Performance of Lock-free Data Structures: Models and Analyses

Philippas Tsigas

Chalmers University of Technology

Motivation

- Lock-free Data Structures:
  - Limitations of their lock-based counterparts: deadlocks, convoying and programming flexibility
  - Provide high scalability
  - Existing analyses focus on asymptotic behavior
- Framework to estimate the performance:
  - Facilitate efficient lock-free designs
  - Compare lock-free implementations
  - Facilitate analytically data structure implementation optimizations (*i.e.* back-off, memory management)

Model

Output of the analysis: Data structure throughput, *i.e.* number of successful operations per unit of time

Procedure AbstractAlgorithm

```
Initialization();
while ! done do
    Parallel_Work();        /* Application specific code, conflict-free */
    while ! success do
        current ← Read(AP);
        new ← Critical_Work(current);
        success ← CAS(AP, current, new);
```

Model

Output of the analysis: Data structure throughput, *i.e.* number of successful operations per unit of time

Procedure AbstractAlgorithm

```
Initialization();
while ! done do
    Parallel_Work();                /* Application specific code, conflict-free */
    while ! success do
        current ← Read(AP);
        new ← Critical_Work(current);
        success ← CAS(AP, current, new);
```

Model

Output of the analysis: Data structure throughput, *i.e.* number of successful operations per unit of time

Procedure AbstractAlgorithm

```
Initialization();
while ! done do
    Parallel_Work();                 /* Application specific code, conflict-free */
    while ! success do
        current ← Read(AP);
        new ← Critical_Work(current);
        success ← CAS(AP, current, new);
```

Model

Output of the analysis: Data structure throughput, *i.e.* number of successful operations per unit of time

Procedure AbstractAlgorithm

```
Initialization();
while ! done do
    Parallel_Work();                    /* Application specific code, conflict-free */
    while ! success do
        current ← Read(AP);
        new ← Critical_Work(current);
        success ← CAS(AP, current, new);
```

Model

Output of the analysis: Data structure throughput, *i.e.* number of successful operations per unit of time

Procedure AbstractAlgorithm

```
Initialization();
while ! done do
    Parallel_Work();                    /* Application specific code, conflict-free */
    while ! success do
        current ← Read(AP);
        new ← Critical_Work(current);
        success ← CAS(AP, current, new);
```

Model

Output of the analysis: Data structure throughput, *i.e.* number of successful operations per unit of time

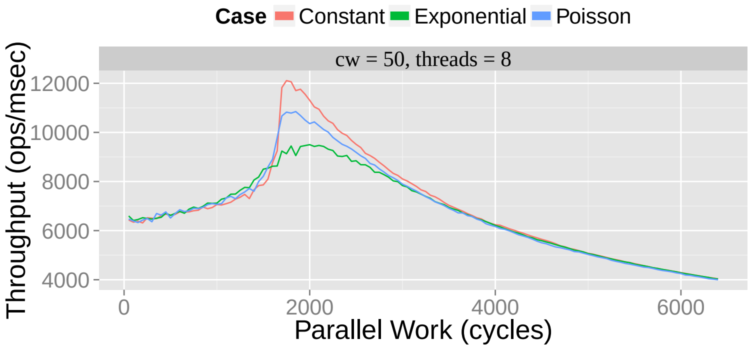
Procedure AbstractAlgorithm

```
Initialization();
while ! done do
    Parallel_Work();                    /* Application specific code, conflict-free */
    while ! success do
        current ← Read(AP);
        new ← Critical_Work(current);
        success ← CAS(AP, current, new);
```

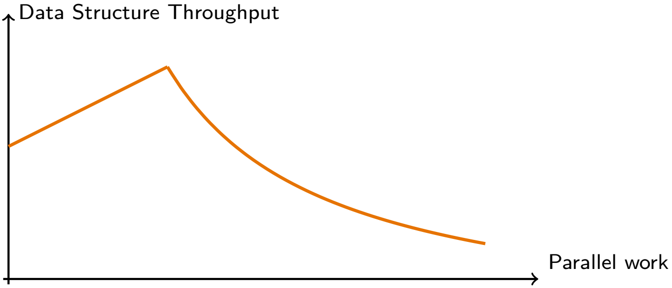
Inputs of the analysis:

- Platform parameters: *CAS* (*cc*) and Read (*rc*) latencies, in clock cycles
- Algorithm parameters:
  - Critical Work (*cw*) and Parallel Work (*pw*) latencies, in clock cycles
  - Total number of threads(P)

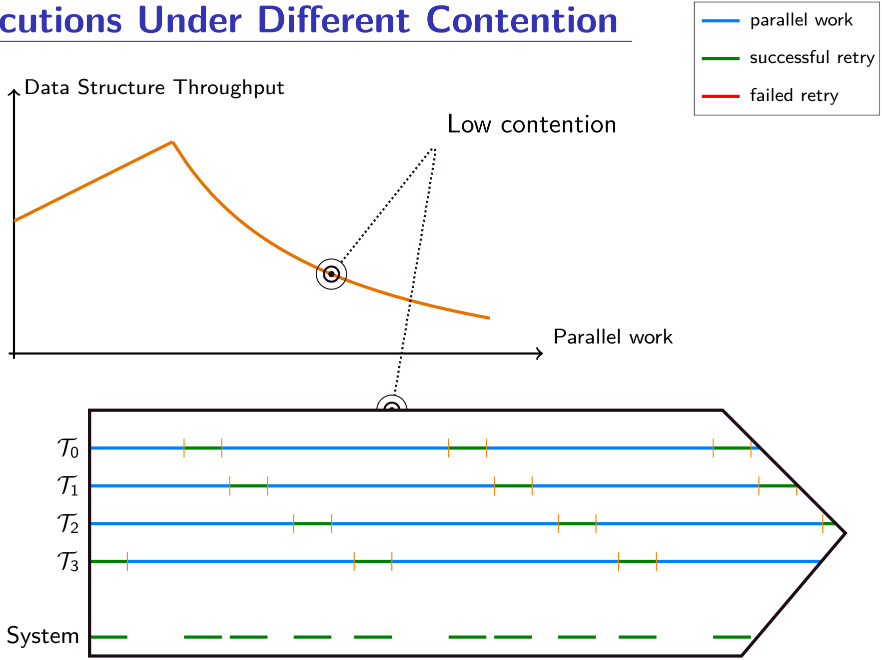
Example: Treiber Stack Pop operation



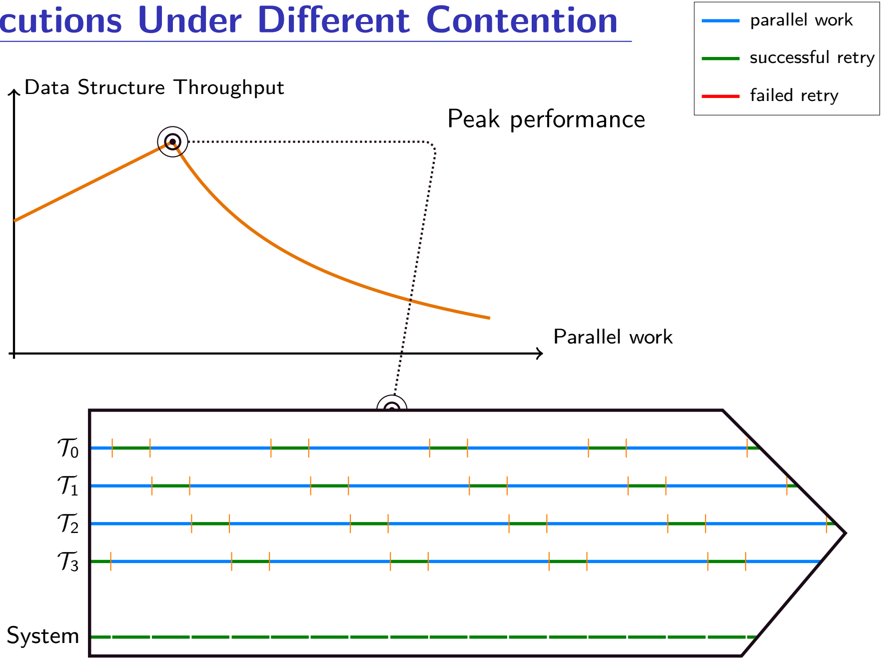
Executions Under Different Contention



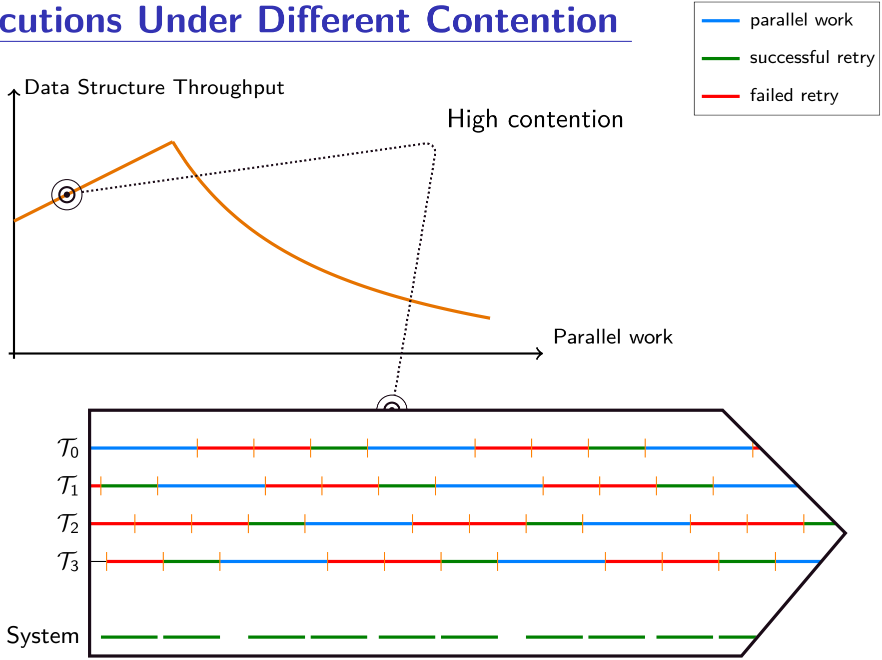
Executions Under Different Contention



Executions Under Different Contention



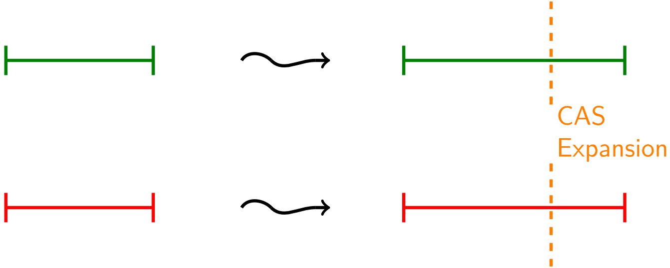
Executions Under Different Contention



Impacting Factors

- Failed Retries

- Atomic CAS Conflicts

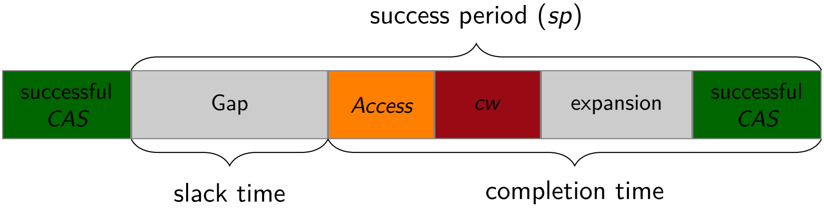


Analyses

- General case: parallel work follows an arbitrary distribution
- Special case 1: parallel work is a constant
- Special case 2: parallel work follows exponential distribution
- The analyses are centered around a single variable P_{rl}, the number threads inside the retry loop

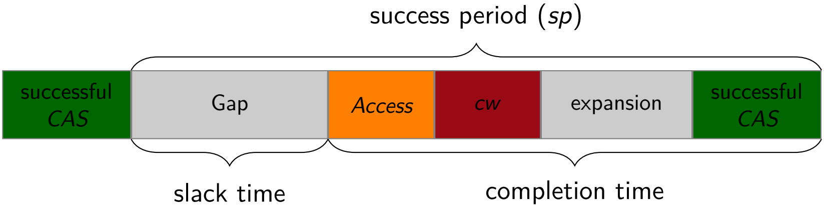
General Case

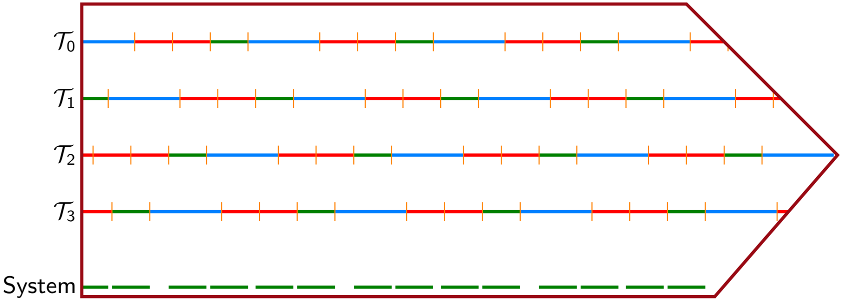
Analyses: Breakdown of the execution



- Slack time: the gaps in between successful retry loops (a successful retry does not start immediately after the previous successful one, See System perspective in the previous figures)
- Completion time: the time from the beginning of a retry loop to its end
- Expected success period = Expected Completion time + Expected Slack time
- Throughput = 1 / Expected success period

Analyses: Breakdown of the execution





General Case: Average-Based Approach

- Throughput: expectation of success period at a random time
- Relies on queueing theory (Little's law) and focus on average behaviour

$$\overline{sp}\left(\overline{P_{rl}}\right) = pw/(P - \overline{P_{rl}}) \tag{1}$$

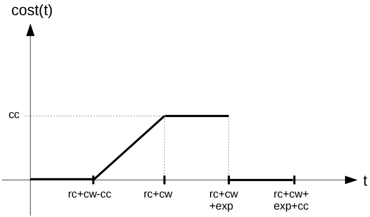
- Assuming two modes of contention:
  - Non-contended:

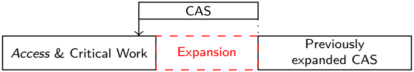
$$\overline{sp}\left(\overline{P_{rl}}\right) = (rc + cw + cc + pw)/P = (rc + cw + cc)/\overline{P_{rl}} \tag{2}$$

  - Contended:
    (i) Given $\overline{P_{rl}}$, calculate the expected expansion: $\overline{e}\left(\overline{P_{rl}}\right)$
    (ii) Given $\overline{P_{rl}}$, calculate the expected slack time: $\overline{st}\left(\overline{P_{rl}}\right)$

CAS Expansion

- Input: P_{rl} threads already in the retry loop
- A new thread attempts to CAS during CompletionTime $(Access + cw + \overline{e}\left(\overline{P_{rl}}\right) + CAS)$, within a probability h:
- Cost function:





$$\rightsquigarrow \overline{e}\left(\overline{P_{rl}} + h\right) = \overline{e}\left(\overline{P_{rl}}\right) + h \times \int_0^{CompletionTime} \frac{cost(t)}{CompletionTime}\, dt.$$

Lemma

The expansion of a CAS operation is the solution of the following system of equations:

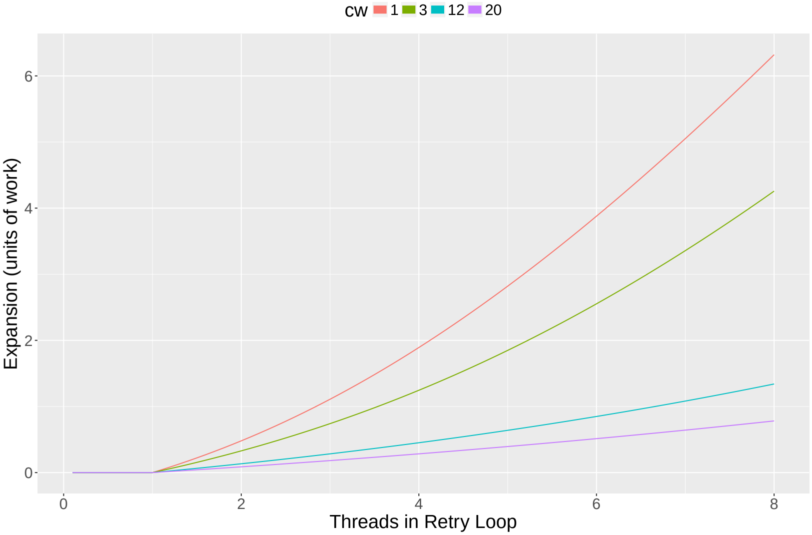
$$\begin{cases} e'\left(P_{rl}\right) & = & cc \times \dfrac{\frac{cc}{2} + e\left(P_{rl}\right)}{rc + cw + cc + e\left(P_{rl}\right)} \\ e\left(P_{rl}^{(0)}\right) & = & 0 \end{cases}$$

We compute $e\left(P_{rl} + h\right)$, where $h \leq 1$, by assuming that there are already P_{rl} threads in the retry loop, and that a new thread attempts to *CAS* during the retry, within a probability h:

$$
\begin{aligned}
e\left(P_{rl}+h\right) &= e\left(P_{rl}\right) + h \times \int_0^{rlw^{(+)}} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \\
&= e\left(P_{rl}\right) + \Big(\int_0^{rc+cw-cc} \frac{d\left(t\right)}{rlw^{(+)}}\, dt + \int_{rc+cw-cc}^{rc+cw} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \\
&\quad + \int_{rc+cw}^{rc+cw+e(P_{rl})} \frac{d\left(t\right)}{rlw^{(+)}}\, dt + \int_{rc+cw+e(P_{rl})}^{rlw^{(+)}} \frac{d\left(t\right)}{rlw^{(+)}}\, dt \Big) h \\
&= e\left(P_{rl}\right) + \Big(\int_{rc+cw-cc}^{rc+cw} \frac{t}{rlw^{(+)}}\, dt + \int_{rc+cw}^{rc+cw+e(P_{rl})} \frac{cc}{rlw^{(+)}}\, dt \Big) h \\
&= e\left(P_{rl}\right) + h \times \frac{\frac{cc^2}{2} + e\left(P_{rl}\right) \times cc}{rlw^{(+)}}.
\end{aligned}
$$

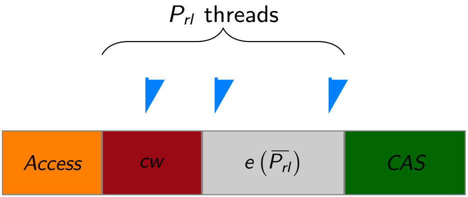
This leads to $\dfrac{e\left(P_{rl}+h\right) - e\left(P_{rl}\right)}{h} = \dfrac{\frac{cc^2}{2} + e\left(P_{rl}\right) \times cc}{rlw^{(+)}}$. When making h tend to 0, we finally obtain $e'\left(P_{rl}\right) = cc \times \frac{\frac{cc}{2} + e(P_{rl})}{rc+cw+cc+e(P_{rl})}$.

Expansion Model



Slack Time

- Input: P_{rl} threads already in the retry loop
- Assume a thread has equal probability to be anywhere in the critical work or expansion



- Expectation of the minimum distance to *Access* (failed *CAS*)

$$\overline{st}\left(\overline{P_{rl}}\right) = \left(cw + e\left(\overline{P_{rl}}\right)\right)/\left(\overline{P_{rl}} + 1\right) \tag{3}$$

Unified Solving for Throughput Estimate

- Unified solving:

$$\frac{rc + cw + cc}{\overline{P_{rl}}} = \frac{\overline{P_{rl}} + 2}{\overline{P_{rl}} + 1} \left(cw + \overline{e} \left(\overline{P_{rl}} \right) \right) + 2cc, \tag{4}$$

The system switches from being non-contended to being contended at $\overline{P_{rl}} = P_{rl}^{(0)}$, where

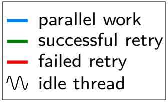
$$P_{rl}^{(0)} = \frac{cc + cw - rc}{2(cw + 2cc)} \left(\sqrt{1 + \frac{4(rc + cw + cc)(cw + 2cc)}{(cc + cw - rc)^2}} - 1 \right).$$

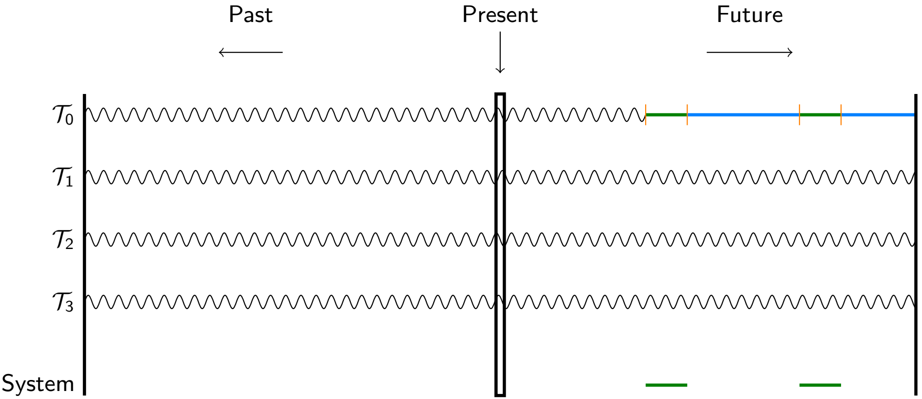
- Fixed point iteration on $\overline{P_{rl}}$ to find the value that obeys Little's Law

Special Case (Constant parallel work)

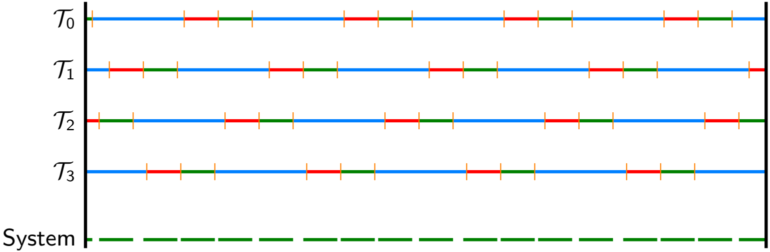
(f)-Cyclic Executions

- Periodic: every thread is in the same state as one period before
- Shortest period contains exactly 1 successful attempt and exactly f fails per thread

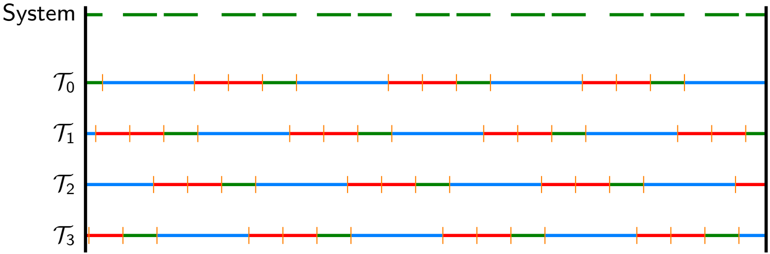




Inevitable and Wasted Failures



vs.



Throughput: Combining Impacting Factors

- Input: P_{rl} (Average number of threads inside retry loop)
  1. Calculate expansion: $e\left(P_{rl}\right)$
  2. Compute amount of work in a retry:
  $$Retry = Read + Critical\_Work + e\left(P_{rl}\right) + CAS$$
  3. Estimate number of logical conflicts:
  $$LogicalConflicts(Retry, Parallel\_Work, Threads)$$

  $\rightsquigarrow$ Average number of threads inside the retry loop

Throughput: Combining Impacting Factors

- Input: P_{rl} (Average number of threads inside retry loop)
  1. Calculate expansion: $e\left(P_{rl}\right)$
  2. Compute amount of work in a retry:

     $$Retry = Read + Critical\_Work + e\left(P_{rl}\right) + CAS$$

  3. Estimate number of logical conflicts:

     $$LogicalConflicts(Retry, Parallel\_Work, Threads)$$

     $\rightsquigarrow$ Average number of threads inside the retry loop

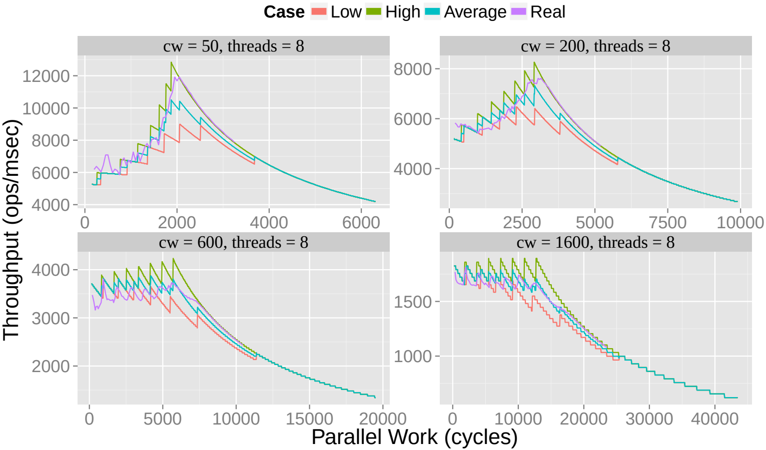
- Convergence via fixed point iteration

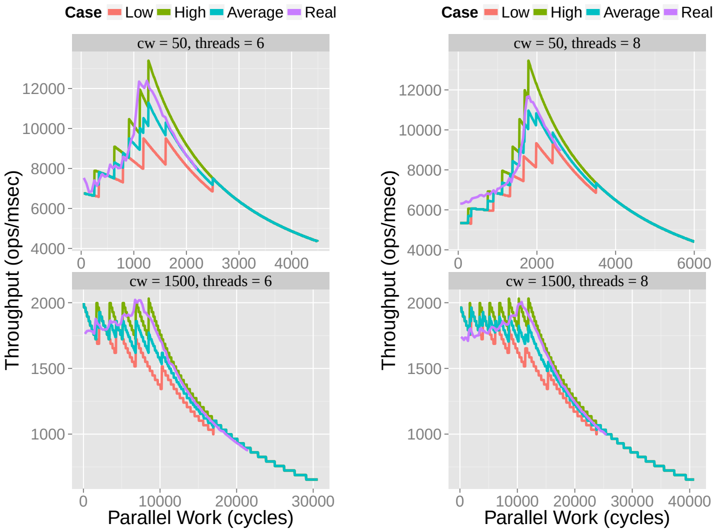
Special Case (Parallel work follows exponential distribution)

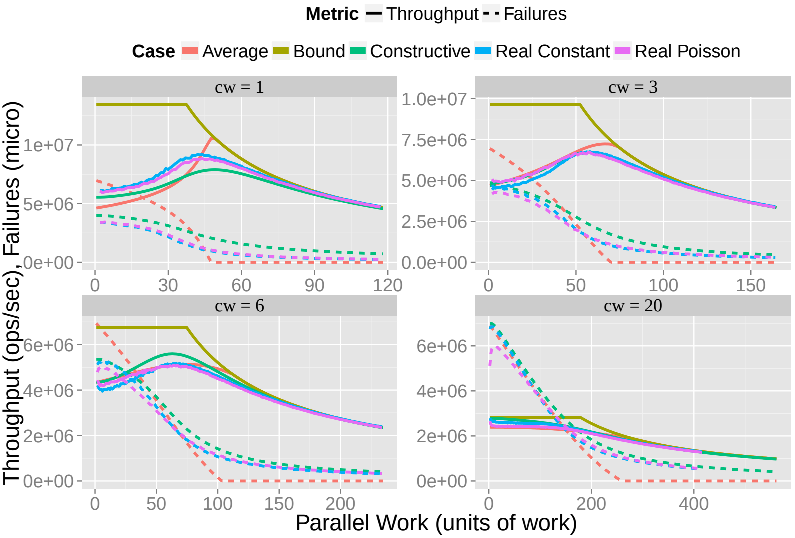
Special Case: Constructive Approach

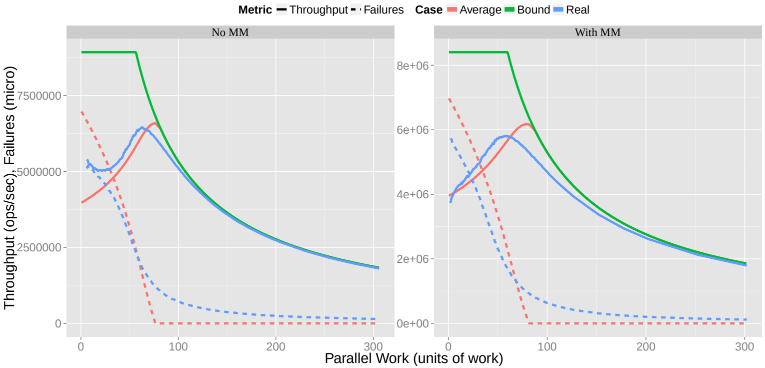
- Construct the execution step by step (based on Markov Chains)
- P_{rl} renders the state of the system. System is in state i, if there are i threads inside the retry loop, and the system changes state after success *CAS*.
- In state i, we know that there are $P - i$ threads in the parallel work
- Parallel work follows exponential distribution (memoryless), we do not need to track $P - i$ threads in the parallel work
- Transition probabilities (state i to $i + k$): estimate the success period given that we are in state i then consider the probability of $k + 1$ threads to leave the parallel work and enter to the retry loop during this interval
- Calculate stationary distribution and stochastic sequence of success periods results in the throughput estimate

Results


Case Low High Average Real
cw = 50, threads = 8
cw = 200, threads = 8
cw = 600, threads = 8
cw = 1600, threads = 8
12000
10000
8000
6000
4000
0
2000
4000
6000
8000
6000
4000
0
2500
5000
7500
10000
4000
3000
2000
0
5000
10000
15000
20000
1500
1000
0
10000
20000
30000
40000
Throughput (ops/msec)
Parallel Work (cycles)


Case
Low
High
Average
Real
cw = 50, threads = 6
cw = 50, threads = 8
cw = 1500, threads = 6
cw = 1500, threads = 8
Throughput (ops/msec)
Parallel Work (cycles)
12000
10000
8000
6000
4000
0
1000
2000
3000
4000
6000
1500
10000
20000
30000
40000


Metric Throughput Failures
Case Average Bound Constructive Real Constant Real Poisson
cw = 1
cw = 3
cw = 6
cw = 20
Throughput (ops/sec), Failures (micro)
Parallel Work (units of work)

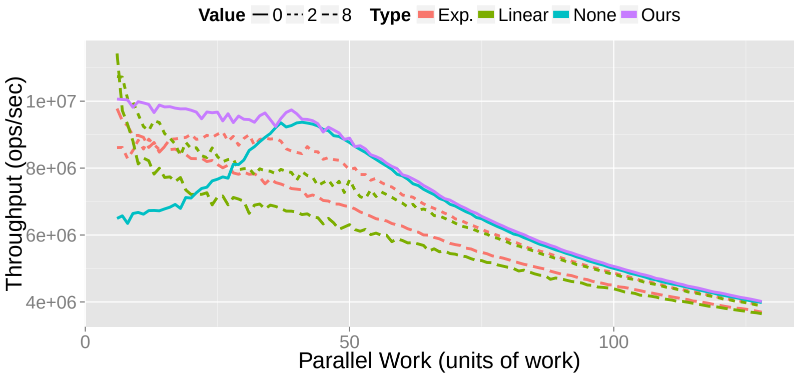


Metric
Throughput
Failures
Case
Average
Bound
Real
No MM
With MM
Throughput (ops/sec), Failures (micro)
7500000
5000000
2500000
0
8e+06
6e+06
4e+06
2e+06
0e+00
0
100
200
300
Parallel Work (units of work)

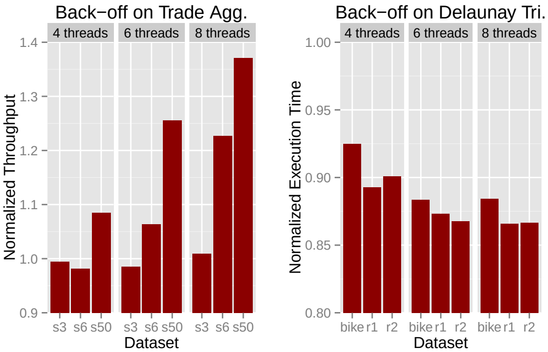
Applications

Back-off: Treiber Stack Pop

- Our Back-off vs. Exponential and Linear Back-off



- Delaunay Triangulation (**pw is known**): back-off for the time difference between the peak pw (computed by our analysis) and the actual pw
- Workload originated from global operators of exchanges for financial markets (**pw is unknown**): estimate the pw value from the number of fails with a sliding window



- Memory management introduces extra work
- Traditionally, a big block of work that is executed once in a while, after reaching a threshold for the number of object waiting for reclamation
- Twist:
  - Split this big block into equally sized smaller chunks
  - Track the number of fails to determine contention
  - No MM execution under low contention
  - Call MM (as back-off) only under high contention

Conclusion

- Three new analyses for the performance of lock-free data structures
- Validate our model using synthetic tests and several reference data structures (deque, queue, stack, shared counter, priority queue)
- Exploit our analyses for back-off and memory management optimization
- For details, please see [1] and [2]

References

[1] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. "Analyzing the Performance of Lock-Free Data Structures: A Conflict-Based Model". In: *Distributed Computing - 29th International Symposium, DISC 2015, Tokyo, Japan, October 7-9, 2015, Proceedings*. 2015, pp. 341–355. URL: https://doi.org/10.1007/978-3-662-48653-5_23.

[2] Aras Atalar, Paul Renaud-Goud, and Philippas Tsigas. "How Lock-free Data Structures Perform in Dynamic Environments: Models and Analyses". In: *20th International Conference on Principles of Distributed Systems, OPODIS 2016, December 13-16, 2016, Madrid, Spain*. 2016, 23:1–23:17. URL: https://doi.org/10.4230/LIPIcs.OPODIS.2016.23.