30. Concurrent Caching

In the previous handout we studied the fault-tolerance aspects of replication. In this handout we study many of the performance and concurrency aspects, under the label ‘caching’. A cache is of course a form of replication. It is usually a copy of some ‘ground truth’ that is maintained in some other way, although ‘all-cache’ systems are also possible. Normally a cache is not big enough to hold the entire state (or it’s not cost-effective to pay for the bandwidth needed to get all the state into it), so one issue is how much state to keep in the cache. The other main issue is how to keep the cache up-to-date, so that a read will obtain the result of the most recent write as the `Memory` spec requires. We concentrate on this problem.

This handout presents several specs and codes for caches in concurrent systems. We begin with a spec for `CoherentMemory`, the kind of memory we would really like to have; it is just a function from addresses to data values. We also specify the `IncoherentMemory` that has fast code, but is not very nice to use. Then we show how to change `IncoherentMemory` so that it codes `CoherentMemory` with as little communication as possible. We describe various strategies, including invalidation-based and update-based strategies, and strategies using incoherent memory plus locking.

Since the various strategies used in practice have a lot in common, we unify the presentation using successive refinements. We start with cache code `GlobalImpl` that clearly works, but is not practical to code directly because it is extremely non-local. Then we refine `GlobalImpl` in stages to obtain (abstract versions of) practical code.

First we show how to use reader/writer locks to get a practical version of `GlobalImpl` called a coherent cache. We do this in two stages, an ideal cache `CurrentCaches` and a concrete cache `ExclusiveLocks`. The caches change the guards on internal actions of `IncoherentMemory` as well as on the external read and write actions, so they can’t be coded externally, simply by adding a test before each read or write of `IncoherentMemory`, but require changes to its insides.

There is another way to use locks to get a different practical version of `GlobalImpl`, called `ExternalLocks`. The advantage of `ExternalLocks` is that the locking is decoupled from the internal actions of the memory system so that it can be coded separately, and hence `ExternalLocks` can run entirely in software on top of a memory system that only implements `IncoherentMemory`. In other words, `ExternalLocks` is a practical way to program coherent memory on a machine whose hardware provides only incoherent memory.

There are many practical codes for the methods that are described abstractly here. Most of them originated in the hardware of shared-memory multiprocessors.[1] It is also possible to code shared memory in software, relying on some combination of page faults from the virtual memory and checks supplied by the compiler. This is called ‘distributed shared memory’ or DSM.[2] Intermediate schemes do some of the work in hardware and some in software.[3] Many of the techniques have been re-invented for coherent distributed file systems.[4]

All our code makes use of a global memory that is modeled as a function from addresses to data values; in other words, the spec for the global memory is simply `CoherentMemory`. This means that actual code may have a recursive structure, in which the top-level code for `CoherentMemory` using one of our algorithms contains a global memory that is coded with another algorithm and contains another global memory, etc. This recursion terminates only when we lose interest in another level of virtualization. For example,

a processor’s memory may consist of a first level cache plus
a global memory made up of a second level cache plus
a global memory made up of a main memory plus
a global memory made up of a local swapping disk plus
a global memory made up of a file server ....

Specs

First we recall the spec for ordinary coherent memory. Then we give the spec for efficient but ugly incoherent memory. Finally, we discuss an alternative, less intuitive way of writing these specs.

Coherent memory

The first spec is for the memory that we really want, which ensures that all memory operations appear atomic. It is essentially the same as the `Memory` spec from Handout 5 on memory specs, except that `m` is defined to be total. In the literature, this is sometimes called a ‘linearizable’ memory; in the more general setting of transactions it is ‘serializable’ (see handout 20).

```
MODULE CoherentMemory [P, A, V] EXPORT Read, Write =
% Arguments are Processors, Addresses and Data

TYPE M          =  A -> D SUCHTHAT (\ f: A->D | (ALL a | f!a))
VAR  m

APROC Read(p, a) -> D = << RET m(a) >>
APROC Write(p, a, d) = << m(a) := d >>

END CoherentMemory
```

[1] J. Hennessy and D. Patterson, *Computer Architecture: A Quantitative Approach*, 2nd ed., Morgan Kaufmann, 1996, chapter 8, pp 635-754.

[2] K. Li and P. Hudak, Memory coherence in shared virtual memory systems. *ACM Transactions on Computer Systems* **7**, 4 (Nov. 1989), pp 321-359. For recent work in this active field see any ISCA, ASPLOS, OSDI, or SOSP proceedings.

[3] David Chaiken and Anant Agarwal. Software-extended coherent shared memory: performance and cost. *Proceedings of the 21st Annual Symposium on Computer Architecture*, pages 314-324, April 1994 (http://www.cag.lcs.mit.edu/alewife/papers/soft-ext-isca94.html). Jeffrey Kuskin et al., The Stanford FLASH multiprocessor. In *Proceedings of the 21st International Symposium on Computer Architecture*, pages 302-313, Chicago, IL, April 1994 (http://www-flash.stanford.edu/architecture/papers/ISCA94).

[4] M. Nelson et al., Caching in the Sprite network file system. *ACM Transactions on Computer Systems* **11**, 2 (Feb. 1993), pp 228-239. For recent work in this active field see any OSDI or SOSP proceedings.

From this point we drop the `a` argument and study a memory with just one location; that is, we study a cached register. Since everything about the specs and code holds independently for each address, we don't lose anything by doing this, and it reduces clutter. We also write the `p` argument as a subscript, again to make the specs easier to read. The previous spec becomes

```
MODULE CoherentMemory [P, V] EXPORT Read, Write =
% Arguments are Processors and Data

TYPE M          =  D                                % Memory
VAR  m

APROC Read_p -> D = << RET m >>
APROC Write_p(d) = << m := d >>

END CoherentMemory
```

Of course, code usually has limits on the size of a cache, or other resource limitations that can only be expressed by considering all the addresses at once, but we will not study this kind of detail here.

Incoherent memory

The next spec describes the minimum guarantees made by hardware: there is a private cache for each processor, and internal actions that move data back and forth between caches and the main memory, and between different caches. The only guarantee is that data written to a cache is not overwritten in that cache by anyone else's data. However, there is no ordering on writes from the cache to main memory.

This is not enough to get any useful work done, since it allows writes to remain invisible to others forever. We therefore add a `Barrier` synchronization operation that forces the cache and memory to agree. This can be used after a `Write` to ensure that an update has been written back to main memory, and before a `Read` to ensure that the data being read is current. `Barrier` was called `Sync` when we studied disks and file systems in handout 7, and eventual consistency in handouts 12 and 28.

Note that `Read` has a guard `Live` that it makes no attempt to satisfy (hardware usually has an explicit flag called `valid`). Instead, there is another action `MtoC` that makes `Live` true. In a real system an attempt to do a `Read` will trigger a `MtoC` so that the `Read` can go ahead, but in Spec we can omit the direct linkage between the two actions and let the non-determinism do the work. We use this coding trick repeatedly in this handout. Another example is `Barrier`, which forces the cache to drop its data by waiting until `Drop` happens; if the cache is dirty, `Drop` will wait for `CtoM` to store its data into memory first.

You might think that this is just specsmanship and that a nondeterministic `MtoC` is silly, but in fact transferring data from `m` to `c` without a `Read` is called prefetching, and many codes do it under various conditions: because it's in the next block, or because a past reference sequence used it, or because the program executes a prefetch instruction. Saying that it can happen nondeterministically captures all of this behavior very simply.

We adopt the convention that an invalid cache entry has the value `nil`.

```
MODULE IncoherentMemory [P, A, V] EXPORT Read, Write, Barrier =

TYPE M          =  D                                          % Memory
     C          =  P -> (D + Null)                            % Cache

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

% INVARIANT Inv1: (ALL p | c!p)                               % each processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> Live_p)                % dirty data is in the cache

APROC Read_p -> D = << Live_p => RET c_p >>                   % MtoC gets data into cache
APROC Write_p(d)  = << c_p := d; dirty_p := true >>

APROC Barrier_p = << ~ Live_p => SKIP >>                      % wait until not in cache

FUNC Live_p -> Bool = RET (c_p # nil)

% Internal actions

THREAD Internal_p = DO MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p [] SKIP OD

APROC MtoC_p = << ~ dirty_p => c_p := m >>                    % copy memory to cache
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >>    % copy cache to memory
APROC CtoC_p,p' = << ~ dirty_p' /\ Live_p => c_p' := c_p >>   % copy from cache p to p'
APROC Drop_p = << ~ dirty_p => c_p := nil >>                  % drop clean data from cache

END IncoherentMemory
```

In real code some of these actions may be combined. For example, if the cache is dirty, a real barrier operation may do `CtoM; Barrier; MtoC` by just storing the data. These combinations don't introduce any new behavior, however, and it's simplest to study the minimum set of actions presented here.

This memory is 'incoherent': different caches can have different data for the same address, so that adjacent reads by different processors may see completely different data. Thus, it does not implement the `CoherentMemory` spec given earlier. However, after a `Barrier`$_p$, `c`$_p$ is guaranteed to agree with `m` until the next time `m` changes or `p` does a `Write`.[5] There are commercial machines whose memory systems have essentially this spec.[6] Others have explored similar specs.[7]

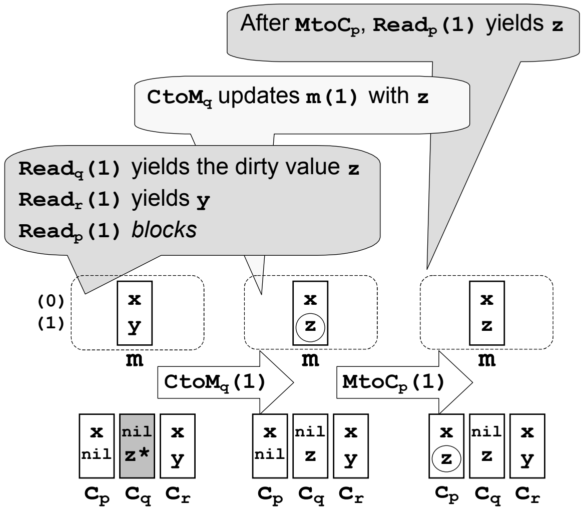
Here is a simple example that shows the contents of two addresses `0` and `1` in `m` and in three processors `p`, `q`, and `r`. A dirty value is marked with a `*`, and circles mark values that have

[5] An alternative version of `Barrier` has the guard `~ live`$_p$ `\/ (c`$_p$ `= m)`; this is equivalent to the current `Barrier`$_p$ followed by an optional `MtoC`$_p$. You might think that it's better because it avoids a copy from `m` to `c`$_p$ in case they already agree. But this is a spec, not an implementation, and the change doesn't affect its external behavior.

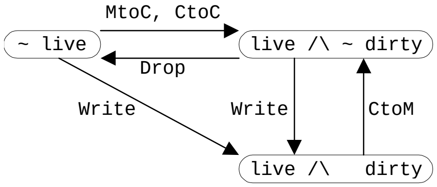
[6] Digital Equipment Corporation, *Alpha Architecture Handbook*, 1992. IBM, *The PowerPC Architecture*, Morgan Kaufmann, 1994.

[7] Gharachorloo, K., et al., Memory consistency and event ordering in scalable shared-memory multiprocessors, *Proc. 17th Symposium on Computer Architecture*, 1990, pp 15-26. Gibbons, P. and Merritt, M., Specifying nonblocking shared memories*, Proc. 4th ACM Symposium on Parallel Algorithms and Architectures*, 1992, pp 158-168.

changed. Initially Read$_q$(1) yields the dirty value z, Read$_r$(1) yields y, and Read$_p$(1) blocks because c$_p$(1) is nil. After the CtoM$_q$ the global location m(1) has been updated with z. After the MtoC$_p$, Read$_p$(1) yields z. One way to ensure that the CtoM$_q$ and MtoC$_p$ actions happen before the Read$_p$(1) is to do Barrier$_q$ followed by Barrier$_p$ between the Write$_q$(1) that makes z dirty in c$_q$ and the Read$_p$(1).



Here are the possible transitions of IncoherentMemory for a given address. This kind of state transition picture is the standard way to describe cache algorithms in the literature; see pages 664-5 of Hennessy and Patterson, for example.



This is the weakest shared-memory spec that seems likely to be useful in practice. But perhaps it is too weak. Why do we introduce this messy incoherent memory? Wouldn't we be much better off with the simple and familiar coherent memory? There are two reasons to prefer IncoherentMemory:

- Code for IncoherentMemory can run faster—there is more locality and less communication. As we will see later in ExternalLocks, software can batch the communication that is needed to make a coherent memory out of IncoherentMemory.
- Even CoherentMemory is tricky to use when there are concurrent clients. Experience has shown that it's necessary to have wizards to package it so that ordinary programmers can use it safely. This packaging takes the form of rules for writing concurrent programs and procedures that encapsulate references to shared memory. We studied these rules in handout 14 on practical concurrency, under the name 'easy concurrency'. The two most common examples are:

  Mutual exclusion / critical sections / monitors together with a "lock before touching' rule, which ensure that a number of references to shared memory can be done without interference from other processors, just as in a sequential program. Reader/writer locks are an important variation.

  Producer-consumer buffers.

For the ordinary programmer only the simplicity of the package is important, not the subtlety of its code. We need a smarter wizard to package IncoherentMemory, but the result is as simple to use as the packaged CoherentMemory.

Specifying legal histories directly

It's common in the literature to write the specs CoherentMemory and IncoherentMemory explicitly in terms of legal sequences of references in each processor, rather than as state machines (see the references in the previous section). We digress briefly to explain this approach informally; it is similar to what we did to specify concurrent transactions in handout 20.

For CoherentMemory^{LH}, there must be a total ordering of *all* the Read$_p$ and Write$_p$(v) actions done by the processors (for all the addresses) that

- respects the order at each p, and
- such that for each Read and closest preceding Write(v), the Read returns v.

For IncoherentMemory^{LH}, *for each address separately* there must be a total ordering of the Read$_p$, Write$_p$, and Barrier$_p$ actions done by the processors that has the same properties. IncoherentMemory is weaker than CoherentMemory because it allows references to different addresses to be ordered differently. If there were only one address and no other communication (so that you couldn't see the relative ordering of the operations), you couldn't tell the difference between the two specs. A real barrier operation usually does a Barrier for every address, and thus forces all the references before it at a given processor to precede all the references after it.

It's not hard to show that CoherentMemory^{LH} is equivalent to CoherentMemory. It's less obvious that IncoherentMemory^{LH} is almost equivalent to IncoherentMemory. There's more to this spec than meets the eye, because it doesn't say anything about how the chosen ordering is related to the real times at which different processors do their operations. Actually it is somewhat more permissive than IncoherentMemory. For example, it allows the following history

- Initially x=1, y=1.

- Processor p reads 4 from x, then writes 8 to y.
- Processor q reads 8 from y, then writes 4 to x.

For x we have the ordering $Write_q(4);\ Read_p$, and for y the ordering $Write_p(8);\ Read_q$.

We can rule out this kind of predicting the future by observing that the processors make their references in some total order in real time, and requiring that a suitable ordering exist for the references in each prefix of this real time order. With this restriction, the two versions of $IncoherentMemory^{LH}$ and IncoherentMemory are equivalent. But the restriction may not be an improvement, since it's conceivable that a processor might be able to predict the future in this way by speculative execution. In any case, the memory spec for the Alpha is in fact $IncoherentMemory^{LH}$ and allows this freedom.

Coding coherent memory

We give a sequence of refinements that implement CoherentMemory and are successively more practical: GlobalImpl, Current Caches, and ExclusiveLocks. Then we give a different kind of code that is based on IncoherentMemory.

Global code

Now we give code for CoherentMemory. We obtain it simply by strengthening the guards on the operations of IncoherentMemory (omitting Barrier, which we don't need). This code is not practical, however, because the guards involve checking global state, not just the state of a single processor. This module, like later ones, maintains the invariant Inv3 that an address is dirty in at most one cache; this is necessary for the abstraction function to make sense. Note that the definition of Current says that the cache agrees with the abstract memory.

We show only the code that differs from IncoherentMemory, boxing the new parts.

```
MODULE GlobalImpl [P, A, V] EXPORT Read, Write =    % implements CoherentMemory

TYPE ...                                            % as in IncoherentMemory
VAR  ...

% ABSTRACTION: CoherentMemory.m = (Clean() => m [*] {p | dirty_p | c_p}.choose)

% INVARIANT Inv3: {p | dirty_p}.size <= 1           % dirty in at most one cache

APROC Read_p -> D = << Current_p  => RET c_p >>     % read only current data
APROC Write_p(d) =                                  % Write maintains Inv3
    << Clean() \/ dirty_p  => c_p := d; dirty_p := true >>

FUNC Current_p =                                    % p's cache is current?
    RET c_p = (Clean() => m [*] {p | dirty_p | c_p}.choose)
FUNC Clean() = RET (ALL p | ~ dirty_p)              % all caches are clean?
```

% Same internal actions as IncoherentMemory.

```
END GlobalImpl
```

Notice that the guard on Read checks that the data in the processor's cache is current, that is, 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. These guards make it obvious that GlobalImpl implements CoherentMemory, but both require checking global state, so they are impractical to code directly.

Code in which caches are always current

We can't code the guards of GlobalImpl directly. In this section, we refine GlobalImpl a bit, replacing some (but not all) of the global tests. We carry this refinement further in the following sections. Our strategy for correctness is to always strengthen the guards in the actions, without changing the rest of the code. This makes it obvious that we simulate the previous module and that existing invariants hold. The only thing to check is that new invariants hold.

The main idea of CurrentCaches is to always keep the data in the caches current, so that we no longer need the Current guard on Read. In order to achieve this, we impose a guard on a write that allows it to happen only if no other processor has a cached copy. This is usually coded by having a write invalidate other cached copies before writing; in our code Write waits for Drop actions at all the other caches that are live. Note that Only implies the guard of GlobalImpl.Write because of Inv2 and Inv3, and Live implies the guard of GlobalImpl.Read because of Inv4. This makes it obvious that CurrentCaches implements GlobalImpl. CurrentCaches uses the non-local functions Clean and Only, but it eliminates Current. This is progress, because Read, the most common action, now has a local guard, and because Clean and Only just test Live and dirty, which is much simpler than Current's comparison of c_p with m.

As usual, the parts not shown are the same as in the last module, GlobalImpl.

```
MODULE CurrentCaches ... =                          % implements GlobalImpl

TYPE ...                                            % as in IncoherentMemory
VAR  ...

% ABSTRACTION to GlobalImpl: Identity on m, c, and dirty.

% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)    % data in caches is current

...

FUNC Only_p -> Bool = RET {p' | Live_p'} <= {p}     % appears at most in p's cache

APROC Read_p -> D = << Live_p  => RET c_p >>        % read locally; OK by Inv4
APROC Write_p(d) =                                  % write locally the only copy
    << Only_p  => c_p := d; dirty_p := true >>

...

APROC MtoC_p = << Clean()  => c_p := m >>           guard maintains Inv4
...

END CurrentCaches
```

Code using exclusive locks

The next code refines `CurrentCaches` by introducing an exclusive (write) lock with a `Free` test and `Acquire` and `Release` actions. A writer must hold the lock on an object while it writes, but a reader need not hold any lock (`Live` acts as a read lock according to `Inv4` and `Inc6`). Thus, multiple readers can read in parallel, but only one writer can write at a time, and only if there are no concurrent readers. This means that before a write can happen at `p`, all other processors must drop their copies; making this happen is called 'invalidation'. The code ensures that while a processor holds a lock, no other cache has a copy of the locked object. It uses the non-local functions `Clean` and `Free`, but everything else is local. Again, the guards are stronger than those in `CurrentCaches`, so it's obvious that `ExclusiveLocks0` implements `CurrentCaches`. We show the changes from `CurrentCaches`.

```
MODULE ExclusiveLocks0 ... =                          % implements CurrentCaches

TYPE ...                                              % as in IncoherentMemory
VAR  ...
     lock        :  P -> Bool := {*->false}           % p has lock on cache?

% ABSTRACTION to CurrentCaches: Identity on m, c, and dirty.

% INVARIANT Inv5: {p | lock_p}.size <= 1              % lock is exclusive
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)         % locked data is only copy

...

APROC Write_p(d) =                                    % write with exclusive lock
     << lock_p => c_p := d; dirty_p := true >>
...

FUNC Free() -> Bool = RET (ALL p | ~ lock_p)          % no one has cache locked?

THREAD Internal_p =
     DO    MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
        [] Acquire_p [] Release_p  [] SKIP OD

APROC MtoC_p =                                        % guard maintains Inv4, Inv6
     << Clean() /\  (lock_p \/ Free()) => c_p := m >>
APROC CtoC_p,p' =                                     % guard maintains Inv6
     << Free()  /\ ~ dirty_p' /\ Live_p => c_p' := c_p >>

APROC Acquire_p = << Free() /\ Only_p => lock_p :=true >> % exclusive lock is on cache
APROC Release_p = << lock_p := false >>               % release at any time

...

END ExclusiveLocks0
```

Note that this all works even in the presence of cache-to-cache copying of dirty data; a cache can be dirty without being locked. A strategy that allows such copying is called *update-based*. The usual code broadcasts (on the bus) every write to a shared location. That is, it combines with each `Write`$_p$ a `CtoC`$_{p,\ p'}$ for each live `p'`. If this is done atomically, we don't need the `Only`$_p$ in `Acquire`$_p$ This is good if for each write of a shared location, the average number of reads on a

different processor is near 1. It's bad if this average is much less than 1, since then each read that goes faster is paid for with many bus cycles wasted on updates.

It's possible to combine updates and invalidation. They you have to decide when to update and when to invalidate. It's possible to make this choice in a way that's within a factor of two of an optimal algorithm that knows the future pattern of references.[8] The rule is to keep updating until the accumulated cost of updates equals the cost of a read miss, and then invalidate.

Both `Read` and `Write` now do only local tests, which is good since they are supposed to be the most common actions. The remaining global tests are the `Only` test in `Acquire`, the `Clean` test in `MtoC`, and the `Free` tests in `Acquire`, `MtoC`, and `CtoC`. In hardware these are most commonly coded by snooping on a bus. A processor can broadcast on the bus to check that:

- No one else has a copy (`Only`).
- No one has a dirty copy (`Clean`).
- No one has a lock (`Free`).

It's called 'snooping' because these operations always go along with transfers between cache and memory (except for `Acquire`), so no extra bus cycles are need to give every processor on the bus a chance to see them.

For this to work, another processor that sees the test must either abandon its copy or lock, or signal `false`. The `false` signals are usually generated at exactly the same time by all the processors and combined by a simple 'or' operation. The processor can also request that the others relinquish their locks or copies; this is called 'invalidating'. Relinquishing a dirty copy means first writing it back to memory, whereas relinquishing a non-dirty copy means just dropping it from the cache. Sometimes the same broadcast is used to invalidate the old copies and update the caches with new copies, although our code breaks this down into separate `Drop`, `Write`, and `CtoC` actions.

Keeping dirty data locked

In the next module, we eliminate the cache-to-cache copying of dirty data; that is, we eliminate updates on writes of shared locations. We modify `ExclusiveLocks` so that locks are held longer, until data is no longer dirty. Besides the delayed lock release, the only significant change is in the guard of `MtoC`. Now data can only be loaded into a cache `p` if it is not dirty in `p` and is not locked elsewhere; together, these facts imply that the data item is clean, so we no longer need the global `Clean` test.

[8] A. Karlin et al, Competitive snoopy caching. *Algorithmica* **3**, 1 (1988), pp 79-119.

```
MODULE ExclusiveLocks ... =                        % implements ExclusiveLocks0

TYPE ...                                           % as in ExclusiveLocks0
VAR  ...

% ABSTRACTION to ExclusiveLocks0: Identity on m, c, dirty, and lock.

% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)     % dirty data is locked

...

APROC MtoC_p =                                     % guard implies Clean()
    << ~ dirty_p /\ (lock_p \/ Free()) => c_p := m >>
APROC Release_p = << ~ dirty_p => lock_p := false >>   % don't release if dirty
...
END ExclusiveLocks
```

For completeness, we give all the code for `ExclusiveLocks`, since there have been so many incremental changes. The non-local operations are boxed.

```
MODULE ExclusiveLocks[P,A,V] EXPORT Read,Write =   % implements CoherentMemory

TYPE M = D                                         % Memory
     C = P -> (D + Null)                           % Cache

VAR  m          :  CoherentMemory.M                % main memory
     c          := C{* -> nil}                     % local caches
     dirty      :  P -> Bool := {*->false}         % dirty flags
     lock       :  P -> Bool := {*->false}         % p has lock on cache?

% ABSTRACTION to ExclusiveLocks: Identity on m, c, dirty, and lock.

% INVARIANT Inv1: (ALL p | c!p)                    % every processor has a cache
% INVARIANT Inv2: (ALL p | dirty_p ==> Live_p)     % dirty data is in the cache

% INVARIANT Inv3: {p | dirty_p}.size <= 1          % dirty in at most one cache
% INVARIANT Inv4: (ALL p | Live_p ==> Current_p)   % data in caches is current
% INVARIANT Inv5: {p | lock_p}.size <= 1           % lock is exclusive
% INVARIANT Inv6: (ALL p | lock_p ==> Only_p)      % locked data is only copy
% INVARIANT Inv7: (ALL p | dirty_p ==> lock_p)     % dirty data is locked

APROC Read_p -> D = << Live_p => RET c_p >>        % read locally; OK by Inv4
APROC Write_p(d) =                                 % write with exclusive lock
    << lock_p => c_p := d; dirty_p := true >>

FUNC Live_p -> Bool = RET (c_p # nil)
FUNC Only_p -> Bool = RET {p' | Live_p'} <= {p}    % appears at most in p's cache?
FUNC Free() -> Bool = RET (ALL p | ~ lock_p)       % no one has cache locked?

THREAD Internal_p =
    DO   MtoC_p [] CtoM_p [] VAR p' | CtoC_p,p' [] Drop_p
       [] Acquire_p [] Release_p [] SKIP OD

APROC MtoC_p =                                     % guard implies Clean()
    << ~ dirty_p /\ (lock_p \/ Free()) => c_p := m >>
```

```
APROC CtoM_p = << dirty_p => m := c_p; dirty_p := false >>  % copy cache to memory.
APROC CtoC_p,p' =                                           % guard maintains Inv6
    << Free() /\ ~ dirty_p' /\ Live_p => c_p' := c_p >>
APROC Drop_p = << ~ dirty_p => c_p := nil >>                % drop clean data from cache

APROC Acquire_p = << Free() /\ Only_p => lock_p :=true >>   % exclusive lock is on cache
APROC Release_p = << ~ dirty_p => lock_p := false >>        % don't release if dirty

END ExclusiveLocks
```

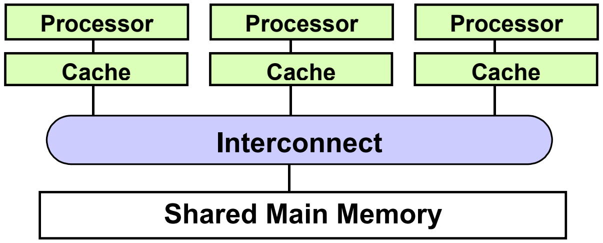
Practical code

The remaining global tests are the `Only` test in the guard of `Acquire`, and the `Free` tests in the guards of `Acquire`, `MtoC` and `CtoC`. There are many ways to code them. Here are a few:

- Snooping on the bus, as described above. This is only practical when you have a cheap synchronous broadcast, that is, in a bus-based shared memory multiprocessor. The shared bus limits the maximum performance, so typically such systems are not built with more than about 8 processors. As processors get faster, a shared bus gets less practical.

- Directory-based: Keep a "directory", usually associated with main memory, containing information about where locks and copies are currently located. To check `Free`, a processor need only interact with the directory. To check `Only`, the same strategy can be used; however, there is a difficulty if cache-to-cache copying is permitted—the directory must be informed when such copying occurs. For this reason, directory-based code usually eliminates cache-to-cache copying entirely. So far, there's no need for broadcast. To acquire a lock, the directory may need to communicate with other caches to get them to relinquish locks and copies. This can be done by broadcast, but usually the directory keeps track of all the live processors and sends a message to each one. If there are lots of processors, it may fall back to broadcast for locations that are shared by too many processors.

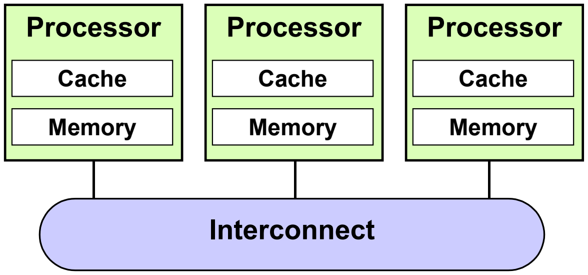
  These schemes, both snooping and directory, are based on a model in which all the processors have uniform access to the shared memory.



  The directory technique extends to large-scale multiprocessor systems like Flash and Alewife, distributed shared memory, and locks in clusters[9], in which the memory is attached

[9] Kronenberg, N. et al, The VAXCluster concept: An overview of a distributed system, *Digital Technical Journal* **1**, 3 (1987), pp 7-21.

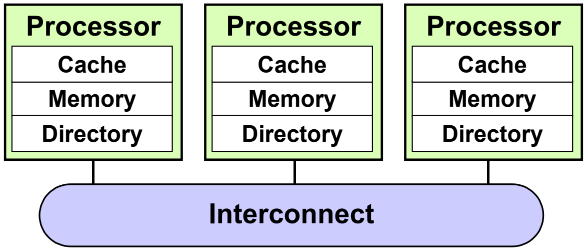
to processors. When the abstraction is memory rather than files, these systems are often called 'non-uniform memory access', or NUMA, systems.



The directory itself can be distributed by defining a 'home' location for each address that stores the directory information for that address. This is inefficient if that address turns out to be referenced mainly by other processors. To make the directory's distribution adapt better to usage, store the directory information for an address in a 'master' processor for that address, rather than in the home processor. The master can change to track the usage, but the home processor always remembers the master. Thus:

```
FUNC Home(a) -> P = …                                  % some fixed algorithm
VAR  master: P -> A -> P                               % master(p) is partial
     copies: P -> A -> SET P                           % defined only at the master
     locker: P -> A -> P                               % defined only at the master
INVARIANT (ALL a, p, p' |
            master(Home(a))!a                          % master is defined at a's home P,
         /\ master(p)!a /\ master(p')!a ==>            % where it's defined, it's the same
              master(p)(a) = master(p')(a)
         /\ copies!p = (p = master(Home(a))(a)) )      % and copies is defined only at master
```

The `Home` function is often a hash of `a`; it's possible to change the hash function, but if this is not atomic it must be done very carefully, because `Home` will be different at different processors and the invariants must hold for all the different `Home`'s.



- Hierarchical: 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. The hierarchy may have more levels, with the processor sets further subdivided, as in Flash.

There are many issues for high-performance code: communication cost, bandwidth into the cache into tag store, interleaving, and deadlock. The references at the start of this handout go into a lot of detail.

Purely software code is also possible. This form of DSM makes `V` be a whole virtual memory page and uses page faults to catch memory operations that require software intervention, while allowing those that can be satisfied locally to run at full speed. A live page is mapped, read-only unless it is dirty; a page that isn't live isn't mapped.[10]

Code based on `IncoherentMemory`

Next we consider a different kind of code for `CoherentMemory` that runs on top of `IncoherentMemory`. This code guarantees coherence by using an external read/write locking discipline. This is an example of an important general strategy—using weaker memory together with a programming discipline to guarantee strong coherence.

The code uses read/write locks, as defined earlier in the course, one per data item. There is a module $\texttt{ExternalLocks}_p$ for each processor `p`, which receives external `Read` and `Write` requests, obtains the needed locks, and invokes low-level `Read`, `Write`, and `Barrier` operations on the underlying `IncoherentMemory` memory. The composition of these pieces implements `CoherentMemory`. We give the code for $\texttt{ExternalLocks}_p$.

```
MODULE ExternalLocks_p [A, V] EXPORT Read, Write =     % implements CoherentMemory

% ReadAcquire_p acquires a read lock for processor p.
% Similarly for ReadRelease, WriteAcquire, WriteRelease

PROC Read_p =
    ReadAcquire_p; Barrier_p; VAR d| d := IncoherentMemory.Read_p; ReadRelease_p; RET d

PROC Write_p(d) = WriteAcquire_p; IncoherentMemory.Write_p(d); Barrier_p; WriteRelease_p

END ExternalLocks_p
```

This code does not satisfy all the invariants of `CurrentCaches` and its code. In particular, the data in caches is not always current, as stated in `Inv4`. It is only guaranteed to be current if it is read-locked, or if it is write-locked and dirty.

Invariants `Inv1`, `Inv2`, and `Inv3` are still satisfied. Invariants `Inv5` and `Inv6` no longer apply because the lock discipline is completely different; in particular, a locked copy need not be the only copy of an item. Let `wLockPs` be the set of processors that have a write-lock, and `rLockPs` be those with a read-lock.

We thus have `Inv1-3`, and new `Inv4a-Inv7a` that replace `Inv4-Inv7`:

[10] K. Li and P. Hudak, Memory coherence in shared virtual memory systems, *ACM Transactions on Computer Systems* **7**, 4 (Nov 1989), pp 321-359.

```
% INVARIANT Inv4a:                                       % Data is current
    (ALL p | dirty_p \/ (p IN rLockPs /\ Live_p) ==> Current_p())

% INVARIANT Inv5a:                                       % Write lock is exclusive.
     wLockPs.size <= 1

% INVARIANT Inv6a:                                       % Write lock excludes read locks.
     wLockPs # {} ==> rLockPs = {}

% INVARIANT Inv7a: (ALL p | dirty_p ==> p IN wLockPs)    % dirty data is write-locked
```

With these invariants, the identity abstraction to GlobalImpl works:

% ABSTRACTION to GlobalImpl: Identity on m, c, and dirty.

We note some differences between ExternalLocks and ExclusiveLocks, which also uses exclusive locks for writing:

- In ExclusiveLocks, Read can always proceed if there is a cache copy. In ExternalLocks, Read has a stronger guard in ReadAcquire (requiring a read lock).
- In ExclusiveLocks, MtoC checks that no other processor has a lock on the item. In ExternalLocks, an MtoC can occur as long as it doesn't overwrite dirty writes.
- In ExternalLocks, the guard for Acquire only involves lock conflicts, and does not check Only. (In fact, ExternalLocks doesn't use Only at all.)
- Additional Barrier actions are required in ExternalLocks.
- In ExclusiveLocks, the data in the cache is always current. In ExternalLocks, it is only guaranteed to be current for read-lock holders, and for write-lock holders who have already written.

In practice we don't surround every read and write with Acquire and Release. Instead, we take advantage of the rules for easy concurrency and rely on the fact that any reference to a shared variable must be in a critical section, surrounded by Acquire and Release of the lock that protects it. All we need to add is a Barrier at the beginning of the critical section, after the Acquire, and another at the end, before the Release. Sometimes people build these barrier actions into the acquire and release actions; this is called 'release consistency'.

Note—here we give up the efficiency of continuing to hold the lock until someone else needs it.

Remarks

Costs of incoherent memory

IncoherentMemory allows a multiprocessor shared memory to respond to Read and Write actions without any interprocessor communication. Furthermore, these actions only require communication between a processor and the global memory when a processor reads from an address that isn't in its cache. The expensive operation in this spec is Barrier, since the sequence Write$_p$; Barrier$_p$; Barrier$_q$; Read$_q$ requires the value written by p to be communicated to q. In most code Barrier is even more expensive because it acts on all addresses at once. This means that roughly speaking there must be at least enough communication to record globally every address that p wrote before the Barrier$_p$, and to drop from p's cache every address that is globally recorded as dirty.

Read-modify-write operations

Although this isn't strictly necessary, all current codes have additional external actions that make it easier to program mutual exclusion. These usually take the form of some kind of atomic read-modify-write operation, for example an atomic swap or compare-and-swap of a register value and a memory value. A currently popular scheme is two actions: ReadLinked(a) and WriteConditional(a), with the property that if any other processor writes to a between a ReadLinked$_p$(a) and the next WriteConditional$_p$(a), the WriteConditional leaves the memory unchanged and returns an indication of failure. The effect is that if the WriteConditional succeeds, the entire sequence is an atomic read-modify-write from the viewpoint of another processor, and if it fails the sequence is a SKIP. Compare-and-swap is obviously a special case; it's useful to know this because something as strong as compare-and-swap is needed to program wait-free synchronization using a shared memory. Of course these operations also incur communication costs, at least if the address a is shared.

We have shown that a program that touches shared memory only inside a critical section cannot distinguish memory that satisfies IncoherentMemory from memory that satisfies the serial spec CoherentMemory. This is not the only way to use IncoherentMemory, however. It is possible to program other standard idioms, such as producer-consumer buffers, without relying on mutual exclusion. We leave these programs as an exercise for the reader.

Caching as easy concurrency

We developed the coherent caching code by evolving from the obviously correct GlobalImpl to code that has no global operations except to acquire locks. Another way to look at it is that coherent caching is just a variation on easy concurrency. Each Read or Write touches a shared variable and therefore must be done with a lock held, but there are no bigger atomic operations. The read lock is Live and the write lock is lock. In order to avoid the overhead of acquiring and releasing a lock on every memory operation, we use the optimization of holding onto a lock until some other cache needs it.

Write buffering

Hardware caches, especially the 'level 1' caches closest to the processor, usually come in two parts, called the cache and the write buffer. The latter holds dirty data temporarily before it's written back to the memory (or the level 2 cache in most modern systems). It is small and optimized for high write bandwidth, and for combining writes to the same cache block that happen close together in time into a single write of the entire block.

Invalidation

All caching systems have some provision for invalidating cache entries. A system that implements CoherentMemory usually must invalidate a cache entry that is written on another processor. The invalidation must happen before any read that follows the write touches the entry.

Many systems, however, provide less coherence. For example, NFS simply times out cache entries; this implements `IncoherentMemory`, with the clumsy property that the only way to code `Barrier` is to wait for the timeout interval. The web does caching in client browsers and also in proxies, and it also does invalidation by timeout. A web page can set the timeout interval, though not all caches respect this setting. The Internet caches the result of DNS lookups (that is, the IP address of a DNS name) and of ARP lookups (that is, the LAN address of an IP address). These entries are timed out; a client can also discard an entry that doesn't seem to be working. The Internet also caches routing information, which is explicitly updated by periodic OSPF packets.

Think about what it would cost to make all these loosely coherent schemes coherent, and whether it would be worth it.

Locality and granularity

Caching works because the patterns of memory references exhibit locality. There are two kinds of locality.

- Temporal locality: if you reference an address, you are likely to reference it again in the near future, so it's worth keeping that item in the cache.
- Spatial locality: if you reference an address, you are likely to reference a neighboring address in the near future. This makes it worthwhile to transfer a large block of data to the cache, since the overhead of a miss is only paid once. Large blocks do have two drawbacks: they consume more bandwidth, and they introduce or increase 'false sharing'. A whole block has to be invalidated whenever any part of it is written, and if you are only reading a different part, the invalidation makes for extra work.

Both temporal and spatial locality can be improved by restructuring the program, and often this restructuring can be done automatically. For instance, it's possible to rearrange the basic blocks of a program based on traces of program execution so that blocks that normally follow each other in traces are in the same cache line or virtual memory page.

Distributed file systems

A distributed file system does caching which is logically identical to the caching that a memory system does. There are some practical differences:

- A DFS is usually built without any hardware support, whereas most DSM's depend at least on the virtual memory system to detect misses while letting hits run at full local memory speed, and perhaps on much more hardware support, as in Flash.
- A DFS must deal with failures, whereas a DSM usually crashes a program that is sharing memory with another program that fails.
- A DFS usually must scale better, to hundreds or thousands of nodes.
- A DFS has a wider choice of granularity: whole files, or a wide range of block sizes within files.

.