User-Level Interprocess Communication for Shared Memory Multiprocessors

Bershad, B. N., Anderson, T. E., Lazowska, E.D., and Levy, H. M.

Presented by Akbar Saidov
Introduction

• Interprocess communication (IPC)
  – Central to contemporary OS design
  – Encourages decomposition across address space boundaries. Decomposition advantages:
    • Failure isolation
      – AS boundaries prevent a fault in one module from leaking to another
    • Extensibility
      – New modules can be added to the system without having to modify existing ones
    • Modularity
      – Interfaces are enforced by mechanism rather than by convention
  – In slow cross-address space communication decomposition advantages are traded for better system performance

1. B. N. Bershad et al., p. 176
Problems

• Interprocess Communication has been the responsibility of the kernel
• Two problems with kernel based IPC communication:
  – Architectural performance barriers
    • Performance of kernel-based synchronous communication is limited by the cost of invoking the kernel and reallocating processor to a another address space.
    • In previous work, LRPC’s 70% overhead can be attributed to kernel-mediated cross-address space call.
  – Interaction between kernel-based communication and high-performance user-level threads
    • To obtain satisfactory performance, medium and fine-grained parallel applications must use user-level thread management.
    • In terms of performance and system complexity, the cost for partitioning strongly interdependent communication and thread management across protection boundaries is high
Solution
(on a shared memory multiprocessor)

• Remove kernel from cross-address space communication
  – Use shared memory for data transfer
  – Processor reallocation can be avoided
    • take advantage of already active processor in target AS

• Improved performance, because:
  – Messages are sent between address spaces directly
  – Unnecessary processor reallocation is eliminated
  – Overhead is amortized over several independent calls, when processor reallocation is needed.
  – Parallelism in message passing can be exploited
    • Improves call performance
User-Level Remote Procedure Call (URPC)

- Allows communication between address spaces without kernel intervention
- Use shared memory for data transfer
- Make use of a processor already in address space
- User-level Thread management
- Kernel’s only responsibility is to allocate processors to the address space
URPC

• Synchronization
  – To the programmer, cross-address space procedure call is synchronous
  – At and beneath the thread management level, the call is asynchronous.
    • Client thread T1 invokes a procedure in a server
    • While blocked, another thread T2 can be run in the same AS
    • When the reply arrives, the blocked thread T1 can be rescheduled to any processor assigned to its address space.
  – The scheduling operations can be handled by a user-level thread management system, thus the need to reallocate any processors to a new address space can be avoided, as long as there is a processor assigned to the current AS.
  – Server side: execution of the call can be done by a processor already executing in the context of server’s address space
Example

<table>
<thead>
<tr>
<th>Editor</th>
<th>WinMgr</th>
<th>FCMgr</th>
</tr>
</thead>
<tbody>
<tr>
<td>T1 Call (send/recv WinMgr)</td>
<td>Recv &amp; process reply T1</td>
<td></td>
</tr>
<tr>
<td>Context switch</td>
<td></td>
<td></td>
</tr>
<tr>
<td>T2 Call (send/recv FCMgr)</td>
<td>Recv &amp; process reply T2</td>
<td></td>
</tr>
<tr>
<td>Context switch</td>
<td>Recv &amp; process reply T1</td>
<td>Processor realloc</td>
</tr>
<tr>
<td>T1 Call (send/recv FCMgr)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Processor realloc</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Time</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Context switch – terminate T2

Context switch – terminate T1
URPC Components

• URPC isolates three components of IPC
  – Thread management
    • Block caller thread. Run a thread through the procedure in server’s AS. Resume caller thread on return
  – Data transfer
    • Move arguments between client and server AS
  – Processor reallocation
    • Make sure there is a physical processor to handle client’s call in the server and the server’s reply in the client
URPC Components

Fig. 2 The software components of URPC.
Processor Reallocation

• Context Switching vs. Processor reallocation
  – Significantly less overhead involved in switching a processor to another thread in the same AS (context switching) than reallocating to a thread in a different AS (processor reallocation).

• Processor reallocation costs
  – Scheduling costs
    • Decide the AS
  – Immediate costs
    • Update virtual memory mapping registers
    • Transfer the processor between AS
  – Long-term costs
    • Due to poor cache and TLB performance from constant locality switches.

• Minimal latency same-address space context switch takes approximately 15 microseconds on the C-VAX.
• Cross-address space processor reallocation takes approximately 55 microseconds (without long-term costs).
Processor Reallocation

• Optimistic reallocation policy
  – Assumptions:
    • The Client has other work to do
    • The Server has or will soon have a processor available to service messages

• Policy may not always hold
  – Single-threaded applications
  – Real-time applications (bounded call latency)
  – High-latency I/O operations
  – Priority Invocations

• Solution:
  – URPC allows client AS to force processor reallocation to server AS
Processor Reallocation

• Kernel handles Processor Reallocation
  – Processor.Donate
    • idle processor donates itself to underpowered address space
    • transfers control of an idle processor down through the kernel, and then back up to a specified address in the receiving space

• Voluntary return of processors cannot be guaranteed
  – No way to enforce protocol regarding return of processors.
  – Processor working in server may never return to client. May handle requests of other clients.
  – URPC takes care of load balancing only for communicating applications
  – Preemptive policies, which force processor reallocations from AS to other, are required in order to avoid starvation.
Data Transfer

• Data flows in URPC in different address spaces via a bidirectional shared memory queue. The queue is non-spinning test-and-set locks on either end.
  – Prevent processors from waiting indefinitely on message channels (non-spinning locks)
• Message channels created & mapped once for every client/server pairing
• No kernel copying needed.
Data Transfer

• Security
  – URPC procedures are accessed through Stubs layer
  – Stubs unmarshal data into procedure parameters, and
  – Do the necessary copying and checking to guarantee application’s safety
  – Arguments are passed in buffers and are pair-wise mapped during binding
  – Application level thread management monitors data queues
Thread Management

• Strong interaction between thread management synchronization functions and communications functions
  – Send <-> Receive of Messages
  – Start <-> Stop of Threads

• Classification:
  – Heavyweight
    • For kernel, no distinction between thread and address space
  – Middleweight
    • Address spaces and kernel-managed threads are decoupled
  – Lightweight
    • Threads are managed by user-level libraries
Thread Management

• Arguments
  – Fine-grained parallel programs need high-performance thread management,
  – High-performance thread management only possible with user-level threads,
  – Close interaction between communication and thread management can be exploited to achieve extremely good performance for both (when both are implemented at user level)

• Two-level scheduling
  – Lightweight user-level threads are scheduled on top of weightier kernel-level threads.
  – Communication implemented at kernel level will result in synchronization at both user level and kernel level
## Performance

<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><strong>Total</strong></td>
<td><strong>54</strong></td>
<td><strong>53</strong></td>
</tr>
</tbody>
</table>
Performance

• Call Latency and Throughput
  – Call Latency
    • the time from which a thread calls into the stub until control returns from the stub

• Both latency and throughput are load dependent
  – Depend on
    • C = Number of Client Processors
    • S = Number of Server Processors
    • T = Number of runnable threads in the client’s AS
Performance

- **Call Latency**
  - Latency increases when $T > C + S$
  - Latency is proportional to the number of threads per CPU
  - $T = C = S = 1$ call latency is 93 microseconds
Performance

• **Throughput**
  - Improves until \( T > C+S \)
  - Worst case URPC latency for one \( T=1, C=1, S = 0 \) is 375 microseconds (2 processor reallocations and 2 kernel invocations)
  - Similar setup, LRPC call latency is 157 microseconds
  - Reasons:
    - URPC requires two level scheduling
    - URPC ‘s low level scheduling is done by LRPC
Conclusion

• Motivation, design, implementation, and performance of URPC
• Approach, which addressed problems of kernel-based communication, by moving traditional OS functionality out of kernel and up to user level
• URPC represents appropriate division for OS kernels of shared memory multiprocessors
• Further work in the field
  – Scheduler Activations - present better abstraction for kernel support of user-level threads.