Understanding the performance of mutual exclusion algorithms on modern multicore machines

Ph.D. thesis defense, Université Grenoble Alpes

Hugo Guiroux

Supervised by:
Prof. Vivien Quéma
Dr. Renaud Lachaize

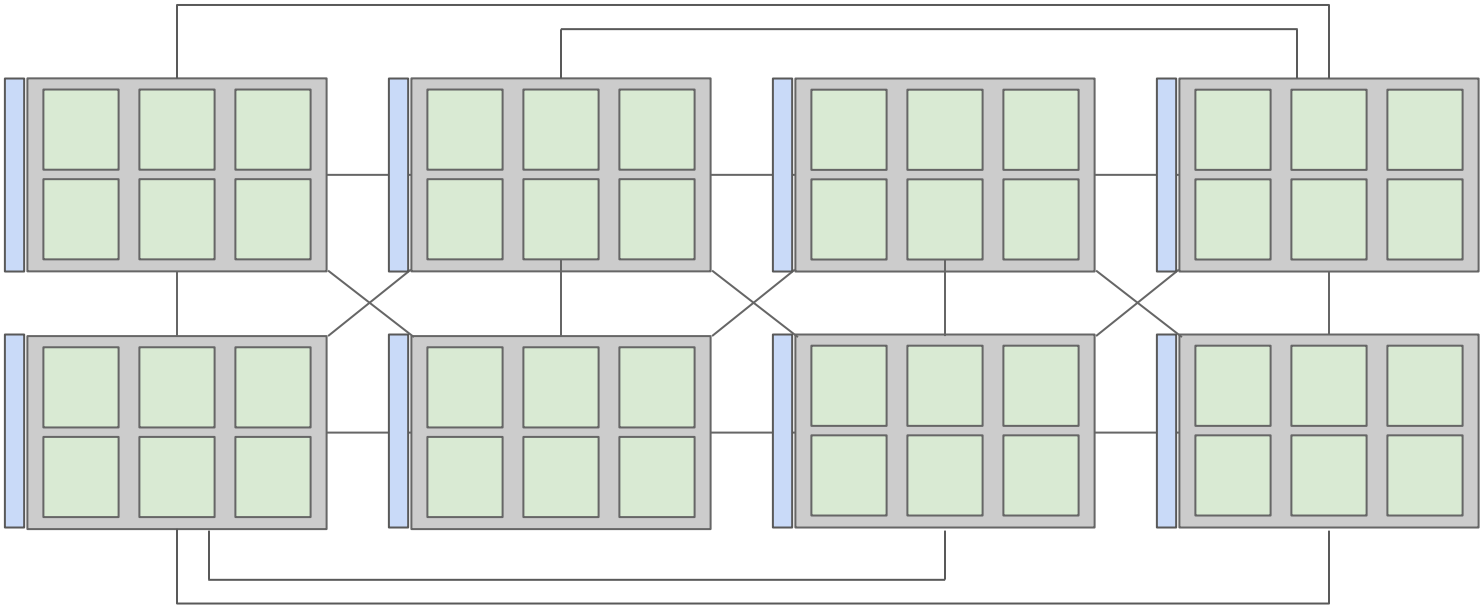
Outline

1. Context
2. Background and Related Work
3. LiTL
4. Study
5. Lessons Learned and Future Research

Context

Modern multicore machines

- More and more cores per machine
- A modern multicore machine is a “distributed computer” with a single coherent shared memory (with NUMA: Non Uniform Memory Accesses)

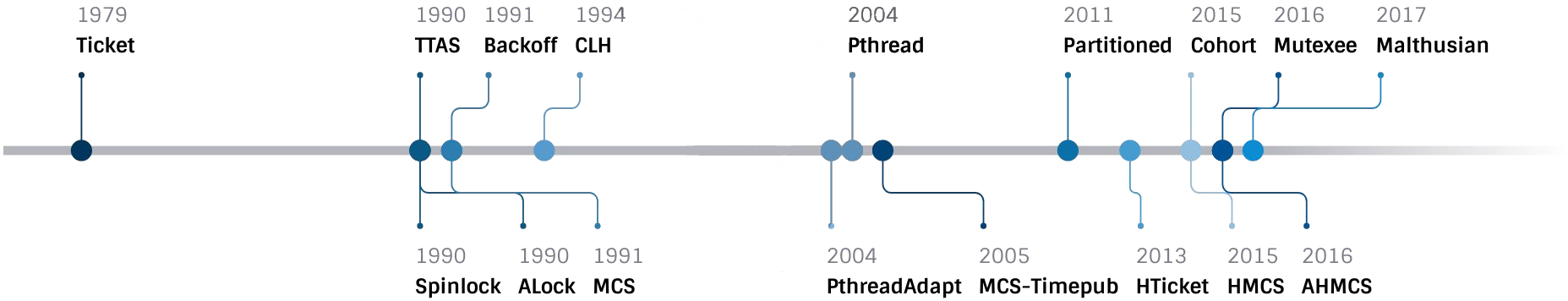


Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize
- Using mutual exclusion locks is the most popular synchronization technique
- Locks introduce sequential parts
- The scalability of an application is always constrained by its sequential parts (Amdahl’s law)

Thesis statement

- Many lock algorithms, still an active field of research



- Limitations of existing studies:
  - Microbenchmarks, limited number of algorithms/applications/workloads, focus only on throughput...

Thesis contribution

Thorough and practical analysis of **mutual exclusion lock algorithms**,
with the goal of
providing software developers with enough information
to **choose** and **design**
fast, **scalable** and **energy-efficient** synchronization in their systems

Background & Related Work

Locking 101

- Lock/unlock
  - Protect a critical section (CS), only one thread can *hold* the lock, others *block* waiting for it
- Trylock
  - Try to acquire the lock and if not available, execute other work instead of blocking
- Condition variables
  - Allow a thread to wait for a software-level condition while inside the critical section, by temporarily releasing the lock and acquiring it again when the thread has been notified that the condition is met

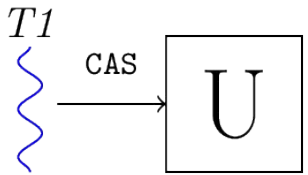
Deconstructing a lock: three main design questions

- How to acquire the lock?
  - (Almost) always use atomic processor instructions to ensure the atomicity of the CS
- How to release the lock?
  - When there are other threads waiting for it (which successor)?
  - Where there is no thread waiting for it
- What to do while waiting for a lock already held?
  - Often orthogonal to the choice with respect to lock acquisition

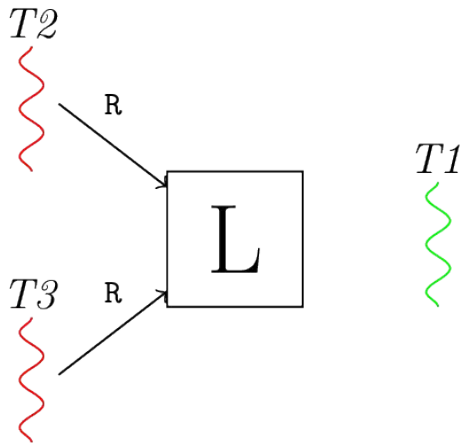
Waiting policy

- Spinning
  - Wait actively for the lock
- Immediate parking
  - Thread is descheduled until the lock is available
- Hybrid approaches
  - Spin-then-park (mitigate the cost of parking)

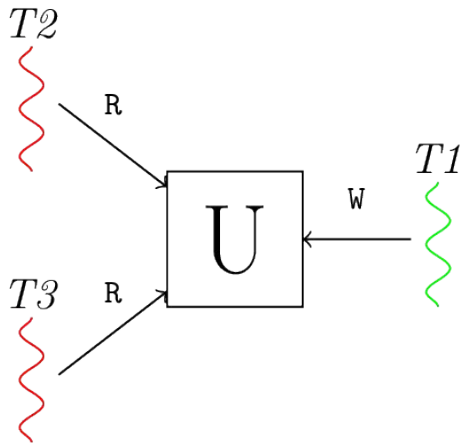
TTAS lock algorithm (*Test-and-Test-And-Set)*



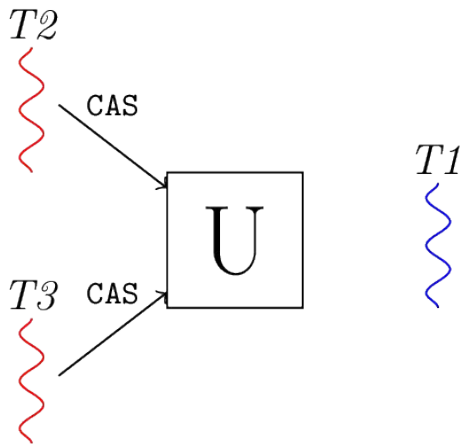
TTAS lock algorithm



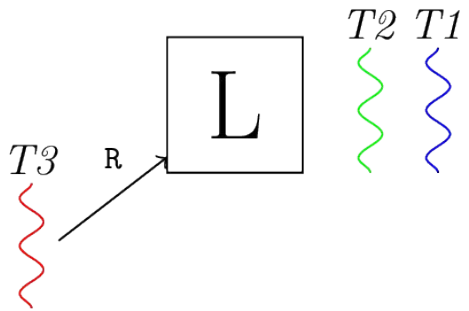
TTAS lock algorithm



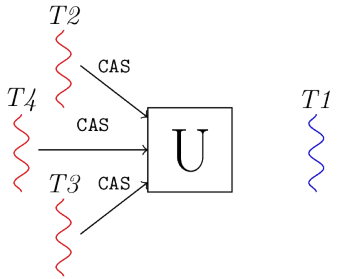
TTAS lock algorithm



TTAS lock algorithm

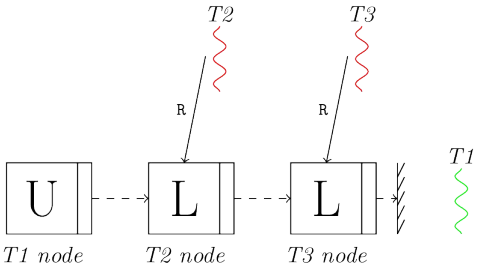


Categorizing lock algorithms



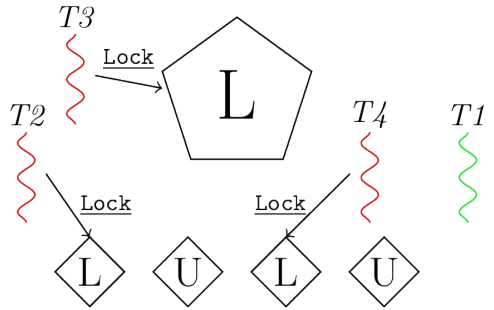
- Competitive succession
  - At unlock time, all waiting threads try to acquire the lock concurrently, with atomic instructions
- Direct handoff succession
  - Waiting threads are ordered, give the lock to a specific successor
- Hierarchical approaches
  - Hierarchical scheme of lock acquisition, minimize lock migrations between NUMA nodes
- Delegation-based
  - Delegate the execution of the CS to another thread
- Load-control
  - Adapt the lock algorithm to runtime conditions (level of contention, scheduler inefficiencies)

Categorizing lock algorithms



- Competitive succession
  - At unlock time, all waiting threads try to acquire the lock concurrently, with atomic instructions
- Direct handoff succession
  - Waiting threads are ordered, give the lock to a specific successor
- Hierarchical approaches
  - Hierarchical scheme of lock acquisition, minimize lock migrations between NUMA nodes
- Delegation-based
  - Delegate the execution of the CS to another thread
- Load-control
  - Adapt the lock algorithm to runtime conditions (level of contention, scheduler inefficiencies)

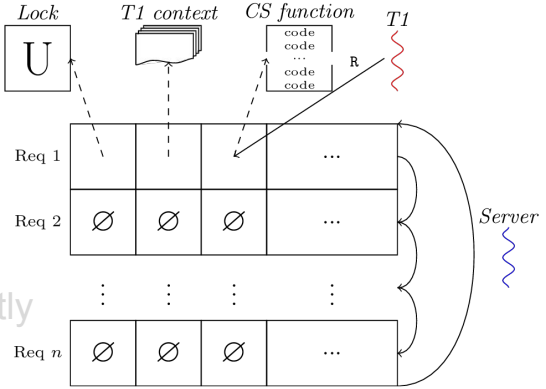
Categorizing lock algorithms



- Competitive succession
  - At unlock time, all waiting threads try to acquire the lock concurrently, with atomic instructions
- Direct handoff succession
  - Waiting threads are ordered, give the lock to a specific successor
- Hierarchical approaches
  - Hierarchical scheme of lock acquisition, minimize lock migrations between NUMA nodes
- Delegation-based
  - Delegate the execution of the CS to another thread
- Load-control
  - Adapt the lock algorithm to runtime conditions (level of contention, scheduler inefficiencies)

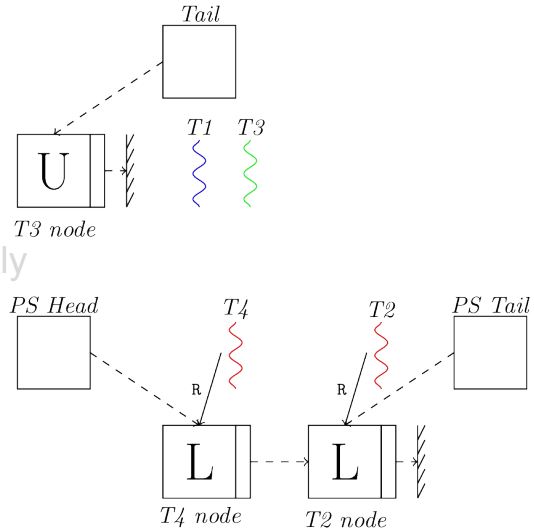
Categorizing lock algorithms

- Competitive succession
  - At unlock time, all waiting threads try to acquire the lock concurrently
- Direct handoff succession
  - Waiting threads are ordered, give the lock at a specific successor
- Hierarchical approaches
  - Hierarchical scheme of lock acquisition, minimize lock migrations between NUMA nodes
- Delegation-based
  - Delegate the execution of the CS to another thread
- Load-control
  - Adapt the lock algorithm to runtime conditions (level of contention, scheduler inefficiencies)



Categorizing lock algorithms

- Competitive succession
  - At unlock time, all waiting threads try to acquire the lock concurrently
- Direct handoff succession
  - Waiting threads are ordered, give the lock to a specific successor
- Hierarchical approaches
  - Hierarchical scheme of lock acquisition, minimize lock migrations between NUMA nodes
- Delegation-based
  - Delegate the execution of the CS to another thread
- Load-control
  - Adapt the lock algorithm to runtime conditions (level of contention, scheduler inefficiencies)



Related work: other studies

- Limited number of lock algorithms
  - Approximately 10 algorithms: [EVERYTHING], [MALTHUSIAN], [RCL]
- Mainly microbenchmarks
  - Test a property of a lock (scalability), but do not capture how the locks *behave* inside the application
  - Often ignore the interactions of locks with scheduling and memory management
- Limited number of applications and metrics
  - Approximately 10 applications, mostly focus on throughput

Objective

- We want to compare many locks on many applications
- Tedious because it requires changing the source code of each application

How to do this practically?

LiTL

Library for Transparent Lock interposition

General principles

- Almost all the application use the pthread_mutex_(un)lock functions
  - Use a dynamic library to provide different implementations of the locking-related functions

- Advantages:
  - Testing a new lock is very easy: switching between lock algorithms by loading a specific dynamic library (e.g., via LD_PRELOAD)
  - Stacking libraries is possible: e.g., for collecting lock statistics of any lock implementation (hold time, acquisition throughput, etc.)
  - Providing condition variables to workload-specific locks

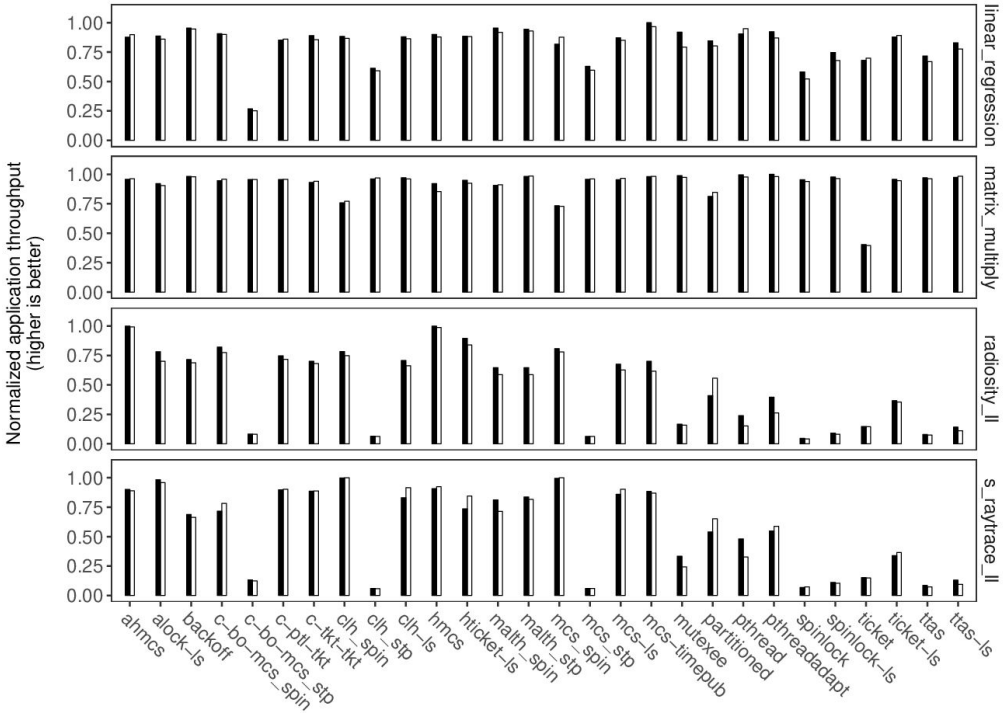
Challenges

- Different lock semantics
  - Supporting per-thread contexts
    - Lock algorithms like MCS require a per-thread context
    - The Pthread locking API does not support this
    - Use an array of per-thread contexts
  - Linking the original lock instance with the optimized lock instance
    - Hashmap between pthread mutex pointer and lock implementation

- Condition variables
  - Used by many applications, ignored by lock designers
  - Reuse the Pthread condition variables implementation by acquiring an (almost always) uncontended Pthread lock

Validation

- Modify the source code of 4 applications, compare manual vs LiTL



Study

Methodology

- 4 machines
  - **A-64**: 64 cores, 4x AMD Opteron 6272 (2011), Bulldozer, 8 NUMA nodes (2-hops)
  - A-48: 48 cores, 4x AMD Opteron 6344 (2012), Piledriver, 6 NUMA nodes (2-hops)
  - I-48: 48 cores, 4x Intel Xeon E7-4830 v3 (2015), Haswell, 4 NUMA nodes (1-hop)
  - I-20: 20 cores, 2x Intel Xeon E5-2680 v2 (2013), Ivy Bridge, 2 NUMA nodes (1-hop)
- 28 locks
- 40 applications

Methodology

- 4 machines
- 28 locks
  - Competitive succession:
    - Backoff, Mutexee, Pthread, PthreadAdapt, Spinlock, Spinlock-ls, TTAS, TTAS-ls
  - Direct handoff:
    - ALock-ls, CLH_Spin, CLH_STP, CLH-ls, MCS_Spin, MCS_STP, MCS-ls, Ticket, Ticket-ls, Partitioned
  - Hierarchical:
    - C-BO-MCS_Spin, C-BO-MCS_STP, C-PTL-TKT, C-TKT-TKT, HTicket-ls, HMCS
  - Load-control:
    - AHMCS, Malth_Spin, Malth_STP, MCS-TimePub
- 40 applications

Methodology

- 4 machines
- 28 locks
- 40 applications
  - Real-world applications (8 workloads)
    - Kyotocabinet, Memcached, MySQL, RocksDB, SQLite, SSL_Proxy, Upscaledb
  - PARSEC 3.0 (14 workloads)
    - Representative mix of emerging multithreaded applications
  - Phoenix 2 (7 workloads)
    - Multicore MapReduce benchmark
  - SPLASH2x (16 workloads)
    - Multithreaded applications

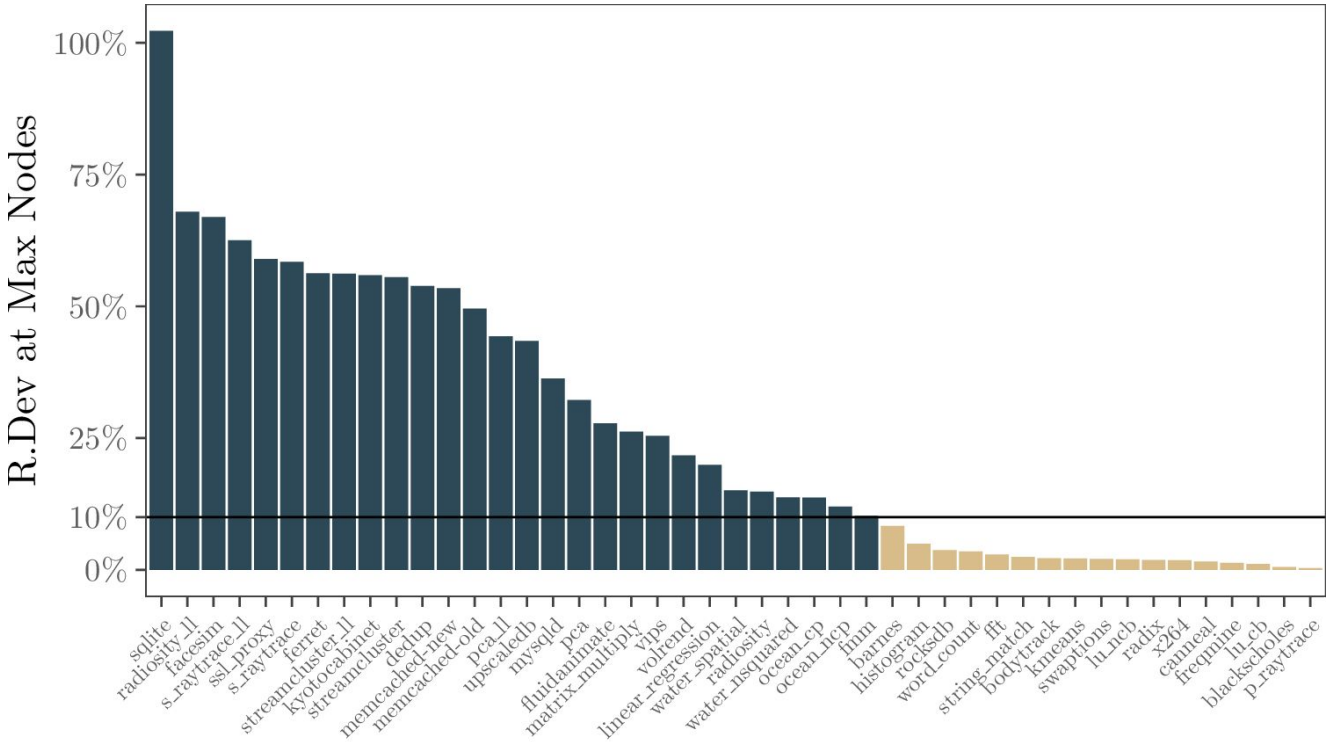
Methodology

- Lock parameters: from original papers
  - *Very few people carefully tune lock algorithm parameters, very specific to the workload*
- Pinning: **no pinning** and thread-to-node pinning
  - *Many applications do not pin threads by default*
- Disk I:O: in memory (tmpfs)
  - *HDD disks, and many applications load inputs from disk to memory*
- Memory: interleaving
  - *Avoid memory contention, often a problem on the A-64/A-48 machines*
- BIOS: no hyperthreading, **performance** and energy-saving modes
  - *Hyperthreading disabled for reproducibility*

- Average of 5 runs + ramp-up period
  - *More runs for configurations with high variability*

Lock-sensitive applications

60% of the studied applications are lock-sensitive



Impact of the number of cores

- The performance of a lock depends on the number of cores
  - At one node (lower contention)
  - At maximum number of nodes (8 on A-64 and A-48, 4 on I-48, 2 on I-20)
  - At optimized number of nodes (**take the best for each lock**)

	A-64	A-48
1 Node	19%	16%
2 Nodes	23%	21%
4 Nodes	26%	23%
6 Nodes	11%	16%
8 Nodes	21%	24%

	I-48
1 Node	37%
2 Nodes	17%
3 Nodes	17%
4 Nodes	29%

	I-20
1 Node	39%
2 Nodes	61%

Breakdown of the *(lock-sensitive application, lock)* pairs according to their optimized number of nodes

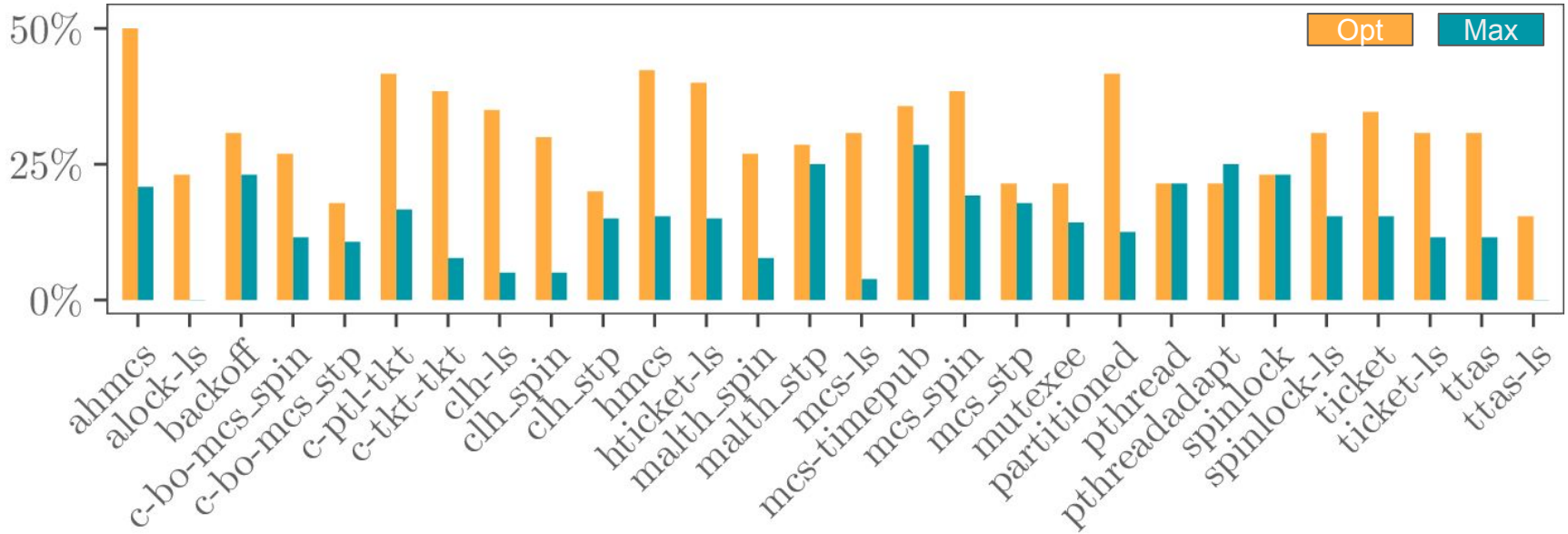
How much do locks impact applications?

	Impact	[Min; Max]	Average	Median	R.Dev
1 Node	Reduced	[1%; 819%]	73%	15%	9%
Max Nodes	Huge	[42%; 2382%]	768%	479%	41%
Opt Nodes	Significant	[5%; 819%]	132%	87%	18%

For lock-sensitive applications, statistics about the throughput of the best vs. worst lock at different numbers of nodes

Are some locks always among the best?

- At max or opt nodes, no lock is the best in more than 50% of the cases



Fraction of lock-sensitive applications where a lock is optimal (the best or within 5% of the best)

Additional observations

- All locks are potentially harmful
  - Any lock will exhibit poor performance
- There is no clear hierarchy between lock algorithms
  - It significantly changes with the application, the machine and the number of nodes
- Impact of thread pinning and BIOS configuration
  - Same observations and conclusions with thread-to-node pinning / with BIOS configured in energy-efficiency mode
- Pthread locks
  - Pthread locks perform reasonably well (i.e., are among the best locks) for many applications

Implications of the study of lock throughput

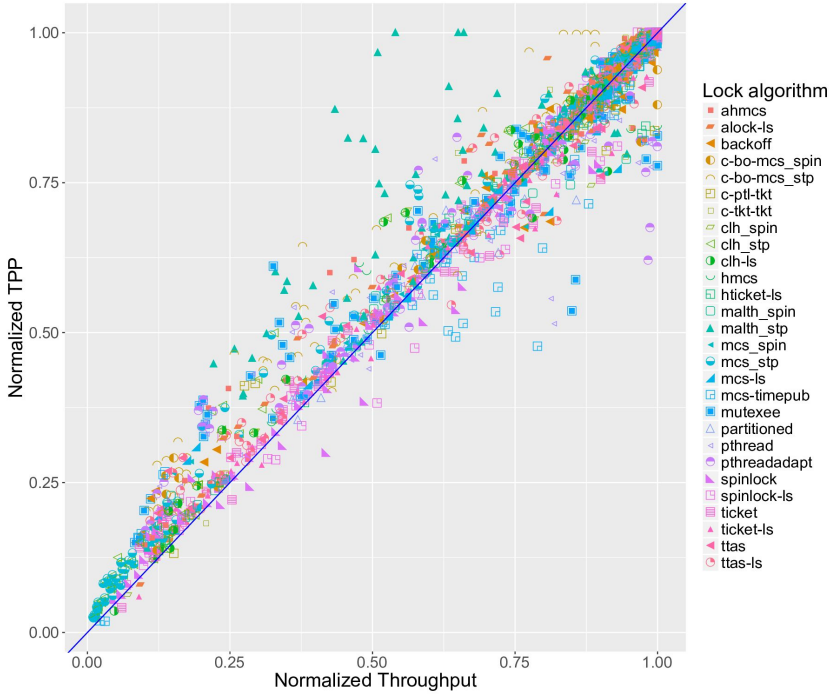
- Do not hardwire the choice of a lock algorithm into an application
  - There is no single best lock
- The Pthread library should provide multiple lock implementations
  - Pthread is not the best for all applications
- Further research on optimized locks is needed, especially on
  - Dynamic approaches
  - Fully supporting the complete locking API

Study of lock energy efficiency

- Energy efficiency = throughput per power (TPP)
  - Amount of work produced for a fixed amount of energy
- Similar conclusions for energy efficiency and throughput
  - No single best lock, all locks are harmful, etc.
- Other observations
  - Almost the same set of lock-sensitive applications for throughput and energy efficiency
  - Under (very) high contention, the energy efficiency gap is higher than the throughput gap

The POLY conjecture

- “Energy efficiency and throughput go hand in hand in the context of locks algorithms” [UNLOCKING]
- Verified that POLY holds for a large number of locks and applications



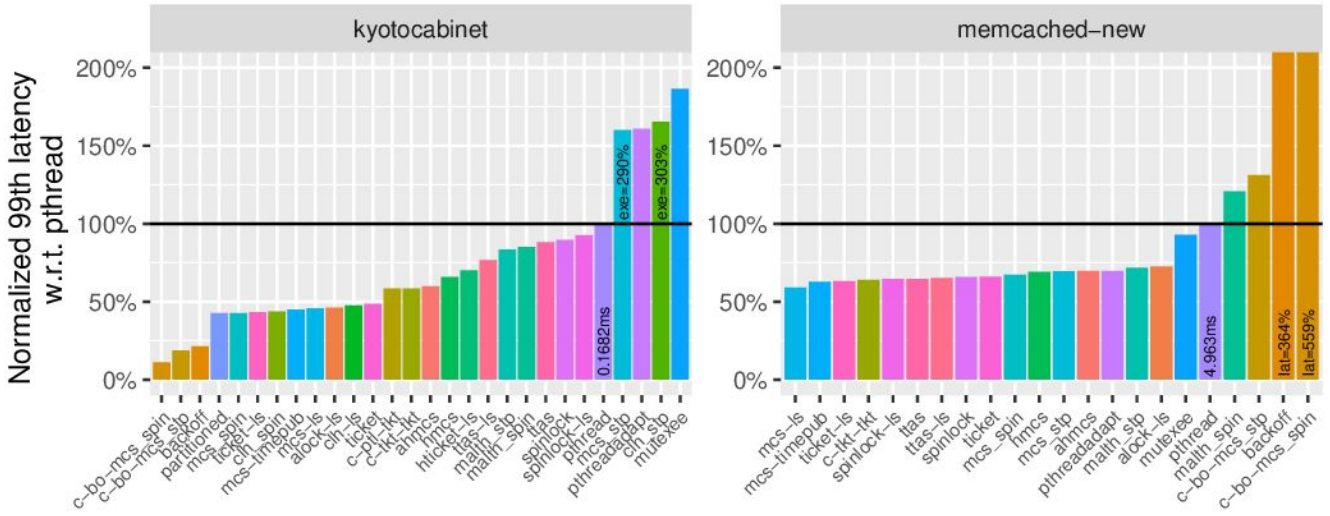
Implications of the study of lock energy efficiency

- Insights from previous throughput-oriented research can be applied almost as-is in the design of energy-efficient locks
  - Lowering the energy consumption at the expense of latency (spin-then-park, DVFS)

- Improving throughput improves energy efficiency and vice-versa
  - The quest for scalable lock algorithms not only benefits throughput but also energy efficiency

Study of lock tail latency

- Tail latency = 99th percentile of the client response time
  - Seven lock-sensitive server applications
- Do fair lock algorithms improve the application tail latency?



Study of lock tail latency

- If an operation is mostly implemented as a single critical section
  - Lock properties affecting lock acquisition tail latency and throughput affect application tail latency and throughput
    - Low tail latency can be achieved with FIFO locks
    - One can trade fairness for throughput by using hierarchical locks for example

- For applications with many critical sections and/or critical sections protected by different lock instances and accessed by different threads
  - The tail latency of the lock does not necessarily affect the application tail latency

Study of lock tail latency

- How does tail latency behave when locks suffer from high levels of contention?
    - Tail latency skyrockets: from one node to max node, average latency increases by 3.3x while tail latency increases by 22.9x (3.4x / 21x from opt to max)
    - In other words, the fairness among threads degrades

Analysis: lock-related bottlenecks

- Lock contention: *multiple threads want to acquire the same lock*
  - High levels of contention: 8 applications
    - 10-40/64 threads waiting for the same lock
  - Extreme levels of contention: 7 applications
    - 40+/64 threads waiting for the same lock
  - Trylock contention: 2 applications
    - Trylock used to implement busy-waiting
  - Many uncontended lock acquisitions: 1 application
    - Importance of the uncontended acquisition code path

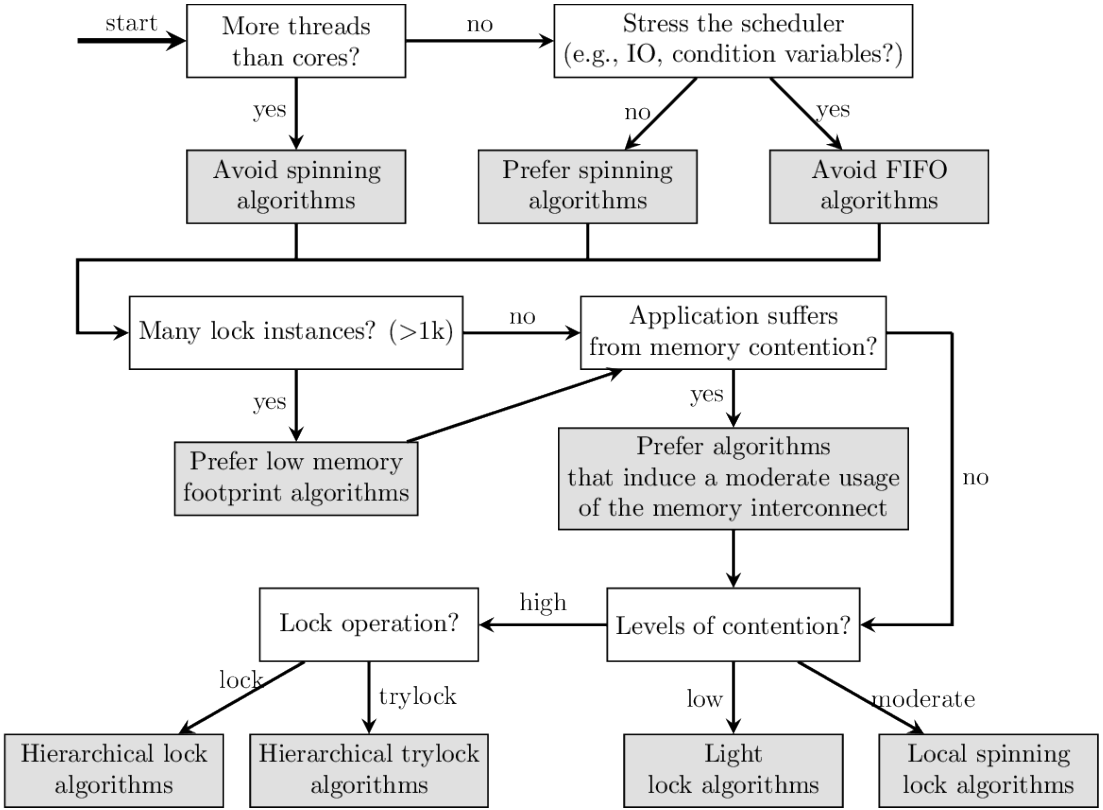
Analysis: lock-related bottlenecks

- Scheduling issues: *the scheduler choices trigger pathological behaviors*
  - Lock holder preemption: 2 applications
    - Preemption of the lock holder, delays the end of the critical section
    - Can lead to lock convoy: all threads eventually try to acquire the lock and delay the lock holder rescheduling
    - Often observed when there are more threads than cores (highly-threaded)
  - Lock handover: 6 applications
    - Happens with locks with a direct-handoff succession policy
    - Descheduling of the next-in-line thread for lock acquisition
    - Also happens when the application is not highly-threaded, as the scheduler migrates threads for other background tasks and/or saving energy

Analysis: lock-related bottlenecks

- Memory footprint: *the size of a lock instance affects performance*
  - Erasing new memory pages inside the page fault handler: 4 applications
    - Allocation of 10k-100k lock instances, the kernel needs to zero memory pages, which takes time (seen at initialization time)
  - Kernel lock contention inside the page fault handler: 1 application
    - Contention inside the kernel with many concurrent memory allocation for lock instances
- Memory contention: *saturation of the memory controller*
  - Locks that are “too” fast increase memory controller saturation: 2 applications

Choice guidelines



Lessons Learned
&
Future Research

Lessons learned

- Locking is not only about lock/unlock, but also trylock and condition variables
- Importance of the OS scheduler decisions
  - Many lock algorithms expect to be alone on the machine and always have 100% of the CPU available for them
- Effect of the memory footprint
  - Complex lock algorithms require more memory (e.g., to store statistics, thread-specific data), which is not cost-free

Future research

- Automatic and dynamic solutions
  - Changing lock algorithms to account for runtime conditions (scheduling, memory, ...)
- Delegation algorithms
  - Better integration with the Pthread locking API
- Lock profiling tools should give a full profile of a lock and how it behaves
  - Interactions with scheduling and memory, other synchronization primitives, access patterns
- Multicore performance
  - Study other performance factors such as the OS scheduler, memory allocation, ...

Publications

- **Multicore Locks: The Case Is Not Closed Yet.**
  Hugo Guiroux, Renaud Lachaize, and Vivien Quéma.
  In Proceedings of the USENIX Annual Technical Conference (USENIX ATC), June 2016.

- **Lock - Unlock: Is That All?**
  Rachid Guerraoui, Hugo Guiroux, Renaud Lachaize, Vivien Quéma, and Vasileios Trigonakis.
  To appear in ACM Transactions on Computer Systems (ACM TOCS), 2019.

Bibliography

- [EVERYTHING] David, Tudor, Rachid Guerraoui, and Vasileios Trigonakis. "Everything you always wanted to know about synchronization but were afraid to ask." In Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles, pp. 33-48. ACM, 2013.

- [MALTHUSIAN] Dice, Dave. "Malthusian locks." In Proceedings of the Twelfth European Conference on Computer Systems, pp. 314-327. ACM, 2017.

- [RCL] Lozi, Jean-Pierre, Florian David, Gaël Thomas, Julia Lawall, and Gilles Muller. "Fast and portable locking for multicore architectures." ACM Transactions on Computer Systems (TOCS) 33, no. 4 (2016): 13.

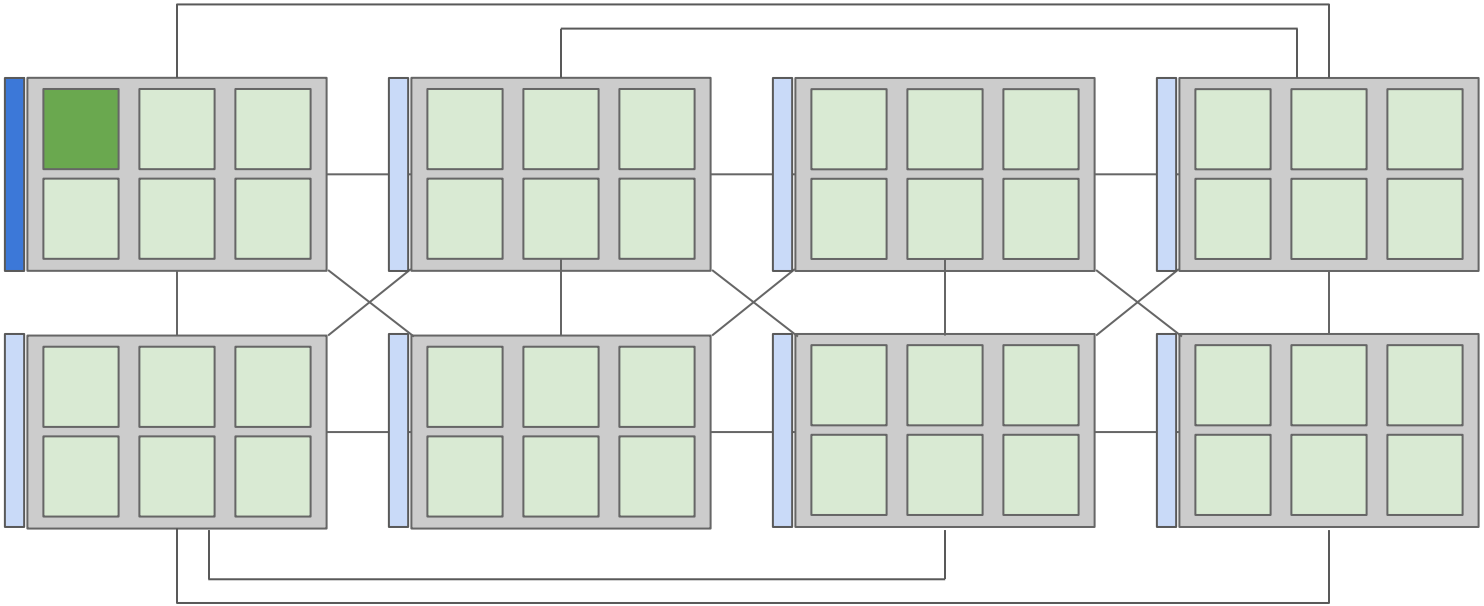
- [UNLOCKING] Falsafi, Babak, Rachid Guerraoui, Javier Picorel, and Vasileios Trigonakis. "Unlocking Energy." In USENIX Annual Technical Conference, pp. 393-406. 2016.

Thank you

https://github.com/multicore-locks/litl

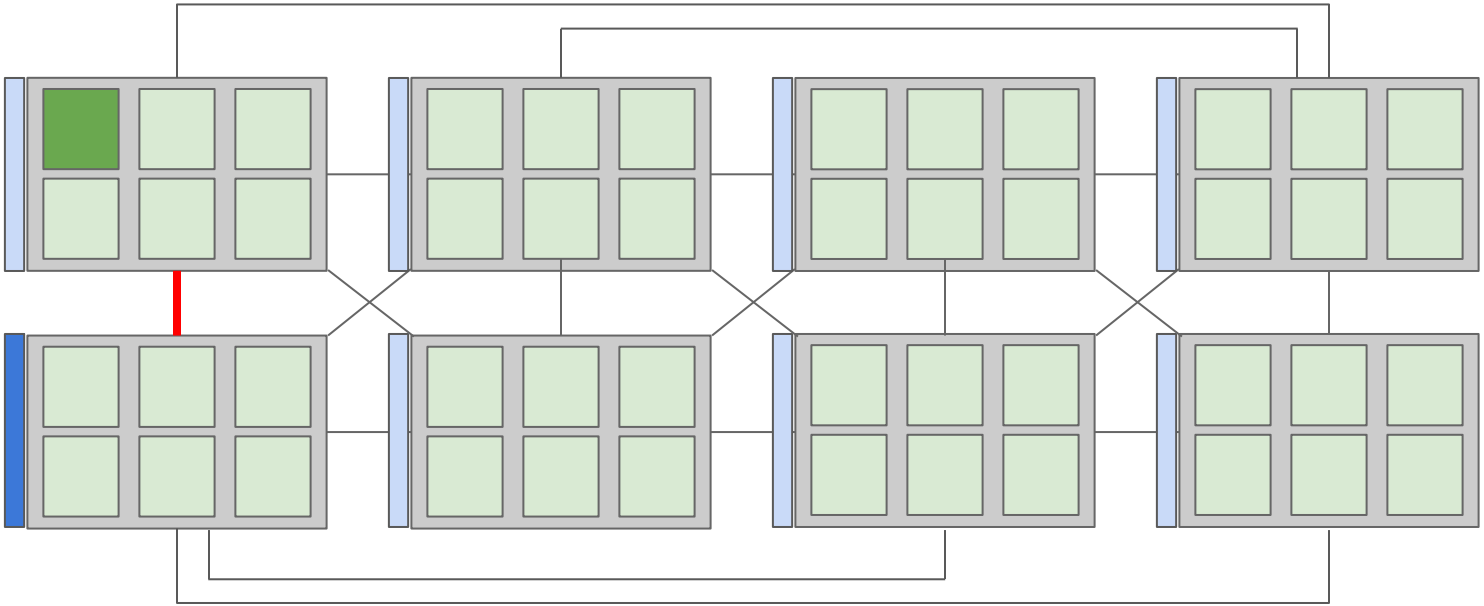
Modern multicore machines

- More and more cores per machine
- A modern multicore machine is a “distributed computer”
- NUMA factor: **Local Access**



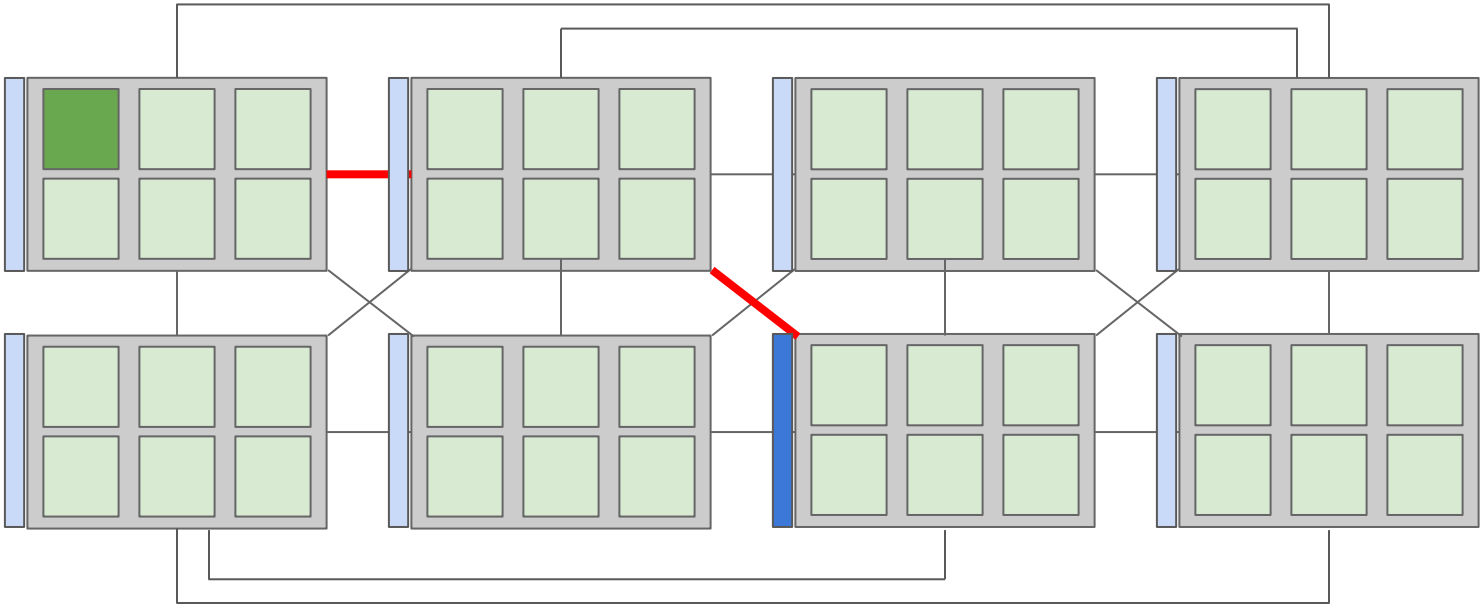
Modern multicore machines

- More and more cores per machine
- A modern multicore machine is a “distributed computer”
- NUMA factor: **1-hop request**



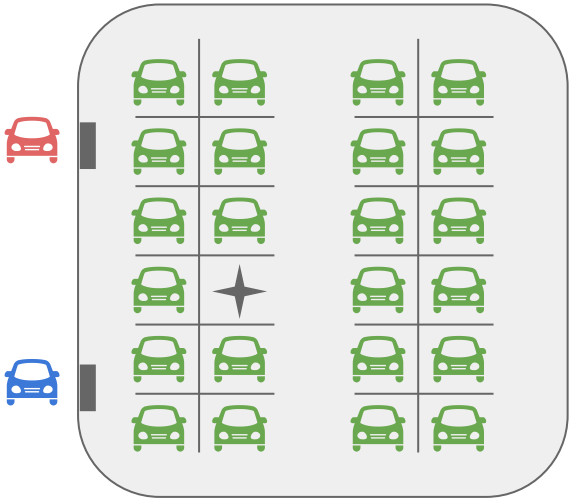
Modern multicore machines

- More and more cores per machine
- A modern multicore machine is a “distributed computer”
- NUMA factor: **2-hops request**



Parallel programming, mutual exclusion and locks

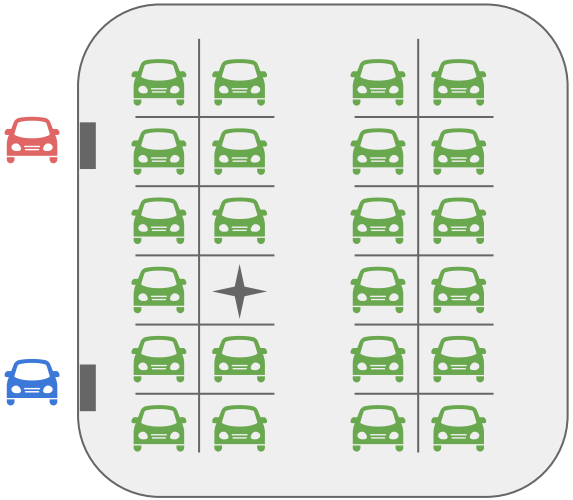
- Because accesses to data can be concurrent, execution contexts need to synchronize



```
bool try_enter_parking():
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        return true;
    } else {
        return false;
    }
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

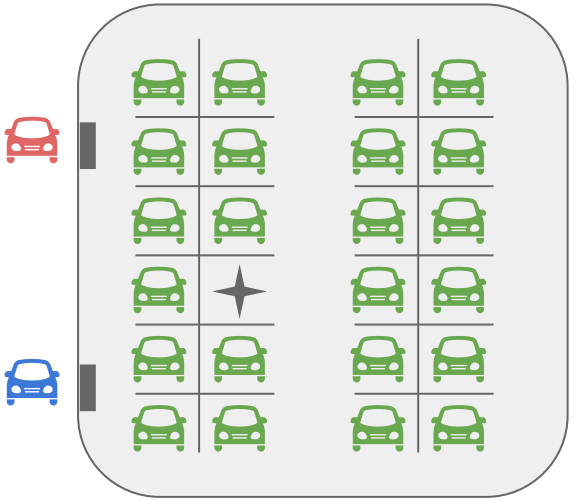


```
bool try_enter_parking():
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        return true;
    } else {
        return false;
    }
}
```

```
bool try_enter_parking():
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        return true;
    } else {
        return false;
    }
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

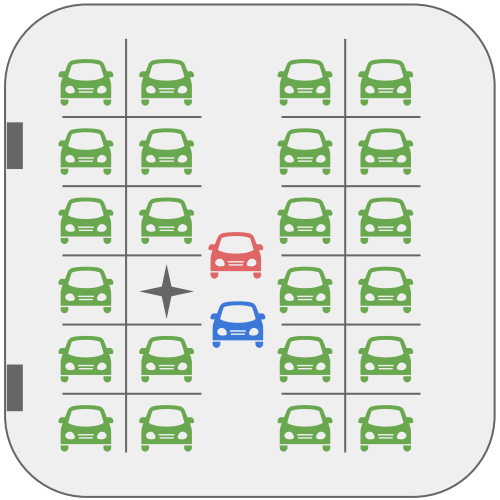


```
bool try_enter_parking():
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        return true;
    } else {
        return false;
    }
}
```

```
bool try_enter_parking():
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        return true;
    } else {
        return false;
    }
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

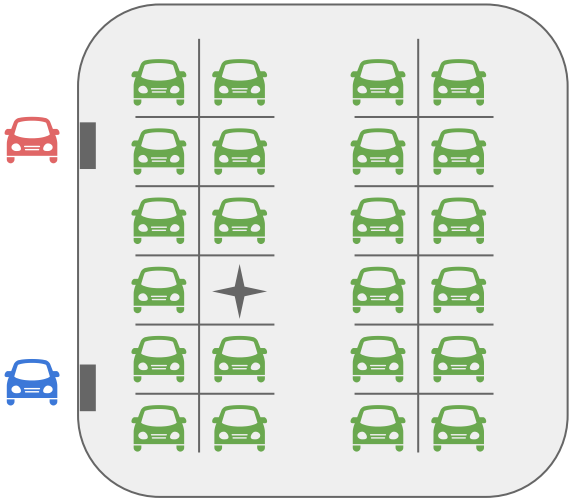


```
bool try_enter_parking():
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        return true;
    } else {
        return false;
    }
}
```

```
bool try_enter_parking():
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        return true;
    } else {
        return false;
    }
}
```

Parallel programming, mutual exclusion and locks

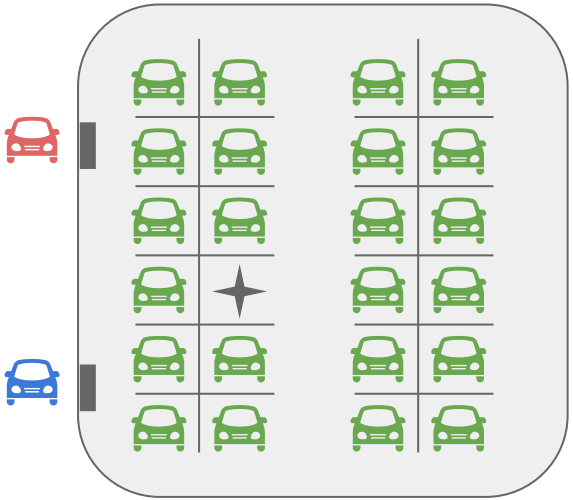
- Because accesses to data can be concurrent, execution contexts need to synchronize



```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

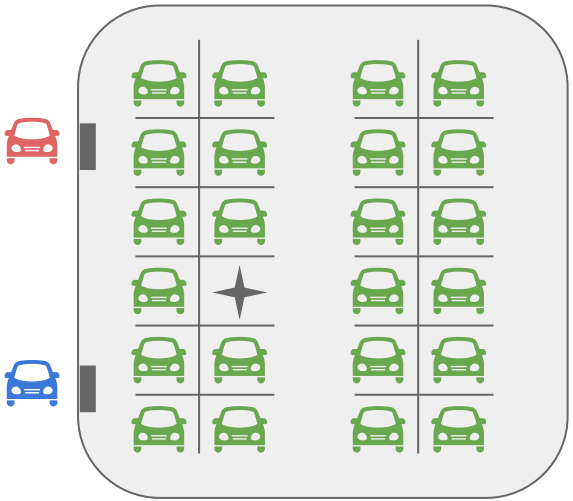


```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

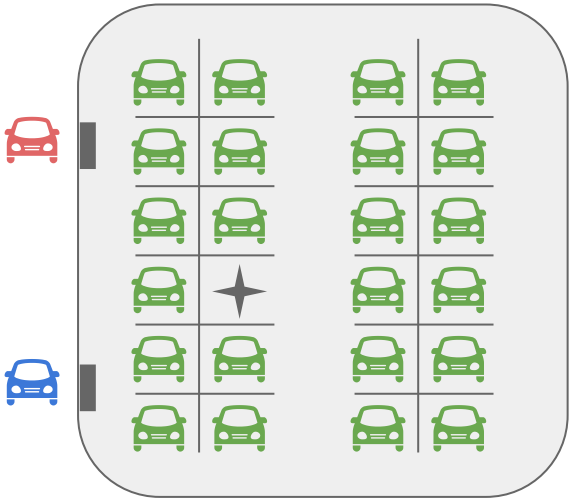


```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

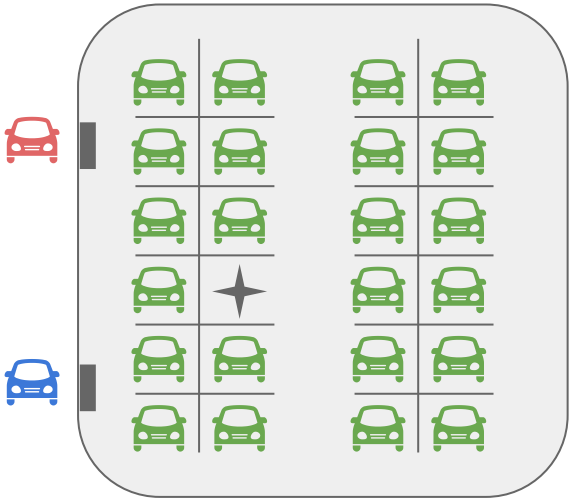


```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

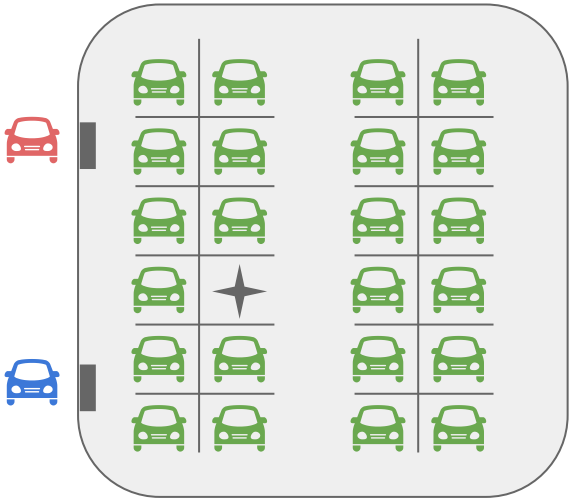


```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

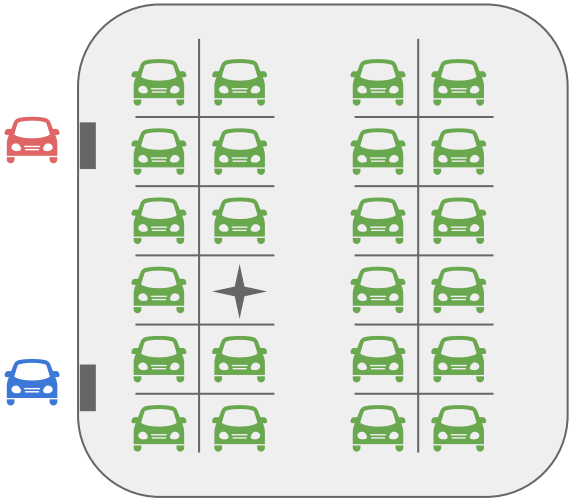


```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

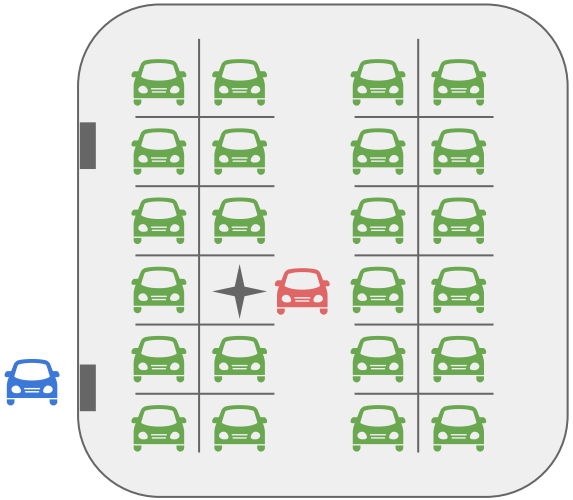


```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize

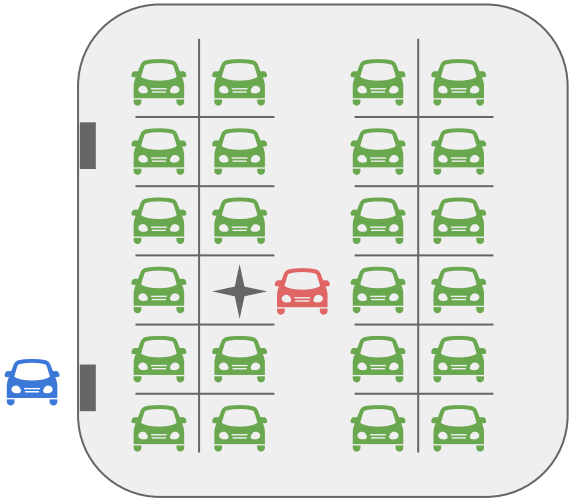


```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Parallel programming, mutual exclusion and locks

- Because accesses to data can be concurrent, execution contexts need to synchronize



```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

```
bool try_enter_parking():
    bool can_enter = false;
    lock();
    if (num_spots_available >= 1) {
        num_spots_available -= 1;
        can_enter = true;
    }
    unlock();
    return can_enter;
}
```

Waiting policy

- Spinning
  - Wait actively for the lock
    - on-cpu: using atomic instructions, memory loads (ttas, backoff)
    - descheduled for a limited amount of time: sched_yield / sleep
    - HW support: lower CPU frequency (DVFS), MONITOR/MWAIT
- Immediate parking
  - Thread is descheduled until the lock is available:
    - Scheduler (OS or runtime) support (futex on Linux for kernel threads)
- Hybrid approaches
  - Spin-then-park (mitigate the cost of parking)
  - Mix of policy (sched_yield and backoff)

Competitive succession

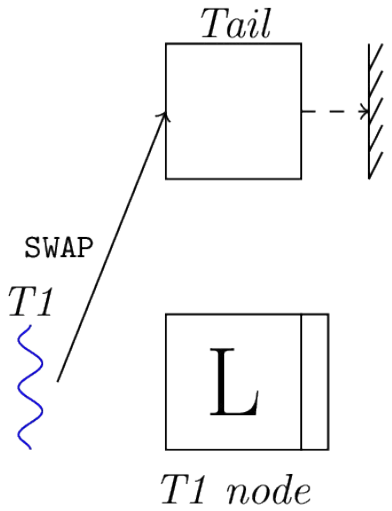
- The lock holder releases the lock, all competing threads tries to acquire it concurrently
- All threads tries to acquire the lock with an atomic instruction, which stress the cache-coherence protocol. But only one succeeds
- Allow barging, which might lead to unfairness and starvation between threads (bad when latency is important)

Direct handoff succession

- The unlock operation identifies a waiting successor and passes the ownership to that thread
  - E.g., MCS construct a linked-list of waiting threads
- Allow each thread to wait on a non-globally shared memory address, avoiding unnecessary cache line invalidations
  - E.g., with MCS, each thread waits on a private variable until it is woken by its predecessor
- Better fairness, and generally better throughput than competitive succession under contention
  - The order of arrival is similar to the order of acquisition, less atomic instructions and cache line transfers

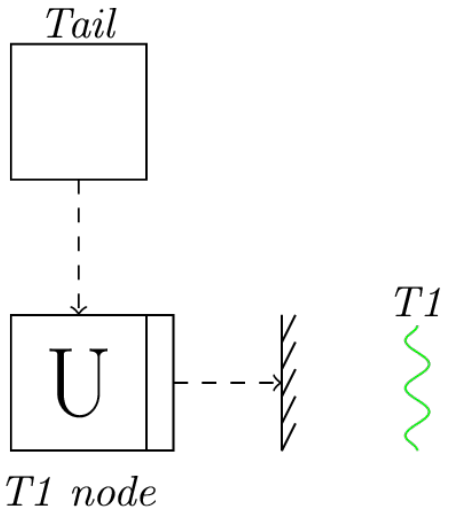
Direct handoff succession

- Example of the MCS lock algorithm



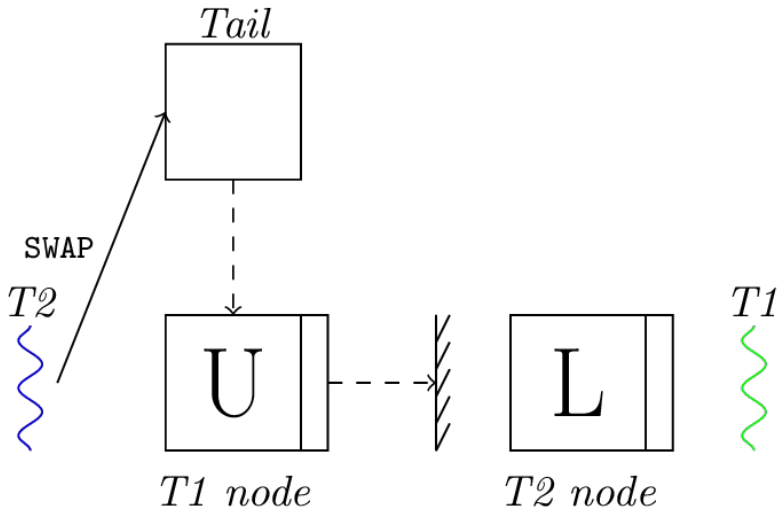
Direct handoff succession

- Example of the MCS lock algorithm



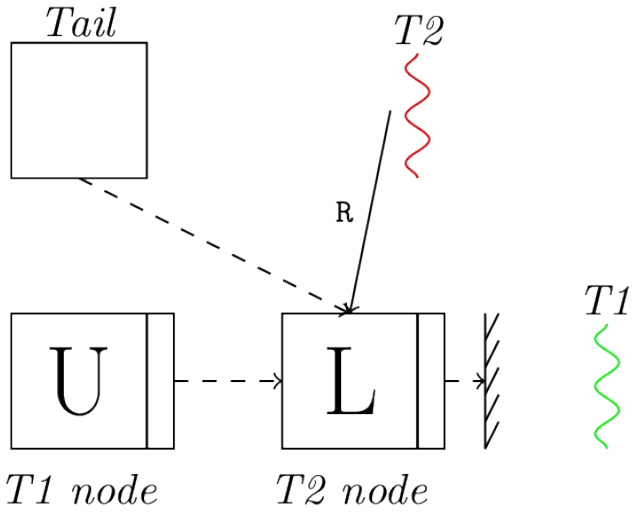
Direct handoff succession

- Example of the MCS lock algorithm



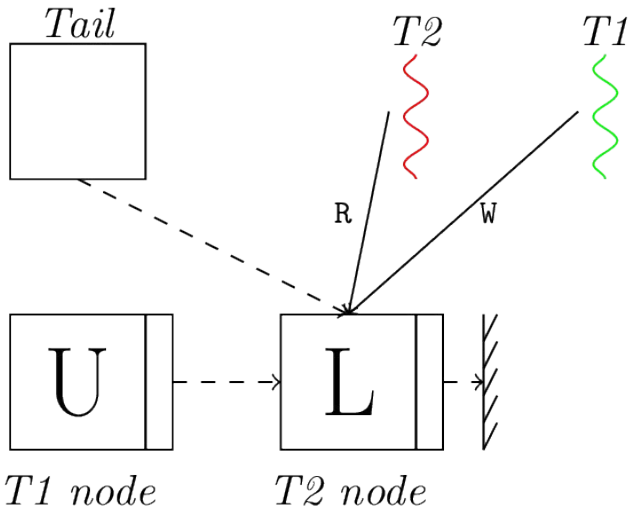
Direct handoff succession

- Example of the MCS lock algorithm



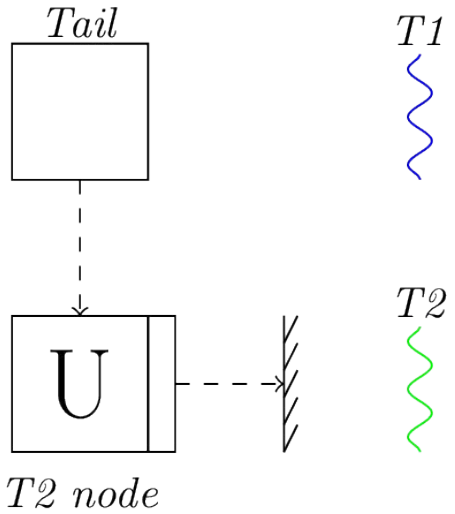
Direct handoff succession

- Example of the MCS lock algorithm



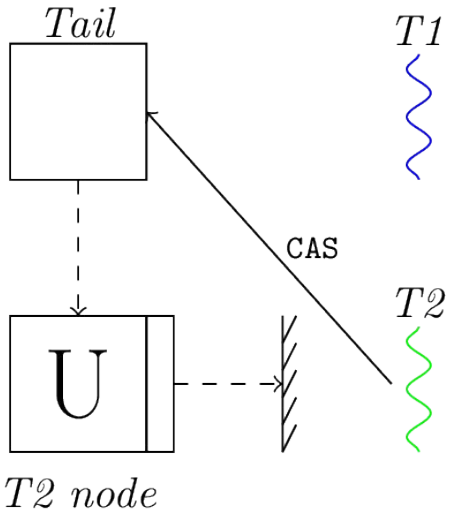
Direct handoff succession

- Example of the MCS lock algorithm



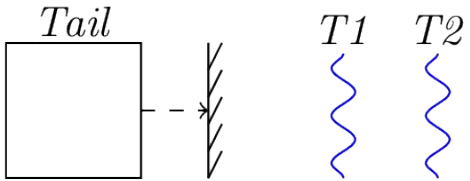
Direct handoff succession

- Example of the MCS lock algorithm



Direct handoff succession

- Example of the MCS lock algorithm



Hierarchical approaches

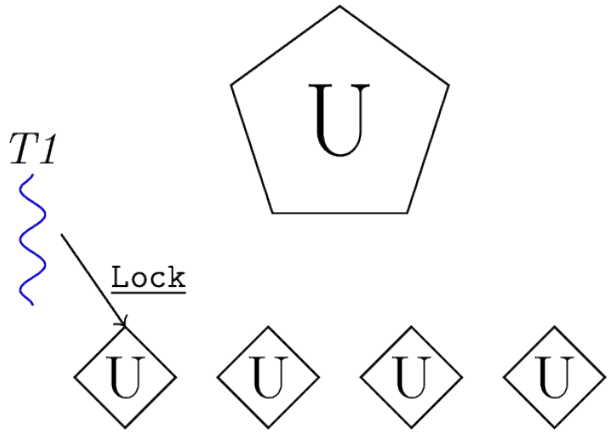
- Provide scalable performance on NUMA machines, by attempting to reduce lock migrations

- Favor threads running on the same NUMA node as the lock holder
  - Exchanging a cache line between cores of the same socket is less expensive than crossing the interconnect

- One lock algorithm for threads on the same NUMA socket, one algorithm for the global lock (can be the same)
  - E.g., Cohort locks

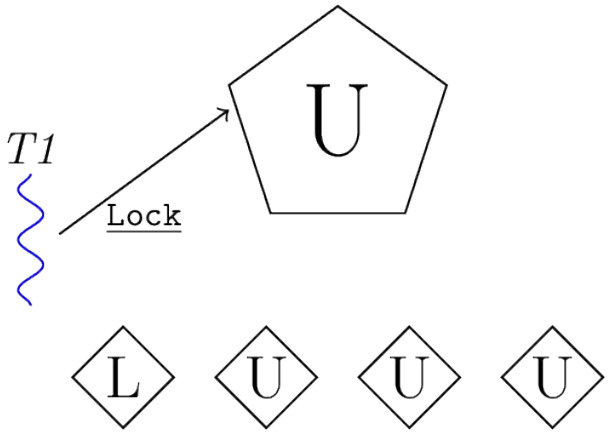
Hierarchical approaches

- Example of a cohort lock algorithm



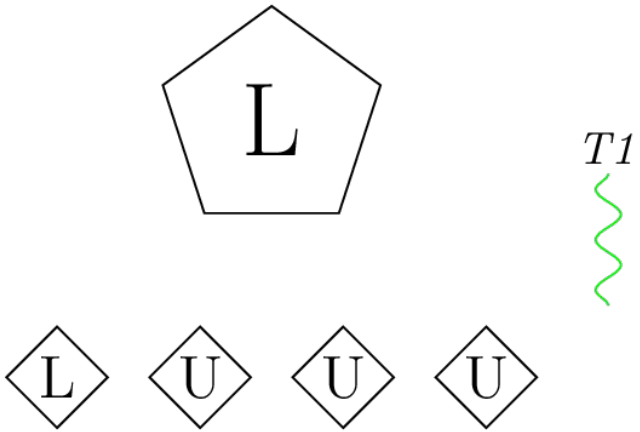
Hierarchical approaches

- Example of a cohort lock algorithm



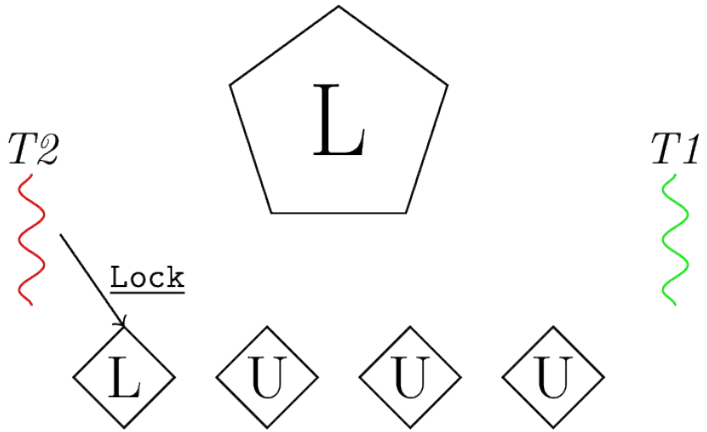
Hierarchical approaches

- Example of a cohort lock algorithm



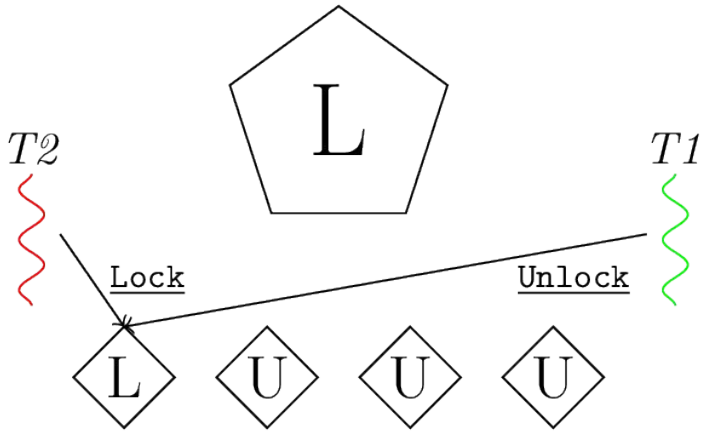
Hierarchical approaches

- Example of a cohort lock algorithm



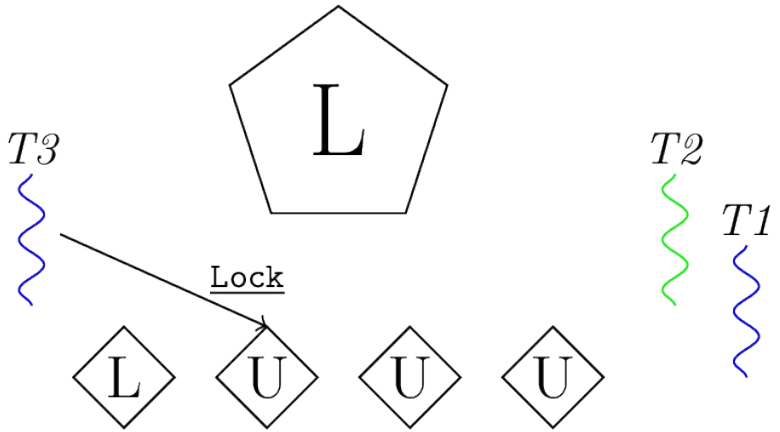
Hierarchical approaches

- Example of a cohort lock algorithm



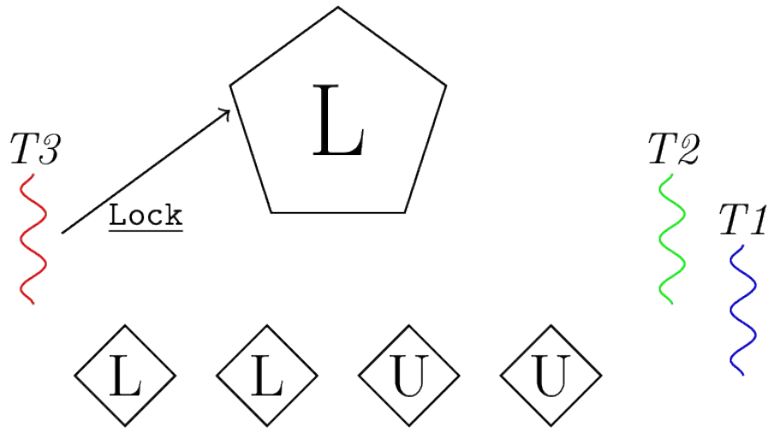
Hierarchical approaches

- Example of a cohort lock algorithm



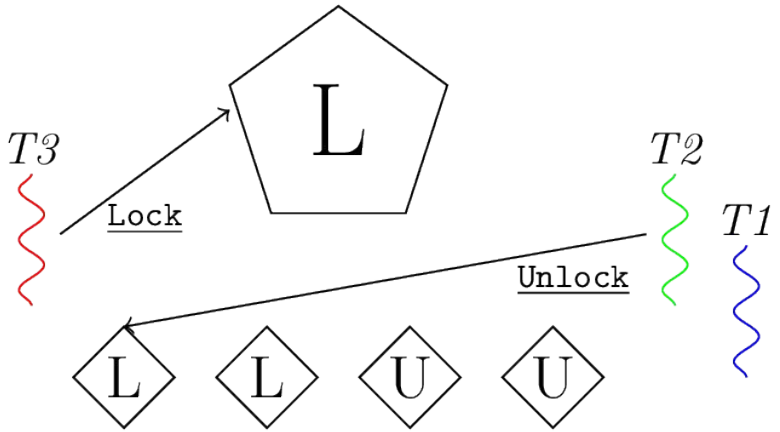
Hierarchical approaches

- Example of a cohort lock algorithm



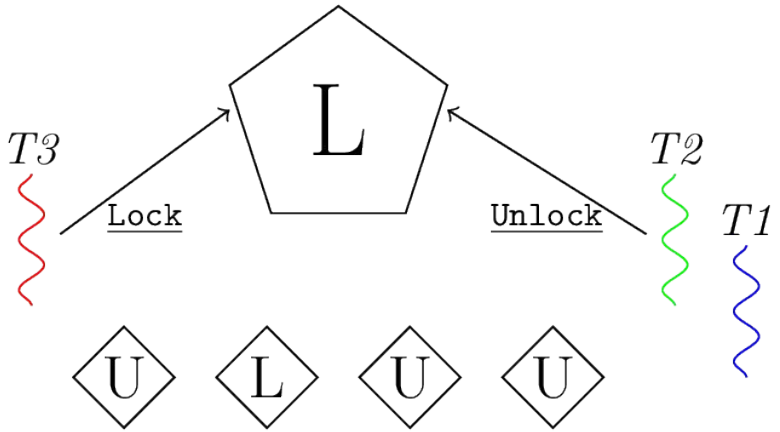
Hierarchical approaches

- Example of a cohort lock algorithm



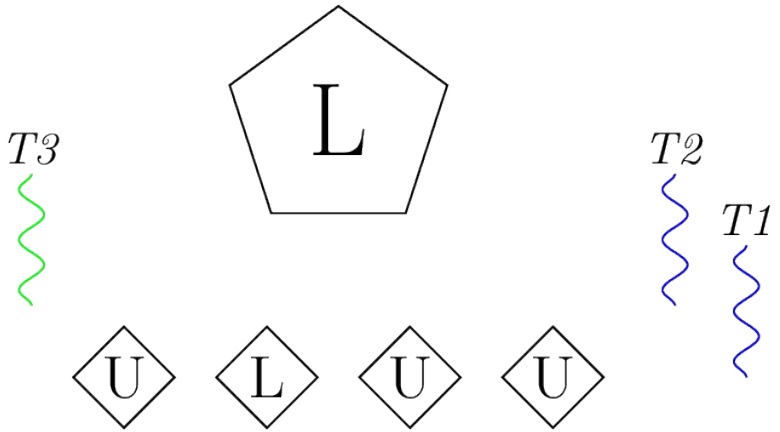
Hierarchical approaches

- Example of a cohort lock algorithm



Hierarchical approaches

- Example of a cohort lock algorithm



Delegation-based approaches

- A thread delegates the execution of a critical section to another thread
  - E.g., RCL dedicates one core to a server threads receiving CS execution requests from the client threads
- Improves cache locality within the critical section and better throughput under very high lock contention
  - Data is most likely to be already inside the caches
- Require the critical section to be expressed as a closure (e.g., a function), which is not compatible with the lock()/unlock() API
  - Need to modify the application source code

Load-control mechanisms

- Detect situations when a lock needs to adapt itself
- Varying levels of contention
  - Change locking scheme: AHMCS / Malthusian algorithms
  - Dynamically switch between lock algorithms: GLS / SANL
- Pathological lock-related behaviors (e.g., scheduler related)
  - MCS-TimePub / LC

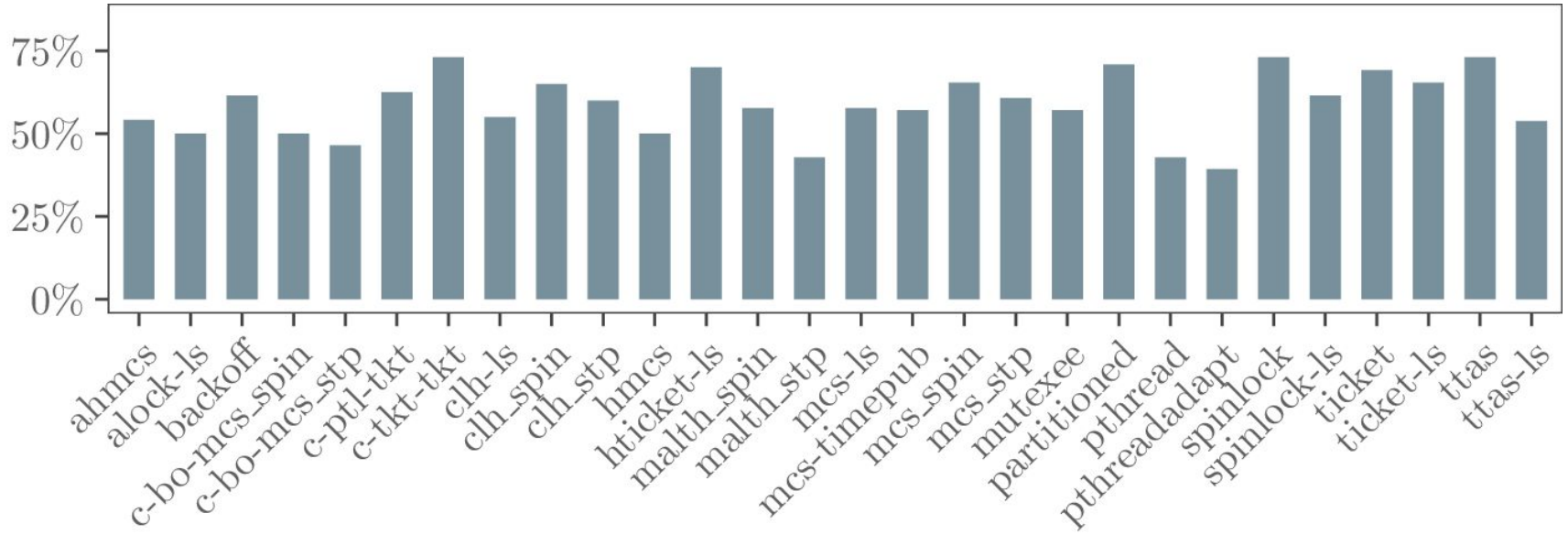
Statistical test

- Test if we can make meaningful comparison between locks with LiTL
  - Order and distance between lock algorithms performance is the same with and without interposition (relative comparisons)
  - Use a Student paired t-test (accept if p-value > 0.05 in general)

Application	C	p-value
linear_regression	-1.8%	0.84
matrix_multiply	-0.2%	0.60
radiosity_ll	-3.1%	0.72
s_raytrace_ll	-0.2%	0.85

Are some locks always among the best?

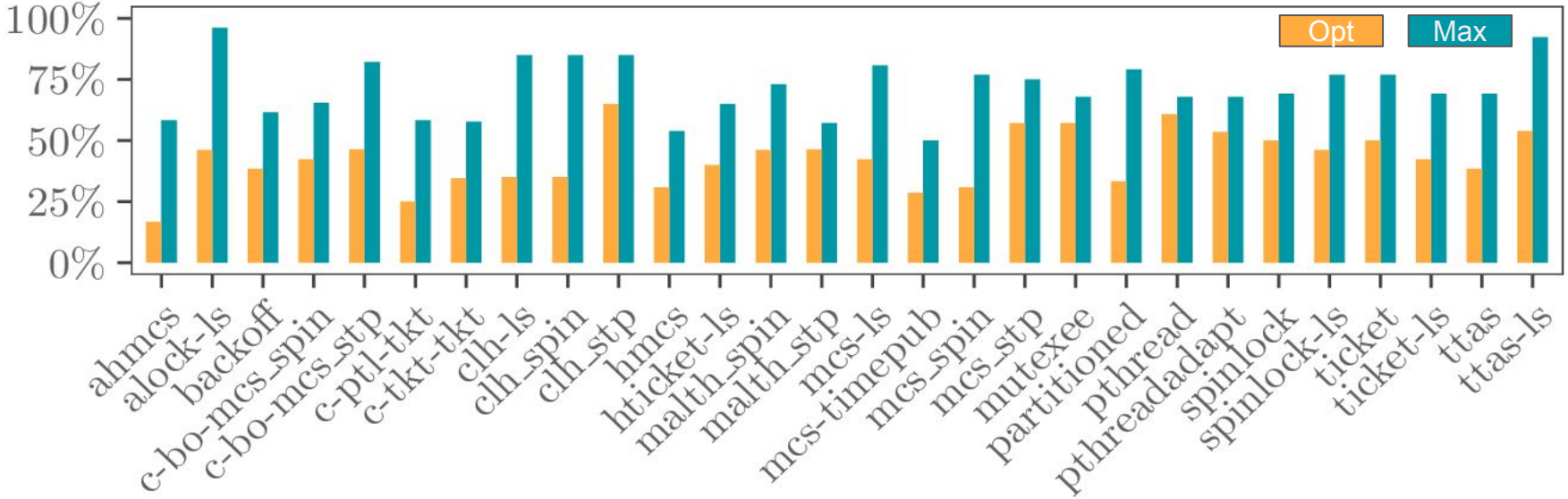
- At one node, no always-winning lock (max 73%)



Fraction of lock-sensitive application where a lock is optimal (the best at 5% of the best)

Are all locks potentially harmful?

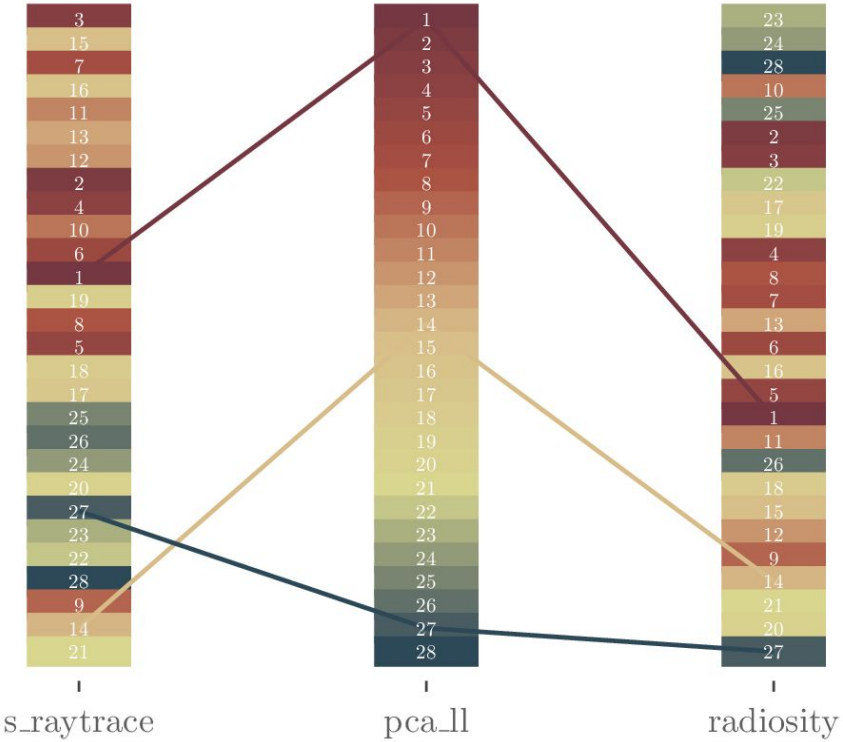
- Best case 17%, worst case 96%



Fraction of lock-sensitive application where a lock is harmful (at least 15% worse than the best lock)

Is there a clear hierarchy among locks?

- No clear hierarchy



Analysis: lock-related bottlenecks

- Lock contention: *multiple threads want to acquire the same lock*
  - High/extreme levels of contention, uncontended acquisitions, trylock contention
- Scheduling issues: *the scheduler choices trigger pathological behaviors*
  - Lock holder preemption, lock handover
- Memory:
  - Footprint: kernel taking time to zero memory pages for lock instances (at initialization time)
  - Concurrent allocations: induce lock contention inside the kernel
  - Controller saturation: “too performant” locks might exacerbate an application bottleneck

Lock properties

- Light:
  - Short code path to acquire the lock when the lock is uncontented
    - *Backoff, Mutexee, Pthread, Spinlock, TTAS*
- Hierarchical lock:
  - NUMA-aware locks
    - *Cohort locks, HMCS, HTicket, AHMCS*
- Contention-hardened trylock
  - Trylock operation that tolerates moderate to high levels of contention
    - *Partitioned, Cohort locks, HMCS, MCS-TimePub*
    - Not all locks can have trylocks (CLH, HTicket)

Lock properties

- Parking
  - Spin-then-park/park waiting policy
    - *Mutexee, Pthread, STP versions of locks, MCS-TimePub*
- FIFO
  - Impose an order on the acquisitions of a lock instance according to thread arrival times
    - *Direct-handoff succession locks, hierarchical locks, AHMCS, Malthusian*
- Low memory footprint
    - *Backoff, Pthread, Spinlock, Ticket, TTAS*
- Low memory traffic
  - Induce a moderate traffic on the memory interconnect/memory controllers of the machine
    - *Backoff, TTAS-ls, Malthusian*

Lessons learned

- Lock profiling tools should give a full profile of a lock
  - Interactions with scheduling and memory, other synchronization primitives, lock access patterns
- Need for dynamic and more complete approaches
  - The choice of the lock algorithm should not be hardwired into the application
  - Existing adaptive lock algorithms (e.g., AHMCS) are a step in the right direction, but they do not consider the full spectrum of lock-related performance bottlenecks

Future research

- Multicore performance
  - Study other performance factors such as the OS scheduler, memory allocation, compiler
  - Revisit scheduler and memory allocation for the micro/nano scale era
- Delegation algorithms
  - Better integration with the Pthread locking API
  - Interaction with the scheduler
- Automatic and dynamic solutions
  - Changing lock at run time to account for runtime conditions (scheduling, memory, ...)
- Leveraging transactional memory
  - Mixing transactional memory and locking should allow to deal with varying levels of contention