Memory Consistency

Kathy Yelick

EECS Professor, UC Berkeley
Safety and Liveness

• A safety property says:
  • informally: nothing bad ever happens
  • formally: any violation of a safety property can be observed on a finite subsequence of the execution
  • this is a partial correctness condition

• A liveness property says:
  • informally: something good eventually happens
  • formally: it is a property of an infinite execution that cannot be checked on finite subsequences
  • this is a total correctness condition (combined with safety)

• Neither of these address real time constraints
Stable Properties

• A stable property: once true, it remains true “forever.”

• Termination detection, garbage detection, and deadlock are stable properties

• Stable properties are amenable to a set of proof techniques based on “snapshots algorithms”
Memory Consistency Model Definition #1 
(Sarita Adve and Kourosh Gharachorloo)

• A memory model provides a formal specification of the effect of read and write operations on the memory system and describes how memory appears to the programmer

• Bridges the gap between the behavior expected by the programmer and the actual behavior of the program.

• Memory model affects:
  • Programmability (easy-of-programming)
  • Performance (optimizations that it allows)
  • Portability (moving software across different systems)
Memory Consistency Model
(Wikipedia)

• In computer science, consistency models are used in distributed systems like distributed shared memory systems or distributed data stores (such as filesystems, databases, optimistic replication systems or Web caching).

• The systems supports a given model if operations on memory follow specified rules. The *data consistency model specifies a contract between programmer and system*, wherein the system guarantees that if the programmer follows the rules, memory will be consistent and the results of memory operations will be predictable.
Memory Consistency
*(Culler, Singh and Gupta)*

Def: A *memory consistency model* for a shared address space specifies constraints on the order in which memory operations must appear to be performed (i.e. to become visible to the processors) with respect to one another.

\[
\begin{array}{c|c|c}
P1 & P2 & (A, \text{flag are zero initial}) \\
A=1 & \text{while(flag == 0);} & \text{print A;} \\
\text{flag}=1 & \text{} & \text{} \\
\end{array}
\]

*(Culler, Singh, Gupta)*
Sequential Consistency

- **Sequential Consistency (Lamport)** “A multiprocessor is sequentially consistent if 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 occur in this sequence in the order specified by its program.”
Sequential Consistency Intuition

- Sequential consistency says the machine behaves as if processors take turns in an arbitrary order.

- Informal definition #1: The program behaves as if the threads take turns executing instructions (not in any fair order), i.e., only one thread executes an instruction at a given time.

- Note that "behaves as if" only says the reads and writes give such answers; it says nothing about how the machine actually executes instructions.

Katherine Yelick, Computer Science Division, EECS, University of California, Berkeley
Sequential Consistency

- Given a set of executions from n processors, each defines a total order $P_i$.
- The program order is the partial order given by the union of these $P_i$'s.
- The overall execution is **sequentially consistent** if there exists a correct total order that is consistent with the program order.

When this is serialized, the read and write semantics must be preserved.
Definitions of Sequential Consistency (More Formal)

- Given a parallel execution, the series of read and write operations from a given thread are totally ordered. The union of those total orders is a partial order $<_p$. The execution is **sequentially consistent** if there exists a total order consistent with $<_p$ that is a correct serial execution.
  - Correct = Reads return the value of the last write
  - All thread-local orderings are preserved
  - Still plenty of opportunities for races, other bugs
Sequential consistency

• Claim: Sequential consistency is so intuitive that most programmers do not need to read and understand a formal definition.

• And anything else is much harder to specify!
What does this imply about program behavior?

- No process ever sees “garbage” values, i.e., \( \frac{1}{2} \) of 2 values
- Processors always see values written by some processor
- The value seen is constrained by program order on all processors
  - Time always moves forward
- Example:
  - P1 writes data=1, then writes flag=1
  - P2 reads flag, then reads data

<table>
<thead>
<tr>
<th>If P2 reads flag</th>
<th>Then P2 may read data</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

Initially:
- flag=0
- data=0

<table>
<thead>
<tr>
<th>P1 data = 1 flag = 1</th>
<th>P2 if flag=1 ( \ldots = ) data</th>
</tr>
</thead>
</table>

Katherine Yelick, Computer Science Division, EECS, University of California, Berkeley
If Caches are Not Coherent

- p1 and p2 both have cached copies of data (as 0)
- p1 writes data=1
  - May “write through” to memory
- p2 reads data, but gets the “stale” cached copy
  - This may happen even if it read an updated value of another variable, flag, that came from memory
There are many other sources of non-Sequentially-Consistent Behavior

- Would like compiler to introduce put/get/store
- Hardware also reorders
  - out-of-order execution
  - write buffered with read by-pass
  - non-FIFO write buffers
  - weak memory models in general
- Software already reorders too
  - register allocation
  - any code motion
- System provides enforcement primitives
  - e.g., memory fence, volatile, etc.
  - tend to be heavyweight and have unpredictable performance
- Open question: Can the compiler hide all this?
Sequential Consistency Intuition

Sequential consistency says that:

- The compiler may only reorder operations if another processor cannot observe it.
- Writes (to variables that are later read) cannot result in garbage values being written.
- The program behaves as if processors take turns executing instructions.

Comments:

- In a legal execution, there will typically be many possible total orders – limited only the reads and writes to shared variables.
- This is what you get if all reads and writes go to a single shared memory, and accesses serialized at memory cell.
Two Problems

- Compiler writers would like to move code around
- The hardware folks also want to build hardware that dynamically moves operations around

- When is reordering correct?
  - Because the programs are parallel, there are more restrictions, not fewer
  - The reason is that we have to preserve semantics of what may be viewed by other processors
How Can Sequential Consistency Fail?

• The compiler saves a value in a register across multiple read accesses
  • This “moves” the later reads to the point of the first one

• The compiler saves a value in a register across writes
  • This “moves” the write until the register is written back from the standpoint of other processors.

• The compiler performance common subexpression elimination
  • As if the later expression reads are all moved to the first
  • Once contiguous in the instruction stream, they are merged

• The compiler performs other code motion

• The hardware has a write buffer
  • Reads may by-pass writes in the buffer (to/from different variables)
  • Some write buffers are not FIFO

• The hardware may have out-of-order execution

Katherine Yelick, Computer Science Division, EECS, University of California, Berkeley
Weaker Correctness Models

- Many systems use weaker memory models:
  - Sun has TSO, PSO, and RMO
  - Alpha has its own model
- Some languages do as well
  - Java also has its own, currently undergoing redesign
  - C spec is mostly silent on threads – very weak on memory mapped I/O
- These are variants on the following, **sequential consistency under proper synchronization**:
  - All accesses to shared data must be protected by a lock, which must be a primitive known to the system
  - Otherwise, all bets are off (extreme)
Why Don’t Programmers Care?

- If these popular languages have used these weak models successfully, then what is wrong?
  - They don’t worry about what they don’t understand
  - Many people use compilers that are not very aggressive about reordering
  - The hardware reordering is non-deterministic, and may happen very infrequently in practice

- Architecture community is way ahead of us in worrying about these problems.
- Open problem: A hardware simulator and/or Java (or C) compiler that reorders things in the “worst possible way”
Using Software to Mask Hardware

• Recall our two problems:
  1. Compiler writers would like to move code around
  2. The hardware folks also want to build hardware that dynamically moves operations around

• The second can be viewed as compiler problem
  • Weak memory models come extra primitives, usually called fences or memory barriers
    • Write fence: wait for all outstanding writes from this processor to complete
    • Read fence: do not issue any read pre-fetches before this point
Use of Memory Fences

• Memory fences can turn a particular memory model into sequential consistency under proper synchronization:
  • Add a read-fence to acquire lock operation
  • Add a write fence to release lock operation

• In general, a language can have a stronger model than the machine it runs if the compiler is clever

• The language may also have a weaker model, if the compiler does any optimizations
Aside: Volatile

• Because Java and C have weak memory models at the language level, they give programmers a tool: volatile variables
  • These variables should not be kept in registers
  • Operations should not be reordered
  • Should have mem fences around accesses

• General problem
  • This is a big hammer which may be unnecessary
  • No fine-grained control over particular accesses or program phases (static notion)
  • To get SC using volatile, many variables must be volatile
How Can Compilers Help?

• To implement a stronger model on a weaker one:
  • Figure out what can legal be reordered
  • Do optimizations under these constraints
  • Generate necessary fences in resulting code

• Open problem: Can this be used to give Java a sequentially consistent semantics?

• What about C?
Compiler Analysis Overview

- When compiling sequential programs, compute dependencies:

  \[
  \begin{align*}
  x &= \text{expr1;} \\
  y &= \text{expr2;}
  \end{align*}
  \]

  Valid if \( y \) not in \( \text{expr1} \) and \( x \) not in \( \text{expr2} \) (roughly)

- When compiling parallel code, we need to consider accesses by other processors.

  Initially \( \text{flag} = \text{data} = 0 \)

  \[
  \begin{align*}
  \text{Proc A} & & \text{Proc B} \\
  \text{data} &= 1; & \text{while (flag == 0);} \\
  \text{flag} &= 1; & \ldots = \ldots \text{data} \ldots;
  \end{align*}
  \]
**Cycle Detection**

- Processors define a “program order” on accesses from the same thread
  - $P$ is the union of these total orders
- Memory system define an “access order” on accesses to the same variable
  - $A$ is access order (read/write & write/write pairs)

  ![Diagram](image)

  - write data, read flag
  - write flag, read data

- A violation of sequential consistency is cycle in $P \cup A$ [Shash&Snir]
Cycle Analysis Intuition

- Definition is based on execution model, which allows you to answer the question: Was this execution sequentially consistent?

- Intuition:
  - Time cannot flow backwards
  - Need to be able to construct total order

- Examples (all variables initially 0)
  
  ```
  write data 1      read flag 1
  ↓    ↓    ↓
  write flag 1      read data 0
  write data 1 →  read data 1
  ↓    ↓    ↓
  write flag 1 ← read flag 0
  ```
**Cycle Detection Generalization**

- Generalizes to arbitrary numbers of variables and processors
- Cycles may be arbitrarily long, but it is sufficient to consider only **minimal cycles** with 1 or 2 consecutive stops per processor
- Can simplify the analysis by assuming all processors run a copy of the same code
**Static Analysis for Cycle Detection**

- Approximate P by the control flow graph
- Approximate A by undirected “conflict” edges
  - Bi-directional edge between accesses to the same variable in which at least one is a write
  - It is still correct if the conflict edge set is a superset of the reality

Let the “delay set” D be all edges from P that are part of a minimal cycle
  - The execution order of D edge must be preserved; other P edges may be reordered (modulo usual rules about serial code)

Katherine Yelick, Computer Science Division, EECS, University of California, Berkeley
**Cycle Detection in Practice**

- Cycle detection was implemented in a prototype version of the Split-C and Titanium compilers.
  - Split-C version used many simplifying assumptions.
  - Titanium version had too many conflict edges.

- **What is needed to make it practical?**
  - Finding possibly-concurrent program blocks
    - Use SPMD model rather than threads to simplify
    - Or apply data race detection work for Java threads
  - Compute conflict edges
    - Need good alias analysis
    - Reduce size by separating shared/private variables
  - Synchronization analysis
Synchronization Analysis

- Enrich language with synchronization primitives
  - Lock/Unlock or “synchronized” blocks
  - Post/Wait or Wait/Notify on condition variables
  - Global barriers: all processors wait at barrier

- Compiler can exploit understanding of synchronization primitives to reduce cycles
  - Note: use of language primitives for synchronization may aid in optimization, but “rolling your own” is still correct
Edge Ordering

- Post-Wait operations on the a variable can be ordered

- Although correct to treat these as shared memory accesses, we can get leverage by ordering them

- Then turn edges
  - ? → post c into delay edges
  - wait c → ? into delay edges

- And oriented corresponding conflict edges
**Edge Deletion**

- In SPMD programs, the most common form of synchronization is global barrier

- If we add to the delay set edges of the form
  - ? → barrier
  - barrier → ?

Then we can remove corresponding conflict edges
Synchronization in Cycle Detection

• Iterative algorithm
  • Compute delay set restrictions in which at least one operation is a synchronization operation
  • Perform edge orientation and deletion
  • Compute delay set on remaining conflict edges

• Two important details
  • For locks (and synchronized) we need good alias information about the lock variables. (Conservative would probably work…)
  • For barriers, need to line up corresponding barriers
Static Analysis for Barriers

• Lining up barriers is needed for cycle detection.
• Mis-aligned barriers are also a source of bugs inside branches or loops.
• Includes other global communication primitives
  barrier, broadcast, reductions
• Titanium uses barrier analysis, based on the idea of single variables and methods:
  • A “single” method is one called by all procs
    public single static void allStep(...)  
  • A “single” variable has same value on all procs
    int single timestep = 0;
**Single Analysis**

- The underlying requirement is that barriers only match the same textual instance.
- Complication from conditionals:
  ```java
  if (this processor owns some data) {
    compute on it
    barrier
  }
  ```
- Hence the use of “single” variables in Titanium.
- If a conditional or loop block contains a barrier, all processors must execute it:
  - expression in such loops headers, if statements, etc. must contain only single variables.
Single Variable Example in Titanium

• Barriers and single in N-body Simulation
  ```java
class ParticleSim {
  public static void main (String [] argv) {
    int single allTimestep = 0;
    int single allEndTime = 100;
    for (; allTimestep < allEndTime; allTimestep++){
      // read all particles and compute forces on mine
      computeForces(...);
      Ti.barrier();
      // write to my particles using new forces
      spreadParticles(...);
      Ti.barrier();
    }
  }
}
```

• Single methods are automatically inferred, variables not
Discussion

• One level does not inherit the level below.
  • A compiler may, for example, provide the programmer with a memory model that is either *stronger or weaker* than the underlying hardware.

• Any compiler that saves values in a register will break SC, so "just get over it." -- Burton Smith

• On the other hand, the compiler can insert fences to make non-SC hardware present an SC model to the user.

• Should Languages provide SC behavior?
  • For all programs?
  • For all “properly synchronized” programs?
  • (with or without enforcement of proper synchronization)
Ways to Avoid Memory Model Discussions

• (Or how to get SC trivially)
• Use a pure functional language (no assignment statements)
• User a data parallel language (serial semantics)
• Avoid shared data; use message passing
  • (and not the corners of MPI which allow non-blocking messages on data buffers shared by the application and message passing implementation).
Out of Thin Air

• What behaviors are allowed?

Initially, \( x = y = 0 \)

<table>
<thead>
<tr>
<th>Thread 1</th>
<th>Thread 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>( r_1 = x )</td>
<td>( r_2 = y )</td>
</tr>
<tr>
<td>( y = r_1 )</td>
<td>( x = r_2 )</td>
</tr>
</tbody>
</table>

- How about:
  - Thread 1 speculates that \( x \) is 24 and reads into \( r_1 \)
  - Thread 1 writes 24 into \( y \)
  - Thread 2 reads 24 into \( r_2 \)
  - Thread 2 writes 24 into \( x \)
  - This validates Thread 1’s speculation. QED

- The question is not does your hardware do this;
- It’s whether you have to argue about whether spec allows it
How many programmers can understand weak models?

- 10% of all programmers (maybe the efficiency layer can be non-SC)
- 1% of all programmers (maybe enough for lowest level runtimes)
- 2 programmers per commercial compiler team? (enough to insert the fences in code generation)
- 2 people at any given point in time
Some Open Problems

• What is the right semantic model for shared memory parallel languages?

• Is cycle detection practical on real languages?
  • How well can synchronization be analyzed?
  • Aliases between non-synchronizing variables?
  • Can we distinguish between shared and private data?
  • What is the complexity on real applications?

• Analysis in programs with dynamic thread creation
UPC Memory Model

- UPC has two types of memory accesses
  - Relaxed:
    - operation must respect local (on-thread) dependencies
    - other threads may observe these operations happening in different orders
  - Strict:
    - operation must appear atomic
    - all relaxed operations issued earlier must complete before
    - all relaxed operations issued later must happen later

- Several ways to specify the access:
  - strict shared int x; type qualifier
  - #pragma upc_relaxed pragma
  - #include <upc_relaxed.h> include file
Behavioral Approach

- Problems with operational specifications
  - Implicit assumptions about implementation strategy (e.g., caches)
  - May unnecessarily restrict implementations
  - Intuitive in principle, but complicated in practice

- A behavioral approach for UPC
  - Based on partial and total orders
  - Using Sequential Consistency definition as model
    - Processor order defines a total order on each thread
    - Their union defines a partial order
    - There exists a consistent total order that is correct as a serial execution
Some Basic Notation

• The set of operations is
  • $O_t = \text{the set of operations issued by thread } t$

• The set of memory operations is:
  • $M = \{m_0, m_1, \ldots\}$
  • $M_t = \text{the set of memory operations from thread } t$

• Each memory operations has properties
  • $Thread(m_i)$ is the thread that executed the operation
  • $Location(m_i)$ is the memory location involved

• Memory operations are partitioned into 6 sets, given by
  • $S = \text{Strict, } R=\text{Relaxed, } P=\text{Private}$
  • $W=\text{Write, } R=\text{Read (in the 2\textsuperscript{nd} position)}$
  • Some useful groups: $Strict(M) = SW(M) [ SR(M)$
    
    $W(M) = SW(M) [ RW(M) [ PW(M)$
Compiler Assumption

• For specification purposes, assume the code is compiled by a naïve compiler into ISO C machine
  • Real compilers may do optimizations
    • E.g., reorder, remove, insert memory operations
    • Even strict operations may be reordered with sufficient analysis (cycle detection)
  • These must produce an execution whose input/output and volatile behavior is identical to that of an unoptimized program (ISO C)
**Orderings on Strict Operations**

Threads must agree on an ordering of:

$$AllStrict(M) \overset{\text{def}}{=} StrictPairs(M) \cup StrictOnThreads(M)$$

- For pairs of strict accesses, it will be total:

$$StrictPairs(M) \overset{\text{def}}{=} \{(m_1, m_2) \mid m_1 \neq m_2 \land m_1 \in Strict(M) \land m_2 \in Strict(M)\}$$

- For a strict/relaxed pair on the same thread, they will all see the program order:

$$StrictOnThreads(M) \overset{\text{def}}{=} \{(m_1, m_2) \mid m_1 \neq m_2 \land \text{Thread}(m_1) = \text{Thread}(m_2) \land (m_1 \in Strict(M) \lor m_2 \in Strict(M))\}$$
Orderings on Local Operations

- Conflicting accesses have the usual definition

\[
\text{Conflicting}(M) \overset{\text{def}}{=} \{(m_1, m_2) \mid \\
\text{Location}(m_1) = \text{Location}(m_2) \\
\land (m_1 \in W(M) \lor m_2 \in W(M))\}\]

- Given a serial execution \(S = [o_1, \ldots, o_n]\) defining \(\prec_S\) let \(S_t\) be the subsequence of operations issued by \(t\)

- \(S\) conforms to program order for thread \(t\) iff:
  - \(S_t\) is consistent with the program text for \(t\) (follows control flow)

- \(S\) conforms to program dependence order for \(t\) iff there exists a permutation \(P(S)\) such that:
  - \(P(S)\) conforms to program order for \(t\)
  - forall \((m_1, m_2)\) in \(\text{Conflicting}(M)\), \(m_1 \prec_S m_2\) iff \(m_1 \prec_{P(S)} m_2\)
**UPC Consistency**

An execution on T threads with memory ops M is **UPC consistent** iff:

- There exists a partial order $<_\text{strict}$ that orients all pairs in $\text{allStrict}(M)$.
- And for each thread $t$, there exists a total order $<_t$ on $O_t$ such that:
  - $<_t$ is consistent with $<_\text{strict}$: All threads agree on ordering of strict operations.
  - $<_t$ conforms to program dependence order: Local dependencies are observed.
  - $<_t$ is a correct execution: Reads return most recent write values.
Intuition on Strict Orderings

- Each thread may “build” its own total order to explain behavior
- They all agree on the strict ordering shown above in black, but
  - Different threads may see relaxed writes in different orders
    - Allows non-blocking writes to be used in implementations
  - Each thread sees own dependencies, but not those of other threads
    - Weak, but otherwise there would be consistency requirements on some relaxed operations
    - Preserving dependencies requires usual compiler/hw analysis
**Synchronization Operations**

- UPC has both global and pairwise synchronization.
- In addition to the synchronization properties, they also have memory model implications:
  - **Locks**
    - `upc_lock` is a strict read
    - `upc_unlock` is a strict write
  - **Barriers (which may be split-phase)**
    - `upc_notify` (begin barrier) is a strict write
    - `upc_wait` (end of barrier) is a strict read
    - `upc_barrier = upc_notify; upc_wait`
  - (More technical details in definitions as to the variable being read/written)
Properties of UPC Consistency

• A program containing only strict operations is sequentially consistent
• A program that produces only race-free executions is sequentially consistent
  • A UPC consistent execution of a program is race-free if for all threads \( t \) and all enabling orderings \(<_t\)
  • For all potential races:
    \[
    (m_1, m_2) \text{Thread}(m_1) \neq \text{Thread}(m_2) \\
    \land \text{Location}(m_1) = \text{Location}(m_2) \\
    \land (m_1 \in W(M) \lor m_2 \in W(M))
    \]
  • If \( m_1 <_t m_2 \) then 9 synchronization operations \( o_1, o_2 \) such that \( m_1 <_t o_1 <_t o_2 <_t m_2 \) and \( \text{Thread}(o_1) = \text{Thread}(m_1) \) and \( \text{Thread}(o_2) = \text{Thread}(m_2) \) and either
    • \( o_1 \) is upc_notify and \( o_2 \) is upc_wait or
    • \( o_1 \) is upc_unlock and \( o_2 \) is upc_lock on the same lock variable

Katherine Yelick, Computer Science Division, EECS, University of California, Berkeley
Alternative Models

- As specified, two relaxed writes to the same location may be viewed differently by different processors
  - Nothing to force eventual consistency (likely in implementations)
  - May add this to barrier points, at least
  - So far it looks ad hoc

- Adding directionality to reads/writes seems reasonable
  - Strict reads “fence” things that follows
  - Strict writes “fence” things that proceed
  - Simple replace for `StrictOnThreads` definition

- Support user-defined synchronization primitive built from strict operations
Future Plans

• Show that various implementations satisfy this spec
  • Use of non-blocking writes for relaxed writes with write fence/synch at strict points
  • Compiler-inserted prefetching of relaxed reads
  • Compiler-inserted “message vectorization” to aggregate a set of small operations into one larger one
  • A software caching implementation with cache flushes at strict points

• Develop an operational model and show equivalence (or at least that it implements the spec)

• Define the data unit of atomicity
  • Fundamental unit of interleaving, Data tearing, Conflicts
Conclusions

• Behavioral specifications
  • Are relatively concise
  • Not intended for most end-users: they would see “properties” part
  • Avoids reference to implementation-specific notions, and is likely to constrain implementations less than operational specs

• UPC
  • Has user-control specification model at the language level
  • Language model need not match that of the underlying machine
    • It may be stronger (by inserting fences)
    • It may be weaker (by reordering operations at compile-time)
  • Seems to be acceptable within high end programming community (also evidence in the MPI-2 spec)