



Distributed
Computing
and Systems

Analyzing the Performance of Lock-Free Data Structures: A Conflict-based Model

Aras Atalar, Paul Renaud-Goud
and Philippas Tsigas

Chalmers University of Technology

- ▶ Lock-free Data Structures:
 - ▶ Literature and industrial applications (Intel's Threading Building Blocks Framework, Java concurrency package)
 - ▶ Limitations of their lock-based counterparts: deadlocks, convoying and programming flexibility
 - ▶ Provide high scalability

- ▶ Lock-free Data Structures:
 - ▶ Literature and industrial applications (Intel's Threading Building Blocks Framework, Java concurrency package)
 - ▶ Limitations of their lock-based counterparts: deadlocks, convoying and programming flexibility
 - ▶ Provide high scalability

- ▶ Framework to characterize the scalability:
 - ▶ Facilitate the lock-free designs
 - ▶ Rank implementations within a fair framework

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

Procedure AbstractAlgorithm

```
1 Initialization();
2 while ! done do
3   Parallel_Work();           /* Application specific code, conflict-free */
4   while ! success do
5     current ← Read(AP);
6     new ← Critical_Work(current);
7     success ← CAS(AP, current, new);
```

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

Procedure AbstractAlgorithm

```
1 Initialization();
2 while ! done do
3   Parallel_Work();           /* Application specific code, conflict-free */
4   while ! success do
5     current ← Read(AP);
6     new ← Critical_Work(current);
7     success ← CAS(AP, current, new);
```

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

Procedure AbstractAlgorithm

```
1 Initialization();
2 while ! done do
3   Parallel_Work();                               /* Application specific code, conflict-free */
4   while ! success do
5     current ← Read(AP);
6     new ← Critical_Work(current);
7     success ← CAS(AP, current, new);
```

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

Procedure AbstractAlgorithm

```
1 Initialization();
2 while ! done do
3   Parallel_Work();           /* Application specific code, conflict-free */
4   while ! success do
5     current ← Read(AP);
6     new ← Critical_Work(current);
7     success ← CAS(AP, current, new);
```

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

Procedure AbstractAlgorithm

```
1 Initialization();
2 while ! done do
3   Parallel_Work();           /* Application specific code, conflict-free */
4   while ! success do
5     current ← Read(AP);
6     new ← Critical_Work(current);
7     success ← CAS(AP, current, new);
```

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

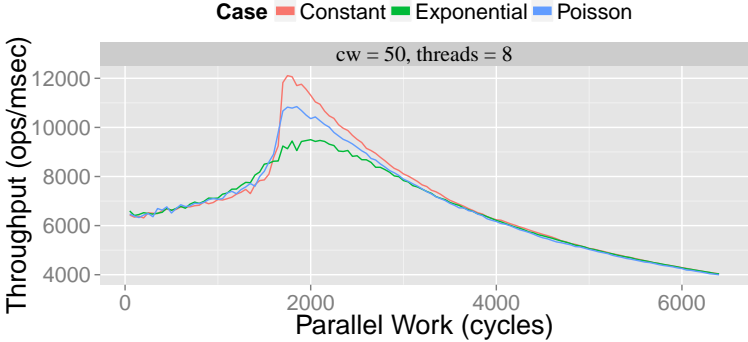
Procedure AbstractAlgorithm

```
1 Initialization();
2 while ! done do
3   Parallel_Work();           /* Application specific code, conflict-free */
4   while ! success do
5     current ← Read(AP);
6     new ← Critical_Work(current);
7     success ← CAS(AP, current, new);
```

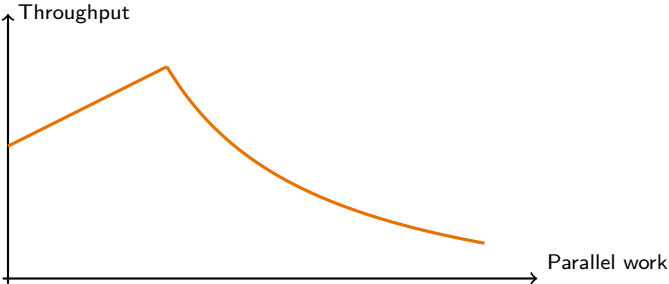
Inputs of the analysis:

- ▶ Platform parameters: CAS and Read Latencies, in clock cycles
- ▶ Algorithm parameters:
 - ▶ Critical Work and Parallel Work Latencies, in clock cycles
 - ▶ Total number of threads

Overview

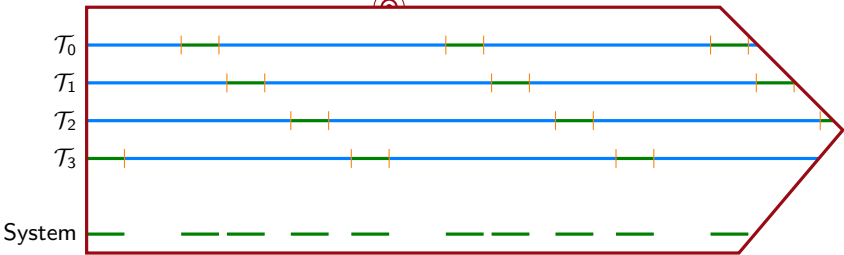
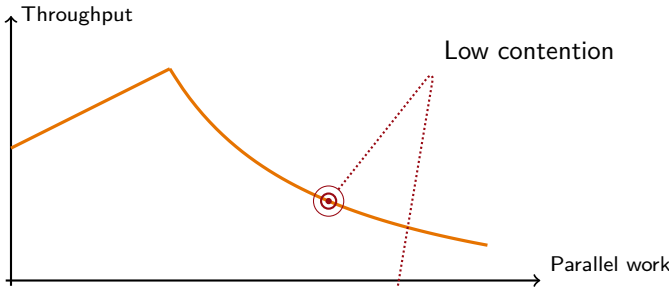


Executions Under Contention Levels



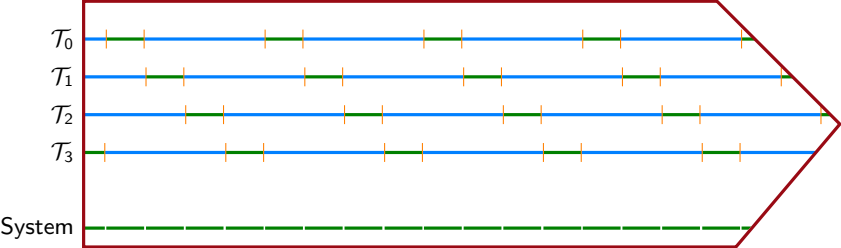
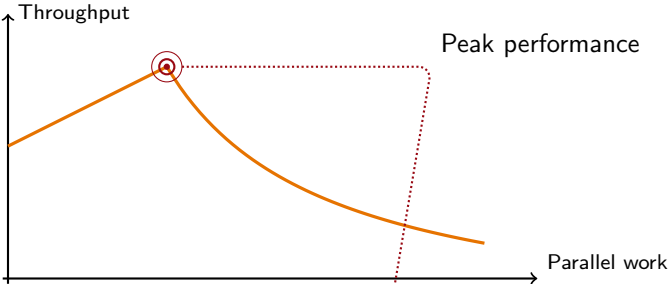
Executions Under Contention Levels

- parallel work
- successful retry
- failed retry



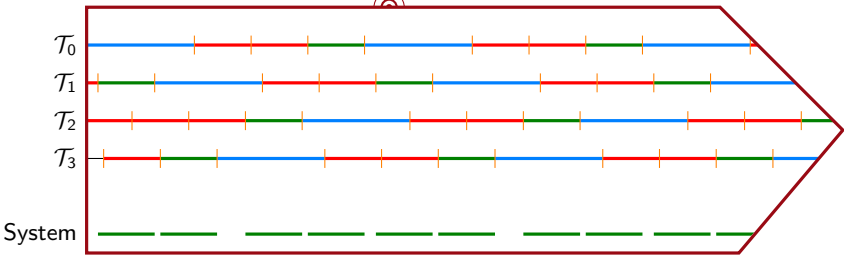
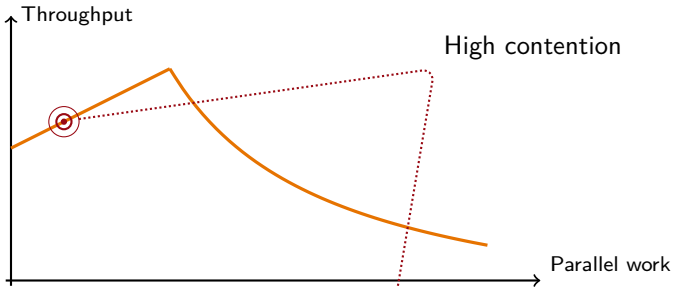
Executions Under Contention Levels

- parallel work
- successful retry
- failed retry



Executions Under Contention Levels

- parallel work
- successful retry
- failed retry

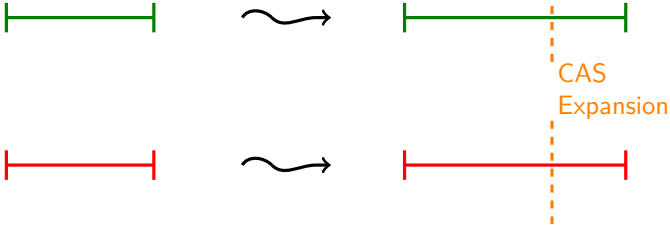


Impacting Factors

▶ Logical Conflicts



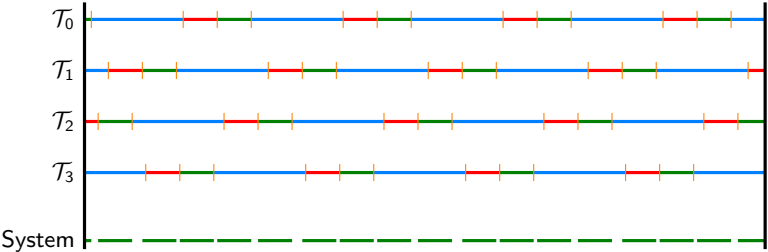
▶ Hardware Conflicts



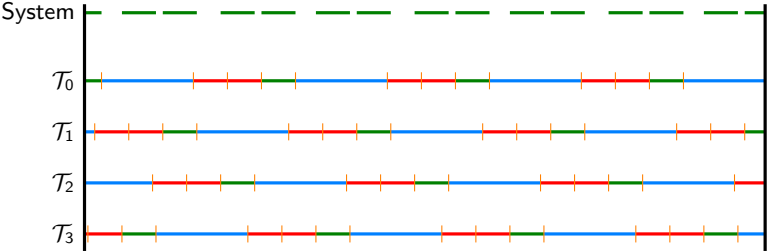
Logical Conflicts: (f) -Cyclic Executions

- ▶ 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

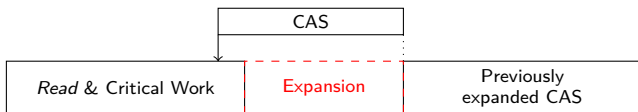
Inevitable and Wasted Failures



vs.



Hardware Conflicts: CAS Expansion



- ▶ Input: P_{rl} threads already in the retry loop
- ▶ A new thread attempts to CAS during the retry (Read + Critical_Work + $e(P_{rl})$ + CAS), within a probability h :

$$\rightsquigarrow e(P_{rl} + h) = e(P_{rl}) + h \times \int_0^{retry} \frac{cost(t)}{retry} dt.$$

Throughput: Combining Impacting Factors

► Input: P_{rl} (Average number of threads inside retry loop)

1. Calculate expansion: $e(P_{rl})$

2. Compute amount of work in a retry:

$$Retry = Read + Critical_Work + e(P_{rl}) + CAS$$

3. Estimate number of logical conflicts:

$$LogicalConflicts(Retry, Parallel_Work, Threads)$$

↪ 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: $e(P_{rl})$

2. Compute amount of work in a retry:

$$Retry = Read + Critical_Work + e(P_{rl}) + CAS$$

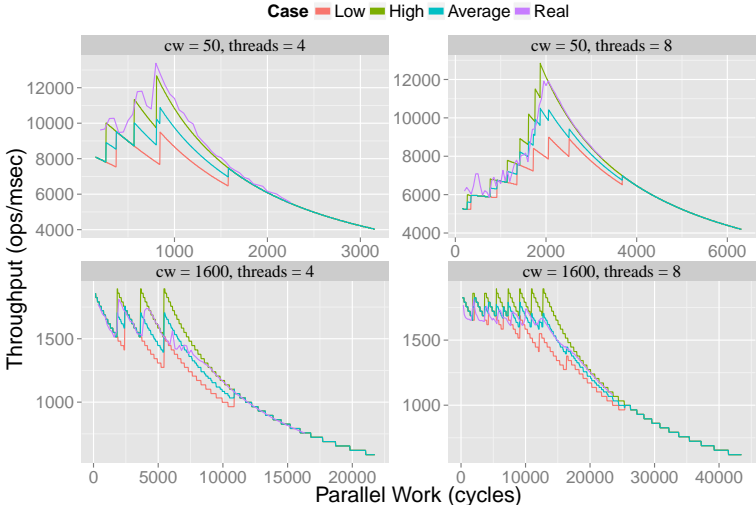
3. Estimate number of logical conflicts:

$$LogicalConflicts(Retry, Parallel_Work, Threads)$$

↪ Average number of threads inside the retry loop

- ▶ Convergence via fixed point iteration

Results: Synthetic Tests



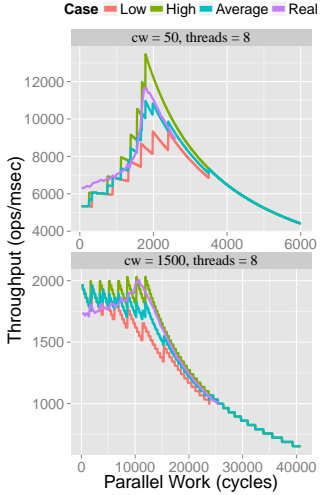
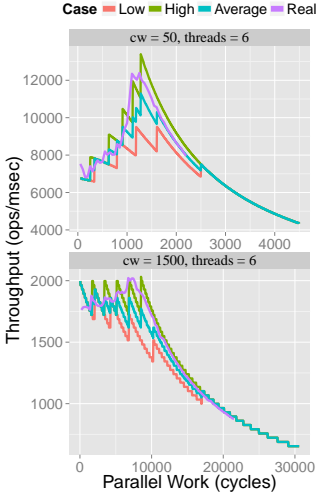
Back-off Optimization: Michael-Scott Queue



Conclusion

- ▶ Focus on the cases where parallel work is constant
- ▶ An approach based on the estimation of logical and hardware conflicts
- ▶ Validate our model using synthetic tests and several reference data structures
- ▶ Linear combination of retry loops

Results: Treiber's Stack



Discussion

