CACHING

Cache and Memory Consistency

Caching is a technique used to increase performance by reducing the cost of accessing system objects (such as blocks in main memory, names from a name server, etc.).

Implementations can be in hardware or software.

Caching implementations try to:

- maximise the number of operations that can be done from the cache (choosing the right information to reside in the cache – replacement policy)
- reduce the loss of information when the server containing the cache fails (depending on the importance for system consistency of the information being cached)
- minimize the cost of keeping caches consistent
- reduce the likelihood that the cache does not reflect the real value of shared system objects (it doesn't matter in all situations, but in some circumstances it is critical)

Two general classes of cache protocols are: snoopy and directory.

Cache coherence schemes use hardware, software, or a combination of both. (One summary article is *A survey of cache coherence schemes for multiprocessors* by Per Stenström in IEEE Computer, June 1990)

Cache coherence protocols are complicated by problems that arise when data is changed.

- Read only objects are OK
- private read/write objects might be a problem if the process owning them migrates (resulting in the object being cached in a different cache)
- shared read/write objects have consistency problems

With **weak cache consistency** a client may see an inconsistent view of cache data. This happens on an access when a remote cached copy of the data has been changed but other copies (including the local copy) have not yet been invalidated or updated. **Strong cache consistency** will not permit this.

Maintaining Consistency and Recovery

Avoiding out of date cache information:

- If the information is critical for system consistency, a write back to the system object (not just to the cache) can be forced, or
- cache information can be expired after a set time, forcing the information to be reread

and for other caches holding the same information:

- invalidate all entries referring to the same information, or
- update all the other cache entries

System crashes (loss of cache contents):

- A cache for read operations only do not pose any recovery problems after a crash. The cache is refilled as needed.
- Where writes are cached, any changed objects in the cache (not flushed) will be lost possibly leaving inconsistencies.

Because distributed systems are implemented over a network rather than a bus and the network has greater latency than a bus (proportion of time spent to do a remote access compared with time spent for a local access) the use of caching becomes even more important.

The usual bus based algorithms for cache consistency are not suited for a message passing system:

- Network latency makes some bus based protocols too expensive on a network.
- Caching protocols that rely on an atomic memory request cycle cannot be duplicated on a network. (If the protocol relies on an acknowledgement to tell whether the remote processor has done an invalidation as requested, the acknowledgement makes the operation non-atomic.)

write invalidate (all extra copies of an object are invalidated) and **write update** (all extra copies of an object are updated) policies require that cache invalidation and update commands (called **consistency commands**) be sent to at least those caches having copies of the objects.

In the **snoopy** protocols, consistency commands are broadcast to all caches. Each cache decides whether it has a copy of the object and decides what is appropriate action.

In **directory** based protocols, consistency commands are only sent to caches containing copies of the cached object.

Write Update Snoopy Cache Protocol

This is a brief description of one snoopy cache protocol.

Possible object states are:

- **valid-exclusive** (this is the only cached copy and it is consistent with the memory copy)
- **shared** (this copy is consistent with main memory but other consistent copies exist also)
- **dirty** (this is the only cached copy and it is inconsistent with memory)

The protocol ensures that other states do not occur.

Actions are:

- Read miss
  - if shared copies exist — caches holding the object supply it by synchronising transmission on the bus
  - if a dirty copy exists — the cache holding it supplies it and updates main memory (new state is shared)
  - no copies are cached — the object is read from main memory and cached (new state is valid-exclusive)
- write hit
  - dirty or valid exclusive — write the changes locally (the new state is dirty)
  - shared — all cached copies and the copy in memory are updated (making this a write-update protocol)
- write miss
  - a copy is read in from memory (new state is dirty)
  - a copy (dirty) is supplied from another cache and all copies plus the memory copy are updated (new state is shared)
- replacement
  - if the cached copy is dirty it is written back to memory
  - otherwise, nothing need be done.

Directory Schemes

Where a broadcast is too expensive (e.g., where there are large numbers of processors), a directory of caches containing the copies of an object may be kept. This is called a **directory** protocol.

A directory is used to store information as to where copies of an object reside.

Different schemes vary mainly in how the directory maintains the information and where it is stored. For example, in one scheme a flag field is associated with each object (a presence flag vector). One bit for each cache indicates which cache has a copy of the object. (other status information may be required depending on the cache coherence policy)

Directories can be centralised or distributed.

Caching performed by a server will use some variation of a directory based caching scheme.

A simple example using the directory scheme to identify where copies reside.

When there is a read miss in one cache (A) and there is one dirty copy of the block in another cache (B):

- the request is goes to the memory controller (which knows that the block is at B)
- the memory controller retransmits it to the cache holding the copy (at B)
- which supplies the updated block to A.

When there is a write hit at cache B, that cache sends invalidations to all other caches marked as having copies of the page.

If all caches have a directory entry it is called a **full map directory scheme**. Other schemes exist to reduce the directory size, which can get expensive when large numbers of caches exist (e.g., a hierarchy of caches with distributed directory).

Software Controlled Caching using a Hardware Cache

Inconsistent cached data can be avoided by caching only data that is “safe” to cache (this reduces the complexity of cache hardware). This data can be identified during compilation. For example, a very simple scheme might mark all writable data as non-cachable. However, it is possible to do better than that.

Types of access:

1. read only, multiple processes — always cachable
2. read only for multiple processes and read/write for one — only the process able to write to the data may cache it, but must use a write-through update policy to ensure that the cached copy is consistent with main memory (so that the other processes read a consistent copy)
3. read/write for one process — cachable using copy-back (copy to memory when cache flushed).
4. read/write for multiple processes — non-cachable

To implement, the following cache instructions are needed:

- cache-off — all shared data accesses bypass the cache
- cache-on — cache shared variables of type 1 or 3
- cache-invalidate — indiscriminate invalidation (all data is invalidated) or selective (where only those variables that could introduce inconsistencies are invalidated)

Selective validation requires more instructions and a change bit.

- memory read
- cache read
- cache invalidate

If the change bit is true then a read request will be passed to memory and then the block loaded into the cache.

Cache (In)validation (Server)

In an environment where weak consistency is acceptable, client cache contents may be re-validated by:

- the client contacting the server to revalidate its data (e.g., using time to live in DNS to invalidate the data and force another access to the server at regular intervals)
- the server notifying the client when cached data is invalid and the client revalidating the data when next needed. This is more complex but may reduce client server interaction
  - only clients holding the information need be told and
  - they only need to be told when a change occurs

  When strong consistency is required, changes to the data must be propagated to all interested parties.

Propagation of Modifications (Server)

Modifications may be propagated from the client to the server

- when all changes to the cached object are completed (as in the Andrew file system)
- after a fixed length of time
- when dirty pages need to be reclaimed

Deferred propagation

- improves performance (several changes may be made for each write back)
- increases the risk of the server being out of date in the event of a client crash

Distributed Shared Memory

Distributed shared memory, where more than one process (or thread) from more than one host has access to the (logically) same piece of memory, is primarily an alternative to message passing for interprocess communication. The basic operations are read and write.

An advantage is that marshalling is not required. (e.g., no problems with pointers) The Distributed shared memory can also be persistent.

A disadvantage is that access to the shared memory needs to be controlled ((distributed) mutual exclusion).

Shared memory appears as part of the address space of each process sharing it. This makes it an easy concept to understand and algorithms for programming with it already exist, making a good candidate for adaption to a distributed environment.

Algorithms Implementing Distributed Shared Memory, Michael Stunm and Songnian Zhou, IEEE Computer May 1990, pp 54 - 64.

Algorithms for implementing distributed shared memory:

- central-server
- migration
- read replication
- full replication

Central Server Algorithm

A central server manages all accesses to shared data and only one copy of it is maintained.

A read operation involves sending a request to the server and receiving back the requested data.

A write operation involves sending a request and receiving an acknowledgement.

recovery:

- read requests are idempotent and can be retried
- by using sequence numbers, write requests may be resent by the client and duplicates detected and ignored by the server.
- A failure condition can be raised after several timeout periods which pass without any response.
- Unusable when central server dies.

Migration Algorithm

This is a single reader/single writer protocol (only one host has access to the data at a time). Data is migrated to the site where it is accessed.

There are no communications costs while the date is on location (multiple accesses per migration reduce the overall cost).

Heavily used pages may spend a high proportion of their time moving as compared with the number of accesses while at a location (thrashing) which leads to poor performance.

Locating a remote block can be done:

- using multicasting
- by having a server that knows the location of all blocks (block management can be partitioned between hosts). Additional messages may be required to locate a particular block server, depending on the method used.

Read Replication Algorithm

Multiple reads can occur simultaneously, however when a write operation occurs replicas have to be invalidated or updated to maintain consistency (multiple reader/single writer protocol).

A read or write fault causes the fault handler to contact the owner of the block.

For a read on a non-local data item it is necessary to communicate with other sites to:

- acquire a read-only copy of the block
- change the access rights on any writable copy of the block to read-only

For a write to data that is non-local or for which the local host has no write permission:

- determine the location of the block and send a request
- the block is returned to the requesting host
- an invalidation request is sent out to all other hosts holding (read only) copies of the block

The efficiency of the algorithm can be improved by designating an **owner** for each block that is responsible for maintaining a list of servers (a **copyset**) having read-only copies of the block.

Full Replication Algorithm

This algorithm allows data blocks to be replicated while being written to (multiple readers/multiple writers). This requires accesses to data to be sequenced properly or controlled to ensure consistency.

This can be done by sequencing write operations globally while only sequencing read operations relative to the write operations that occur local to the site where reads are executed.

(This is similar to the write update algorithm for cache consistency implemented in hardware)

One way to do this is to use a **sequencer** which is a process on one of the hosts participating in distributed shared memory.

The intended modification is sent to the sequencer, which adds a sequence number to the modification request and multicasts it to all interested sites.

Each site processes the writes in sequence order.

If a gap is detected by a host it requests a retransmission (which implies that a log of requests is kept).

Ivy — a directory based scheme

Ivy's caching is:

- a multiple reader/single writer algorithm.
- directory based with acknowledgements to invalidation.

It is superficially a little like its bus based equivalent, but different in practice.

Ivy took two different approaches:

- a centralised memory allocator
- each node having its own allocator in addition to a centralised one from which the local ones requested a large chunk of memory and then managed local allocations requests.

A fault is raised when a shared address is accessed and the host has no right of access.

The cache state and corresponding Ivy memory access rights are as followed:

cache	Ivy
dirty	write
shared dirty	owned read
valid	read
invalid	nothing

e.g., a cached block with status "dirty" is equivalent, in Ivy, to the host having write permission on a page (because it has to be assumed that the host has taken advantage of that permission).

Each host has a page table with entries that contain at least:

- access rights
- physical location of page
- copyset (those hosts that have a copy of the page)
- a lock (for synchronisation — prevention of multiple faults on the same page)

Read fault

- find the owner of the page
- owner adds faulting host to copyset
- owner change its access rights to read only
- owner sends page to faulting host

Write fault

- find the owner of the page
- owner sends page and copyset to faulting host and marks its copyset to invalid (because it is no longer the owner)
- invalidations based on the copyset are sent out
- acknowledgement to invalidations come back and the process proceeds

In Ivy three caching protocols were tried, differing mainly in the way the directory was handled.

Ivy's Centralised Caching protocol

A central page manager keeps track of the owner of all pages.

The identity of the page manager is well known to all hosts.

Read fault:

- the faulting host sends a read request to the page manager
- the page manager sends the request to the page owner
- the page owner adds the faulting host to the copyset and sends the page back to it

write fault

- as per the read fault, but the central page table is also modified so that the owner of the page becomes the faulting host
- when a response is received by the page manager pages held by other hosts are invalidated

A read fault requires 3 messages, a write fault requires 4 plus the messages required for invalidating pages (the number of messages equal to the number of hosts recorded in the copyset).

Ivy's Fixed Distribution Caching Protocol

Every processor is given a predetermined subset of the pages to manage.

When a fault occurs, the mapping function is consulted and the page request sent to its page manager.

The number of messages is similar to the previous scheme.

Ivy's Dynamic Distributed Caching Protocol

The central page manager is eliminated and the page table entry has an additional "probable owner" field (called a **hint**).

On receipt of a request on a page it doesn't own, it forwards the request to the probable owner.

Hints are updated when a host:

- receives an invalidation request (it knows there must have been a transfer of ownership)
- relinquishes ownership
- receives a page (it becomes the new owner)
- forwards a page fault request (on a write it will be the new owner, after a read request is satisfied it will know the right owner — even though it may take numerous accesses to find the page)

In the worst case this scheme may take $N - 1$ messages to find the owner, however hints make it much better than that on average.