An Almost Non-Blocking Stack

Hans-J. Boehm
HP Labs

Motivation

- Update data structures in signal/interrupt handlers.
  - Sampling code profilers.
  - Perhaps just log the signal.
- Underlying application may be multithreaded.
- Need to guard against concurrent accesses by
  - Multiple threads.
  - Multiple signal handlers.
  - Thread code and signal handler.
- But locking in signal handlers is unsafe.
  - In Pthreads, pthread_mutex_lock is not "async-signal-safe".
  - Interrupted thread may already hold lock ➔ signal handler can't safely reacquire it.

The “obvious” solution

- Use lock-free data structures!
  - Blocked processes (e.g. interrupted by signal) are no problem.
- Many known algorithms
  - Linked stack solution dates back to Treiber, 1986.
- But for pointer-based operations these algorithms either:
  - Require CAS instruction wide enough for a (pointer, version) pair, with “wrap-proof” version number, or
  - Constrain the underlying storage manager to prevent reuse, or
  - Are reasonably complex, and usually use per-thread memory.
- They are also completely lock-free, more than we require.

The real requirements:

- At most *n* threads; main program and single handler share data structure; handler can't be reentered:
  - Requirement: With at most *n* inactive threads, a data structure access by an active thread will progress.
- Threaded main program and *n* handlers share data structure; handler can't be reentered; main program locks data structure accesses:
  - Requirement: With at most *n* inactive threads, a data structure access by an active thread will progress.
- We can bound the number of blocked threads (often by one).

A definition:

- A data structure is *N-non-blocking* (*N-lock-free*) if
  - It supports concurrent access by any number of concurrent threads.
  - If at least one active process is trying to access the data structure, then one such thread will make progress, provided
    - At most *N* inactive processes are concurrently trying to access the data structure.
- It is *almost non-blocking* (*almost-lock-free*) if it is *N*-non-blocking for some *N*.
- This is good enough for the signal-handler case.
- It helps for page fault or preemption tolerance.

Our specific algorithm

- We give a simple, performance competitive, almost non-blocking, linked stack implementation.
- Linked stack is illustrative, and often sufficient.
  - E.g. simple memory allocation.
    - But see also Michael, PLDI04
- Can be interface compatible with non-blocking implementation based in wide CAS.
  - Client assumes almost non-blocking.
  - Use wide CAS where available
    - E.g. X86-32, Intel X86-64, future Itanium, …
  - Use almost non-blocking algorithm where unavailable
    - E.g. AMD X86-64, current Itanium, …

Problem details

- Naïve pop operation fails:

```
/* WRONG !! */
node *pop(node **list)
{
   node *result, *second;
   do {
     result = *list;
     node *second = result -> next;
   } while (!CAS(list, result, second));
   return result;
}
```

- CAS may succeed if *list reverts to original value.
  - Known as “ABA problem”.

Our approach

- Combine two techniques and a hack:
  - Don’t reinsert list element that another thread is trying to remove.
    - Keep track of such elements in a black-list.
    - Similar to Michael’s “hazard pointers”.
  - Use very short version numbers.
    - Use nonzero version number only to make a newly inserted item different from a black-listed one.
  - Steal version number space from pointers:
    - Pointers, and list nodes normally have to be at least 4 byte aligned.
    - At least low order two bits of list pointers are zero.
    - Use low order two bits for version number.

List header layout

Pointer	ver

Black-listed Pointer 0 incl. version
Black-listed Pointer 1 incl. version

Our approach (contd.)

- Pop operation:
  - Insert head of list into black-list before attempting removal (requires CAS to find empty slot).
  - Remove black-list element when done.
- Push operation:
  - Check that inserted element is not in black-list.
  - If it is, increment version (*perturb* pointer), try again.
  - Requires read of black-list (typically 2 words).

The code

```
void push(node *perturbed * list,
          node * element,
          node *perturbed bl[])
{
    node *perturbed my_element =
          element;

  retry:
    for (int i = 0; i <= N; ++i) {
      if (bl[i] == my_element) {
        my_element =
             perturb(my_element);
        goto retry;
      }
    }
    do {
      node *perturbed first = *list;
      element -> next = first;
    } while (!CAS(list, first,
                  my_element));
}
```

```
node * pop(node *perturbed * list,
           node *perturbed bl[])
{
    unsigned bl_index;
  retry:
    node *perturbed result = *list;
    for (bl_index = 0; ; ) {
      if (CAS(&(bl[bl_index]), 0, result))
        break;
      if (++bl_index > N) bl_index = 0;
    }
    if (result != *list) {
      bl[bl_index] = 0;
      goto retry;
    }
    node *perturbed second =
               strip(result) -> next;
    if (!CAS(list, result, second)) {
      bl[bl_index] = 0;
      goto retry;
    }
    bl[bl_index] = 0;
    return strip(result);
}
```

The real code

```
void
HSD_list_insert(volatile HSD_list_ptr *list, HSD_list_element *x,
                HSD_list_aux *a)
{
  int i;
  AO_t x_bits = (AO_t)x;
  HSD_list_ptr next;

  /* No deletions of x can start here, since x is not currently in the    */
  /* list.                                                                 */

 retry:
  for (i = 0; i < HSD_BL_SIZE; ++i)
    {
      if (AO_load(a -> __list_bl + i) == x_bits)
        {
          /* Entry is currently being removed.  Change it a little.    */
            ++x_bits;
            if ((x_bits & _HSD_BIT_MASK) == 0)
              /* Version count overflowed; EXTREMELY unlikely, but possible. */
              x_bits = (AO_t)x;
          goto retry;
        }
    }
  /* x_bits is not currently being deleted */
  do
    {
      next = (HSD_list_ptr)AO_load((volatile AO_T *)list);
      x -> next = next;
    }
  while(!AO_compare_and_swap_release((volatile AO_T *)list, (AO_T)next,
                                     (AO_T)x_bits));
}
```

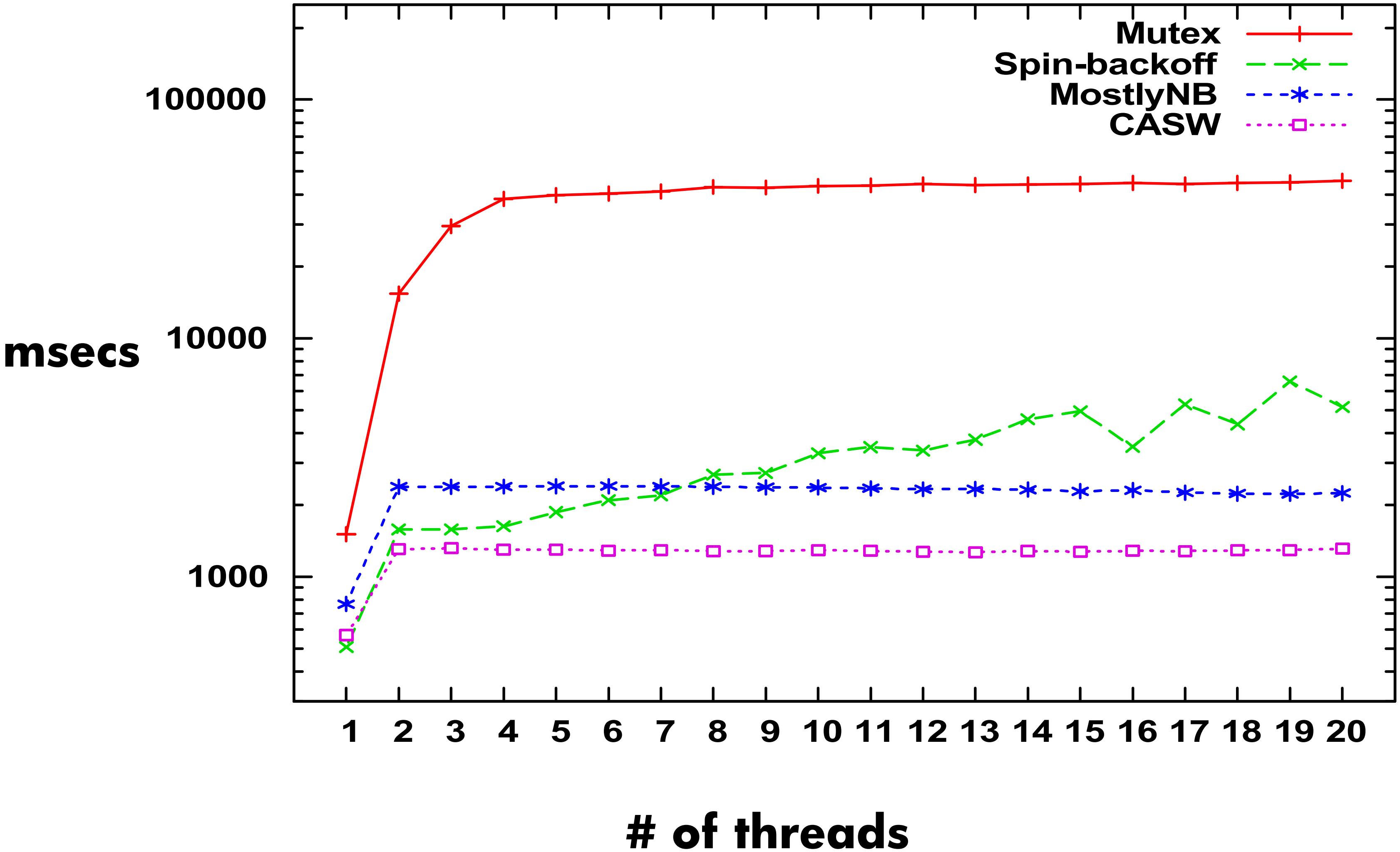
Exponential back-off

```
#ifdef __i386__
# define PRECHECK(a) (a) == 0 &&
#else
# define PRECHECK(a)
#endif

HSD_list_element *
HSD_list_remove(volatile HSD_list_ptr *list, HSD_list_aux * a)
{
  unsigned i;
  int j = 0;
  HSD_list_ptr first;
  HSD_list_element * first_ptr;
  HSD_list_ptr next;

 retry:
  first = (HSD_list_ptr)AO_load((volatile AO_T *)list);
  if (0 == first) return 0;
  /* Insert first into aux black list.
                                                          */
  /* This may spin if more than HSD_BL_SIZE removals using auxiliary      */
  /* structure a are currently in progress.
                                       */
  for (i = 0; ; )
    {
      if (PRECHECK(a -> __list_bl[i])
          AO_compare_and_swap_acquire((volatile AO_T *)(a->__list_bl+i), 0,
                                                              (AO_T)first))
        break;
      ++i;
      if ( i >= HSD_BL_SIZE )
        {
          i = 0;
          AO_pause(++j);
        }
    }
  assert(i >= 0 && i < HSD_BL_SIZE);
  assert(a -> __list_bl[i] == first);
  /* First is on the auxiliary black list.  It may be removed by      */
  /* another thread before we get to it, but a new insertion of x     */
  /* cannot be started here.
                                                                      */
  /* Only we can remove it from the black list.
                                       */
  /* We need to make sure that first is still the first entry on the  */
  /* list.  Otherwise it's possible that a reinsertion of it was      */
  /* already started before we added the black list entry.            */
  if (first != (HSD_list_ptr)AO_load((volatile AO_T *)list)) {
    AO_store_release((AO_T *)(a->__list_bl+i), 0);
    goto retry;
  }
  first_ptr = HSD_REAL_PTR(first);
  next = (HSD_list_ptr)AO_load((volatile AO_T *)&(first_ptr -> next));
  if (!AO_compare_and_swap_release((volatile AO_T *)list, (AO_T)first,
                                                        (AO_T)next)) {
    AO_store_release((AO_T *)(a->__list_bl+i), 0);
    goto retry;
  }
  assert(*list != first);
  /* Since we never insert an entry on the black list, this cannot have   */
  /* succeeded unless first remained on the list while we were running.   */
  /* Thus its next link cannot have changed out from under us, and we */
  /* removed exactly one entry and preserved the rest of the list.        */
  /* Note that it is quite possible that an additional entry was          */
  /* inserted and removed while we were running; this is OK since the */
  /* part of the list following first must have remained unchanged, and */
  /* first must again have been at the head of the list when the          */
  /* compare_and_swap succeeded.
                                                          */
  AO_store_release((AO_T *)(a->__list_bl+i), 0);
  return first_ptr;
}
```

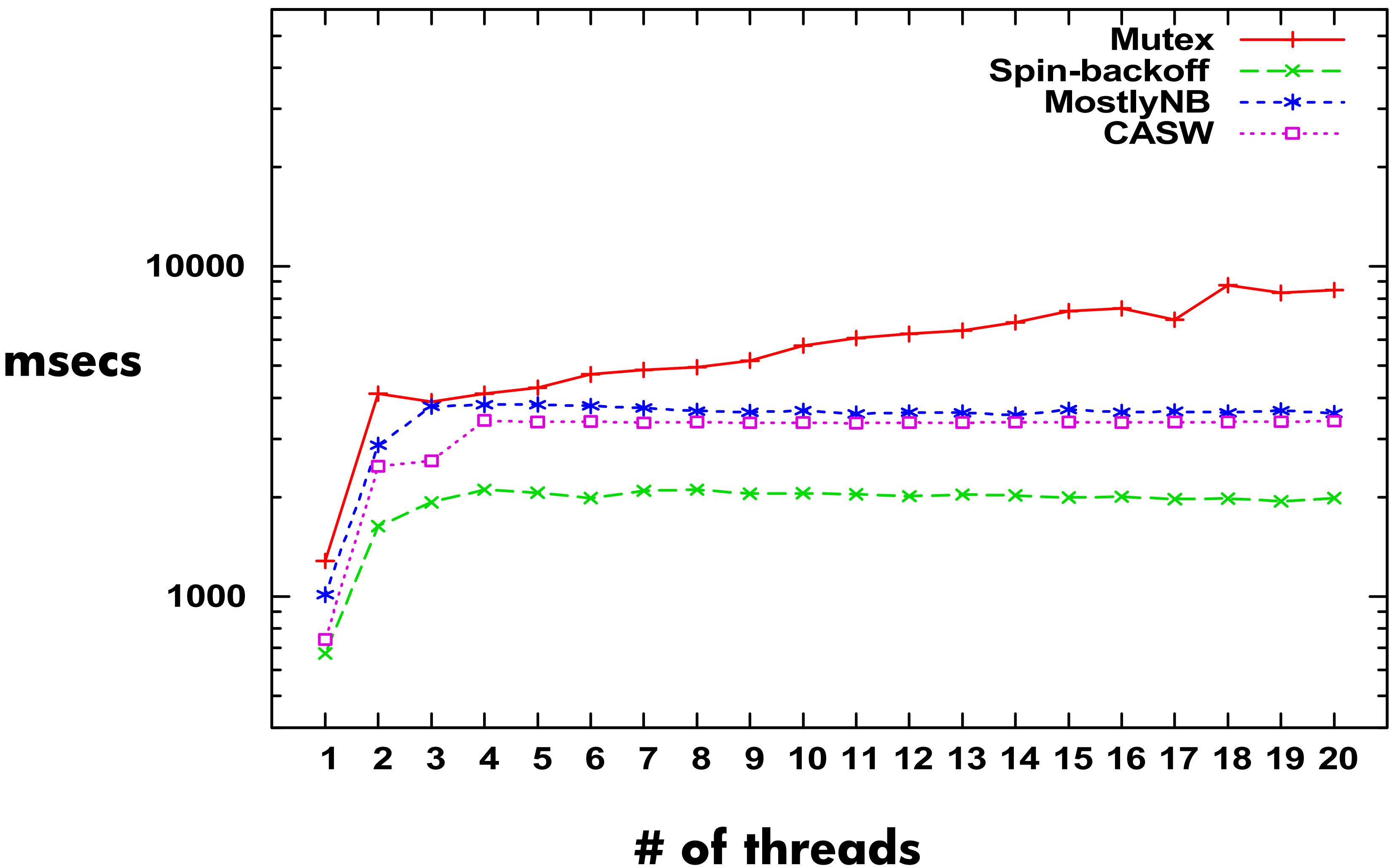
The benchmark

- The i^{th} of n threads alternately:
  - Pops i elements of the stack.
  - Pushes the i elements back onto the stack.
- All threads terminate at the end of their cycle when a global counter indicates a total of more than a million completed push and pop operations.
- Check that stack is permutation of original.
- Intentionally somewhat irregular.
- We report times in milliseconds (lower is better).
- Log scale to accommodate older `pthread_mutex_lock` implementations.

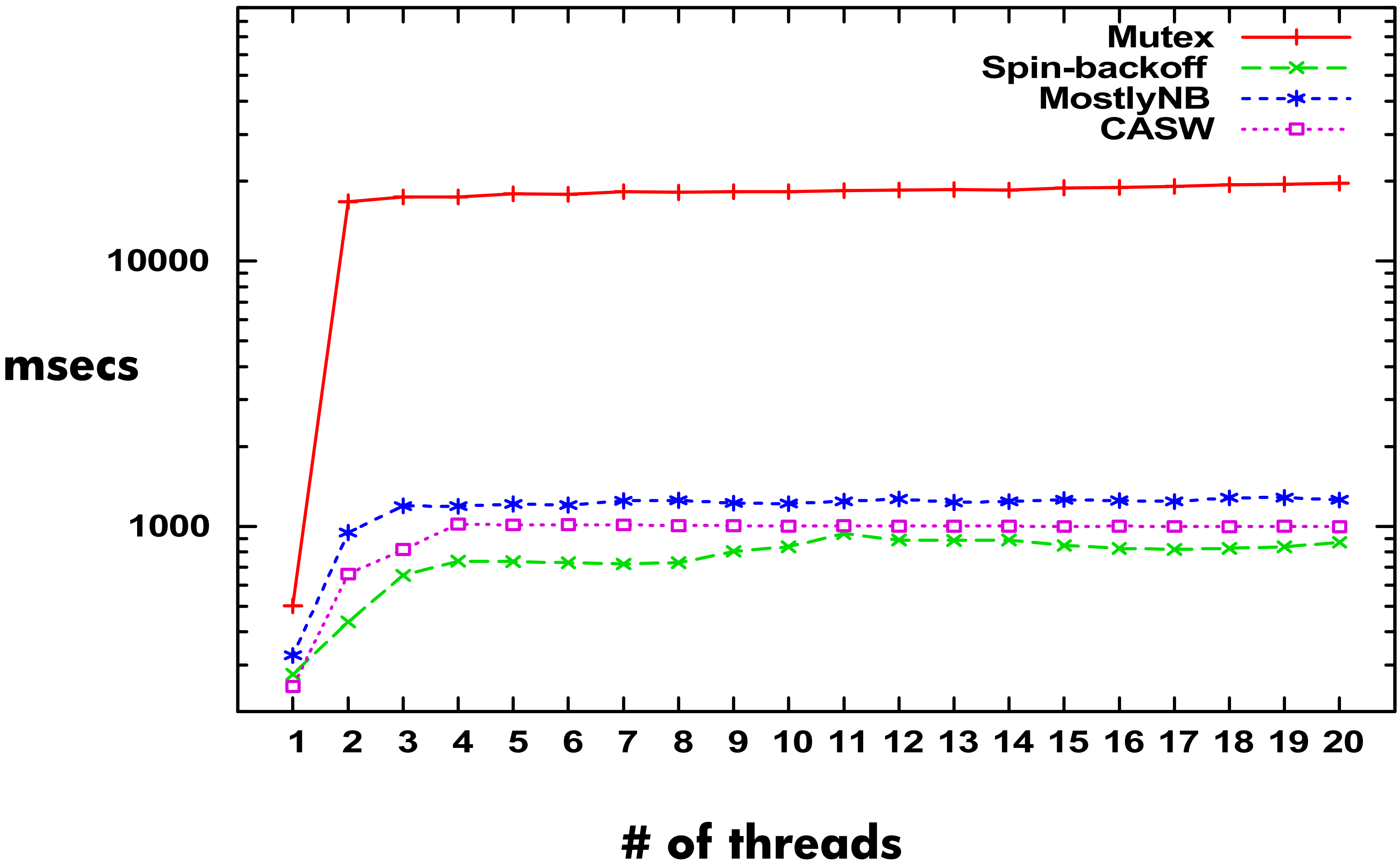
Benchmark execution time (2xPII/266, RedHat8)



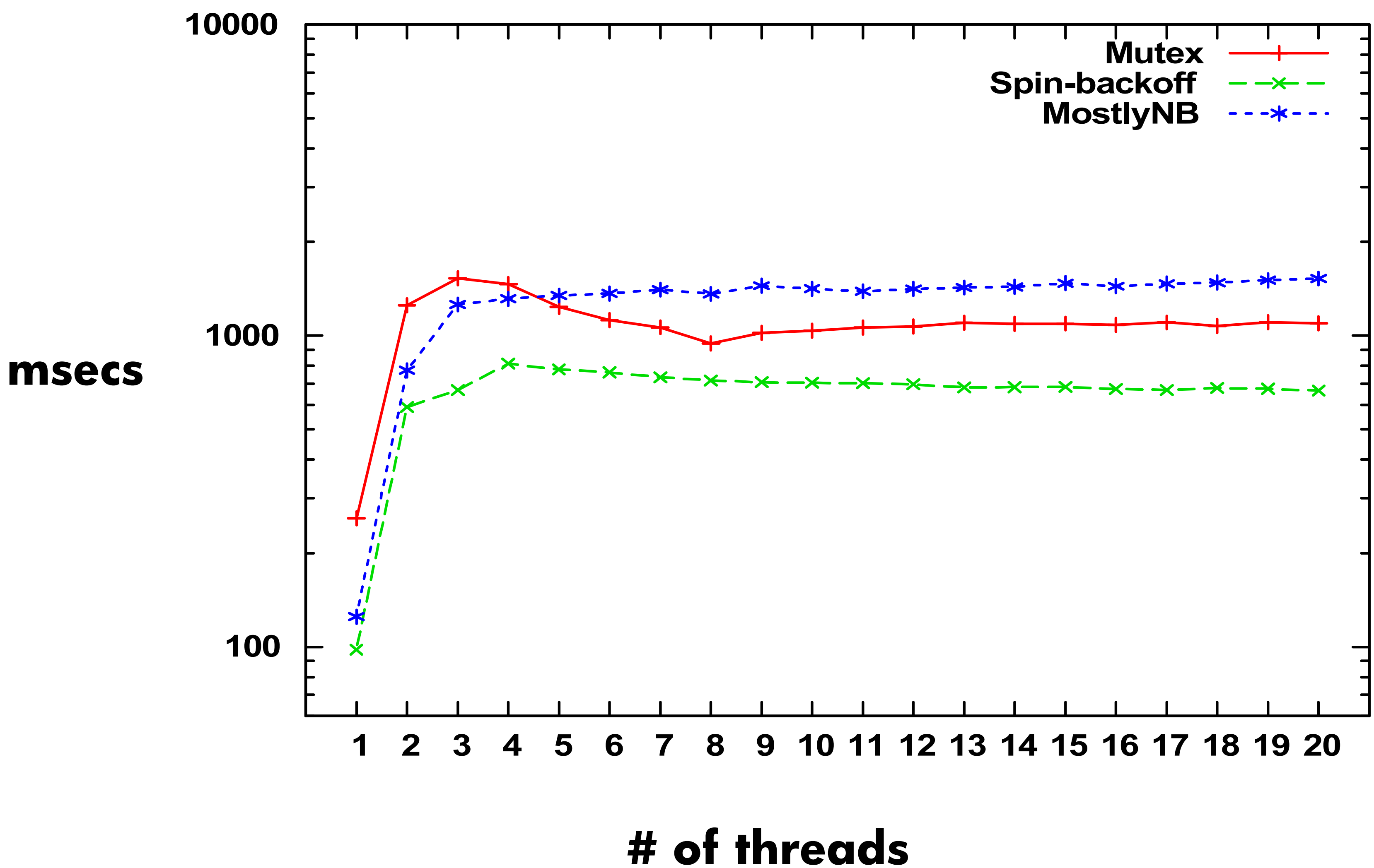
Benchmark execution time (4xPPro/200, RedHat 9, NPTL)



Benchmark execution time (2xP4 Xeon 2GHz, RedHat 7.2)



Benchmark execution time (4x Itanium 2, Debian Linux, NPTL)



Conclusions

- Performance is competitive with other good synchronization techniques
  - And far better than some.
- Wide CAS is better, but sometimes unavailable.
- Performance of almost non-blocking algorithm is close.
- Many applications can be written for almost non-blocking algorithm, and can thus use either.

Open issues

- Are almost non-blocking algorithms useful for fault-tolerance?
  - Good enough for recoverable faults …
- Other data structures?
  - This is really an ABA solution.
  - Construct general LL/SC variables analogously to Jayanti and Petrovic (PODC 2003) or previous talk?