Memory Consistency and Multiprocessor Performance

Memory Consistency Model

Define memory correctness for parallel execution

- Execution appears to the that of some correct execution of some theoretical parallel computer which has $n$ sequential processors
- Particularly, remote writes must appear in a local processor in some correct sequence

Sequential Consistency Model

Sequential consistency

- Memory read/writes are globally serialized; assume every cycle only one processor can proceed for one step, and write result appears on other processors immediately
- Processors do not reorder local reads and writes

Memory Consistency Examples

Initially $A = 0$ and $B = 0$

$$
P_1: \begin{align*}
A &= 1; \\
L1: &\text{ if (B == 0) ...}
\end{align*}
$$

$$
P_2: \begin{align*}
B &= 1; \\
L2: &\text{ if (A == 0) ...}
\end{align*}
$$

Impossible for both if statements $L1$ & $L2$ to be true?

- Note the number of possible total orderings is an exponential function of $\#inst$

Performance of Sequential Consistency

- SC must delay all memory accesses until all invalidates done
- What if write invalidate is delayed & processor continues?
- How about memory level parallelism?

Relaxed Memory Consistency

- Total storing order
  - Only writes are globally serialized; assume every cycle at most one write can proceed, and the write result appears immediately
  - Processors may reorder local reads/writes without RAW dependence
- Processor consistency
  - Writes from one processor appear in the same order on all other processors
  - Processors may reorder local reads/writes without RAW dependence
Relaxed Memory Consistency

An architect has designed an SMP that uses processor consistency.

P1 P2 P3
A = get_data(); if (updated) if (flag) updated = 1; flag = 1; process_data(A);

An programmer found that P3 uses stale data of A, sometimes. Initially updated==0 and flag==0.

Can you defend the architect?

Memory Consistency and ILP

Speculate on loads, recovered on violation
- With ILP and SC what will happen on this?

P1 code P2 code P1 exec P2 exec
A = 1 B = 1 spec. load B spec. load A
print B print A store A store B
retine load B shuffle at load A

SC can be maintained, but expensive, so use TSO or PC
- Speculative execution and rollback can still improve performance
- Performance: ILP + Strong MC = Weak MC

Memory Consistency in Real Programs

Not really an issue for many common programs; they are synchronized
- A program is synchronized if all access to shared data are ordered by synchronization operations
  write (x)
  ... release (s) (unlock)
  ... acquire (s) (lock)
  ... read(x)

Synchronization

Why Synchronize? Need to know when it is safe for different processes to use shared data

Issues for Synchronization:
- Uninterruptable instruction to fetch and update memory (atomic operation);
- Synchronization operation needs this kind of primitive;
- For large scale MPs, synchronization can be a bottleneck; techniques to reduce contention and latency of synchronization

Uninterruptable Instructions to Fetch and Update Memory

- Atomic exchange: interchange a value in a register for a value in memory
- Test-and-set: tests a value and sets it if the value passes the test
- Fetch-and-increment: it returns the value of a memory location and atomically increments it

Parallel App: Commercial Workload

Online transaction processing workload (OLTP) (like TPC-B or -C)
Decision support system (DSS) (like TPC-D)
Web index search (Altavista)

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>% Time User Made</th>
<th>% Time Kernel</th>
<th>% Time I/O (CPU Idle)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OLTP</td>
<td>71%</td>
<td>18%</td>
<td>11%</td>
</tr>
<tr>
<td>DSS (range)</td>
<td>82-94%</td>
<td>3-5%</td>
<td>4-13%</td>
</tr>
<tr>
<td>DSS (avg)</td>
<td>87%</td>
<td>4%</td>
<td>9%</td>
</tr>
<tr>
<td>Altavista</td>
<td>&gt; 98%</td>
<td>&lt; 1%</td>
<td>&lt; 1%</td>
</tr>
</tbody>
</table>
**Alpha 4100 SMP**
- 4 CPUs
- 300 MHz Apha 211264 @ 300 MHz
- L1\$ 8KB direct mapped, write through
- L2\$ 96KB, 3-way set associative
- L3\$ 2MB (off chip), direct mapped
- Memory latency 80 clock cycles
- Cache to cache 125 clock cycles

**OLTP Performance as vary L3\$ size**

**L3 Miss Breakdown**

**Memory CPI as increase CPUs**

**OLTP Performance as vary L3\$ block size**

**SGI Origin 2000**
- A pure NUMA
- 2 CPUs per node.
- Scales up to 2048 processors
- Design for scientific computation vs. commercial processing
- Scalable bandwidth is crucial to Origin
Parallel App: Scientific/Technical

FFT Kernel: 1D complex number FFT
- 2 matrix transpose phases => all-to-all communication
- Sequential time for n data points: O(n log n)
- Example is 1 million point data set

LU Kernel: dense matrix factorization
- Blocking helps cache miss rate, 16x16
- Sequential time for nxn matrix: O(n²)
- Example is 512 x 512 matrix

LU kernel

Parallel App: Scientific/Technical

Barnes App: Barnes-Hut n-body algorithm solving a problem in galaxy evolution
- n-body alg rely on forces drop off with distance; if far enough away, can ignore (e.g., gravity is 1/d²)
- Sequential time for n data points: O(n log n)
- Example is 16,384 bodies

Ocean App: Gauss-Seidel multigrid technique to solve a set of elliptical partial differential eq's
- red-black Gauss-Seidel colors points in grid to consistently update points based on previous values of adjacent neighbors
- Multigrid solve finite diff. eq. by iteration using hierarch. Grid
- Communication when boundary accessed by adjacent subgrid
- Sequential time for nxn grid: O(n²)
- Input: 130 x 130 grid points, 5 iterations
Multiprocessor Conclusion

• Some optimism about future
  • Parallel processing beginning to be understood in some domains
  • More performance than that achieved with a single-chip microprocessor
  • MPs are highly effective for multiprogrammed workloads
  • MPs proved effective for intensive commercial workloads, such as OLTP (assuming enough I/O to be CPU-limited), DSS applications (where query optimization is critical), and large-scale, web searching applications
  • On-chip MPs appears to be growing
    1) embedded market where natural parallelism often exists an obvious alternative to faster less silicon efficient, CPU.
    2) diminishing returns in high-end microprocessor encourage designers to pursue on-chip multiprocessing