Multicore Locks: The Case is not Closed Yet

Hugo Guiroux, Renaud Lachaize, Vivien Quéma

June 24, 2016

Université Grenoble Alpes
Grenoble INP
Synchronization on Modern Multicore Machines
Most multi-threaded applications require synchronization.
As the number of cores increases, the synchronization primitives become a bottleneck.
The design of efficient multicore locks is still a hot research topic: (e.g., [ASPLOS’10], [ATC’12], [OLS’12], [PPoPPP’12], [SOSP’13], [OOPSLA’14], [PPoPP’15], [PPoPP’16]).
Lock-based synchronization:

```c
pthread_mutex_lock(&mutex);
// Critical section:
// at most 1 thread here at
// a time
...
pthread_mutex_unlock(&mutex);
```
Pthread Mutex Lock

Lock-based synchronization:

```c
pthread_mutex_lock(&mutex);
// Critical section:
// at most 1 thread here at a time
...
pthread_mutex_unlock(&mutex);
```

Plethora of locking algorithms.

Goals:

- **Performance**
  - Throughput: at high contention
  - Latency: at low contention
- **Fairness**
- **Energy efficiency**
Applications suffer from lock contention

Plethora of locks algorithms

Developers are puzzled:
- Does it really matter for my application/my setup?
- How to choose?
- Will the chosen lock perform reasonably well on most setups?
- Should we simply discard old/simple locks?
Previous studies:
- Are mostly based on microbenchmarks . . .
- . . . or on workloads for which a new lock was specifically designed
- Do not consider state-of-the-art algorithms that are known to perform well (e.g., recent hierarchical locks) or important parameters (e.g., the choice of waiting policy)
1. Extended study:
1. Extended study:

27 locks
1. Extended study:

   27 locks

   39 applications
1. Extended study:

27 locks

39 applications

3 machines
Contributions

1. Extended study:

27 locks

39 applications

3 machines

With/without pinning
Contributions

1. Extended study:

27 locks  
39 applications  
3 machines  
With/without pinning

2. Library for transparent replacement of the lock algorithm
Outline

Locks Algorithms

LiTL: Library for Transparent Lock Interposition

Study of lock/application behavior

Conclusion
Flat Algorithms

- e.g., Spinlock, Backoff
- **Principle:**
  - Loop on a single memory address
  - Use atomic instruction
- **Pros:**
  - Very fast under low lock contention
- **Cons:**
  - Collapse under high contention due to cache coherence traffic
Queue-Based Algorithms

- e.g., MCS, CLH

**Principle:**
- List of waiting threads
- Each thread spins on a private variable

**Pros:**
- Mitigation of cache invalidations
- Fairness

**Cons:**
- Inefficient lock handover if successor has been descheduled
- Memory locality issue (lack of NUMA awareness)
Hierarchical Algorithms

- e.g., Cohort locks, AHMCS
- **Principle:**
  - One local lock per NUMA node + one global lock
  - Per-node batching
- **Pros:**
  - Good behavior on NUMA hierarchies under high contention
- **Cons:**
  - Short-time unfairness
  - High costs under low lock contention
Load-control Algorithms

- e.g., MCS-TimePub, Malthusian locks

**Principle:**
- Bypass threads in the waiting list
- Reduce the number of threads trying to acquire the lock

**Pros:**
- Better resilience under resource contention

**Cons:**
- Fairness
Delegation-Based Algorithms

- e.g., RCL, CC-Synch
- **Principle:**
  - One thread executes the critical section on behalf of the others
  - Not general purpose, designed for highly contended locks
- *Not considered here:*
  - Need to rewrite the code application
  - Does not support thread-local data, nested locking, ...
Waiting Policy

- Another design dimension (for most locks)
- What should a thread do while waiting for a lock?
  - Park: sleep (default Pthread policy)
  - Spin: busy-wait (active)
  - Spin-Then-Park: spin a little, then go to sleep
LiTL: Library for Transparent Lock Interposition
LiTL: Overview

• Motivation
  • Implementing all existing locks into all applications is laborious
  • No existing library to try a lock implementation easily

• LiTL: lock library on top of Pthread Mutex lock API
  • Support unmodified application via library interposition
  • Supports condition variables
  • Supports nested critical sections
  • 27 locks (easy to add new ones)

https://github.com/multicore-locks/litl
LiTL Design Challenges: Lock Context

- Many lock algorithms rely on “contexts”
  - The Pthread Mutex lock API does not consider contexts

- Solution:
  - Each lock instance comes with an array of contexts, with one entry per thread to support nested critical sections
  - Pthread Mutex lock → custom lock via hash table (CLHT)
LiTL Design Challenges: Supporting Condition Variables

- Approach: reuse Pthread Condition variable
  1. Take an uncontended Pthread lock with the optimized lock
  2. Use the Pthread lock on cond_wait (paper)
Overhead Evaluation

- Comparison with manual implementation of all locks on 3 lock-intensive applications
- General trends are preserved
- Average performance difference is below 5%
Study of lock/application behavior
Methodology

- 5% tolerance margin to take into account deviation
  - Optimal lock: best or at most 5% of performance degradation of the best
Methodology

- 5% tolerance margin to take into account deviation
  - Optimal lock: best or at most 5% of performance degradation of the best
- Linux (3.17.6)
- 3 machines
  - A-64: AMD 64 cores, 8 nodes
  - A-48: AMD 48 cores, 8 nodes
  - I-48: Intel 48 cores, 4 nodes (no hyperthreading)
- Most results presented here are from the A-64 machine
- We vary the number of threads used to launch the applications.
Lock-Sensitive Applications

- 60% of the studied applications are lock sensitive

Locks impact application performance
We consider 2 configurations per application:

- Maximum number of nodes: use all cores of the machine
- Optimized number of nodes: take the number of nodes for a given lock/application maximizing performance
  - not all locks have the same optimized number of nodes
  - avoid performance collapse
  - take the best of each lock

Number of nodes impacts lock performance
How Much do Locks Impact Applications?

- At 1 node, **reduced impact**
  - From 2% to 683%:
    - avg. 4%, med. 7%

- At max nodes, **huge impact**
  - From 42% to 3343%:
    - avg. 727%, med. 91%

- At opt nodes, **significant impact**
  - From 6% to 683%:
    - avg. 105%, med. 17%
Are Some Locks Always Among the Best? I

Fraction of lock-sensitive applications for which a given lock is optimal

At 1 node, no always-winning lock
80% coverage
Are Some Locks Always Among the Best? II

At max and opt nodes, even worse
52% coverage
Are All Locks Potentially Harmful?

Fraction of lock-sensitive applications for which a given lock degrades the performance w.r.t. the best lock (by at least 15%)

Always several applications for which a given lock hurts performance
The lock hierarchy for an application strongly changes with:

- **The number of nodes:**
  - On average, only 27% of the pairwise comparisons are conserved

- **The machine:**
  - On average, only 30% of the pairwise comparisons are conserved
• Using thread pinning does not change the general observations

• Pthread Mutex locks perform relatively well (i.e., are among the best locks) for a significant share of the studied applications
Conclusion
Summary of Observations

- 60% of the studied applications are lock sensitive
- Lock behavior is strongly impacted by the number of nodes
- Locks impact applications both at max and opt nodes
- No lock is always among the best
- There is no stable hierarchy between locks
  - The number of threads impacts the lock hierarchy
  - The machine impacts the lock hierarchy
- All locks are potentially harmful
- Using thread pinning leads to the same conclusions
- Pthreads locks perform reasonably well
• Lock algorithms should not be hardwired into the code of applications.
• The observed trends call for further research on
  • new lock algorithms
  • runtime support for
    • parallel performance
    • contention management

Extended version of the paper + Data Sets + Source Code
https://github.com/multicore-locks/litl/
Conclusion

- Lock algorithms should not be hardwired into the code of applications.
- The observed trends call for further research on
  - new lock algorithms
  - runtime support for
    - parallel performance
    - contention management

Extended version of the paper + Data Sets + Source Code
https://github.com/multicore-locks/litl/

Thank you for your attention.