Locks on Multicore and Multisocket Platforms

John Mellor-Crummey

Department of Computer Science
Rice University

johnmc@rice.edu

Context

- Last lecture: locks and barriers
- Lock synchronization on multicore platforms
- Upcoming
  - —transactional memory
  - —practical non-blocking concurrent objects

Papers for Today

- **Everything you always wanted to know about synchronization but were afraid to ask.** David Tudor, Rachid Guerraoui, and Vasileios Trigonakis. In Proceedings of SOSP '13. ACM, New York, NY, USA, 33-48.
- **Lock cohorting: a general technique for designing NUMA locks.** David Dice, Virendra J. Marathe, and Nir Shavit. In Proceedings PPoPP '12. ACM, New York, NY, USA, 247-256. 2012.

Motivation for Studying Lock Performance

- There are many types of locks and architectures
- Does lock performance depend on architecture?
- How?
- Which lock is best?

Locks

- Test and set (TAS)
- Test and test and set (TTAS) Ticket lock
- Array-based lock
- MCS lock
- CLH lock
- Hierarchical CLH lock (HCLH)
- Hierarchical Ticket lock (HTICKET)
- Hierarchical backoff lock

FIFO

Locks

- Test and set (TAS)
- Test and test and set (TTAS) Ticket lock
- Array-based lock
- MCS lock
- CLH lock
- Hierarchical CLH lock (HCLH)
- Hierarchical Ticket lock (HTICKET)
- Hierarchical backoff lock

Queuing Locks

Locks

- Test and set (TAS)
- Test and test and set (TTAS) Ticket lock
- Array-based lock
- MCS lock
- CLH lock
- Hierarchical CLH lock (HCLH)
- Hierarchical Ticket lock (HTICKET)
- Hierarchical backoff lock

Locks we haven’t discussed

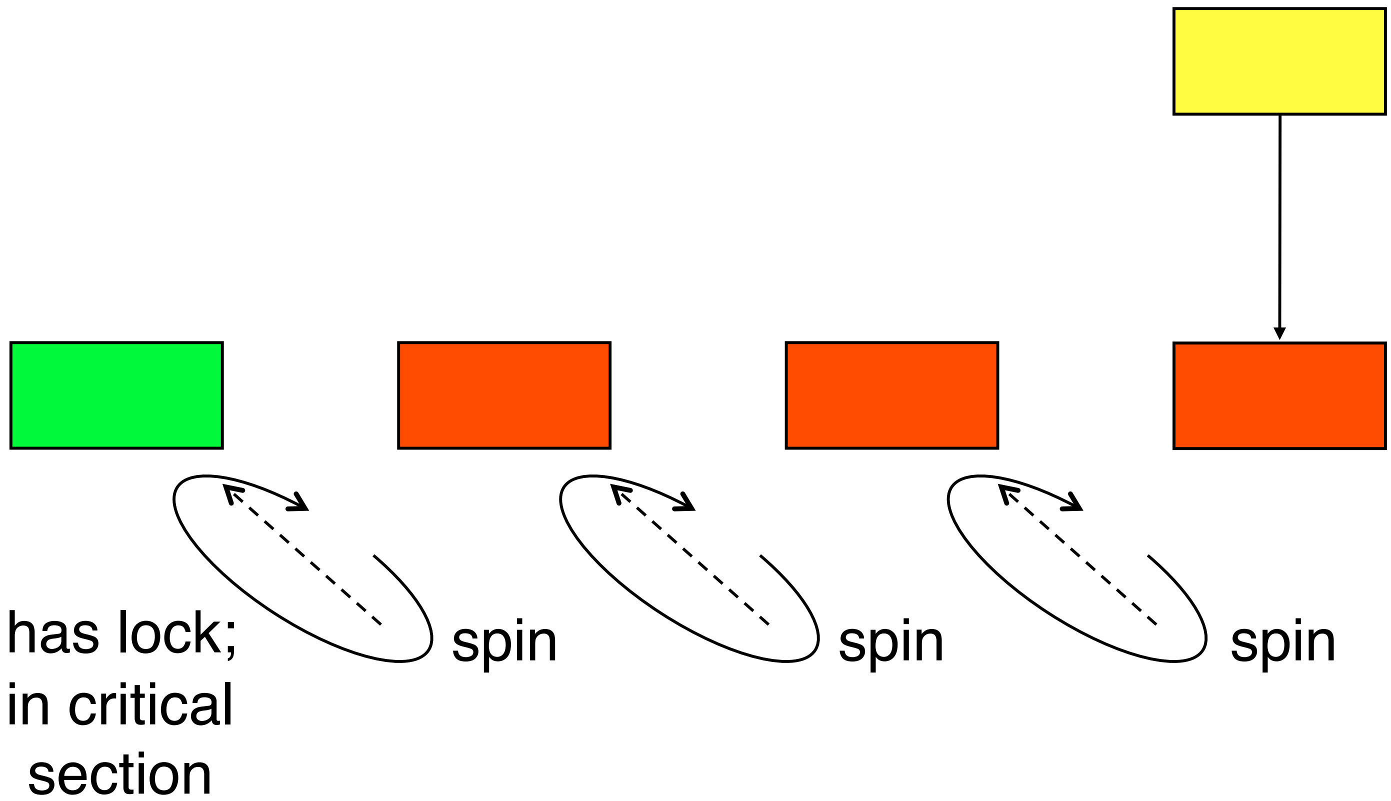
CLH List-based Queue Lock

```
type qnode = record
  prev : ^qnode
  succ_must_wait : Boolean

type lock = ^qnode    // initialized to point to an unowned qnode

procedure acquire_lock (L : ^lock, I : ^qnode)
  I->succ_must_wait := true
  pred : ^qnode := I->prev := fetch_and_store(L, I)
  repeat while pred->succ_must_wait

procedure release_lock (ref I : ^qnode)
  pred : ^qnode := I->prev
  I->succ_must_wait := false
  I := pred          // take pred's qnode
```



CLH

CLH Queue Lock Notes

- Discovered twice, independently
  - Travis Craig (University of Washington)
    - TR 93-02-02, February 1993
  - Anders Landin and Eric Hagersten (Swedish Institute of CS)
    - *IPPS*, 1994
- Space: $2p + 3n$ words of space for p processes and n locks
  - MCS lock requires $2p + n$ words
- Requires a local "queue node" to be passed in as a parameter
- Spins only on local locations on a cache-coherent machine
- Local-only spinning possible when lacking coherent cache
  - can modify implementation to use an extra level of indirection

    (local spinning variant not shown)
- Atomic primitives: fetch_and_store

Why Hierarchical Locks?

NUMA architectures

- Not all memory is equidistant to all cores
  - —each socket has its own co-located memory
  - —consequence of scaling memory bandwidth with processor count
- Today's systems: system-wide cache coherence
- Access latency depends on the distance between the core and data location
  - —memory or cache in local socket
  - —memory or cache in remote socket
- Multiple levels of locality
  - —0 hop, 1 hop, 2 hop, ...

Locks on NUMA Architectures

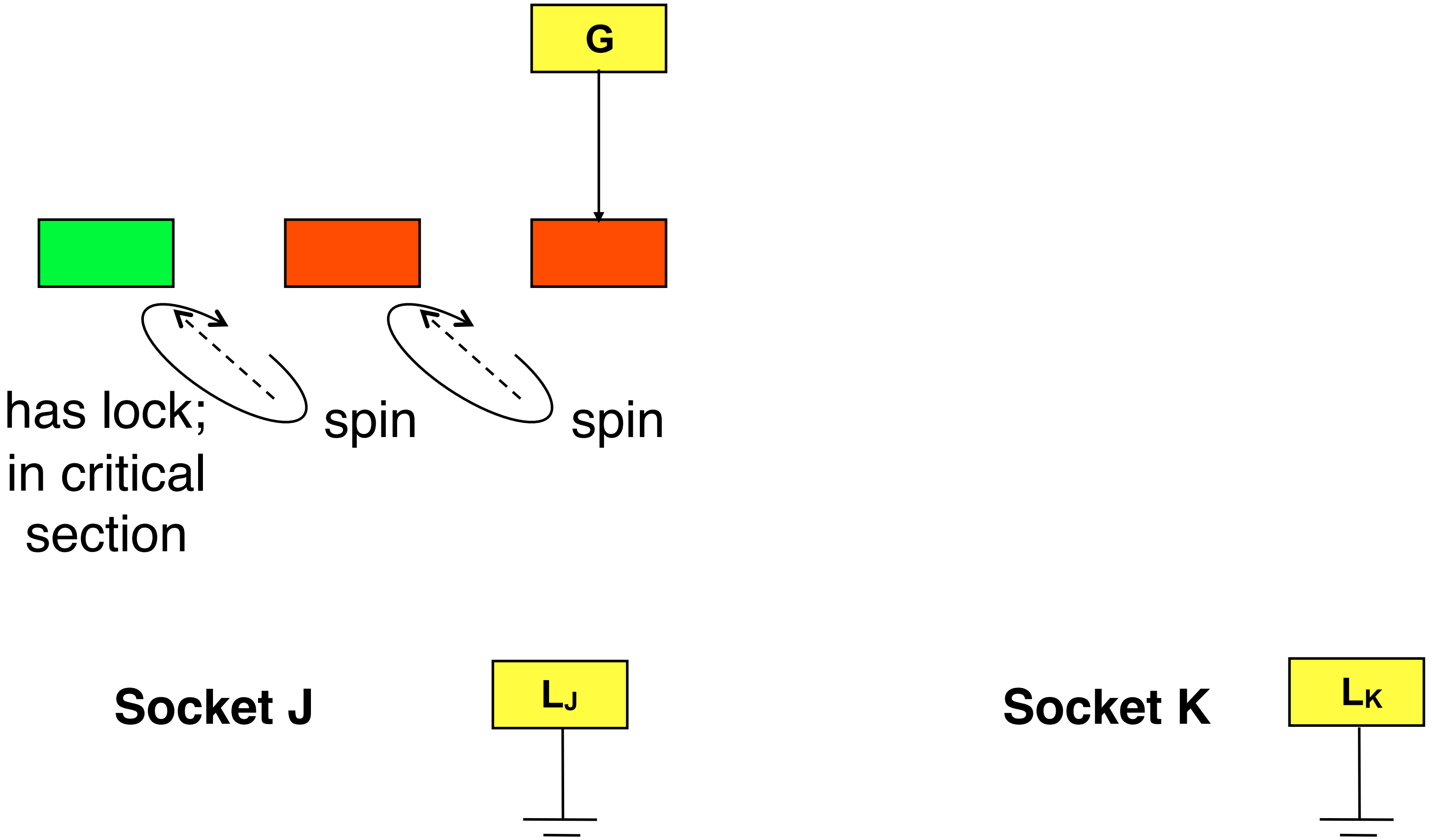
- **Problem:**
  - —passing locks between threads on different sockets can be costly
  - —overhead from passing lock and data it protects
  - —data that has been accessed on a remote socket produces long latency cache misses
- **Solution:**
  - —design locks to improve locality of reference
  - —encourage threads with mutual locality to acquire a given lock consecutively
- **Benefits:**
  - —reduce migration of locks between NUMA nodes
  - —reduce cache misses for data accessed in a critical section

Hierarchical CLH

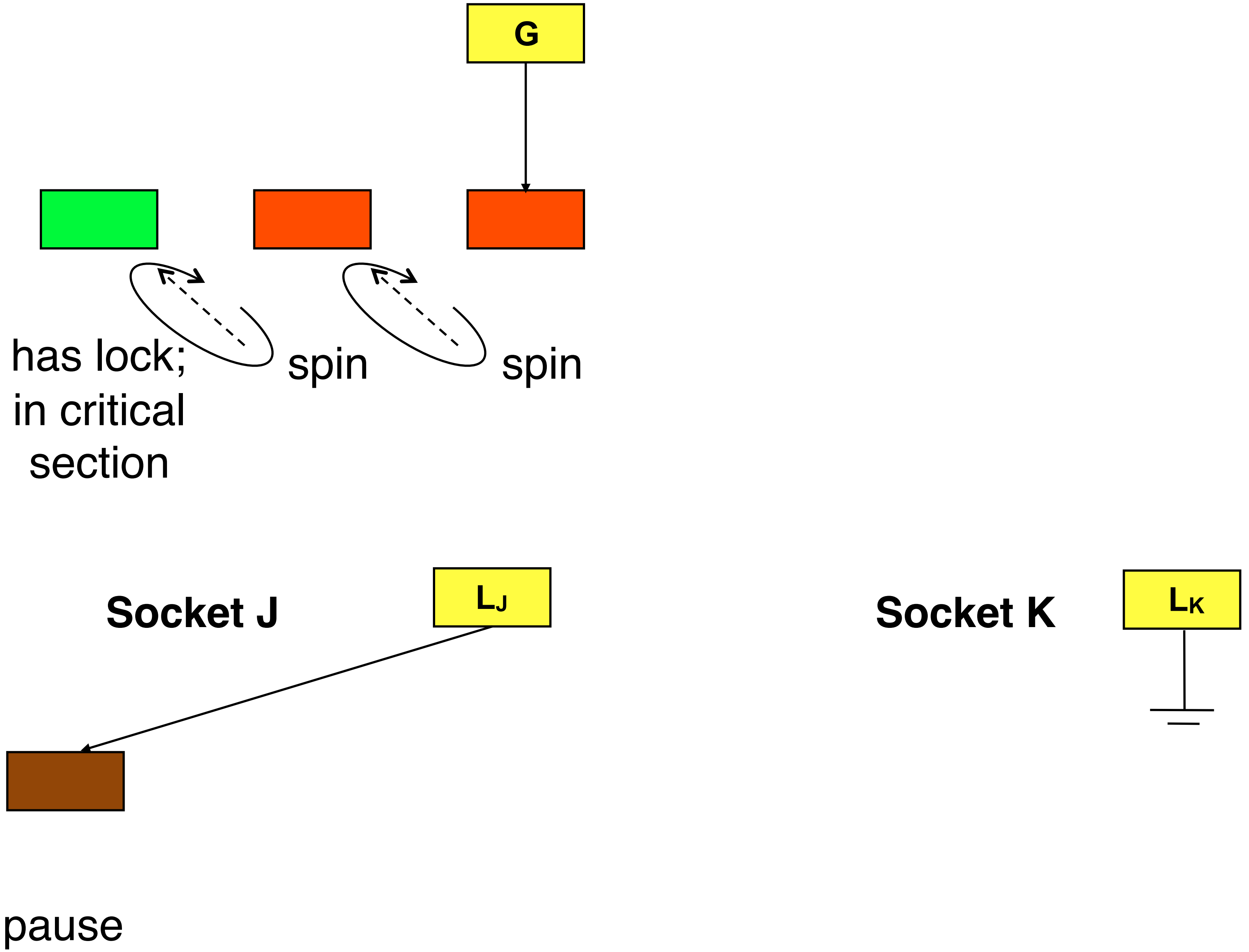
- **Structure**
  - **—local CLH queue per cluster (socket)**
  - **—one global queue**
  - **—qnode at the head of the global queue holds the lock**
- **Operation: when a node arrives in the local queue …**
  - **—delay for a bit to let successors arrive**
  - **—move a batch from a socket queue to the global queue**
    - **CAS local tail into global tail**
    - **link local head behind previous global tail**

Victor Luchangco, Dan Nussbaum, and Nir Shavit. A hierarchical CLH queue lock. In Proceedings of Euro-Par '06, Wolfgang E. Nagel, Wolfgang V. Walter, and Wolfgang Lehner (Eds.). Springer-Verlag, Berlin, Heidelberg, 801-810. 2006.

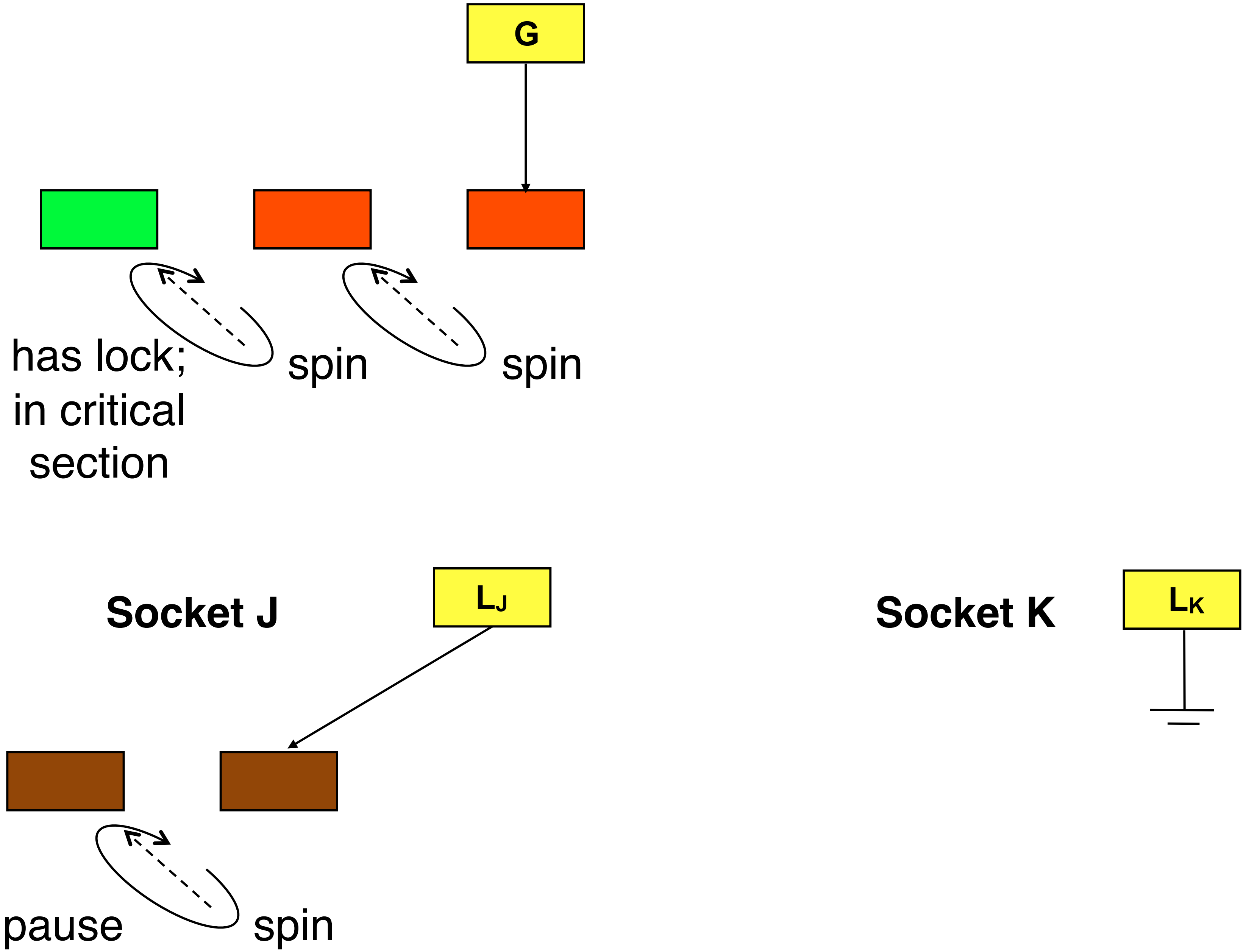
Hierarchical CLH in Action



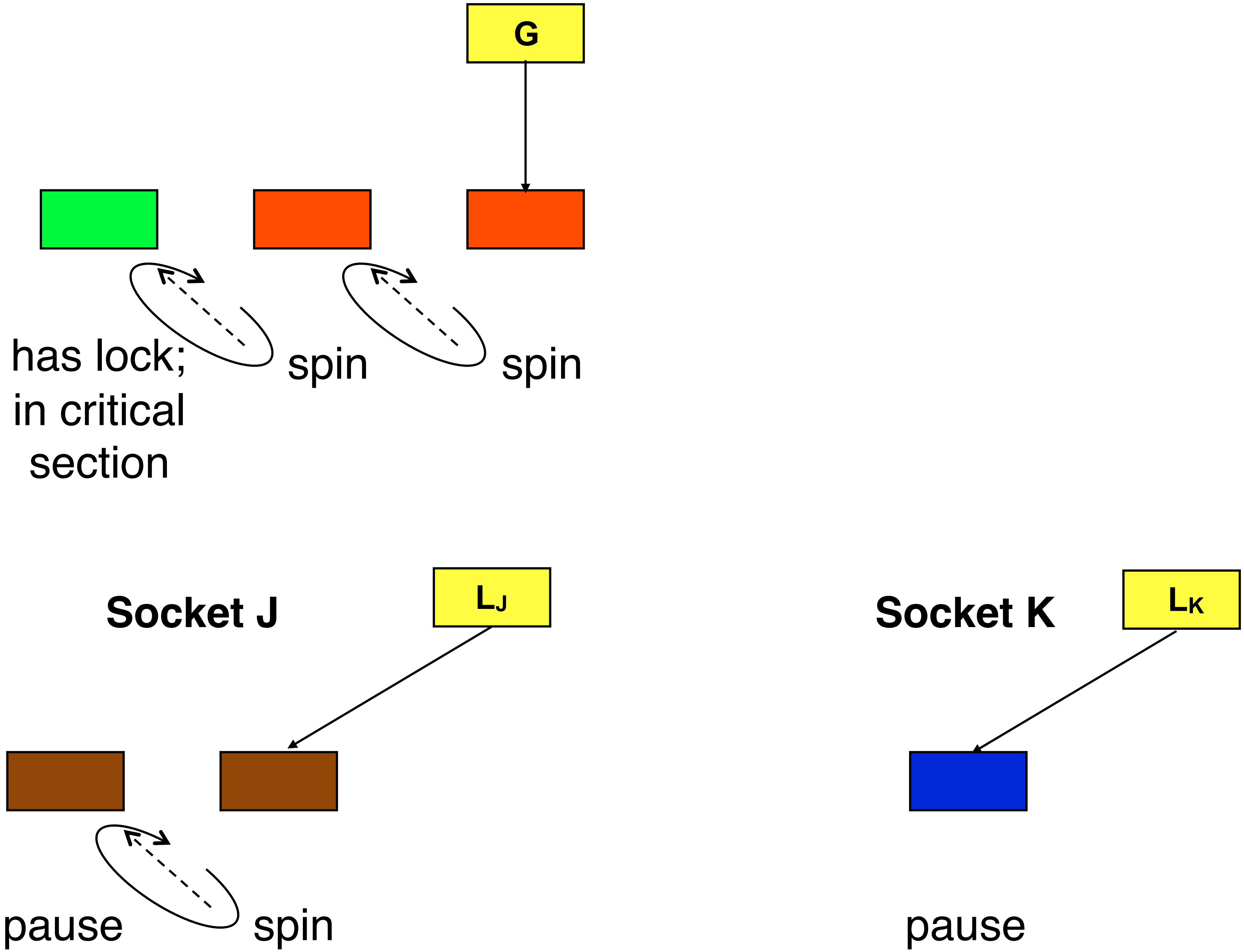
Hierarchical CLH in Action



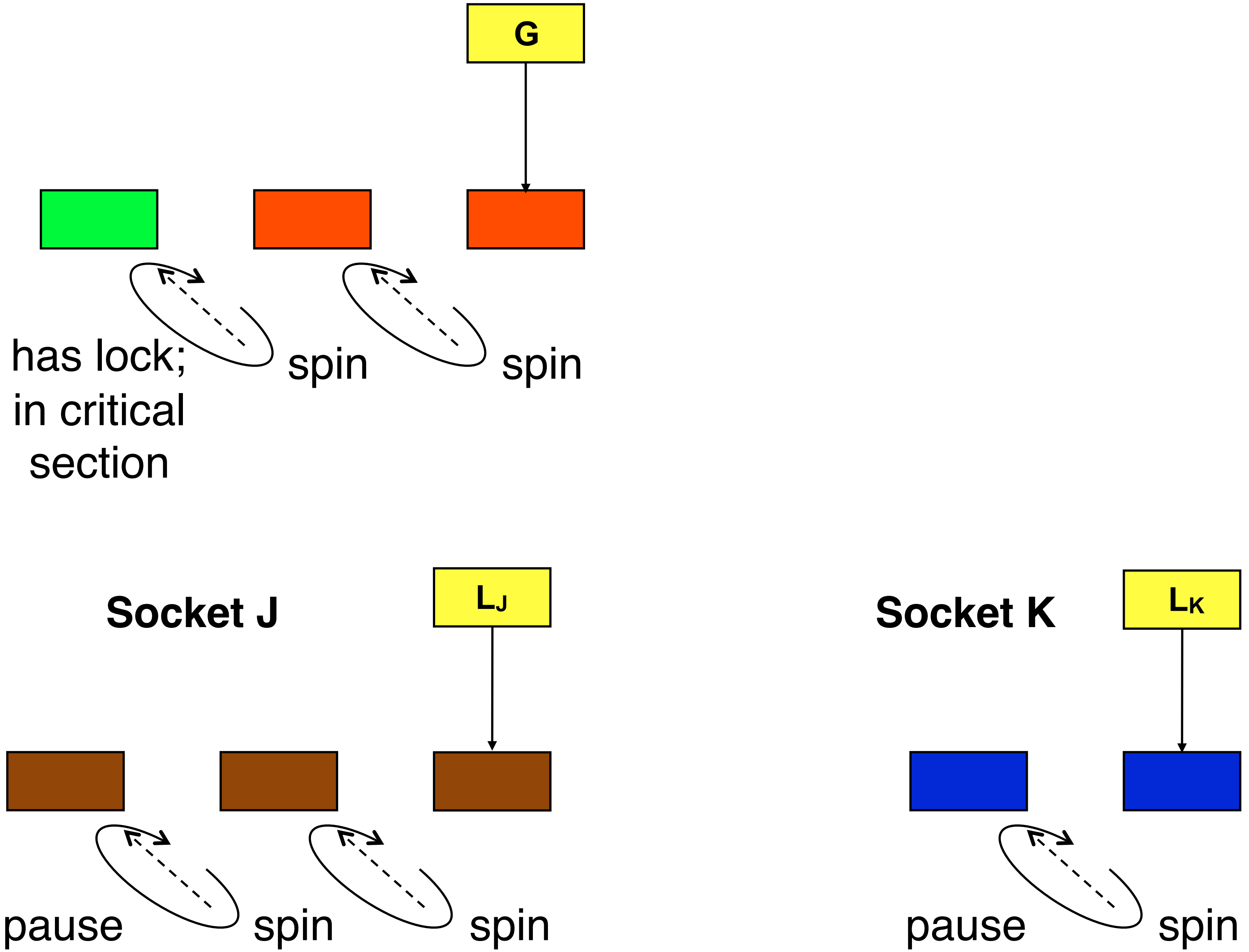
Hierarchical CLH in Action



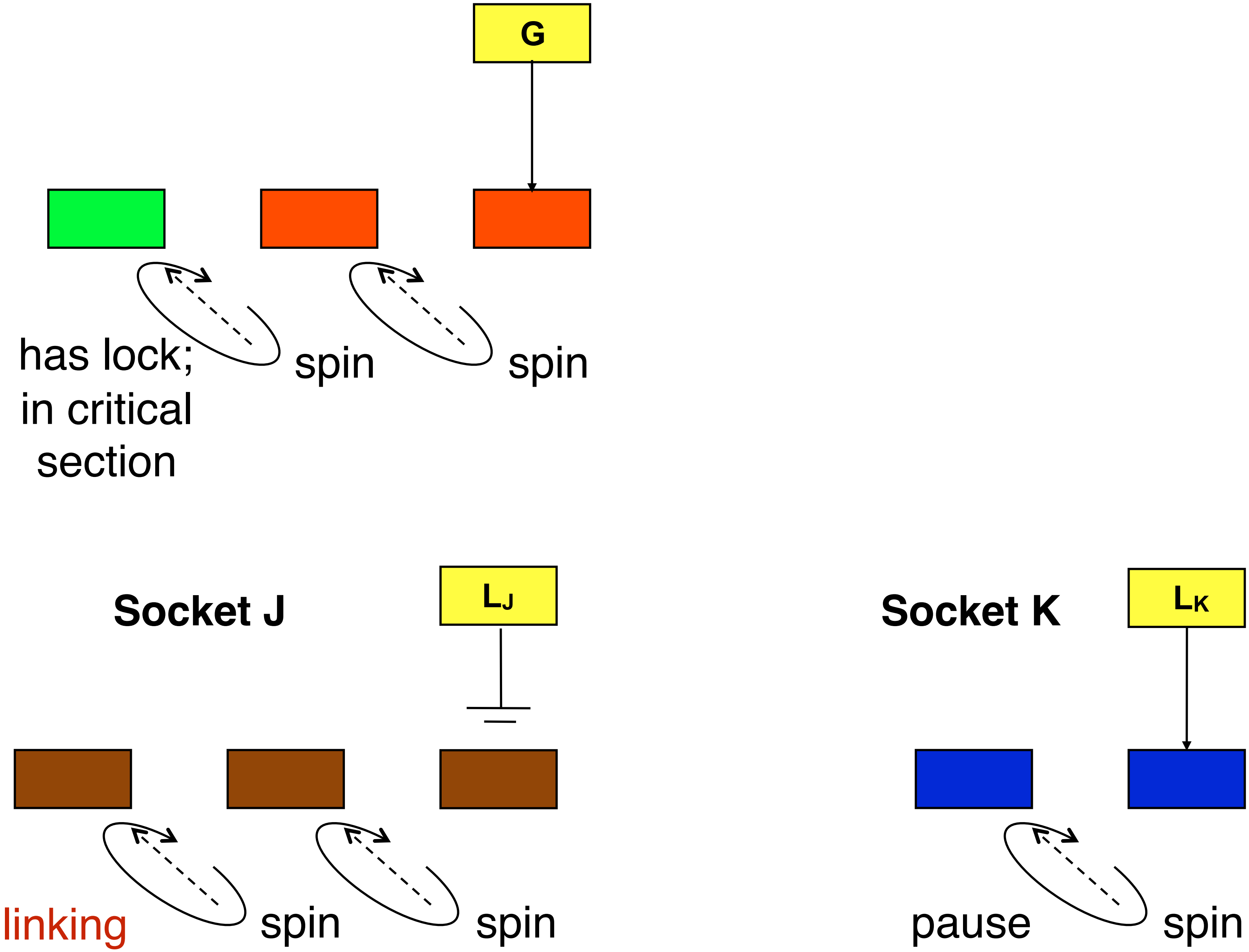
Hierarchical CLH in Action



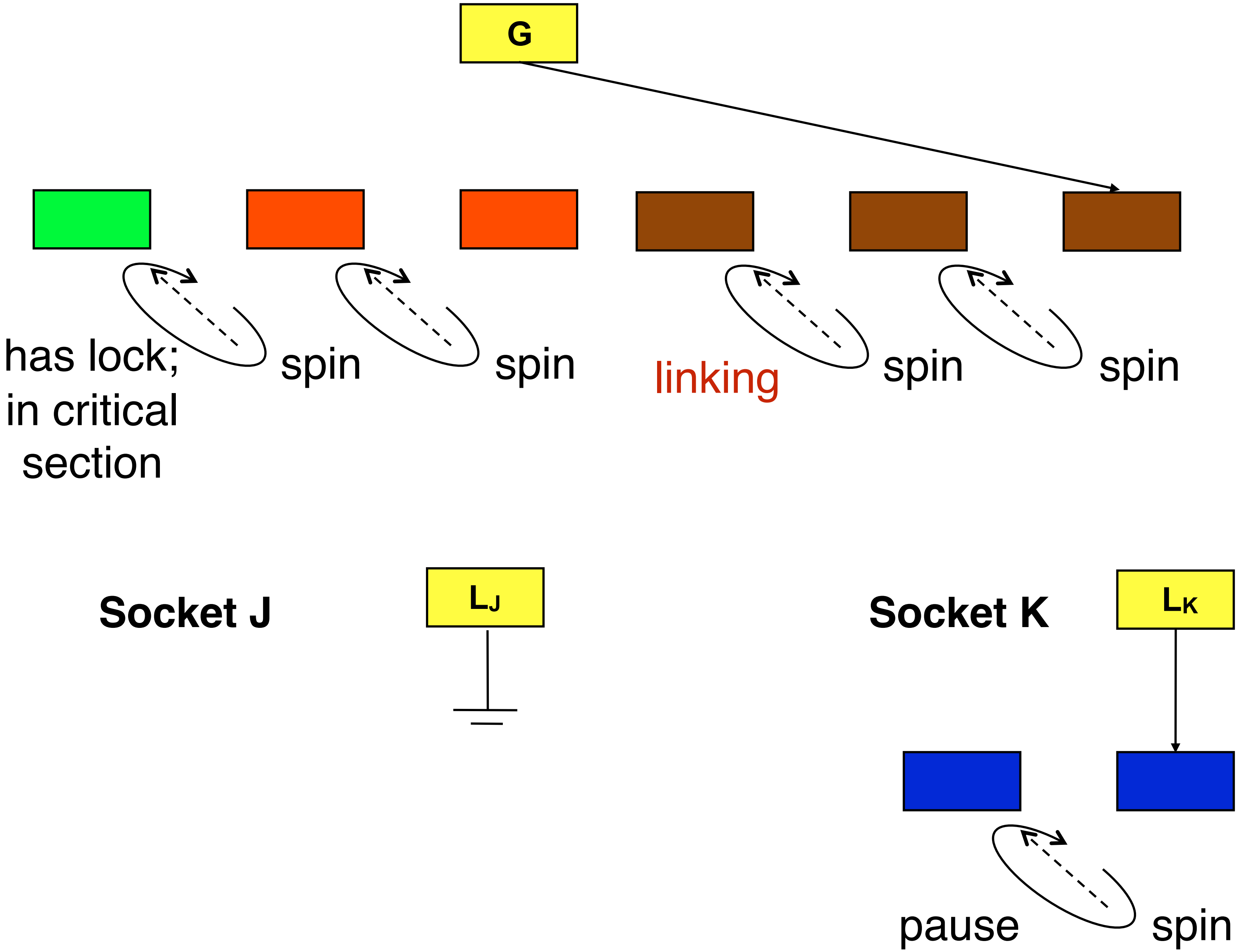
Hierarchical CLH in Action



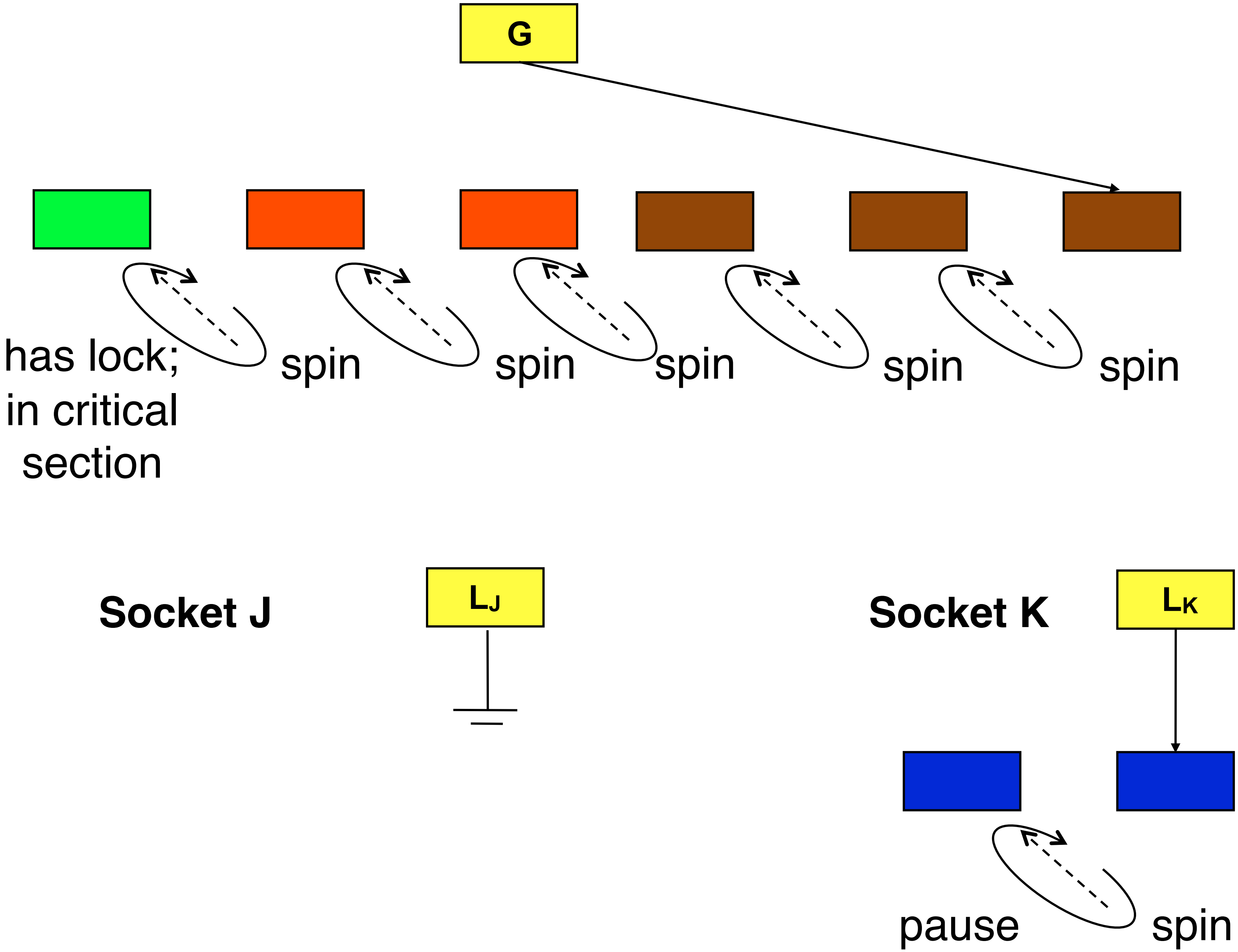
Hierarchical CLH in Action



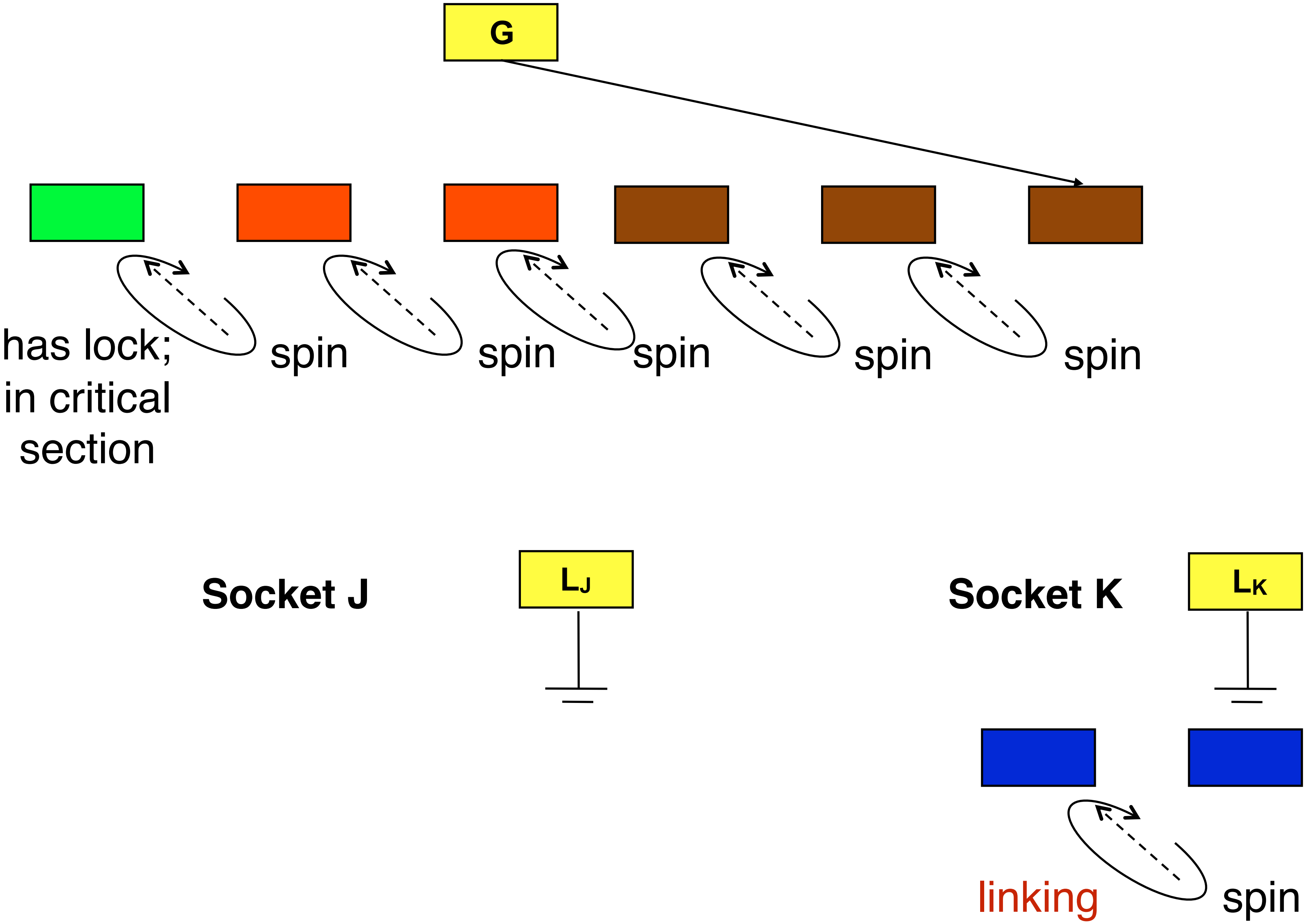
Hierarchical CLH in Action



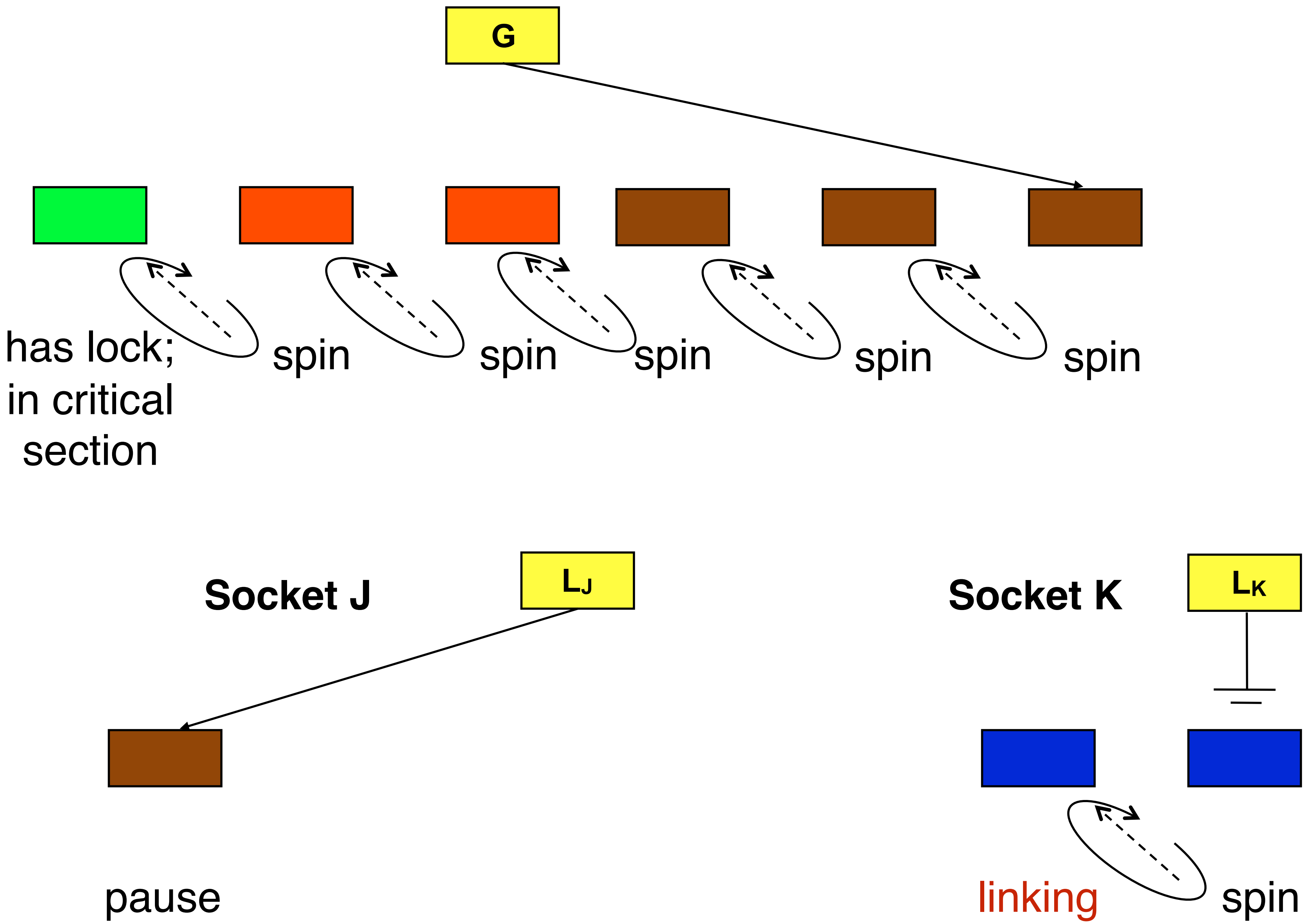
Hierarchical CLH in Action



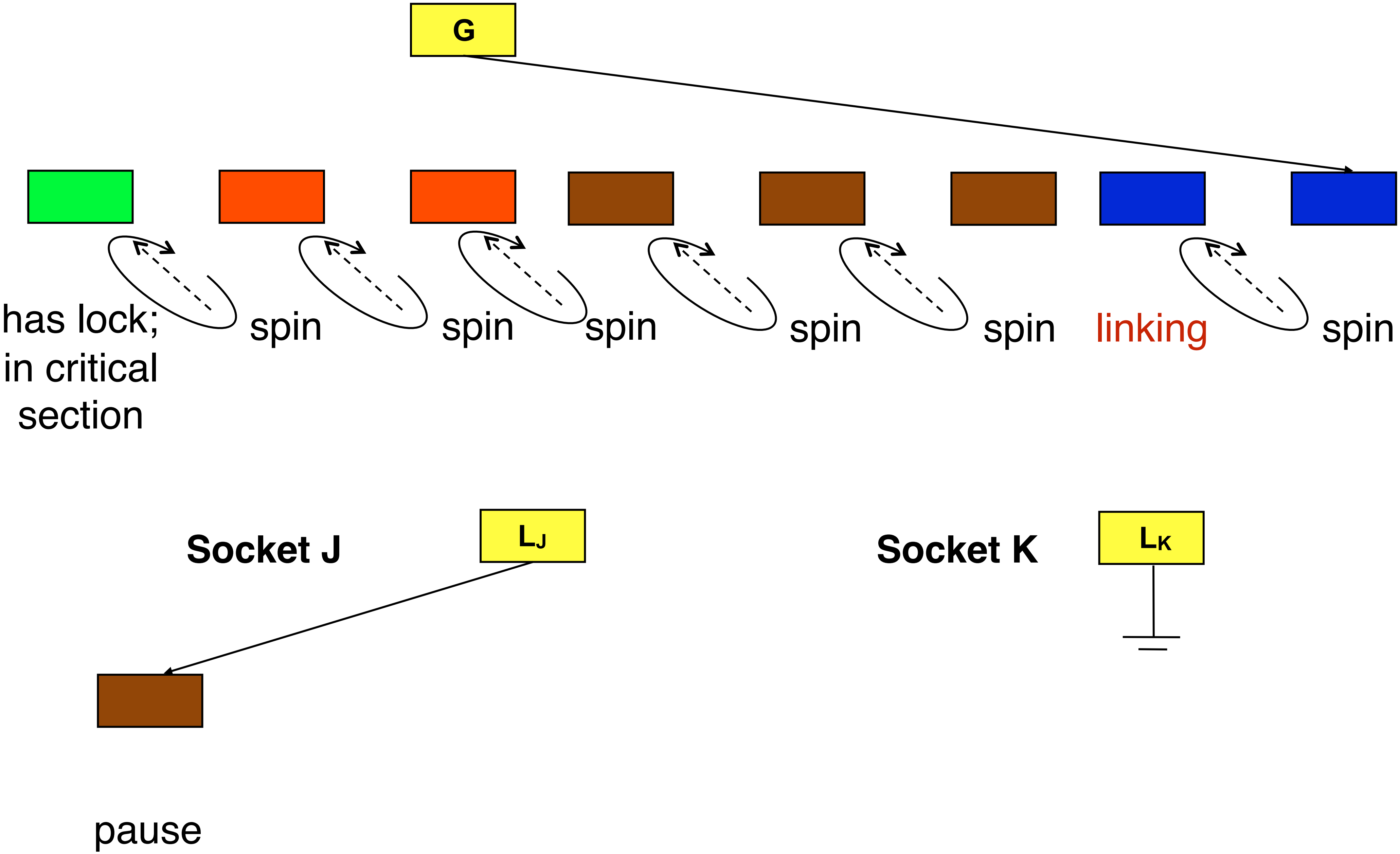
Hierarchical CLH in Action



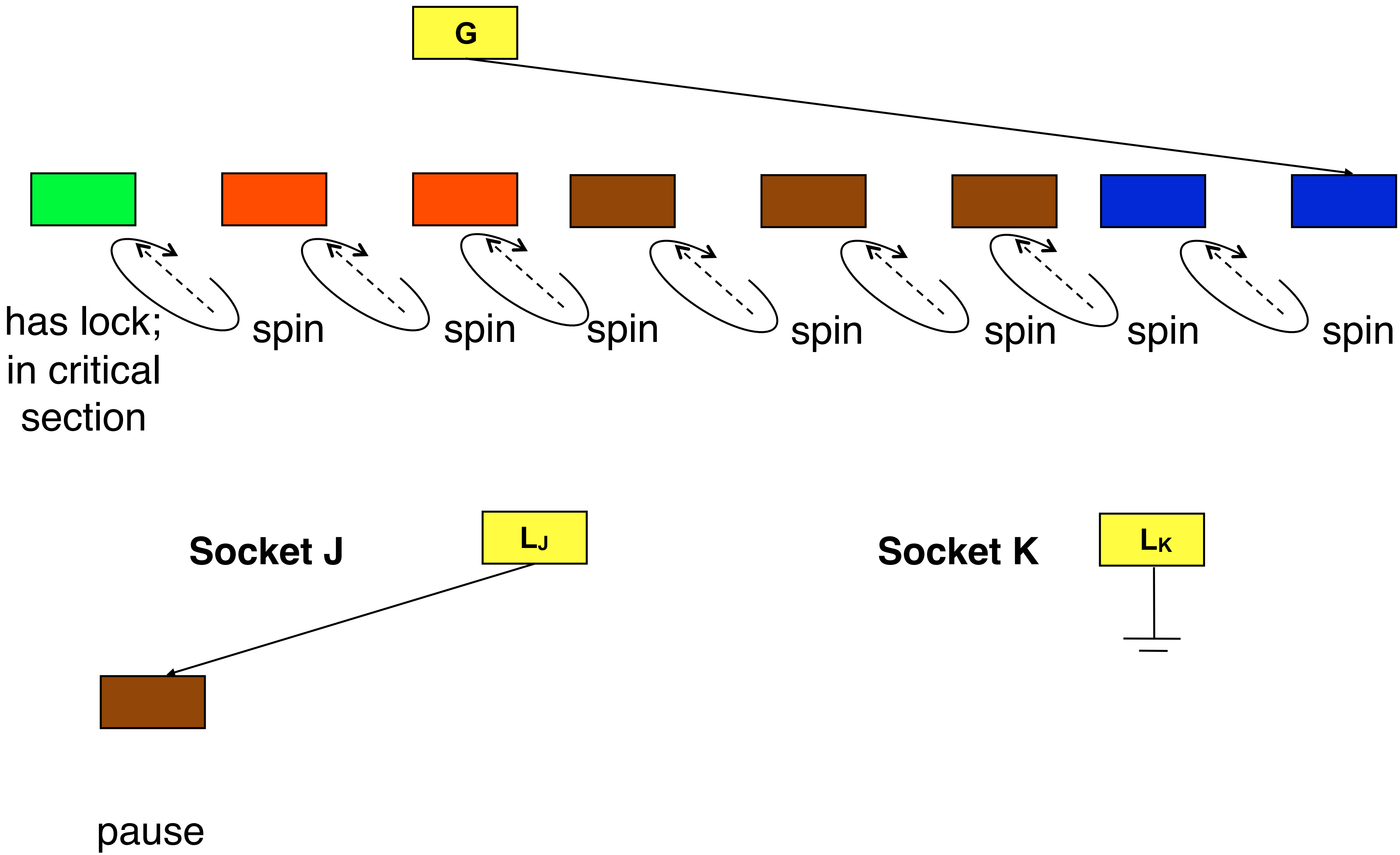
Hierarchical CLH in Action



Hierarchical CLH in Action



Hierarchical CLH in Action



Hierarchical Ticket

- Two levels of ticket locks
  - —global
  - —local: one per socket
- Two-level ticket lock (cohorting version by Dice et al.)
  - —acquire
    - – acquire local ticket
    - – if flag "global granted" is set, proceed
    - – else acquire global ticket lock
  - —release
    - – if successors available in local lock, set "global granted" for local lock and increment local ticket
    - – otherwise, clear "global granted" for local lock and increment global ticket
- "Everything…" paper used a more complex version
  - —https://github.com/tudordavid/libslock/blob/master/src/htlock.c

Hierarchical Backoff Lock

- **Test-and-test-and-set lock with back off scheme to reduce cross node contention of a lock variable**
- **Use thread locality to tune backoff delay**
  - —when acquiring a lock
    - – assign thread ID to lock state
  - —when spin waiting
    - – compare thread ID with lock holder and back off proportionally
- **Limitations:**
  - —reduce lock migration only probabilistically
  - —lots of invalidation traffic: costly for NUMA

Z. Radovic and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *HPCA-9*, pages 241–252, Anaheim, California, USA, Feb. 2003.

Systems with Different Characteristics

- **Opteron: 4 x AMD Opteron 6172 (48 cores)**
  - —directory based cache coherence
  - —directory located in LLC
- **Xeon: 8 x Intel Xeon E7-8867L (80 cores; SMT disabled)**
  - —broadcast snooping
- **Niagara: SUN UltraSPARC-T2 (8 cores; 64 threads)**
  - —coherence via shared L2 cache on far side of chip
- **Tilera: TILE-Gx CPU (36 cores)**
  - —coherence via distributed, shared L2 cache

Opteron Platform

- **Opteron: 4 x AMD Opteron 6172 (48 cores)**
- **Each chip contains two 6-core dies**
- **MOESI protocol, directory based cache coherence**
  - **—directory located in LLC**
- **Average distance: 1.25 hops**

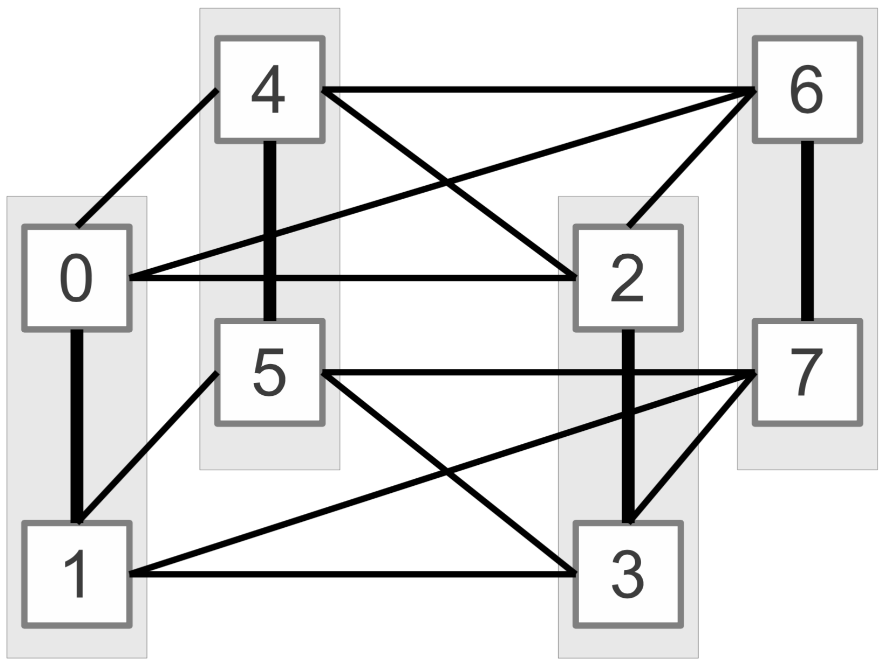


Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP '13. ACM, New York, NY, USA, 33-48.

Xeon Platform

- **Xeon: 8 x Intel Xeon E7-8867L (80 cores; SMT disabled)**
  - **—broadcast snooping**
- **10 cores per socket**
- **Average distance: 1.375 hops**

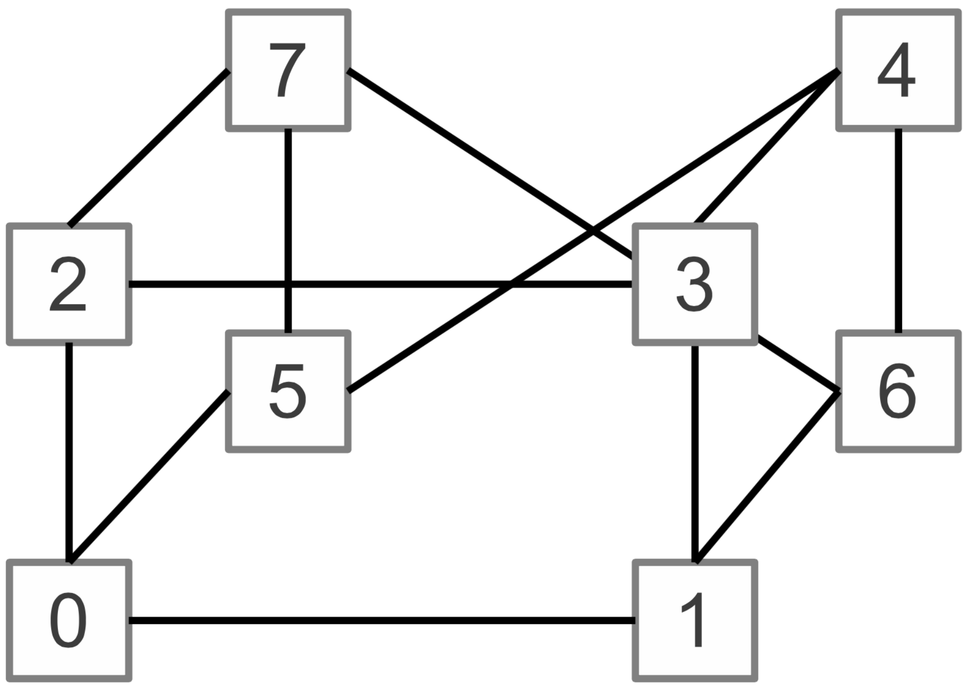


Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP '13. ACM, New York, NY, USA, 33-48.

Niagara

- **Niagara: SUN UltraSPARC-T2 (8 cores; 64 threads)**
  - **—coherence via shared L2 cache on far side of chip**

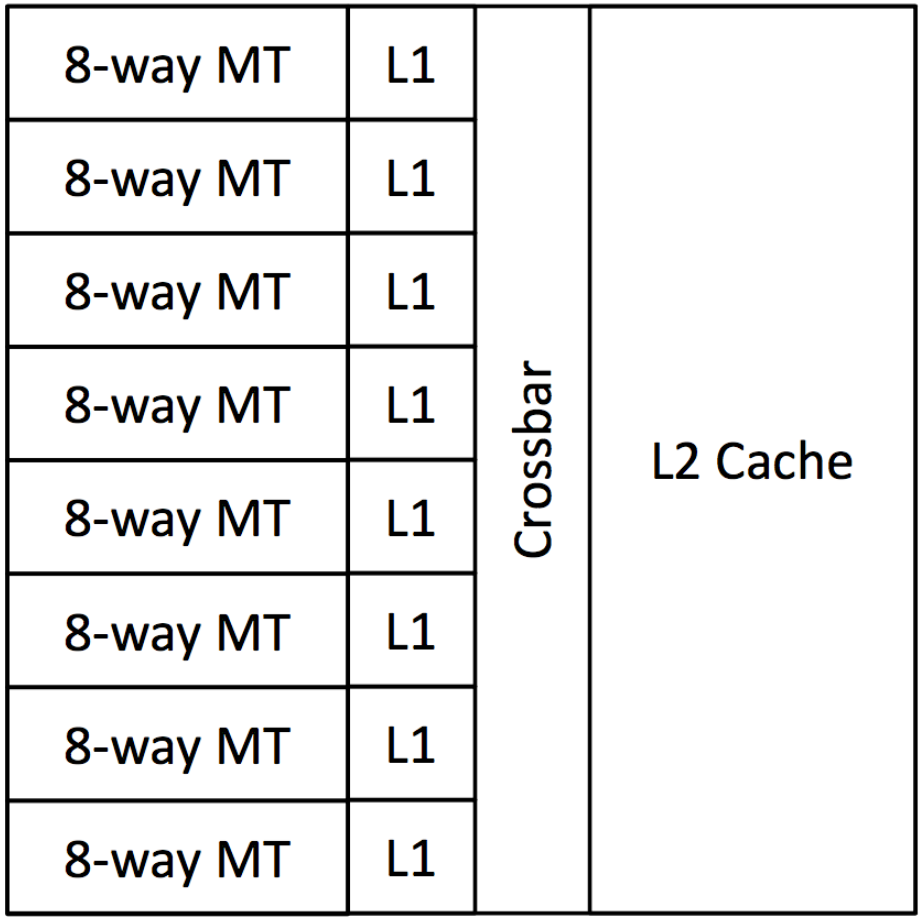


Figure credit: Niagara: A 32-way Multithreaded SPARC Processor; P. Kongetira, K. Aingaran, K. Olukotun

Tilera

- **Tilera: TILE-Gx CPU (36 cores)**
  - —coherence via distributed, shared L2 cache

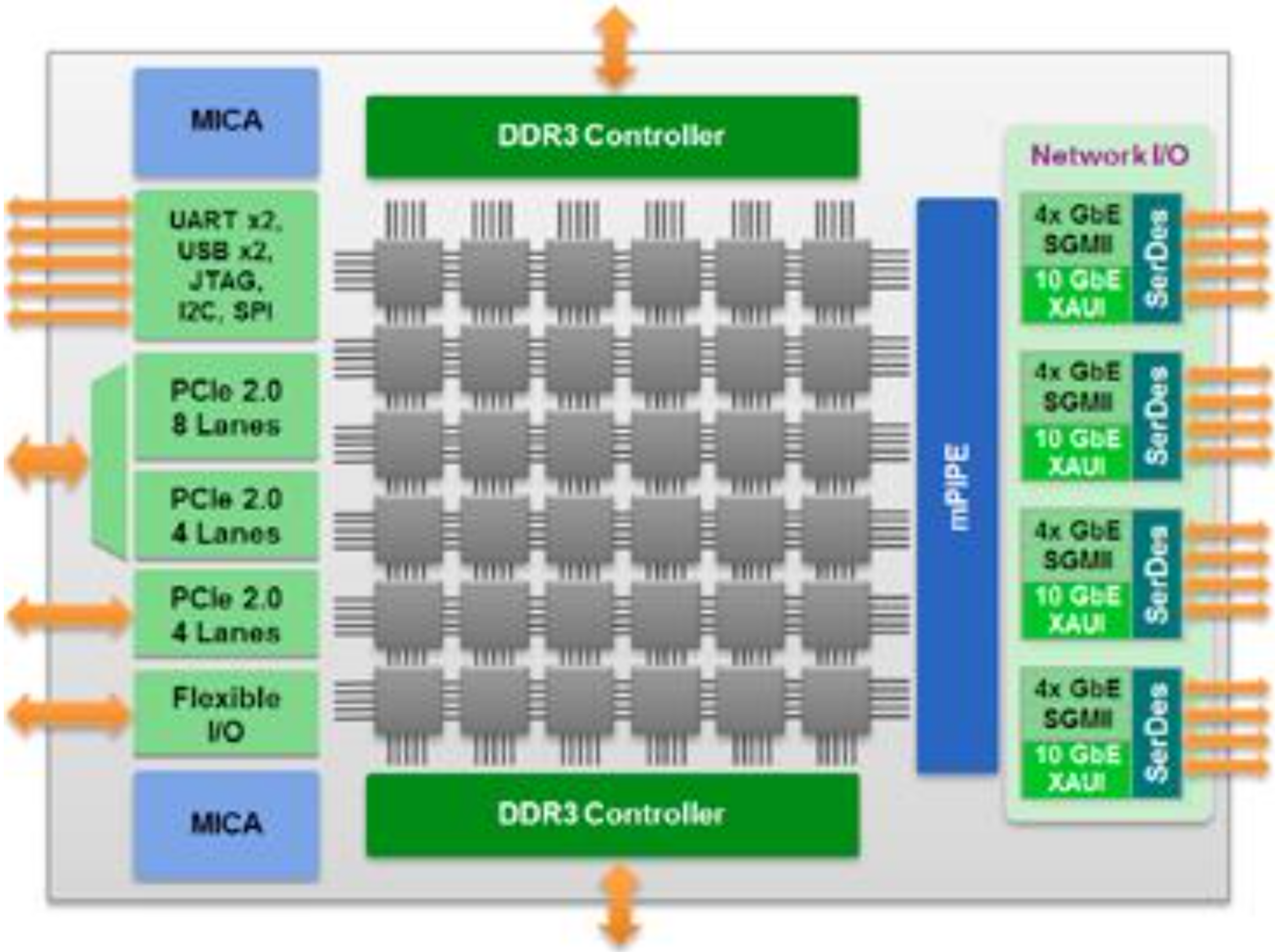


Figure credit: http://www.tilera.com/sites/default/files/productbriefs/TILE-Gx8036_PB033-02_web.pdf

Operation Latency Across Platforms

Latencies depend upon distance and (sometimes) state

System	Opteron (2.1 GHz)				Xeon (2.13 GHz)			Niagara (1.2 GHz)		Tilera (1.2 GHz)	
Hops	same die	same MCM	one hop	two hops	same die	one hop	two hops	same core	other core	one hop	max hops
loads											
Modified	81	161	172	252	109	289	400	3	24	45	65
Owned	83	163	175	254	-	-	-	-	-	-	-
Exclusive	83	163	175	253	92	273	383	3	24	45	65
Shared	83	164	176	254	44	223	334	3	24	45	65
Invalid	136	237	247	327	355	492	601	176	176	118	162
stores											
Modified	83	172	191	273	115	320	431	24	24	57	77
Owned	244	255	286	291	-	-	-	-	-	-	-
Exclusive	83	171	191	271	115	315	425	24	24	57	77
Shared	246	255	286	296	116	318	428	24	24	86	106

atomic operations: Compare & Swap (C), Fetch & Increment (F), Test & Set (T), Swap (S)

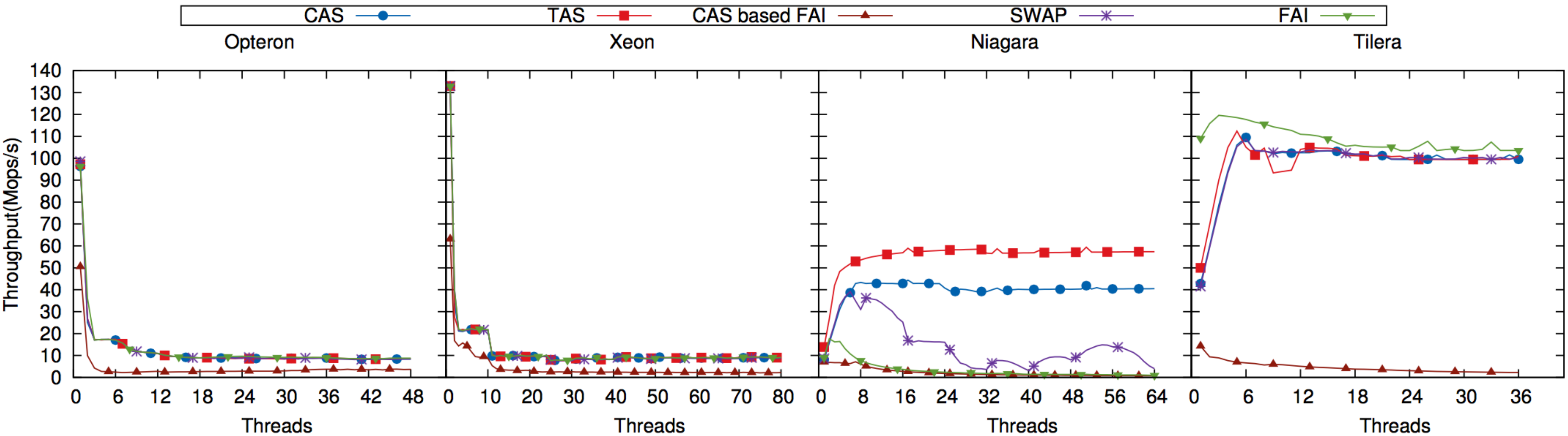
System	Opteron (2.1 GHz)				Xeon (2.13 GHz)			Niagara (1.2 GHz)								Tilera (1.2 GHz)							
Hops	same die	same MCM	one hop	two hops	same die	one hop	two hops	same core				other core				one hop				max hops			
Operation	all	all	all	all	all	all	all	C	F	T	S	C	F	T	S	C	F	T	S	C	F	T	S
Modified	110	197	216	296	120	324	430	71	108	64	95	66	99	55	90	77	51	70	63	98	71	89	84
Shared	272	283	312	332	113	312	423	76	99	67	93	66	99	55	90	124	82	121	95	142	102	141	115

Opteron: load latency independent of state

Xeon: load latency depends on state

Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP '13. ACM, New York, NY, USA, 33-48.

Variation in Performance of Atomics

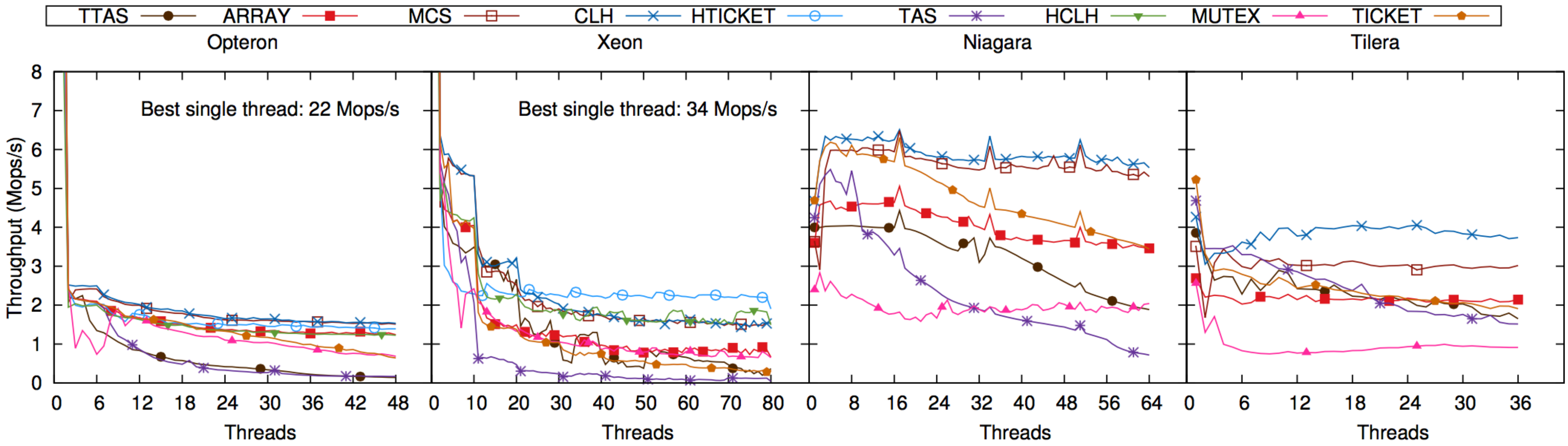


Throughput: Higher is better

Observations

- **relative performance of atomic primitives and cache operations varies widely in the hardware**
- **varying performance of locks is in part due to varying performance of atomic operations**

Lock Performance vs. Platform



Throughput: Higher is better

Observations

- —throughput on multi-socket systems is lower than on single chips
- —there is no universally best lock

Lock Acquisition vs. Previous Owner

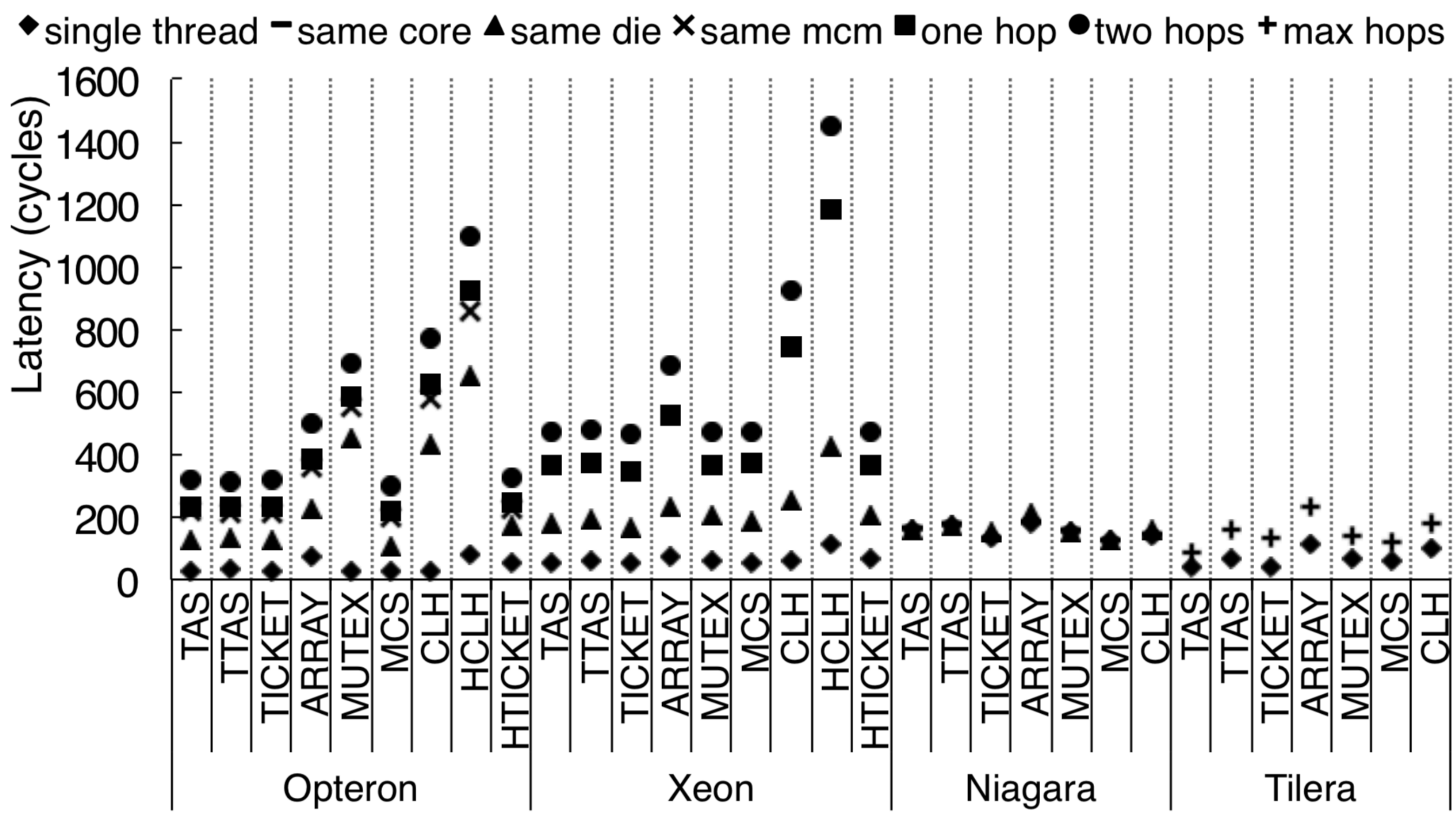


Figure 6: Uncontested lock acquisition latency based on the location of the previous owner of the lock.

Figure credit: Everything you always wanted to know about synchronization but were afraid to ask. D. Tudor, R. Guerraoui, and V. Trigonakis. In Proceedings of SOSP '13. ACM, New York, NY, USA, 33-48.

Impact of Contention on Performance

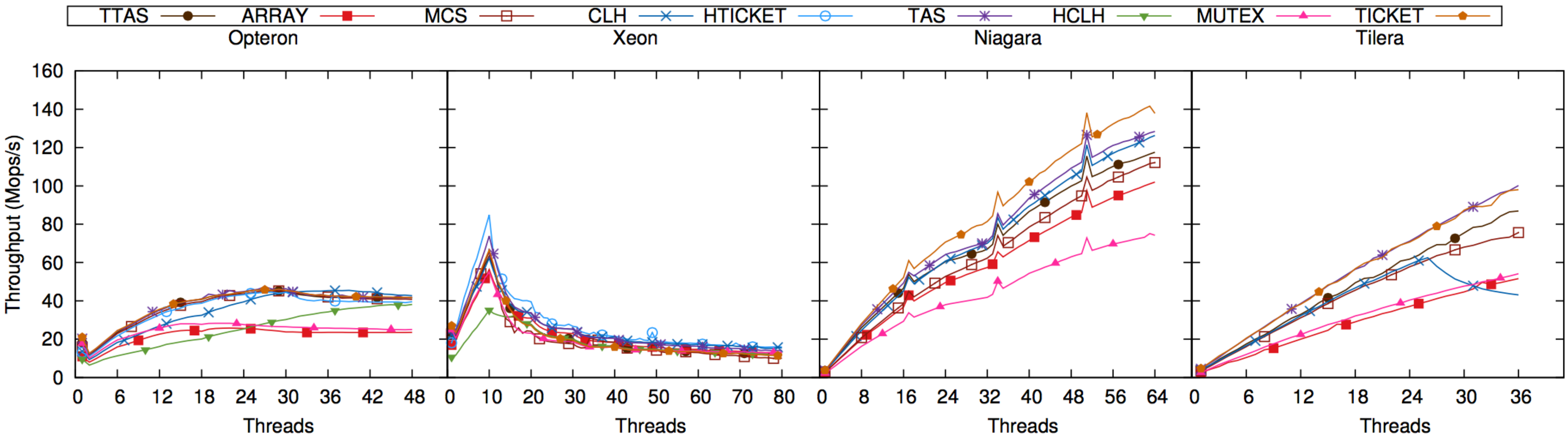


Figure 7: Throughput of different lock algorithms using 512 locks.

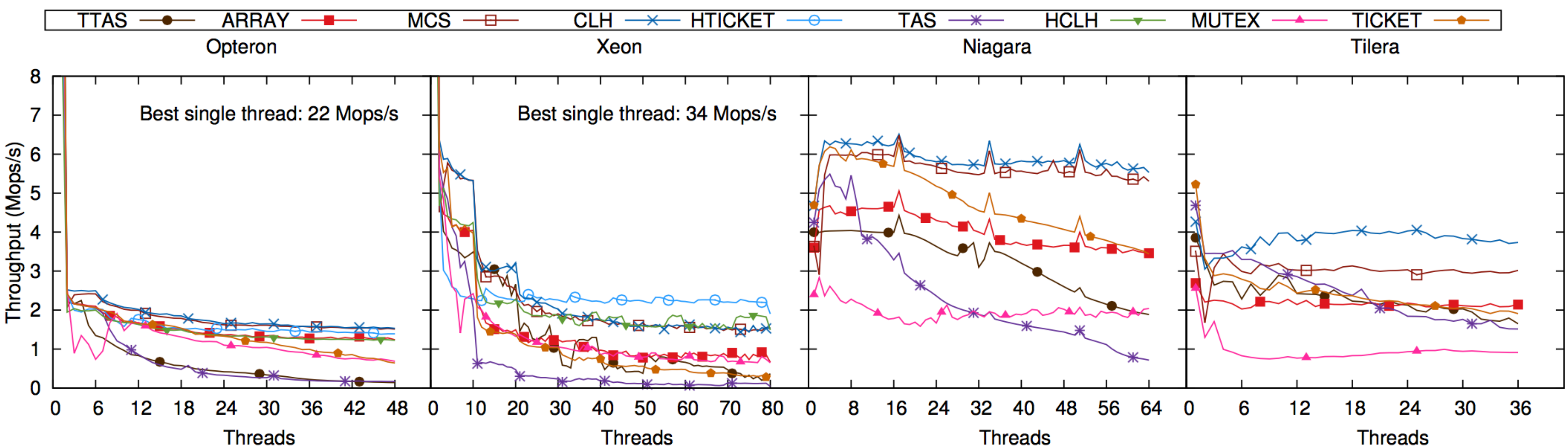


Figure 5: Throughput of different lock algorithms using a single lock.

Study Conclusions

- Crossing sockets is expensive
  - — 2x to 7.5x slower than intra-socket
  - —hard to avoid cross-socket communication
    - — e.g., Opteron: incomplete cache directory (no sharer info)
- Loads, stores can be as expensive as atomic operations
  - —non-local access can be a bottleneck
- Intra-socket non-uniformity matters (e.g., Tilera vs. Niagara)
  - —hierarchical locks scale better on non-uniform systems
- Simple locks can be effective
  - —ticket lock performs best in many cases
- There's no universally optimal lock
  - —optimal lock depends upon architecture and expected contention

An Unwise Conclusion?

Simple locks are powerful. Overall, an efficient implementation of a ticket lock is the best performing synchronization scheme in most low contention workloads. Even under rather high contention, the ticket lock performs comparably to more complex locks, in particular within a socket. Consequently, given their small memory footprint, ticket locks should be preferred, unless it is sure that a specific lock will be very highly contended.

Locks in Linux

Non-scalable Locks are Dangerous

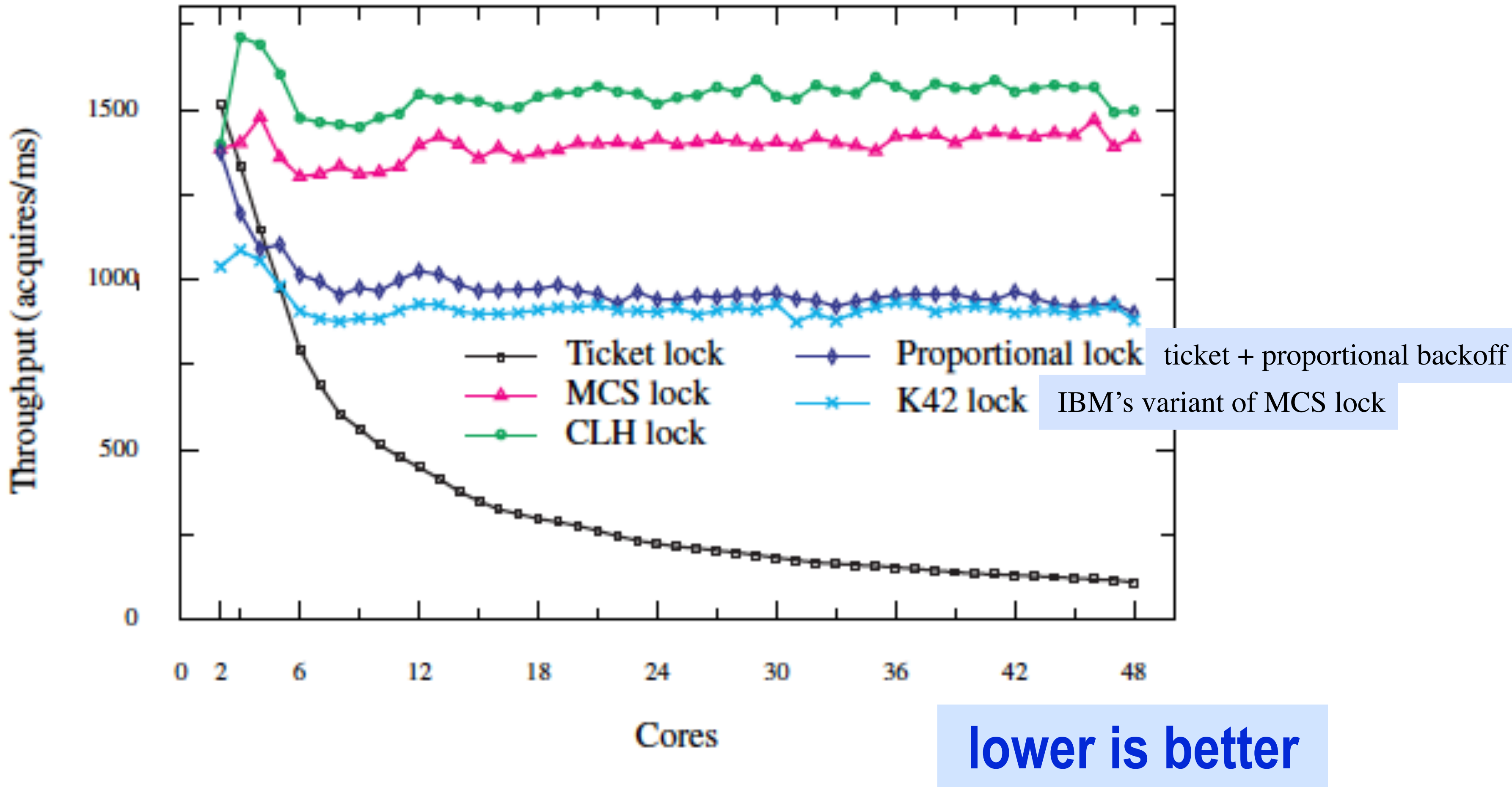


Figure 10: Throughput for cores acquiring and releasing a shared lock. Results start with two cores.

Non-scalable locks are dangerous Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. ***In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.***

Linux Benchmarks

Benchmark	Operation time (cycles)	Top lock instance name	Acquires per operation	Average critical section time (cycles)	% of operation in critical section
FOPS	503	`d_entry`	4	92	73%
MEMPOP	6852	`anon_vma`	4	121	7%
PFIND	2099 M	`address_space`	70 K	350	7%
EXIM	1156 K	`anon_vma`	58	165	0.8%

Figure 3: The most contended critical sections for each Linux microbenchmark, on a single core.

FOPS creates a single file and starts one process on each core. Each thread repeatedly opens and `closes` the file.

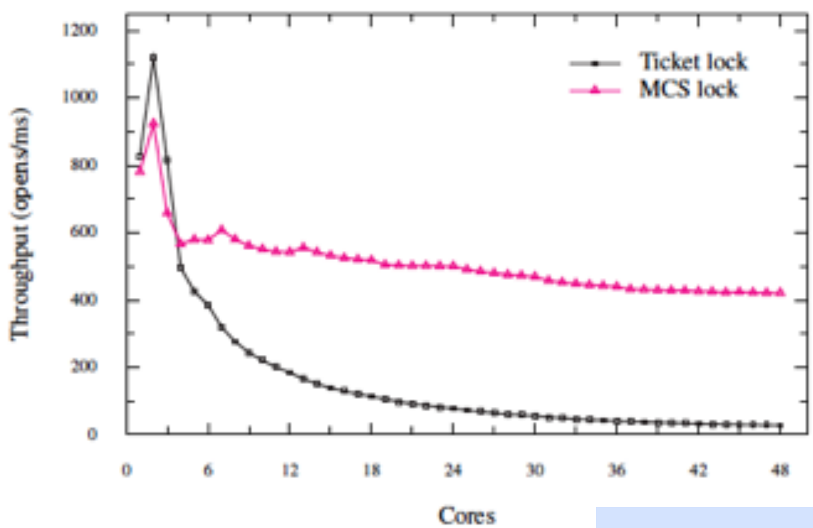
PFIND searches for a file by executing several instances of the GNU find utility. PFIND takes a directory and filename as input, evenly divides the directories in the first level of input directory into per-core inputs, and executes one instance of find per core, passing in the input directories. Before we execute the PFIND, we create a balanced directory tree so that each instance of find searches the same number of directories.

MEMPOP creates one process per core. Each process repeatedly `mmaps` 64 kB of memory with the `MAP_POPULATE` flag, then `munmaps` the memory. `MAP_POPULATE` instructs the kernel to allocate pages and populate the process page table immediately, instead of doing so on demand when the process accesses the page.

EXIM is a mail server. A single master process listens for incoming SMTP connections via TCP and forks a new process for each connection, which accepts the incoming message. We use the version of EXIM from MOSBENCH [3].

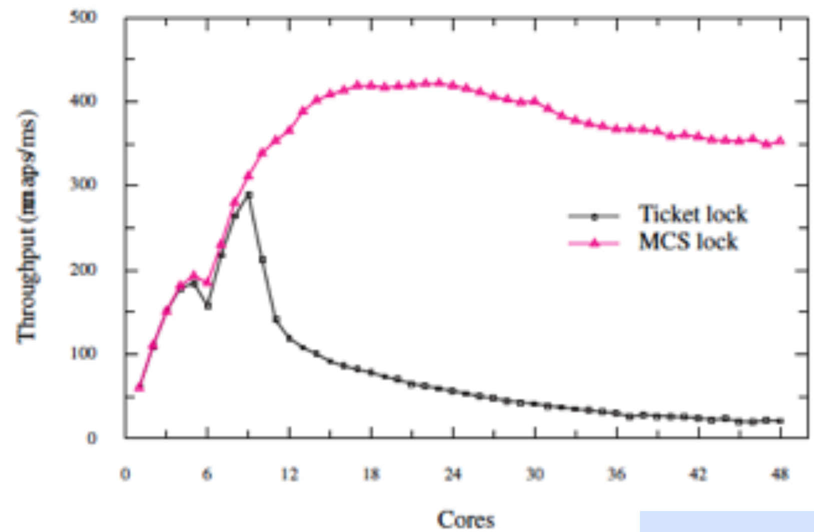
Non-scalable locks are dangerou Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. ***In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.***

MCS vs. Ticket Lock in Linux



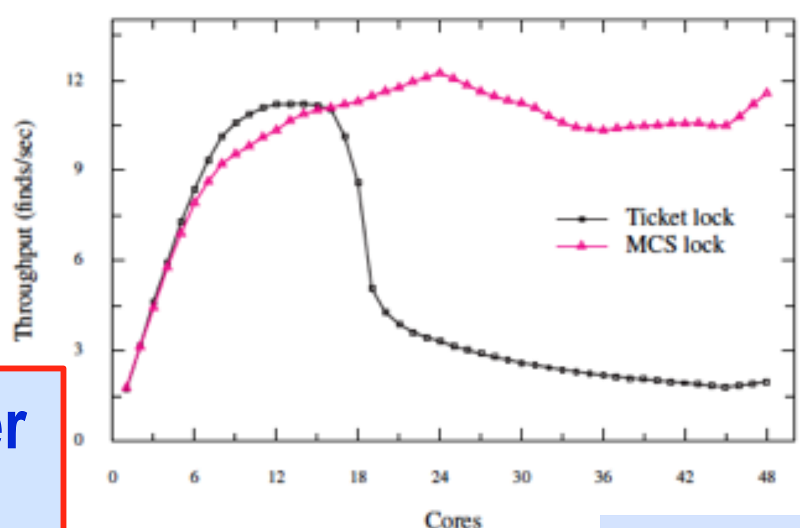
(a) Performance for FOPS.

73% (92)



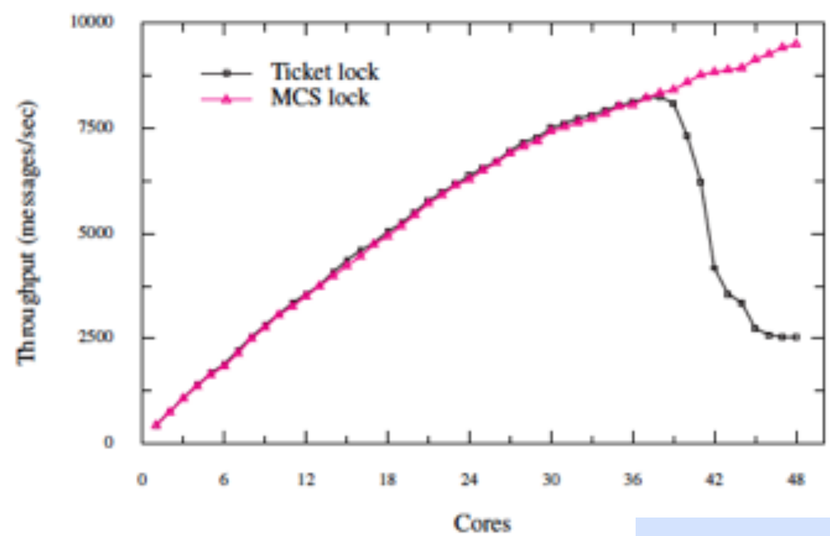
(b) Performance for MEMPOP.

7% (121)



(c) Performance for PFIND.

7% (350)



(d) Performance for EXIM.

0.8% (165)

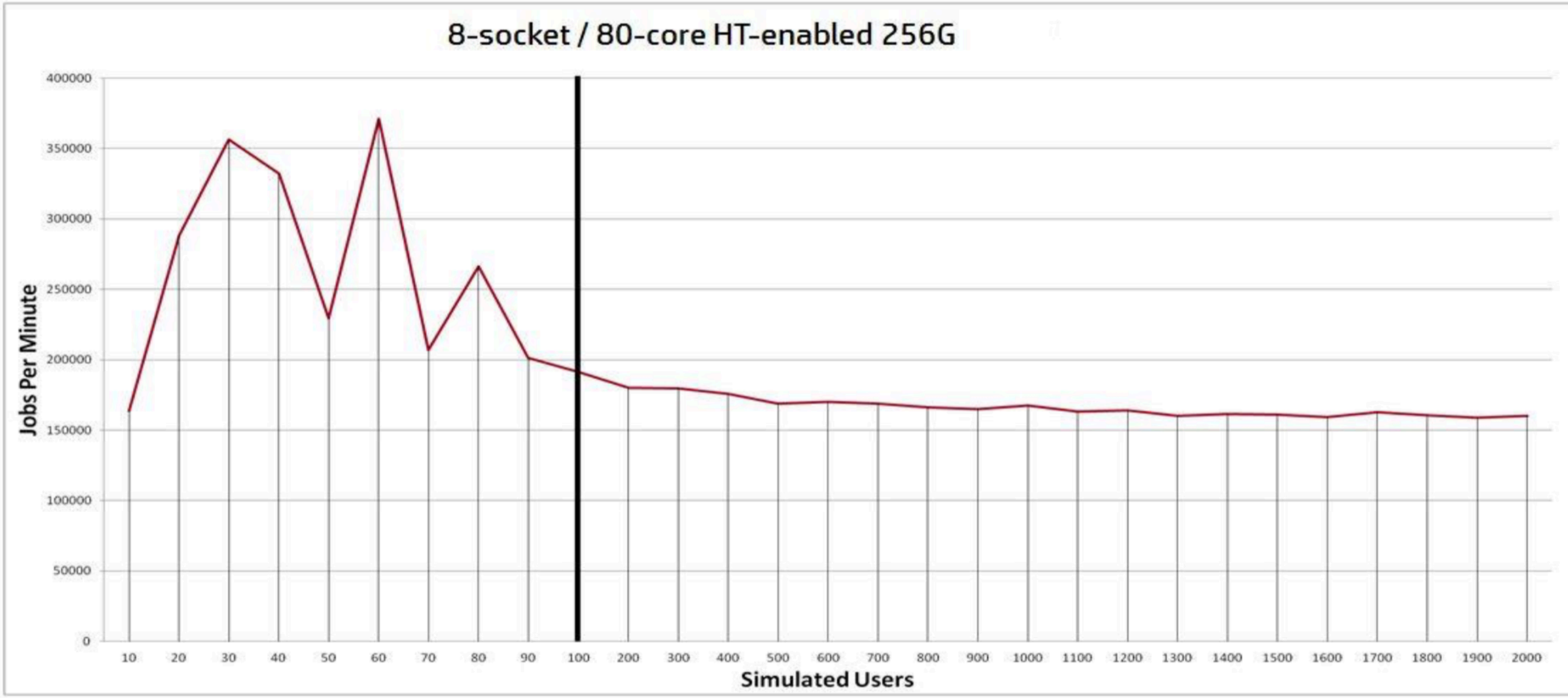
higher is better

Non-scalable locks are dangerous Silas Boyd-Wickizer, M. Frans Kaashoek, Robert Morris, and Nickolai Zeldovich. ***In the Proceedings of the Linux Symposium, Ottawa, Canada, July 2012.***

Lock Performance In Linux

Background

The AIM7 fserver workload* scales poorly on 8s/80core NUMA platform with a 2.6 based kernel



* The workload was run with ramfs.

Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. LinuxCon North America, Chicago, August 2014.

Why is Scaling Poor?

Analysis (1-2)

From the perf -g output, we find most of the CPU cycles are spent in file_move() and file_kill().

40 Users (4000 jobs)

```
+  9.40%  reaim  reaim           [.] add_int
+  6.07%  reaim  libc-2.12.so     [.] strncat
.....
-  1.68%  reaim  [kernel.kallsyms]  [k] _spin_lock
   - _spin_lock
     + 50.36% lookup_mnt
     + 7.45% __d_lookup
     + 6.71% file_move
     + 5.16% file_kill
     + 2.46% handle_pte_fault
```

Proportion of file_move() = 1.68% * 6.71% = 0.11%
Proportion of file_kill() = 1.68% * 5.16% = 0.09 %
Proportion of file_move() + file+kill() = 0.20%

400 users (40,000 jobs)

```
-  79.53%  reaim  [kernel.kallsyms]  [k] _spin_lock
   - _spin_lock
     + 34.28% file_move
     + 34.20% file_kill
     + 19.94% lookup_mnt
+  8.13%  reaim  [kernel.kallsyms]  [k] mutex_spin_on_owner
+  0.86%  reaim  [kernel.kallsyms]  [k] _spin_lock_irqsave
+  0.63%  reaim  reaim           [.] add_long
```

Proportion of file_move() = 79.53% * 34.28% = 27.26%
Proportion of file_kill() = 79.53% * 34.20% = 27.20%
Proportion of file_move() + file+kill() = 54.46%

This is significant spinlock contention!

Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. LinuxCon North America, Chicago, August 2014.

Why is Scaling Poor?

Analysis (2-2)

We use the ORC tool to monitor the coherency controller results

(ORC is a platform dependent tool from HP that reads performance counters in the XNC node controllers)

Coherency Controller Transactions Sent to Fabric Link (PRETRY number)

Socket	Agent	10users	40users	400users
0	0	17,341	36,782	**399,670,585**
0	8	36,905	45,116	**294,481,463**
1	0	0	0	49,639
1	8	0	0	25,720
2	0	0	0	1,889
2	8	0	0	1,914
3	0	0	0	3,020
3	8	0	0	3,025
4	1	45	122	1,237,589
4	9	0	110	1,224,815
5	1	0	0	26,922
5	9	0	0	26,914
6	1	0	0	2,753
6	9	0	0	2,854
7	1	0	0	6,971
7	9	0	0	6,897

- PRETRY indicates the associated read needs to be re-issued.
- We can see that when users increase, PRETRY on socket 0 increases rapidly.
- There is serious cache line contention on socket 0 with 400 users. Many jobs are waiting for the memory location on Socket 0 which contains the spinlock.
- PRETRY number on socket 0:
  400 users = 400M + 294M = 694M

18

Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. LinuxCon North America, Chicago, August 2014.

MCS vs. Ticket Lock in Linux

Removing Cache Line Contention

- Code snippet from the 2.6 based kernel for file_move() and file_kill():

```
extern spinlock_t files_lock;
#define file_list_lock()   spin_lock(&files_lock);
#define file_list_unlock() spin_unlock(&files_lock);
```

```
void file_move(struct file *file,
               struct list_head *list)
{
    if (!list)         return;
    file_list_lock();
    list_move(&file->f_u.fu_list, list);
    file_list_unlock();
}
```

```
void file_kill(struct file *file)
{
    if (!list_empty(&file->f_u.fu_list)) {
        file_list_lock();
        list_del_init(&file->f_u.fu_list);
        file_list_unlock();
    }
}
```

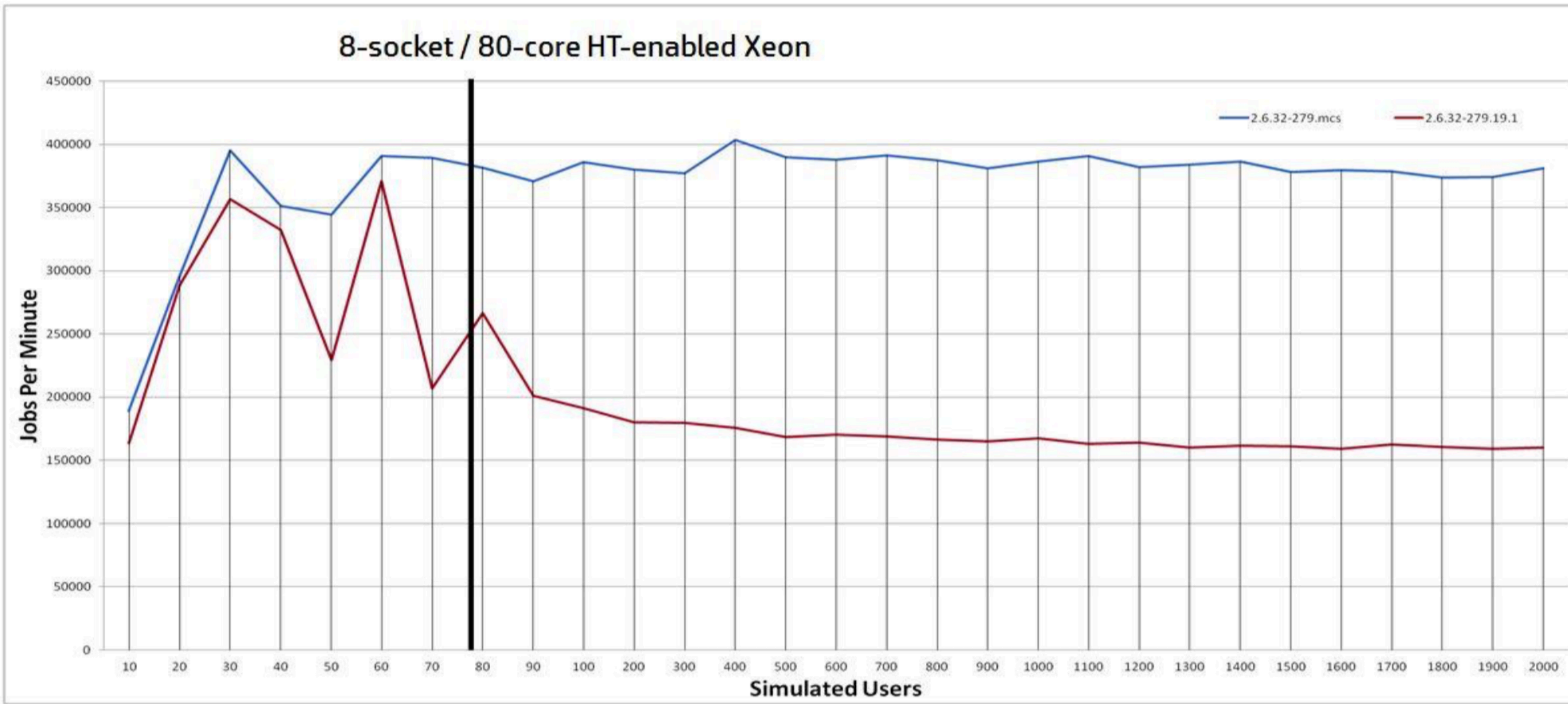
- Contention on this global spinlock is the cause of all the cache line contention
- We developed a prototype MCS/Queued spinlock to see its effect on cache line traffic
  - MCS/Queued locks are NUMA aware and each locker spins on local memory rather than the lock word
  - Implementation is available in the back-up slides
- No efforts were made to make this a finer grained lock

Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. LinuxCon North America, Chicago, August 2014.

MCS vs. Ticket Lock in Linux

Prototype Benchmark Results

Comparing the performance of the new kernel (blue line) vs. the original kernel (red line)



2.4x improvement in throughput with the MCS/Queued spinlock prototype!

20

Davidlohr Bueso and Scott Norton. An Overview of Kernel Lock Improvements. LinuxCon North America, Chicago, August 2014.

Lock Cohorting

Lock Cohorting

- Idea: use two levels of locks
  —global locks
  —local locks, one for each socket or cluster (NUMA node)
- First in socket to acquire local lock
  —acquire socket lock then the global lock
  —pass local lock to other waiters in the local node
  —eventually relinquish global lock to give other nodes a chance
- Recipe for NUMA-aware locks without special algorithms
- Cohorting can compose any kind of lock into a NUMA lock
  —augments properties of cohorted locks with locality preservation
- Benefits
  —reduces average overhead of lock acquisition
  —reduces interconnect traffic for lock and protected data

Global and Local Locks for Cohorting

- Global lock G
  - —thread-oblivious: acquiring thread can differ from releasing thread
  - —globally available to all nodes of the system
- Local lock S
  - —supports cohort detection
    - – a releasing thread can detect if other threads waiting
  - —records last state of release as global or local
- Once S is acquired
  - —local release → proceed to critical section
  - —global release → try to acquire G
- Upon release of S
  - —if NOT (may_pass_local OR alone) → release globally
  - —else → release locally

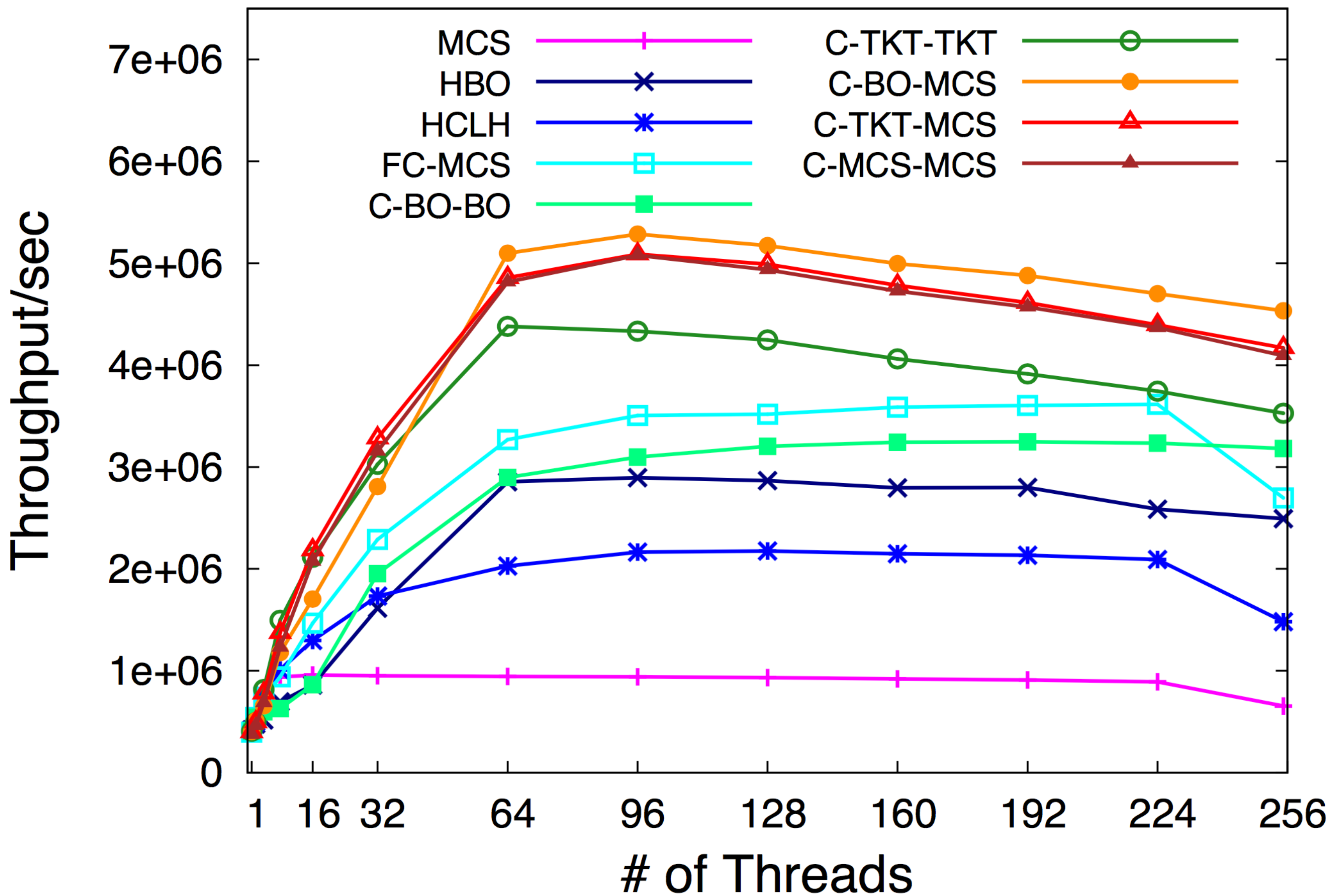
Global and Local Locks for Cohorting

- C-BO-BO lock
  - —Global backoff (BO) lock and local backoff locks per node
  - —requires additional cohort detection mechanism in local BO lock
- C-TKT-TKT lock
  - —Global ticket lock and local ticket (TKT) locks per node
- C-BO-MCS lock
  - —global backoff lock and local MCS lock
- C-MCS-MCS lock
- C-TKT-MCS lock
- Use of abortable locks in cohort designs needs extra features to limit aborting while in a cohort
  - —A-C-BO-BO lock
  - —A-C-BO-CLH lock (queue lock of Craig, Landin, & Hagersten)

Experiments

- Microbenchmark LBench is used as a representative workload
- LBench launches identical threads
- Each thread loops as follows
  —acquire central lock
  —access shared data in critical section
  —release lock
  —~4ms of non-critical work
- Run on Oracle T5440 series machine
  —256 hardware threads
  —4 NUMA clusters
- Evaluation shows that cohort locks outperform previous locks by at least 60%

Average Throughput vs. # of Threads

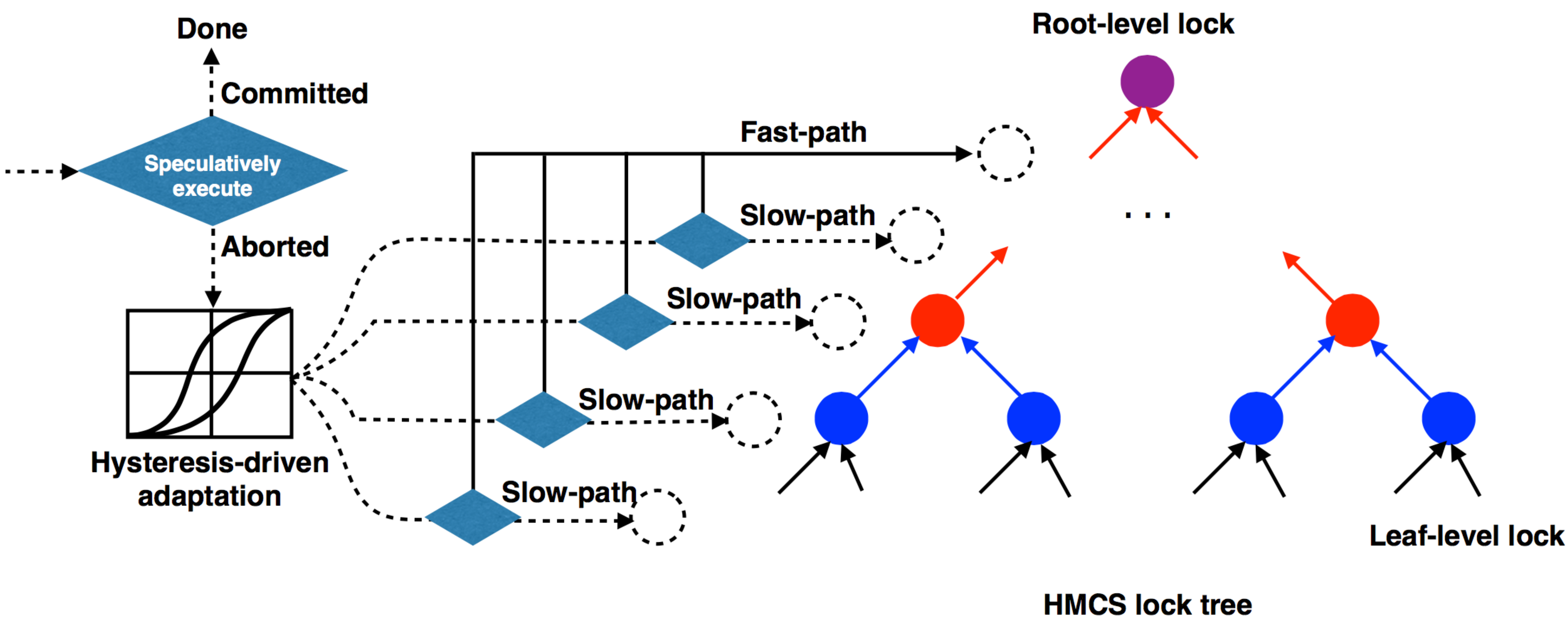


Conclusions: Cohorting is Useful

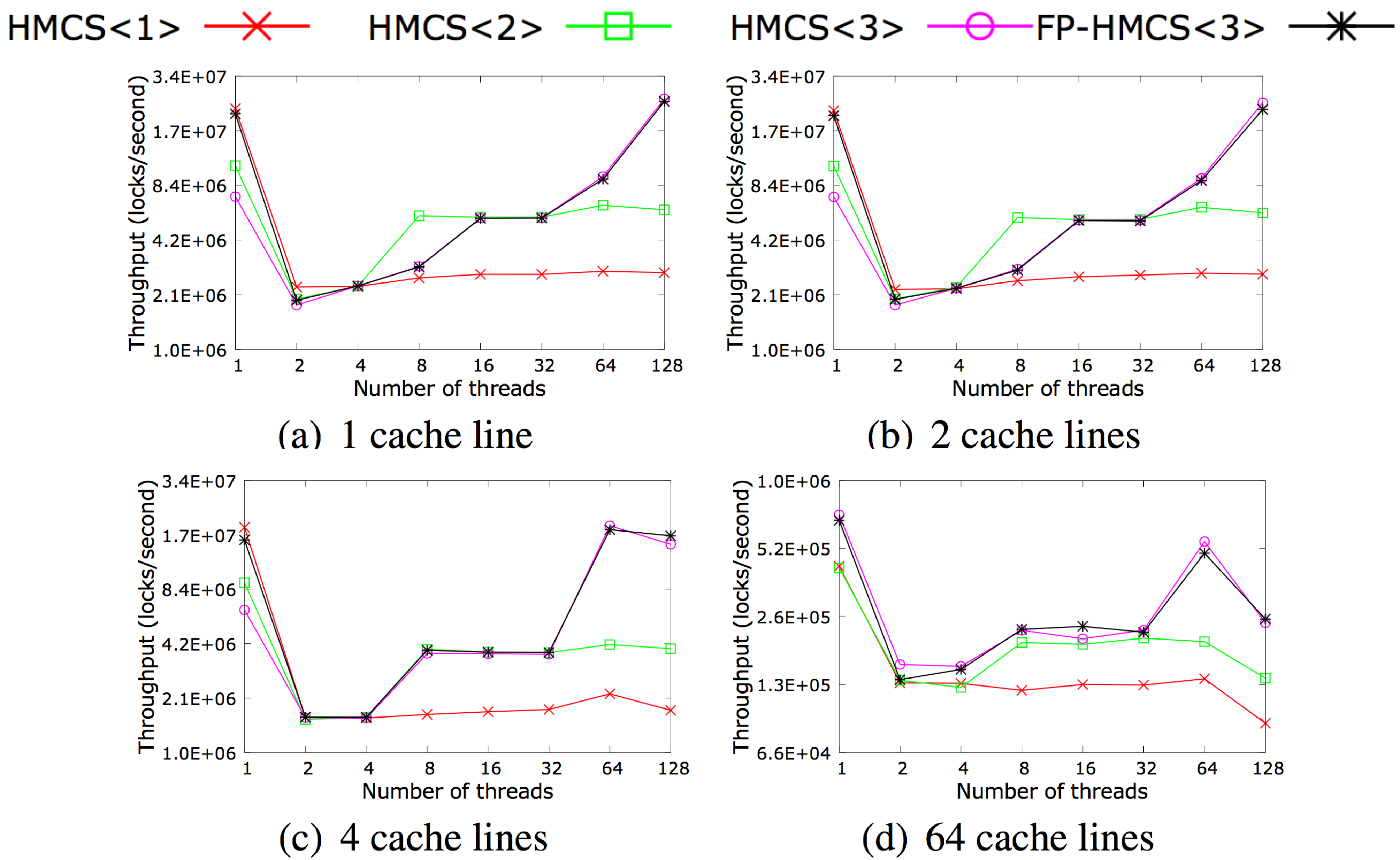
- **Useful design methodology**
  - **—no special locks required**
  - **—can be extended to additional levels of locality**
    - **e.g., tile based systems where locality is based on grid position**
    - **multiple levels of lock cohorts**
- **Cohort locks improve performance over previous NUMA aware lock designs**
- **Performance scaling with thread count is better with locality-preserving cohort locks**

New Work: Adaptive HMCS Lock

- Tree of MCS locks to exploit multiple levels of locality
- Fast path: directly acquire root if lock is available
- Hysteresis: adaptively select at which level to compete



Performance of AHMCS on Power 4-Socket



Throughput: Higher is better

Dice and Kogan's CNA Lock

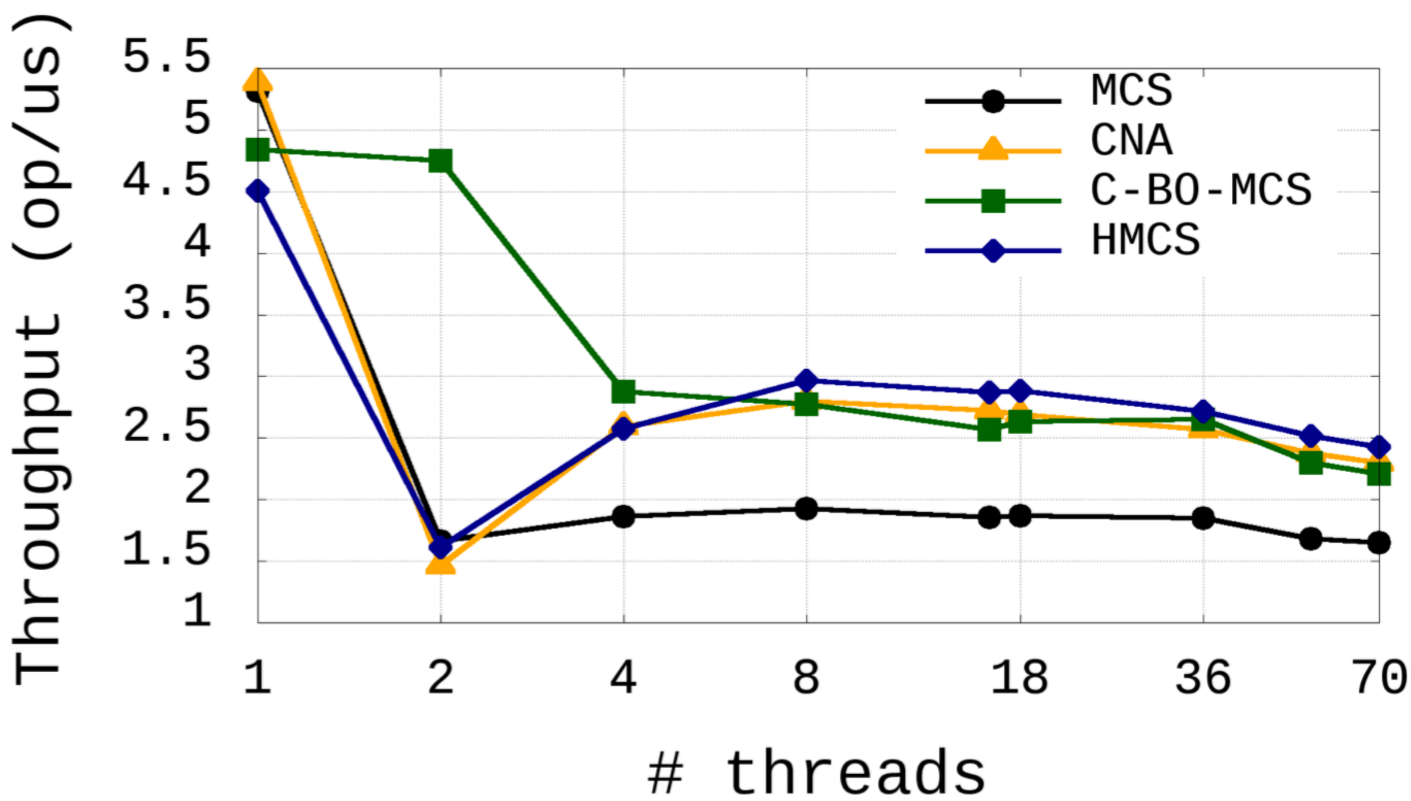


Figure 6. Total throughput for the key-value map microbenchmark.

Throughput: Higher is better

Dice and Kogan’s CNA Lock in Linux

Benchmark	Contended spin locks	Call sites
lock1_threads	files_struct.file_lock	__alloc_fd fcntl_setlk
lock2_threads	file_lock_context.flc_lock	posix_lock_inode
open1_threads	files_struct.file_lock	__alloc_fd __close_fd
	lockref.lock	dput d_alloc lockref_get_not_zero lockref_get_not_dead
open2_threads	files_struct.file_lock	__alloc_fd __close_fd

Table 1. Contention in the `will-it-scale` benchmarks.

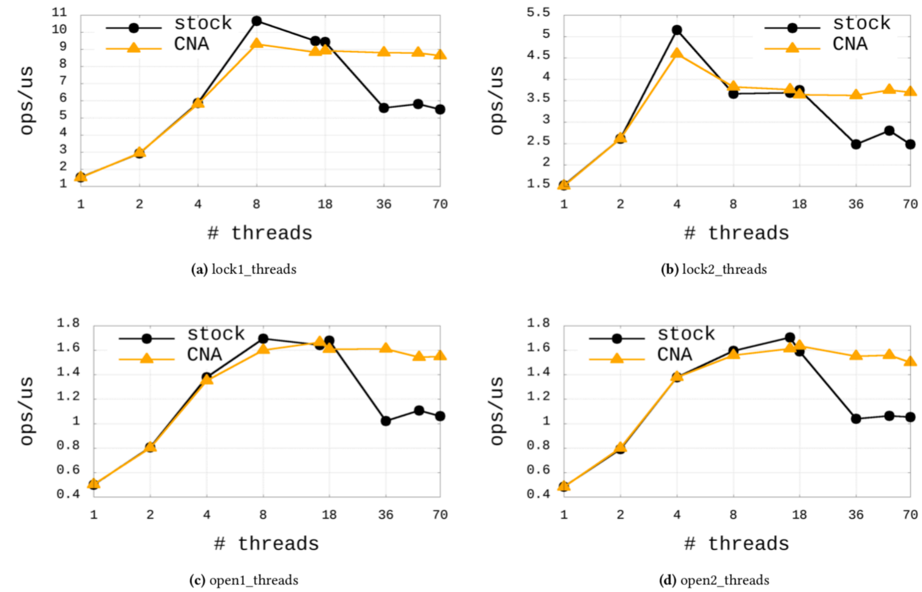


Figure 15. Performance results for the `will-it-scale` benchmarks.

Throughput: Higher is better

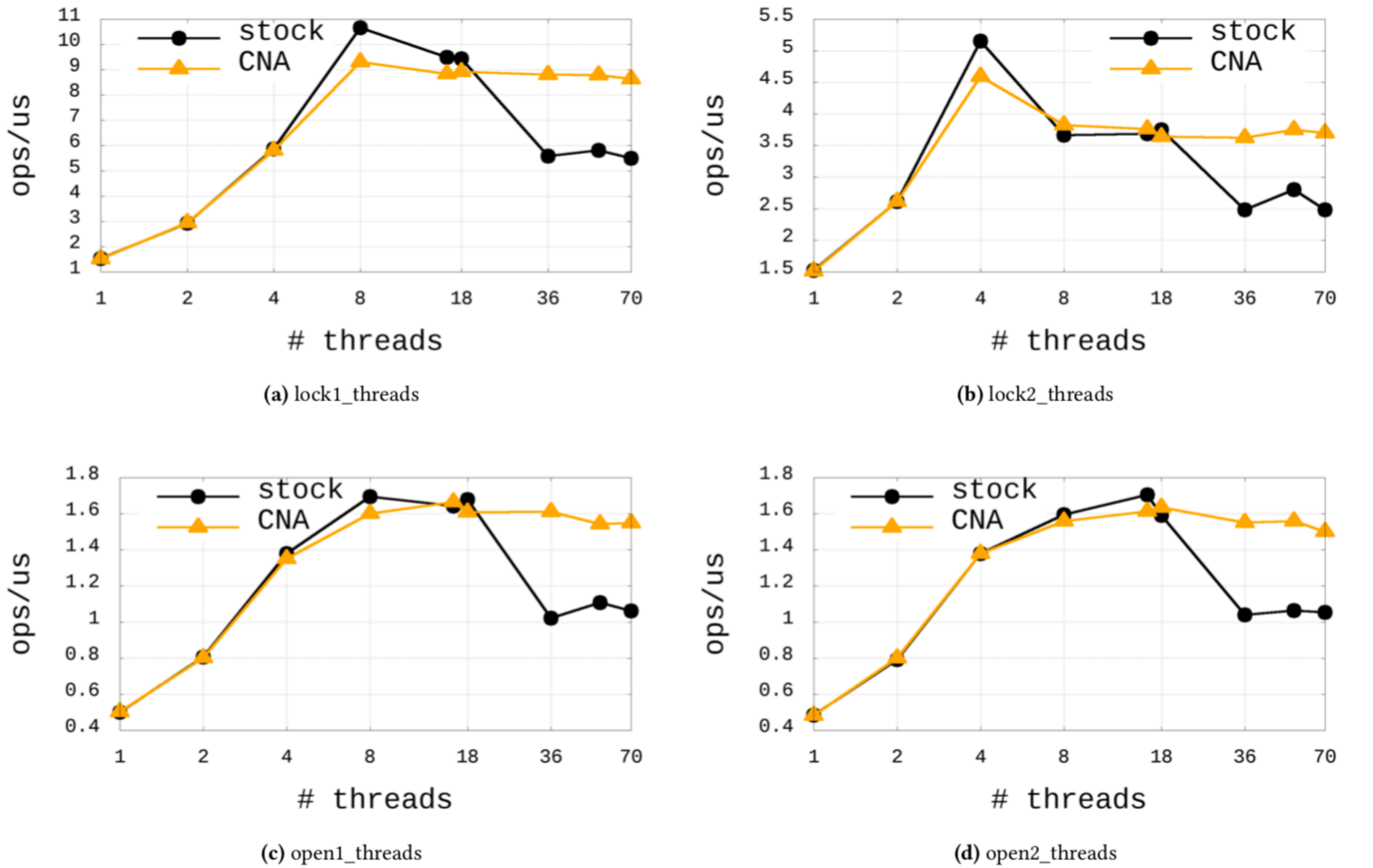
Dice and Kogan's CNA Lock in Linux

Benchmark	Contended spin locks	Call sites
lock1_threads	files_struct.file_lock	__alloc_fd fcntl_setlk
lock2_threads	file_lock_context.flc_lock	posix_lock_inode
open1_threads	files_struct.file_lock	__alloc_fd __close_fd
	lockref.lock	dput d_alloc lockref_get_not_zero lockref_get_not_dead
open2_threads	files_struct.file_lock	__alloc_fd __close_fd

Table 1. Contention in the will-it-scale benchmarks.

Throughput: Higher is better

Dice and Kogan’s CNA Lock in Linux



(a) lock1_threads (b) lock2_threads

(c) open1_threads (d) open2_threads

Figure 15. Performance results for the `will-it-scale` benchmarks.

Throughput: Higher is better