Coherence and Consistency
The Problem is Multi-Fold

Cache Consistency (taken from web-cache community)

In the presence of a cache, reads and writes behave (to a first order) no differently than if the cache were not there

Three main issues:

- Consistent with backing store
- Consistent with self
- Consistent with other clients of same backing store
Consistency w/ Backing Store

For example, write-through vs. write-back

Write buffer commonly used in write-through caches:

For example, write-through vs. write-back

Write buffer commonly used in write-through caches:
Consistency w Self

Virtual cache synonym problem & hardware solutions

Address Space A

Physical Memory

Virtual Cache

Address Space B

Address Space A

Physical Memory

Virtual Cache

Address Space B

Direct-Mapped Virtual Cache

OR

Set-Associative Virtual Cache (w/ physical tags)
The discussion is relatively long, so we have placed it in Chapter 31, Section 31.1.7, "Perspective: Segmented Addressing Solves the Synonym Problem."

An important item to note regarding aliasing and set-associative caches is that set associativity is usually a transparent mechanism (the client is not usually aware of it), and the cache is expected to guarantee that the implementation of set associativity does not break any models. Thus, a set-associative cache cannot use virtual tags unless the set associativity is exposed to the client. If virtual tags are used by the cache, the cache has no way of identifying aliases to the same physical block, and so the cache cannot guarantee that a block will be unique within a set—two different references to the same block, using different virtual addresses, may result in the block being homed in two different blocks within the same set.

Perspective on Aliasing

Virtual-address aliasing is a necessary evil. It is useful, yet it breaks many simple models. Its usefulness outweighs its problems. Therefore, future memory-management systems must continue to support it.

Most of the software solutions for the virtual cache synonym problem address the consistency problem by limiting the choices where a process can map a physical page in its virtual space. In some cases, the number of choices is reduced to one; the page is mapped at one globally unique location or it is not mapped at all. While disallowing virtual aliases would seem to be a simple and elegant way to solve the virtual-cache-consistency problem, it creates another headache for operating systems—virtual fragmentation.

FIGURE 4.4: Synonym problem solved by operating system policy. OS/2 and the operating system for the SPUR processor guarantee the consistency of shared data by mandating that shared segments map into every process at the same virtual location. SunOS guarantees data consistency by aligning shared pages on cache-size boundaries. The bottom few bits of all virtual page numbers mapped to any given physical page will be identical, and the pages will map to the same location in the cache. Note that this works best with a direct-mapped cache.
Consistency w Self

Segmentation as a solution to the aliasing problem
**Consistency w Self**

Segmentation as a solution to the aliasing problem

Process A  
Process B  
Process C  

Global Virtual Space

Paged Segment

Physical Memory

NULL (segment only partially-used)
Consistency w Self

ASID remapping

Virtual Address

ASID  VPN  PFN
ASID  VPN  PFN
...  ...  ...
ASID  VPN  PFN

Physical Address

Page Frame Number  Page Offset
Consistency w Other Clients

i.e. Cache Coherence & various Consistency Models

First, a look at some of the things that can go wrong, just inside a SINGLE CHIP:
Proc B reads data from dev A, signals proc C when done (producer-consumer pair)

Process C (consumer):

```c
global char data[SIZE];
global int ready=0;

while (1) {
    while (!ready)
        ;
    process( data );
    ready = 0;
}
```

Process B (producer):

```c
int fd = open("dev A");

while (1) {
    while (ready)
        ;
    dma( fd, data, SIZE );
    ready = 1;
}
```
Proc B reads data from dev A, signals proc C when done (producer-consumer pair) — more detail

1. After 'ready' is set to 0, device A transfers data into the memory system.
2. A communicates with device driver.
3. B is signaled via driver.
4. B updates synchronization variable 'ready' to 1.
5. C uses both data buffer and synchronization variable: while (!ready) // spin
   ;
   x = data[i]; // read data buffer
Proc B reads data from dev A, signals proc C when done (producer-consumer pair)
Proc B reads data from dev A, signals proc C when done (producer-consumer pair)

- Proc B reads data from dev A.
- Signals proc C when done.

Key components:
- DMA
- I/O
- GPU
- Core
- DRAM

1. Data - delayed in DMA
2. Driver sync
3. "Ready" variable
4. Data sent by DMA
5. ... arrives too late

Legend:
- Red dashed line: Data - delayed in DMA
- Green line: Driver sync
- Yellow line: "Ready" variable
- Blue line: Data sent by DMA... arrives too late

Bruce Jacob
University of Crete

SLIDE 13
Proc B reads data from dev A, signals proc C when done (producer-consumer pair)

1. I/O
2. DMA
3. GPU
4. CTL
5. Core
6. ... Core
7. DRAM

- Data held in CTL
- Driver sync
- "Ready" variable
- Data is stale
- Data sent to DRAM
Problem: causal relationships

![Diagram showing causal relationships between A, B, and C with data, done, and ready timelines.]
Problem scales with the system size

Solve system linear eqs: \( x_{i+1} = Ax_i + b \)

while (!converged) {
  doparallel(N) {
    int i = myid();
    xtemp[i] = b[i];
    for (j=0; j<N; j++) {
      xtemp[i] += A[i,j] * x[j];
    }
  }
  // implicit barrier sync
  doparallel(N) {
    int i = myid();
    x[i] = xtemp[i];
  }
}
Some Consistency Models

**Strict Consistency:** A read operation shall return the value written by the most recent store operation.

**Sequential Consistency:** The result of an execution is the same as a single interleaving of sequential, program-order accesses from different processors.

**Processor Consistency:** Writes from a process are observed by other clients to be in program order; all clients observe a single interleaving of writes from different processors.
Strict Consistency

Fails to satisfy strict consistency:

Satisfies strict consistency:

\[ A \text{ writes 1} \]

\[ B \text{ reads 0} \quad B \text{ reads 1} \]

\[ \Delta t \]

\[ \text{time} \]

\[ A \text{ writes 1} \]

\[ B \text{ reads 1} \quad B \text{ reads 1} \]

\[ \Delta t \]

\[ \text{time} \]
Sequential Consistency

Fails to satisfy strict consistency:
But satisfies Sequential Consistency

Satisfies strict consistency:
... and Sequential Consistency

A writes 1

B reads 0

B reads 1

\( A \) writes 1

\( B \) reads 0

\( B \) reads 1

\( \Delta t \)
Sequential Consistency

Handles our earlier problem:

Note: for this to work, memory controller may reorder internally, but not externally
Sequential Consistency

Requirements:

- Everyone can reorder internally but not externally
- All I/O & memory references must go through the same sync point (e.g. memory-mapped I/O)
- Write of data and driver signal must be same client
- Write buffering presents significant problems
- Reads must be delayed by system latency

…let’s look at this last one more closely
Really Famous Example (Goodman 1989)

**Process P1:**

*Initially, A=0*

A=1;

if (B==0) {
    kill P2;
}

**Process P2:**

*Initially, B=0*

B=1;

if (A==0) {
    kill P1;
}

Sequential Consistency allows 0 or 1 processes to die *(not both)*
Race-Condition Example

Fails to satisfy sequential consistency:

<table>
<thead>
<tr>
<th>Time</th>
<th>P1</th>
<th>P2</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>write A →</td>
<td>write B →</td>
</tr>
<tr>
<td></td>
<td>A=1</td>
<td>B=1</td>
</tr>
<tr>
<td></td>
<td>read B</td>
<td>read A</td>
</tr>
<tr>
<td></td>
<td>B=1</td>
<td>A=1</td>
</tr>
</tbody>
</table>

Figure 4.12: Sequential consistency and racing threads. A memory system that satisfies sequential consistency must delay all memory operations following a write until the write is observed by all other clients. Otherwise, it would be possible to have both P1 and P2 try to kill each other (as in the scenario on the left), which is disallowed by the sequential model. For example, the earliest that a subsequent read can follow a write is the message-propagation time within the system. Alternatively, a processor can speculate, allowing reads to execute early and patching up if problems are later detected.
Race-Condition Example

Satisfies sequential consistency:

- **P1**
  - Write A
  - A = 1
  - Read B
  - B = 1

- **P2**
  - Write B
  - B = 1
  - Read A
  - A = 1

Time
Race-Condition Example

In practice:

• speculate
  • throw exception if problem occurs

HOWEVER — from Jaleel & Jacob [HPCA 2005]:

• increasing the reorder buffer from 80 to 512 entries results in an increase in memory traps by 6x and an increase in total execution overhead by 10–40%
  
• reordering memory instructions increases L1 data cache accesses by 10–60% and L1 data cache misses by 10–20%
Processor Consistency

Also called **Total Store Order**

All writes in program order, reads freely reordered

Both of these scenarios are satisfied in this model:
Some Other Consistency Models

**Partial store order** — a processor can freely reorder local writes with respect to other local writes

**Weak consistency** — a processor can freely reorder local writes ahead of local reads

**Release consistency** — different classes of synchronization

... enforces synchronization only w.r.t. *acquire/release* operations. On *acquire*, memory system updates all protected variables before continuing; on *release*, memory system propagates changes to the protected variables out to the rest of the system
Cache Coherence Schemes

Ways to implement a consistency model:

- **in software** (e.g. in virtual memory system, via page table)
- **in hardware**
- combine hardware & software

The hardware component is called “cache coherence”
Coherence Implementations

Cache-Block States:

I — Invalid

M — Modified — read-writable, forwardable, dirty

S — Shared — read-only (can be clean or dirty)

E — Exclusive — read-writable, clean

O — Owned — read-only, forwardable, dirty
Coherence Implementations: SI

Works with write-through caches

Block is either present (Shared) or not (Invalid)

Problem: Nobody wants to use write-through caches

Both schemes require broadcast or multicast of coherence information and/or write data

Note: write-update and sequential consistency don’t play nice together
Coherence Implementations: MSI

Write-back caches: dirty bit (Modified state)

Problem: when the app reads data and then writes, sends a second broadcast
Coherence Implementations: MESI

Reduces write broadcasts

Problem: When you ask for a block, potentially many clients may respond
Coherence Implementations: MESIF

Shared broken into two: Shared (1+) and Forwardable (1)

Compare MESI (left) vs. MESIF (right):

Problem: All coherence info goes through central point: the backing store
Coherence Implementations: MOESI

- **I: Invalid**
  - Load miss from memory
  - Instruction cache miss, or load miss from another cache

- **O: Owned**
  - Successive stores to a line shared by another processor. Broadcast invalidate to all sharing processors

- **S: Shared**
  - Store miss on an invalidate cache line
  - Stored to a shared line. Broadcast invalidate to all sharing processors

- **M: Modified**
  - Request from another processor for a modified line

- **E: Exclusive**
  - Store to an exclusive line in cache
### MESI vs MOESI (AMD) vs MESIF (Intel)

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Dirty</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Must writeback to share or replace</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Clean</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>MSIF</td>
</tr>
<tr>
<td>Shared</td>
<td>Clean</td>
<td>No</td>
<td>No</td>
<td>I</td>
<td>Does not forward</td>
</tr>
<tr>
<td>Invalid</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>Cannot Read</td>
</tr>
<tr>
<td>Forwarding</td>
<td>Clean</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
<td>SI</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Dirty</td>
<td>Yes</td>
<td>Yes</td>
<td>O</td>
<td>Can share without writeback</td>
</tr>
<tr>
<td>Owned</td>
<td>Dirty</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Must writeback to transition</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Clean</td>
<td>Yes</td>
<td>Yes</td>
<td>MSi</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>Either</td>
<td>No</td>
<td>No</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Modified</td>
<td>Dirty</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
<td>Must writeback to share or replace</td>
</tr>
<tr>
<td>Exclusive</td>
<td>Clean</td>
<td>Yes</td>
<td>Yes</td>
<td>MSi</td>
<td></td>
</tr>
<tr>
<td>Shared</td>
<td>Clean</td>
<td>No</td>
<td>No</td>
<td>I</td>
<td></td>
</tr>
<tr>
<td>Invalid</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td>NA</td>
<td></td>
</tr>
</tbody>
</table>

- MESI: Modified, Exclusive, Shared, Invalid
- MOESI: Modified, Exclusive, Shared, Invalid
- MESIF: Modified, Owned, Exclusive, Shared, Invalid
Some System Configurations

Each cache block in a data structure associated with that cache block. The data structure contains such information as the block's ownership, its sharing status, etc. These data structures are all held together in a directory, which can be centralized or distributed; when a client makes a request for a cache block, its corresponding directory entry is first consulted to determine the appropriate course of action.

A snoop-based scheme uses no such per-block data structure. Instead, the appropriate course of action is determined by consulting every client in the system. On every request, each cache in the system is consulted and responds with information on the requested block; the collected information indicates the appropriate response. For instance, rather than looking up the owner of a block in the block's directory entry as would be the appropriate step in a directory-based scheme, in a snoopy scheme the owner of the block actively responds to a coherence broadcast, indicating ownership and returning the requested data (if such is the appropriate response).

Snoopy Protocols

In a snoopy protocol, all coherence-related activity is broadcast to all processors. All processors analyze all activity, and each reacts to the information passing through the system based on the contents of its caches. For example, if one processor is writing to a given data cache line, and another processor has a copy of the data cache line, then the second processor must invalidate its own cache line. After writing the block, the first processor now has a dirty copy. If the second processor then makes a read request to that block, the first processor must provide it.

Snoopy protocols seem to imply the existence of a common bus for their implementation, but they need not use common busses if there is agreement.

FIGURE 4.18: The many faces of backing store. The backing store in a multiprocessor system can take on many forms. In particular, a primary characteristic is whether the backing store is distributed or not. Moreover, the choices within a distributed organization are just as varied. The two organizations on the bottom right are different implementations of the design on the bottom left.
Bus-Based, Hierarchical

Backcoing Store

The many faces of backing store. The backing store in a multiprocessor system can take on many forms. In particular, a primary characteristic is whether the backing store is distributed or not. Moreover, the choices within a directory-based scheme, in a snoopy scheme the owner of a block in the block’s directory is consulted and responds with information on the requested block; the collected information indicates which processor must take which action is determined by consulting every client in the data structure. Instead, the appropriate course of action is determined by consulting every client in the data structure. Even this:

CPU
dRAM
Controller
System & Memory
Controller
Backing Store
Backcoing Store

Core
Core
Core
Core
... Core

A snoopy-based scheme uses no such per-block information. Even this:
Directory-Based Protocols

Can run on any configuration—the main idea is to eliminate the need to broadcast every coherence event.
Directory-Based Protocols

Each memory block has a directory entry

- P+1 bits where P is the number of processors
- One dirty bit per directory entry
- If dirty bit is on then only one presence bit can be on
- Nodes only communicate with other nodes that have the memory block
Directory-Based Protocols

Flavors:

- **Centralized Directory, Centralized Memory**
  Poor scalability, but better than bus-based ...

- **Decentralized Directory, Distributed Memory**
  Each node stores small piece of entire directory corresponding to the memory resident at that node. Directory queries sent to the node that the address of the block corresponds to (i.e. its **Home Node**).

- **Clustered**
  Presence bits are coarse-grained
Problem with Scalability

What if latency to other procs > latency to local DRAM?

- Other processor(s)
- Coherence status
- Coherence check
- Read command
- Read data
- Processor
- Most recent data
- Coherence point

Diagram:

- CPUs
- I/O
- System Controller
- Memory controller
- DRAM
- Coherence check
- Data retrieval (from DRAM)
- Bus-level error checking
- MARD
- Parallel tasks
- Transaction request
- Serialize at respective points of coherence-synchronization
- Controller
- System
- Other CPU(s)
- Other CPU(s) and system controller