NTNU

Norwegian University of Science and Technology

Latency Impact on Spin-Lock Algorithms for Modern Shared-Memory Multiprocessors:

Jan C. Meyer and Anne C. Elster
Scalable Computing: Practice & Experience
Vol .9, No.3, pp. 197—206

Presentation for DT8105, March 30th, 2009

A long time ago, in a galaxy far, far away...

- ...supercomputers were called things like “BBN Butterfly” and “Sequent Symmetry”
- ...certain processors (such as the MC68000 in the BBNB) had no cache
- ...certain interconnects (such as the S.Sym.) were shared buses which would retransmit on collision
- ...John M. Mellor-Crummey and Michael L. Scott published *“Algorithms for Scalable Synchronization on Shared-Memory Architectures”*
- A landmark paper on scalability of spin-lock and barrier primitives, does empirical evaluation of all significant innovations to (1991) date, and contributes a spin-lock and a barrier algorithm
- An impressive effort, but a somewhat dense read

More recently

- A great number of articles have built upon the work of Anderson and MCS, proposing several alternative spin-lock algorithms, mostly either from creating a cost model which differentiates between local and remote access, or examining the relative power of various atomic operations
- Many good ideas, but somewhat detached from empirical studies
- Honorable exceptions: further work by Michael & Scott (ref. 4), and the Stanford DASH team (ref. 8)
- *Interesting side-note:* DASH begat Origin 2000, which begat Origin 3800, which is one of the two test platforms in the paper, SGI-s ccNUMA architecture (coherency protocol & memory subsystem, ref. 7) reads like a checklist from DASH (ref. 8)
- 2006 Dijkstra prize (awarded to lasting contributions in concurrent computing) went to the MCS lock

What is a spin-lock, anyway?

- Mutual exclusion by persistent nagging
- N participants busy-wait for a resource, only one gets it at a time
- Wins no awards for subtle elegance, but a useful approach when critical sections are shorter than sleep/ wake overhead (tight synchronization)
- Admits a few important considerations:
  - How is the lock structured in memory?
  - Where is the lock located?
  - What to do in the meantime when another processor has the lock?

Fetch-and-Φ

- Fancy name for atomic read-modify-write sequences
- Often part of ISA in days of yore, e.g. supporting *fetch-and-add* made your ISA cooler than *test-and-set* did...
- There is a hierarchy of the power of these operations, but it's not as interesting for empirical work, as...
- ...ever since DASH (1992), the order of the day is mostly to roll your own fetch-and-Φ using *Load Linked* (LL) and *Store Conditional* (SC) instructions, plus arbitrary arithmetic to implement whatever Φ the heart may desire

Meet the locks part 1: the simple ones

- *Test and Set*
  - Trivial, all participants try to atomically update a bit until they get it
- *Test and Test and Set*
  - Same, but reduces # of write requests by testing whether the lock is already held before going for it
- *Test and Set + Backoff*
  - Lightens write load by imposing an exponentially growing wait on every failure to get the lock

Meet the locks, part 2: the fair ones

- *Ticket*
  - Every acquisition attempt is given a strictly increasing ticket #, processor spins on a counter of served tickets until its number comes up
- *Anderson's Lock*
  - Similar to the ticket lock, but distributes the spin-locations across an array; leaving process unlocks the next one
- *MCS (Mellor-Crummey & Scott) Lock*
  - Much like Anderson's lock, but uses a linked list for more flexibility in memory layout
  - (also suggests an almost-fair implementation using less powerful atomic ops, but that's not so relevant for us...)

The Platforms

Njord: IBM system p575+
16 cores share memory,
Kumar hints at coherency

Embla: SGI Origin 3800
512 cores share mem, ccNUMA,
fat tree + distributed directory

The project work for DT8105

- Started with no greater ambition than to repeat the spin-lock experiments of the MCS paper on contemporary hardware, to see what has changed, and how
- Nontrivial amount of hacking
  - Req. some familiarity with the ISAs of the target architectures
  - 6 little nightmares of debugging/verifying lock correctness
  - Find reasonable statistics for performance measurement (close-to-epsilon time intervals are hard to measure)
- *Project report still stepped in it a bit...*
- ...timings were dominated by sync. on run reinitialization, even when timings per run were ok, making all locks appear similar

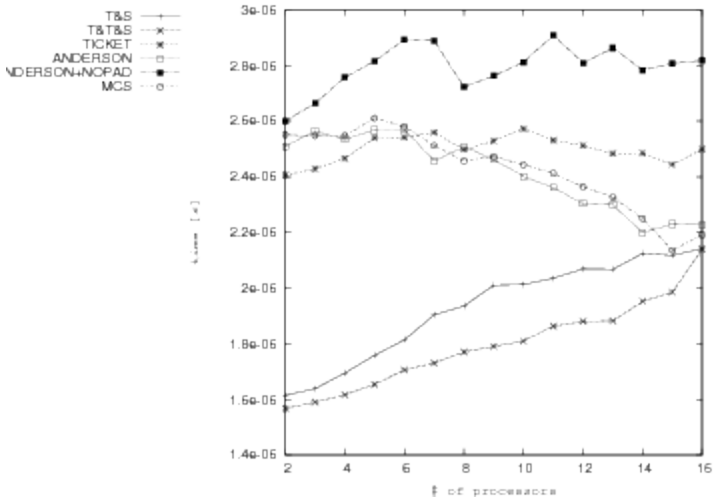
The project work for DT8105

- Lessons learned:
  - Empirical measurement is forgiving: whatever you measure is what you measured
  - Document the methodology, applying all scruples: there's no shame in being wrong
  - Find a good devil's advocate (or two) and bother them repeatedly, applying no scruples whatsoever – it's good to be proven wrong ASAP, and when you're not, it builds confidence
- The perceived value in the report is perhaps structural, it served mostly as a seed for what became publishable work.
- The report deadline is restrictive, but a nice topic lets you complete the course, doing groundwork at the same time

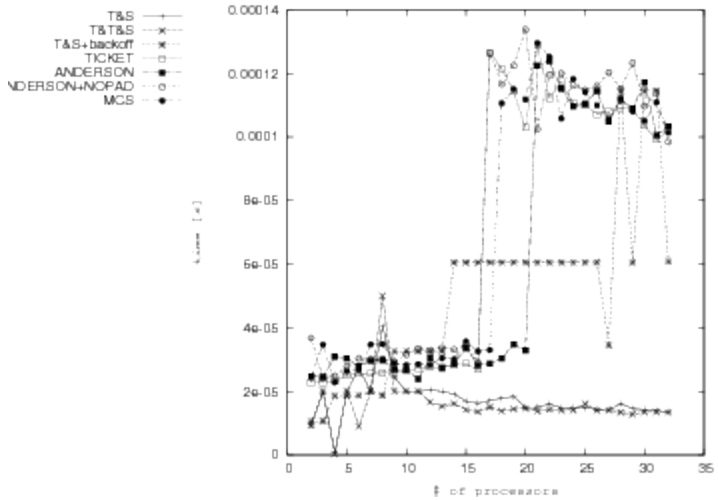
MuCoCos'08, subsequently

- 3 days of hard thinking and bugging the devil's advocates (Magnus, Rune) ironed out the method bug, and showed that even the flawed results were on an interesting trail:

IBM p575+



Origin 3800



NTNU
Norwegian University of Science and Technology

Conclusions and reasoning

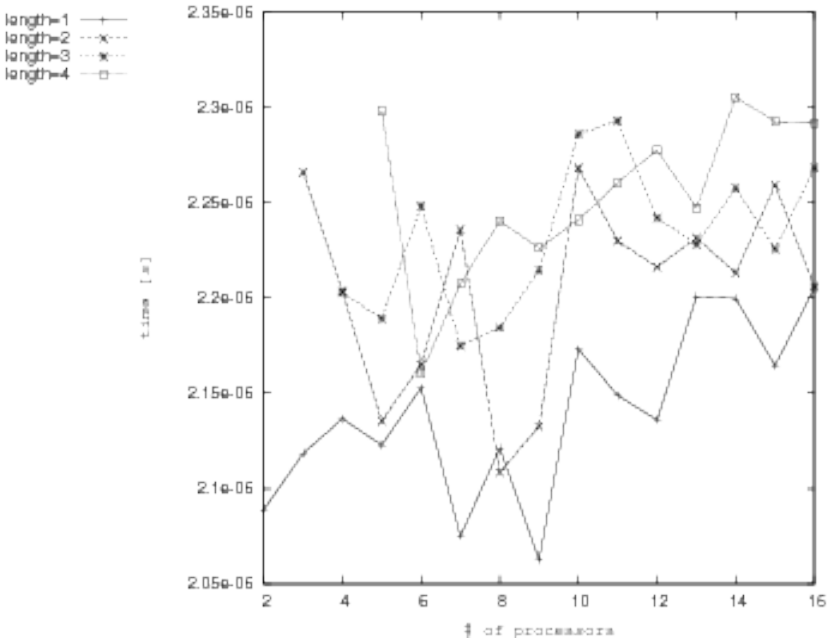
- Categorizing the lock performances showed that
  - Test & set + backoff is not very interesting any more
  - The other locks divide on fairness, with unfair locks scaling surprisingly well
- Important points:
  - The interconnect saturation effect observed in the MCS paper is gone
  - Unfair locks scaling this well indicates that the locality of the lock data structure is very important – a likely cause is starvation of remote processes
  - The Ticket lock behaves much like Anderson, MCS, suggesting that clever memory layout tricks may no longer be as relevant
  - Fair locks are dominated by remote access time, order of magnitude leap for another level of switching on the Origin 3800
- In summary, *fairness is the key performance parameter, and latency is more important than bandwidth*

SCPE Journal edition

- Open question from conference ed.: fairness introduces latency, but what really happens with the unfair locks? (i.e. how severe is the starvation we expect?)
- Introduces queuing (modified T&S, T&T&S) locks, only as vessels to study this behavior (*absolute perf. figures are not particularly good, and fall in with the other fair locks*)
- Modified locks puts the *n* last holders of the lock on a queue, taking them out of the race for the next *n* acquisitions
- Fairness metric shows modified locks strike a middle ground between “pure” fair and unfair

Performance figures of the modified lock

- (Embla was decommissioned in the interim, results from Njord)
- Clear evidence: preventing only the last holder from re-acquiring the lock introduces locality overhead, further queuing does not discriminate



NTNU
Norwegian University of Science and Technology

Conclusions

- Fairness costs, proportional to latency (speed of light)
- Fairness is more critical than it used to be – an unfair lock will be monopolized by one lucky process
- Relative costs of remote vs. local access will require more detailed modeling of the memory hierarchy than has been done so far

Meta-conclusions

- What is necessary to make a decent report/paper?
  - Time
  - Dedication
  - Placing results in a greater context
  - A few good enemies
- What is *not* necessary?
  - Having the right answer from the start
  - Inventing something fantastic
- Why bother?
  - Because the course is the most fun you're likely to have with your clothes on ;)

...are there any questions?