Announcements

- We cannot meet on December 12th
  - Classes run according to a Thursday schedule
- All assignments/project reports must be turned in by the last day of the semester (December 21)
  - Note that I will not be available after December 13th

Outline

- Last lecture
  - Optimizing shared memory programs
    - Improved synchronization primitives
  - Software support for shared memory
    - Page based shared memory
      - Ivy, LRC (TreadMarks), HLRC
- This lecture: Improving performance of software shared memory
  - Munin: Customizing protocols to application patterns
  - Shasta: Maintaining coherence at smaller granularity
  - CRL: Region-based shared memory

(Review) Page-Based Shared Memory

- LRC + multiple writer protocols improve performance dramatically
- However, still a wide gap as compared to hardware shared memory for applications with fine-grained sharing
  - false sharing
  - extra communication and processing overhead
  - page faults and fetches are expensive to satisfy
  - synchronization through software messages: dilates critical sections
  - scalability problems because of auxiliary data structures
- Research challenge
  - can one ever match performance of hardware shared memory using software-only approaches?
  - Key: software approaches can take advantage of protocol flexibility
    - customize protocols to application behavior
    - more processing but reduced communication (favored by architectural trends)
Improving Software Shared Memory Performance (1): The Munin Approach

- **Rationale**: All pages in a page-based system use the same protocol
  - There might be a benefit in using a different protocol/page
    - Determined by application usage patterns
  - Why is this hard(er) to do in hardware cache-coherent machines?

- **Munin** (Carter and Zwaenepoel, Rice University, 1992)
  - **Idea**: Annotate shared variables with usage pattern
  - The DSM system uses these annotations to maintain coherence
    - Implement a generalized coherence protocol whose behavior is customized by various control parameters
    - Choose different values of these parameters depending on the usage pattern involving a particular shared variable
  - Munin was the first software DSM to use release consistency

**Munin Implementation**

- Low-level parameters modify a baseline directory-based write invalidation protocol
  - (I) Invalidate or update? Choose between invalidation/update messages
  - (R) Replicas allowed? Disallow more than one copy
  - (D) Delayed operations allowed? Use a delayed update queue
  - (FO) Fixed owner? Don’t propagate ownership of object
  - (M) Multiple writers? Use twinning protocol
  - (S) Stable sharing pattern? Eagerly send updates
  - (W) Writable?

<table>
<thead>
<tr>
<th>Annotation</th>
<th>I</th>
<th>R</th>
<th>D</th>
<th>FO</th>
<th>M</th>
<th>S</th>
<th>H</th>
<th>W</th>
</tr>
</thead>
<tbody>
<tr>
<td>Read-only</td>
<td>N</td>
<td>Y</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>N</td>
</tr>
<tr>
<td>Migratory</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Write-shared</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Producer-Consumer</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Reduction</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>Y</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>Y</td>
</tr>
<tr>
<td>Result</td>
<td>N</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>-</td>
<td>Y</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Conventional</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>Y</td>
</tr>
</tbody>
</table>

**Munin (cont’d)**

Examples of using Munin protocols (coherence granularity is a page)

- **Matrix multiply**
  - A and B matrices are tagged as read-only
    - Avoids having to maintain replica information: not a big deal
  - C matrix is tagged as result
    - Multiple writers are allowed
    - Upon end of the computation, writes folded back into the “root” copy
      - All replicas are invalidated

- **SOR (Jacobi)**
  - Each grid block is tagged as producer-consumer
    - Big savings: Single network transaction propagates updates to neighbors
  - Write propagation is delayed until the end of the iteration (release consistency)
  - First iteration establishes the stable sharing relationship
  - In subsequent iterations
    - Data is propagated eagerly at the end of the iteration (barrier) as one message

Improving Software Shared Memory Performance (2): The Shasta Approach

- **Rationale**: Page-based shared memory systems subject to false sharing
  - Because of maintaining coherence at page granularity
    - Required, in order to benefit from zero-cost access control
  - Can we tradeoff access-control costs vs. support for smaller granularities?
  - How small can we make these costs?

- **Shasta** (Scales and Ghoracharloo, Digital WRL, 1996)
  - Compiler inserts code before each load/store, which verifies that the memory location has and continues to have the desired access rights
  - Big contribution was several optimizations to reduce the resulting costs
    - Memory layout
      - Aggregation of checks
      - Exploit superscalar processors
    - Were able to support line sizes as small as **64 bytes**
Shasta: Access Control through Code Instrumentation

- Compiler inserts code before each load/store which verifies that the memory location has and continues to have the desired access rights

```
line += 3;
v = v - line;
```

Compile

```
load r1, ptr[line]
load r2, ptr[v]
add r1, 3h
store r1, ptr[line]
sub r2, r1
store r2, ptr[v]
```

Code Instr.

```
push ptr[line]
call __check_w
load r1, ptr[line]
push ptr[v]
call __check_w
load r2, ptr[v]
add r1, 3h
store r1, ptr[line]
sub r2, r1
push ptr[line]
call __check_r
sub r2, r1
push ptr[line]
call __done
store r2, ptr[v]
push ptr[v]
call __done
```

Opt.

Shasta Optimizations

- Check only loads and stores that refer to “shared” data
  - No checks for stack and static data (identified by base registers)
- Poll for messages
  - No need to ensure atomicity of check and operation
- Optimized check sequences
  1. lda rx, offset[base]
  2. srl rx, 39, ry
  3. beq ry, nomiss
  4. srl rx, 6, rx
  5. ldq u ry, 0[rx]
  6. extb li ry, rx, ry
  7. beq ry, nomiss
  8. ... call function to handle store miss
  9. nomiss:
  10. ... original store instruction ...
- Schedule the instructions to fill any idle slots (long-latency operations)
  - Take advantage of superscalar nature of processors

Shasta Optimizations (cont’d)

- Optimize load checks by storing a “flag” value in the cache line
  - Check just verifies if value is different from flag
- Optimize store checks by first looking up compressed version of state table
  - I: exclusive
  - Advantage: avoid cache misses on state table lookups
- Batch load/store checks whenever they refer to the same register
- Does all this pay off?

Improving Software Shared Memory Performance (3): The CRL Approach

- **Rationale:** Programmers typically associate a different synchronization object (lock) with different data objects (regions)
  - Reductions in coherence traffic possible by
    - Delaying update propagation until someone acquires the lock
    - Only propagating information about the “region” protected by the lock
- **Entry Consistency**
  - Originally proposed in the context of a DSM system called Midway
  - Weaker than release consistency
    - Propagates updates to all pages
    - In this case
      - LD/ST within a synchronization block can be reordered
      - Only need to wait for acquire/release operations on same synchronization variable
  - Naturally suited for use in high-level shared object languages (Java)
    - Method entry/exit correspond to lock acquire/release

<table>
<thead>
<tr>
<th>Miss Checks</th>
<th>No Batching</th>
<th>Batching</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Integer Load</td>
<td>FP Load</td>
</tr>
<tr>
<td></td>
<td>Integer/FP Store</td>
<td>Integer/FP Op</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Instructions</th>
<th>2</th>
<th>3</th>
<th>6</th>
<th>7</th>
<th>6</th>
<th>10</th>
<th>12</th>
<th>1</th>
<th>1</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycles (21064A)</td>
<td>4</td>
<td>4</td>
<td>10</td>
<td>11</td>
<td>9</td>
<td>11</td>
<td>13</td>
<td>3</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>Cycles (21164)</td>
<td>2</td>
<td>3</td>
<td>2</td>
<td>6</td>
<td>4</td>
<td>7</td>
<td>8</td>
<td>2</td>
<td>1</td>
<td>4</td>
</tr>
</tbody>
</table>

- Application run-times increased by 5% - 40% (SPLASH-2 benchmarks)
CRL: The C Region Library

Very simple API (Johnson et al, MIT, 1995)

- Allocation
  
  ```c
  rgn_id rgn_create( int size );
  void rgn_delete( rgn_id id );
  ```

- Renaming
  
  ```c
  void * rgn_map( rgn_id id );
  void rgn_unmap( void *addr );
  ```

- Access
  
  ```c
  void rgn_start_read( void *addr );
  void rgn_end_read( void *addr );
  void rgn_start_write( void *addr );
  void rgn_end_write( void *addr );
  ```

- More responsibility on the programmer
  - Manage region IDs
  - Handle explicit renaming into own address space
  - Explicitly identify usage mode (read/write)

- Assumes “disciplined use”
  - Very easy to deadlock: nested access to regions

More suitable for use by a compiler/translator

CRL (cont’d)

- The flip side: API enables efficient all-software implementation
- Simple state machines at the home and “cache” ends
  - Home: HomeExclusive(-,Rip,Wip), HomeShared(-,Rip), HomeIip, HomeInvalid
  - Cache: RemoteInvalid(-,Req), RemoteShared(-,Rip,Req), RemoteModified(-,Rip,Wip)
- 4 call events: start_read, end_read, start_write, end_write
- 17 message types

- CRL-like DSMs have been implemented on the CM-5, T3D/T3E, network of workstations, …
  - 17 µs to read a clean copy of an object
    - on an SGI Origin 2000, with a 10µs round-trip message time
    - On a Fast Ethernet cluster, likely to be around 150-200 µs
  - 36 µs to write a copy (with one invalidate)
- Good performance without overly complicating the protocols

Improving Software Shared Memory Performance (4):
The View Caching Approach

- Further improve performance by exploiting additional application-level knowledge
  - What portions of an object does the application care about?
  - What are the consistency requirements of these portions?

- Challenge is to come up with a natural interface for exposing this information

Next Lecture

- High-level Parallel/Distributed Programming Models
- Future directions