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

Lecture 23: Dec 7, 1992

Lecturer: Bill Weihl — *Scribe: Srikanth Kannan*

1 Introduction

In this lecture, we consider caching in multiprocessor and distributed systems. Caching, we know, helps decrease latency. In concurrent systems, caching is important because:

- It reduces the access time - Operations on a local faster memory take lesser time than those on a remote and/or slower storage.
- Caching reduces contention of shared resources - in a multiprocessor system, the contention for the shared bus and memory; and in distributed system, that for the network channel and say a file system. The reduction in contention leads to an increased throughput.

We will specify a simple implementation that clearly works and then proceed to refine it till we get one that is also practical to implement. As a first attempt at caching, consider a system in which any data is in at most one cache. This method works fine for data that need not be accessed by several clients. However, some data needs to be accessed by more than one client, eg: shared code, system files, synchronization data and application specific user data.

Some systems do not allow caching of shared data at all. Caching of shared data, however, has potential performance benefits. In these cases, coordinating caching of shared data in the caches of several clients is important.

2 Coherent Caches

Many systems implement caches with the "*strong coherence*" property. This term has been used in the literature for two different semantics:

- Linearizability — this guarantees that if, in real time, A writes X before B reads X, then the effect of A's write is seen by B. For instance, write-through caches provide linearizability.
- Sequential Consistency — operations on main memory are consistent with each client's request, but this order may not be consistent with real time ordering of the requests from different clients.

Sequential consistency is weaker than linearizability. There is currently a lot of argument about what the right spec for cache coherence is, but the net effect is to achieve "strong coherence". The method also depends on the support provided by the hardware.

3 Incoherent Memory

We now look at a spec that can be easily implemented in hardware. It resembles the simple memory of Handout 5. This, however, is *not* an implementation of simple memory — because different processors use their caches independently. In particular, data may be simultaneously dirty in more than one cache and it is unknown which of the dirty data might get lost. The `Read()` and `Write()` operations affect the local cache only; `Toc()` and `ToM()` are internal actions.

An application can make use of `Barrier()` to ensure that data written makes it to the memory. When used before a `Read()`, it ensures that data read is current except for pending `Barriers()`s of other clients.

This specification, however, is easy to implement in hardware and is amenable to performance improvements. Programming applications with this model needs more discipline.

4 Implementations of Coherent Memory

We now consider various implementations of coherent caches. The initial one clearly works, but is not practical; later refinements build one that is also realistic. Most of the material is based on work done by Butler at DEC. We would like the computations to depend on data in the local cache as much as possible. For cache coherence, however, one needs to access global information - this is the key issue to be considered.

4.1 Global Implementation

Section 3 of the handout gives an implementation that uses global state. The global check is made in the function `Clean()`, which individually checks each cache to see if data is dirty. This implementation is clearly not practical. However, the useful invariants are specified in page 3 of the handout.

`Read()` checks to see that data is current. Both `Read()` and `AF()` are based on `Clean()` and `Dirty()`. `Write()` is implemented to follow the invariants — data is only written if that will be the sole dirty copy.

4.2 Invalidation Based Implementations

We refine the above implementation to overcome having to check every cache for dirty data. In the implementations of section 4, a write to a cache proceeds only when data is not defined in any other cache. Further, data is loaded into caches directly from memory. Section 5 considers reading data from other caches.

The additional invariant maintained is that data in cache is always current. A `Read()` therefore just fetchs the data from the cache. The data is read into a cache in the first place by the internal function `ToC()`. A `Write()` proceeds only if there are no copies of the data in other caches. Thus, there is some internal action to invalidate cached copies (when data is written to memory or when explicitly requested). `Barrier()` is not interesting in this implementation because the cached data is always current.

The next implementation on pages 6–7 of handout considers the basic issues in more detail. We use locks to ensure that only one cache has data (for a given address) at a time. The lock is exclusive, and can be obtained only if no other cache has data. `Read()`s do not need to acquire locks, as in the previous implementation — data read is in the local cache. Therefore `ToC()` should check to see that the data is not dirty and that only the local client, if at all anyone, holds a lock[1]. `Release()` does not free the lock if data in the local cache is dirty.

The essential problem now is to implement the `Acquire()` function. Various approaches are possible —

- In large scale multi-processor systems and in network file systems, we can maintain a directory at the main memory site. The directory keeps track of who have copies of or locks on data. A request for a lock to the directory generates an "invalidate" request on the processors that have copies or locks.
- In bus based systems, a processor can "snoop" on the bus and wait till all others have invalidated their caches. After a broadcast of the request, the processor acquires a lock if it determines that no other client has a lock already.
- In hierarchical systems, we can maintain a directory for each subsystem. The main directory keeps track of which sets of processors have locks or copies of data. The sub-directories keep track of the particular locks or data that each processor has.

[1] An equivalent check is to verify that data is not defined in the cache.