Memory Coherence in Shared Virtual Memory Systems
Kai Li and Paul Hudak, ACM TOCS '89

Invents software distributed shared virtual memory

- lots of work prior to this on hardware shared physical memory
- elevates to software shared virtual memory
- all of the same issues, just with page-granularity instead of cacheline, and a slower network

Demonstrates rich set of memory coherence protocols and ownership policies

- page synchronization and ownership, who tracks, when

Why DSM?

- a set of problems that people want to parallelize
- two basic programming models:
    - message passing: separate address spaces, use RPC (or other model) to copy data between address spaces
        - programmer decides:
            - when to communicate
            - with whom to communicate
            - what data to send
        - pluses:
            - can minimize or optimize communication
            - no hidden overhead, as communication is explicit
        - minuses: complex, programmer is responsible for coherence and consistency

    - shared memory: program as though it were threads in one address space
        - no explicit communication
            - no need to decide when to communicate – system does it for you (coherence/consistency)
            - no need to decide with whom to communicate – don't even need to name other parties in system
            - no need to decide what data to communicate – shared address space means everything is accessible
        - but
            - usual concurrency issues (synchronization, races)
            - unit of sharing (data structure elements) is hidden to communication/coherence subsystems (pages)
            - crashes hurt everybody

Major issues in a SDSM system

- how do you detect modifications?
  - universal answer: VM hardware
- what coherence/consistency model?
  - sequential consistency vs. release consistency – will get into this
- what granularity of coherence?
  - pages: 2K/4K; see answer 1
- is programmer aware of local vs. remote?
  - Cache-aware programming; kind of defeats the purpose, but turned out to be pretty necessary.

Granularity of coherence

- page size?
- tradeoff:
  - smaller pages: less likely to have “false sharing”. (This paper calls it contention.)
  - larger pages: less overhead
    - throughput vs. message size
    - fewer pages spanned for same data structure – fewer DSM ops
    - less metadata

- false sharing
  - imagine two independent variables that compiler puts on same page
  - two processors writing/reading independently
  - ping pong effect

- particularly egregious case: synchronization
  - when synchronization variable experiences contention:
    - algorithm is not making forward progress (loss of scalability)
    - frequent writes (e.g., compare-and-swap)
    - so, lots of invalidations getting pushed all over

Basic scheme

- each page has an “owner”
  - owner is person who did most recent write
  - only one owner per page: why? [coherence]
- replicate pages if doing read-sharing
  - so don’t need to cross communication network for every load
- invalidate replicas when owner does a write
  - to enforce coherence
- if want to write to page, must first become owner
  - implies stealing rights from current owner

Go over centralized directory algorithm:

Read fault handler:

```
Lock( PTable[ p ].lock );
IF I am manager THEN BEGIN
    Lock( Info[ p ].lock );
    Info[ p ].copyset
        := Info[ p ].copyset ⋃ {ManagerNode};
    receive page p from Info[ p ].owner;
    Unlock( Info[ p ].lock );
    END;
ELSE BEGIN
    ask manager for read access to p and a copy of p;
    receive p;
    send confirmation to manager;
    END;
PTable[ p ].access := read;
Unlock( PTable[ p ].lock );
```

Read server:

```
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
    PTable[ p ].access := read;
    send copy of p;
    END;
Unlock( PTable[ p ].lock );
IF I am manager THEN BEGIN
    Lock( Info[ p ].lock );
    Info[ p ].copyset
        := Info[ p ].copyset ⋃ {RequestNode};
    ask Info[ p ].owner to send copy of p to RequestNode;
    receive confirmation from RequestNode;
    Unlock( Info[ p ].lock );
    END;
```

Write fault handler:

```
Lock( PTable[ p ].lock );
IF I am manager THEN BEGIN
    Lock( Info[ p ].lock );
    Invalidate( p, Info[ p ].copyset );
    Info[ p ].copyset := {};
    Unlock( Info[ p ].lock );
    END;
ELSE BEGIN
    ask manager for write access to p;
    receive p;
    send confirmation to manager;
    END;
PTable[ p ].access := write;
Unlock( PTable[ p ].lock );
```

Write server:

```
Lock( PTable[ p ].lock );
IF I am owner THEN BEGIN
    send copy of p;
    PTable[ p ].access := nil;
    END;
Unlock( PTable[ p ].lock );
IF I am manager THEN BEGIN
    Lock( Info[ p ].lock );
    Invalidate( p, Info[ p ].copyset );
    Info[ p ].copyset := {};
    ask Info[ p ].owner to send p to RequestNode;
    receive confirmation from RequestNode;
    Unlock( Info[ p ].lock );
    END;
```

Questions:

- Why do you need to Lock (PTable[p].lock])?
    - Why is OK for different pages to have their own lock?
- Why does the manager need to get confirmation from nodes on transfer of ownership?

Hard question: does IVY provide sequential consistency?

- claim yes; meets the requirements we went over in previous part
- **Program order requirement**
    - a processor must ensure that its previous memory operation is complete before proceeding with its next memory operation in program order.
        - ensuring completeness means getting an acknowledgement back from cache or memory
    - in a cache-based system, a write must generate invalidate or update messages for all cached copies, and the write can be considered complete only when the generated invalidates and updates are acknowledged by the target caches.

- **Write atomicity requirement**
    - writes to the same location must be serialized (i.e., writes to the same location be made visible in the same order to all processors)
    - the value of a write cannot be returned by a read until all invalidates or updates generated by the write are acknowledged (i.e., the write is visible to all processors)

Wrinkles:

- distributed directory
    - hash on page to partition manager functionality
    - exactly the same
    - issue is of load balancing:
        - some pages hotter than others
        - can end up with uneven ownership management overhead across nodes, which will slow down entire computation
        - also, more natural for current owner to manage, on assumption that owner will do more writes, and hope that others are just read

- dynamic distributed directory
    - keep forwarding pointers to current owner
        - pass ownership on both read and write fault.
            - on read fault just to make analysis easier; optimize away
    - how long do chains get?
        - invalidation broadcast truncates chains – want to do it every now and then

Speedup curves -- Y = speedup, X = # processors

- perfect parallelization:
    - X=Y line
- superlinear:
    - slope > 1 for a while – working set effects
- sublinear:
    - either problem itself isn't parallelized well
        - there are some sequential components
        - algorithm or architecture!
    - or DSM system introduces increasing overhead
        - e.g., false sharing
        - e.g., broadcast traffic increases

Wrapup:

- In the end, software DSM wasn't widely used
- message passing dominated
    - specialized message passing systems (MPI)
    - one-time cost of getting right, perpetual benefit of running faster
- hardware DSM works great for small scale systems
    - for larger scale systems, non-sequential consistency models
    - programmers need to think again!
- coming back to us in spades with multicore systems
    - hardware cache coherence; today's L2 cache is about the same size as processor memory back then. (multikernel argument)