

Multiprocessors

The Problems with OOO

- Limited per-thread ILP
 - Bigger windows don't buy you that much
- Complexity
 - Building and verifying large OOO machines is hard (but doable)
- Area efficiency (per-xtr efficiency)
 - Doubling the area devoted to OOO mechanisms doesn't come close to doubling performance
- Power efficiency
 - Large OOO don't provide good power efficiency returns either.
- For all these reasons, OOO growth has almost stopped.

Frequency and Power

- $P = CfV^2$
- f = processor frequency
- V = supply voltage
- C = circuit capacitance (basically xtr count)
- To increase f you need to increase V as well
 - Approximately: $P = Cf^3$
- This means that even for in-order processors, frequency scaling is not power efficient
 - doubling the frequency doubles performance
 - increased power by 8x
- It is, however, very area-efficient/xtr-efficient

Multi-processors

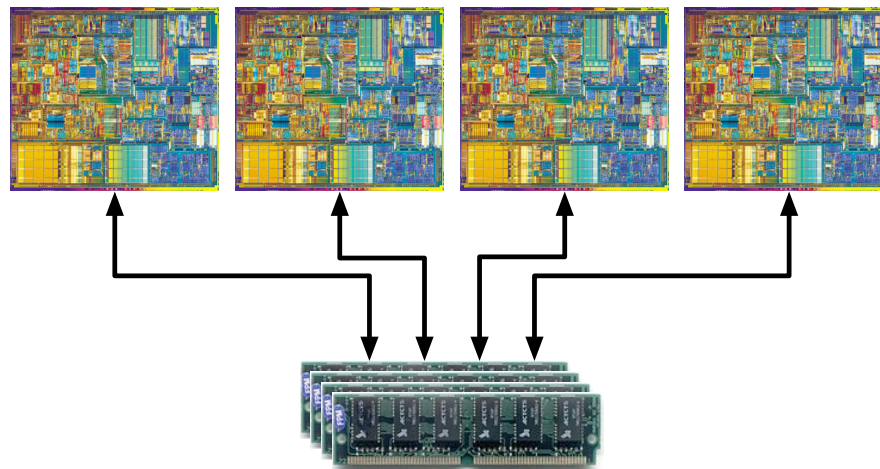
- An alternative approach to increased performance: Build more processors
- N processors will do N times as much work per time
- Area efficiency:
 - Pretty good -- twice the area -> twice the performance (Maybe. Sometimes. More on this in moment)
- Power efficiency:
 - $P = Cf^3$
 - Two processors means doubling C, so 2x the power.

In short

- Building bigger OOO processors doesn't pay
- Power budgets are fixed.
- Moore's law keeps delivering more xtrs
- Consequences
 - Power efficiency is more important than area efficiency
 - Multi-processors are now more attractive.

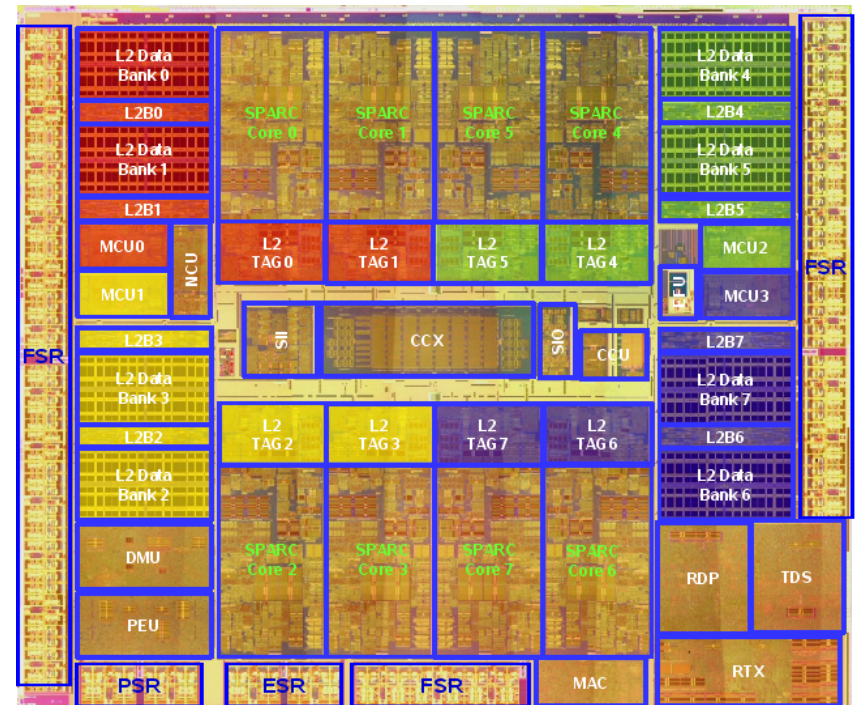
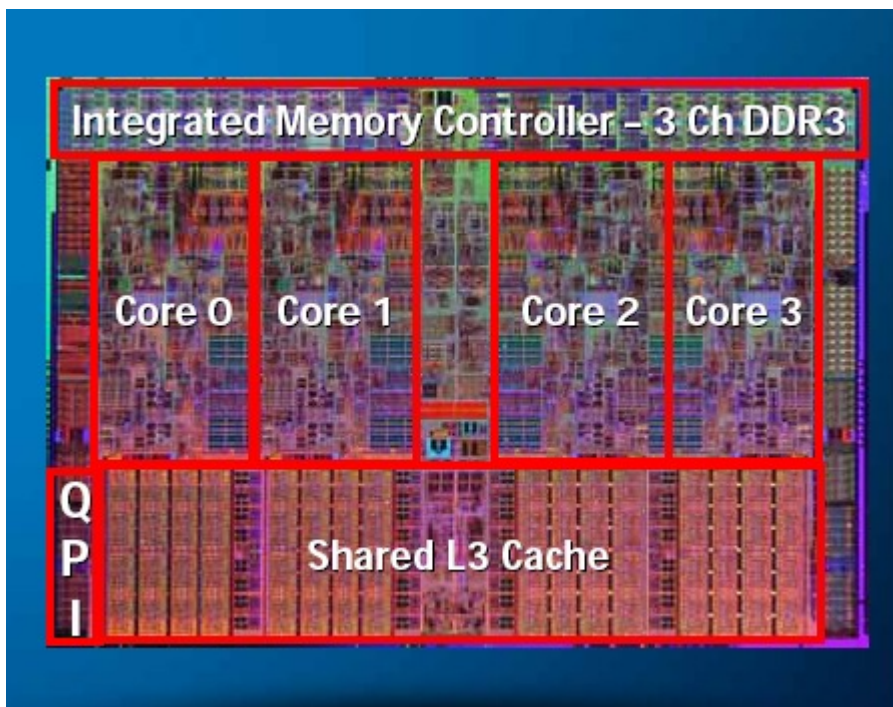
Multiprocessors

- Specifically, shared-memory multiprocessors have been around for a long time.
- Originally, put several processors in a box and provide them access to a single, shared memory.
- Expensive and mildly exotic.
 - Big servers
 - Sophisticated users/data-center applications



Chip Multiprocessors (CMPS)

- Multiple processors on one die
- An easy way to spend xtrs
- Now common place
 - Laptops/desktops/game consoles/mobile phones etc.
 - Less sophisticated users, all kinds of applications.



The Trouble With CMPs

- Amdahl's law
 - $Stot = 1/(x/S + (1-x))$
- In order to double performance with a 2-way CMP
 - $S = 2$
 - $x = 1$
 - Usually, neither is achievable

Threads are Hard to Find

- To exploit CMP parallelism you need multiple processes or multiple “threads”
- Processes
 - Separate programs actually running (not sitting idle) on your computer at the same time.
 - Common in servers
 - Much less common in desktop/laptops
- Threads
 - Independent portions of your program that can run in parallel
 - Most programs are not multi-threaded.
- We will refer to these collectively as “threads”
 - A typical user system might have 1-4 actively running threads.
 - Servers can have more if needed (the sysadmins will hopefully configure it that way)

Parallel Programming is Hard

- Difficulties
 - Correctly identifying independent portions of complex programs
 - Sharing data between threads safely.
 - Using locks correctly
 - Avoiding deadlock
- There do not appear to be good solutions
 - We have been working on this for 30 years (remember, multi-processors have been around for a long time.)
 - It remains stubbornly hard.

Critical Sections and Locks

- A critical section is a piece of code that only one thread should be executing at a time.

```
int shared_value = 0;
void IncrementSharedVariable()
{
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
}
```

- If two threads execute this code, we would expect the shared_value to go up by 2
- However, they could both execute line 1, and then both execute line 2 -- both would write back the same new value.

Instructions in the two threads can be interleaved in any way.

Critical Sections and Locks

- By adding a lock, we can ensure that only one thread executes the critical section at a time.

```
int shared_value = 0;
lock shared_value_lock;
void IncrementSharedVariable()
{
    acquire(shared_value_lock);
    int t = shared_value + 1; // Line 1
    shared_value = t;         // line 2
    release(shared_value_lock);
}
```

- In this case we say `shared_value_lock` “protects” `shared_value`.

Locks are Hard

- The relationship between locks and the data they protect is not explicit in the source code and not enforced by the compiler
- In large systems, the programmers typically cannot tell you what the mapping is (!!!)
- As a result, there are many bugs.

Locking Bug Example

```
void Swap(int * a, lock * a_lock,  
          int * b, lock * b_lock) {  
    lock(a_lock);  
    lock(b_lock);  
    int t = a;  
    a = b;  
    b = t;  
    unlock(a_lock);  
    unlock(b_lock);  
}
```

Thread 1

```
...  
Swap(foo, foo_lock,  
      bar, bar_lock);  
...
```

Thread 2

```
...  
Swap(bar, bar_lock,  
      foo, foo_lock);  
...
```

Thread 1 locks foo_lock, thread 2 locks bar_lock, both wait indefinitely for the other lock.

Finding, preventing, and fixing this kind of bug are all hard

Granularity

- Frequent hash_map operations: insert, lookup, delete.
 - fine-grain locking is worth it.
- Infrequent operations: rebalance
 - Fine-grain locking is not worth it (amdahl's law)
- Since each piece of data is protected by a particular lock, both must be implemented with fine-grain locking.
 - What a pain! and a waste of time. For (almost) no speedup!

Composability

- Lock-based systems are often not composable.
- Example: Given a highly concurrent `hash_map<int>` implementation
 - It does internal locking to allow many threads to access it efficiently at once.
- You want to do this:
 - `t = map["foo"];`
 - `map.delete("foo");`
 - `// map is shared.`

Composability

- “Solution”
 - `// protect map with a lock`
 - `lock(...);`
 - `t = map[“foo”];`
 - `map.delete(“foo”);`
 - `unlock(...);`
 - So much for the fine-grain locking.
- Solution?
 - make `map.delete()` return the value.
 - hurrah!

Composability

- Lock-based systems are often not composable.
- Example: Given a highly concurrent `hash_map<int>` implementation
 - It does internal locking to allow many threads to access it efficiently at once.
- You want to do this:
 - `t = map.delete("foo");`
 - `map2["foo"] = t;`
 - `// map and map2 are shared`

Composability

- “Solution”
 - // protect map and map2 with locks
 - lock(...); lock(...);
 - t = map.delete(“foo”);
 - map2[“foo”] = t;
 - unlock(...); unlock(...)
 - :-()
- Solution
 - a new method in the map to do “X”
- Problem
 - I want to move t to a linked list instead of another map...
 - and so on...

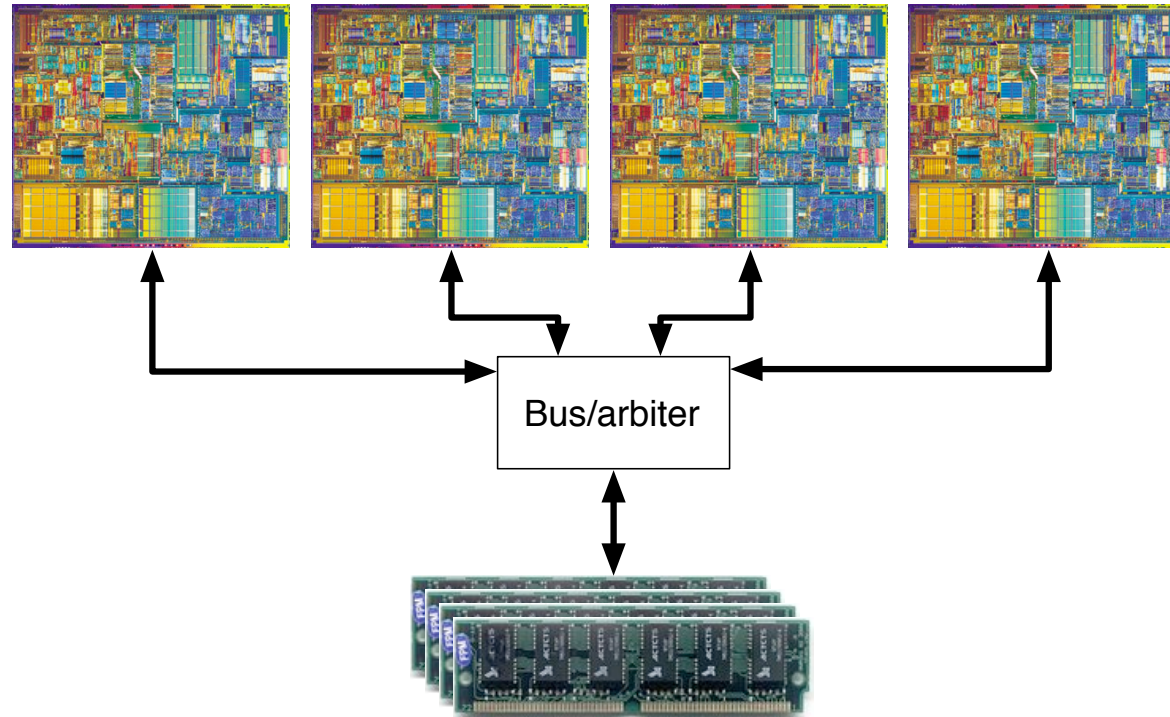
The Future of Threads

- Optimists believe that we will solve the parallel program problem this time!
 - New languages
 - New libraries
 - New paradigms
 - Revamped undergraduate programming courses
- Pessimists believe that we won't
 - There is probably not a good, general solution
 - We will make piecemeal progress
 - Most programs will stop getting faster
 - CMPs just make your spyware run faster.
- Intel and Microsoft believe typical users can utilize up to about 8 cores effectively.
 - Your laptop will be there in 2-3 years.

Architectural Support for Multiprocessors

- Allowing multiple processors in the same system has a large impact on the memory system.
 - How should processors see changes to memory that other processors make?
 - How do we implement locks?

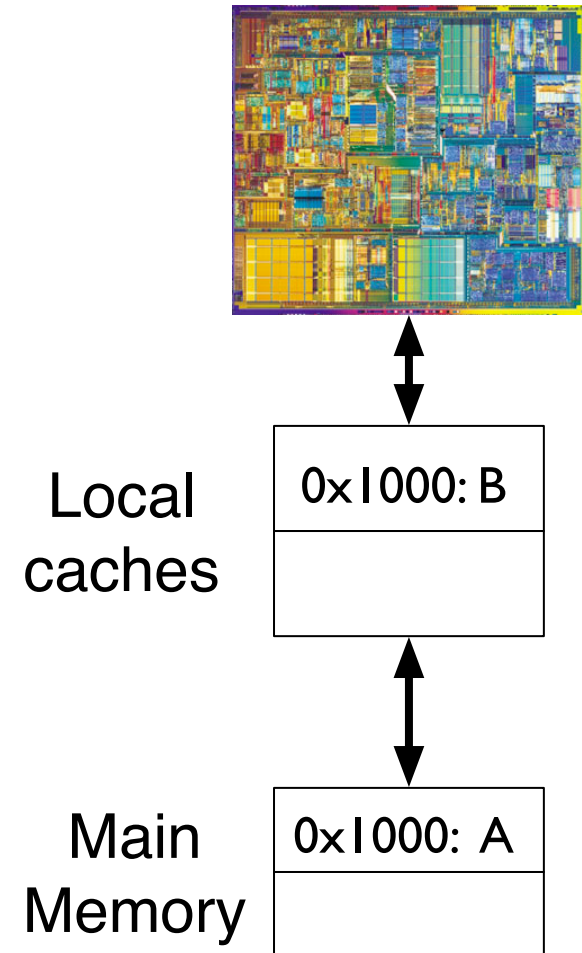
Shared Memory



- Multiple processors connected to a single, shared pool of DRAM
- If you don't care about performance, this is relatively easy... but what about caches?

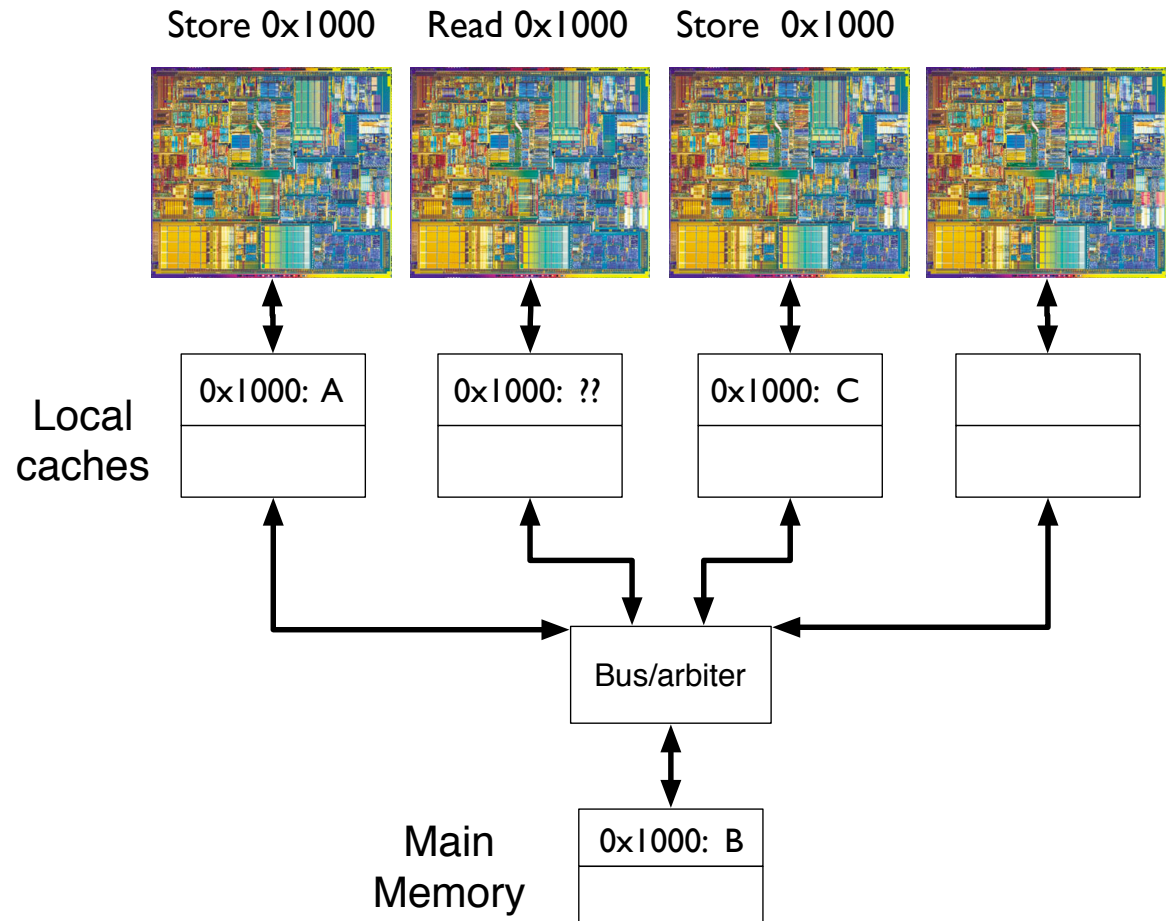
Uni-processor Caches

- Caches mean multiple copies of the same value
- In uniprocessors this is not a big problem
 - From the (single) processor's perspective, the “freshest” version is always visible.
 - There is no way for the processor to circumvent the cache to see DRAM's copy.



Caches, Caches, Everywhere

- With multiple caches, there can be many copies
- No one processor can see them all.
- Which one has the “right” value?



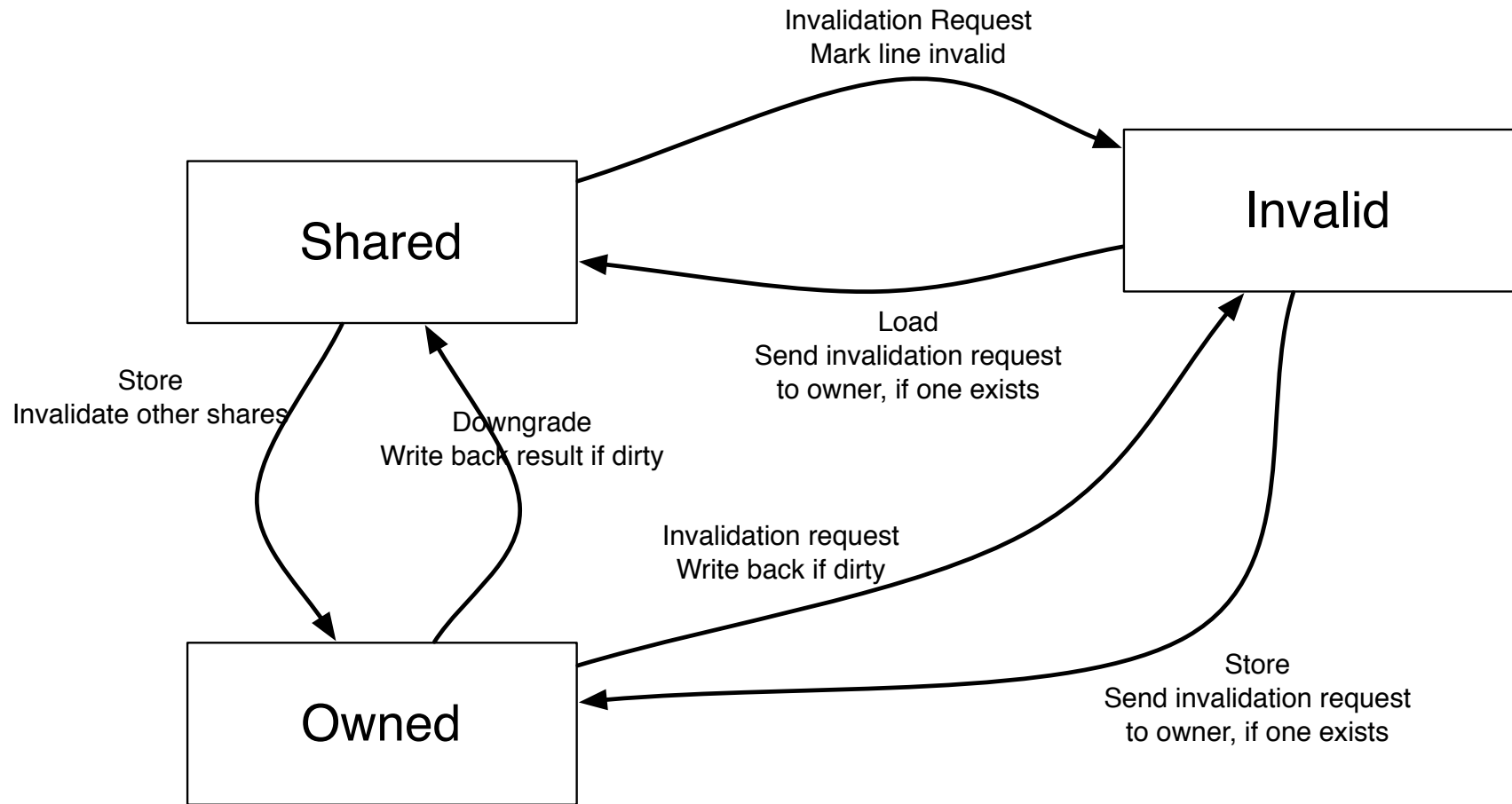
Keeping Caches Synchronized

- We must make sure that all copies of a value in the system are up to date
 - We can update them
 - Or we can “invalidate” (i.e., destroy) them
- There should always be exactly one current value for an address
 - All processors should agree on what it is.
- We will enforce this by forcing eviction of all copies of data from other's cache prior to when a core performs a store. This is called “Cache Coherence”

The Basics of Cache Coherence

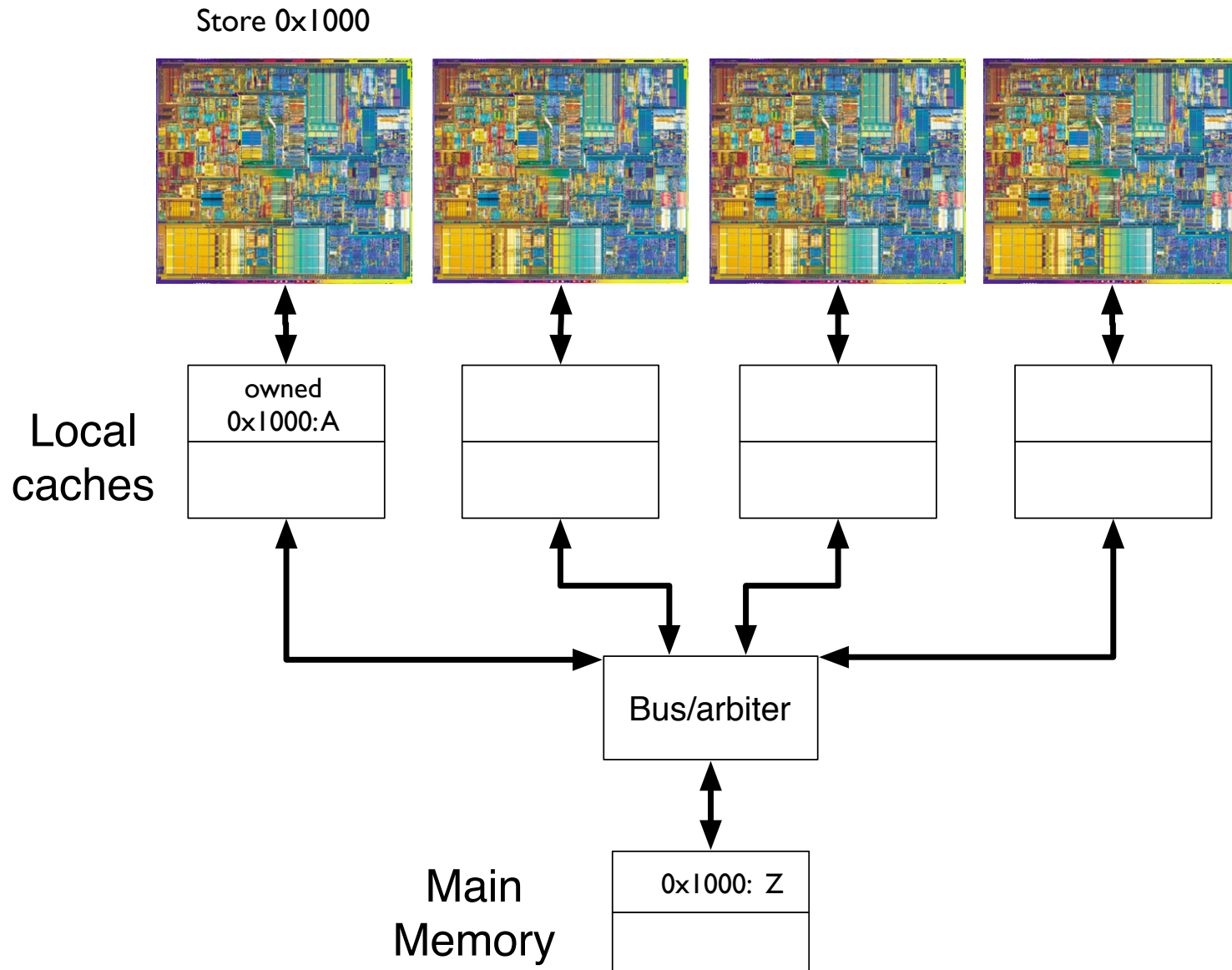
- Every cache line (in each cache) is in one of 3 states
 - Shared -- There are multiple copies but they are all the same. Only reading is allowed
 - Owned -- This is the only cached copy of this data. Reading and write are allowed
 - Invalid -- This cache line does not contain valid data.
- There can be multiple sharers, but only one owner.
- Cache coherence only applies to updates/ accesses to a single address.
 - It enforces a total ordering over those updates/ accesses.

Simple Cache Coherence

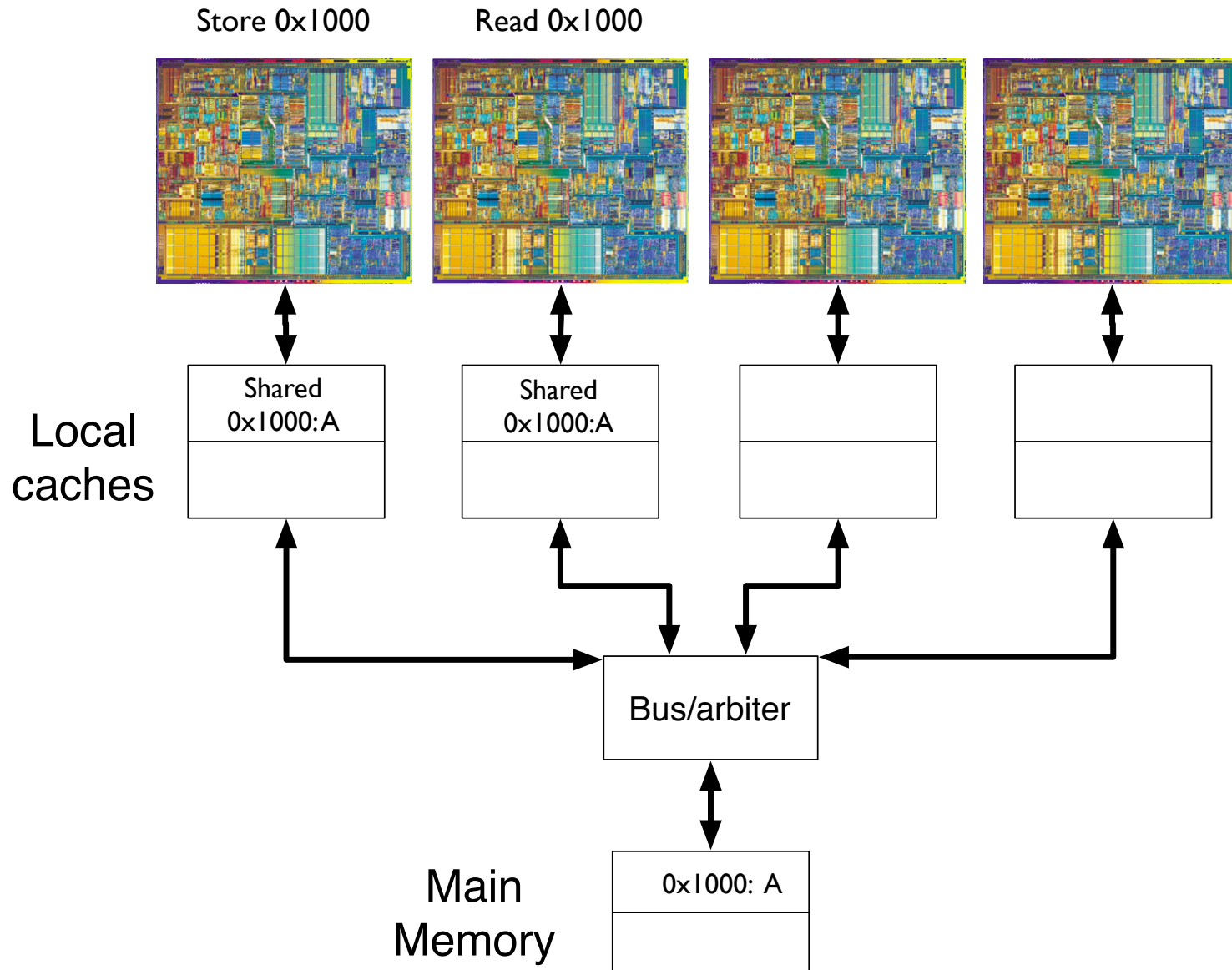


- There is one copy of the state machine for each line in each coherence cache.

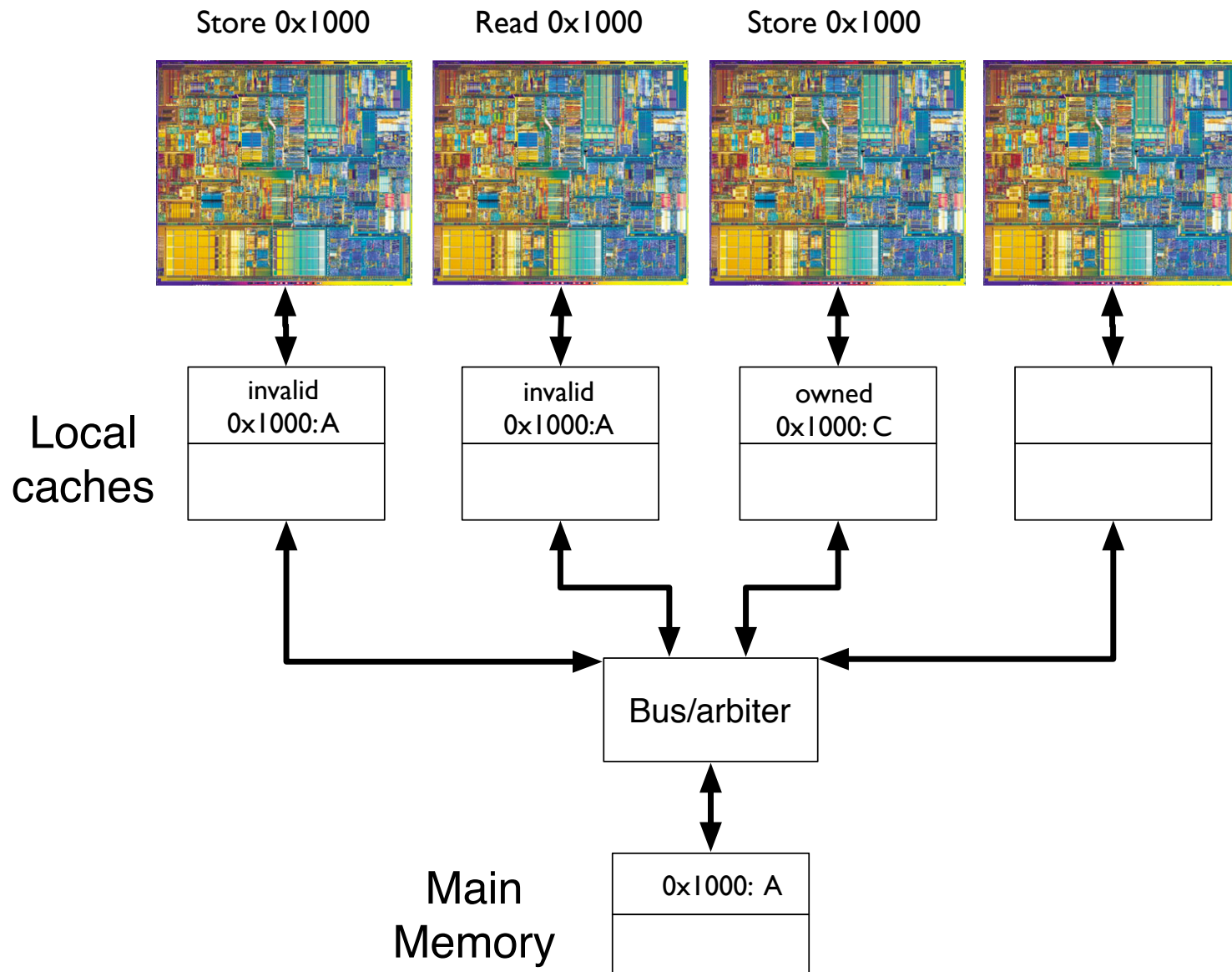
Caches, Caches, Everywhere



Caches, Caches, Everywhere



Caches, Caches, Everywhere



Coherence in Action

		Sample outputs		
		1	1	1
		2	1	2
		3	1	5
		4	1	8
		5	100	3
		6	100	5
		7	100	2
		8	100	4
possible?		yes	yes	no

a = 0

Thread 1

```
while (1) {
    a++;
}
```

Thread 2

```
while (1) {
    print(a);
}
```

Live demo.

Coherence In The Real World

- Real coherence have more states
 - e.g. “Exclusive” -- I have the only copy, but it's not modified
- Often don't bother updating DRAM, just forward data from the current owner.
- If you want to learn more, take 240b
- Here's a interesting thought:
 - In big systems everyone's talking about relaxed consistency for data, because tight coupling is too expensive.
 - How does that notion map to processors and memory.

Cache Consistency

- If two operations occur in an order in one thread, we would like other threads to see the changes occur in the same order.
 - Example:

Thread 0

```
A = 10;  
A_is_valid = true;
```

Thread 1

```
while(!A_is_valid);  
B = A;
```

- We want B to end up with the value 10
- Coherence does *not* give us this assurance, since the state machine only applies to a *single cache line*
- This is called “cache consistency” or “the consistency model”

Simple Consistency

- The simplest consistency model is called “sequential consistency”
- Corresponds to our most intuitive notion of how memory behaves.

Thread 0

```
A = 10;  
A_is_valid = true;
```

Thread 1

```
while(!A_is_valid);  
B = A;
```

- If thread 1 sees the write to `A_is_valid`, it will also see the write to `A`.

Sequential Consistency

- "... the result of any execution is the same as if the operations of all the processors were executed in some sequential order, and the operations of each individual processor appear in this sequence in the order specified by its program."
- Leslie Lamport, "How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs", IEEE Trans. Comput. C-28,9 (Sept. 1979), 690-691.
- Alternately: There appears to be a single, global ordering for all memory operations, and all processors observe memory operations in that order.

What about this?

Sample outputs

a = b = 0

Thread 1

```
while(1) {  
    a++;  
    b++;  
}
```

Thread 2

```
while(1) {  
    print(a, b);  
}
```

possible
under sequential
consistency?

1 1	
2 2	
3 3	1 1
4 4	2 2
5 5	2 1000
6 6	3 1000
7 7	4 1000
8 8	
yes	no

Consistency in the Real World

- Consistency is probably the most subtle aspect of computer architecture
- Sequential consistency has high overhead
- Real machines use “relaxed” models.
 - All manner of non-intuitive things can happen
 - Special instructions to enforce sequential consistency when it's needed (i.e., around locks)
- Threading libraries (like pthreads) provide locking routines that use those special instructions to make locks work properly.
- For more, take 240b

Alternatives to Locking

- What we really want to write is something like this:

```
int shared_value = 0;
void IncrementSharedVariable()
{
    atomic {
        int t = shared_value + 1;
        shared_value = t;
    }
}
```

- Which means “do these operations together, and don’t let anyone interfere”
 - Atomic, isolated, and consistent

Transactional Memory

- Instead of (poorly) specifying a mapping between locks and data, declare atomic regions and let the system figure it out.
- Fixes composability:
 - `atomic {`
 - `t = map["foo"];`
 - `map.delete("foo");`
 - `list.insert(t);`
 - `}`
- Fixes granularity
- It's beautiful!!! (mostly)

Software TM

- Start a transactions
 - As you go, acquire locks on all the data you touch.
 - The mapping between locks and data is managed for you, so you can't mess it up. In fact there may not even be locks per se.
 - If you find that someone holds a lock you need, “abort” by undoing your changes, releasing your locks. Try again.
 - If you get to the end, “commit” by making your changes permanent and releasing your locks.

TM Problems

- IO operations -- “you can’t unlaunch the missiles”
 - atomic {
 - Launch missiles.
 - if (missiles hit their target) i++;
 - }
 - Conflict occurs updating i.
 - Roll back...
 - This causes it’s own composability problems.
- Sloooooow....
 - 10x slowdown is not uncommon.
 - Not a problem if performance is not the key concern -- some apps easier to structure as multiple threads.

Hardware TM

- Leverage caches and the coherence system to hide local changes from the rest of the system.
- As you access data, hold it your local cache in an irrevocable exclusive state.
 - This corresponds to locking it.
- If another processor tries to access it, that access will fail or stall.
- On abort, drop the cache lines.
- On commit, grab the bus and write out all your updates at once.

Hardware TM Problems

- Fixed size max transaction size (limited by private cache)
 - Leads to SWE problems and breaks composability.
- Monopolizing the bus is potentially expensive.

TM and consistency/coherencve

- Sequential consistency is equivalent to putting each memory operation in its own transaction.
- You can use TM as a coherence protocol and consistency model.
 - [Kozyrakis 04]

TM was a hot research topic

- It was heralded as a solution to the MP programming problem
 - It is probably worthwhile
 - but it is no magic bullet.
 - Many, many, many, many, many, many, many papers on the subject
- At least two commercial products
 - Azul systems -- custom java hardware with TM support
 - A canceled (?) Sun processor.