Implementing caches

Doug Woos

What if clients use a sharded key-value store to coordinate their output?
Or CPUs use memory to coordinate?

Write buffering: Can we start to write $done_1$ before we finish write to $k_1$?
Yes, if order enforced at the server

What if caches can hold out of date data?
What might go wrong?
Rule for caches and shards

Suppose each process specifies operations in some order

Sequentially consistent if:
1. Operations applied in processor order, and
2. All operations to a single key are serialized (as if to a single copy)

How do we ensure #2?
- Can study each memory location in isolation

Invalidations vs. Leases

Invalidations
- Track where data is cached
- When doing a write, invalidate all (other) locations
- Data can live in multiple caches during reads

Leases
- Permission to serve data for some time period
  - (if weak) eventually consistent
  - (if strong) Wait until lease expires before update

Write-through vs. write-back

Write-through
- Writes go to the server
- Caches only hold clean data

Write-back
- Writes go to cache
- Dirty cache data written to server when necessary

Write-through vs. write-back

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Invalidations</th>
<th>Leases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write policy</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Write-through</td>
<td>AFS (Andrew FS)</td>
<td>DNS</td>
</tr>
<tr>
<td>Write-back</td>
<td>Sprite</td>
<td>NFS</td>
</tr>
</tbody>
</table>

Write-through invalidations

Track all caches with read copies

On a write:
- Send invalidations to all caches with a copy
- Each cache invalidates, responds
- Wait for all invalidations, do update
- Return

Reads can proceed:
- If there is a cached copy
- or if cache miss and no write waiting at server
Client

\[
\text{put (k1, f(data))}
\]
\[
\text{put (done1, true)}
\]
\[
\text{while(get(done1) == false)}
\]
\[
\text{put (k2, g(get(k1))}
\]
\[
\text{put (done2, true)}
\]
\[
\text{while(get(done2) == false)}
\]
\[
\text{rslt = h(get(k1), get(k2))}
\]

\[
done1=false
done2=false
\]
Questions

While a write is waiting on invalidations, can clients read old values from caches?

Questions

While a write is waiting on invalidations, can the writing client perform a different write?

Questions

While a write is waiting on invalidations, can the server process a read to a different location?

Questions

While a write is waiting on invalidations, can the server process a read to the same location?

Questions

While a write is waiting on invalidations, can the server process a write to a different location?
Questions

While a write is waiting on invalidations, can the server process a write to the same location?

More Questions

Why does the server wait until write is applied before returning to the client?
Why queue incoming requests during a write?
How much directory state is needed at server?

Write-back invalidations

Track all reading and writing caches
On a write:
- Send invalidations to all caches
- Each cache invalidates, responds (possibly with updated data)
- Wait for all invalidations
- Return
Reads can proceed when there is a local copy
Order requests carefully at server
- Enforce processor order, avoid deadlock

MSI/MESI

Protocols used for processor caches
Similar to protocol used e.g. in Sprite
Useful to understand

MSI

Three cache states:
- Modified: this is the only copy, it’s dirty
- Shared: this is one of many copies, it’s clean
- Invalid
Allowed states between pairs of caches:

Client

put (k1, f(data))
put (done1, true)
while(get(done1) == false);
put (k2, g(get(k1));
put (done2, true)
while(get(done2) == false);
rslt = h(get(k1), get(k2))

Server

k1 = 0
k2 = 0
done1 = false
done2 = false
Client

```
put (k1, f(data))
put (done1, true)
```

Server

```
k1 = 42; M
done1 = false; I
done2 = false; S
```

```
k1 = 0
done1 = false
```

```
done2 = false
```

```
k1: America
done1: America
done2: Africa
```

```
rslt = h(get(k1), get(k2))
```

Client

```
while(get(done1) == false)
```

```
put (k2, g(get(k1));
put (done2, true)
```

Server

```
k1 = 42; M
done1 = true; S
```

```
done1 = false; I
```

```
done2 = false; S
```

```
k1 = 0
done1 = true
```

```
done2 = false
```

```
k1: America
```

```
done1: America, Asia
```

```
done2: Africa
```

Client

```
while(get(done1) == false)
```

```
put (k2, g(get(k1));
put (done2, true)
```

Server

```
k1 = 42; M
done1 = true; S
```

```
done1 = true; S
```

```
done2 = false; S
```

```
k1 = 0
done1 = true
```

```
done2 = false
```

```
k1: America
```

```
done1: America, Asia
```

```
done2: Africa
```

Client

```
while(get(done1) == false)
```

```
put (k2, g(get(k1));
put (done2, true)
```

Server

```
k1 = 42; M
done1 = true; S
```

```
done1 = true; S
```

```
done2 = false; S
```

```
k1 = 0
done1 = true
```

```
done2 = false
```

```
k1: America
```

```
done1: America, Asia
```

```
done2: Africa
```
Client

\[
\text{put (k1, f(data))}
\]

\[
\text{put (done1, true)}
\]

\[
\text{while (get(done1) == false)}
\]

\[
\text{put (k2, g(get(k1));}
\]

\[
\text{put (done2, true)}
\]

\[
\text{while (get(done2) == false)}
\]

\[
\text{rslt = h(get(k1), get(k2))}
\]

Server

\[
k1 = 42; M
\]

\[
done1 = true; S
\]

\[
done2 = false; S
\]

\[
k1 = 42; S
\]

\[
done1 = true; S
\]

\[
done2 = false; S
\]

\[
k1 = 42; S
\]

\[
done1 = true; S
\]

\[
done2 = false; S
\]

\[
k1 = 42; S
\]

\[
done1 = true; S
\]

\[
done2 = false; S
\]

\[
k1 = 42; S
\]

\[
done1 = true; S
\]

\[
k2 = 43; M
\]

\[
done2 = false; S
\]

\[
k1: \text{America, Asia}
\]

\[
done1: \text{America, Asia}
\]

\[
done2: \text{Africa}
\]

\[
k1: \text{America, Asia}
\]

\[
done1: \text{America, Asia}
\]

\[
done2: \text{Africa}
\]
Motivation:
- Common pattern: i++ (read, then a write)
- MSI inefficient when doing a read and then a write
- If no one else has a copy, can “claim” it with the read

Four cache states:
- **Modified**: this is the only copy, it’s dirty
- **Exclusive**: this is the only copy, it’s clean
- **Shared**: this is one of many copies, it’s clean
- **Invalid**
B.1: MESI allowed states

<table>
<thead>
<tr>
<th>M</th>
<th>E</th>
<th>S</th>
<th>I</th>
</tr>
</thead>
<tbody>
<tr>
<td>✅</td>
<td>✅</td>
<td>✅</td>
<td>✅</td>
</tr>
</tbody>
</table>

It is assumed that k1 is "Exclusive" to N. America after first read.
Can modify without sync.

B.2: False Sharing

Expensive to keep track of MESI for every memory location.
Instead, coarse-grained record-keeping
- On CPUs, at the cache line granularity
- In file systems, at the granularity of a file/file block
What if two clients try to modify different memory locations in the same cache line, concurrently?
- Cache line can only be "modified" in one at a time
- Correct behavior, but slow

B.3: Atomic Read-Modify-Write

RMW needed to implement spinlocks and other sync
Request cache line exclusive/modified
Delay concurrent remote read/write misses until entire operation completes

B.4: Software Transactions (CPUs)

Often want multiple instructions to execute atomically
- Critical section, supported in hardware
May involve multiple cache lines
Execute normally: acquire cache lines in MESI state
If remote miss during the software transaction
- Abort transaction, erase modifications, and try again
If reach end of software transaction without remote miss
- Success!
Distributed transactions (with node failures) next time!

B.5: Caching implementations

<table>
<thead>
<tr>
<th>Mechanism</th>
<th>Invalidations</th>
<th>Leases</th>
</tr>
</thead>
<tbody>
<tr>
<td>Write-through</td>
<td>AFS (Andrew FS)</td>
<td>DNS</td>
</tr>
<tr>
<td>Write-back</td>
<td>Sprite</td>
<td>NFS</td>
</tr>
</tbody>
</table>
**Strong leases**

Read request: key, TTL (time to live)
When server returns:
- It won’t accept writes to the key
- For TTL seconds after reply sent
Client invalidates its cache after TTL seconds
- From when request was sent
Assumes bounded physical clock sync

**Strong leases**

For write-through:
- Server queues writes until all leases expire
- Avoid starvation: don’t accept new reads
For write-back:
- Cache can get a write lease (exclusive)
- Server queues read requests until lease expires

**Clock issues**

How long should the server wait on a lease?
How long should the client wait on a lease?
What about clock skew?
- Add \( \varepsilon \) on server, subtract \( \varepsilon \) on client

**Strong leases vs. Invalidations**

What are advantages/disadvantages of each?

**Strong leases vs. Invalidations**

What are advantages/disadvantages of each?

- Strong leases potentially slower
- What if a cache fails when it has a key? Strong leases provide better availability
Can combine techniques
- Short lease on entire cache, periodically revalidated
- All keys invalidated on failure (after lease)

**Weak leases**

Cache valid until lease expires
Allow writes, other reads simultaneously
Semantics?
Weak leases

Examples: NFS, DNS, web browsers

Advantages
- Stateless at server (don't care who is caching)
- Reads, writes always processed immediately

Disadvantages
- Consistency model (!!!)
- Overhead of revalidations
- Synchronized revalidations

Discussion

“Complexity” as a downside
Do the scalability/performance issues mentioned in the paper exist today?
Why do we use NFS?