Highly Efficient Synchronization Techniques

Panagiota Fatourou

FORTH ICS &
University of Crete

Presentation given at Technion on July 6, 2011

joint work with Nikolaos Kallimanis

Parallelism versus Synchronization

- Exploiting parallelism in an efficient way is currently one of the most outstanding challenges of computer systems!

Ideal World

- Upgrading from a uniprocessor to an n-way multiprocessor should provide about an n-fold increase in computational power.

Practice

- This has almost never happened!!!!!
  - This is due to the cost of inter-processor communication and synchronization.

Example (Herlihy & Shavit, *The Art of Multiprocessor programming*, Morgan Kaufmann, 2008)

- Five friends decide to paint a five room house.

☹ What happens if one room is twice as big as each of the other rooms?

Parallelism versus Synchronization

- If each friend is assigned a room to paint, the time required to paint the big room will be much more than the time required to paint the rest.
- The extent to which we can speed up any complex job is limited by how much of the job can be executed sequentially.
- Using Amdahl's Law we can prove that although we have 5 workers, the speedup we obtain is only 3-fold ☹

Idea for doing better

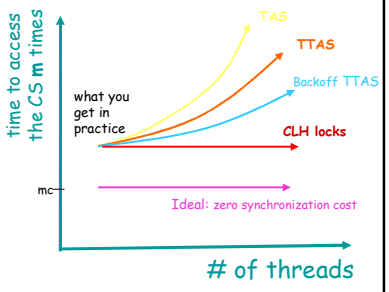
- As soon as a painter's work in a room is done, s/he helps others to paint the remaining room.
- Substantial communication and synchronization is needed!!!

➨ **Achieving synchronization is a hard and costly task ☹**

- **This need for synchronization appears in a vast majority of concurrent applications.**

The cost of synchronization

- Assume that n threads need to synchronize m times, i.e., they should access the critical section (CS) m times in total.
- Each time the threads need to access the critical section, they pay some synchronization cost to achieve mutual exclusion.
- **Ideal World (the objective is to minimize the time to execute the CS m times):** One thread undertakes the task to execute the critical section as many times as needed (i.e., m times in our case) and the rest of the threads do "nothing", so synchronization cost is zero.
  - In this case, the synchronization cost would be mC time units, where C is a constant indicating the time units required to execute the critical section once.
  - **Significant remark:** In the ideal world, as long as m is fixed, the cost of accessing the CS m times is always mC time units, independently of the number n of threads.



In Practice

- This is never the case since extra cost should be paid by the threads to achieve synchronization.

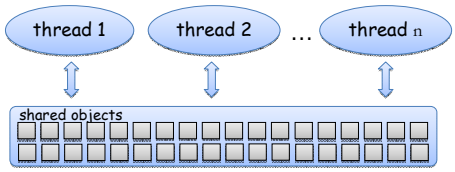
Parallelism versus Synchronization

- After some point, achieving more parallelism may incur high cost for synchronization.
- Parallelism is desirable only if the cost of synchronization incurred for achieving it is not too big.

➢ Achieving synchronization in an efficient way is the problem studied in this presentation!

Our Model

- The threads are asynchronous and communicate by accessing primitive shared objects.
- A CAS object O stores a value and supports two atomic operations:
  - read(O): returns the value of O
  - CAS(O, old, new) which checks if the value of O is equal to old and if this is the case it changes it to new and returns TRUE; otherwise, the value of the object remains unchanged and FALSE is returned.
- An LL/SC object O stores a value and supports two atomic operations:
  - LL(O): returns the current value of O
  - SC(O, v) is successful if and only if no thread has performed a successful SC on O since the execution of p's latest LL on O. If it is successful the value of O changes to v and TRUE is returned; otherwise, SC fails to modify O and returns FALSE.



- A Fetch&Add object O stores a value and supports two operations:
  - read(O): returns the value of O.
  - Fetch&Add(O, v): adds v to O and returns the old value.
- The threads may fail by crashing.

Correctness and Progress

- We assume that each thread wants to repeatedly execute tasks, called operations, which, when executed concurrently to the operations of other threads, require some synchronization to take place.

Correctness – Linearizability (Herlihy & Wing, "Linearizability: A Correctness Condition for Concurrent Objects", *ACM Transactions on Programming Languages and Systems*, Vol. 12, No. 3, pp. 463-492, July 1990)

- For each (parallel) execution a produced by the algorithm, there is a serial execution σ of the operations executed in α, such that:
  - each operation has the same response in σ as in a, and
  - σ respects the partial order determined by the execution intervals of the operations in a.

Progress

Wait-Freedom

- **Each** thread finishes the execution of its operation within a finite number of its own steps.

➧ Wait-free algorithms are highly fault-tolerant!

Lock-Freedom

- **Some** thread finishes the execution of an operation within a finite number of steps.

➧ Starvation is possible!

Blocking Algorithms

- A thread **may have to wait** for an action that will be caused by some other thread before it finishes the execution of its operation.

➧ A failure may cause the system to halt!

Our results [SPAA 2011]

- We present a technique for achieving synchronization. The technique is simple and highly efficient.
- Experiments show that our technique outperforms all current state-of-the-art synchronization techniques.
- It does so although it ensures wait-freedom in executing the threads' operations which is a much stronger progress property than what is provided by the techniques that it outperforms.
- We apply this technique to design wait-free implementations of common concurrent data structures, like stacks and queues.
- Experiments show that our stack (queue) implementation outperforms all current state-of-the-art concurrent stack (queue, respectively) implementations, and this given that it is wait-free whereas state-of-the-art implementations satisfy weaker progress properties.

Our Wait-FreeTechnique

```
RetVal Apply(function op, Pindex i){
   Update(Col, i, op);
   Attempt();
   retval = S.rvals[i];
   Update(Col, i, ⊥);
   Attempt();
   return retval;
}
```

p_i executes Attempt to apply its operation

p_i announces its operation to Col

p_i updates its component with the special value ⊥

p_i executes Attempt once more to eliminate any evidence of op

```
shared CollectObject Col;
```

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

```
shared StateStruct S;
```

State st; ⊥

Boolean applied[1..n];

F	F	F	F	F	F	F	F

RetVal rvals[1..n];

⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

```
void Attempt() {
   StateStruct ls;
   Pindex k;
   Operation ops[1..n];
   repeat twice {
      ls = LL(S);
      ops = Collect(Col);
      for i=1 to n do {
         if(ops[i] ≠ ⊥ AND ls.applied[i]==FALSE)
            apply ops[i] to ls.st and store
             into ls.rvals[i] the return value;
         if(ops[i] ≠ ⊥) ls.applied[i] = TRUE;
         else ls.applied[i] = FALSE;
      }
      SC(S, ls);
   }
}
```

p_i makes a local copy of S

p_i collects all the announced operations

p_i applies all active ops to a local copy of S and calculates return values

p_i executes an SC instruction on S

Our Wait-FreeTechnique – How it works

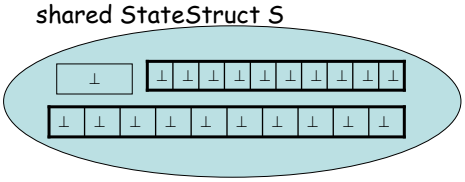
1. Thread p2 calls Apply(op2, 2);
2. Update(Col, 2, op2);
3. Attempt();

```
Repeat twice:
   ls = LL(S);
   ops = Collect(Col);
   for i=1 to n do {
      if(ops[i] ≠ ⊥ AND ls.applied[i]== ⊥)
         apply ops[i] to ls.st and store
          into ls.rvals[i] the return value;
      if(ops[i] ≠ ⊥) ls.applied[i] = T;
      else ls.applied[i] = ⊥;
   }
   SC(S, ls);
```

Collect Object

⊥	op2	⊥	op4	⊥	⊥	⊥	⊥	⊥	⊥

shared StateStruct S

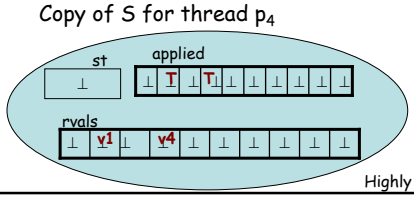


4. Thread p4 calls Apply(op4, 4);
5. Update(Col, 4, op4);
6. Attempt();

ops

⊥	op1	⊥	op4	⊥	⊥	⊥	⊥	⊥	⊥

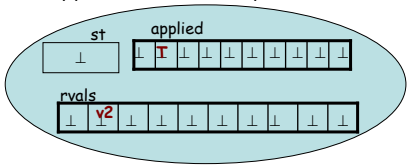
Copy of S for thread p_4



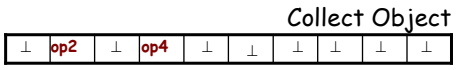
ops

⊥	op1	⊥	⊥	⊥	⊥	⊥	⊥	⊥	⊥

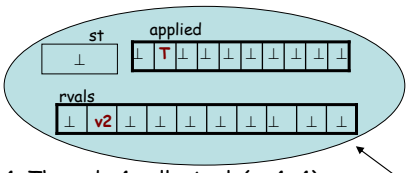
Copy of S for thread p_2



Our Wait-FreeTechnique – How it works



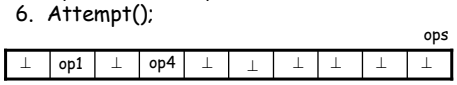
shared StateStruct S



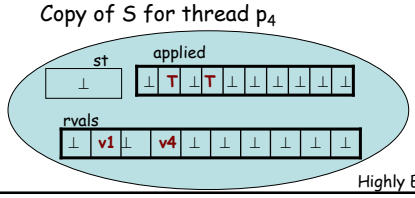
1. Thread p1 calls Apply(op2, 2);
2. Update(Col, 2, op2);
3. Attempt();

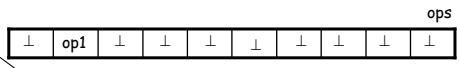
```
Repeat twice:
    ls = LL(S);
    ops = Collect(Col);
    for i=1 to n do {
        if(ops[i] ≠ ⊥ AND ls.applied[i]== ⊥)
            apply ops[i] to ls.st and store
            into ls.rvals[i] the return value;
        if(ops[i] ≠ ⊥) ls.applied[i] = T;
        else ls.applied[i] = ⊥;
    }
7.  SC(S, ls);
```

4. Thread p4 calls Apply(op4, 4);
5. Update(Col, 4, op4);
6. Attempt();

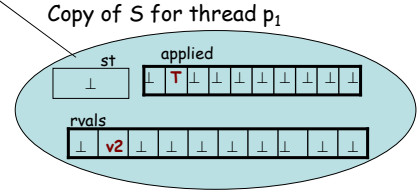


Copy of S for thread p_4

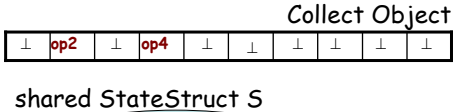




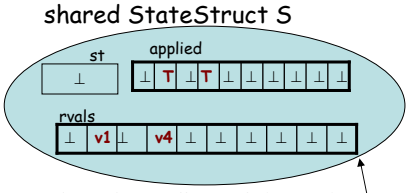
Copy of S for thread p_1



Our Wait-FreeTechnique – How it works



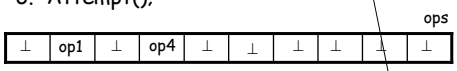
shared StateStruct S



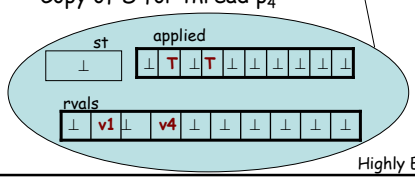
1. Thread p1 calls Apply(op2, 2);
2. Update(Col, 2, op2);
3. Attempt();

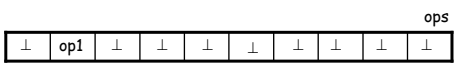
```
Repeat twice:
    ls = LL(S);
    ops = Collect(Col);
    for i=1 to n do {
        if(ops[i] ≠ ⊥ AND ls.applied[i]== ⊥)
            apply ops[i] to ls.st and store
            into ls.rvals[i] the return value;
        if(ops[i] ≠ ⊥) ls.applied[i] = T;
        else ls.applied[i] = ⊥;
    }
SC(S, ls);
```

4. Thread p4 calls Apply(op4, 4);
5. Update(Col, 4, op4);
6. Attempt();

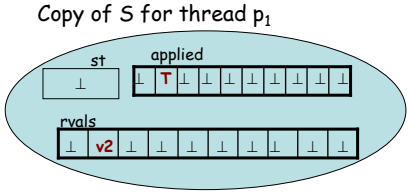


Copy of S for thread p_4

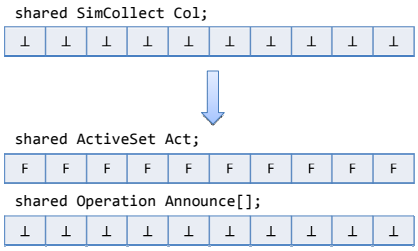




Copy of S for thread p_1



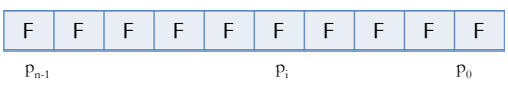
PSim: How can we implement the technique efficiently



- We implement the collect object in a highly efficient way using an active set and an array, called Announce, of n entries, one for each thread, which stores the operation that the thread wants to apply.
- An active set is an object that identifies a set of threads participating in some computation. It supports three operations:
  - **Join**: is called by a thread to identify its willingness to join the computation
  - **Leave**: to request removal from the set of participating threads
  - **GetSet**: returns the set of the currently participating threads.
- When p_i wants to perform an operation op, announces it by writing op (and its parameters) in Announce[i] and performs Join.
- Apply discovers the operations to be performed by first performing GetSet and then reading the appropriate entries of Announce as indicated by the result of GetSet.

A highly-efficient implementation of Active Set using Fetch&Add

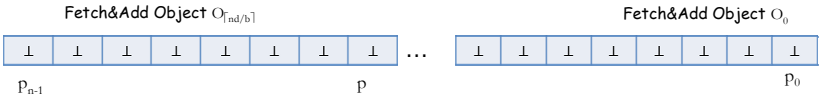
- An active set can be trivially implemented using a Fetch&Add object O containing n bits (1 bit per thread).
- The state (TRUE/FALSE) of thread p_i is recorded at the i-th bit of O (all bits of O are initialized to FALSE).
- Join(p_i) is implemented with a simple Fetch&Add(O, 2^i).
- Leave(p_i) is implemented with a simple Fetch&Add(O, -2^i).
- GetSet is implemented by reading O and returns those threads for which their corresponding bits of O are TRUE.



Fetch&Add object of n bits, all initialized to FALSE (F)

A highly-efficient implementation of Active Set using Fetch&Add

- Several shared memory machines (i.e., x86) support Fetch&Add on up to 64 bit words.
- ⚠ In case $n>64$, the active set can be implemented using more than one Fetch&Add objects, since active set implementations need not satisfy linearizability.
- A typical machine stores 512 bits in each cache-line. Thus, for up to 512 threads, the above algorithm would cause just one cache miss.



A highly-efficient implementation of Active Set using Fetch&Add

- The two experiments below were performed on a machine with four AMD Opteron 8350 processors:
  - Each processor consists of 4 cores.
  - Communication among cores of the same processor is achieved with a fast L3 cache.
  - The processors communicate through hyper-transport links.

Experiment 1 – Active Set Implementation based on Fetch&Add()

- 10^6 Join() or Leave() operations and 10^6 GetSet() are executed in total (independently of the number n of threads that execute them); each thread executes $10^6/n$ Join() or Leave() operations and $10^6/n$ GetSet() operations.
- The performance of the algorithm for large values of n is shown in Figure 1 and resembles a flat line.

Experiment 2 – Collect Object Implementation based on Fetch&Add()

- 10^6 Update() operations and 10^6 Collect() are executed in total (independently of the number n of threads that execute them); each thread executes $10^6/n$ Update() operations and $10^6/n$ Collect() operations.
- The performance of the algorithm for large values of n is shown in Figure 2 and resembles a flat line.

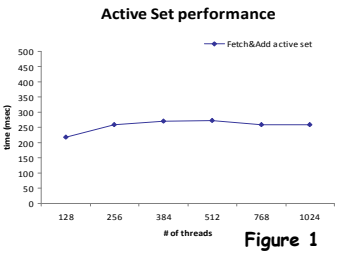


Figure 1

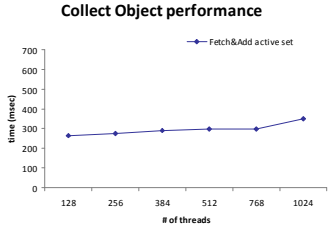
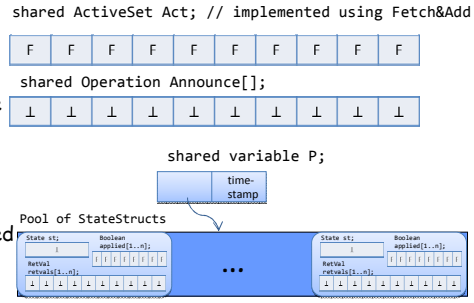


Figure 2

How can we implement Sim efficiently?

- Each thread keeps a pool of c StateStructs, c is a constant.
- A simple recycling scheme is used to recycle StateStructs.
- S has been replaced by a shared variable P which is an index in Pool.
- The majority of the shared memory machines supports CAS rather than LL/SC.
  - We simulate an LL on P with a read(P), and an SC with a CAS on a time-stamped version of P to avoid the ABA problem.
  - Since P is just an index to the Pool of StateStructs, there are enough bits in a word (48 bits) to store both the index and the timestamp.



- We were able to get rid of the second phase of the algorithm were each thread has to perform an Update and call Attempt for a second time to eliminate the evidence of its last operation.
- An adaptive backoff scheme is used to achieve high degree of helping.

Full code of PSim at:
http://code.google.com/p/sim-universal-construction/

Experimental Evaluation

Benchmark

- 10^6 Fetch&Multiply instructions were executed (where each thread executes $10^6/n$ instructions). The number of Fetch&Multiply instructions that are executed are always 10^6 independently of the number n of threads that execute them.
- Notice that the Fetch&Multiply instructions must be executed sequentially (since they are all executed on the same variable), so high synchronization is incurred during the experiment.
  - The experiment aims at measuring the cost paid for achieving this synchronization.
- A random number of (up to 512) dummy loop iterations have been inserted between the execution of two consecutive Fetch&Multiply.
  - In this way, a realistic work load is simulated, long runs and unrealistic low cache miss ratios are avoided.
- This and the next experiments were performed on a 4-processor AMD Magny-Cours machine (32 cores)
  - Each processor consists of 2 dies and each die consists of 4 cores.
  - Communication among cores of the same die is achieved with a fast shared L3 cache.
  - The interconnection network between dies resembles a hypercube topology.

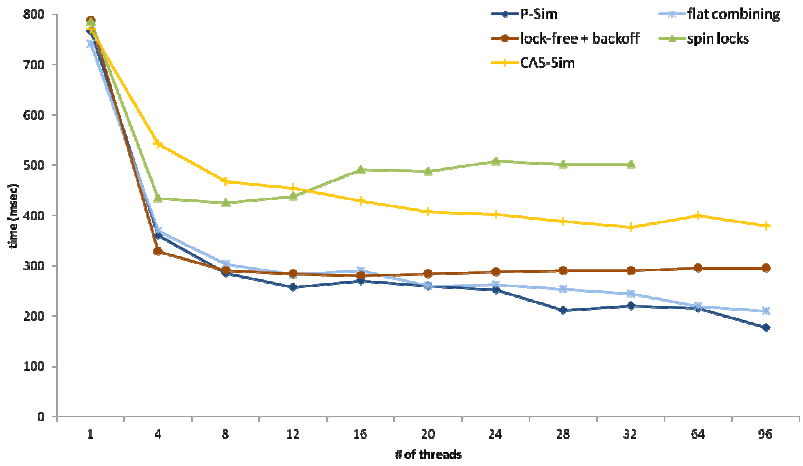
Evaluated Algorithms

- Flat-Combining [Hendler, Incze, Shavit, and Tzafrir, SPAA '10] -> blocking
- CLH spin-locks [Craig, 93] and [Magnusson, Landin, and Erik Hagersten, '94] -> blocking
- A simple lock-free implementation of Fetch&Multiply using CAS which employs a backoff scheme.
  - it uses a CAS object O and executes a CAS on O, repeatedly, until it successfully stores the new value there; it employs exponential backoff to reduce contention
- A lock-free version of PSim in which Fetch&Add is implemented using CAS in a way similar to that employed by the simple lock-free implementations described in the previous item.

PSim Performance

PSim is up to:

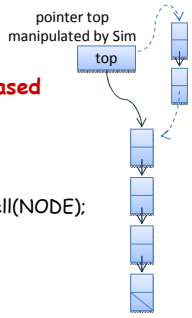
- 2.36 times faster than spin locks
- 1.67 times faster than the simple lock-free algorithm
- better numbers than for flat combining although PSim is wait-free



Why is PSim so efficient?

- Although it is wait-free, it is very simple.
- Due to helping and the fact that each thread works in parallel on a local copy, it achieves performance which is not far from the ideal.
- The algorithm performs a small number of CAS in total (due to the helping mechanism), so the contention and the cache misses are low.
- It employs a highly-efficient implementation of an active set (and of a collect object) based on Fetch&Add which is scalable and fast.

Applications: How to implement a wait-free concurrent stack using PSim? The SimStack Algorithm



A Simple Lock-Based Implementation:

```
void push(DATA x) {
   NODE * oldTop
    NODE *n = newcell(NODE);
    n->value = x;

   lock(L);
   n->next = Top;
   Top = n;
   unlock(L);
}

T pop(void) {

    lock(L);
    Top = Top->next;
     unlock(L);
}
```

- In a way similar to the simple lock-based implementation, PSim is employed to atomically manipulate pointer top (i.e., PSim does not store in StateStruct the entire stack state but only the value of top).
- In SimStack, during the execution of an operation, a thread does the following:
  - It uses a set of nodes allocated by itself, one for each Push it will perform (on behalf of others or itself);
  - it creates a linked list containing these nodes;
  - it initiates the next field of the last element of the list to point to the currently topmost element of the stack, and
  - uses PSim to update top to point to the first element of this list.
  - if Pop() operations are also to be helped, it applies some kind of (local) elimination (if possible), or it appropriately updates top using PSim.

Experimental Evaluation of SimStack

The experiment

- 10^6 pairs of Push() and Pop() operations were performed in total, independently of the number of threads n; each of the n threads performs $10^6/n$ pairs of Push() and Pop() operations, first a Push() and then a Pop().

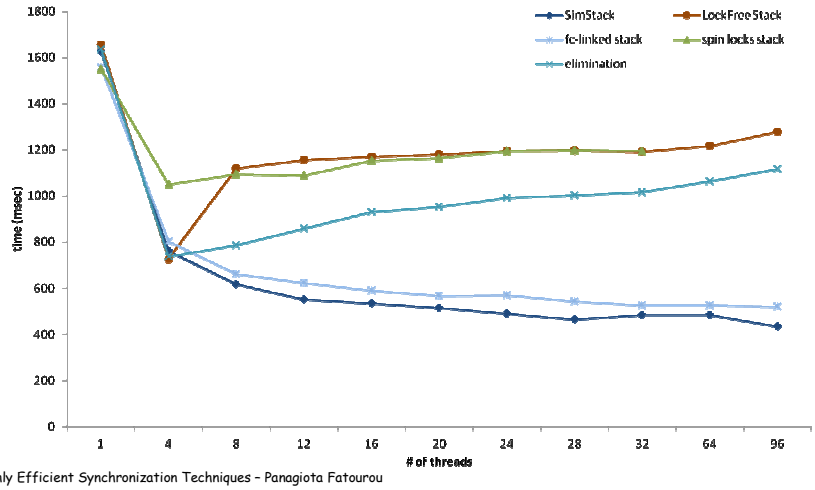
Evaluated Stack Implementations

- A linked stack implementation based on Flat-Combining [Hendler, Incze, Shavit, and Tzafrir, SPAA '10] -> blocking
- A linked stack implementation based on CLH spin-locks [Craig, 93] and [Magnusson, Landin and Erik Hagersten, '94] -> blocking
- The elimination backoff stack [Hendler, Shavit, and Yerushalmi, SPAA '04] -> lock-free
- The lock-free stack implementation by Treiber, '86.

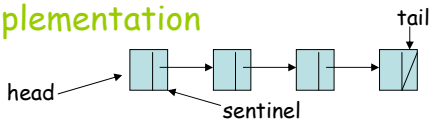
SimStack Performance

SimStack is up to:

- 2.94 times faster than lock-free stack
- 2.58 times faster than spin lock based stack
- 2.57 times faster than elimination backoff stack
- 1.17 times faster than flat combining



A Simple Lock-Based Queue Implementation (Michael & Scott, PODC'96)



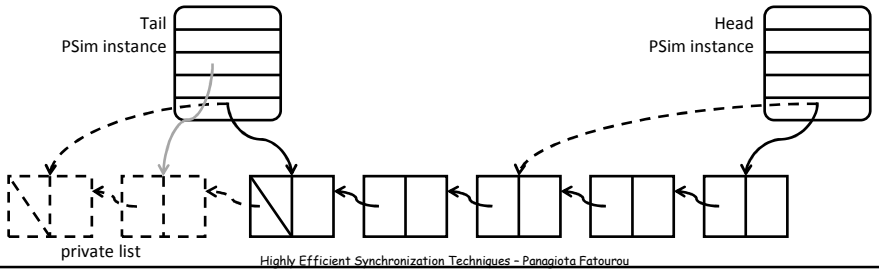

```
typedef struct node {
  T value;                    // initially, there is a sentinel node in the queue where
  struct node *next;          // Head and Tail point to.
} NODE;                       // the sentinel is not always the same node
shared NODE * Head, *Tail;    // initially, HeadL = TailL = FREE
                              // two locks, EnqLock and DeqLock, are used to ensure that at
// most one enqueuer, and at most one dequeuer at a time can manipulate the nodes of the queue.
```

```
void enq(T x) {
  NODE *n = new(NODE);
  n->value = x;
  n->next = NULL;

  lock(TailL);
  Tail->next = n;
  Tail = n;
  unlock(TailL);
}
```

```
T deq(void) {
  T result;

  lock(HeadL);
  if (Head->next == NULL)
      result = EMPTY_QUEUE;
  else {
      result = Head->next->value;
      Head = Head->next;
  }
  unlock(HeadL);
  return result;
}
```


Applications: How to implement a wait-free concurrent queue using PSim? The SimQueue Algorithm

- Two instances of PSim are used (one for manipulating the head pointer and another for the tail pointer).
- Enqueuers and dequeuers may run independently and simultaneously .
- Dequeuers help only dequeuers.
  - Each thread traverses the list and tries to change the Head pointer.
- Similarly, enqueuers help only enqueuers.
  - Each thread creates a private list with the nodes of the enqueue operations that helps.
  - After that, it executes an ApplyOp to the Tail instance of PSim trying to change the Tail pointer.
  - In case of success, it also announces two extra pointers, one to the first node v of the private list, and another to the node u that Tail was pointing to.
  - The thread that successfully executes ApplyOp try to update the next pointer of node u to point to v by executing a CAS instruction.
  - This action is performed by both enqueuers and dequeuers to ensure correctness.



Experimental Evaluation of SimQueue

The experiment

- 10^6 pairs of Enqueue() and Dequeue() operations were performed in total, independently of the number of threads n; each of the n threads performs $10^6/n$ pairs of Enqueue() and Dequeue() operations, first an Enqueue() and then a Dequeue().

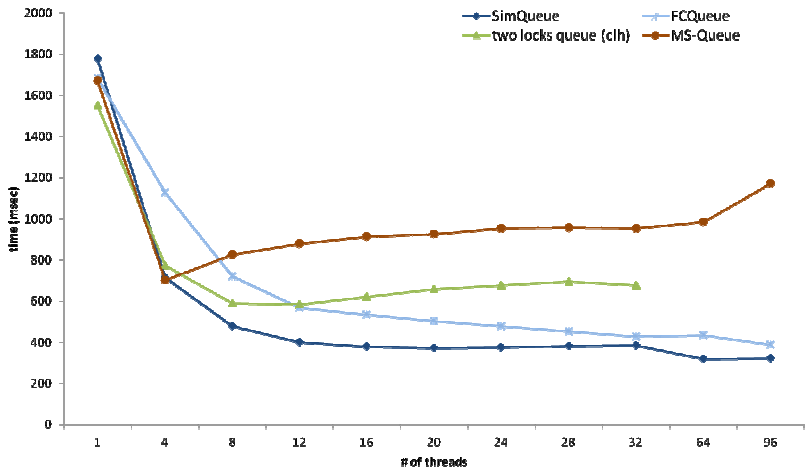
Evaluated Queue Implementations

- A queue implementation based on Flat-Combining [Hendler, Incze, Shavit, and Tzafrir, SPAA '10] -> blocking
- The two lock queue implementation [Michael and Scott, PODC '96], where CLH locks were employed -> blocking
- The lock-free queue implementation [Michael and Scott, PODC '96].

SimQueue Performance

SimQueue is up to:

- 3.06 times faster than lock-free queue
- 1.82 times faster than the spin-lock based queue
- 1.5 times faster than flat combining

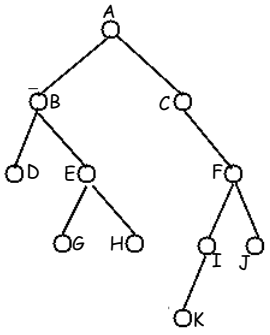


Applicability

- What is the cost of applying k concurrent search operations on a tree?

- A "sophisticated" concurrent implementation *(see e.g., F. Ellen, P. Fatourou, E. Ruppert, F. van Breugel: Non-blocking binary search trees. PODC 2010: 131-140)* could allow multiple searches proceeding in parallel and being executed in O(h) time in total, where h is the height of the tree.

- If we consider each search to comprise the operation that is to be performed by each thread, PSim would require Ω(kh) time.
- Thus, multiple concurrent instances of PSim should be employed in this case to achieve good performance. Achieving this may not be trivial.



Theoretical Significance of Sim

- A universal construction is a shared object that can simulate any type of other shared objects; it supports one operation:
  - `Apply`(op,p) applies operation op (executed by thread p) to the state of the simulated object and returns the response of op to p.

- A Collect object can trivially be implemented (in a way similar to the implementation of an active set) using just a single Fetch&Add object, of large enough size to contain n values.
- So, under the standard **theoretical** assumption that the size of objects may be large, Sim uses a constant number of large objects and performs O(1) operations on them.

- To the best of our knowledge, this is the first wait-free universal construction that achieves constant complexity both in terms of the number of primitive objects used and the number of operations performed on the primitive objects.

Theoretical Significance of Sim

- Jayanti [PODC'98] proved a lower bound of Ω(logn) on the step complexity (i.e., the number of primitive operations) of any (oblivious) implementation of a universal object from LL/SC objects.
- An oblivious universal construction does not exploit the semantics of the object being simulated.
- This lower bound holds even if the objects are of unbounded size.
- One of the open problems mentioned in Jayanti's paper is the following:

  "*"If shared memory supports all of read, write, LL/SC, swap, CAS, move, Fetch&Add, Fetch&Multiply, would the Ω(logn) lower bound still hold?"*
- Sim is oblivious. Thus, it proves that this lower bound can be beaten by using just one Fetch&Add object in addition to an LL/SC object.

Summary

- Sim is a new simple **wait-free** universal construction that has constant step complexity; it employs just one (large) Fetch&Add object in addition to one (large) LL/SC object.
- **Sim works even if the number n of threads is not known** (with appropriate modifications that will be described in the journal version).
- PSim is a highly-efficient version of Sim that has been implemented in a real shared-memory machine.
- In theory terms, PSim has worse complexity that its theoretical analog.
- However, experiments show that PSim outperforms state-of-the-art lock-based and lock-free synchronization techniques, despite the fact that they satisfy weaker progress conditions.
- Based on PSim, highly-efficient wait-free implementations of concurrent stacks and queues have been designed.
- Experiments show that SimStack and SimQueue outperform all state-of-the-art shared stack and queue implementations.
- SimStack and SimQueue ensure wait-freedom which is a much stronger progress property than all other implementations.

The Performance Advantage of Fetch&Add

- The universality result [Herlihy, TOPLAS'91] has motivated major hardware manufacturers to include strong primitives, like LL/SC or CAS in the instruction set of most modern multiprocessors.
- The results of this paper provide good motivation for including Fetch&Add in the instruction set of more architectures in the future.

Questions?

Thank you!