

Requires DCAS (double compare-and-swap)

Classification of Non-blocking Algorithms and Data Structures

- **Obstruction-free**
  - Guarantees progress (of at least one operation) in absence of contention
  - Need an extra module for contention management
- **Lock-free**
  - Guarantees that always at least one operation is making progress
- **Wait-free**
  - Guarantees that any operation will finish within finite time
  - Good for schedulability analysis in real-time systems



Remark on Memory Management

- A dynamic data structure requires support by the heap memory allocator
  - **malloc**()+**free**() or **new**+garbage collector as available
  - These are implemented in the language's run-time system, usually with locks → not lock-free / wait-free
- A lock-free / wait-free data structure must not rely on a lock-based heap memory allocator
  - Otherwise guarantees cannot be given
- Need to implement an own lock-free memory management module
  - E.g., in the NOBLE library

Reasoning about Correctness

- **Linearizability** [Herlihy 1991]
  - An implementation of a concurrent data structure is **linearizable** if, for every concurrent execution, its memory accesses are linearizable,
    i.e., there should exist a semantically equivalent *sequential* execution (assuming a sequential memory consistency model) that respects the partial order of the data structure access operations in the concurrent execution.
- Basic steps for proof of linearizability:
  - Define precise sequential semantics
  - Define linearizability points of the data structure operations (e.g., the CAS positions)
  - Show that operations take effect atomically at these points with respect to sequential semantics
  - Creates a total order using the linearizability points that respects the partial order

Summary: Lock-free concurrent data structures

- Optimistic manipulation of concurrent shared data structures
- Uses atomic primitives, esp. CAS
  - Fix ABA problem of CAS
  - Memory Management
- Common concurrent shared data structures
  - Stack
  - Queue
  - Deque (doubly-ended queue)
  - Priority Queue
  - Dictionary
  - Hash Table
  - Linked Lists
  - Skip List
  - Bag

Transactional Memory

Towards Transactional Memory

- **Critical sections**
  - Operations on shared data, should be executed atomically
  - Risk for races, deadlocks
- **Mutex locks** → serialization of threads
  - Safe but overly restrictive – a data race does not lead to incorrect results every time
- **Transactional Memory**
  - Mark critical sections by **atomic** { … }, leave implementation to the system (e.g., hardware)
  - System can speculate on serializability of actual accesses at runtime …
  - … and roll back if speculation was wrong

Atomic Transactions, Serializability

Serialization (in some arbitrary order, dep. on scheduler) **implies** race-free-ness, but not vice versa.

Serialization is overly restrictive.

Non-serialized execution *can* still be correct if it is ***serializable***, i.e. could be converted into serialized form by swapping non-conflicting memory accesses

Example: Shared variables A, B; transactions T0, T1 e.g. { a=A=..A..; B=..B..a..; }

T0:	T1:
…	…
read(A)	
write(A)	
read(B)	
write(B)	
	read(A)
	write(A)
	read(B)
	write(B)

Serialized as T0→T1, e.g. using a mutex lock

T0:	T1:
…	…
read(A)	
write(A)	
	read(A)
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Concurrent, but serializable as T0→T1

T0:	T1:
…	…
read(A)	
	read(A)
write(A)	
	write(A)
read(B)	
write(B)	
	read(B)
	write(B)

Incorrect – not serializable, not atomic

TM Example: Lock-based vs. Transactional Map (Hash Table) Data Structure

```
class LockBasedMap
    implements Map
{
    Object mutex;
    Map m;

    LockBasedMap (Map m) {
        this.m = m;
        mutex = new Object();
    }

    public Object get() {
        synchronized (mutex) {
            return m.get();
        }
    }

    // other Map methods. . .
}
```

```
class AtomicMap
    implements Map
{
    Map m;

    AtomicMap (Map m) {
        this.m = m;
    }

    public Object get() {
        atomic {
            return m.get();
        }
    }

    // other Map methods. . .
}
```

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.

Example: Thread-safe composite operation

- Move a value from one concurrent hash map to another
- Threads see each key occur in exactly one hash map at a time

```
void move (Object key) {
  synchronized (mutex ) {
    map2.put ( key, map1.remove(key));
  }
}
```

Requires (coarse-grain) locking (does not scale)

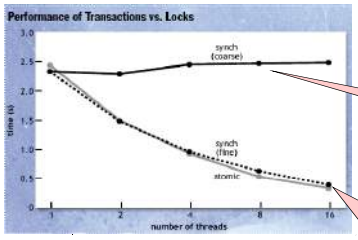
or rewrite hashmap for fine-grained locking (error-prone)

```
void move (Object key) {
  atomic {
    map2.put (key, map1.remove( key));
  }
}
```

Any 2 threads can work in parallel as long as different hash table buckets are accessed.

Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.

Transactions vs. Locks

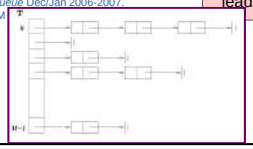


Source: A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007. © ACM

Times for a sequence of Java HashMap insert, delete, update operations, run on a 16-processor shared memory multiprocessor

Coarse-grained locking à la Java "synchronized" does not scale up

Fine-grained locking scales here equally well as atomic transactions, but is more low-level, susceptible to bugs leading to deadlocks or races



Atomic Transactions

- For atomic computations on multiple shared memory words
- Abstracts from locking and mutual exclusion
  - coarse-grained locking does not scale
  - declarative rather than hardcoded atomicity
  - enables lock-free concurrent data structures
- Transaction either commits or fails (then to be repeated)

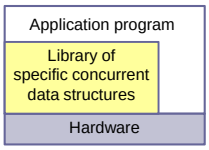
Transactional Memory / Transactional Programming

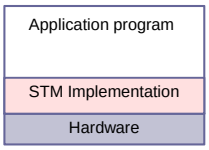
- Variant 1: **atomic** { ... } marks transactional code
- Variant 2: special transactional instructions e.g. **LT**, **LTX**, **ST**; **COMMIT**; **ABORT**
- Implem.: Speculate on serializability of unprotected execution, track time-stamped values of memory cells, and roll-back+retry transaction if speculation was wrong
  - STM Software transactional memory (high overhead)
  - Hardware TM, implemented e.g. as extension of cache coherence protocols [Herlihy,Moss'93], Sun Rock, Intel Haswell

Literature on Transactional Memory

- Good introduction:
  - A. Adl-Tabatabai, C. Kozyrakis, B. Saha: Unlocking Concurrency: Multicore Programming with Transactional Memory. *ACM Queue* Dec/Jan 2006-2007.
- Transaction concept comes from database systems

Nonblocking Concurrent Data Structures vs. Software Transactional Memory





API:
DS-specific operations:
Insert, delete, push, pop, …

Specifically implemented and optimized

API:
Load-transactiona, Store-transactional, Commit, Abort…

General-purpose

Remark: Fine-grained locking can sometimes be a good alternative, too.

Questions?

Further References

- M. Michael: The balancing act of choosing nonblocking features. *Communications of the ACM* **56**(9), Sep. 2013.
- M. Herlihy, N. Shavit: *The Art of Multiprocessor Programming.* Morgan-Kaufmann, 2008.
- Ph. Tsigas and Y. Chang: A simple, fast and scalable non-blocking concurrent FIFO queue for shared memory multiprocessor systems. *Proc. ACM SPAA*, 2001
- Ph. Tsigas and Y. Chang: Integrating Non-blocking Synchronizsation in Parallel Applications: Performance Advantages and Methodologies. *Proc. 3rd ACM Workshop on Software and Performance (WOSP'02),* 2002
- Håkan Sundell: *Efficient and Practical Non-Blocking Data Structures.* PhD thesis, Chalmers University of Technology, Göteborg 2004
- Daniel Cederman: *Concurrent Algorithms and Data Structures for Many-Core Processors*. PhD thesis, Chalmers University of Technology, Göteborg, 2011
- NOBLE – Library of non-blocking data structures www.cse.chalmers.se/research/group/noble/

Acknowledgments

- Some slide material courtesy of Håkan Sundell, Univ. Borås and Parallel Scalable Solutions AB