6.826 Principles of Computer Systems **Fall Semester, 1992**

Lecture 24: December 9, 1992

Lecturer: Bill Weihl *Scribe: Mitchell Charity*

Caching (cont)

This lecture continues our discussion of concurrent caching (Handout 50).
Lecture topics:

- Cache coherence, using Update, Mutex, and competitive strategies
- Competitive algorithms
- Caching in distributed systems
- Limits of caching
- The coherence abstraction

1 Handouts

Lecture Notes #19 and #20.

2 Cache coherence - using Update

To implement coherent memory out of multiple caches, one must maintain cache coherence. Last time, we discussed using invalidation to maintain cache coherence. To recap: one writes to main memory after having invalidated all other caches, and reads are then done from main memory. This can be expensive when the processor is close to the cache, but distant from the main memory.

Update is another way to maintain cache coherence. Rather than invalidating other cache copies on a write, they are updated directly to the new value. Further, the update spec given in Handout 50 differs from invalidation spec in allowing data to be loaded from another cache besides the main memory. Note that the important invariant: "Data in cache is current" is still maintained.

The key issues in deciding the performance tradeoff between invalidation and update are the application's pattern of usage, and the structure of the communication layer. Invalidation is suited for successive local writes because the first write invalidates other cache copies so that consequent writes are purely local operations. On the other hand, updating favors reads by increasing the hit ratio, because cached copies are not invalidated due to remote writes. Thus, if an application thread updates a variable many times between reads by other threads, invalidation is appropriate. But if other threads frequently use the updated variable, the cost of update broadcast can be less than the cost of cache miss messages.

We now evaluate the cost of updating and invalidation for a bus-based network and a point-to-point network separately. The cost is measured in terms of number of messages exchanged. Assume that a processor does k successive writes on a variable x, which is followed by the other n processors reading x.

We consider bus-based networks first. If invalidation is used, the first write broadcasts an invalidate message. Consequent writes do not need any bus transaction. However, later, each of the other n processors miss on trying to read x, resulting in further n transactions. In the update strategy, the k updates each require k update broadcasts. Thereafter, each processor can read its local value of x.

Now, we consider point-to-point networks. With invalidation, the first write results in n invalidation messages to other processors. Remaining writes do not require any messages. Later, each of the n processors needs a net transaction to fetch x. With update, each of the k update broadcasts comprises n messages. No further messages are required for reading x.

	Bus	Net
Update	k	kn
Invalidate	n+1	2n

Thus, a bus-based network often favors update, while a point-to-point network might favor invalidation.

3 Cache coherence - using Mutual Exclusion

The final approach we consider maintains cache consistency using mutual exclusion rather than invalidation or update. Previously, we used R/W locking, permitting one writer or multiple readers. Mutual exclusion allows only one usable cached copy, thus permitting exactly one writer or reader.

The advantage of mutual exclusion is simplicity, of both abstraction and implementation. Internal actions can move data between memory and cache fairly freely.

The invariant about data in a cache is different. Cached data is only current if it is locked.

```
  c(p)!a /\ locked(a) => Current(p,a)
rather than just
  c(p)!a              => Current(p,a)
```

Locking is used explicitly, rather than implicitly by having only a single cached copy. The key invariant is that dirty data is locked.

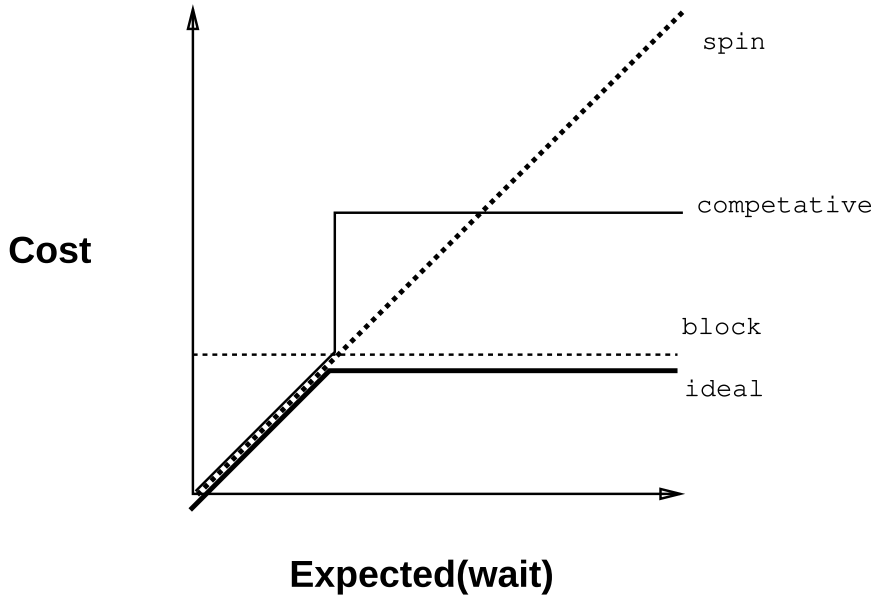

Figure 1: A Competitive Strategy to Acquire a Mutex

The implementation of coherent memory using mutual exclusion is close to that of incoherent memory. Only difference is that `Read` and `Write` now have a lock guard, and `Acquire` and `Release` routines are available. Indeed, this scheme is often used when the underlying hardware provides incoherent memory; the software uses mutexes or critical regions to make it coherent.

`Acquire` does a `Barrier` to guarantee that the value will be fetched afresh, so that it reflects earlier writes. `Release` does a flush to guarantee that the value fetched by the next `Acquire` is up-to-date.

The spec shows data being transferred between caches via main memory. This is just for simplicity. Cache to cache transfers are perfectly reasonable.

4 Competitive algorithms

Another way to maintain cache consistency is to use a combination of invalidation and updates. The choice, of which approach to use in a given circumstance, is made based on the expected application behavior. This is called a “competitive strategy”.

We saw an example earlier in the term: with mutexes. A thread wanting to gain a mutex needs to wait till the mutex is freed. One way of accomplishing this is to spin, repeatedly testing the condition. Another is to block, swapping out until the condition is met. Spinning has a small, but ongoing, cost. Blocking has a fixed large cost of thread swapping. The ideal combination would be to spin for short waits, and block for long ones. A practical combination is to spin for a period equal to that it takes for blocking, and then block.

There has been a lot of research over the last 4-5 years on the theory of competitive algorithms. An algorithm's performance is compared with the best that might be done with complete knowledge of the future. For example, the strategy described above for waiting for a mutex costs no more than twice of an ideal scheme. A practical randomization optimization was recently published which brings this factor down to 1.58.

Cache consistency may increasingly be implemented in software, together with some hardware support. This permits tailoring the competitive algorithm to individual applications, bringing down the factor of 2.

5 Distributed systems and caching

Everything which has been said about cached systems also applies to distributed systems. So how do distributed systems differ from coherent ones? In having an issue of **fault tolerance**. One must address the impact of faults. Define their semantics. Develop implementation techniques.

Distributed systems can be said to have two flavors of failure: machine and network.

In considering a distributed system whose machines have caches, one obvious concern is persistence of writes. One might write "foo" to cache, and then crash, losing "foo". But this behavior is already accepted in non-distributed systems to allow write-behind. In both cases, one accepts an incoherent memory spec, rather than requiring a stronger coherent one which ignores failure.

A more serious complication of machine failure is in the availability of the locks. What happens if a machine acquires a lock located on a server, and then crashes? Is the locked resource inaccessible until the machine recovers? What if that never happens? This availability problem is addressed by replacing "locks" with "leases". A lease is just a lock that is valid for a bounded time interval. The client may periodically ask the server to extend the lease. If the client crashes, the server waits till the lease has expired, whereupon it is free to grant the lease to some other client. The decay time of a lease is a tradeoff between the communication cost of renewal requests, and non-availability after a crash. Note that leases require the client and server clock rates be similar. If not, the client may think that it holds the lease while the server thinks the lease has expired and grants it to some other client, resulting in two clients holding the lease simultaneously.

Suppose there is a network partition because of which a client is unable to renew the lease on a cached item. If it has already made changes to its local copy, it must either abort those changes, or proceed under the awareness that some other client may be changing the item simultaneously. Thus, network partition can lead to divergent copies of data. How this is handled depends on whether machines are expected to ever operate disconnected from the net.

If machines never disconnect, then one can simply add network redundancy to avoid partition. But if machines can operate disconnected, then either write permission should be limited to one partition, or one must reconcile the divergences. Currently the most common approach to reconciliation is manual intervention. New systems, such as Coda, provide some automated support for merging. The minimum support needed is simply divergence detection. Once detected, the most that can generally be done is to notify the user. However, when the divergent object has sufficiently constrained semantics (filesystem directories for example), some automatic update can be done. Note that object immutability doesn't resolve these problems, but just moves them to some other abstraction layer, such as a namespace.

Different caching strategies have different performance implications. For instance, the client might maintain a cache of information that is available at both the server and in its local disk. On a cache miss, the client can go over the net to the server, or to its local disk. While disk latency might be larger than a network round trip time to the server, going to the server consumes network bandwidth and processing time at the server. If there are a large number of clients then it might be better for each client to offload work from the server by accessing its own disk.

While most memory systems are coherent, most distributed filesystems do not provide a coherent abstraction. They permit race conditions, and all the other problems of incoherent memory. Some prototypes are moving towards an atomic semantics (coherent though non-persistent across crashes).

6 Limits of caching

The possible effectiveness of caching is limited in the presence of sharing, both practically and fundamentally.

On a uniprocessor, one might increase cache size enough to basically eliminate cache misses. A large, fully associative cache could have the working sets of all processes. The hit rate asymptotically approaches 1 with increasing cache size.

Multiprocessor distribution inherently bounds cache efficiency. Because of contention, shared data can never be perfectly cached (Figure 2).

In multi-threaded systems with very fast context switches, one can minimize the cost of a miss by swapping to a different thread. An invalidation cache consistency approach can eliminate read misses, but only by adding write misses.

7 False Sharing

There are practical limits on cache efficiency as well. If a cache is not fully associative, then there may be mapped interference. This is an example of **false sharing**, an extreme version of a phenomenon which occurs whenever blocking is used.

Cache lines (a.k.a. blocks) are one thing which cause false sharing. Consider two data items adjacent in memory. If they both end up on the same cache line, then the cache line introduces sharing.

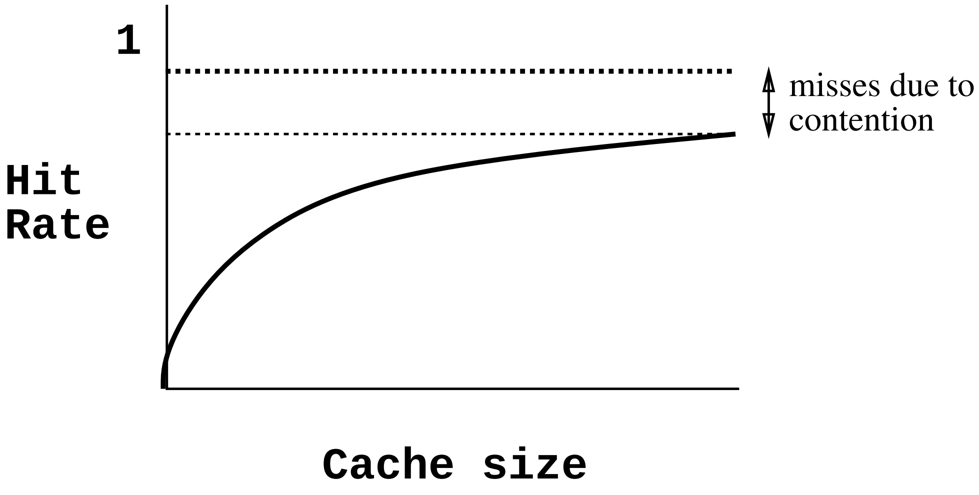

Figure 2: Contention for Shared Data Disallows Miss-less Caching.

Two processors using the two different items may thrash the cache line back and forth between themselves. This happens regardless of cache consistency mechanism (update or invalidation). This thrashing can have a big impact on performance, especially with fine grain concurrency. To avoid this problem, one tries to put different processor working sets on different lines. In this case, one could move the locks apart so that they land on different lines.

An example of the more general blocking problem occurs with cache-naive compilers. The instruction cache often gets loaded with about twice the number of instructions actually used, because of needless loading of infrequently used paths. This is addressed by attempting to move infrequently used instructions to a different block.

8 The coherence abstraction

Generally, one does not implement coherence totally in hardware. One adopts software conventions (perhaps by using a smart compiler). These bridge the gap between incoherent hardware, and a coherent memory abstraction for applications.

One convention is to use critical sections and mutexes. One acquires a mutex, creating a “barrier”, performs an action, and releases it, forming a “flash” or “fence”. The strategy of critical sections is to prevent more than one procedure from using a section simultaneously. Since generally only one or a few threads are sharing some data, the cost of this exclusion can be kept small.

If one has an abstract machine with coherent memory, does one really need to worry about locality? Unfortunately, real machine hardware is incoherent, so one does. An alternative is to build a high power network, and processors with low cost context switching. The tradeoff is the power which can be obtained for a given cost of silicon.