Malthusian Locks

Dave Dice
Oracle Labs
dave.dice@oracle.com

Abstract

Applications running in modern multithreaded environments are sometimes *overthreaded*. The excess threads do not improve performance, and in fact may act to degrade performance via *scalability collapse*, which can manifest even when there are fewer ready threads than available cores. Often, such software also has highly contended locks. We leverage the existence of such locks by modifying the lock admission policy so as to intentionally limit the number of distinct threads circulating over the lock in a given period. Specifically, if there are more threads circulating than are necessary to keep the lock saturated (continuously held), our approach will selectively cull and passivate some of those excess threads. We borrow the concept of *swapping* from the field of memory management and impose *concurrency restriction* (CR) if a lock suffers from contention. The resultant admission order is unfair over the short term but we explicitly provide long-term fairness by periodically shifting threads between the set of passivated threads and those actively circulating. Our approach is palliative, but is often effective at avoiding or reducing scalability collapse, and in the worst case does no harm. Specifically, throughput is either unaffected or improved, and unfairness is bounded, relative to common test-and-set locks which allow unbounded bypass and starvation [1]. By reducing competition for shared resources, such as pipelines, processors and caches, concurrency restriction may also reduce overall resource consumption and improve the overall load carrying capacity of a system.

Categories and Subject Descriptors D.4.1 [*Operating Systems*]: Mutual Exclusion

General Terms Performance, experiments, algorithms

Keywords Concurrency, threads, caches, multicore, locks, mutexes, mutual exclusion, synchronization, contention, scheduling, admission order, admission control, spinning, fairness

1. Introduction

The scaling collapse phenomenon mentioned above arises variously from communication and coordination overheads or from competition for any one of a number of shared resources. This paper focuses on the latter – we explore the etiology of scaling collapse via resource competition in more detail below. For example, one such resource is the shared last-level cache (LLC) on a single-socket system. All the cores on the socket compete for residency in the LLC, and concurrent requests from those cores may cause destructive interference in the LLC, continuously eroding the residency of the data from any one core.

The effect is similar to that of *thrashing* as described in Denning's *working set* model of memory pressure [19]. A system is said to thrash when memory is overcommitted and the operating system spends an inordinate amount of time servicing page faults, reducing overall progress. The solution in that context is *swapping* – the transient deactivation of some subset of the concurrently running programs. The medium-term scheduler responds to excessive paging and potential thrashing by swapping out selected "victim" processes until the thrashing abates. This closely models our approach where we transiently deactivate excess contending threads that do not contribute to improved throughput. CR responds to contention instead of memory pressure. We extend Denning's ideas from memory management to locks, defining the *lock working set* (LWS) as the set of distinct threads that have acquired a given lock in some time interval. We use the ordinal acquisition time of the lock to define the interval instead of wall-clock time. Suppose threads A, B, C, D and E contend for lock L and we have an admission order (also called the *admission history*) of $A\ B\ C\ A\ B\ C\ D\ A\ E$ for admission times $0-8$, respectively. The LWS for L for the period $0-5$ inclusive is threads $A\ B\ C$ and the *lock working set size* (LWSS) for the period is thus 3 threads.

CR may be unfair over the short-term, but our admission policies intentionally impose long-term fairness [2]. To help

Robert Malthus [70] argued for population control, cautioning that societies would collapse as increasing populations competed for resources. His dire predictions did not come to pass as food production – which had previously been stagnant – improved to keep pace with population growth.

[1] *Bypass* occurs when a thread T acquires a lock but there exist other waiting threads that arrived earlier than T.

[2] Fairness measures how admission order deviates from arrival order or from strict FIFO order.

EuroSys '17, April 23-26, 2017, Belgrade, Serbia

 ISBN 978-1-4503-4938-3/17/04…15.00

DOI: http://dx.doi.org/10.1145/3064176.3064203

gauge the trade-off between throughput and fairness we introduce two metrics for short-term fairness. For the first metric, we partition the admission history of a lock into W-sized disjoint abutting windows, compute the LWSS of each window, and take the average of those values. We refer to this value as the *average LWSS* over the measurement interval – it gives an intuitive measure of short-term fairness. In this paper we use a window size of 1000 acquisitions. The second measure of short-term fairness is the *median time to reacquire* (MTTR), computed over the entire acquisition history. *Time to reacquire* is determined at admission time, and is the number of admissions since the current thread last acquired the lock. Time to reacquire is analogous to *reuse distance* in memory management.

CR acts to reduce the number of distinct threads circulating through the lock over short intervals and thus tends to reduce the LWSS, while still providing long-term fairness. The CR admission policy must also be *work conserving* and never under-provision the lock. It should never be the case that the critical section remains intentionally unoccupied if there are waiting or arriving threads that might enter – if such threads exist, then one will promptly be *enabled* to do so.

As noted above, CR partitions and segregates the set of threads attempting to circulate over the lock into the ACS (active circulating set) and the PS (passive set) [3]. Threads in the ACS circulate normally. We desire to minimize the size of the ACS (and thus the LWSS) while still remaining work conserving, ensuring there are sufficient threads in the ACS to saturate the lock – and that the critical section enjoys maximum occupancy – but no more. Surplus threads are culled from the ACS and transferred into the PS where they remain quiesced. Conversely a deficit in the ACS prompts threads to be transferred from the PS back into the ACS as necessary to sustain saturation. To ensure long-term fairness our approach periodically shifts threads between the ACS and PS. Ideally, and assuming a steady-state load, at most one thread in the ACS will be waiting at any moment, reducing wait times for ACS members. That is, at unlock-time we expect there is typically just one thread from the ACS waiting to take the lock. Intuitively, threads in the ACS remain "enabled" and operate normally while threads in the PS are "disabled" and do not circulate over the lock. Threads sequestered in the PS typically busy-wait (spin) in a *polite* [22] fashion on a thread-local flag, or block in the operating system, surrendering their CPU. (Such polite waiting reduces the resources consumed by the waiting threads, and may allow other threads to run faster). Our approach constrains and regulates the degree of concurrency over critical sections guarded by a contended lock in order to conserve shared resources such as residency in shared caches. Specifically, we minimize over the short term the number of distinct threads acquiring the lock and transiting the critical section.

For instance assume a simplified execution model with 10 threads contending for a common lock. The threads loop as follows: acquire the lock; execute the critical section (CS); release the lock; execute their respective non-critical section (NCS). Each such iteration reflects *circulation* over the lock. In our example the NCS length is 5 microseconds and the CS length is 1 microsecond. For the purposes of explication we assume an ideal lock with no administrative overheads. In this case we reach saturation – Amdahl peak speedup – at 6 threads. At any given time 1 thread is in the CS and 5 execute in their respective NCS. Thus under ideal CR the ACS would have 6 threads and 4 of the 10 threads would reside in the PS, transiently made passive. The 6 circulating threads in the ACS would enjoy a round-robin cyclic admission schedule.

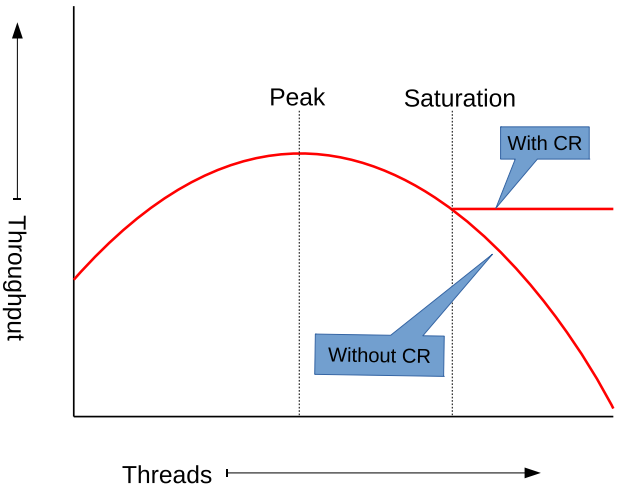

Figure 1: Impact of Concurrency Restriction

2. Scalability Collapse

The scalability collapse phenomenon involves competition for shared hardware resources. A classic example is residency in a shared LLC. As more distinct threads circulate over the lock in a given period, cache pressure and miss rates increase. Critically, as the cache is shared, residency of the data accessed by a given thread decays over time due to the action of other concurrently running threads that share the LLC. The application may start to thrash in the LLC and become memory-bound. As the LLC miss rate rises from cache pressure, contention for the DRAM channels increases, making LLC misses even more expensive and compounding a deleterious effect. CR can serve to reduce such destructive interference in shared caches. By reducing the number of threads circulating over the short term, we reduce cache pressure and retain residency for longer periods, reducing the miss rate and DRAM channel congestion.

Figure 1 depicts the impact of CR via an idealized aggregate throughput graph. Thread count appears on the X-axis and aggregate throughput on the Y-axis. In our depiction there are more logical CPUs than threads, so preemption is not a factor. Such concave scaling graphs are common in practice, and reflect scalability collapse [15, 66] [4]. We show that a properly designed lock with CR can also act to re-

[3] The ACS corresponds to the *balance set* in the working set model, and the PS corresponds to the set of swapped and inactive processes.

[4] Lock implementations themselves are sometimes a causative factor for collapse, for instance via induced coherence traffic on lock metadata or

duce collapse stemming from competition for shared hardware resources. Assume an execution model with one contended lock L, where each thread repeatedly acquires L, executes a critical section, releases L, and then executes a non-critical section. All threads start at the same time and run concurrently throughout the measurement interval. Throughput on the Y-axis reflects the total number of iterations completed by the threads in the measurement interval. Maximum throughput appears at the threading level corresponding to *Peak*, representing the key inflection point where performance drops as thread counts increase. Beyond *peak*, additional threads do not contribute to performance, and in fact may degrade performance. This behavior is also called *retrograde scaling* [42]. *Saturation* reflects the minimum threading level where there is always at least one waiting thread when the owner releases L – the onset of sustained contention where the lock is expected to be held continuously (or nearly so, for test-and-set locks in transition) and the critical section is continuously occupied. We say threads beyond *saturation* are *excess* or surplus threads – threads not necessary to achieve saturation. The thread count for *peak* will always be less than or equal to *saturation*. CR can begin to operate and provide benefit when the thread count exceeds *saturation*. The value for *peak* is imposed by platform architectural factors, overall system load, and offered application load, and is unrelated and orthogonal to *saturation* [5]. The value for *peak* is not usually amenable to analytic calculation, and, when required, is determined empirically.

We note two regions of interest. First, when the thread count is less than *saturation*, CR would be ineffective and does not operate. CR does not impact performance in this region, providing neither harm nor benefit. Second, when the thread count exceeds *saturation*, CR can operate, ideally avoiding the subadditive scalability collapse evident in the graph when CR is not enabled. CR acts by clamping the effective thread count – over the short term – to *saturation*. Beyond *saturation* and under fixed load we expect the LWSS to always be greater than or equal to *saturation*.

3. Taxonomy of Shared Resources

We provide a limited taxonomy of inter-thread shared resources that are subject to competition and are amenable to conservation via CR. Each of the following shared resources identifies a potential mode of benefit for CR.

- Socket-level resources
  - LLC residency and DRAM channel bandwidth
  - Thermal and energy headroom – enablement of Turbo mode[68]
- Core-level resources
  - Pipeline and floating point unit availability
  - Core-private L1 and L2 residency – cache pressure
  - Translation lookaside buffer (TLB) residency
- System-wide resources such as logical CPUs

Competition for core-level resources such as pipelines typically starts to manifest when the number of ready threads exceeds the number of cores, and more than one thread is running on a core. The onset of competition for socket-level resources may start at lower thread counts. Contention for CPUs occurs when the number of ready threads exceeds the number of logical CPUs, where preemption (multiprogramming) starts.

As noted previously, a key socket-level shared resource is LLC residency. Suppose we have a contended lock that is fully saturated. In this mode the critical section duration solely dictates throughput [33]. Data accessed in non-critical sections is thread-private and multiple independent non-critical sections may execute concurrently with a single CS. NCS accesses displace and evict critical data [6]. As the set of threads circulating over the lock grows, the total non-critical footprint increases, and we find more cache pressure in the communal LLC. In turn, the critical section suffers more LLC misses, increasing the duration of the CS and decreasing throughput over the contended lock. CR can afford benefit in this circumstance by restricting the set of circulating threads, reducing cache pressure and thus increasing throughput compared to a perfectly fair FIFO lock.

We next provide a detailed example to motivate the benefit of CR on a single-socket SPARC T5 processor where the shared LLC (L3 cache) is 8MB. We have a customer database that is 1MB, and each CS operation will access a record in that database. Each record resides on a single cache line. An individual CS will access only one record, but over time most records will be accessed repeatedly by subsequent operations. (The CS may be "short" in average duration but "wide" in the sense that a sequence of CS operations will eventually access a large fraction of the records). We have 16 threads, and on an otherwise unloaded system the NCS duration is 4 times that of the CS duration. The $(NCS + CS)/CS$ ratio is such that only 5 threads are needed to fully saturate the lock and provision the ACS. Furthermore, the NCS footprint of each thread is 1MB. Even though an individual NCS operation might be short, over time a thread will access all 1MB of its thread-private data. Recall that the CS data is shared and the NCS data is per-thread and thread-private. Under a classic FIFO MCS lock [57], all 16 threads will circulate over the lock in round-robin cyclic order. The total footprint is 17MB : $(16\ threads * 1MB/thread) + 1MB$ for the CS, exceeding the 8MB capacity of the LLC. The NCS operations will erode and decay the residency of the CS data, slowing execution of the CS, and degrading overall throughput. But with CR the lock subsystem is able to limit the size of the ACS to 5 threads. In this mode, the total short-term footprint is 6MB : $(5\ threads * 1MB/thread) + 1MB$ for the CS. The total footprint – the CS data plus the NCS data of the ACS threads – fits comfortably within the LLC.

where lock algorithmic overheads increase with the number of contending or participating threads.

[5] Contended locks just happen to be a convenient and opportunistic vehicle with which to restrict concurrency.

[6] CS invocations under the same lock typically exhibit *reference similarity*: acquiring lock L is a good predictor that the critical section protected by L will access data that was accessed by recent prior critical sections protected by L. That is, CS invocations tend to access data accessed by prior CS invocations, exhibiting inter-CS inter-thread locality and reuse.

Consequently, the NCS instances do not erode CS residency, the CS does not suffer from misses arising from destructive interference in the LLC, and throughput is improved. CR reduces cache pressure and in particularly on CS data. "Hot" threads – those that have run recently and have residual LLC residency – tend to remain "hot".

Another socket-level shared and rationed resource is thermal and energy headroom. By running fewer threads in a given interval relative to other locks, CR may reduce energy use and heat dissipation. Furthermore, by quiescing threads in the PS and allowing more processors to enter and remain in deeper low-power *sleep states* while idle, our approach can enable *turbo mode* [28, 68] for the remaining active threads – critically including the lock holder – accelerating their progress and improving throughput.

The *waiting policy* of a lock implementation (discussed below) defines how a thread waits for admission, and can have a significant impact on competition for core-level resources such as pipelines, socket-level resources such as thermal and energy headroom, and global resources such as logical CPUs.

4. The MCSCR lock algorithm

We now describe the implementation of **MCSCR** – a classic MCS lock [57] modified to provide CR by adding an explicit list for members of the PS [7]. At unlock-time, if there exist any intermediate nodes in the queue between the owner's node and the current tail, then we have surplus threads in the ACS and we can unlink and excise one of those nodes and transfer it to the head of the passive list where excess "cold" threads reside. This constitutes the culling operation. Conversely, at unlock-time if the main queue is empty except for the owner's node, we then extract a node from the head of the passive list, insert it into the main queue at the tail, and pass ownership to that thread, effectively transferring an element from the PS back into the ACS. This ensures MCSCR is work conserving and provides progress and liveness. The element at the head of passive list is the most recently arrived member of the PS. Absent sufficient contention, MCSCR operates precisely like classic MCS. MCSCR directly edits the MCS chain to shift threads back and forth between the main chain and the explicit list of passivated threads . The ACS list is implicit, while the PS – the excess list – is explicit.

To ensure long-term fairness, the unlock operator periodically selects the tail T of the PS as the successor and then grafts T into the main MCS chain immediately after the lock-holder's element, passing ownership of the lock to T. Statistically, we cede ownership to the tail of the PS – which is the least recently arrived thread – on average once every 1000 unlock operations. We use a thread-local Marsagalia xor-shift pseudo-random number generator [56] to implement Bernoulli trials with probability $P = 0.001$. The probability parameter is tunable and reflects the trade-off between fairness and throughput. Transferring a thread from the PS into the ACS typically results in some other member of the ACS being displaced and shifted into the PS in subsequent culling operations.

Culling acts to minimize the size of the ACS. Under fixed load, aggressive culling causes the system to devolve to a desirable state where there is at most one member of the ACS waiting to acquire the lock. In this state, the ACS consists of that one waiting thread, the current owner of the lock, and a number of threads circulating through their respective non-critical sections. The size of the ACS is determined automatically and is not a tunable parameter. At unlock-time, the owner will usually pass ownership of the lock to that waiting thread. Subsequently, some member of the ACS will complete its non-critical section and wait for the lock. In this mode, admission order is effectively cyclic over the members of the ACS.

All changes to support MCSCR are implemented in the unlock path; the MCS lock operator remains unchanged. Operations on the PS occur within the unlock operator while the MCS lock is held – the PS is protected by the MCS lock itself. This artificially increases the length of the critical section, but the additional manipulations are short and constant-time.

5. Lock Design Fundamentals

5.1 Waiting Policies

The choice of waiting policy used by a lock implementation influences competition for CPUs, pipelines and thermal headroom, making the selection of a waiting policy critical for CR. The waiting policy also dictates key latencies, further informing our design. We identify a number of commonly used policies:

Unbounded spinning

Classic MCS and test-and-set spin locks (TAS locks)[3] use unbounded spinning, also called busy-waiting or polling. Waiting threads simply loop, re-checking the variable of interest. While unbounded spinning appears often in academic literature, actual deployed software generally avoids indefinite spinning. At some point a spinning thread is expected to deschedule itself. While convenient and simple, unbounded spinning can interfere with the performance of other threads on the system by consuming pipeline resources. Spinning also expends energy and consumes available thermal headroom, possibly to the detriment of sibling cores that might otherwise enjoy turbo mode acceleration. In addition, a spinning thread occupies a processor, possibly prohibiting some other ready thread from running in a timely fashion. (In fact spinning threads might wait for the lock holder which has itself been preempted.) If there are more ready threads than logical CPUs, then preemption by the kernel would eventually ensure those other threads run, but those ready threads may languish on dispatch queues until the spinners exhaust their time slice. Typical quanta durations far exceed the latency of a voluntary context switch. Despite those concerns,

[7] Under classic MCS, arriving threads append an element to the tail of the list of waiting threads and then busy-wait on a flag within that element. The lock's `tail` variable is explicit and the head – the current owner – is implicit. When the owner releases the lock it reclaims the element it originally enqueued and sets the flag in the next element, passing ownership.

An extended version of this paper is available at http://arxiv.org/abs/1511.06035.

spinning remains appealing because it is simple and the lock handover latency (discussed below) – absent preemption – is low.

Spinning can be made more *polite* to sibling threads by using the `PAUSE` instruction on x86, or the `RD CCR,G0` idiom, a long-latency no-op, on SPARC. These instructions transiently cede pipeline resources to siblings – logical CPUs that share the core with the spinning thread – allowing those siblings to run faster [8]. Such instructions may also reduce power usage.

Parking

Our lock implementations employ a *park-unpark* infrastructure for voluntary context switching. The park-unpark facilities allows a waiting thread to surrender its CPU directly to the operating system while the thread waits for a contended lock. The *park* primitive blocks the caller, rendering itself ineligible to be scheduled or dispatched onto a CPU. A corresponding *unpark(T)* system call wakes or resumes the target thread T, making it again ready for dispatch and causing control to return from park if T was blocked. An unpark(T) operation can occur before the corresponding park call by T, in which case park returns immediately and consumes the pending unpark action. Waiting for a lock via parking is polite in the sense that the waiting thread can make its CPU immediately available to other ready (runnable) threads. The Solaris operating system exposes *lwp_park* and *lwp_unpark* system calls while the *futex* facility can be used to park and unpark threads on Linux. The park-unpark facility is often implemented via a restricted-range semaphore, allowing values only of 0 (neutral) and 1 (unpark pending). The park-unpark interface moves the decision of *which* thread to wake out of the kernel and into the user-space lock subsystem, where explicit lists of parked threads are typically maintained.

Parking suspends the calling thread and voluntarily surrenders the CPU on which the caller ran, making it immediately available to run other ready threads. If no other threads are ready, then the CPU may become idle and be able to drop to deeper sleep states, reducing power consumption and potentially enabling other ready threads on the same chip to run at faster speeds via turbo mode [9]. Parking also reduces competition for intra-core pipeline resources, and promotes fusion. In turn, other threads – possibly including the lock holder running in its critical section – may run faster, improving scalability. Parking also allows the operating system to rebalance the set of running threads over the available cores via intra-socket migration. Spinning does not allow such redistribution. Parking also reduces the number of concurrently ready threads, in turn reducing involuntary preemption by the operating system. However the costs to enter and exit the parked state are high and require operating system services. Thus our policies strive to reduce the rate of voluntary context switching.

CPUs transition to deeper (lower power) sleep states the longer they remain idle. Deeper sleep states, however, take longer to enter and exit. Exit latency significantly impacts unpark latency – the time between an unpark(T) operation and the time when T returns from park. Deeper sleep states, while useful for energy consumption and turbo mode, may also increase the time it takes to wake a thread. To effectively leverage the benefits of deeper sleep states, the CPU needs to stay in that state for some period to amortize the entry and exit costs. Frequent transitions between idle and running states also attenuates the turbo mode benefit for sibling CPUs as the CPU may not idle long enough to reach deeper states. Lock implementations that act to reduce thread park-unpark rates will also reduce CPU idle-running transitions and will incur less unpark latency – by avoiding sleep state exit latencies – and also allow better use of turbo mode. By keeping the ACS stable and minimal, CR reduces the park-unpark voluntary context switch rate, and in turn the idle-running CPU transition rate.

Spin-Then-Park

To reduce the impact of park-unpark overheads, lock designers may opt to use a hybrid two-phase spin-then-park strategy. Threads spin for a brief period – optimistically waiting – anticipating a corresponding unpark operation and then, if no unpark has occurred, they revert to parking as necessary. The maximum spin period is commonly set to the length of a context-switch round trip. A thread spins for either the spin period or until a corresponding unpark is observed [10]. If no unpark occurs within the period, the thread deschedules itself by blocking in the kernel. (Unparking a thread that is spinning or otherwise not blocked in the kernel is inexpensive and does not require calling into the kernel). Karlin et al. note that spinning for the length of a context switch and then, if necessary, parking, is 2-competitive [46, 51]. The spinning phase constitutes local spinning. We prefer parking – passive waiting – over spinning – active waiting – when the latencies to unpark a thread exceed the expected waiting period.

Hybrid spin-then-park [21] waiting strategies may reduce the rate of voluntary blocking and provide some relief from such voluntary context switching costs. However spin-then-park tends not to work well with strict FIFO queue-based locks. The next thread to be granted the lock is also the one that has waited the longest, and is thus most likely to have exceeded its spin duration and reverted to parking, in which case the owner will need to be unparked, significantly lengthening the critical section with context switching latencies. Spin-then-park waiting favors a predominantly LIFO admis-

[8] When only one logical CPU is active in a core, the per-core pipelines automatically fuse and provide better performance for the single active CPU. Intel processors with *hyperthreading* exhibit similar behavior. Polite spinning via the `WRPAUSE` instruction or the `RD CCR,G0` idiom also enables fusion.

[9] Turbo mode is controlled directly by hardware instead of software and requires sufficient energy headroom to be enabled. Software indirectly influences the availability of turbo mode via waiting policies.

[10] As a thought experiment, if parking and unparking had no or low latencies, then we would never use spinning or spin-then-park waiting strategies, but would instead simply park in a prompt fashion. Spinning is an optimistic attempt to avoid park-unpark overheads. Parking and spinning both reflect wasted administrative work – coordination overheads – that do not contribute directly to the forward progress of the application. Spinning is arguable greedy, optimistic and opportunistic, whiling parking reflect altruism.

sion policy. Generally, a waiting strategy that parks and unparks threads is inimical to locks that use direct handoff, and to FIFO locks specifically.

All locks evaluated in this paper use a spin-then-park waiting policy with a maximum spin duration of approximately 20000 cycles, where 20000 cycles is an empirically derived estimate of the average round-trip context switch time. On SPARC the loop consists of a load and test followed by a single `RD CCR,G0` instruction for polite spinning.

5.2 Lock Handover Latency

We define *lock handover latency* as follows. Say thread *A* holds lock *L* and *B* waits for lock *L*. *B* is the next thread to acquire ownership when *A* releases *L*. The handover latency is the time between *A*'s call to unlock and when *B* returns from lock and can enter the critical section. Handover latency reflects overheads required to convey ownership from *A* to *B*. Lock implementations attempt to minimize handover latency, also called *responsiveness* in the literature. Excessive handover latency degrades scalability. As noted above, if *A* must call into the kernel to wake and resume *B*, making *B* eligible for dispatch, then lock handover latency increases significantly.

5.3 Fairness

The default POSIX `pthread_mutex_lock` specification does not dictate fairness properties giving significant latitude and license to implementors. Fairness is considered a quality-of-implementation concern. In fact common mutex constructions, such as those found in Solaris or Linux, are based on test-and-set (TAS) locks [3], albeit augmented with parking, and allow unbounded bypass with potentially indefinite starvation and unfairness. Similarly, the `synchronized` implementation in the HotSpot Java Virtual Machine allows indefinite bypass as does `java.util.concurrent ReentrantLock`.

5.4 Succession Policies

Broadly, lock implementations use one of two possible *succession policies*, which describes how ownership is transferred at unlock-time when threads are waiting. Under *direct handoff* the unlock operation passes ownership to a waiting successor, without releasing the lock during the transfer, enabling the successor to enter the critical section. If no successor exists then the lock is set to an available state. MCS employs direct handoff. Under *competitive succession*[20] – also called *renouncement*[58] – the owner sets the lock to an available state, and, if there are any waiters, picks at least one as the *heir presumptive*, enabling that thread to re-contend for the lock [11]. Enabling an heir presumptive is necessary to ensure progress. The heir presumptive may compete with arriving threads for the lock. TAS-based locks use competitive succession and in the simplest forms all waiting threads act as heir presumptive and no specific enabling is needed.

[11] Competitive succession is also called *barging*, as arriving threads can barge in front of other waiting threads, allowing unbounded bypass and grossly unfair admission.

Locks that use direct handoff can exhibit poor performance if there are more ready threads than CPUs and involuntary context switching – preemption – is in play. The successor may have been be preempted, in which case lock handover latency will suffer. Specifically, an unlock operation may pick thread *T* as a successor, but *T* has been preempted. Circulation stalls until the operating system eventually dispatches *T* . This leads to the undesirable *convoying phenomenon* [4] . With competitive succession, the new owner must take explicit actions to acquire the lock, and is thus known to be running, albeit at just the moment of acquisition. Competitive succession reduces succession latency and works well in conditions of light contention [50]. Direct handoff performs well under high contention [52], except when there are so many ready threads that successor preemption comes into play, in which case competitive succession may provide better throughput.

Direct handoff suffers from an additional performance concern related to the waiting policy. If the successor *T* parked itself by calling into the operating system, then the unlock operator needs to make a corresponding system call to wake and unpark *T*, making *T* eligible for dispatch. The time from an unpark(*T*) call until the corresponding blocked thread *T* returns and resumes from park can be considerable. Latencies of more than 30000 cycles are common even in the best case on an otherwise unloaded system where there are fewer ready threads than CPUs and an idle CPU is available on which to dispatch *T* [12]. Crucially, these administrative latencies required by succession to resume threads accrue while the lock is held, artificially lengthening the critical section. Such lock handover latency greatly impacts throughput over the contented lock, and can dominate performance under contention.

All strictly FIFO locks use direct handoff. Relatedly, all locks that use *local spinning* [29], such as MCS, also use direct handoff. With local spinning, at most one waiting thread spins on a given location at any given time. Local spinning often implies the existence of an explicit list of waiting threads [13]. Depending on the platform, local spinning may reduce the "invalidation diameter" of the writes that transfer ownership, as the location to be written should be monitored by only one thread and thus reside in only one remote cache. Lock algorithms such as TAS use *global spinning*, where all threads waiting on a given lock busy-wait on a single memory location .

Given its point-to-point nature where thread *A* directly unparks and wakes *B*, using park-unpark for locks requires the lock algorithm to maintain an explicit list of waiting threads, visible to the unlock operator. Most locks that use local spinning, such as MCS, can therefore be readily converted to use parking. A simple TAS lock with global spinning and competitive succession requires no such list be maintained – the set of of waiting threads is implicit and invisible to the unlock operator. Lock algorithms that use global spinning,

[12] Unpark itself incurs a cost of more than 9000 cycles to the caller on our SPARC T5 system.

[13] More precisely, at unlock-time the owner thread must be able to identify the next waiting thread – the successor.

such as ticket locks or TAS locks, are more difficult to adapt to parking. As noted above, parking is typically inimical to locks that use direct handoff, as the context switch overheads artificially increase the critical section length.

We note the following tension. Locks, such as MCS, that use succession by direct handoff and local spinning can be more readily adapted to use spin-then-park waiting, the preferred waiting policy. Under high load, however, with long waiting periods, direct handoff can interact poorly with parking because of increased handover latency, where the successor has reverted to parking and needs to be explicitly made ready. Spinning becomes less successful and the lock devolves to a mode where all waiting threads park. MCSCR uses direct handoff, but can provide relief, relative to a pure FIFO lock, from handover latency as the successor is more likely to be spinning instead of fully parked.

6. Evaluation

We used an Oracle SPARC T5-2 [60] for all experiments. The T5-2 has 2 sockets, each with a single T5 processor running at 3.6 GHz. Each processor has 16 cores, and each core has 2 pipelines supporting 8 logical CPUs ("strands"), yielding 128 logical CPUs per socket. If there is only one active CPU on a core, both pipelines promptly and automatically fuse to provide improved performance. The extra strands exist to exploit available memory-level parallelism (MLP) [14]. Each socket has an 8MB unified L3 LLC shared by all cores on that socket. Each core has a fully associative 128-entry data TLB shared by all logical CPUs on that core. Each TLB entry can support all the available page sizes. Each core also has a 16KB L1 data cache and a 128KB L2 unified cache. For all experiments we took all the CPUs on the second T5-2 socket offline, yielding a non-NUMA T5 system with 128 logical CPUs.

The system ran Solaris 5.11. Unless otherwise specified, all code was compiled with gcc 4.9.1 in 32-bit mode. We observed that the performance and scalability of numerous benchmarks were sensitive to the quality of the `malloc-free` allocator. The default Solaris allocator protects the heap with a single global lock and scales poorly. The poor performance of the default allocator often dominated overall performance of applications, and masked any sensitivity to lock algorithms. We therefore used the scalable LD_PRELOAD *CIA-Malloc* allocator [1] for all experiments, except where noted. CIA-Malloc does not itself use the `pthread_mutex` primitives for synchronization.

All locks were implemented as LD_PRELOAD interposition libraries, exposing the standard POSIX `pthread_mutex` programming interface. LD_PRELOAD interposition allows us to change lock implementations by varying the LD_PRELOAD environment variable and without modifying the application code that uses locks.

We use the default free-range threading model, where the operating system is free to migrate threads between processor and nodes in order to balance load or achieve other scheduling goals. Modern operating systems use aggressive intra-node migration to balance and disperse the set of ready threads equally over the available cores and pipelines, avoiding situations where some pipelines are overutilized and others underutilized [14].

We use a number of small carefully constructed benchmarks to exhibit various modes of contention for shared hardware resources. The first examples are intentionally simple so as to be amenable to analysis.

We measure long-term fairness with the relative standard deviation (RSTDDEV), which describes the distribution of work completed by the set participating threads. We also report the Gini Coefficient [25, 37], popular in the field of economics as in index of income disparity and unfairness. A value of 0 is ideally fair (FIFO), and 1 is maximally unfair.

6.1 Random Access Array

The `RandArray` microbenchmark spawns N concurrent threads. Each thread loops as follows: acquire a central lock L; execute a critical section (CS); release L; execute a non-critical section (NCS). At the end of a 10 second measurement interval the benchmark reports the total number of aggregate iterations completed by all the threads. RandArray also reports average LWSS, median time to reacquire, and long-term fairness statistics. We vary N and the lock algorithm and report aggregate throughput results in Figure 2, taking the median of 7 runs. The number of threads on the X-axis is shown in log scale.

The NCS consists of an inner loop of 400 iterations. Each iteration generates a uniformly distributed random index into a thread-private array of 256K 32-bit integers, and then fetches that value. The CS executes the same code, but has a duration of 100 iterations and accesses a shared array of 256K 32-bit integers. The ideal speedup is 5x. The 1MB arrays reside on large pages to avoid TLB concerns. The random number generators are thread-local. We used random indexes to avoid the impact of automatic hardware prefetch mechanisms.

`MCS-S` is the classic MCS algorithm where the waiting loop is augmented to include a polite `RD CCR,G0` instruction. `MCS-STP` uses spin-then-park waiting. `MCSCR-S` is MCSCR where the waiting loop uses the `RD CCR,G0` instruction on every iteration, and `MCSCR-STP` is MCSCR with spin-then-park waiting. For reference, we include `null` where the lock acquire and release operators are degenerate and return immediately. `Null` is suitable only for trivial microbenchmarks, as other more sophisticated applications will immediately fail with this lock.

As we can see in Figure 2, ignoring `null`, the *peak* appears at about $N = 5$, where the maximum observed speedup is slightly more than 3 times that of a single thread. MCS-S and MCS-STP start to show evidence of collapse at 6 threads where the total NCS and CS footprint is 7MB, just short of the total 8MB LLC capacity. The LLC is not perfectly associative, so the onset of thrashing appears at footprints slightly below 8MB. Absent CR, the NCS instances erode LLC CS residency and impair scalability. As noted above, MCS-STP performs poorly because spin-then-parking waiting is unsuitable for direct handoff FIFO locks such as MCS. Crucially,

[14] We observe that explicit binding of threads to CPUs or indefinite spinning precludes this benefit.

spin-then-park delivers good performance for MCSCR over all thread counts, but decreases performance of classic MCS except in the case where there are more ready threads than CPUs, where pure unbounded spinning breaks down. Interestingly, MCSCR-STP achieves better performance than `null` beyond 48 threads.

While not immediately visible in the figure, at 256 threads MCS-STP yields 120x better throughput than MCS-S. Under MCS-S, as we increase the number of ready spinning threads, we increase the odds that the lock will be transferred to a preempted successor, degrading performance. Spinning threads must exhaust their allotted time slice until the owner is eventually scheduled onto a CPU. At 256 threads, MCS-STP requires a voluntary context switch for each lock handover, but it sustains reliable and consistent – but relatively low – performance even if we further increase the number of threads. This demonstrates why lock designers conservatively opt for parking over unbounded spinning. Typical time slice periods used by modern operating systems are far longer than park-unpark latencies. As such, we prefer progress via voluntary context switching over involuntary preemption.

In addition to competition for LLC residency, this graph reflects competition for pipelines [15]. At 16 threads – recall that we have 16 cores – we see MCSCR-S fade. In this case the spinning threads in the PS compete for pipelines with the "working" threads in the ACS. (The polite spin loop helps reduce the impact of pipeline competition, which would otherwise be far worse). Using a spin-then-park waiting strategy avoids this concern. MCSCR-STP manages to avoid collapse from pipeline competition.

MCS-S and MCS-STP depart from MCSCR-S and MCSCR-STP at around 8 threads because of LLC thrashing. MCSCR-S departs from MCSCR-STP at 16 threads because of competition for pipelines. The slow-down arises from the spin-only waiting policy of those locks. MCS-S and MCSCR-S exhibit an abrupt cliff at 128 threads because of competition for logical CPU residency arising from unbounded spinning. Beyond 128 threads there is system-wide competition for logical processors. MCSCR-STP is the only algorithm that maintains performance in this region, again reflecting the importance of waiting policies.

In Figure 3 we include more details of RandArray execution at 32 threads. The L3 miss rate is considerably lower under the CR forms. As would be expected, the average LWSS and the CPU utilization correspond closely under MCSCR-STP. Note too that the CPU utilization for MCSCR-STP is low, providing lower energy utilization and improved opportunities for multi-tenancy. Despite consuming the least CPU-time, MCSCR-STP yields the best performance. We use the Solaris `ldmpower` facility to measure the wattage above idle, showing that power consumption is also the lowest with MCSCR-STP. As evidenced by the LWSS and MTTR values, CR-based locks reduce the number of distinct NCS instances accessed in short intervals, in turn reducing pressure and miss rates in the LLC, accelerating CS execution, and improving overall throughput.

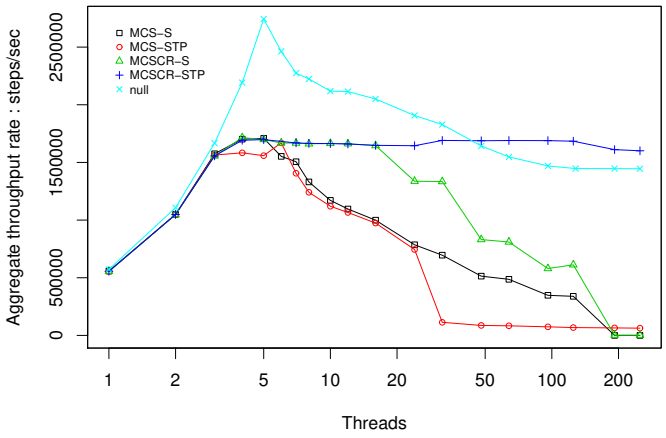

Figure 2: Random Access Array

Locks	MCS-S	MCS-STP	MCSCR-S	MCSCR-STP
Throughput (ops/sec)	0.7M	0.1M	1.3M	1.6M
Average LWSS (threads)	32	32	5.3	5.1
MTTR (threads)	31	31	3	3
Gini Coefficient	0.001	0.001	0.076	0.078
RSTDDEV	0.000	0.000	0.152	0.155
Voluntary Context Switches	0	798K	11	6K
CPU Utilization	32x	16.8x	32x	5.2x
L3 Misses	11M	10M	152K	172K
Δ Watts above idle	113	79	91	63

Figure 3: In-depth measurements for Random Access Array benchmark at 32 threads and a 10 second measurement interval

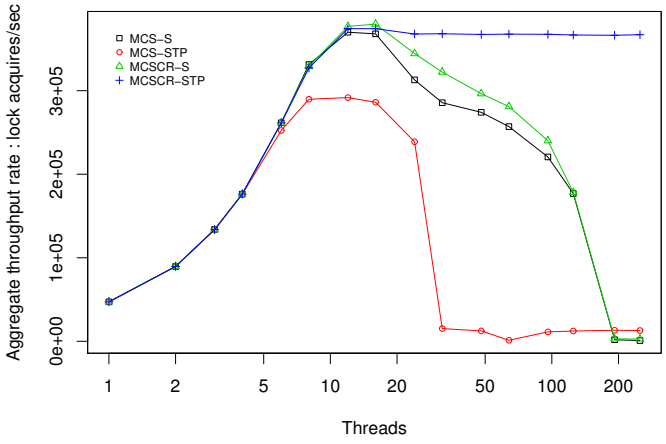

Figure 4: libslock

6.2 libslock

Figure 4 shows the performance of the `stress_latency` benchmark from [18] [16]. The benchmark spawns the spec-

[15] Other core-level resources such as TLB residency are similarly vulnerable to competition and can benefit from CR.

[16] We use the following command line: `./stress_latency -l 1 -d 10000 -a 200 -n <threads> -w 1 -c 1 -p 5000`.

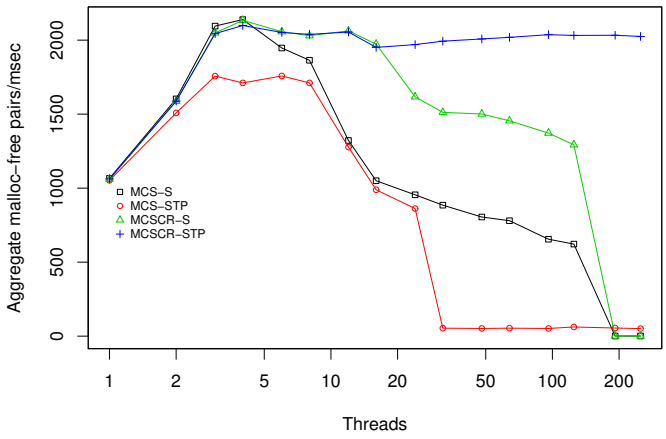

Figure 5: mmicro

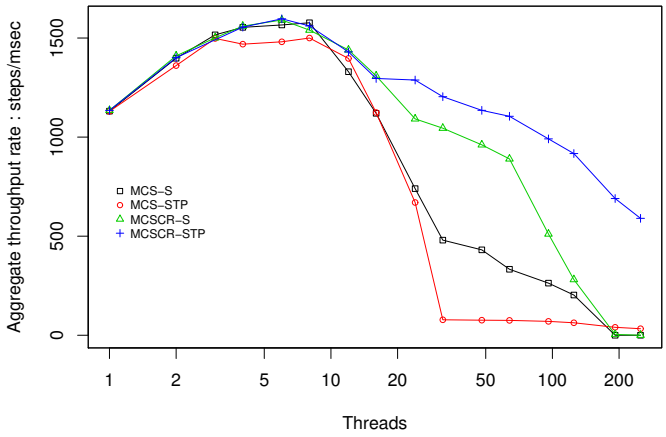

Figure 6: KyotoCabinet kccachetest

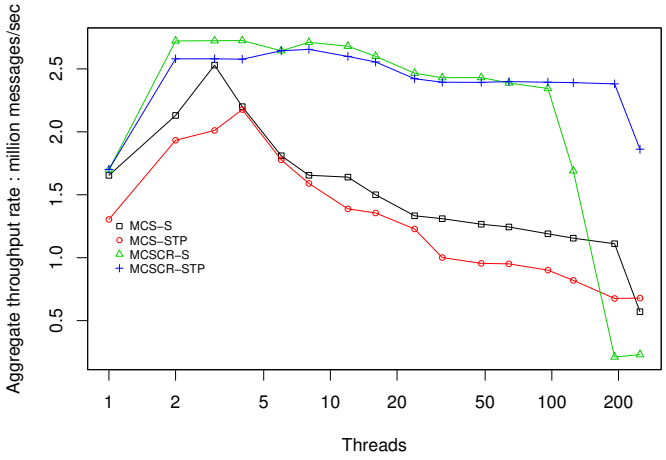

Figure 7: producer_consumer with 3 consumer threads

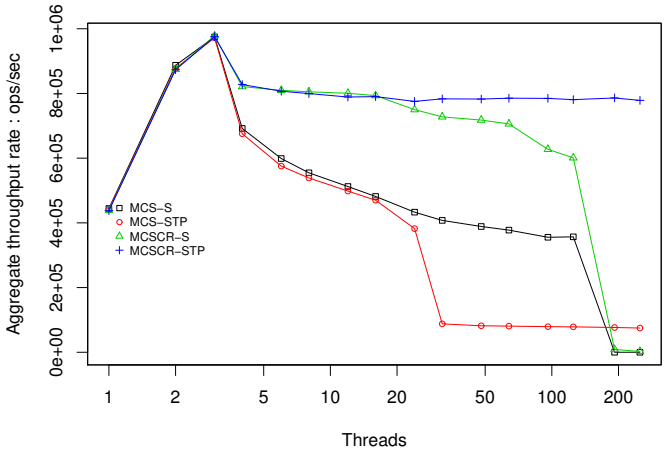

Figure 8: keymap

ified number of threads, which all run concurrently during a 10 second measurement interval. Each thread iterates as follows: acquire a central lock; execute 200 loops of a delay loop; release the lock; execute 5000 iterations of the same delay loop. The benchmark reports the total number of iterations of the outer loop. This delay loop and thus the benchmark itself are cycle-bound, and the main inflection point appears 16 threads where threads that wait via spinning compete with working threads for core-level pipelines. This again demonstrates the impact of waiting policy. Similar to many other synthetic lock microbenchmarks, very few distinct locations are accessed: there is only one shared variable and there are no memory accesses within the non-critical section.

6.3 malloc scalability benchmarks

In Figure 5 we use the `mmicro` malloc-free scalability benchmark from [29]. In this case we use the default Solaris `libc` memory allocator, which is implemented as a splay tree protected by a central mutex. While not scalable, this allocator yields a dense heap and small footprint and thus remains the default. `Mmicro` spawns a set of concurrent threads, each of which iterates as follows: allocate and zero 1000 blocks of length 1000 bytes and then release those 1000 blocks. The measurement interval is 50 seconds and we report the median of 7 runs. The benchmark reports the aggregate malloc-free rate. Each malloc and free operation acquires the central mutex. The benchmark suffers from competition for LLC residency, and, at above 16 threads, from pipeline competition. Under CR, fewer threads circulating over the central mutex lock in a given period also yields fewer malloc-ed blocks in circulation which in turn yields better hit rates for core-level TLBs and caches.

6.4 Kyoto Cabinet kccachetest

In Figure 6 we show the benefits of CR for the Kyoto Cabinet [34] `kccachetest` benchmark, which exercises an in-memory database. The performance of the database is known to be sensitive to the choice of lock algorithm [9]. We modified the benchmark to use standard POSIX `pthread_mutex` locks and to run for a fixed time and then report the aggregate work completed. We used a 300 second measurement interval and took the median of 3 runs. Finally, the key range for a run was originally computed as a function

of the number of threads, making it difficult to compare scaling performance while varying the thread count. We fixed the key range at 10M elements.

Peak performance occurs at 5 threads, dropping rapidly as we increase the number of threads. Analysis of the program with hardware performance counters shows a marked increase in LLC miss rate above 5 threads. After 16 threads MCS-S and MCS-STP suffer from both increasing LLC misses and from pipeline competition. MCSCR-STP manages to avoid the collapse exhibited by the basic MCS forms.

6.5 producer-consumer benchmark

Figure 7 illustrates the benefits of CR on the `producer_consumer` benchmark from the `COZ` package [17]. The benchmark implements a bounded blocking queue by means of a pthread mutex, a pair of pthread condition variables to signal *not-empty* and *not-full* conditions, and a standard C++ `std::queue<int>` container for the values. (This implementation idiom – a lock; a simple queue; and two condition variables – is common). Threads take on fixed roles, acting as either producers or consumers. The benchmark spawns N concurrent threads, each of which loops, producing or consuming according to its role. We fix the number of consumers at 3 threads and vary the number of producers on the X-axis, modeling an environment with 3 server threads and a variable number of clients. We report the number of messages conveyed at the end of a 10 second measurement interval, taking the median of 7 distinct trials. The queue bound was 10000 elements.

Under a classic FIFO lock, when the arrival rate of producers exceeds that of consumer threads, producers will acquire the lock and then typically find the queue is full and thus block on the condition variable, releasing the lock. Eventually they reacquire the lock, insert the value into the queue, and finally release the lock [17]. Each conveyed message requires 3 lock acquisitions – 2 by the producer and one by the consumer. The critical section length for producers is artificially increased by futile acquisitions where the producer immediately surrenders the lock and blocks on the condition variable. When the condition variable is subsequently signaled, the producer moves to the tail of the lock queue. Producers typically block 3 times : first on arrival to acquire the lock; on the condition variable; and on reacquisition of the lock. Ownership of the lock circulates over all participating threads. The queue tends to remain full or nearly so, and consumers do not need to wait on the *not-empty* condition variable.

Under a CR lock we find the system tends to enter a desirable "fast flow" mode where the futile acquisition by producers is avoided and each conveyed message requires only 2 lock acquisitions. Threads tend to wait on the mutex instead of on condition variables. Given sufficient threads, ownership continuously circulates over a small stable balanced set of producers and consumers. (As usual, long-term fairness enforcement ensures eventual participation of all threads). We note that CR's mode of benefit for the other benchmarks involves competition for fixed shared resources, whereas producer_consumer demonstrates benefits from reduced lock acquisition rates and hold times

[17] The condition variable implementation used in these experiments provides FIFO order.

6.6 keymap benchmark

The `keymap` benchmark in Figure 8 spawns set of concurrent threads, each of which loops executing a critical section followed by a non-critical section. At the end of a 10-second measurement interval the benchmark reports the aggregate throughput as the total number of loop iterations completed by all the threads. The non-critical section advances a C++ `std::mt19937` pseudo-random number generator 1000 times. The critical section acquires a central lock and then picks a random index into its thread-local *keyset* array. Each keyset array contains 1000 elements and is initialized to random keys before the measurement interval. With probability $P = .9$ the thread then extracts a key from its keyset and updates a central C++ `std::unordered_map<int,int>` instance with that key. Otherwise the thread generates a new random key in the range $[0, 10000000)$, updates the keyset index with that key, and then updates the shared map. All pseudo-random generators are thread-local and uniform. To reduce allocation and deallocation during the measurement interval, we initialize all 10000000 keys in the map prior to spawning the threads.

`Keymap` models server threads with short-lived session connections and moderate temporal key reuse and memory locality between critical sections executed by a given thread. There is little or no inter-thread CS access locality or similarity, however. Threads tend to access different regions of the CS data. The NCS accesses just a small amount of memory, and CR provides benefit by moderating inter-thread competition for occupancy of CS data in the shared LLC.

7. Discussion

MCSCR is robust under varying load and adapts the size of the ACS quickly and automatically, providing predictable performance. The implementation of MCSCR is entirely in user-space and requires no special operating system support. No stateful adaptive mechanisms are employed, resulting in more predictable behavior and faster response to changing conditions. The only tunable parameter, other than the spin duration, is how frequently the unlock operator should pick the eldest thread from the passive set, which controls the fairness-throughput trade-off.

CR also actively reduces the voluntary context switch rate. Since the passive set can remain stable for prolonged periods, threads in the passive set perform less voluntary context switching (park-unpark activity), which in turn means that the CPUs on which those threads were running may be eligible to use deeper sleep states and enjoy reduced power consumption and more thermal headroom for turbo mode. Relatedly, CR acts to reduce the number of threads concurrently spinning on a given lock, reducing wastage of CPU cycles. Voluntary blocking reduces the involuntary preemption rate as having fewer ready threads results in less preemption. That is, concurrency restriction techniques may reduce involuntary preemption rates by reducing the number of ready

threads competing for available CPUs. This also serves to reduce lock-holder preemption and convoying [18].

A common admonition is to never run with more threads than cores. This advice certainly avoids some types of scaling collapse related to core-level resource competition, but is not generally valid, ignoring the potential benefit of memory-level parallelism (MLP), threads that alternate between computation and blocking IO, etc. Many applications achieve peak throughput with far more threads than cores. Such advice also assumes a simplistic load with just one application, whereas servers may run in conditions of varying load and multi-tenancy, with multiple concurrent unrelated and mutually-unaware applications. Even within a single complex application we can find independent components with their own sets of threads, or thread pools. CR provides particular benefit in such real-world circumstances.

8. Related Work

Locks continue to underpin most applications and remain a key synchronization construct. They remain the topic of numerous recent papers [5, 8, 10, 11, 16, 27, 30, 35, 38, 39, 47, 64, 69].

Our work is most closely related to that of Johnson et al. [44], which also addresses performance issues arising from overthreading, using load and admission control to bound the number of threads allowed to spin concurrently on contended locks. Their key contribution is controlling the spin/block waiting decision based on load. If the system is overloaded, in which case there are more ready threads than logical CPUs, then some of the excess threads spinning on locks are prompted to block, reducing futile spinning and involuntary preemption. Their scheme operates only when the number of ready threads exceeds the number of logical CPUs, and some of those threads are spinning, waiting on locks, whereas ours responds earlier, at the onset of contention, and controls the size of the each individual lock's active circulating set. This allows our approach to moderate competition for other shared resources such as residency in shared caches and pipelines. Their approach operates system-wide and requires a daemon thread to detect and respond to contention and load whereas ours uses timely decentralized local per-lock decisions and is easier to retrofit into existing lock implementations. They also requires locks which are abortable, such as TP-MCS [40]. Threads that abort – shift from spinning to blocking – must "re-arrive", with undefined fairness properties. Their approach can easily leave too many spinning threads with ensuing intra-core competition for pipelines, whereas ours is more appropriate for modern multicore processors. We treat the spin/block waiting policy as a distinct albeit important concern.

Chadha et al. [12] identified cache-level thrashing as a scalability impediment and proposed system-wide concurrency throttling. Throttling concurrency to improve throughput was also suggested by Raman et al. [65] and Pusukuri et al. [62]. Chandra et al. [13] and Brett et al. [6] analyzed the impact of inter-thread cache contention. Heirman et al. [41] suggested intentional undersubscription of threads as a response to competition for shared caches. Mars et al. [55] proposed a runtime environment to reduce cross-core interference. Porterfield et al. [61] suggested throttling concurrency in order to constrain energy use. Zhuravlev et al. [72] studied the impact of kernel-level scheduling decisions – deciding which and where to dispatch ready threads – on shared resources, but did investigate the decisions made by lock subsystems. Cui et al. [15] studied lock thrashing avoidance techniques in the linux kernel where simple ticket locks with global spinning caused scalability collapse. They investigated using spin-then-park waiting and local spinning, but did not explore CR.

Like our approach, *Cohort locks* [29] explored the trade-off between throughput and short-term fairness. Cohort locks restrict the active circulating set to a preferred NUMA node over the short term. They sacrifice short-term fairness for aggregate throughput, but still enforce long-term fairness. NUMA-aware locks exploit the inter-socket topology, while our approach focuses on intra-socket resources.

Johnson et al. [45] and Lim et al. [51] explored the trade-offs between spinning and blocking.

Hardware and software transactional memory systems use *contention managers* to throttle concurrency in order to optimize throughput [71]. The issue is particularly acute for transactional memory as failed optimistic transactions are wasteful of resources.

9. Conclusion

Modern multicore systems present the illusion of having a large number of individual independent "classic" processors, connected via shared memory. This abstraction, which underlies the symmetric multiprocessing SMP programming model, is a useful simplification for programmers. In practice, however, the logical processors comprising these multicore systems share considerable infrastructure and resources. Contention for those shared resources manifests in surprising performance issues.

We describe a lock admission policy – concurrency restriction – that is intentionally unfair over the short term. Our algorithm intentionally culls excess threads – supernumerary threads not required to sustain contention – into an explicit passive set. CR moderates and reduces the size of the active circulating set, often improving throughput relative to fair FIFO locks. Periodically, we reschedule, shifting threads between the active and passive sets, affording long-term fairness. CR conserves shared resources and can reduce thrashing effects and performance drop that can occur when too many threads compete for those resources, demonstrating that judiciously managed and intentionally imposed short term unfairness can improve throughput. We further show the subtle interplay of waiting policy, which must be carefully selected to fully leverage CR.

While scalability collapse is not uncommon, it remains a challenge to characterize which shared resources underly

[18] Solaris provides the *schedctl*[23, 49] facility to request advisory deferral of preemption for lock holders – lock-holder preemption avoidance. Edler [32] proposed a similar mechanism. Schedctl can also be used to detect if the lock holder itself is running, allowing better informed waiting decisions. We did not utilize schedctl in the experiments reported in this paper.

a drop in performance. The analysis is difficult and in our experience, multiple resources are often involved [19]. While CR typically does no harm, it is also difficult to determine in advance if CR will provide any benefit. CR gates access to the resources involved in scalability collapse by moderating access to locks – an unrelated resource. In the future we hope to employ more direct means to measure and control scalability collapse. Locks remain convenient, however, and detecting oversubscription (contention) is relatively simple compared to determining when some of the complex hardware resources are oversubscribed. Contention is a convenient but imprecise proxy for overthreading.

9.1 Future Work

Throttling in current CR designs is driven by the detection of contention. In the future we hope to vary the admission rate (and the ACS size) in order to maximize lock transit rates, possibly allowing non-working conserving admission [36]. This attempts to close the performance gap between *saturation* and *peak* shown in Figure 1. We also plan to apply intentionally unfair CR-based activation policies to semaphores and the `pthread_cond` condition variable construct, tending to wake the most recently arrived threads.

Classic CR is concerned with the size of the ACS. But we can easily extend CR to be NUMA-aware by taking the demographics of the ACS into account in the culling criteria. For NUMA environments we prefer the ACS to be homogeneous and composed of threads from just one NUMA node. This reduces the NUMA-diversity of the ACS, reduces lock migrations and improves performance. Our **MCSCRN** design starts with MCSCR, but we add two new fields: the identity of the currently preferred "home" NUMA node, and a list of remote threads. At unlock-time, the owner thread inspects the next threads in the MCS chain and culls remote threads from the main chain to the remote list. A thread is considered remote if it runs on some node other than the currently preferred node. Periodically, the unlock operator also selects a new home node from the threads on the remote list, and drains threads from that node into the main MCS chain, conferring long-term fairness. If we encounter a deficit on the main list at unlock-time, then we simply reprovision from the remote list.

Unlike cohort locks, MCSCRN locks are small and of fixed size. In the uncontended case, cohort locks require acquisition of both the node-level and top-level, although a fast-path can be implemented that tries to avoid that overhead by opportunistically bypassing the node-level locks under conditions of no or light contention when cohort formation is not feasible. MCSCRN is non-hierarchical, and avoids that concern, always using the fast-path. The system tends to converge quickly to a steady-state where the arriving threads are largely from the home node, so accesses to lock metadata elements avoids inter-node coherence traffic.

[19] Suggesting the need for enhanced hardware performance facilities to detect excessive competition for shared resources.

Acknowledgments

We thank Alex Kogan, Doug Lea, Jon Howell and Paula J. Bishop for useful discussions.

References

[1] Y. Afek, D. Dice, and A. Morrison. Cache Index-aware Memory Allocation. International Symposium on Memory Management – ISMM, 2011. URL http://doi.acm.org/10.1145/1993478.1993486.

[2] H. Akkan, M. Lang, and L. Ionkov. HPC runtime support for fast and power efficient locking and synchronization. In *2013 IEEE International Conference on Cluster Computer – CLUSTER*, 2013. URL http://dx.doi.org/10.1109/CLUSTER.2013.6702659.

[3] T. E. Anderson. The Performance of Spin Lock Alternatives for Shared-Memory Multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 1(1), Jan. 1990. URL http://dx.doi.org/10.1109/71.80120.

[4] M. Blasgen, J. Gray, M. Mitoma, and T. Price. The Convoy Phenomenon. *SIGOPS Operating Systems Review*, 1979. URL http://doi.acm.org/10.1145/850657.850659.

[5] S. Boyd-Wickizer, M. F. Kaashoek, R. Morris, and N. Zeldovich. Non-scalable locks are dangerous. In *Proceedings of the Linux Symposium*, 2012.

[6] B. Brett, P. Kumar, M. Kim, and H. Kim. CHiP: A Profiler to Measure the Effect of Cache Contention on Scalability. International Parallel and Distributed Processing Symposium Workshops PhD Forum – IPDPSW. IEEE Computer Society, 2013. URL http://dx.doi.org/10.1109/IPDPSW.2013.49.

[7] F. P. J. Brooks. *The Mythical Man-Month*. Addison-Wesley, 1975. ISBN 0-201-00650-2.

[8] D. Bueso. Scalability Techniques for Practical Synchronization Primitives. *Communications of the ACM – CACM*, 2014. URL http://doi.acm.org/10.1145/2687882.

[9] I. Calciu, D. Dice, Y. Lev, V. Luchangco, V. J. Marathe, and N. Shavit. NUMA-aware Reader-writer Locks. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP. ACM, 2013. URL http://doi.acm.org/10.1145/2442516.2442532.

[10] M. Chabbi and J. Mellor-Crummey. Contention-conscious, Locality-preserving Locks. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP. ACM, 2016. URL http://doi.acm.org/10.1145/2851141.2851166.

[11] M. Chabbi, M. Fagan, and J. Mellor-Crummey. High Performance Locks for Multi-level NUMA Systems. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP. ACM, 2015. URL http://doi.acm.org/10.1145/2688500.2688503.

[12] G. Chadha, S. Mahlke, and S. Narayanasamy. When Less is More (LIMO):Controlled Parallelism For Improved Efficiency. Conference on Compilers, Architectures and Synthesis for Embedded Systems – CASES, 2012. URL http://doi.acm.org/10.1145/2380403.2380431.

[13] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting Inter-Thread Cache Contention on a Chip Multi-Processor Architecture. Symposium on High-Performance Computer Architecture – HPCA. IEEE Computer Society, 2005. URL http://dx.doi.org/10.1109/HPCA.2005.27.

[14] Y. Chou, B. Fahs, and S. Abraham. Microarchitecture Optimizations for Exploiting Memory-Level Parallelism. International Symposium on Computer Architecture – ISCA. IEEE Computer Society, 2004. URL http://dl.acm.org/citation.cfm?id=998680.1006708.

[15] Y. Cui, Y. Chen, and Y. Shi. Comparison of Lock Thrashing Avoidance Methods and Its Performance Implications for Lock Design. Workshop on Large-scale System and Application Performance – LSAP. ACM, 2011. URL http://doi.acm.org/10.1145/1996029.1996033.

[16] Y. Cui, Y. Wang, Y. Chen, and Y. Shi. Requester-Based Spin Lock: A Scalable and Energy Efficient Locking Scheme on Multicore Systems. *IEEE Transactions on Computers*, 2015. URL http://dx.doi.org/10.1109/TC.2013.196.

[17] C. Curtsinger and E. D. Berger. Coz: Finding Code That Counts with Causal Profiling. Symposium on Operating Systems Principles – SOSP. ACM, 2015. URL http://doi.acm.org/10.1145/2815400.2815409.

[18] T. David, R. Guerraoui, and V. Trigonakis. Everything You Always Wanted to Know About Synchronization but Were Afraid to Ask. Symposium on Operating Systems Principles – SOSP. ACM, 2013. URL http://doi.acm.org/10.1145/2517349.2522714.

[19] P. J. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, 1980. doi: 10.1109/TSE.1980.230464. URL http://dx.doi.org/10.1109/TSE.1980.230464.

[20] D. Dice. Implementing Fast Java Monitors with Relaxed-locks. In *Proceedings of the 2001 Symposium on JavaTM Virtual Machine Research and Technology Symposium - Volume 1*, JVM. USENIX Association, 2001. URL https://www.usenix.org/legacy/event/jvm01/full_papers/dice/dice.pdf.

[21] D. Dice. Adaptive spin-then-block mutual exclusion in multi-threaded processing, Sept. 2009. URL http://www.google.com/patents/US7594234. US Patent 7,594,234.

[22] D. Dice. Polite busy-waiting with WRPAUSE on SPARC, 2011. URL https://blogs.oracle.com/dave/entry/polite_busy_waiting_with_wrpause.

[23] D. Dice. Inverted schedctl usage in the JVM, 2011. URL https://blogs.oracle.com/dave/entry/inverted_schedctl_usage_in_the.

[24] D. Dice. Using MWAIT in spin loops, 2011. URL https://blogs.oracle.com/dave/resource/mwait-blog-final2.txt.

[25] D. Dice. Measuring long-term fairness for locks, 2014. URL https://blogs.oracle.com/dave/entry/measuring_long_term_fairness_for.

[26] D. Dice. Preemption Tolerant MCS Locks, 2016. URL https://blogs.oracle.com/dave/entry/preemption_tolerant_mcs_locks.

[27] D. Dice and T. Harris. Lock Holder Preemption Avoidance via Transactional Lock Elision. ACM SIGPLAN Workshop on Transactional Computing – Transact, 2016. URL https://blogs.oracle.com/dave/resource/LHPA-TLE-Feb16.pdf.

[28] D. Dice, N. Shavit, and V. J. Marathe. US Patent US8775837 - Turbo Enablement, 2012. URL http://www.google.com/patents/US8775837.

[29] D. Dice, V. J. Marathe, and N. Shavit. Lock Cohorting: A General Technique for Designing NUMA Locks. *ACM Transactions on Parallel Computing – TOPC*, 1 (2), Feb 2015. URL http://doi.acm.org/10.1145/2686884.

[30] J. Eastep, D. Wingate, M. D. Santambrogio, and A. Agarwal. Smartlocks: Lock Acquisition Scheduling for Self-aware Synchronization. International Conference on Autonomic Computing – ICAC, 2010. URL http://doi.acm.org/10.1145/1809049.1809079.

[31] E. Ebrahimi, R. Miftakhutdinov, C. Fallin, C. J. Lee, J. A. Joao, O. Mutlu, and Y. N. Patt. Parallel Application Memory Scheduling. In *International Symposium on Microarchitecture – MICRO-44*. ACM, 2011. URL http://doi.acm.org/10.1145/2155620.2155663.

[32] J. Edler, J. Lipkis, and E. Schonberg. Process Management for Highly Parallel UNIX Systems. *In Proc. 1988 USENIX Workshop on UNIX and Supercomputers*, 1988.

[33] S. Eyerman and L. Eeckhout. Modeling Critical Sections in Amdahl's Law and its Implications for Multicore Design. International Symposium on Computer Archtecture – ISCA. ACM, 2010. URL http://doi.acm.org/10.1145/1815961.1816011.

[34] FAL Labs. Kyoto cabinet. URL http://fallabs.com/kyotocabinet/.

[35] B. Falsafi, R. Guerraoui, J. Picorel, and V. Trigonakis. Unlocking Energy. In *USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/falsafi.

[36] C. Gershenson and D. Helbing. When Slower is Faster. *CoRR*, 2011. URL http://arxiv.org/abs/1506.06796v2.

[37] C. Gini. Variabilità e Mutabilità. *Memorie di Metodologica Statistica*, 1912.

[38] H. Guiroux, R. Lachaize, and V. Quéma. Multicore Locks: The Case Is Not Closed Yet. In *USENIX Annual Technical Conference (USENIX ATC 16)*. USENIX Association, 2016. URL https://www.usenix.org/conference/atc16/technical-sessions/presentation/guiroux.

[39] J. Gustedt. Futex Based Locks for C11's Generic Atomics. Symposium on Applied Computing – SAC. ACM, 2016. URL http://doi.acm.org/10.1145/2851613.2851956.

[40] B. He, W. N. Scherer, and M. L. Scott. Preemption Adaptivity in Time-published Queue-based Spin Locks. High Performance Computing – HiPC. Springer-Verlag, 2005. URL http://dx.doi.org/10.1007/11602569_6.

[41] W. Heirman, T. Carlson, K. Van Craeynest, I. Hur, A. Jaleel, and L. Eeckhout. Undersubscribed Threading on Clustered Cache Architectures. In *2014 IEEE 20th International Symposium on High Performance Computer Architecture – HPCA*. URL http://dx.doi.org/10.1109/HPCA.2014.6835975.

[42] J. Holtman and N. J. Gunther. Getting in the Zone for Successful Scalability. *CoRR*, 2008. URL http://arxiv.org/abs/0809.2541.

[43] Intel. Improving Real-Time Performance by Utilizing Cache Allocation Technology. URL http://www.intel.com/content/dam/www/public/us/en/documents/white-papers/cache-allocation-technology-white-paper.pdf.

[44] F. R. Johnson, R. Stoica, A. Ailamaki, and T. C. Mowry. Decoupling Contention Management from Scheduling. Architectural Support for Programming Languages and Operating Systems – ASPLOS XV. ACM, 2010. URL http://doi.acm.org/10.1145/1736020.1736035.

[45] R. Johnson, M. Athanassoulis, R. Stoica, and A. Ailamaki. A New Look at the Roles of Spinning and Blocking. Proceedings of the Fifth International Workshop on Data Management on New Hardware – DaMoN. ACM, 2009. URL http://doi.acm.org/10.1145/1565694.1565700.

[46] A. R. Karlin, K. Li, M. S. Manasse, and S. Owicki. Empirical Studies of Competitve Spinning for a Shared-memory Multiprocessor. *SIGOPS Operating Systems Review*, 1991. URL http://doi.acm.org/10.1145/121133.286599.

[47] S. Kashyap, C. Min, and T. Kim. Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds. *SIGOPS Operating Systems Review*, 2016. URL http://doi.acm.org/10.1145/2903267.2903271.

[48] L. I. Kontothanassis, R. W. Wisniewski, and M. L. Scott. Scheduler-Conscious Synchronization. *ACM Transations on Computing Systems*, 1997. URL http://doi.acm.org/10.1145/244764.244765.

[49] N. Kosche, D. Singleton, B. Smaalders, and A. Tucker. Method and apparatus for execution and preemption control of computer process entities: US Patent number 5937187, 1999. URL http://www.google.com/patents/US5937187.

[50] D. Lea. java.util.concurrent abstractqueuedsynchronizer, 2016. URL http://download.java.net/java/jdk9/docs/api/java/util/concurrent/locks/AbstractQueuedSynchronizer.html.

[51] B.-H. Lim and A. Agarwal. Waiting Algorithms for Synchronization in Large-scale Multiprocessors. *ACM Transactions on Computing Systems*, 1993. URL http://doi.acm.org/10.1145/152864.152869.

[52] B.-H. Lim and A. Agarwal. Reactive Synchronization Algorithms for Multiprocessors. Architectural Support for Programming Languages and Operating Systems – ASPLOS. ACM, 1994. URL http://doi.acm.org/10.1145/195473.195490.

[53] V. Luchangco, D. Nussbaum, and N. Shavit. A Hierarchical CLH Queue Lock. In *Euro-Par 2006 Parallel Processing*. 2006. URL http://dx.doi.org/10.1007/11823285_84.

[54] E. P. Markatos and T. J. LeBlanc. Multiprocessor synchronization primitives with priorities. 8th IEEE Workshop on Real-Time Operating Systems and Software. IEEE, 1991.

[55] J. Mars, N. Vachharajani, R. Hundt, and M. L. Soffa. Contention Aware Execution: Online Contention Detection and Response. International Symposium on Code Generation and Optimization – CGO. ACM, 2010. URL http://doi.acm.org/10.1145/1772954.1772991.

[56] G. Marsaglia. Xorshift RNGs. *Journal of Statistical Software*, 8(1), 2003. doi: 10.18637/jss.v008.i14. URL http://dx.doi.org/10.18637/jss.v008.i14.

[57] J. M. Mellor-Crummey and M. L. Scott. Algorithms for Scalable Synchronization on Shared-memory Multiprocessors. *ACM Transactions on Computing Systems*, 9(1), Feb. 1991. URL http://doi.acm.org/10.1145/103727.103729.

[58] R. Odaira and K. Hiraki. Selective Optimization of Locks by Runtime Statistics and Just-in-Time Compilation. International Parallel and Distributed Processing Symposium Workshops – IPDPS. IEEE Computer Society, 2003. URL http://dl.acm.org/citation.cfm?id=838237.838665.

[59] Open Solaris. Synch.c : pthread_mutex implementation. URL https://github.com/sonnens/illumos-joyent/blob/master/usr/src/lib/libc/port/threads/synch.c.

[60] Oracle Corporation. Oracle's SPARC T5-2, SPARC T5-4, SPARC T5-8, and SPARC T5-1B Server Architecture, 2014. URL http://www.oracle.com/technetwork/server-storage/sun-sparc-enterprise/documentation/o13-024-sparc-t5-architecture-1920540.pdf.

[61] A. K. Porterfield, S. L. Olivier, S. Bhalachandra, and J. F. Prins. Power Measurement and Concurrency Throttling for Energy Reduction in OpenMP Programs. International Parallel and Distributed Processing Symposium Workshops – IPDPSW. IEEE Computer Society, 2013. URL http://dx.doi.org/10.1109/IPDPSW.2013.15.

[62] K. K. Pusukuri, R. Gupta, and L. N. Bhuyan. Thread Reinforcer: Dynamically Determining Number of Threads via OS Level Monitoring. International Symposium on Workload Characterization – IISWC. IEEE Computer Society, 2011. URL http://dx.doi.org/10.1109/IISWC.2011.6114208.

[63] Z. Radović and E. Hagersten. Hierarchical Backoff Locks for Nonuniform Communication Architectures. In *International Symposium on High Performance Computer Architecture – HPCA*. IEEE Computer Society, 2003. URL http://dl.acm.org/citation.cfm?id=822080.822810.

[64] P. Ramalhete and A. Correia. Tidex: A Mutual Exclusion Lock. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP. ACM, 2016. URL http://doi.acm.org/10.1145/2851141.2851171.

[65] A. Raman, H. Kim, T. Oh, J. W. Lee, and D. I. August. Parallelism Orchestration Using DoPE: The Degree of Parallelism Executive. Programming Language Design and Implementation – PLDI. ACM, 2011. URL http://doi.acm.org/10.1145/1993498.1993502.

[66] K. Ren, J. M. Faleiro, and D. J. Abadi. Design Principles for Scaling Multi-core OLTP Under High Contention. *CoRR*, 2015. URL http://arxiv.org/abs/1512.06168.

[67] G. E. Suh, L. Rudolph, and S. Devadas. Dynamic Partitioning of Shared Cache Memory. *Journal of Supercomputing*, 2004. URL http://dx.doi.org/10.1023/B:SUPE.0000014800.27383.8f.

[68] J.-T. Wamhoff, S. Diestelhorst, C. Fetzer, P. Marlier, P. Felber, and D. Dice. The TURBO Diaries: Application-controlled Frequency Scaling Explained. In *2014 USENIX Annual Technical Conference (USENIX ATC 14)*. URL https://www.usenix.org/conference/atc14/technical-sessions/presentation/wamhoff.

[69] T. Wang, M. Chabbi, and H. Kimura. Be My Guest: MCS Lock Now Welcomes Guests. ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming – PPoPP. ACM, 2016. URL http://doi.acm.org/10.1145/2851141.2851160.

[70] Wikipedia. Malthusianism, 2015. URL https://en.wikipedia.org/wiki/Malthusianism [Online; accessed 2015-08-07].

[71] R. M. Yoo and H.-H. S. Lee. Adaptive Transaction Scheduling for Transactional Memory Systems. ACM Symposium on Parallelism in Algorithms and Architectures – SPAA, 2008. URL http://doi.acm.org/10.1145/1378533.1378564.

[72] S. Zhuravlev, J. C. Saez, S. Blagodurov, A. Fedorova, and M. Prieto. Survey of Scheduling Techniques for Addressing Shared Resources in Multicore Processors. *ACM Computing Surveys*, 2012. URL http://doi.acm.org/10.1145/2379776.2379780.

A. Additional lock formulations that provide concurrency restriction

We provide additional examples to illustrate generality and show that other locks providing concurrency restriction can be constructed.

A.1 LOITER Locks

Simple TAS or more polite test-and-test-and-set spin locks can be deeply unfair. A thread can repeatedly barge in front of and bypass threads that have waited longer. A simple TAS lock without back-off can also suffer from considerable futile coherence traffic when the owner releases the lock and the waiting threads observe the transition and N such spinning threads *pounce*, trying to obtain ownership via an atomic instruction, producing a *thundering herd* effect. $N-1$ will fail, but in doing so force coherence traffic on the underlying cache line. As such, modern TAS locks are typically augmented with randomized back-off, which reduces coherence traffic from polling and also reduces the odds of futile attempts to acquire the lock. Back-off strives to balance those benefits against reduced lock responsiveness. Longer back-off periods entail longer possible "dead time" where the lock has been released but the waiting threads have not detected that transition [20]. Traditional randomized back-off for TAS locks is *anti-FIFO* in the sense that threads that have waited longer are less likely to acquire the lock in unit time. Absent remediation, such back-off may partition threads into those that wait for long periods and those that wait for short periods and circulate rapidly [21].

Fairness of TAS locks is further determined by platform-specific aspects of the system such as the underlying hardware arbitration mechanism for cache lines. On some platforms, threads running "near" the most recent owner – near in the system topology – may enjoy a persistent statistical advantage acquiring the lock, dominating ownership. On some platforms, threads on the home node of the memory underlying the lock will have a persistent advantage. Somewhat perversely, such behavior can be NUMA-friendly over the short-term as it tends to reduce lock migrations. The unfairness can persist for long periods, however.

Despite these disadvantages, TAS locks confer a key benefit: the lock is never passed to a preempted thread as might be the case with MCS This reduces undesirable convoying behavior and latencies waiting for a ready but descheduled thread to again be dispatched onto a CPU. Furthermore, waiting threads do not need to register or otherwise make themselves visible to threads performing the unlock operation, reducing administrative overheads and coherence costs related to lock metadata. As such, these locks perform better under mixed load, and in particular when the number of runnable threads exceeds the number of logical CPUs. They also have very low latency hand-off under light or no contention.

We design a simple TAS lock enhanced with CR as follows. Our new **LOITER** (Locking : Outer-Inner with ThRottling) lock has an *outer* TAS lock. Arriving threads try to obtain the outer lock using a bounded spin phase – busy waiting – with randomized back-off. If they acquire the outer lock, they can enter the critical section. We refer to this as the fast-path. If the spinning attempt fails, control then reverts to an *inner lock*. An MCS lock with spin-then-park waiting is suitable for use as the inner lock. The thread that manages to acquire the inner lock is called the *standby* thread – there is at most one standby thread per lock at any given moment. The inner lock constitutes a so-called slow path. The standby thread then proceeds to contend for the outer lock. Again, it uses a spin-then-park waiting policy. When the standby thread ultimately acquires the outer lock it can enter the critical section. At unlock time, if the current owner acquired the lock via the slow path, it releases both the outer lock and the inner lock. Otherwise if it releases the outer lock and, if a standby thread exists, it unparks that standby thread as the heir presumptive.

The ACS consists of the owner, threads passing through their non-critical sections, and threads spinning in the fast path arrival phase. The PS consists of threads waiting for the inner lock. The standby thread is on the cusp and is transitional between the two sets. Under steady state the system converges to a mode where we have a stable set of threads circulating over the outer lock (the ACS), at most one thread spinning or parking in the standby position, and the remainder of the threads are blocked on the inner locks (the PS).

We impose long-term fairness by detecting that the standby thread has waiting too long and is "impatient", in which case it requests direct handoff of ownership to the standby thread upon the next unlock operation. This construction attempts to retain the desirable properties of TAS-based lock while providing CR and long-term fairness. The result is a hybrid that uses competitive handoff in most cases, reverting to direct handoff as part of an anti-starvation mechanism when the standby thread languishes too long.

Arriving threads start with global spinning on the outer lock, and if they can't manage to obtain the lock within the arrival spinning phase, they then revert to the MCS lock, which uses local waiting. Global spinning allows more efficient lock hand-over, but local spinning generates less coherence traffic and provides gracefully performance under high contention [52]. Threads waiting on the inner MCS lock simply spin or spin-then-park on the thread-local variable, avoiding concerns about back-off policies. All park-unpark activity takes place on paths outside the critical section. The inner lock provides succession by direct handoff via MCS, while the outer lock provides succession by competitive handoff. This constitutes a 3-stage waiting policy : threads first spin globally; then, if necessary, enqueue and spin locally; and then park.

The LOITER transformation allows us to convert a lock such as MCS, which uses direct handoff, into a composite form that allows a fast path with barging. The resultant composite LOITER lock enjoys the benefits of both direct handoff and competitive succession, while mitigating the undesirable aspects of each of those policies. Specifically, the new construct uses direct handoff for threads in the slow contention path, but allows competitive succession for threads circulating outside the slow path, retaining the best properties of both MCS and TAS locks.

A.2 LIFO-CR

This design starts with a pure LIFO lock [22] with an explicit stack of waiting threads. Contended threads push an MCS-like node onto the stack and then spin or spin-then-park on a thread-local flag. When threads are waiting, the unlock operator pops the head of stack – the most recently arrived thread – and directly passes ownership to that thread. Both "push" and "pop" operations are implemented via atomic compare-and-swap CAS instructions. Only the lock holder can "pop" elements, so the approach is immune to ABA pathologies. The stack is multiple-producer but, by virtue of the lock itself, single-consumer. The ACS consists of the owner, the threads circulating through their respective NCS regions, and the top of the stack. The PS consists of the threads deeper on the stack. Admission order is effectively cyclic round-robin over the members of the ACS, regardless of the prevailing LIFO lock admission policy. We then augment the lock to periodically pick the tail of the stack – the eldest thread – to be the next owner. This imposes long-term fairness. We refer to the resultant lock as **LIFO-CR**. LIFO admission order may improve temporal locality and reduce misses in shared caches. Both LIFO-CR and LOITER offer performance competitive with MCSCR.

[20] Arguably, back-off is not work conserving.

[21] The back-off can also provide inadvertent and unintended but beneficial concurrency restriction.

[22] If we use a pure LIFO lock then the LWSS should correspond to the ACS size, giving an easy way to measure the ideally minimal ACS size and maximum benefit afforded by CR.