CHALMERS

Throughput Based Energy Efficiency Modeling of Lock-Free Data Structures

Aras Atalar, Anders Gidenstam, Paul Renaud-Goud and Philippas Tsigas

Chalmers University of Technology

Motivation

- Why multi-core:
  - Heat dissipation, memory bottleneck, physical limits
  - Multi-core challenges: Synchronization, load balance, *etc.*

Motivation

- Why multi-core:
  - Heat dissipation, memory bottleneck, physical limits
  - Multi-core challenges: Synchronization, load balance, *etc.*

- Lock-free Data Structures:
  - Lock-Freedom: Non-blocking system-wide progress guarantee
  - Optimistic Conflict Control
  - Limitations of their lock-based counterparts: deadlocks, convoying and programming flexibility
  - High scalability

Motivation

- Why multi-core:
  - Heat dissipation, memory bottleneck, physical limits
  - Multi-core challenges: Synchronization, load balance, *etc.*
- Lock-free Data Structures:
  - Lock-Freedom: Non-blocking system-wide progress guarantee
  - Optimistic Conflict Control
  - Limitations of their lock-based counterparts: deadlocks, convoying and programming flexibility
  - High scalability
- Major optimization criterion (road to Exascale, battery lifetime for embedded systems, *etc.*) decomposed into:
  - Power
  - Throughput (ops/unit of time)

Settings

Output: 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);
```

Settings

Output: 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);
```

Settings

Output: 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);
```

Settings

Output: 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);
```

Settings

Output: 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);
```

Settings

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

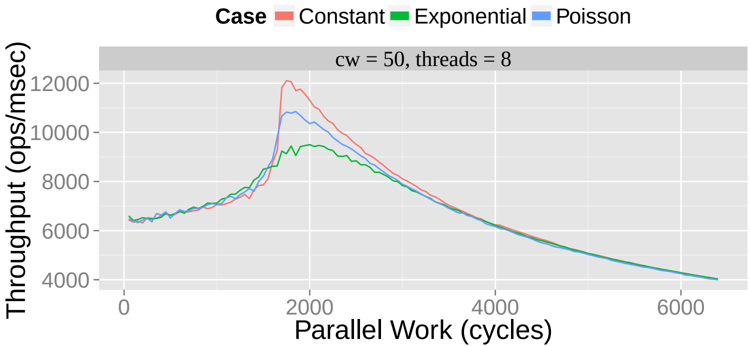
Procedure AbstractAlgorithm

```
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);
```

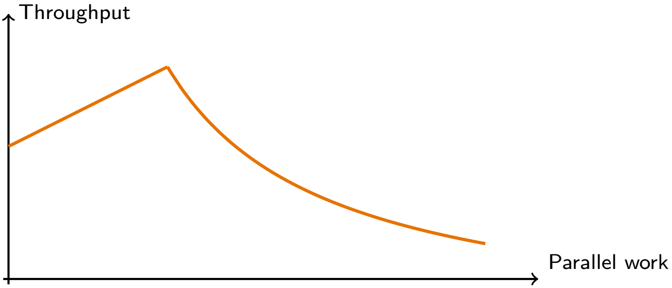
Inputs of the analysis:

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

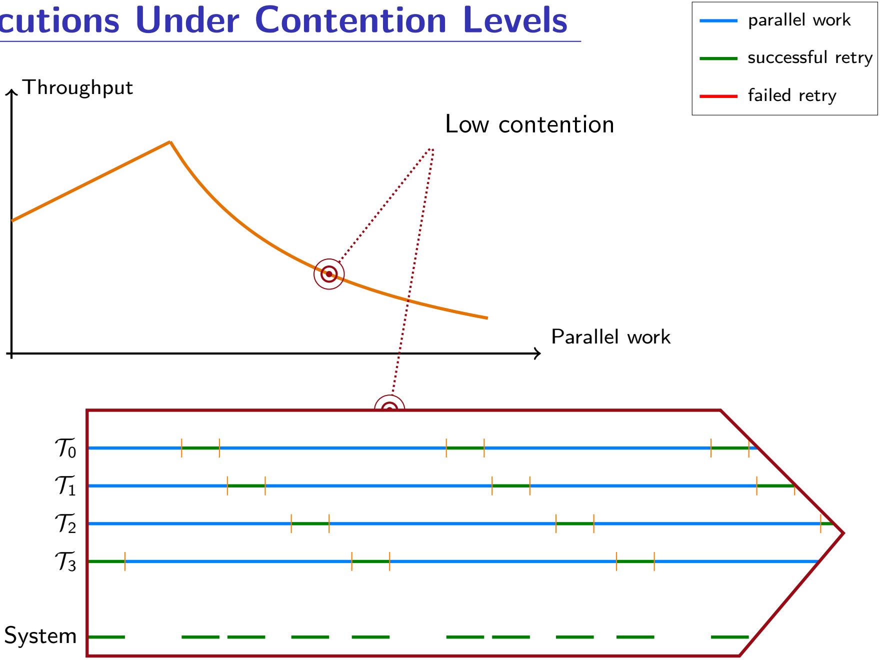
Example: Treiber's Stack Pop operation



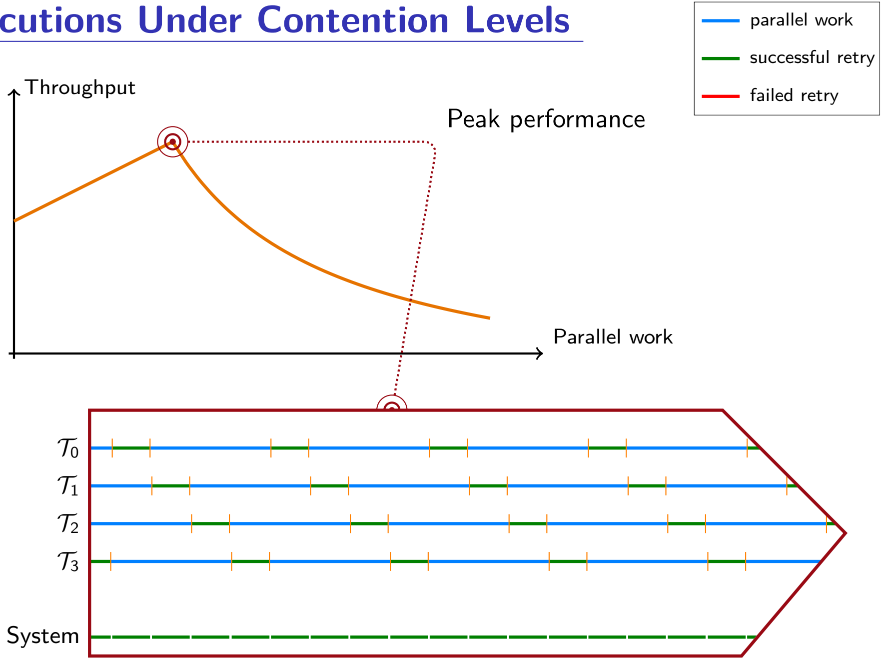
Executions Under Contention Levels



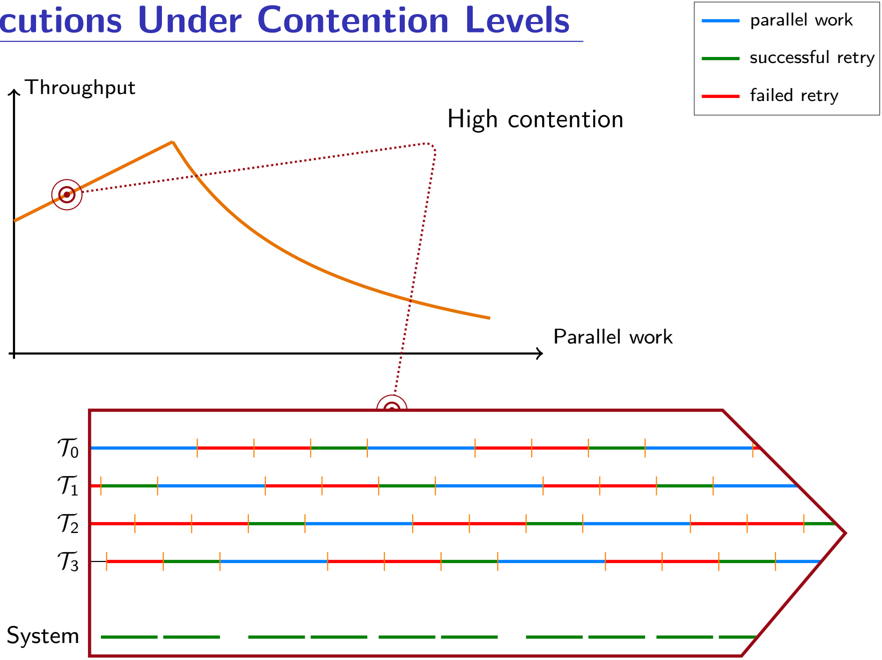
Executions Under Contention Levels



Executions Under Contention Levels



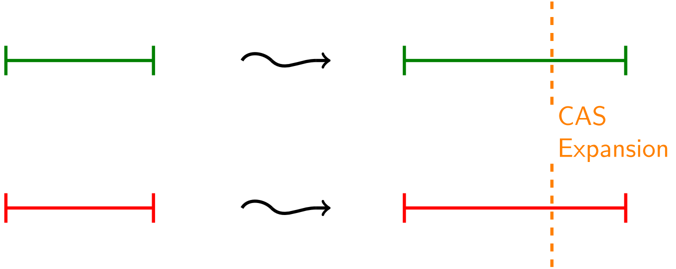
Executions Under Contention Levels



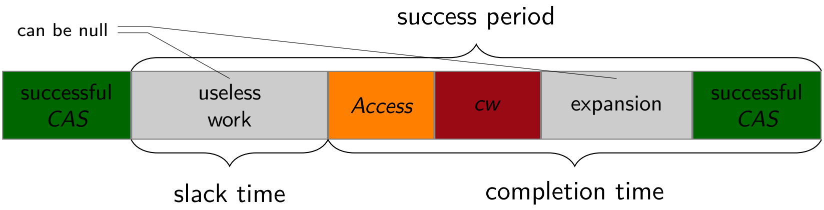
Impacting Factors

- Failed Retries

- Atomic CAS Conflicts



Analyses



- The analyses are centered around a single variable P_{rl}, the number threads inside the retry loop

Average-Based Approach

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

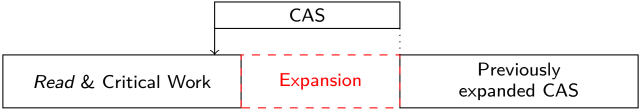
$$\overline{sp}\left(\overline{P_{rl}}\right) = pw/(P - \overline{P_{rl}}) \quad (1)$$

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

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

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

CAS Expansion and Slack Time



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

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

- Assume a thread has equal probability to be anywhere in the retry loop

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

Unified Solving and Throughput Estimate

- Unified Solving:

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

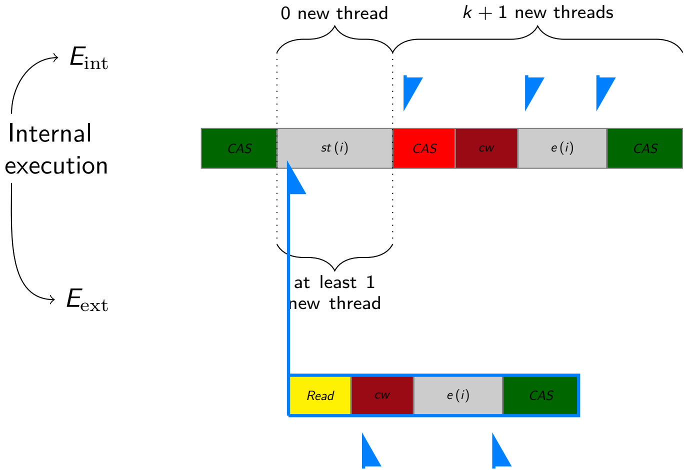
The system switches from being non-contended to being contended at $\overline{P_{rl}} = P_{rl}^{(0)}$, where

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

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

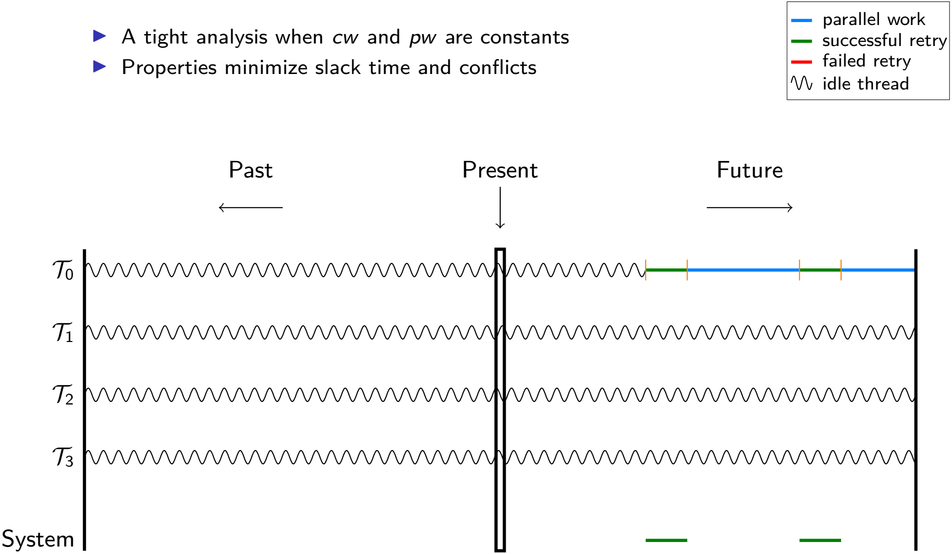
Stochastic Approach

- Analysis based on Markov Chains and stochastic sequence of success periods results in the throughput estimate
- P_{rl}, just after a successful *CAS*, renders the state of the system

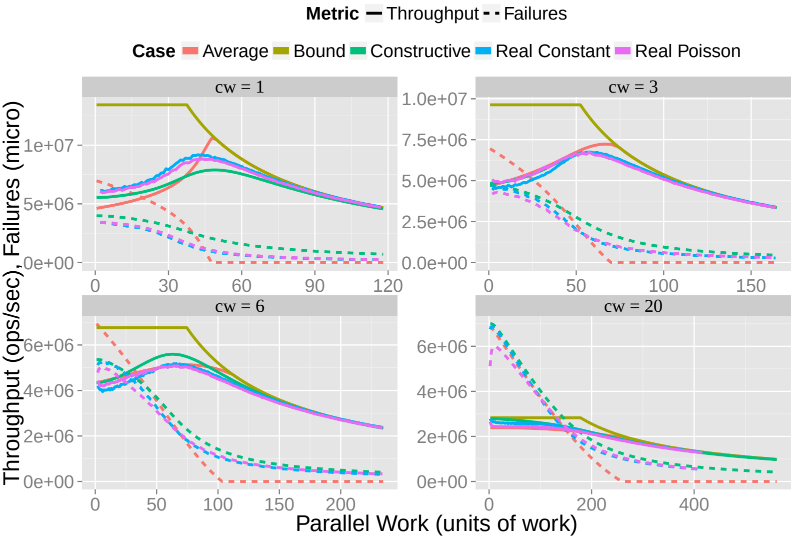


Deterministic Approach

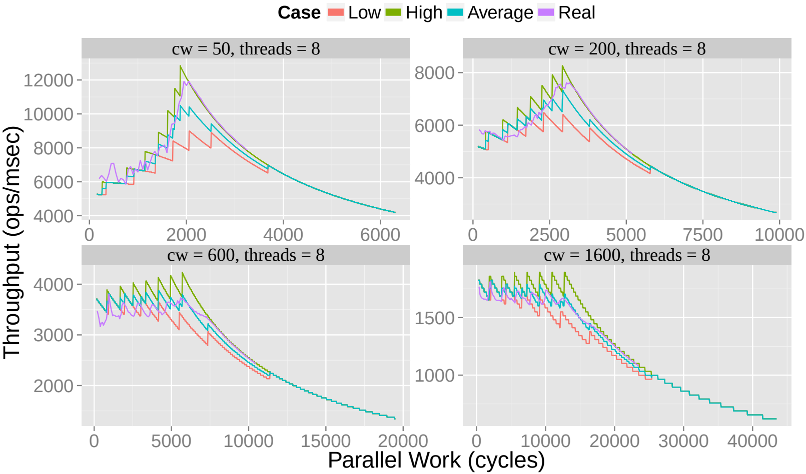
- A tight analysis when *cw* and *pw* are constants
- Properties minimize slack time and conflicts



Throughput Estimation: Synthetic tests



Throughput Estimation: Synthetic tests



Power Estimation

Power split into:

- *Static* part: cost of turning the machine on
- *Activation* part: fixed cost for each socket in use
- *Dynamic* part: supplementary cost depending on the running application

In accordance with the RAPL energy counters, each part further decomposed per-component:

- Memory
- CPU
- *Uncore*

Finally,

$$Pow = \sum_{X \in \{M,C,U\}} \left(Pow^{(stat,X)} + Pow^{(active,X)} + Pow^{(dyn,X)} \right)$$

Power Estimation

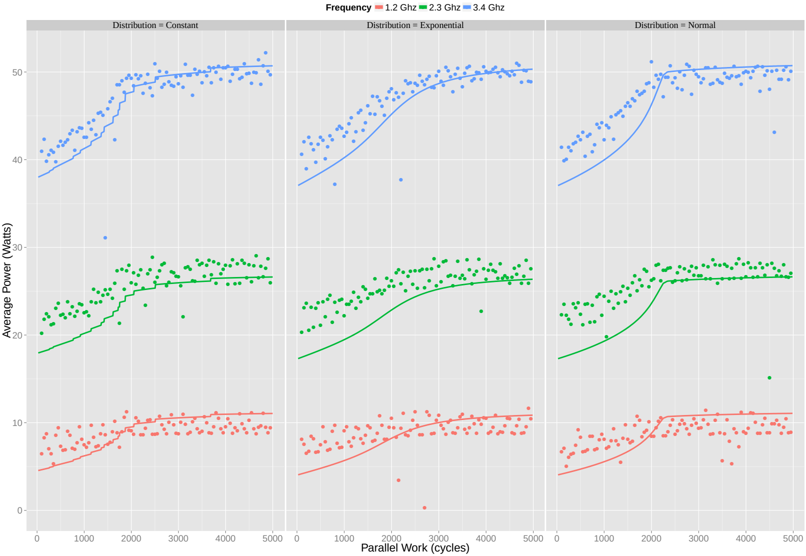
- Dynamic memory and uncore power is proportional to the intensity of main memory accesses and remote accesses
- Each thread mapped on a dedicated core

$$Pow^{(C)}_{total} = Threads \times Pow^{(C)}$$

- Dyn. Cpu Power: IPC (different for the retry loop and parallel work)
- Time segmentation (r: ratio of time spent in retry loop)

$$Pow^{(C)} = r \times Pow^{(C)}_{rl} + (1 - r) \times Pow^{(C)}_{ps}$$

- Two samples are used to obtain $Pow^{(C)}_{rl}$ and $Pow^{(C)}_{ps}$


Frequency 1.2 Ghz 2.3 Ghz 3.4 Ghz
Distribution = Constant
Distribution = Exponential
Distribution = Normal
Average Power (Watts)
0
10
20
30
40
50
0
1000
2000
3000
4000
5000
Parallel Work (cycles)

- Three approaches based on the estimation of success period
- Validate our model using synthetic tests and several reference data structures
- Power Model for CPU platform
- Energy efficiency of lock-free data structures based on the ratio of time spent in retry loops