Everything you always wanted to know about synchronization but were afraid to ask, SOSP 2013

This paper is about understanding the low-level details of the hardware

This paper do HW/SW-level analyze and show the performance tendency (latency and throughput) relate to synchronization properties on different types of architectures (single/multi-socket, uniform/non-uniform socket, directory/broadcast-based).

Hardware-Level Analysis: Local Accesses / Remote Accesses / Locality / Atomic Operations

Software-Level Analysis: Lock / Message Passing / Hash Table / Key-value store

By doing that it imply that scalability of synchronization is mainly a property of the hardware.

They also suggest a cross-platform synchronization suite, SSYNC. which can be used to assess synchronization scalability on different platforms.

Pros

wide analysis on synchronization on various system architecture

Cons

They do experiment on one product for each kinds of HW architecture. (ex. multi-socket Broadcast based - Xeon). I was not sure whether the experimental consequences was about a specific product (Xeon) or a kinds of HW architecture they use(multi-socket Broadcast based)

Ideas

I think it is better to list more product for each kinds of HW architecture.

Read-Log-Update: A Lightweight Synchronization Mechanism for Concurrent Programming, SOSP 2015

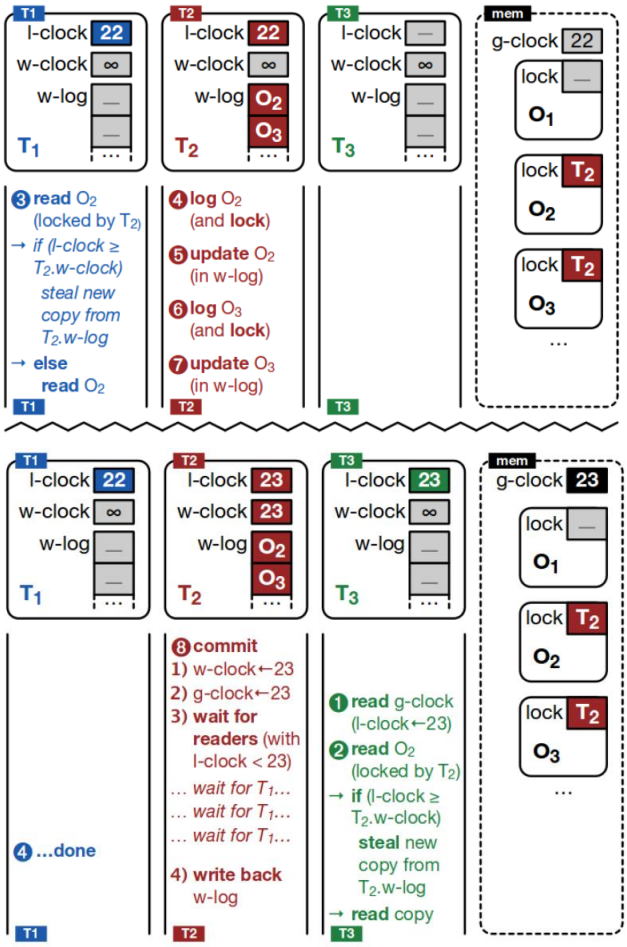
Operation summary:

reader sudo code as I understood

```
curT.l-clock = g-clock
while  target Ot not empty  then
    if  Ot locked  then
        if     curT.l-clock   >   Ot.lockingT.w-clock
then
            steal and read lockingT.w-log.Ot
        else  then
            read Ot
    else  then
        read Ot
```

writer sudo code as I understood

```
while  target Ot not empty  then
        lock Ot
        update Ot in T.w-log
// committing write
curT.w-clock and g-clock = g-clock+1
wait  for  all  reader  done  with  rdT.l-clock  <  g-
clock
while target Ot not empty  then
    Ot = curT.w-log.Ot // write back
```



By adding time stamp implementation, RLU overcome the limitation of RCU, (cannot guaranteeing multi object write atomicity)

Pros

can guaranteeing multi object write atomicity

Cons

working well only on read domestic workload

for adopting the concept of time stamp, global clock added which possibly can raise scalability issues

Ideas

do some optimization on global clock access

An Analysis of Linux Scalability to Many Cores, OSDI 2010

Moving into the multicore era, the use of traditional kernels that do not take into account concurrent processing incurs overhead in multicore targeting applications.

This paper said that we do not have to fully redesign traditional kernel for solving those scalability issues. It said that it is possible to use traditional kernel architecture by fixing some scalability bottlenecks.

it analyze Linux scalability for 7 real apps:

Not to scale well on linux : Memcached / Apache / Metis (MapReduce library)

Designed for parallel execution : gmake / PosgtreSQL / Exim / Psearchy

Some scalability issues on papers and suggested solutions are listed below:

non-scalable locking → Lock-free algorithms / Fine-grained locking

reading mount table → per-core mount caches

reference counting → sloppy counters

About sloppy counter:

Linux uses shared counters for reference counting and to manage various resources. Using shared counter raise scalability issues because of cache coherence

Paper suggest sloppy counter which is a hierarchical counter which has local counters and global counter.

Per core local counter prevents tasks from constantly accessing global shared counter which induce cache lookup and copy for cache coherency. When local counter reach pre-defined threshold it is flushed to global count

Pros

Shows possibilities to solve scalability issues on multicore systems while keeping traditional kernel and application designs

Cons

The results is limited to 48 cores and small set of applications. And done in In-memory FS instead of disk

Ideas

Should do experiment with more cores and do things on general environment not only on In-mem FS.

The Scalable Commutativity Rule: Designing Scalable Software for Multicore Processors, SOSP 2013

Current workload-driven approaches has problems that it find scalability bottlenecks bound to current workloads and current environments (ex. # of cores). The paper said that the real bottlenecks, which may not found by testing on limited number of workloads, may be in the interface design. And they suggest interface driven approaches on solving scalability issues

And the interface-driven approach (this paper introduce) based on a rule :

Whenever interface operations commute, they can be implemented in a way that scales

Logistic of the rule:

Operations commute ➔ results independent of order ➔ communication is unnecessary ➔ without communication, no conflicts

COMMUTER is a tool they made based on the rule. It automates checking if an implementation achieves conflict-freedom whenever operations commute. (testing conflict freedom **analyze interface** commutativity and **test** that an implementation **scales** in commutative situations)

Pros

This approach does not require target workload or a physical machine to reason about scalability it check scalability of interfaces

Cons

Considering backward compatibility, even if it found scalability issue on existing interface, it is hard to applying them. But will be effect on building new interfaces or implementing new software

Ideas

Since COMMUTER just analyze and shows the test results, fixing araised problem needs developer. It will be better if the optimization process was automated