Massachusetts Institute of Technology
Department of Electrical Engineering and Computer Science

6.826 – Principles of Computer Systems

Handout 53 December 6, 1993

Concurrent Caching

This handout presents several specifications for caches in concurrent systems. The overall goal is to give a specification for the memory we want, and then to show how to implement it using a specification that is easy to implement. To make it easier to see why the implementation works, we start with an implementation that clearly works (but is not practical), and then refine it through a series of steps to an implementation that is practical to build.

1 Coherent Memory

The first specification is for the memory that we really want, which ensures that all memory operations appear atomic. (It is essentially the same as the `SimpleMemory` specification from Handout 5.) In the literature, this specification is sometimes called a “linearizable” memory.

```
MODULE CoherentMemory [ A, D ] =     % Addresses and Data

VAR m: A -> D

APROC Read (a) -> D = << RET m(a) >>

APROC Write (a,d) = << m(a) := d >>

END CoherentMemory
```

2 Incoherent Memory

Now we give a specification that corresponds fairly directly to the hardware that is easy to build: there is a private cache for each processor, and internal actions that move data back and forth between caches and the main memory.

```
MODULE IncoherentMemory [ P, A, D ] =          % Process names, ...

TYPE
   M = A -> D
   C = P -> M

VAR
   m                                           % main memory
   c := C{* -> M{*->}}                         % local caches
   dirty: P -> A -> Bool := {*->{*->false}}    % dirty flags

% Invariants:
%    1.  Every processor has a cache defined.
%        (ALL p | c!p)
%    2.  If dirty then data is in cache.
%        (ALL p,a | dirty(p)(a) ==> c(p)!a)

% Interface procedures:

APROC Read (p,a) -> D = << c(p)!a => RET c(p) >>

APROC Write (p,a,d) = << c(p)(a) := d; dirty(p)(a) := true >>

% Barrier waits until a is not in p's cache.
APROC Barrier (p,a) = << ~(c(p)!a) => SKIP >>

% Internal actions:

% Copy memory to cache.
THREAD ToC (p,a) = << ~(dirty(p)(a)) => c(p)(a) := m(a) >>

% Copy cache to memory.
THREAD ToM (p,a) = << dirty(p)(a) => m(a) := c(p)(a); dirty(p)(a) := false >>

% Drop clean data from cache.
THREAD Drop (p,a) = << ~(dirty(p)(a)) => c(p) := c(p){a->} >>

END IncoherentMemory
```

This memory is "incoherent": different caches can have different data for the same address, so that reads and writes by different processors may see completely different data. Thus, it does not implement the `CoherentMemory` specification given earlier.

Notice that the `IncoherentMemory` has an additional operation, `Barrier`, that allows a user to tell whether data has been moved from the cache into memory. This can be used after a `Write` to ensure that an update has been written back to main memory, or before a `Read` to ensure that the data read is current.

3 Global Implementation

Now we give an implementation of `CoherentMemory`. We obtain it simply by strengthening the guards on the operations of `IncoherentMemory`. This implementation is not practical, however, because the guards involve checking global state, not just the state of a single processor.

This implementation and later ones will maintain the invariant that an address is dirty in at most one cache; this is necessary for the abstraction function to be well-defined. Later implementations will also use the same abstraction function as this one.

```
MODULE GlobalImpl[ P, A, D ] =

TYPE
   M = A -> D
   C = P -> M

VAR
   m                                            % main memory
   c := C{* -> M{*->}}                          % local caches
   dirty: P -> A -> Bool := {*->{*->false}}     % dirty flags

% Abstraction function to SimpleMemory:
%
FUNC AF () -> M =
   RET ( LAMBDA (a) -> D =
         Clean(a) => RET m(a) [*] VAR p | dirty(p)(a) => RET c(p)(a) )

% Invariants:
%    1.  Every processor has a cache defined.
%        (ALL p | c!p)
%    2.  If dirty then data is in cache.
%        (ALL p,a | dirty(p)(a) ==> c(p)!a)
%    3.  Dirty in at most one cache.
%        (ALL p,p',a | dirty(p)(a) /\ dirty(p')(a) => p=p')

% Useful internal functions (involve global state):

% p's cached value for a is current.
```

```
FUNC Current (p,a) = RET ( c(p)(a) = AF()(a) )

% clean in all caches.
FUNC Clean (a) = RET (ALL p | ~ dirty(p)(a))

% Interface procedures:

% Read only if data in cache is current.
APROC Read (p,a) -> D = << Current(p,a) => RET c(p) >>

% Write only if maintaining invariant that at most one cache is dirty.
APROC Write (p,a,d) =
   << Clean(a) \/ dirty(p)(a) => c(p)(a) := d; dirty(p)(a) := true >>

% Rest is same as IncoherentMemory.

...

END GlobalImpl
```

Notice that the guard on `Read` checks that the data in the processor's cache is current – i.e., that it equals the value currently stored in the abstract memory. This requires finding the most recent value, which is either in the main memory (if no processor has a dirty value) or in some processor's cache (if a processor has a dirty value). The guard on `Write` ensures that a given address is dirty in at most one cache. Both of these guards require checking global state.

4 Invalidation-based Implementations

We can't implement the guards of `GlobalImpl` directly. Here we show a strategy that allows a write to happen only if no other processor has a cached copy. This can be implemented by invalidating other cached copies before writing. In addition, data is loaded into a cache only from main memory. In the next section we show an alternative strategy that allows data to be loaded into a cache directly from another cache.

We give the implementation in two stages. In the first, we still use some non-local functions (`Clean` and `Only`), but we get rid of the use of `Current` by maintaining an invariant that data in a cache is always current. The parts not shown are the same as in `GlobalImpl`.

```
MODULE InvalidateImpl1[ P, A, D ] =

TYPE
   M = A -> D
   C = P -> M
```

```
VAR
   m                                           % main memory
   c := C{* -> M{*->}}                         % local caches
   dirty: P -> A -> Bool := {*->{*->false}}    % dirty flags

% Abstraction function to SimpleMemory as before.

% Invariants:
%    ...
%    4.  Data in a cache is current.
%        (ALL p,a | c(p)!a => Current(p,a))

% Useful internal functions (involve global state):

...

% appears only in p's cache.
FUNC Only (p,a) = RET (ALL p' | c(p')!a => p'=p)

% Interface procedures:

% Okay by invariant (4).
APROC Read (p,a) -> D = << c(p)!a => RET c(p) >>

% Write only if maintaining invariant that at most one cache is dirty;
% do this by waiting until no other copies.
APROC Write (p,a,d) =
   << Only(p,a) => c(p)(a) := d; dirty(p)(a) := true >>

...

% Internal actions:

% Copy memory to cache; guard is needed to maintain invariant (4).
THREAD ToC (p,a) = << Clean(a) => c(p)(a) := m(a) >>

...

END InvalidateImpl1
```

Notice that `Barrier` is not very interesting for this implementation, because a cached copy is always current.

The next implementation refines the previous one to limit the global tests to an action `Acquire` that acquires a lock, and to the function `Free`. The lock is exclusive, but is only required for `Write` operations; thus, the implementation uses "read/write" locks, by allowing multiple readers to read in parallel, but only one writer to write (and only if no readers are reading). We show the changes from `InvalidateImpl1`.

```
MODULE InvalidateImpl2[ P, A, D ] =

TYPE
   M = A -> D
   C = P -> M

VAR
   m                                           % main memory
   c := C{* -> M{*->}}                         % local caches
   dirty: P -> A -> Bool := {*->{*->false}}    % dirty flags
   lock: P -> A -> Bool := {*->{*->false}}     % Does p have address a locked?

% Abstraction function to SimpleMemory as before.

% Invariants:
%    ...
%    5.  Dirty data is locked.
%        (ALL p,a | dirty(p)(a) => lock(p)(a))
%    6.  Locked data is only copy.
%        (ALL p,a | lock(p)(a) => Only(p,a))
%    7.  Lock is exclusive.
%        (ALL p,p',a | lock(p)(a) /\ lock(p')(a) => p=p')

% Useful internal functions (involve global state):

% a is not locked by any processor.
FUNC Free (a) -> Bool = RET (ALL p | ~lock(p)(a))

...

% Interface procedures:

% Write only if have a lock (and thus the only copy).
APROC Write (p,a,d) =
   << lock(p,a) => c(p)(a) := d; dirty(p)(a) := true >>

...
```

```
% Internal actions:

% Copy memory to cache; guard ensures invariant (4).
THREAD ToC (p,a) = << ~(dirty(p)(a)) /\
                        ( lock(p)(a) \/ Free(a) )
                    => c(p)(a) := m(a) >>

% Acquire lock only if not locked by any other processor and already
% have only copy of desired data.
THREAD Acquire (p,a) = << Free(a) /\ Only(p,a) => lock(p)(a) := true >>

% Don't release while still dirty.
THREAD Release (p,a) = << ~dirty(p)(a) => lock(p)(a) := false >>

...

END InvalidateImpl2
```

Note that the guard of `ToC` has also changed: data can be loaded only if it is not overwriting a dirty copy, and if no other cache has a dirty copy (which is true if this processor has a lock or no processor has a lock).

The remaining problem is to implement the guard of `Acquire`. Many approaches are possible; here are a few:

- In large-scale multiprocessors (and in network file systems), a "directory" is maintained (typically at the site keeping the main memory copy of the data) that keeps track of the set of processors that either have copies of the data or have it locked. To acquire a lock, a processor sends a request to the directory, which then asks each processor currently holding a copy or the lock to relinquish it.
- In bus-based systems, each processor's cache can "snoop" on the bus. Thus, a processor can broadcast on the bus to ask all other processors to relinquish their locks/copies, and can "or" the responses to determine whether they have all been relinquished.
- Finally, some systems use a hierarchical structure. We partition the processors into sets, and maintain a directory for each set. The main directory attached to main memory keeps track of which processor sets have copies or locks; the directory for each set keeps track of which processors in the set have copies or locks. (Or the hierarchy may have more levels, with the processor sets further subdivided.)

5 Update-based implementations

Here we show an implementation that allows data to be loaded into a cache directly from another cache. Again, we present it in two stages, corresponding to `InvalidateImpl1` and `InvalidateImpl2`.

Exercise: derive a different implementation that, in addition to obtaining data from another cache, allows writes when there are multiple copies by updating the other copies directly (rather than invalidating them).

```
% The only change from InvalidateImpl1 is the addition
% of the clause in ToC that permits direct cache-to-cache
% transfers.
MODULE C2CImpl1[ P, A, D ] =

TYPE
   M = A -> D
   C = P -> M

VAR
   m                                           % main memory
   c := C{* -> M{*->}}                         % local caches
   dirty: P -> A -> Bool := {*->{*->false}}    % dirty flags

% Abstraction function to SimpleMemory as before:

% Invariants:
%    1.  Every processor has a cache defined.
%        (ALL p | c!p)
%    2.  If dirty is true then data is in cache.
%        (ALL p,a | dirty(p)(a) ==> c(p)!a)
%    3.  Dirty in at most one cache.
%        (ALL p,p',a | dirty(p)(a) /\ dirty(p')(a) => p=p')
%    4.  Data in a cache is current.
%        (ALL p,a | c(p)!a => Current(p,a))

...

% Internal actions:

% Copy memory to cache; guard is needed to maintain invariant (4).
THREAD ToC (p,a) = << Clean(a) => c(p)(a) := m(a)
                    [] VAR p' | c(p')!a => c(p)(a) := c(p')(a) >>

...

END C2CImpl1
```

Note that the second clause in `ToC` allows data to be taken from any cache; since data in a cache is always current, this is always okay.

Again, we refine the previous implementation to limit the global test to acquiring the lock and checking that it is free. We show the changes from `C2CImpl1`, and also from `InvalidateImpl2`.

```
MODULE C2CImpl2[ P, A, D ] =

TYPE
   M = A -> D
   C = P -> M

VAR
   m                                          % main memory
   c := C{* -> M{*->}}                        % local caches
   dirty: P -> A -> Bool := {*->{*->false}}   % dirty flags
   lock: P -> A -> Bool := {*->{*->false}}    % Does p have address a locked?

% Abstraction function to SimpleMemory as before.

% Invariants:
%    ...
%    5.  Dirty data is not necessarily locked any more; other caches
%        can have copies.
%    6.  Locked data is only copy.
%        (ALL p,a | lock(p)(a) => Only(p,a))
%    7.  Lock is exclusive.
%        (ALL p,p',a | lock(p)(a) /\ lock(p')(a) => p=p')

...

% Interface procedures:

% Write only if have a lock (and thus the only copy).
APROC Write (p,a,d) =
   << lock(p,a) => c(p)(a) := d; dirty(p)(a) := true >>

...

% Internal actions:

% Copy memory to cache; guard ensures invariant (4).
THREAD ToC (p,a) = << Clean(a) /\
                        ( lock(p)(a) \/ Free(a) )
                    => c(p)(a) := m(a)
```

```
              [] VAR p' | c(p')!a /\ Free(a) => c(p)(a) := c(p')(a) >>

...

% Acquire lock only if not locked by any other processor and already
% have only copy of desired data.
THREAD Acquire (p,a) = << Free(a) /\ Only(p,a) => lock(p)(a) := true >>

% Can release at any time, even if dirty.
THREAD Release (p,a) = << lock(p)(a) := false >>

END C2CImpl2
```

The change to the guard of `ToC` is needed to ensure that the lock is dropped before creating other cached copies. Otherwise, a processor could write when another still had an old value cached. The global `Clean` in `ToC` is exactly what snoopy implementations do to prevent stale data from being loaded from memory.

6 Mutex-based Implementation

Now we consider a different implementation that uses exclusive locking for both reads and writes. There are several interesting differences with `InvalidateImpl2`, which allows multiple copies as long as none are dirty:

- `Read` has a stronger guard (requiring a lock).
- `ToC` has a weaker guard that simply prevents losing writes that haven't been written back yet.
- `Acquire` has a weaker guard without `Only`. (In fact, we don't use `Only` in the implementation at all.)
- `Acquire` has an additional `Barrier` action.
- The invariant about when data in a cache is current is weaker. For `InvalidateImpl2`, the invariant says that `(ALL p,a | c(p)!a ==> Current(p,a))`; for `ExclImpl`, the invariant is `(ALL p,a | c(p)!a /\ lock(p)(a) ==> Current(p,a))`. In other words, data in a cache is current only if it is locked.

The nice thing about this implementation is that the internal actions are the same as in `IncoherentMemory`. Thus, we could use an unmodified `IncoherentMemory` as the basis for `ExclImpl`; all that is needed is to ensure that the lock is held while reads and writes are done. The standard way of doing this is to confine all reads and writes to critical sections within which the lock is held, with `Acquire` and `Release` done at the start and end of the critical section in the usual way. So `ExclImpl` shows how to build `CoherentMemory` out of `IncoherentMemory` by adding locks and using locks and barriers at appropriate times.

We show the differences from `GlobalImpl`.

```
MODULE ExclImpl[ P, A, D ] =

TYPE
   M = A -> D
   C = P -> M

VAR
   m                                           % main memory
   c := C{* -> M{*->}}                         % local caches
   dirty: P -> A -> Bool := {*->{*->false}}    % dirty flags
   lock: P -> A -> Bool := {*->{*->false}}     % locks

% Abstraction function to SimpleMemory as before.

% Invariants:
%    ...
%    4.  Locked data in a cache is current.
%        (ALL p,a | c(p)!a /\ lock(p)(a) ==> Current(p,a))
%    5.  Dirty data is locked.
%        (ALL p,a | dirty(p)(a) => lock(p)(a))
%    6.  Locked data is not necessarily the only copy.
%    7.  Lock is exclusive.
%        (ALL p,p',a | lock(p)(a) /\ lock(p')(a) => p=p')

% Useful internal functions (involve global state):

% a is not locked by any processor.
FUNC Free (a) -> Bool = RET (ALL p | ~lock(p)(a))

...

% Interface procedures:

% Read only if locked; okay by invariant (4).
APROC Read (p,a) -> D = << c(p)!a /\ lock(p)(a) => RET c(p) >>

% Write only if locked.
APROC Write (p,a,d) =
   << lock(p)(a) => c(p)(a) := d; dirty(p)(a) := true >>
```

```
...

% Lock actions:

% Acquire lock only if not locked by any other processor.
THREAD Acquire (p,a) = << Free(a) => lock(p)(a) := true ; Barrier(a) >>

% Don't release while still dirty.
THREAD Release (p,a) = ~dirty(p)(a) => lock(p)(a) := false

END ExclImpl
```

The `Barrier` at the end of `Acquire` ensures that any old data in the cache is dropped after the lock is obtained; hence, subsequent accesses must first get a current copy from main memory. In some systems, `Release` also does `Barrier`, but this isn't necessary: as long as the dirty data has been written back, the barrier at the end of `Acquire` will guarantee that locked data in a cache is current.