Malthusian Locks

Dave Dice
Senior Research Scientist
Scalable Synchronization Research Group – Oracle Labs
EuroSys 2017 – 2017-4-23
Robert Malthus : population vs resources
Agenda

1. Problem: Concave Scaling
2. Remedy: Concurrency Restriction (CR)
3. Introduce MCSCR – implements solution
- Classic concave scaling
- Failure to scale
- Add threads $\rightarrow$ reduced tput
- Assume #threads $<<$ #CPUs: no preemption in play
- Excess threads don’t contribute
- Why ...
- Communication and synchronization overheads
- Competition for hardware resources
- Often found with contended locks
- ‘Saturation’ point: sustained contention
- Limiting factor
- Saturation and Peak are unrelated
- Leverage existence of contented lock
- Impose CR = Concurrency Restriction
Etiology of Scaling Problem

Overheads of parallelism exceed benefits - unprofitable

• Communication and synchronization overheads – orchestration
  – Fundamental algorithmic costs & limits – intrinsic: messages $O(T^2)$
  – Costs related to implementation of synchronization primitives: Locks, etc - induced
    • Coherence costs – communications on lock metadata
    • Administrative overheads that scale with the number of threads

• Competition for shared hardware resources

Our Focus
Competition for shared hardware resources

• Destructive interference in last-level cache (LLC)
  – competition for residency among threads – cache pressure
  – LLC misses cause DRAM channel congestion

• Core-level DTLBs and pipelines; logical CPUs

• Thermal and energy headroom

• Coherent communication: interconnect bandwidth

• System-wide Memory pressure

• Additional shared resources subject to conservation ...

• Abstraction of modern SMP systems – facade
Impose Concurrency Restriction (CR)

• Limit number of distinct threads that circulate over the lock within a period
• But remain *work conserving*
• Assumes sufficient contention to have excess threads
• Partition threads: *active set and passive set*
  – Active threads circulate as usual over the lock
  – Excess threads culled into passive set: quiesced
  – Draw from passive set when active set becomes empty
• Unfair over short-term
• Explicitly enforce long-term fairness: periodically mix active and passive
• Reduces competition for shared resources
  – May improve scalability
  – Fewer threads circulating ➔ reduced cache pressure
Analogy - Swapping

• Memory pressure and medium-term scheduling
  – Extreme paging ➔ thrashing ➔ swapping
  – Quiesce selected processes
  – Restrict concurrency – admission control

• Threads vs shared resources
  – Thrashing in shared cache
  – Minimize lock working-set size
    • Number of distinct threads that have acquired the lock in N most recent admissions
  – Use locks to quiesce excess threads – convenient mechanism
Classic MCS lock

- **Mellor-Crummey and Scott**
- Gold-standard of lock algorithms
  - Academic vs practical: not used in pthread_mutex implementations
- Arriving threads use SWAP to join tail of queue – then wait
  - Forms explicit list of waiting threads
  - Strict FIFO admission order: fair
- **Local spinning** on flag within enqueued element
  - Predictable and stable performance
  - Preferred for large systems where coherence costs are higher
- Succession policy: Direct handoff over ownership to next in queue
  - Downside: May hand off ownership to a preempted thread
  - Vulnerable when Threads >> Cores: convoying
LIFO lock

• Maximally unfair
• Not for use in production!
  – Thought experiment: admits starvation
  – Demonstrates the advantages of unfairness
• Stack of threads
  – Arrive: push
  – Unlock: pop
• Stack head modified at both arrival and unlock
  – Constitutes coherence hot-spot
• Local spinning
Microbenchmark

• SPARC T5-2 with all CPUs on socket-1 offline: 1x16x8 = 128-way (avoid NUMA)
• 2 pipelines per core; fusion
• LD_PRELOAD: interpose on pthread_mutex_lock() etc
• Specification affords considerable latitude for fairness and admission order
• Various locks
• Naming conventions: waiting policy
  – “-S” = spin with PAUSE in wait loop
  – “-STP” = spin-then-park: parking = block thread in kernel
• Issues appear on multicore x86 as well as SPARC
Microbenchmark

• Spawn T concurrent threads ...
• Loop : acquire lock; critical section; release; non-critical
• Critical section : 100 iterations
  – Generate random number for index (thread-local generator)
  – fetch from shared array of int32 : 256K elements
• Non-critical section : 400 iterations; thread-private array
• Maximum ideal Amdahl speedup is 5x
• Report aggregate throughput rate at end of measurement interval
  Loops/sec : fixed-time-report-work mode
• Threads on X-axis (log scale); Throughput on Y-axis
Threads << Cores << CPUs

Onset of scaling problem
Threads << cores
not pipeline competition
LLC thrashing

Preemption

Waiting Policy
Threads >> Cores
Spinners compete for pipelines with workers
Unfair ➔ Higher Throughput
Next Step : MCSCR

- **MCS with Concurrency Restriction**
- Combines best of LIFO (throughput) and MCS (long-term fairness)
- Edits the queue of waiting threads at unlock()-time
- Move excess threads aside into explicit passive set : culling
- Small constant-time cost increase in unlock() path; lock() unchanged
- Periodically shift threads between active and passive
  - Every 1000 unlocks : probabilistically
  - Trade-off between short-term fairness and throughput
  - Enforce long-term fairness
- Unfair over the short-term : “hot” threads stay hot
- Reduces competition for shared resources
Passive Set

Non-Critical Section

Waiting Threads

Critical Section
Variations and Future Directions

• Relatively easy to impose CR on existing lock algorithms
• LIFO-CR : LIFO, but occasionally select tail of stack to run
• Trivial to implement NUMA-aware forms
  – Basic CR concerned with the number of circulating threads
  – NUMA-aware : demographics of the circulating active threads
  – Constrain NUMA-diversity of the active threads
  – Cull remote threads
  – Related : “Cohort Locks” -- trade fairness for throughput
    keep ownership of lock on a node for short periods
• Apply CR to Condition variables – thread pools; semaphores; etc
Take-away

• Caveats:
  – Moderating competition for shared HW resources via unrelated facility (locks)
  – Peak and Saturation points are unrelated
  – Locks shouldn’t be in the scheduling business
  – Lock is simply convenient point for admission control
  – Don’t always have a contented lock
  – Opportunist and palliative: doesn’t address root cause
  – Points to more general admission control strategies (stay tuned ...)

• Unfairness can improve throughput ... but should be bounded
• Choice of waiting policy is critical
• MCSCR lock
• Scaling “fade” can be difficult to diagnose
• CR doesn’t always help, but doesn’t hurt
Dave Dice
dave.dice@oracle.com
https://blogs.oracle.com/dave/
http://arxiv.org/abs/1511.06035 (extended version)