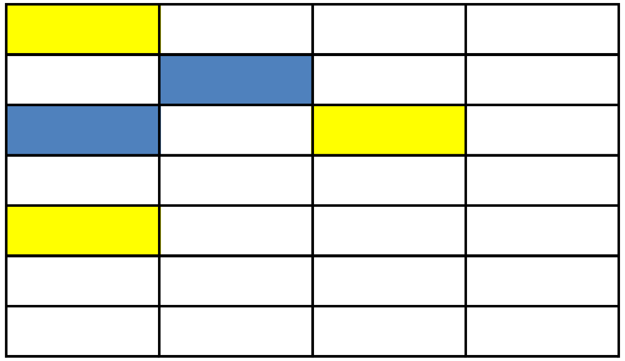
Adaptive Queued Locking to Optimize Transactional Memory

Tim Chen (tim.c.chen@linux.intel.com)

Transactional Memory, where it works great

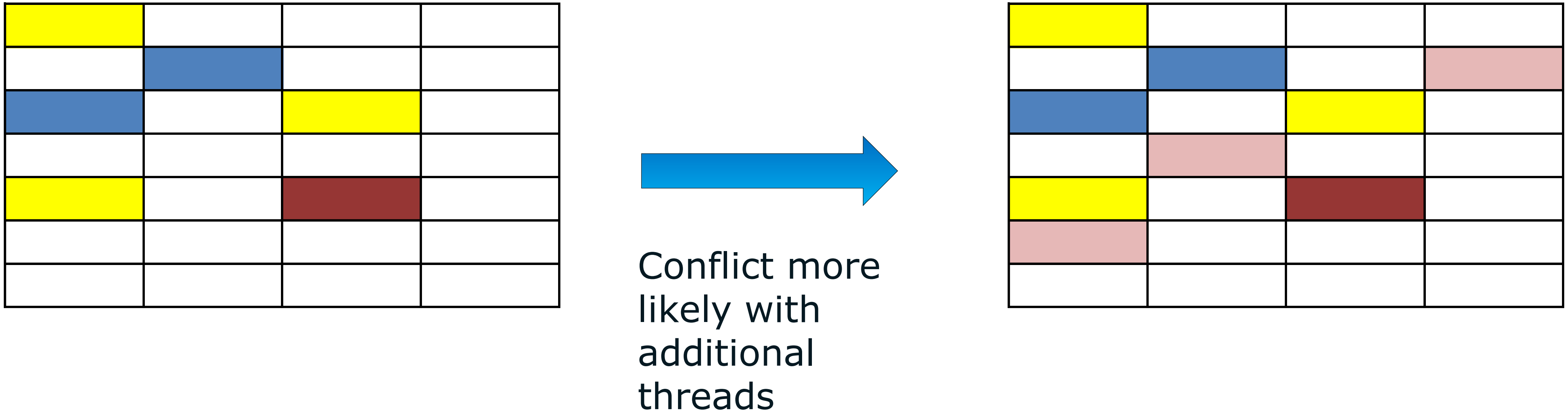
- Hardware tracks conflict of working data set for threads in critical section, very low overhead
- More than 1 thread can run in critical section
- Great parallelism, no locking!



Memory location access
when running in critical section

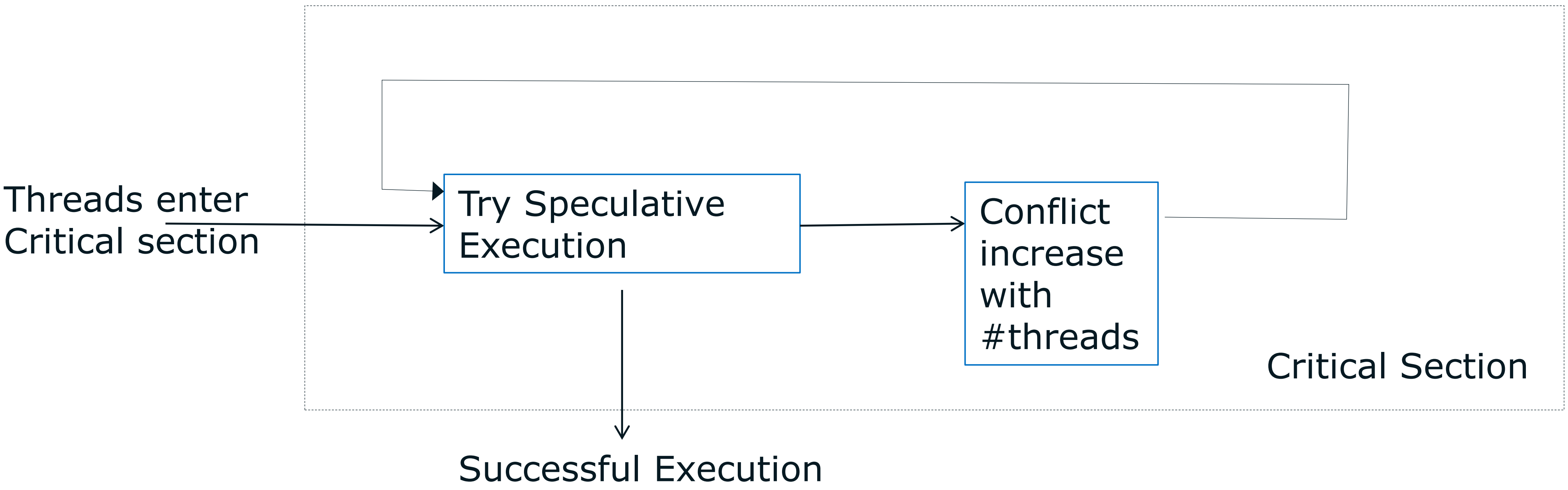
Transactional Memory, where things slow down

- Data conflict when one thread write to memory another thread has read/written, need to abort.
- What can we do: Retry
- Other threads can enter the critical section in the mean time, likelihood of conflict increases if we don’t lock explicitly



Conflict more likely with additional threads

Pile Up when Retrying with Failed Speculations

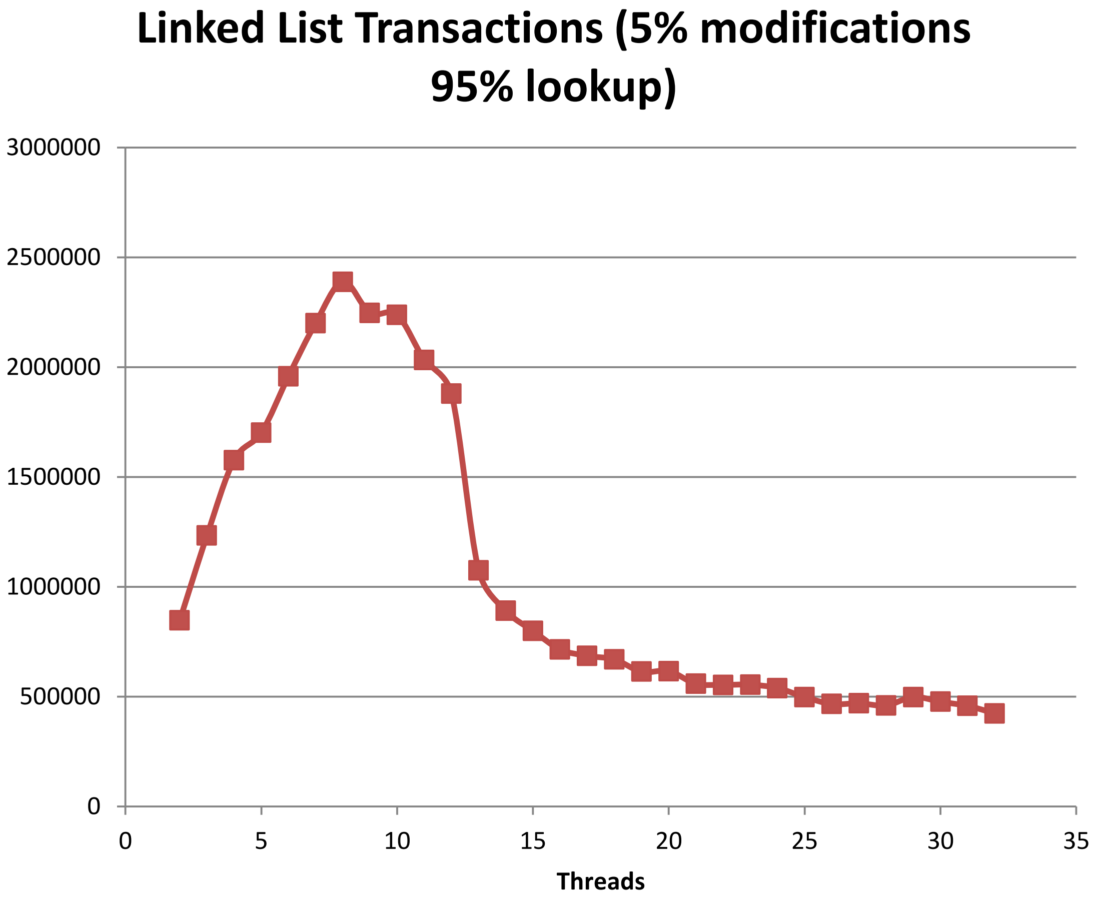

Threads enter Critical section
Try Speculative Execution
Conflict increase with #threads
Critical Section
Successful Execution

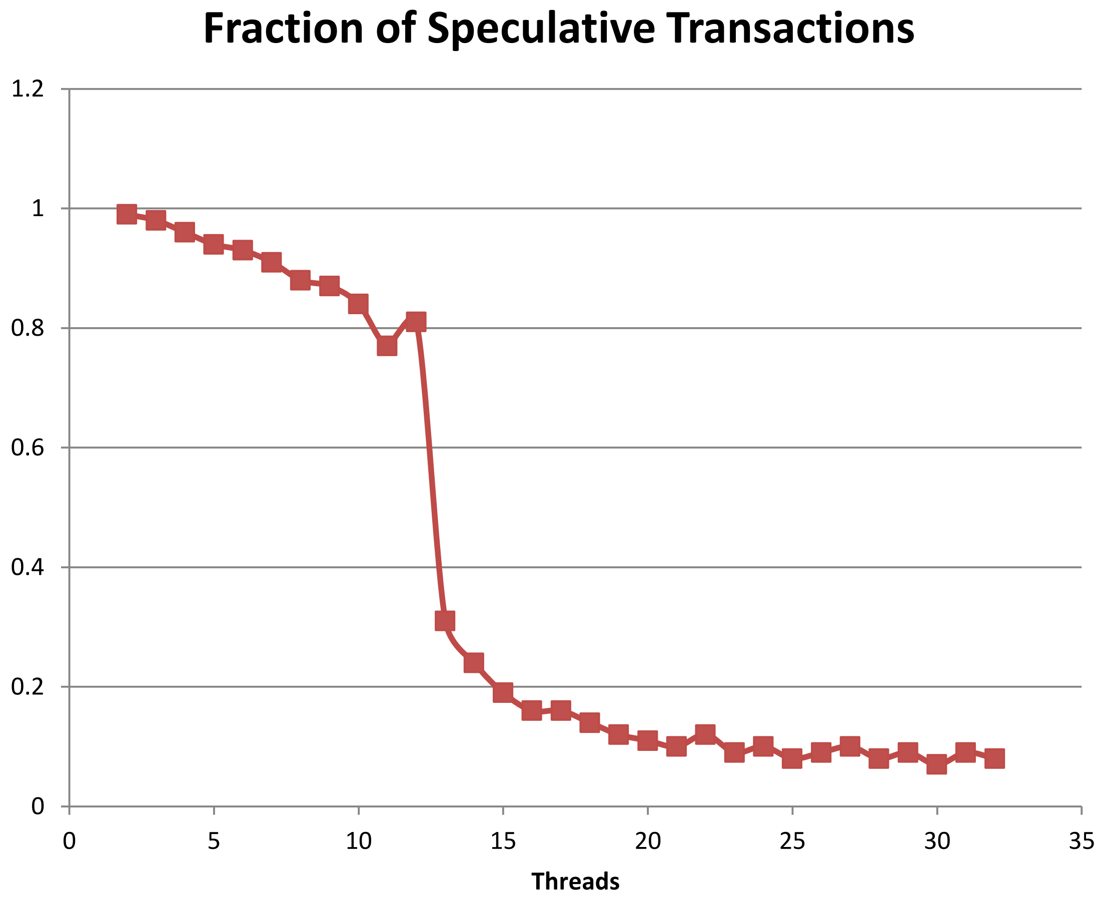
Pileup begins when #threads enter > #threads complete
#threads completed goes down quickly due to increase conflicts
Arrgh! we still need to lock after all, any way to avoid locking?

A mechanism to regulate #threads executing in critical section to prevent pileup causing successful speculation going to zilch

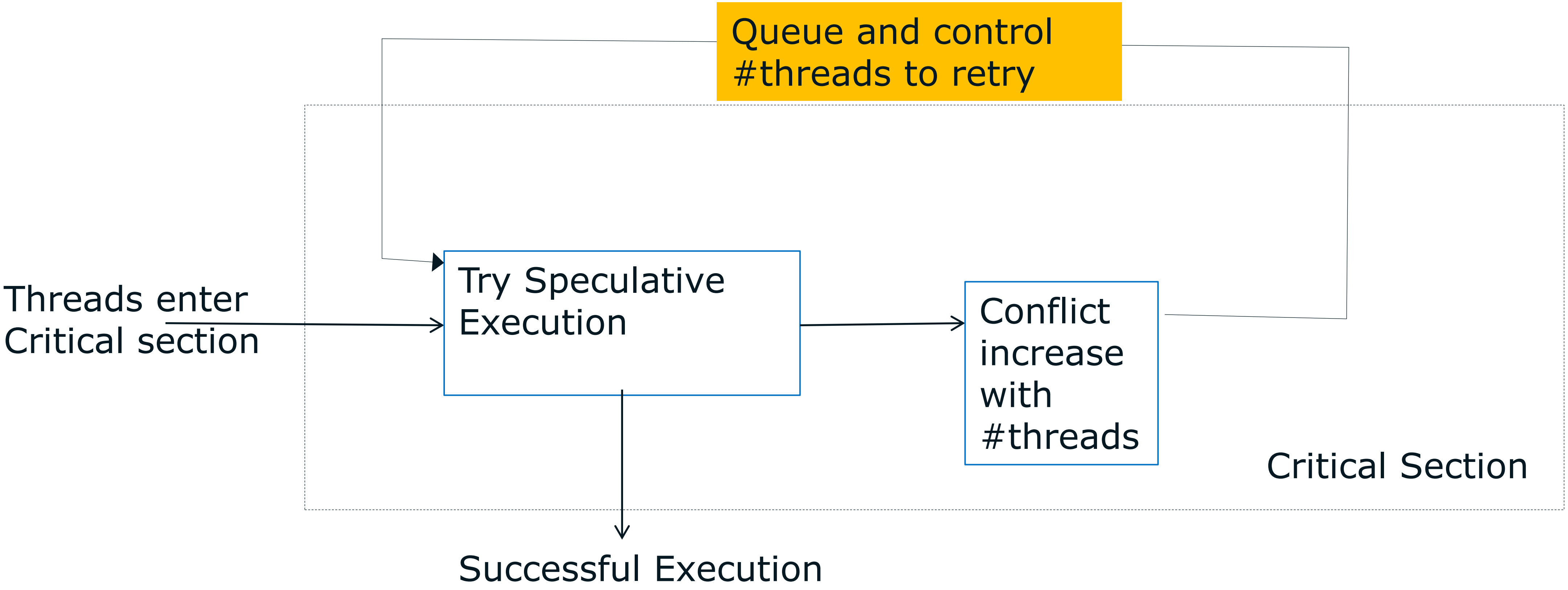
Problem with Retry of Speculative Execution

Linked list access with max of 3 retries allowed





Regulate the Number of Threads



Queue and control #threads to retry

Threads enter Critical section

Try Speculative Execution

Conflict increase with #threads

Critical Section

Successful Execution

Aperture Concept

- Regulate the number of speculative threads entering the critical section after abort

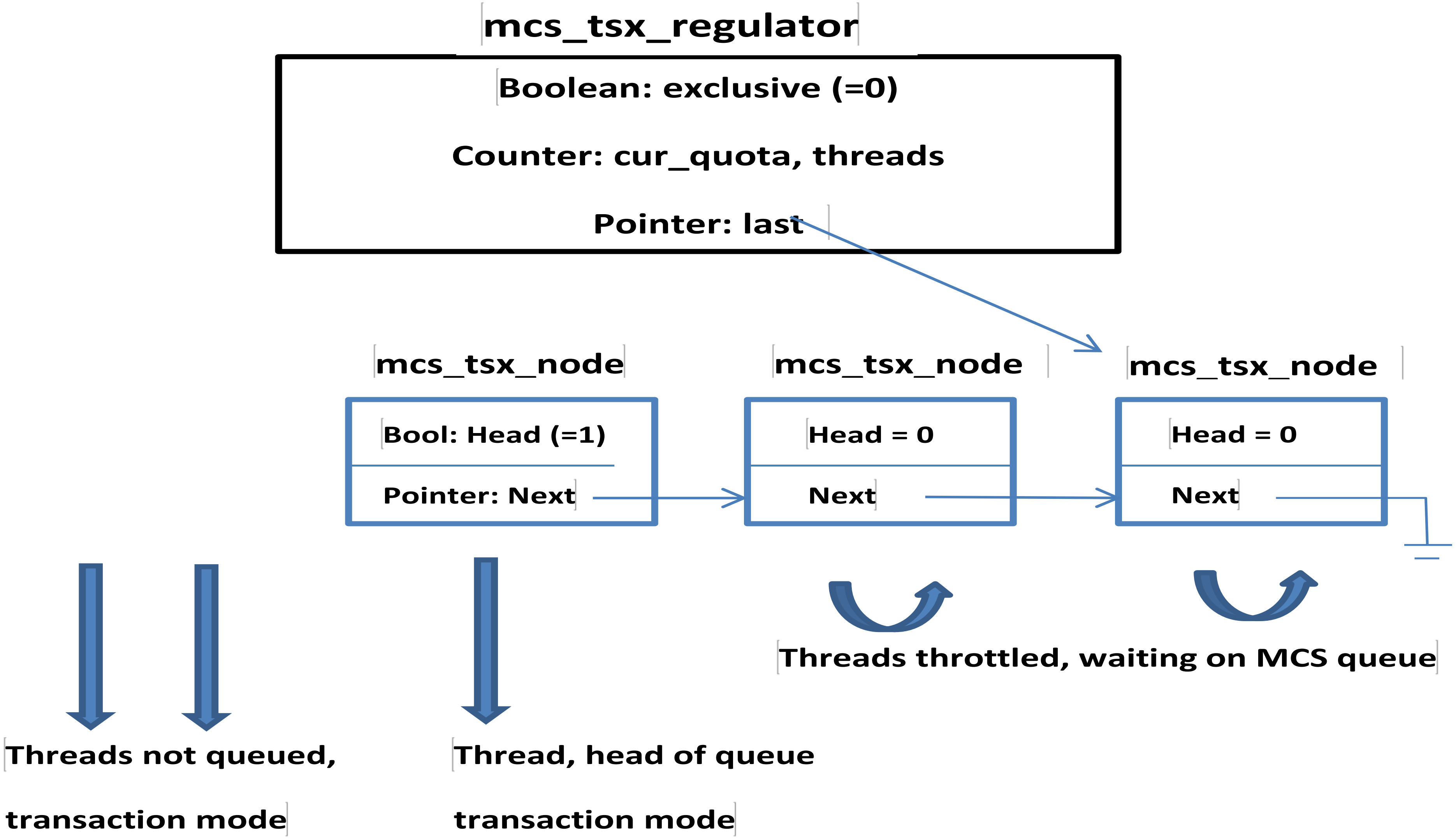
- Increase or decrease the aperture based on the abort rate

- Queue up aborted threads and limit #threads allowed to retry

MCS lock provides a distributed queueing mechanism

- We can take advantage of MCS distributed queueing mechanism,
- Allow more than one thread into the critical section
- Thread at head of MCS queue performing regulation duties: admission to critical section, monitor abort rate, aperture adjustment
- It is a self adaptive scheme, no prior optimization needed

Regulated Speculative Transaction



mcs_tsx_regulator

Boolean: exclusive (=0)

Counter: cur_quota, threads

Pointer: last

mcs_tsx_node

Bool: Head (=1)

Pointer: Next

mcs_tsx_node

Head = 0

Next

mcs_tsx_node

Head = 0

Next

Threads throttled, waiting on MCS queue

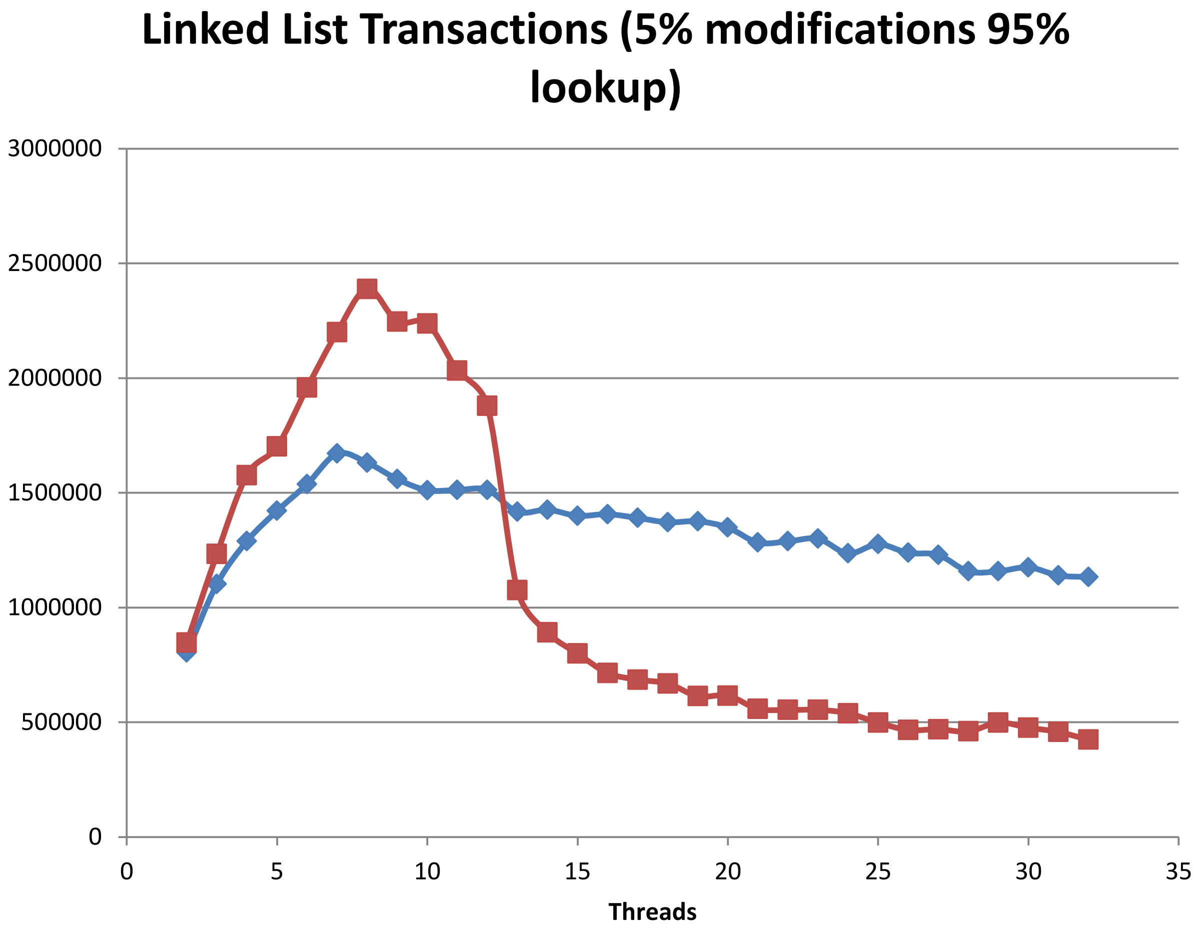
Threads not queued,

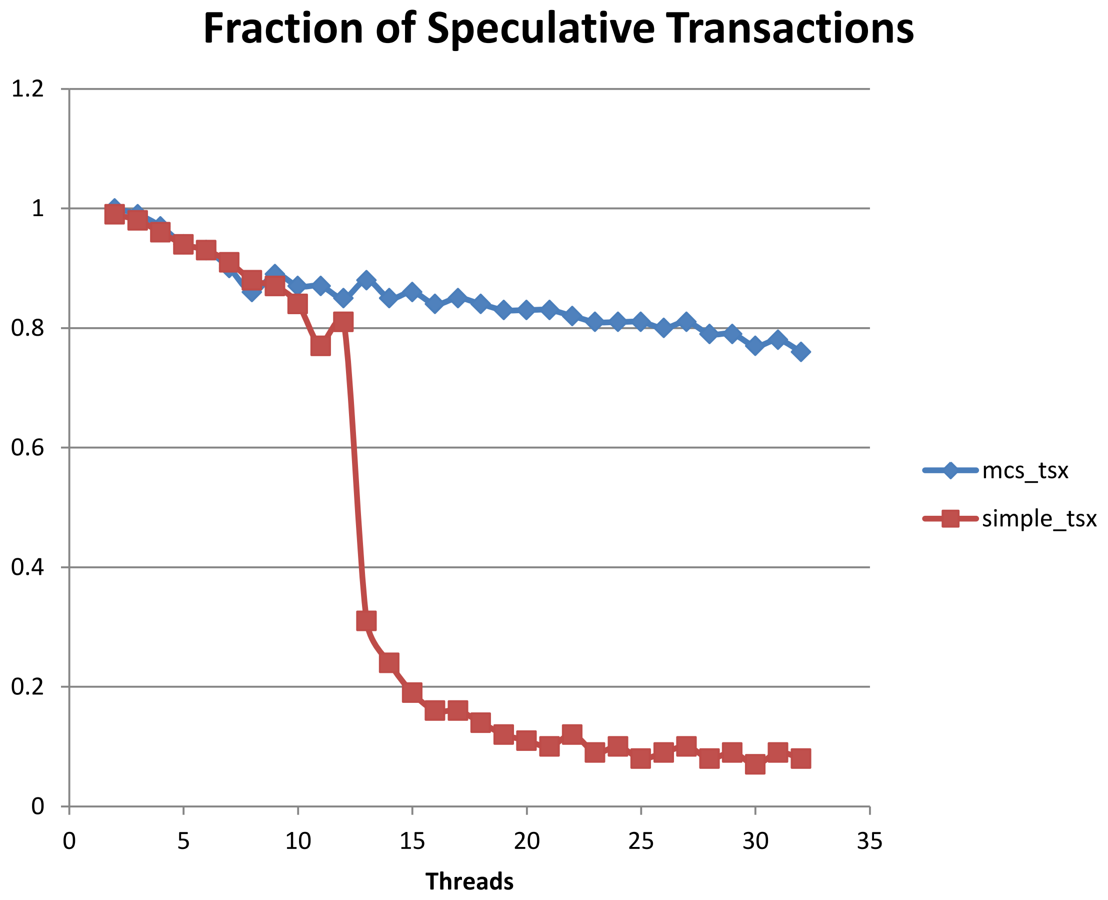
transaction mode

Thread, head of queue

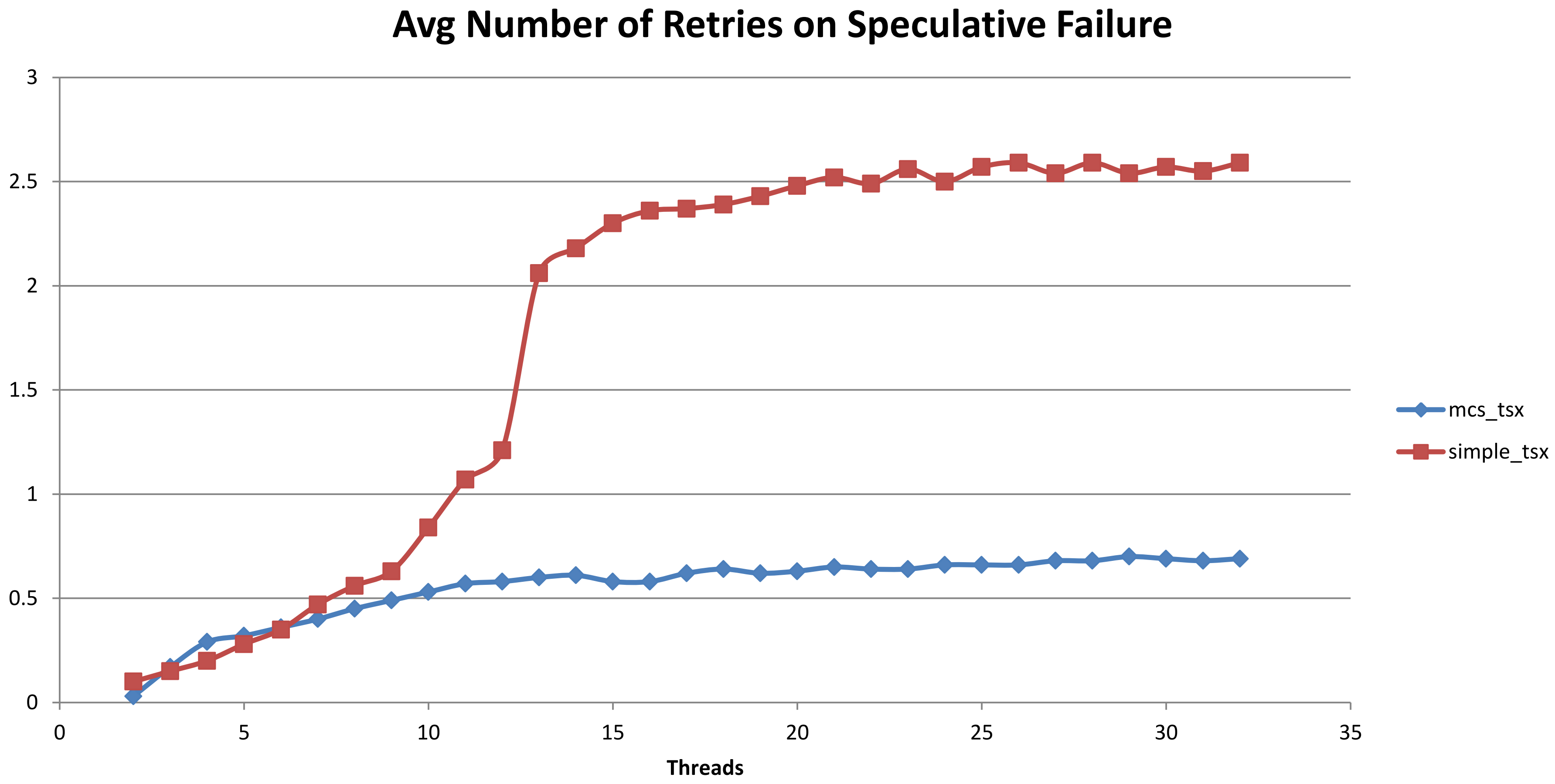
transaction mode

MCS queued Locking with Adaptive Aperture





How Often do We Repeat after Abort?



Observations

- Throughput 2 to 3 times of normal transactional memory that uses retry and locking fallback at high thread counts.
- Does not work as well with small number of threads
  - The aperture adapt down too quickly?
  - Overhead more on updating count of threads in critical region, pointer update to queueing.
- Q-spinklock approach from Waiman to shrink the lock structure, retry and don’t queue on first abort
- Queued locking shows promise, we have more work to do to tune its behavior

Acknowledgements

* Andi Kleen – who provided many great insights to prompt this work

Thank You