Shuffling: A Lock Contention Aware Thread Scheduling Technique

Kishore Pusukuri
Multicores are Ubiquitous

- Deliver computing power via parallelism
- Potential for delivering high performance for multithreaded applications

Oracle SPARC M7-8

Mobile phones
Complexity of Achieving High Performance

Application Characteristics
- Degree of Parallelism
- Lock Contention
- Memory Requirements

Operating System Policies
- Thread Scheduling
- Memory Management

Architecture
- Cache Hierarchy
- Cross-chip Interconnect Protocols
Modern Operating Systems

Improve System Utilization and Provide Fairness

- Thread Scheduling: Time Share → Fairness
- Memory Allocation: Next → Data Locality

Do not consider relationships between threads of a multithreaded application

Application characteristics should be considered
OS Load Balancing vs Lock Contention

- OS load balancing is oblivious of lock contention
- Performance of multithreaded program with high lock contention is sensitive to the distribution of threads across sockets
- Inappropriate distribution of threads → increases frequency of lock transfers
- Increases lock acquisition latencies
- Increases LLC misses in the critical path
Outline

• Introduction
• Motivation
• Shuffling Framework
• Experimental Results
Lock Contention Study

Lock contention is an important performance limiting factor

23 programs (pthreads)
- SPEC JBB2005
- PARSEC
- SPEC OMP2001
- SPLASH 2x

Run with 64 threads
64-core machine
Four 16-core Sockets (AMD Opteron)
Lock Contention on Performance

**Lock time**: the percentage of elapsed time a process spends on waiting for lock operations in user space
Lock Transfers

Overhead of Lock Transfer:

- **T_low** → Lock transfers between threads located on the same Socket
- **T_high** → Lock transfers between threads located on different Sockets

**e.g.:** bodytrack (BT) with 64 threads

<table>
<thead>
<tr>
<th>Lock Transfer</th>
<th>Solaris</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_low</td>
<td>31%</td>
</tr>
<tr>
<td>T_high</td>
<td>69%</td>
</tr>
</tbody>
</table>
High Frequency of LLC misses & Its Cause

BT with 64 threads

- Lock arrival times spread across a wide interval
- The likelihood of lock acquired by a thread on a different socket is very high

Lock arrival times of threads per socket at the entry of a lock within a 100 ms time interval
Outline

- Introduction
- Motivation
- Shuffling Framework
- Experimental Results
Thread Shuffling [ PACT 2014 ]

Minimize variation in lock arrival times of threads

Schedule threads whose lock arrival times are clustered in a small time interval

Once a thread releases the lock it is highly likely that another thread on the same Socket will successfully acquire the lock
Thread Shuffling (algorithm)

Input: \( N \rightarrow \text{Number of Threads}; \ S \rightarrow \text{Number of Sockets} \)

repeat

1. **Monitor Threads** – sample lock times of \( N \) threads

   if lock times exceed threshold then

2. **Form Thread Groups** – sort threads according to lock times and divide them into \( S \) groups

3. **Perform Shuffling** – shuffle threads to establish newly computed groups

until (application terminates)
Shuffling Interval

Impacts Lock transfers between sockets → LLC misses

500 ms as a shuffling interval

BT: LLC miss rate vs Shuffling interval
Shuffling Overhead Negligible

Frequency of monitoring and shuffling

Overhead is negligible ( < 1% of system time)
Lock Transfers: Solaris vs Shuffling

<table>
<thead>
<tr>
<th>Lock Transfer</th>
<th>Shuffling</th>
<th>Solaris</th>
</tr>
</thead>
<tbody>
<tr>
<td>T_low</td>
<td>46%</td>
<td>31%</td>
</tr>
<tr>
<td>T_high</td>
<td>54%</td>
<td>69%</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>Shuffling</th>
<th>Solaris</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLC miss rate</td>
<td>1.9</td>
<td>3.3</td>
</tr>
<tr>
<td>Lock time</td>
<td>72%</td>
<td>86%</td>
</tr>
</tbody>
</table>
Thread Lock Arrival-time Ranges
Lock contention & LLC miss rate

Reduces Lock contention & LLC misses
Evaluating Thread Shuffling (cont.)

Up to 54%
Avg. 13%

Memcached: 17%
TATP: 28%

Relative to Solaris

DINO: only considers LLC misses
PSets: binding a pool of threads to a pool of cores
Conclusions

Problem:
OS thread scheduling is oblivious to lock contention and fails to maximize performance of multithreaded applications on multicore multiprocessor systems

Idea:
Minimize variation in lock arrival times of threads

Advantages:
- Improves performance on average 13% (max of 54%)
- No need to modify application source code