Barrier Synchronization on a Loaded SMP using Two-Phase Waiting Algorithms

Dan Tsafrir and Dror G. Feitelson
School of Computer Science and Engineering
The Hebrew University, 91904 Jerusalem, Israel
{dants,feit}@cs.huji.ac.il

Introduction

- The main synchronization constructs: locks and barriers
- The synchronization dilemma: spin or block?
- Common wisdom for barriers: Always-block
Two-phase blocking

Another solution to the dilemma: two phase blocking

Spinning for the duration of a context-switch: 2-competitive

Problem: lacking a global view
Methodology - The SMP simulator

- An event driven program
- Distinguishes between sync/non-sync jobs
- Simulated applications use two-phase waiting alg.
- Input: machine & workload description
- Output: how well the synchronization policy performed

Methodology - scheduling policy & workloads

- A detailed emulation of the Linux scheduler
- Linux supports 3 policies: FIFO/RR/OTHER
- For general-purpose systems:
  - OTHER is a priority-based preemptive scheduler
- Extensive simulations of six combinations:
  - two scheduling policies: RR and OTHER, and
  - three workloads:
    1) One sync job against backdrop of non-sync threads
    2) Homogeneous set of identical sync-jobs
    3) Heterogeneous mixture
**SSR achieved by Round-Robin**

- Observe two phenomena (homogenous collection):
  1. SSR is above 50% in the intermediate-load range
  2. After which SSR converges to some value below 50%

![Graph showing SSR achieved by Round-Robin](image)

**Alternating-Synchronization**

- The answer to the puzzle: alternating synchronization.
  - Jobs fall into pattern of two alternating sets of threads
- The consequences:
  - SSR converges to positive value < 50%
  - Low effective CPU utilization
  - Common assumption of Poisson arrivals incorrect

![Graph showing Alternating-Synchronization](image)
SSR achieved by OTHER

- One 11-threads job against increasing num of CPU bound jobs
- Results are similar to RR
- SSR shoots up to near 100% for load=16n

Transition point

- The synchronizing-job gains 11 CPUs on a periodic bases
- However, threads perform rapid alt-sync because:
  - running: μ(compute) + CS(spin)
  - awakened: CS(schedule) + μ(compute)
- Rarely, the alt-sync pattern is broken
- Multiple of 16 allows machine to be partitioned
Increase spin duration beyond CS

- Achieve transition regardless of load by increasing spin duration beyond CS
- CS+ shows improvement but not optimal
- Worst case: a thread must wait for its peers to complete blocking, and then to unblock
- 2CS+ provides best combination of stability & performance

![Graph showing performance metrics for different thread counts and grain sizes]

26/3/14 reached here! (took nearly an hour)

Synthetic job mixes

- A single synchronizing job: Allowed to make good progress even when system is heavily overloaded!
- Homogeneous job mix: exactly the opposite
- Heterogeneous job mix: Somewhere in between
SSR of heterogeneous job mix

Spin vs. Always-block

- SSR is not a perfect metric
- Always spin achieves 100%
- Use speedup for comparison
Spin vs. Always-block - Single sync job

Spin vs. Always-block - Heterogeneous mix
Conclusions

- Load is a dominant factor in the decision whether to spin or block
- Spinning for CS is not enough. 2CS+ is optimal
- Alternating-synchronization computation pattern
- Barrier are fundamentally different than locks
- Bug in the Linux scheduler - processors get lost