RWLocks in Erlang/OTP

Rickard Green

rickard@erlang.org

What is an RWLock?

- Read/Write Lock
  - Write locked
    - Exclusive access for one thread
  - Read locked
    - Multiple reader threads
    - No writer threads

RWLocks Used in ERTS

- ETS tables
- Internal tables
  - Atom table
  - Registered names
  - Distribution tables
  - …
- …

What Made us Look at RWLocks?

- Testcase failed
  - Pthread rwlocks on Linux with reader preferred strategy caused starvation of writers
  - Solved by using our writer preferred fallback implementation instead
- Customers complained about poor performance of the fallback implementation
  - Solved by letting them enable the reader preferred pthread rwlocks implementation
- That is, something needed to be done... ☹

NPTL Pthread RWLocks

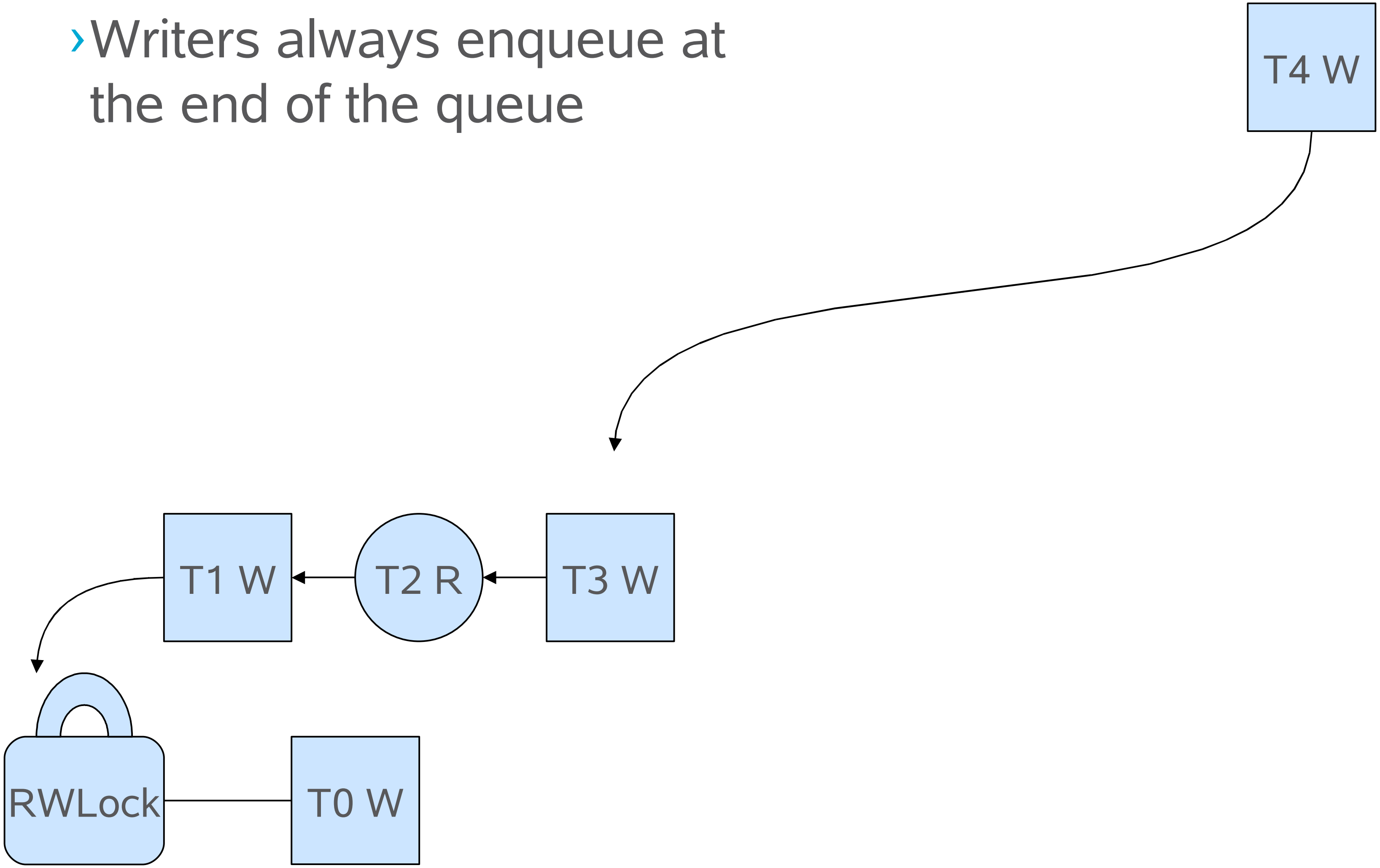
- Why look at NPTL RWLocks?
  - NPTL (Native POSIX Thread Library) is the thread library used on modern Linux distros
  - Linux is our most important platform
    - The vast majority of our customers run on Linux
    - A lot of the other (perhaps most of the other?) users run on Linux
- Strategy used during contention
  - Defaults to reader preferred
    - As long as the lock is read locked or readers are waiting for the lock, writers have to wait
  - Can be configured as writer preferred
    - As long as the lock is write locked or writers are waiting for the lock, readers have to wait
  - Both strategies suffer from starvation issues
    - Writer preferred is, however, not as problematic as reader preferred

ERTS RWLocks

- Doesn't use a writer or reader preferred strategy
- Interleaves readers and writers during contention
- Fair against readers as well as writers

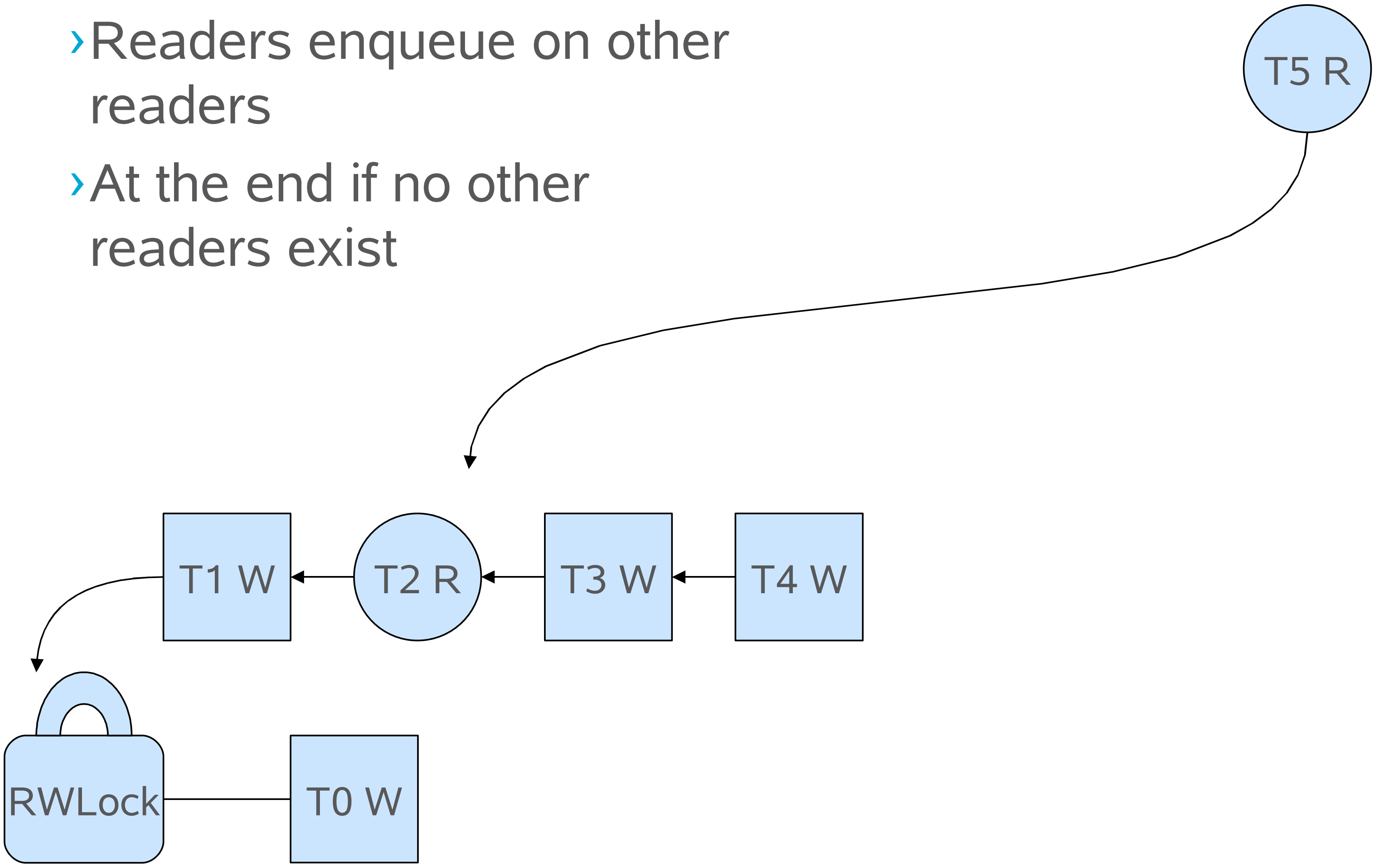
ERTS RWLocks - Enqueue Writer

- Writers always enqueue at the end of the queue



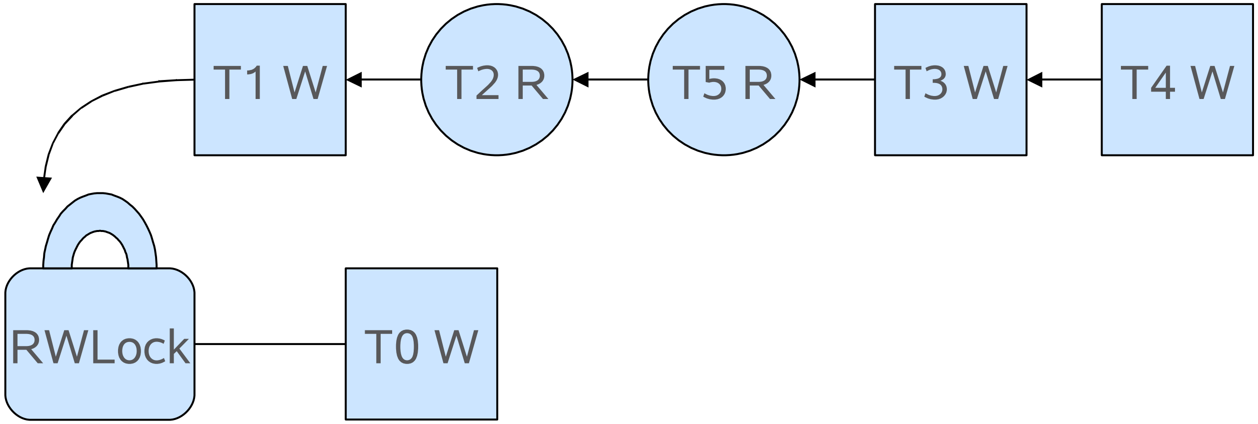
ERTS RWLocks - Enqueue Reader

- Readers enqueue on other readers
- At the end if no other readers exist



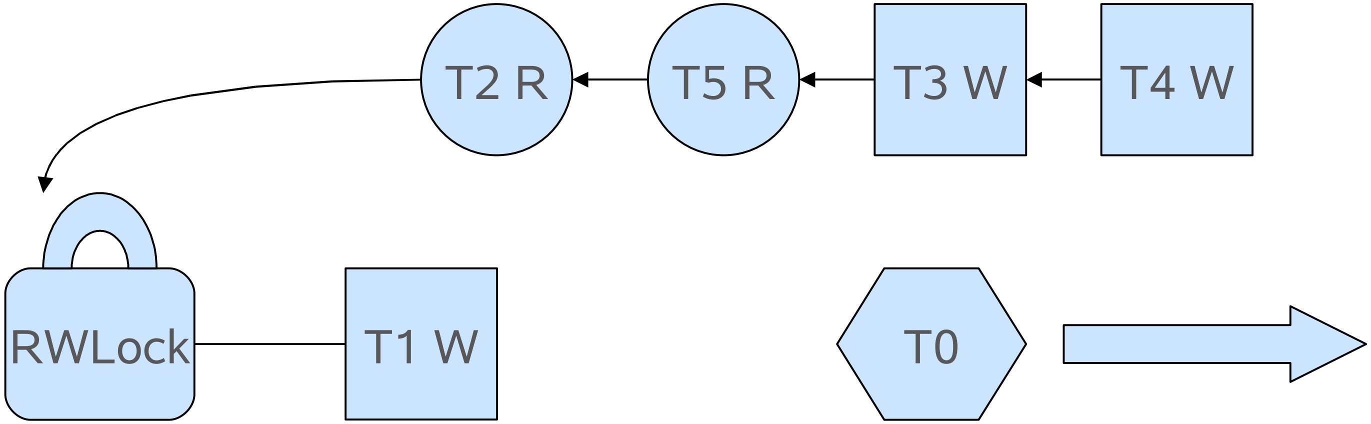
ERTS RWLocks - Queue

- All readers will accumulate at one place in the queue



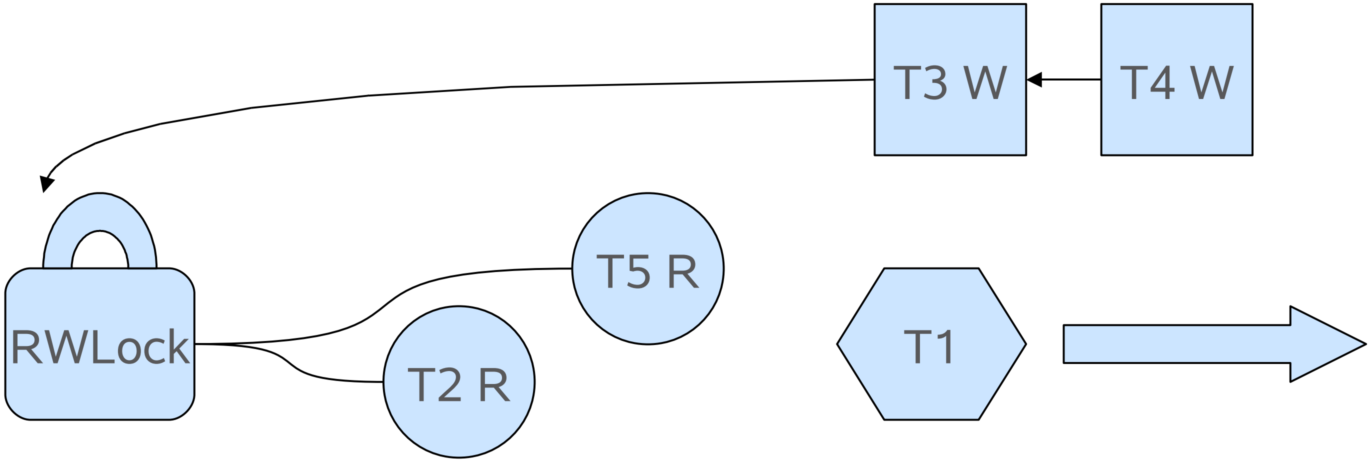
ERTS RWLocks – Dequeue Writer

- Writer at the head of the queue takes over the lock



ERTS RWLocks – Dequeue Readers

- All readers at the head of the queue take over the lock



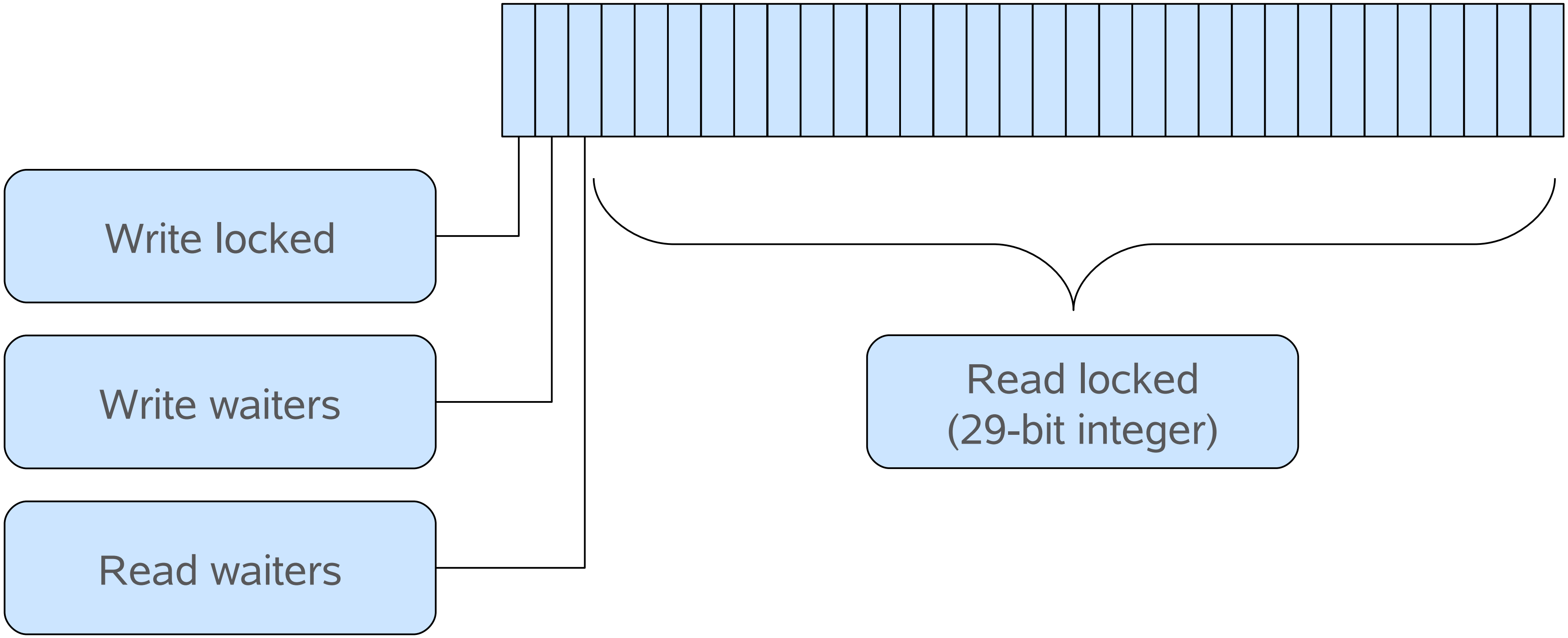
ERTS RWLocks - Queue

- Ensures that all waiting threads eventually will get the lock
- Able to execute readers as much as possible in parallel
- Writers wont be punished too much by readers going past them
  - Writers wont be punished at all in the case where there are as many threads on the system as cores and each thread read locks for the same amount of time

ERTS RWLocks - Default

- Data structure
  - Integer flag field
  - Queue (double linked list of waiting threads)
  - Queue lock
- Uncontended case
  - Atomic operations on the integer flag field
- Contended case
  - Atomic operations on the integer flag field
  - Locked operations on the queue

ERTS RWLocks – Flag Field (default)



Write locked

Write waiters

Read waiters

Read locked (29-bit integer)

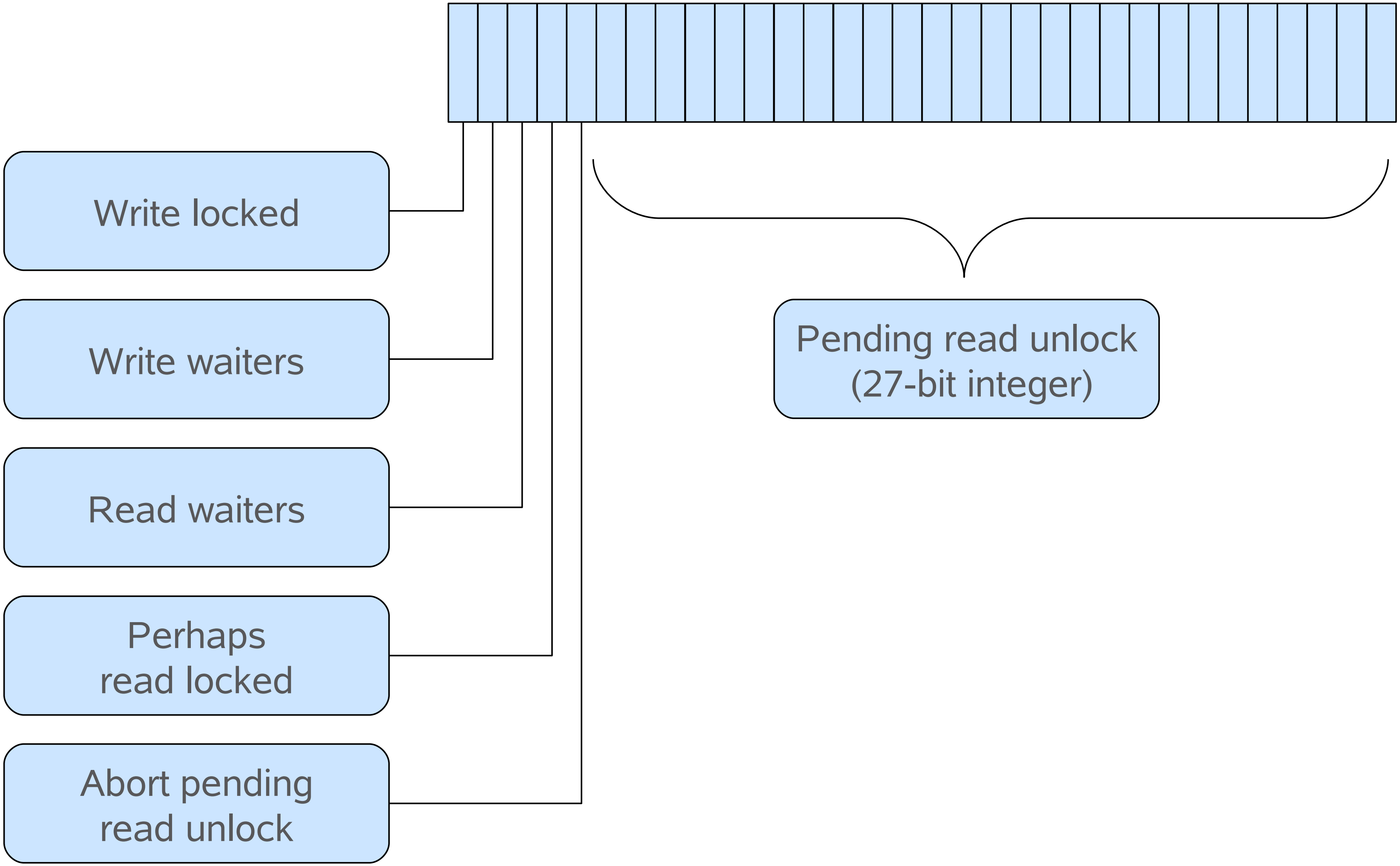
ERTS RWLocks – Performance

- An improvement compared to NPTL RWLocks, but uncontended read lock case is still a bit disappointing
  - Conceptually only reads of memory, however...
  - Writes to the RWLock cache line are ping-ponged between processors
  - In ETS also other stuff are ping-ponged
    - Meta table lock cache-line
    - Table reference counter cache-line
- Would be nice to avoid this cache-line ping-ponging
  - Modified reader optimized RWLock implementation using reader groups
  - ETS modifications:
    - Rewrite of the meta table locking to use the new reader optimized rwlocks
    - No use of table reference counter when read locking

ERTS RWLocks – Reader Optimized

- Data structure
  - Integer flag field (modified)
  - Queue (double linked list of waiting threads)
  - Queue lock
  - Reader groups (integer counters in separate cache-lines)
- Uncontended cases
  - Write lock/unlock
    - Atomic operations on the integer flag field (not completely true)
  - Read lock/unlock
    - Atomic operations in the readers groups (mostly)
- Contended case
  - Atomic operations on the integer flag field
  - Atomic operations in the reader groups
  - Locked operations on the queue

ERTS RWLocks – Flag Field (reader optimized)



Write locked

Write waiters

Read waiters

Perhaps read locked

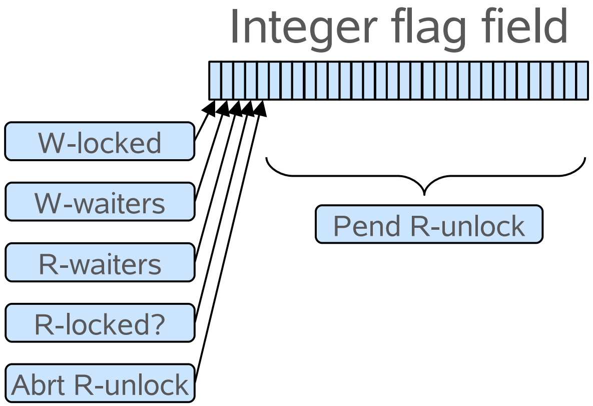
Abort pending read unlock

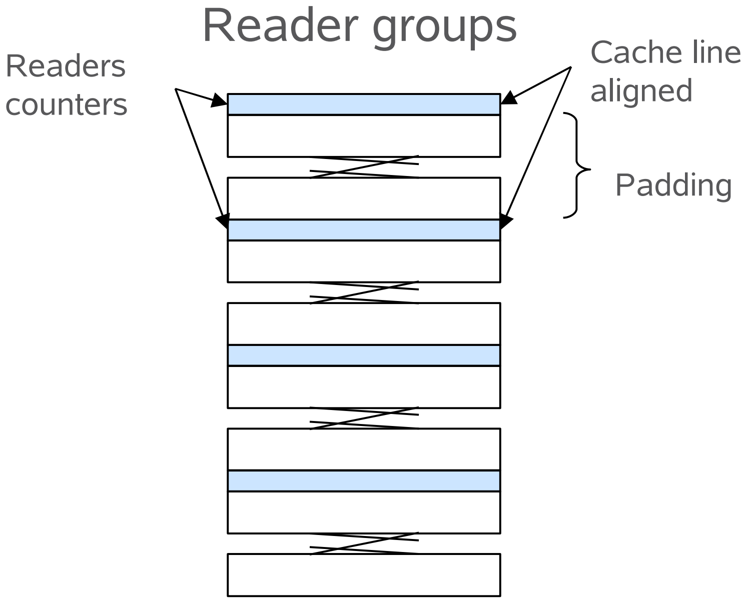
Pending read unlock (27-bit integer)

ERTS RWLocks – Reader Optimized

- Uncontended read-lock
  - Increment reader group counter
  - Read flag field.
    - Verify no “W-locked”, “W-waiter”, nor “Pend R-unlock”
    - Set “R-locked?” if not already set
- Uncontended read-unlock
  - Decrement reader group counter
  - If reader group counter reached zero, read flag field
    - Verify no “{W,R}-waiters” nor “Pend R-unlock”

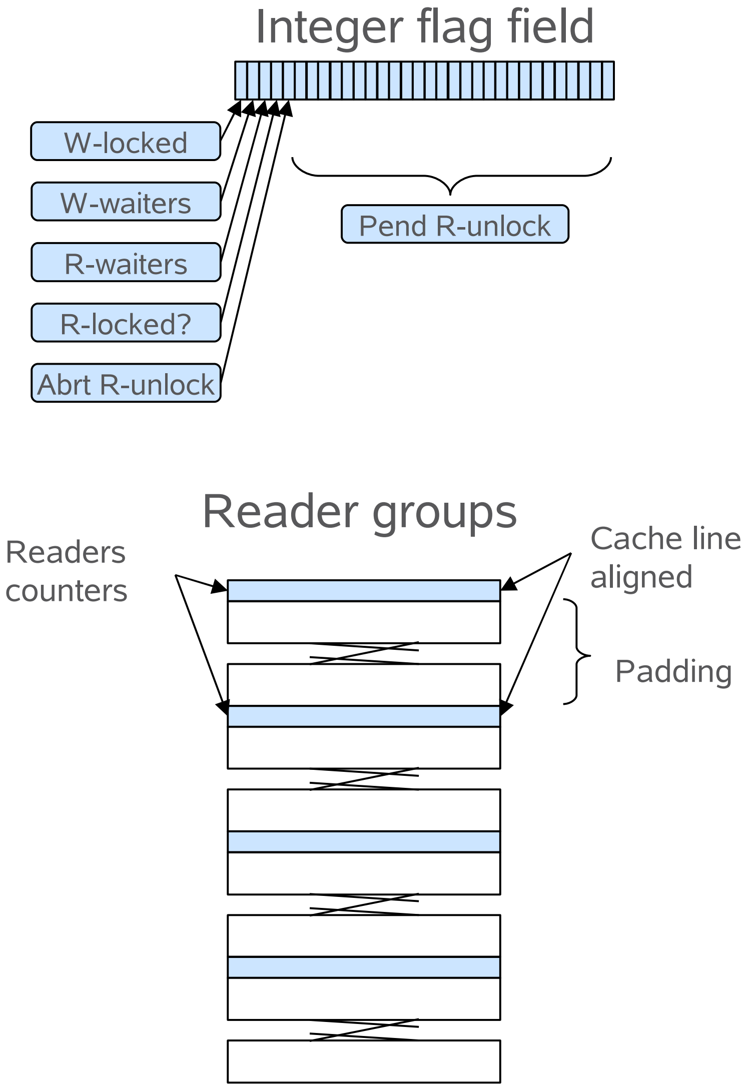
Note! We do *not* reset “R-locked?”





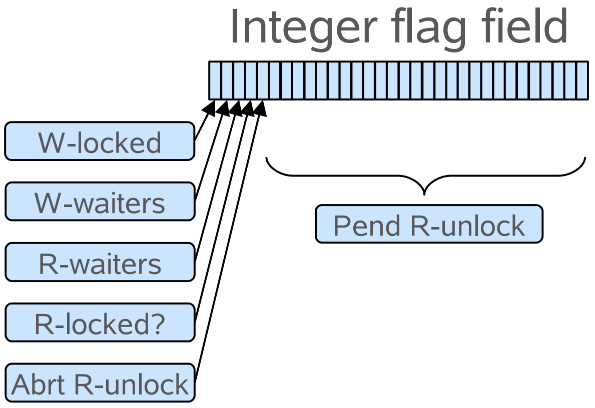
ERTS RWLocks – Reader Optimized

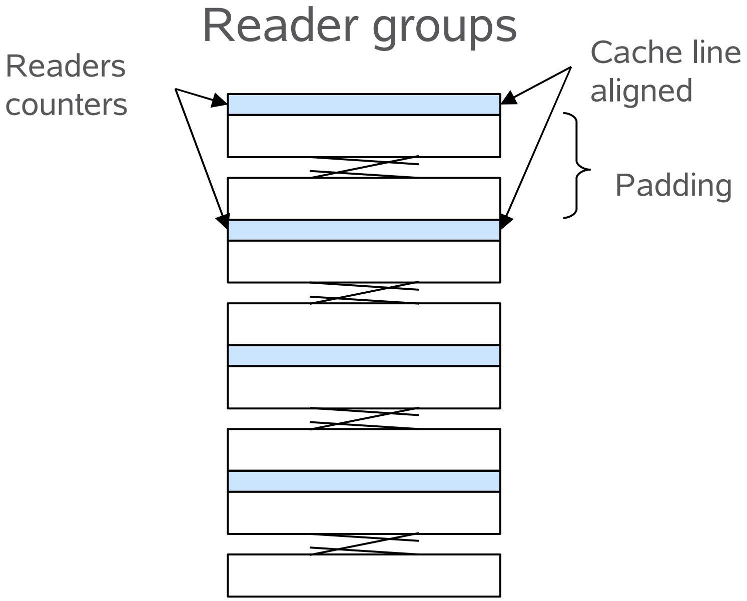
- Uncontended write-lock
  - Read integer flag field.
  - Verify no flags or “R-locked?” set
  - If no flags are set, set “W-locked”
  - If “R-locked?”, check reader groups:
    - Increment “Pend R-unlock”
    - Verify that all groups are zero
    - Reset “R-locked?”, decrement “Pend R-unlock”, set “W-locked”
- Uncontended write-unlock
  - Reset “W-locked”
  - Verify no “{W,R}-waiters”



ERTS RWLocks – Reader Optimized

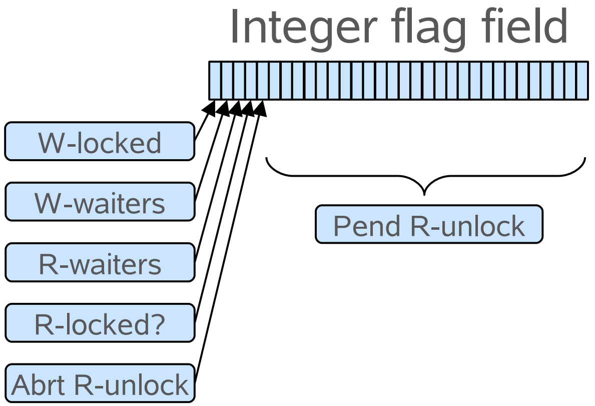
- Contended cases
  - When a lock operation fail, the thread continues spinning
    - trying to lock actively (a few times); then
    - enqueue and spin passively (on another structure); then
    - block
  - When equeueing the “{W,R}-waiters” flag is set while holding the queue lock; then one last effort to acquire the lock is made
  - After this the thread depends on another thread transferring the lock to it and waking it up

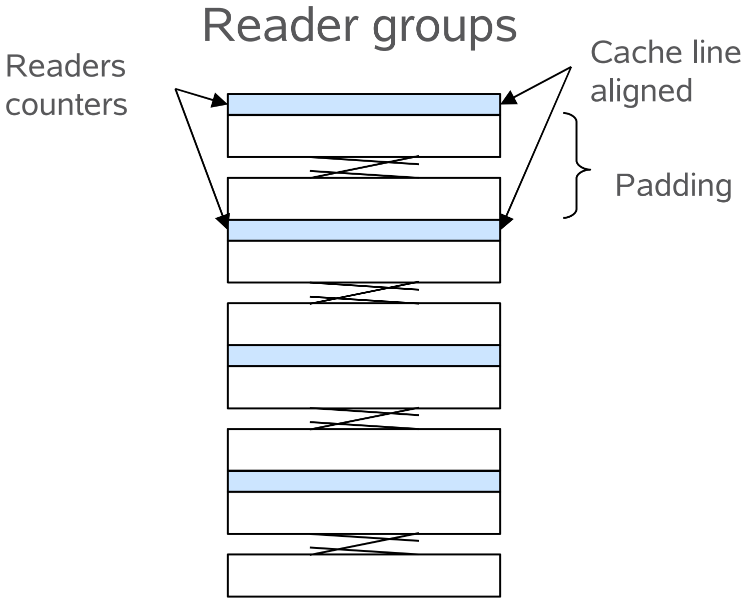




ERTS RWLocks – Reader Optimized

- Contended cases (continued)
  - Write locking when “R-locked?” is set is the complicated case since it can be interrupted by modifications in reader groups
  - A read locking thread aborts
    - pending read unlock operations if no waiters exist
    - its own operation if waiters exist, and then helps waiters out
  - A read unlocking thread modifying a reader group continues by helping pending read unlock threads out
  - Blocking write operations are guaranteed to eventually get the lock, since they eventually will end up in queue if repeatedly interrupted





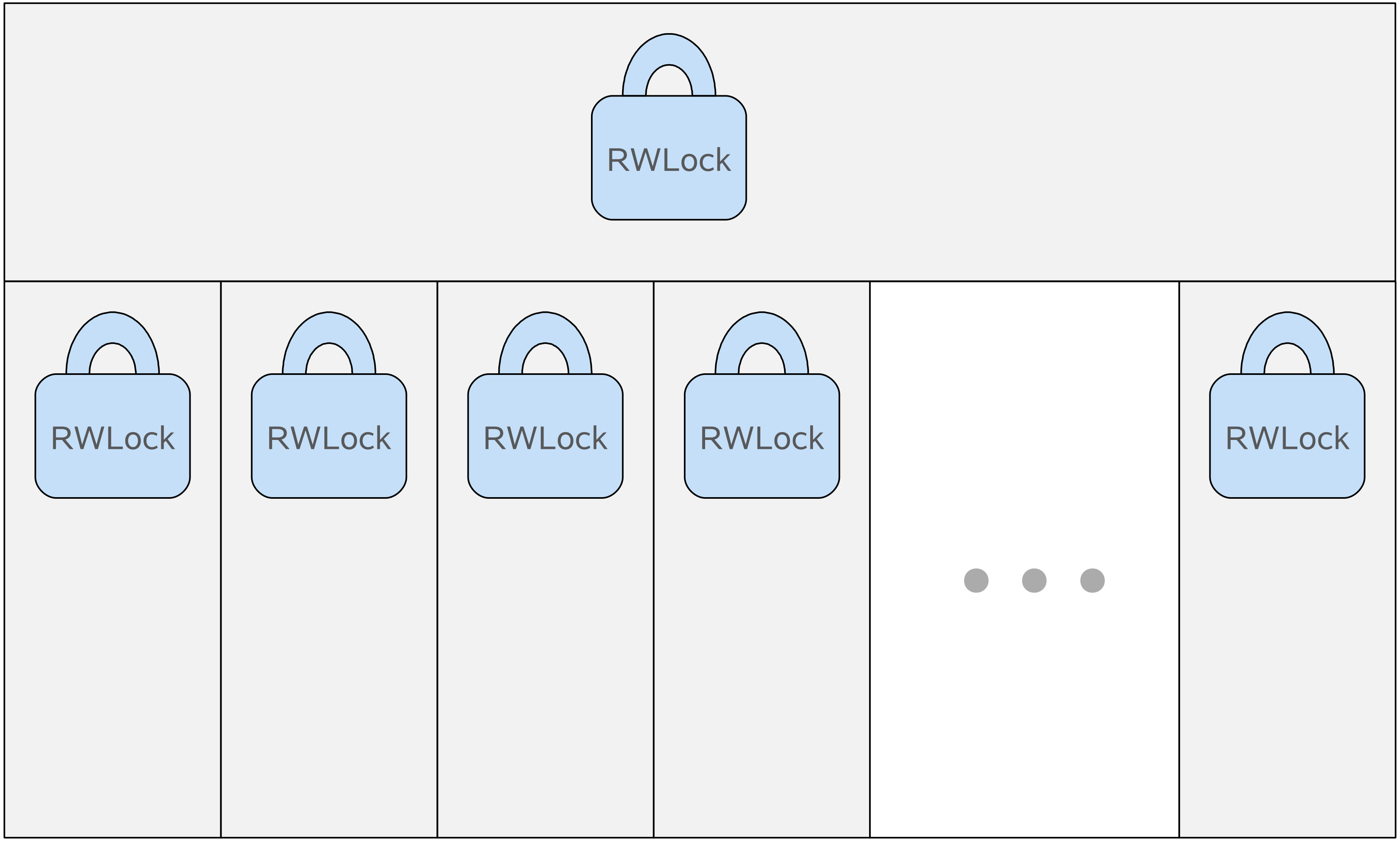
Benchmarking

- The benchmark used and OTP-8925 can be found at `www.erlang.org/~rickard/euc-2010`
- An ETS benchmark where 1000 processes concurrently access a common public ets table of type set
- Accesses consists of `ets:lookup()` (read) and `ets:insert()` (write) in different mixes
- Run with the `thread_spread` scheduler bind type
- R14B NPTL-rwlocks and R14B ERTS-rwlocks only differs in rwlock implementation used
  - R14B NPTL-rwlocks: ./configure force_pthread_rwlocks=yes
- When benchmarking with NPTL-rwlocks
  - No read_concurrency run has been made - same as default
  - No combined read_concurrency and write_concurrency has been made - same as write_concurrency

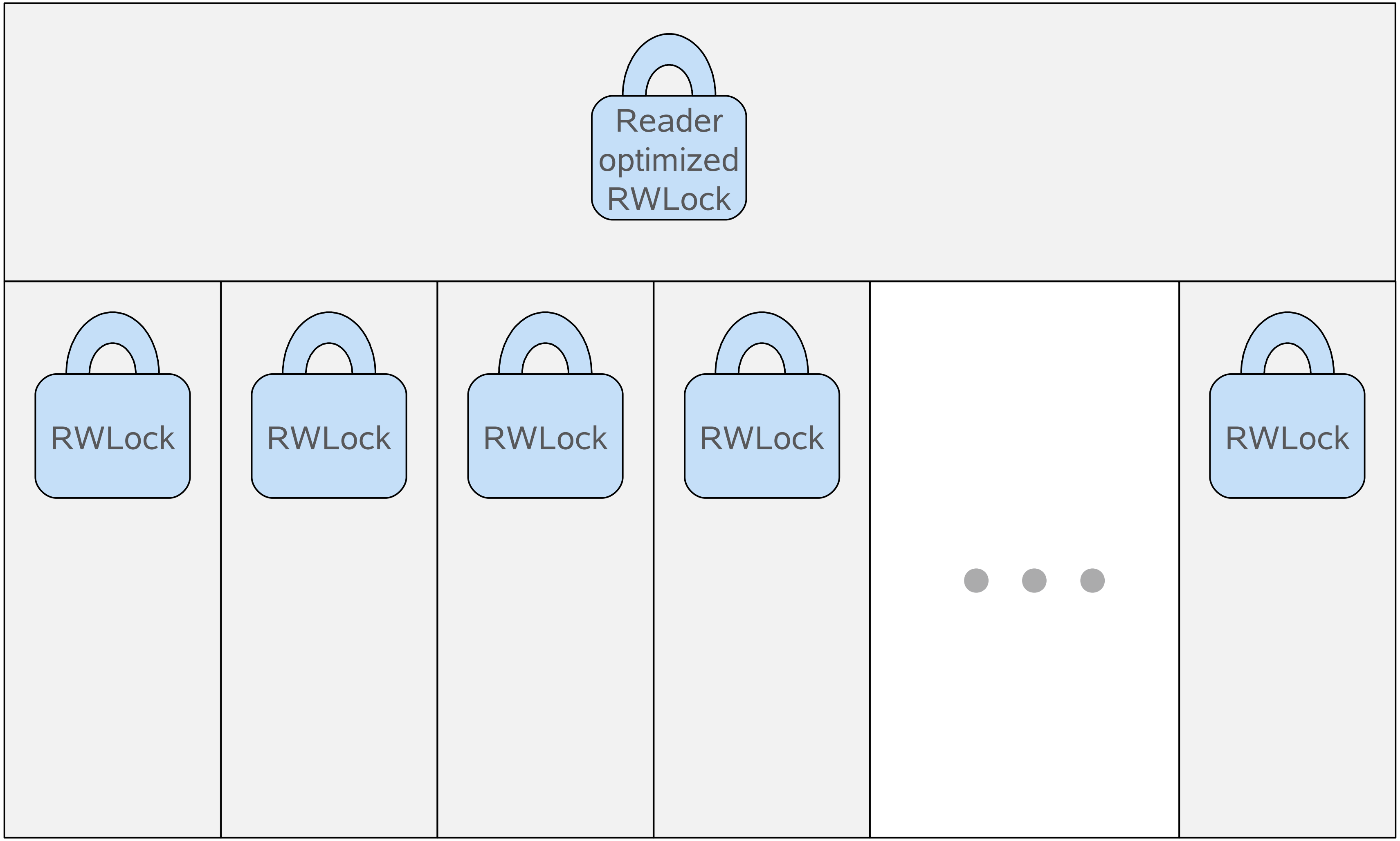
ERTS RWLocks – ETS Options

- The default
  - One table global normal (non-reader optimized) rwlock
- read_concurrency
  - One table global reader optimized rwlock
- write_concurrency
  - One table global reader optimized rwlock (normally read locked), and multiple normal rwlocks protecting different parts of the table
- read_concurrency and write_concurrency combined
  - One table global reader optimized rwlock, and multiple reader optimized rwlocks protecting different parts of the table

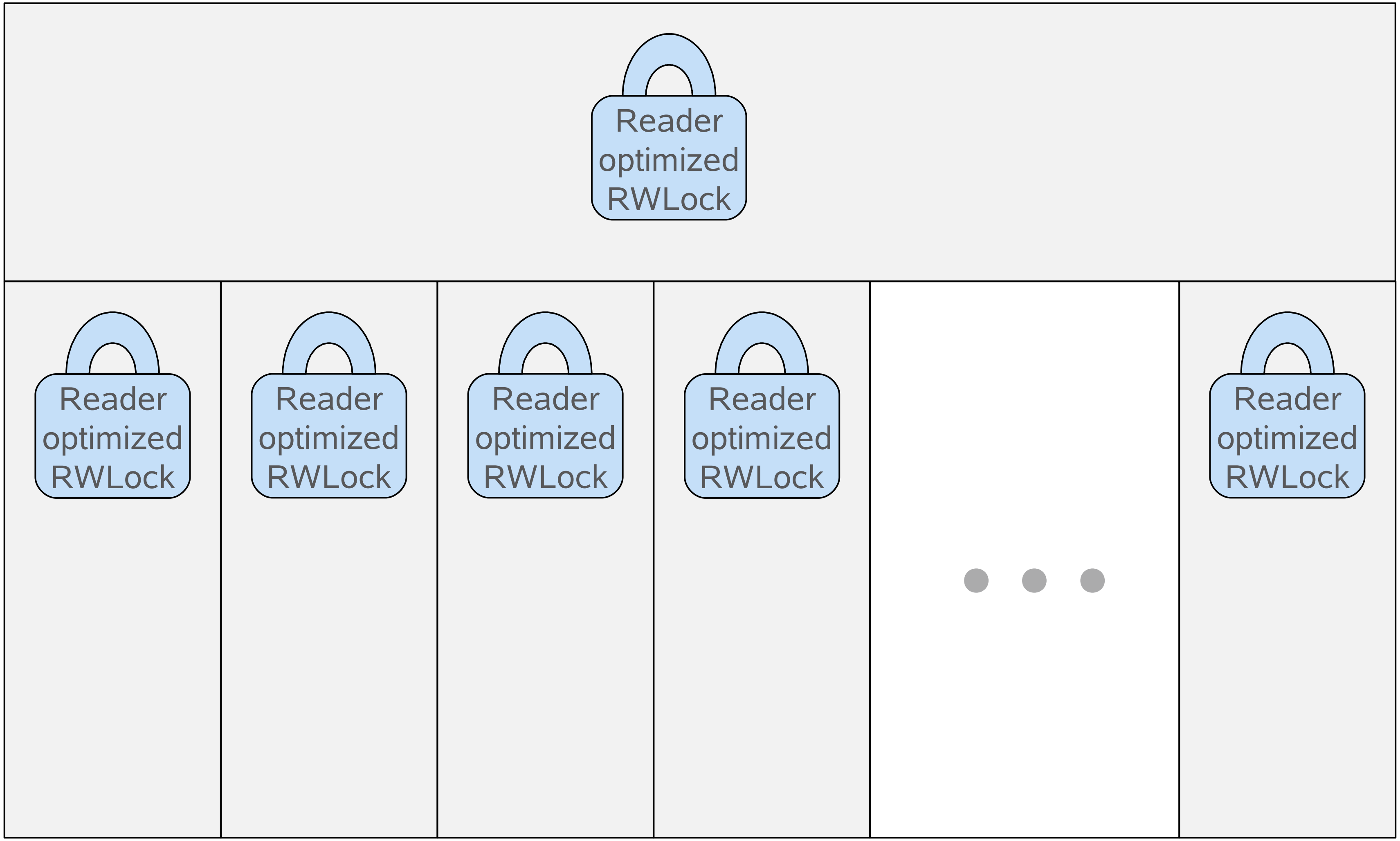
ETS Table with `write_concurrency` Option (before R14B)



ETS Table with `write_concurrency` Option (R14B)



ETS Table with `write_concurrency` and `read_concurrency` Options (R14B)



Benchmarking

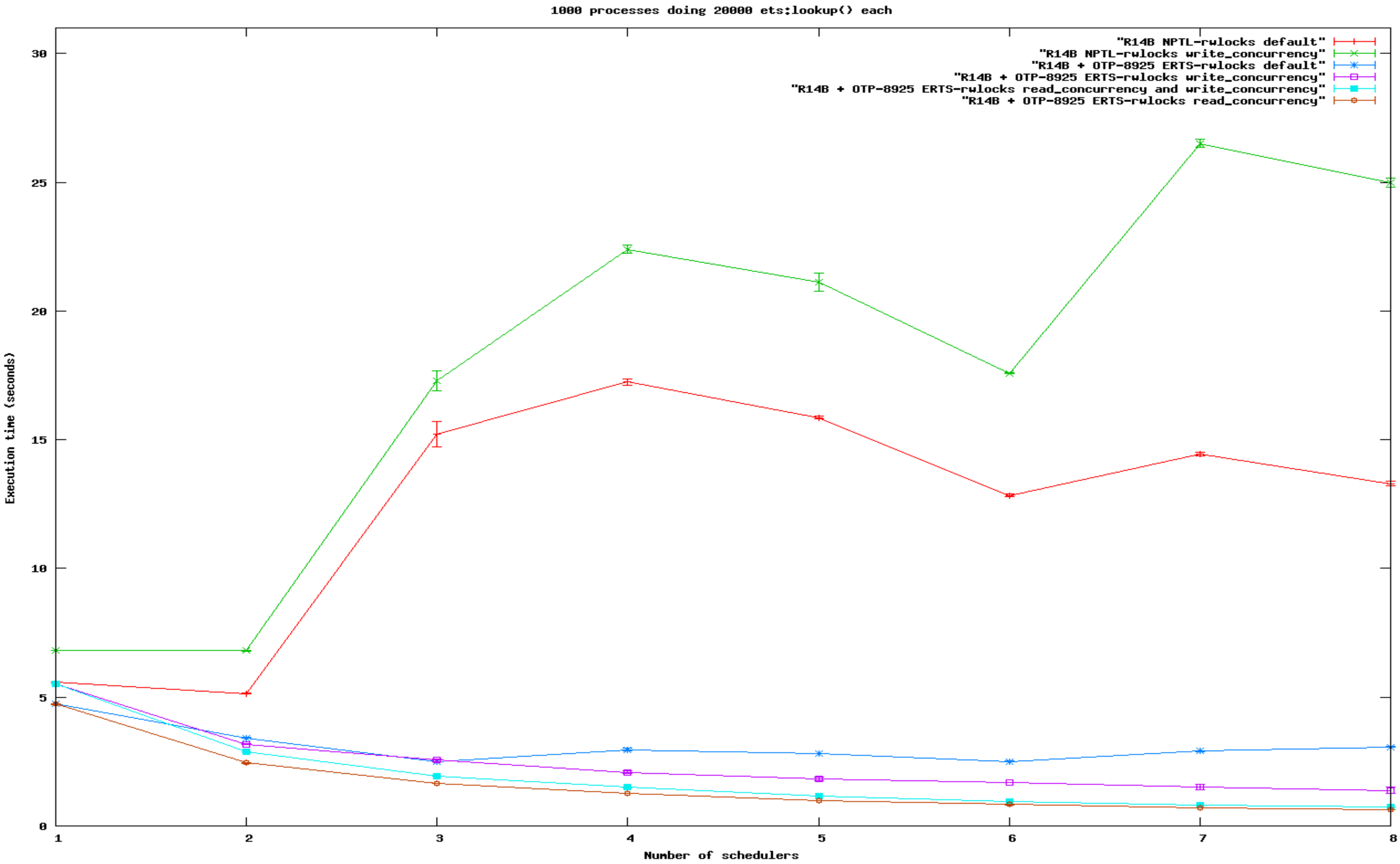
- Machine A
  - SLES 10.2
  - Kernel 2.6.16.60-0.39.3-smp
  - NPTL 2.4
  - x86_64
  - 2x Intel Xeon L5430 @ 2.66 GHz

```
1> erlang:system_info(cpu_topology).
[{processor,[{core,{logical,0}},
             {core,{logical,4}},
             {core,{logical,2}},
             {core,{logical,6}}]},
 {processor,[{core,{logical,1}},
             {core,{logical,5}},
             {core,{logical,3}},
             {core,{logical,7}}]}]
```

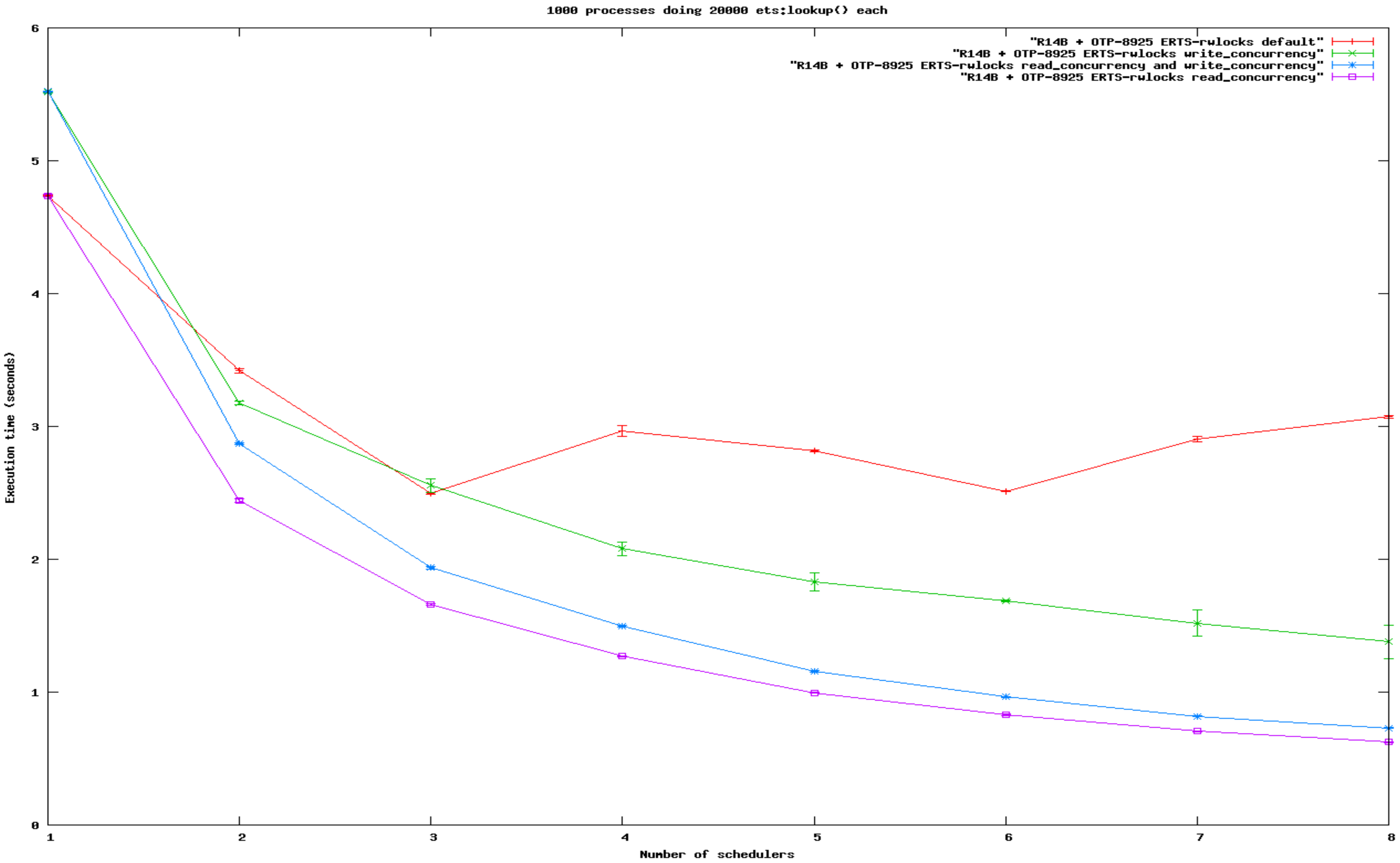
- Machine B
  - Ubuntu 9.10
  - Kernel 2.6.31-22-server
  - NPTL 2.10.1
  - x86_64
  - 2x Intel Xeon <unknown id> @ 2.8 GHz

```
1> erlang:system_info(cpu_topology).
[{processor,[{thread,{logical,0}},
             {thread,{logical,2}}]},
 {processor,[{thread,{logical,1}},
             {thread,{logical,3}}]}]
```

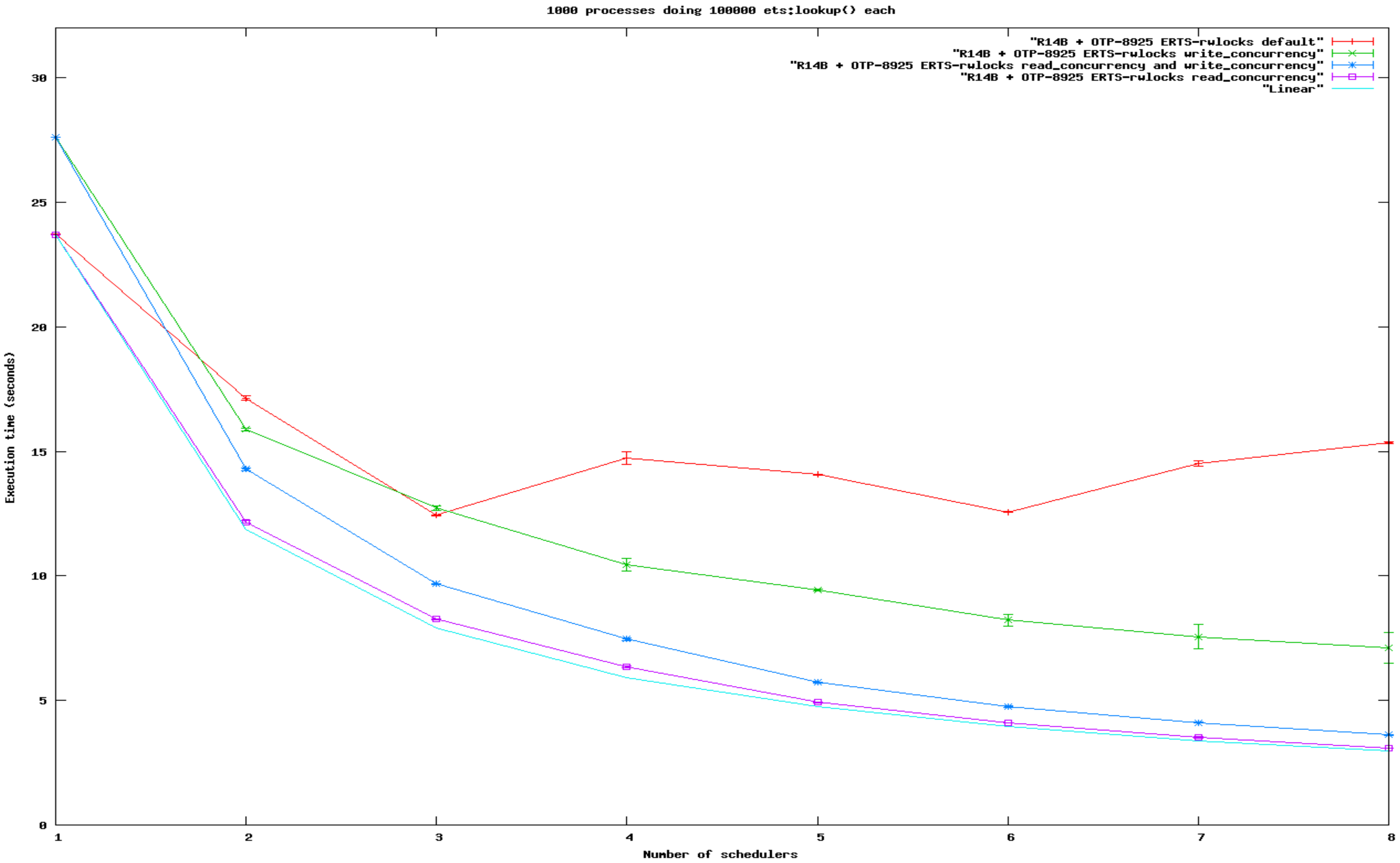
Benchmark Results – Machine A



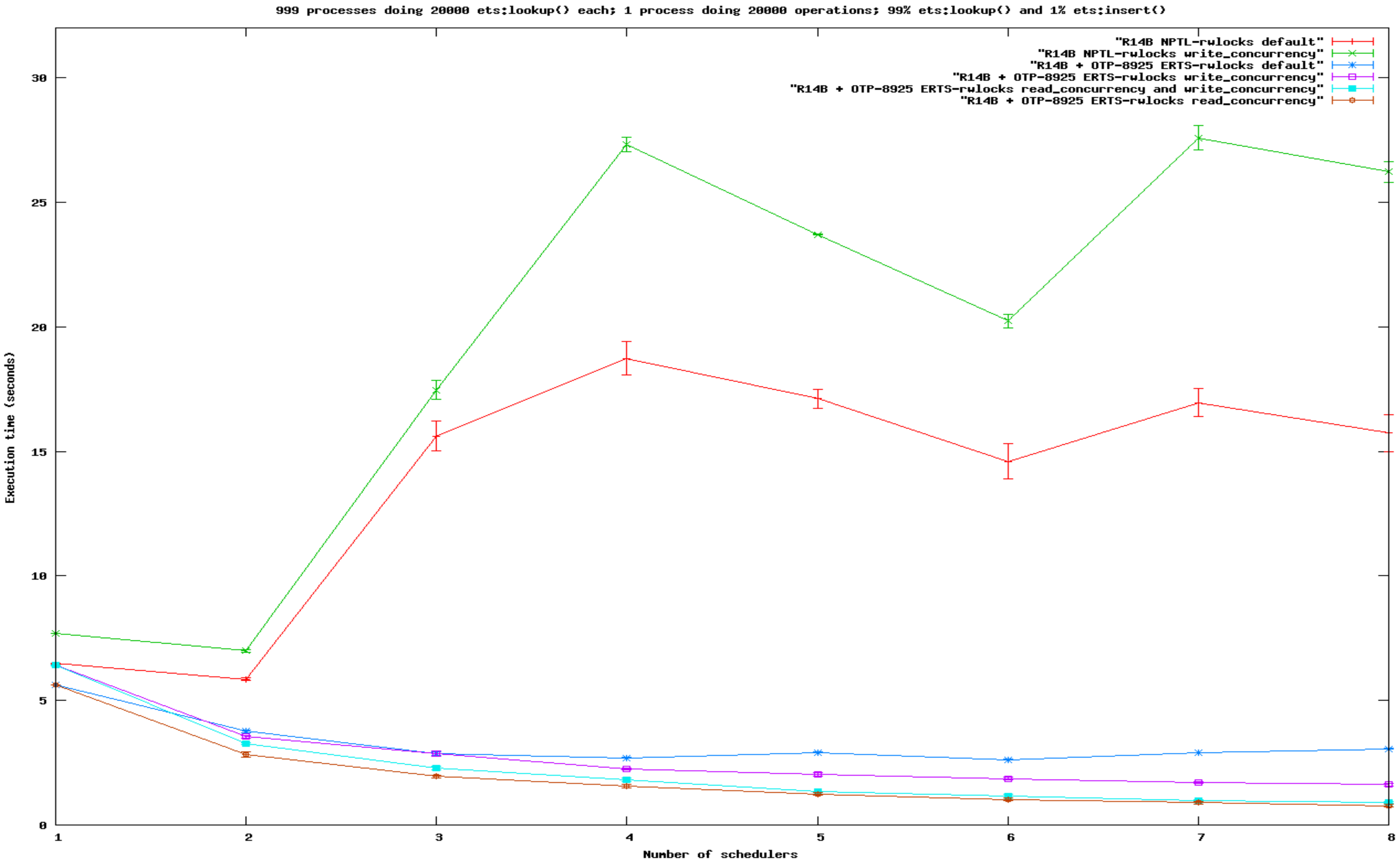
Benchmark Results – Machine A



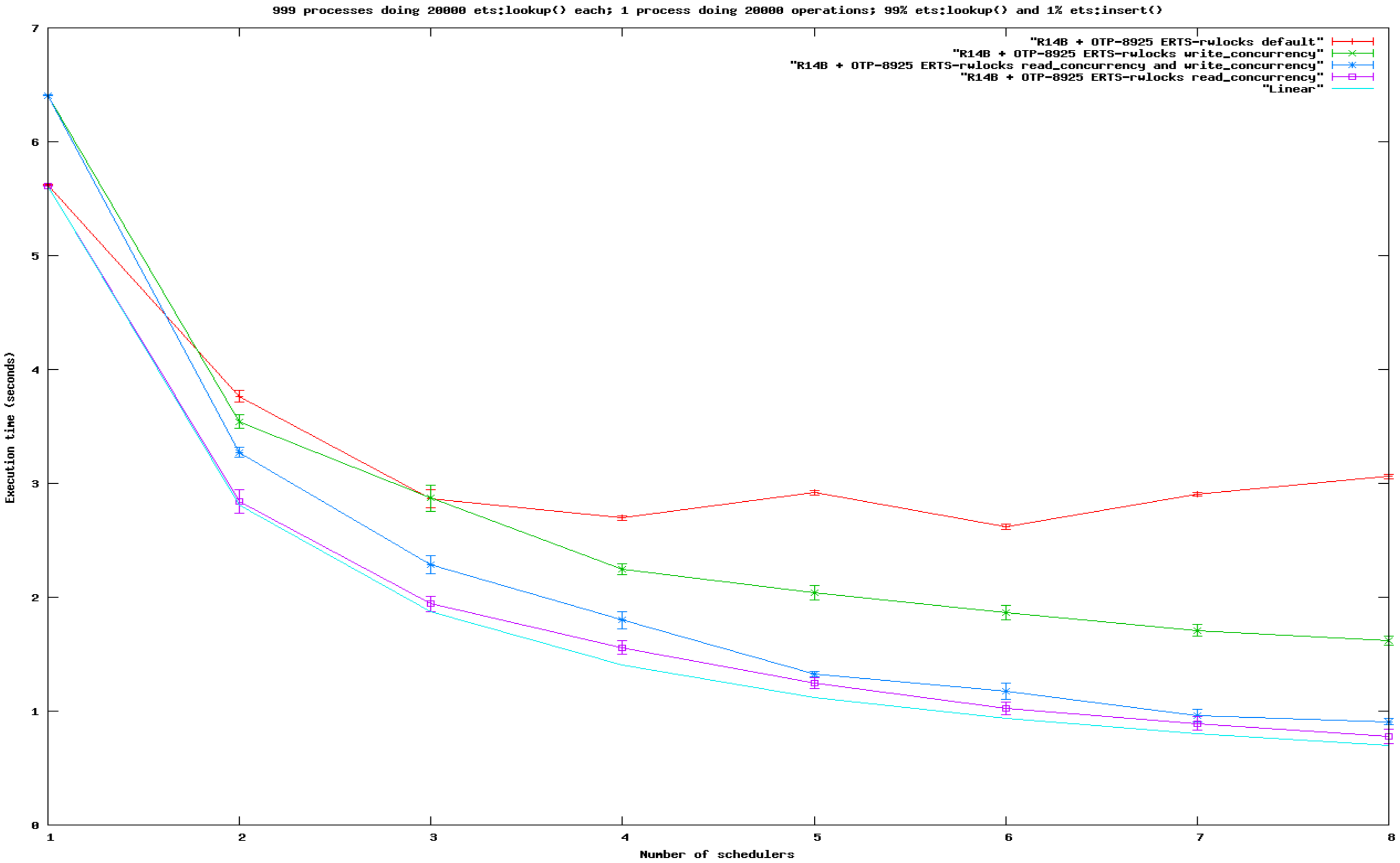
Benchmark Results – Machine A



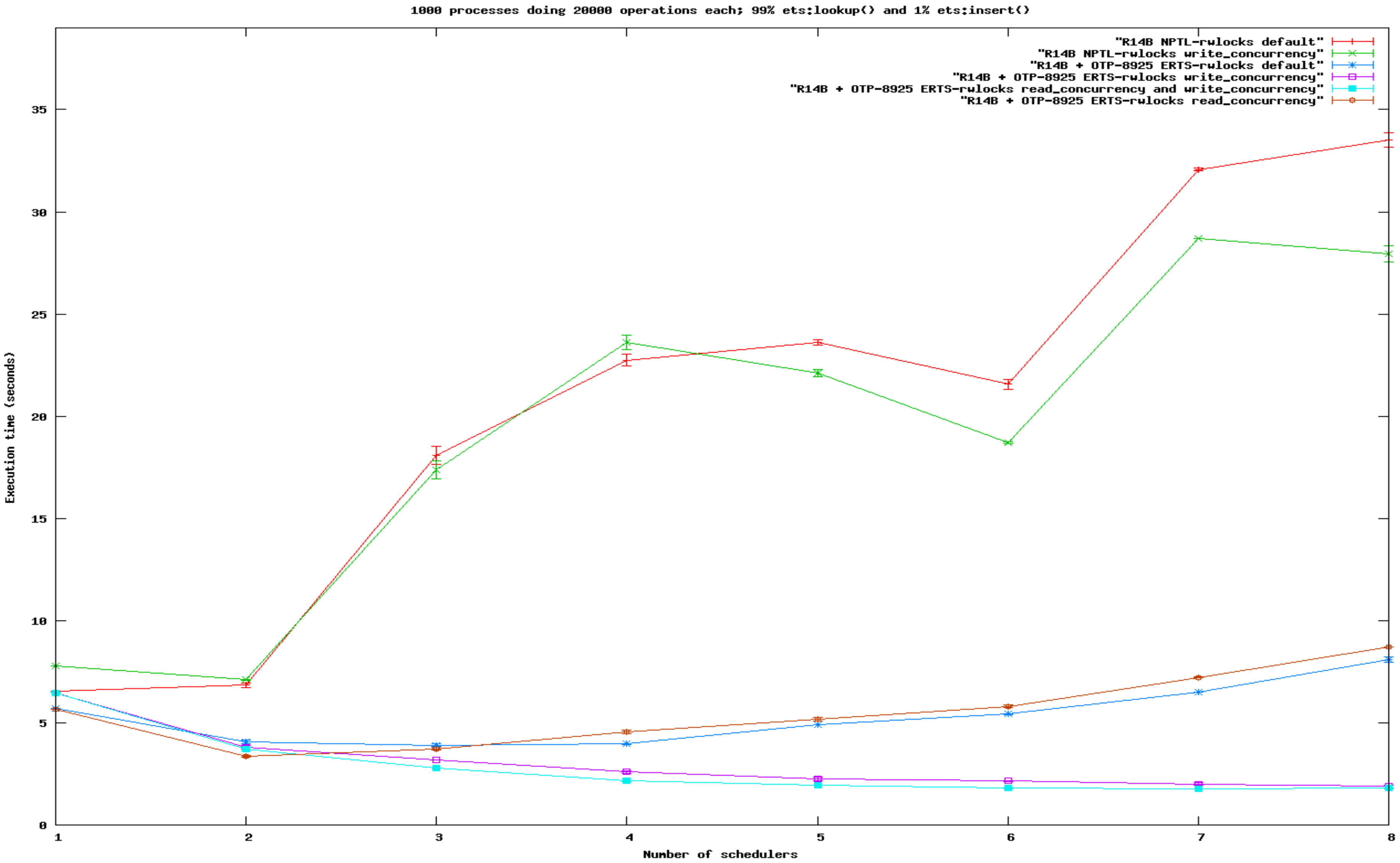
Benchmark Results – Machine A



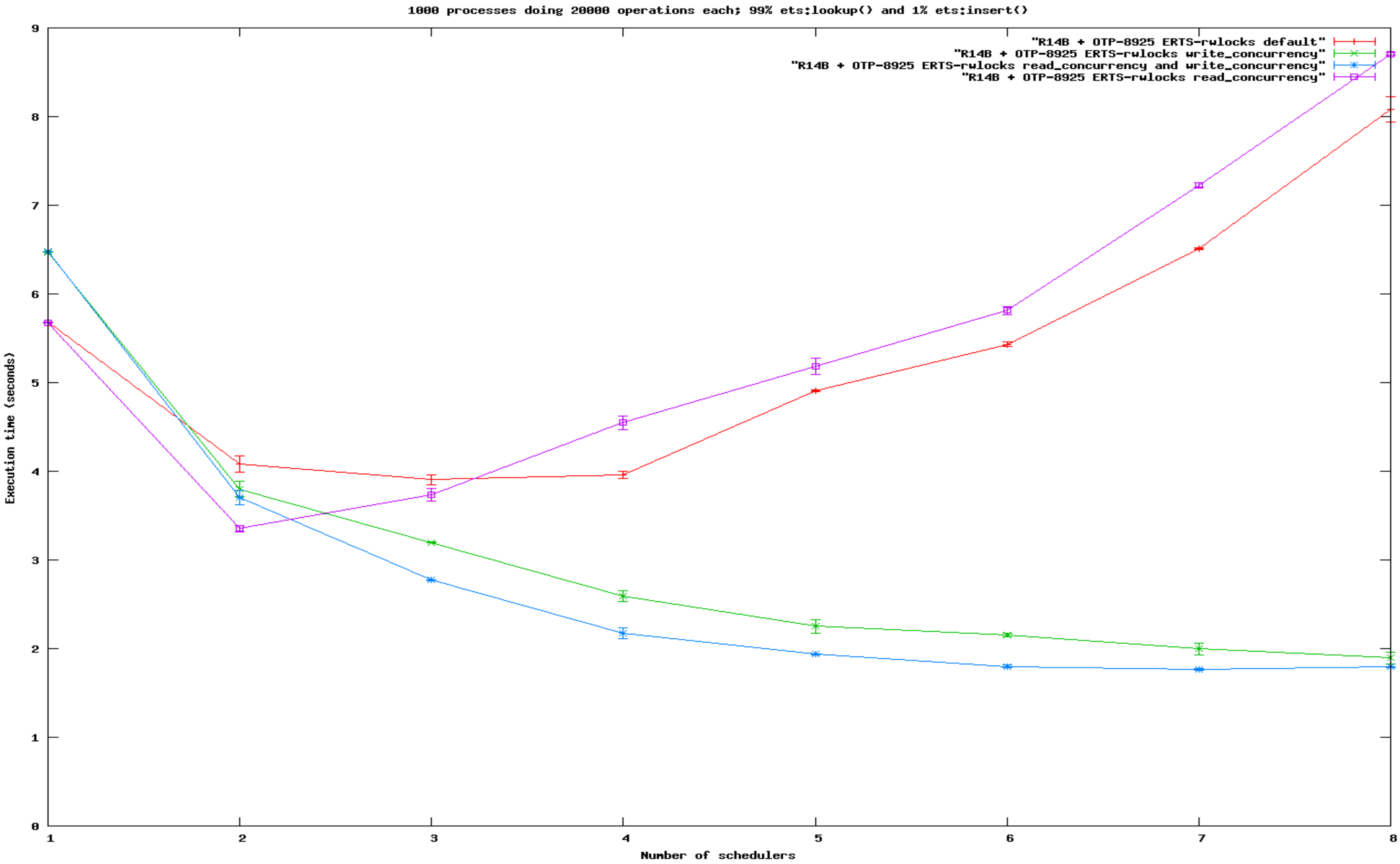
Benchmark Results – Machine A



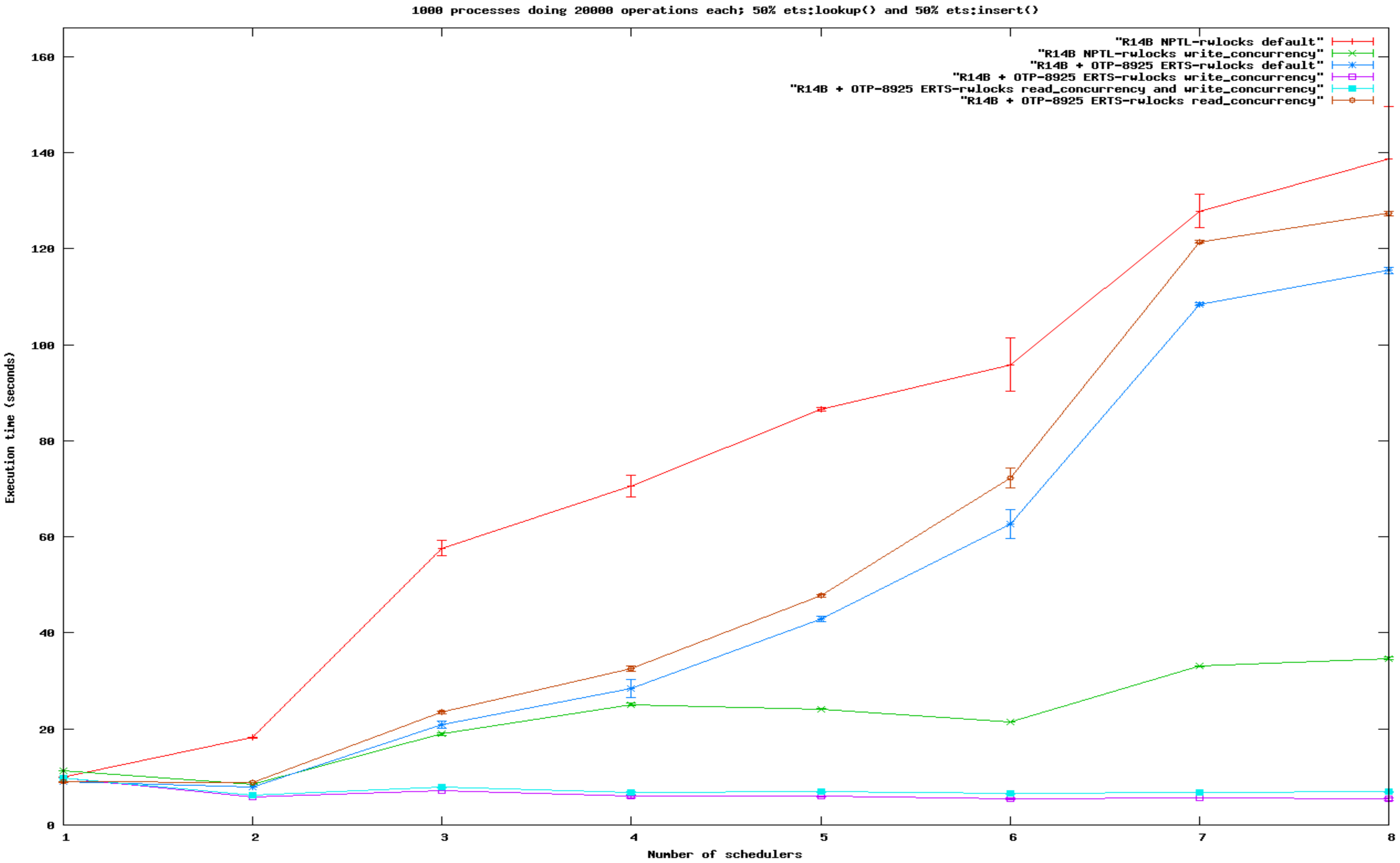
Benchmark Results – Machine A



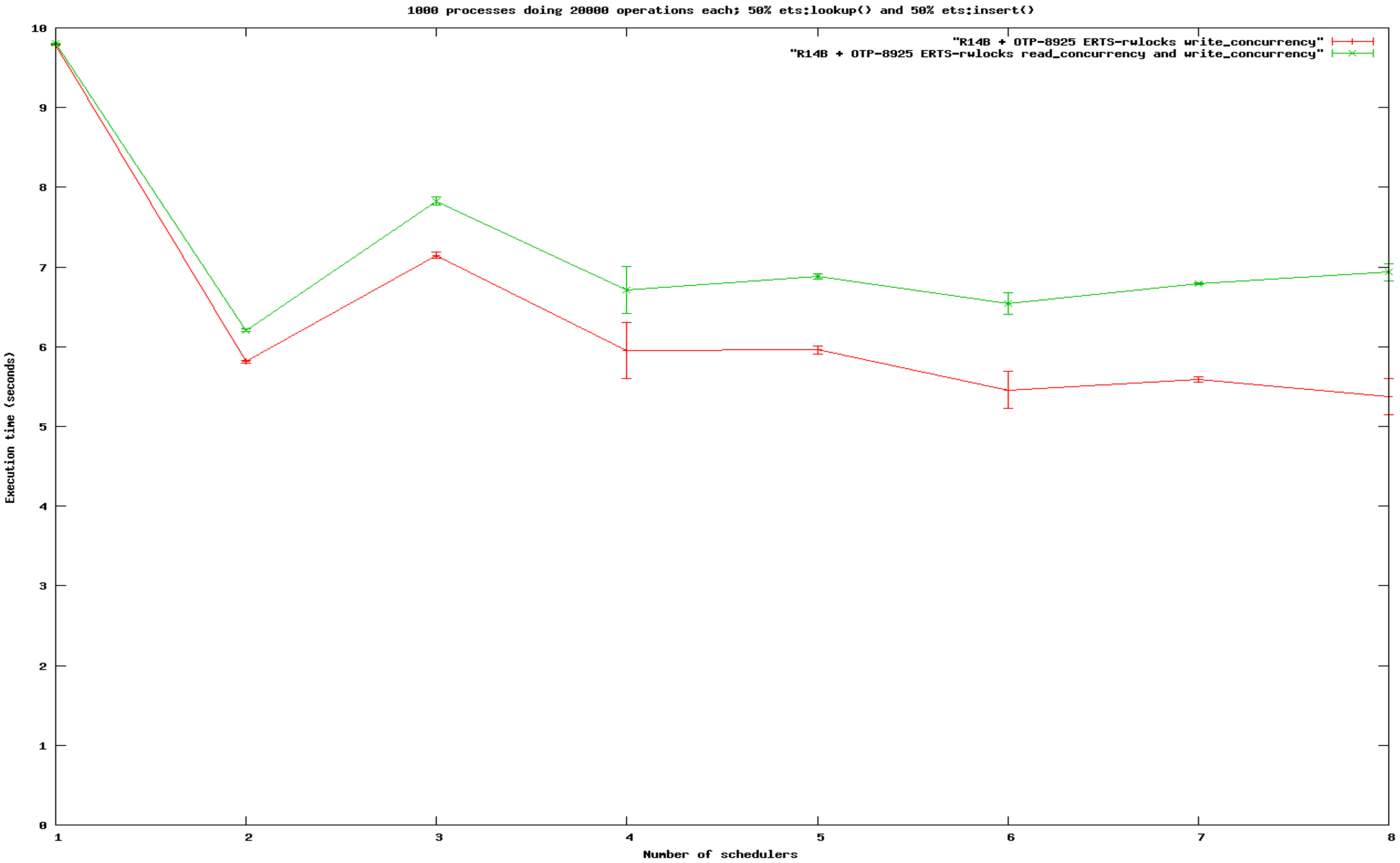
Benchmark Results – Machine A



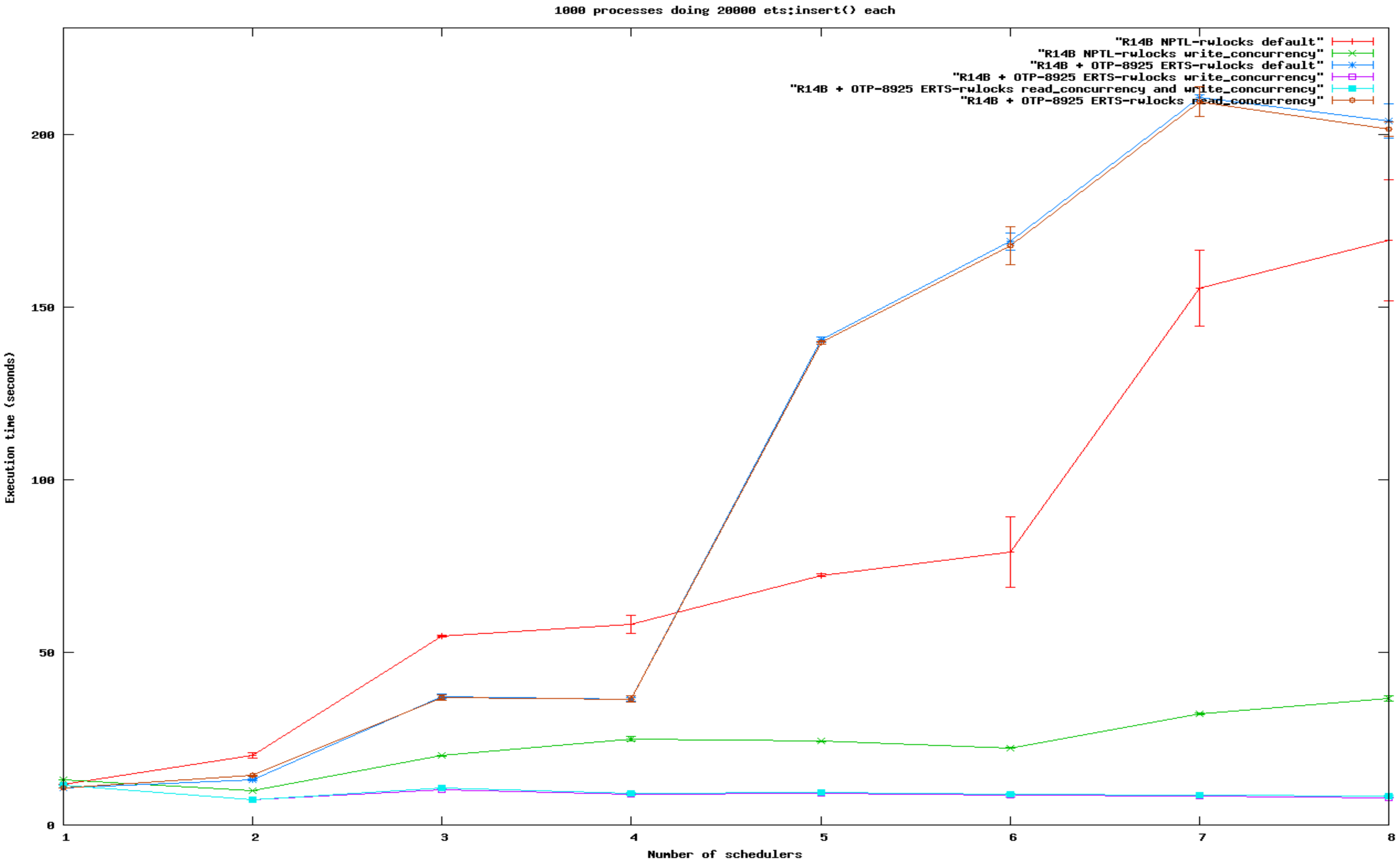
Benchmark Results – Machine A



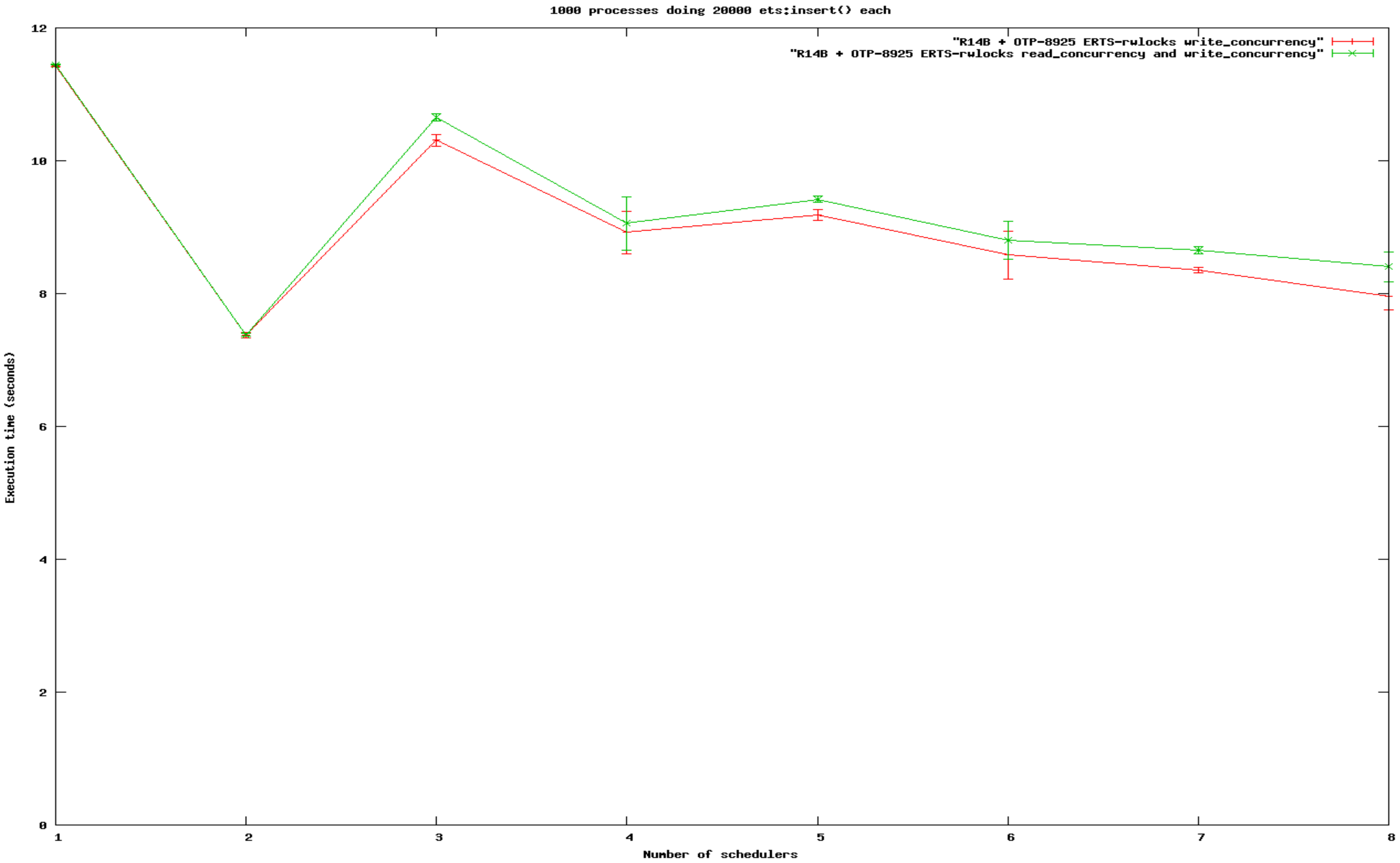
Benchmark Results – Machine A



Benchmark Results – Machine A

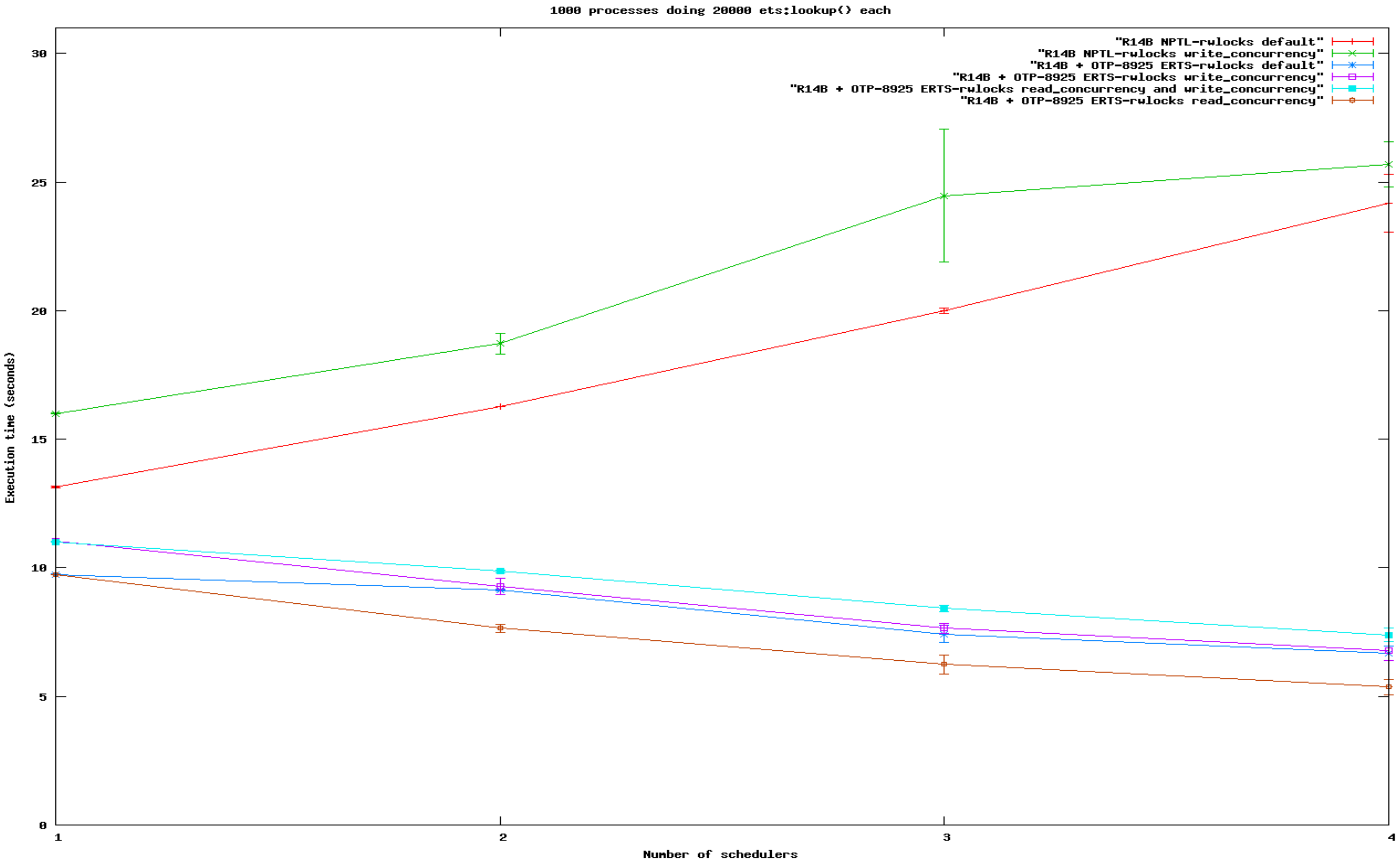


Benchmark Results – Machine A

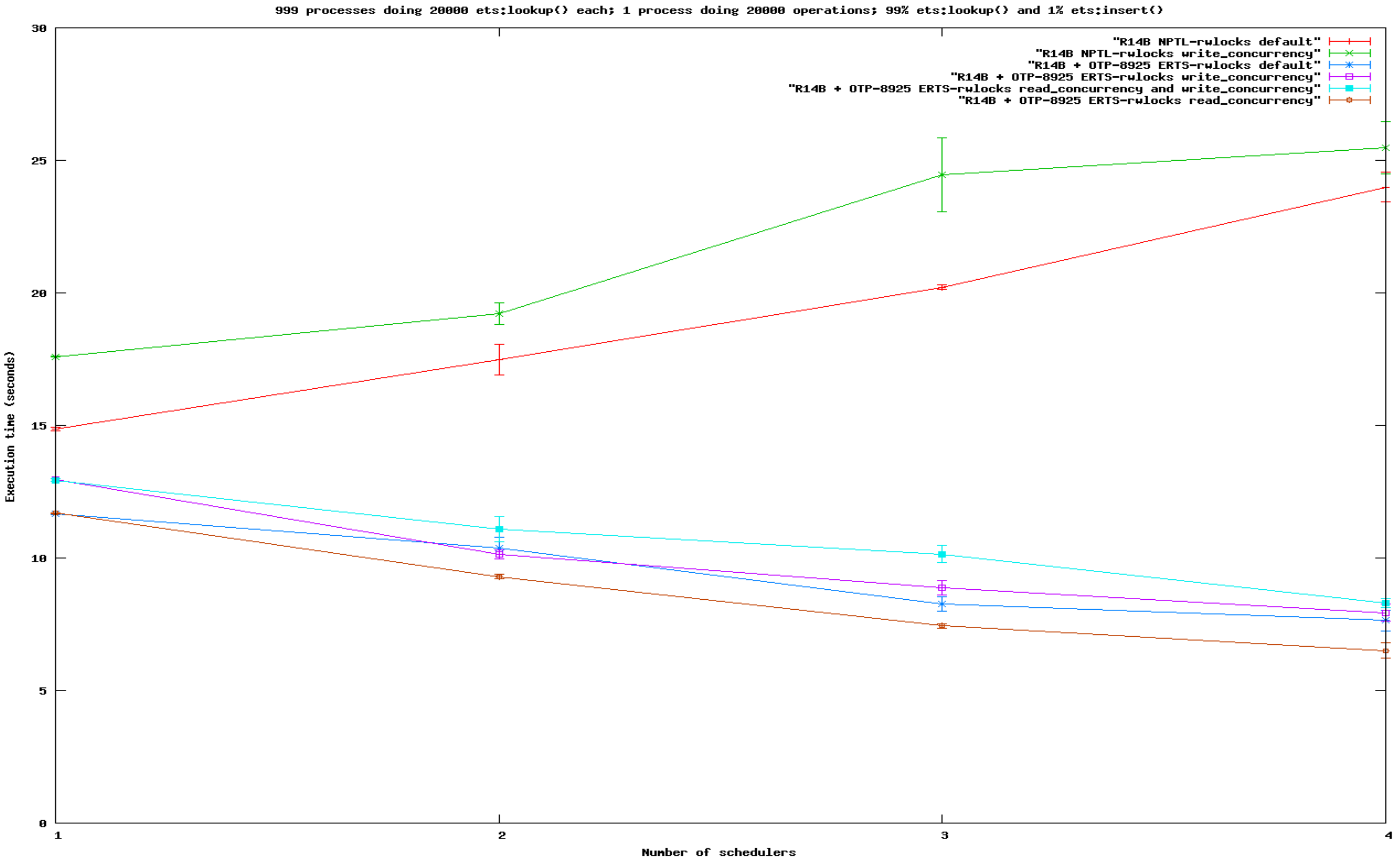


ERICSSON

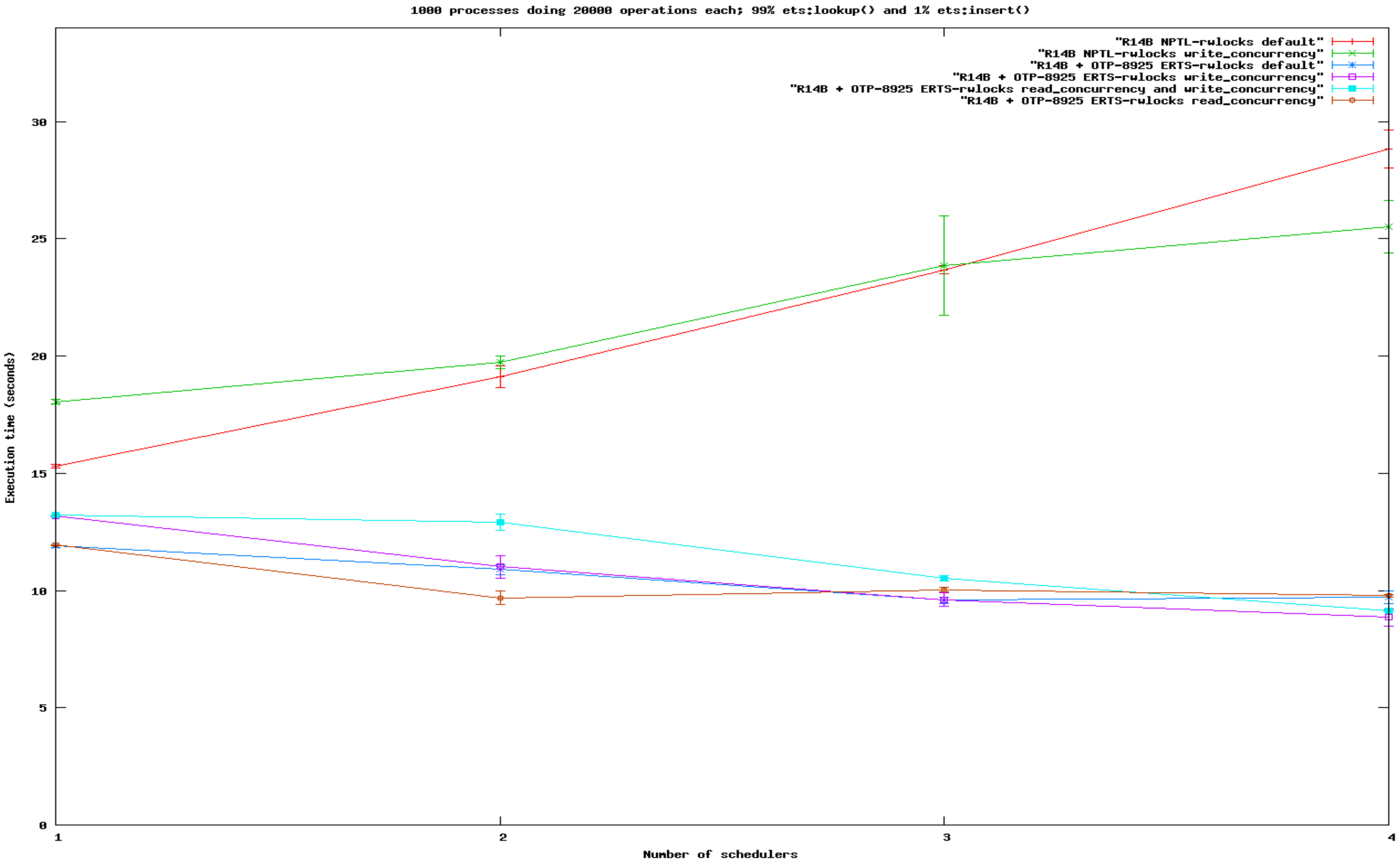
Benchmark Results – Machine B



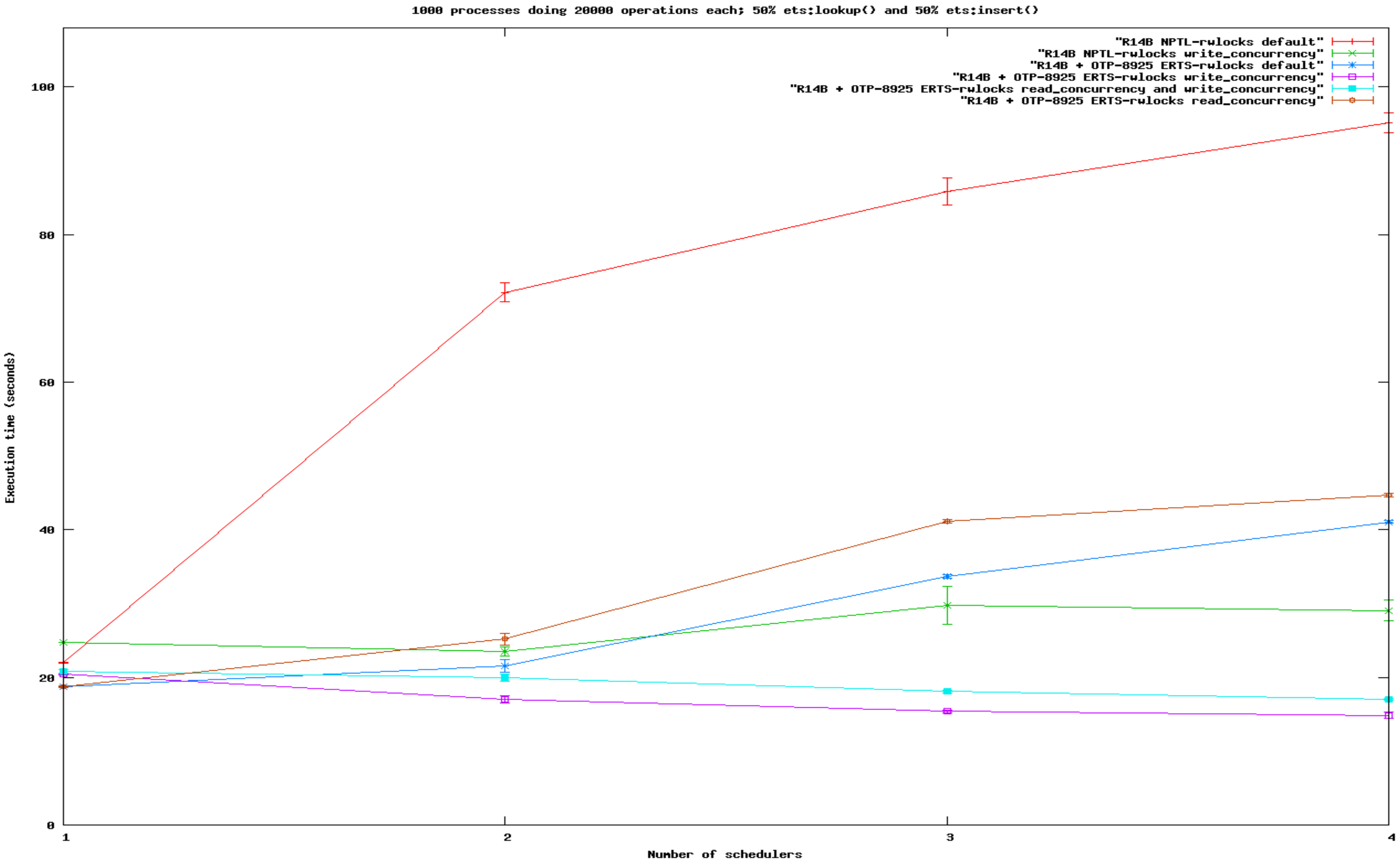
Benchmark Results – Machine B



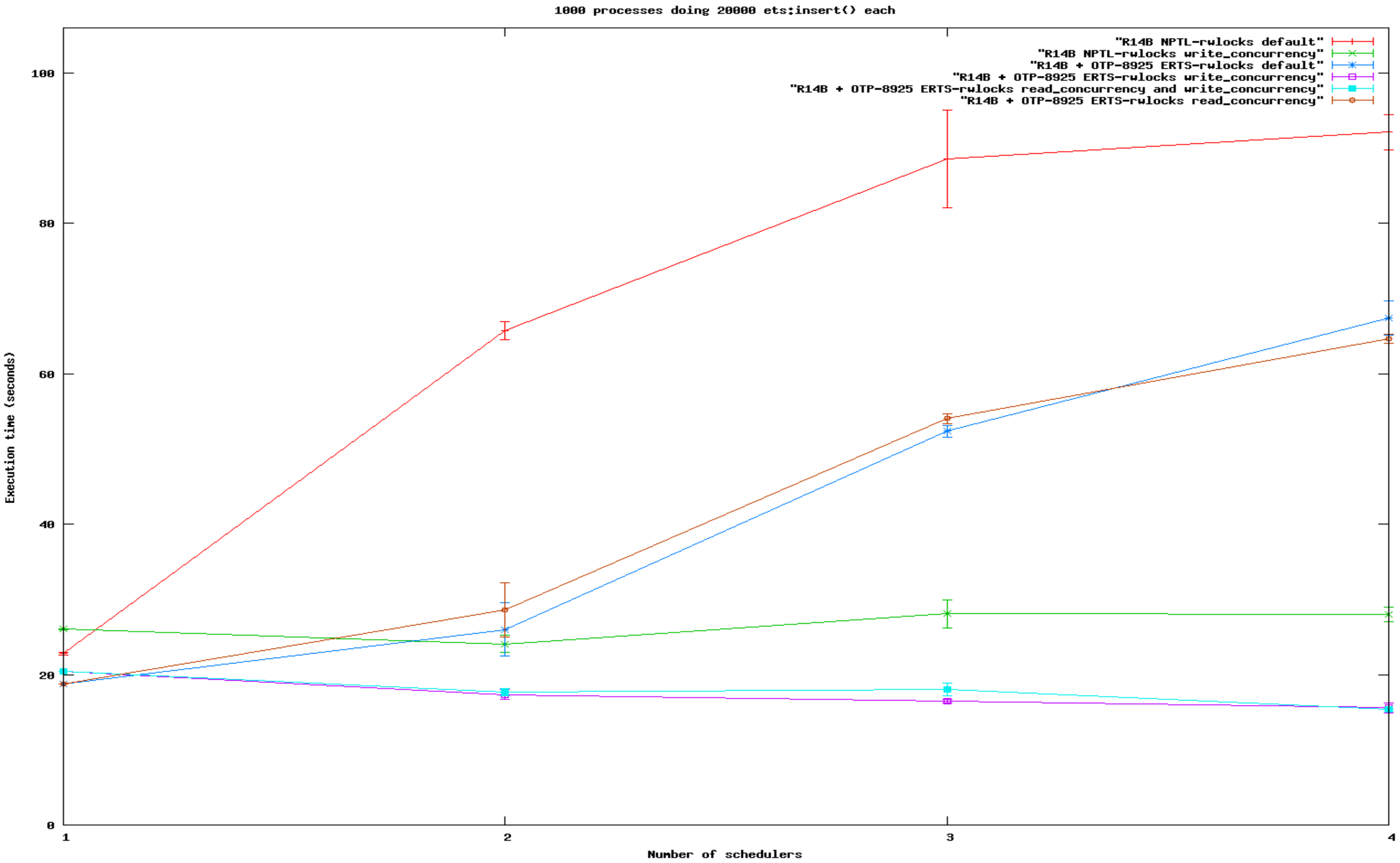
Benchmark Results – Machine B



Benchmark Results – Machine B



Benchmark Results – Machine B



ERICSSON