User-Level Interprocess Communication for Shared Memory Multiprocessors

Brian N. Bershad
Thomas E. Anderson
Edward D. Lazowska
Henry M. Levy

Presented by: Dan Lake
Introduction

- IPC is central to operating system design

Advantages of Decomposed Systems
- Failure Isolation (address space boundaries)
- Extensibility (add new modules)
- Modularity (interfaces enforced)

Kernel traditionally responsible for IPC

Kernel-based IPC has problems
- Architectural performance barriers (LRPC 70%)
- Interaction of kernel-IPC and user-level threads
- Strong interdependencies
- Cost of partitioning these facilities is high
Solution For Shared Memory Multiprocessors

• URPC (User Remote Procedure Calls)

• Separate three components of IPC
  a) Data transfer
  b) Thread management
  c) Processor reallocation

• Goals
  • Move a & b into user-level
  • Limit the kernel to performing only c
  • Eliminate kernel from cross-address space communication
Message Passing

- Logical channels of pair-wise shared memory
- Channels created & mapped once for every client/server pairing
- Channels are bi-directional
- TSL controls access in either direction
- Just as secure as going through the kernel
Data & Security

• Applications access URPC procedures through Stubs layer
• Stubs unmarshal data into procedure parameters
• Stubs copy data in/out, no direct use of shared memory
• Arguments are passed in buffers that are allocated and pair-wise mapped during binding
• Data queues monitored by application level thread management
Thread Management

• LRPC: client threads always cross address-space to server
• URPC: always try to reschedule another thread within address-space
• Switching threads within the same address space requires less overhead than processor reallocation
• Synchronous from programmer pov, but asynchronous to thread mgmt level
Processor Reallocation

• Switching the processor between threads of different address spaces
• Requires privileged kernel mode to access protected mapping registers
• Does include significant overhead
  • As pointed out in the LRPC paper
• URPC strives to avoid processor reallocation
• This avoidance can lead to substantial performance gains
Optimistic Scheduling Policy

- Assumptions
  - Client has other work to do
  - Server will soon have a processor to service a message
Sample Execution Timeline

Optimistic Reallocation Scheduling Policy

pending outgoing messages detected
processor donated

FCMgr “Underpowered”
Why the optimistic approach doesn’t always hold

• This approach does not work as well when the application
  • Runs as a single thread
  • Is Real time
  • Has high latency I/O
  • Priority Invocations

• URPC solves some of these problems by allowing forced processor reallocation even if there is still work to do
Kernel Handles Processor Reallocation

- URPC handles this through call called "Processor.Donate"
- This passes control of an idle processor down to the kernel, and then back up to a specified address in the receiving space
Voluntary Return of Processors

• The policy of a URPC server process:

“…Upon receipt of a processor from a client address, return the processor when all outstanding messages from the client have generated replies, or when the server determines that the client has become ‘underpowered’….”
Parallels to User Threads Paper

- Even though URPC implement a policy/protocol, there is absolutely no way to enforce it. This has the potential to lead to some interesting side effects.

- This is similar to some of the problems discussed in the User Threads paper.
  - For example, a server thread could conceivably continue to hold a donated processor and handle requests from other clients.
What this leads to...

- **Starvation**
  - **URPC** handles this by only directly reallocating processors to load balance.

- The system also needs the notion of preemptive reallocation
  - The Preemptive reallocation must also adhere to
    - No higher priority thread waits while a lower priority thread runs
    - No processor idles when there is work for it to do (even if the work is in another address space)
Performance

Table I. Comparative Performance of Thread Management Operations

<table>
<thead>
<tr>
<th>Test</th>
<th>URPC FastThreads (μsecs)</th>
<th>Taos Threads (μsecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>PROCEDURE CALL</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>FORK</td>
<td>43</td>
<td>1192</td>
</tr>
<tr>
<td>FORK;JOIN</td>
<td>102</td>
<td>1574</td>
</tr>
<tr>
<td>YIELD</td>
<td>37</td>
<td>57</td>
</tr>
<tr>
<td>ACQUIRE,RELEASE</td>
<td>27</td>
<td>27</td>
</tr>
<tr>
<td>PINGPONG</td>
<td>53</td>
<td>271</td>
</tr>
</tbody>
</table>

Table II. Component Breakdown of a URPC

<table>
<thead>
<tr>
<th>Component</th>
<th>Client (μsecs)</th>
<th>Server (μsecs)</th>
</tr>
</thead>
<tbody>
<tr>
<td>send</td>
<td>18</td>
<td>13</td>
</tr>
<tr>
<td>poll</td>
<td>6</td>
<td>6</td>
</tr>
<tr>
<td>receive</td>
<td>10</td>
<td>9</td>
</tr>
<tr>
<td>dispatch</td>
<td>20</td>
<td>25</td>
</tr>
<tr>
<td>Total</td>
<td>54</td>
<td>53</td>
</tr>
</tbody>
</table>

Note: Table II results are independent of load
Performance

- Latency is proportional to the number of threads per cpu’s
- \( T = C = S = 1 \) call latency is 93 microseconds
- \( T = 2, C = 1, S = 1 \), latency increases to 112 microseconds however, throughput raises 75% (benefits of parallelism)
- Call latency effectively reduced to 53 microseconds
- \( C = 1, S = 0 \), worst performance
- In both cases, \( C = 2, S = 2 \) yields best performance
Performance

- Worst case URPC latency for one thread is 375 us
- Similar hardware, LRPC call latency is 157 us
- Reasons:
  - URPC requires two level scheduling
  - URPC 's low level scheduling is done by LRPC
- Small price considering possible gains, this is necessary to have high level scheduling
Conclusions

• Performance gained by moving features out of the kernel not vice-versa
• URPC represents appropriate division for operating system kernels of shared memory multiprocessors
• URPC showcases a design specific to a multiprocessor, not just uniprocessor design that runs on multiprocessor hardware