“To infinity and beyond!”

David Crandall
CS 614
September 26, 2006
Motivation

- Communication overheads are high!
  - e.g. results from last week’s RPC paper

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Minimum</th>
<th>Median</th>
<th>Transmission</th>
<th>Local-only</th>
</tr>
</thead>
<tbody>
<tr>
<td>no args/results</td>
<td>1059</td>
<td>1097</td>
<td>131</td>
<td>9</td>
</tr>
<tr>
<td>1 arg/result</td>
<td>1070</td>
<td>1105</td>
<td>142</td>
<td>10</td>
</tr>
<tr>
<td>2 args/results</td>
<td>1077</td>
<td>1127</td>
<td>152</td>
<td>11</td>
</tr>
<tr>
<td>4 args/results</td>
<td>1115</td>
<td>1171</td>
<td>174</td>
<td>12</td>
</tr>
<tr>
<td>10 args/results</td>
<td>1222</td>
<td>1278</td>
<td>239</td>
<td>17</td>
</tr>
<tr>
<td>1 word array</td>
<td>1069</td>
<td>1111</td>
<td>131</td>
<td>10</td>
</tr>
<tr>
<td>4 word array</td>
<td>1106</td>
<td>1153</td>
<td>174</td>
<td>13</td>
</tr>
<tr>
<td>10 word array</td>
<td>1214</td>
<td>1250</td>
<td>239</td>
<td>16</td>
</tr>
<tr>
<td>40 word array</td>
<td>1643</td>
<td>1695</td>
<td>566</td>
<td>51</td>
</tr>
<tr>
<td>100 word array</td>
<td>2915</td>
<td>2926</td>
<td>1219</td>
<td>98</td>
</tr>
<tr>
<td>resume except’n</td>
<td>2555</td>
<td>2637</td>
<td>284</td>
<td>134</td>
</tr>
<tr>
<td>unwind except’n</td>
<td>3374</td>
<td>3467</td>
<td>284</td>
<td>196</td>
</tr>
</tbody>
</table>

From [Birrell84]
Motivation

- Communication overheads are high!
  - e.g. results from last week’s RPC paper

<table>
<thead>
<tr>
<th>Procedure</th>
<th>Minimum</th>
<th>Median</th>
<th>Transmission</th>
<th>Local-only</th>
</tr>
</thead>
<tbody>
<tr>
<td>no args/results</td>
<td>1059</td>
<td>1097</td>
<td>131</td>
<td>9</td>
</tr>
<tr>
<td>1 arg/result</td>
<td>1070</td>
<td>1105</td>
<td>142</td>
<td>10</td>
</tr>
<tr>
<td>2 args/results</td>
<td>1077</td>
<td>1127</td>
<td>152</td>
<td>11</td>
</tr>
<tr>
<td>4 args/results</td>
<td>1115</td>
<td>1171</td>
<td>174</td>
<td>12</td>
</tr>
<tr>
<td>10 args/results</td>
<td>1222</td>
<td>1278</td>
<td>239</td>
<td>17</td>
</tr>
<tr>
<td>1 word array</td>
<td>1069</td>
<td>1111</td>
<td>131</td>
<td>10</td>
</tr>
<tr>
<td>4 word array</td>
<td>1106</td>
<td>1153</td>
<td>174</td>
<td>13</td>
</tr>
<tr>
<td>10 word array</td>
<td>1214</td>
<td>1250</td>
<td>239</td>
<td>16</td>
</tr>
<tr>
<td>40 word array</td>
<td>1643</td>
<td>1695</td>
<td>566</td>
<td>51</td>
</tr>
<tr>
<td>100 word array</td>
<td>2915</td>
<td>2926</td>
<td>1219</td>
<td>98</td>
</tr>
<tr>
<td>resume except'n</td>
<td>2555</td>
<td>2637</td>
<td>284</td>
<td>134</td>
</tr>
<tr>
<td>unwind except'n</td>
<td>3374</td>
<td>3467</td>
<td>284</td>
<td>196</td>
</tr>
</tbody>
</table>

From [Birrell84]
Sources of overhead

- Memory copies
  - User buffer → kernel buffer → protocol stack → NIC
- System call
- Scheduling delays
- Interrupts/polling overhead
- Protocol overhead (headers, checksums, etc.)
- Generality of networking code
  - Even though most applications do not need all features
How to reduce overhead?

- U-Net, von Eicken et al, 1995
  - Move networking out of the kernel

- Lightweight RPC, Bershad et al, 1990
  - Optimize for the common case: same-machine RPC calls
U-Net: A User-Level Network Interface for Parallel and Distributed Computing

T. von Eicken, A. Basu, V. Buch, W. Vogels
Cornell University
SIGOPS 1995
U-Net goals

- Low-latency communication
- High bandwidth, even with small messages
- Use off-the-shelf hardware, networks
  - Show that Network of Workstations (NOW) can compete with Massively Parallel Processor (MPP) systems
U-Net strategy

- Remove (most) networking code from the kernel
  - Reduces overhead from copies, context switches
  - Protocol stack implemented in user space

- Each application gets a virtualized view of the network interface hardware
  - System multiplexes the hardware, so that separation and protection are still enforced
  - Similar to the exokernel philosophy [Engler95]
U-Net architecture compared

Traditional architecture

- Kernel (K) on critical path (sends and receives)
- Requires memory copies, mode switches between kernel (K) and apps (U)

U-net's architecture

- Kernel (K) removed from critical path (only called on connection setup)
- Simple multiplexer (M) implemented in firmware on NIC

From [von Eicken95]
**U-Net endpoints**

- Application sees network as an *endpoint* containing communication buffers and queues
  - Endpoints pinned in physical memory, DMA-accessible to NIC and mapped into application address space
  - (or emulated by kernel)

From [von Eicken95]
Incoming messages

- U-Net sends incoming messages to endpoints based on a destination channel tag in message
  - Channel tags in messages identify source and destination endpoints, to allow multiplexer to route messages appropriately

- U-Net supports several receive models
  - Block until next message arrives
  - Event-driven: signals, interrupt handler, etc.
  - Polling
    - Polling is fastest for small messages: round-trip latency half that of UNIX signal (60 µsec vs. 120 µsec)

- To amortize notification cost, all messages in receive queue are processed
Endpoints + Channels = Protection

- A process can only “see” its own endpoint
  - Communications segments, messages queues are disjoint, mapped only into creating process’s address space

- A sender can’t pose as another sender
  - U-Net tags outgoing messages with sending endpoint

- Process receives only its own packets
  - Incoming messages de-multiplexed by U-Net

- Kernel assigns tags at connection start-up
  - Checks authorization to use network resources
Kernel-emulated endpoints

- NIC-addressable memory might be scarce, so kernel can emulate endpoints, at additional cost

From [von Eicken95]
U-Net implementation

- Implemented U-Net in firmware of Fore SBA-200 NIC
  - Used combination of pinned physical memory and NIC’s onboard memory to store endpoints

- Base-level vs. direct-access
  - Zero-copy vs. *true* zero-copy: is a copy between application memory and communications segment necessary?
  - Direct access not possible with this hardware. Requires NIC to be able to map all physical memory, and page faults must be handled.
Microbenchmarks

- U-Net saturates fiber with messages >1024 bytes

From [von Eicken95]
TCP, UDP on U-Net

- U-net implementations of UDP and TCP outperform traditional SunOS implementations:

From [von Eicken95]
Application benchmarks

- Split-C parallel programs
- Compare U-Net cluster of Sun workstations to MPP supercomputers

Performance is similar
  - But prices are not!
  - (very) approximate price per node: CM-5: $50,000, NOW: $15,000, CS-2: $80,000

From [von Eicken95]
U-Net: Conclusions

- Showed that NOW could compete with MPP systems
  - Spelled the end for many MPP companies:
    - Thinking Machines: bankrupt, 1995
    - Cray Computer Corporation: bankrupt, 1995
    - Kendall Square Research: bankrupt, 1995
    - Meiko: collapsed and bought out, 1996
    - MasPar: changed name, left MPP business, 1996

- U-Net influenced VIA (Virtual Interface Architecture) standard for user-level network access
  - Intel, Microsoft, Compaq, 1998
Lightweight Remote Procedure Call

B. Bershad, T. Anderson, E. Lazowska, H. Levy
University of Washington
ACM TOCS, 1990
“Forget network overhead!”

- Most (95-99%) RPC calls are to local callees
  - i.e. same machine but different protection domain
  - (presumably not true for all systems, applications)

- Existing RPC packages treat these calls the same as “real” remote calls
  - Local RPC call takes 3.5x longer than ideal

- Lightweight RPC optimizes this common case
Traditional RPC overhead

- Costly! …stubs, message transfers, 2 thread dispatches, 2 context switches, 4 copies
Lightweight Remote Procedure Calls

- Goal: Improve performance, but keep safety

- Optimized for local RPC case
  - Handles “real” remote RPC calls using “real” RPC mechanism
Optimizing parameter passing

- Caller and server share argument stacks
  - Eliminates packing/unpacking and message copies
  - Still safe: a-stacks allocated as pairwise shared memory, visible only to client and server
    - But asynchronous updates of a-stack are possible
  - Call-by-reference arguments copied to a-stack (or to a separate shared memory area if too large)

- Much simpler client and server stubs
  - Written in assembly language
Optimizing domain crossings

- RPC gives programmer illusion of a single abstract thread “migrating” to server, then returning
  - But really there are 2 concrete threads; caller thread waits, server thread runs, then caller resumes

- In LRPC, caller & server run in same concrete thread
  - Direct context switch; no scheduling is needed
  - Server code gets its own execution stack (e-stack) to ensure safety
When an LRPC call occurs…

- **Stub:**
  - pushes arguments onto a-stack
  - puts procedure identifier, binding object in registers
  - traps to kernel

- **Kernel:**
  - Verifies procedure identifier, binding object, a-stack
  - Records caller’s return address in a linkage record
  - Finds an e-stack in the server’s domain
  - Points the thread’s stack pointer to the e-stack
  - Loads processor’s virtual memory registers with those of the server domain [requires TLB flush]
  - Calls the server’s stub for the registered procedure

From [Bershad90]
LRPC Protection

- Even though server executes in client’s thread, LRPC offers same level of protection as RPC
  - Client can’t forge binding object
  - Only server & client can access a-stack
  - Kernel validates a-stack
  - Client and server have private execution stacks
  - Client and server cannot see each other’s memory (Kernel switches VM registers on call and return)
  - Linkage record (caller return address) kept in Kernel space
Other details

- A-stacks allocated at bind time
  - Size and number based on size of procedure call argument list and number of simultaneous calls allowed

- Careful e-stack management

- Optimization with multiprocessor systems
  - Keep caller, server contexts loaded on different processors. Migrate thread between CPUs to avoid TLB misses, etc.

- Need to handle client or server termination that occurs during an LRPC call
LRPC performance

- ~3x speed improvement over Taos (DEC Firefly OS)

\[
\begin{array}{|c|c|c|c|}
\hline
\text{Test} & \text{Description} & \text{LRPC/MP} & \text{LRPC} & \text{Taos} \\
\hline
\text{Null} & \text{The Null cross-domain call} & 125 & 157 & 464 \\
\text{Add} & \text{A procedure taking two 4-byte arguments and returning one 4-byte argument} & 130 & 164 & 480 \\
\text{BigIn} & \text{A procedure taking one 200-byte argument} & 173 & 192 & 539 \\
\text{BigInOut} & \text{A procedure taking and returning one 200-byte argument} & 219 & 227 & 636 \\
\hline
\end{array}
\]

Times in μsec

- ~25% of remaining overhead due to TLB misses after context switches
- (Caveat: Firefly doesn’t support pairwise shared memory; implementation uses global shared memory, so less safety)

From [Bershad90]
LRPC performance

- ~3x speed improvement over Taos (DEC Firefly OS)

<table>
<thead>
<tr>
<th>Test</th>
<th>Description</th>
<th>LRPC/MP</th>
<th>LRPC</th>
<th>Taos</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null</td>
<td>The Null cross-domain call</td>
<td>125</td>
<td>157</td>
<td>464</td>
</tr>
<tr>
<td>Add</td>
<td>A procedure taking two 4-byte arguments and returning one 4-byte argument</td>
<td>130</td>
<td>164</td>
<td>480</td>
</tr>
<tr>
<td>BigIn</td>
<td>A procedure taking one 200-byte argument</td>
<td>173</td>
<td>192</td>
<td>539</td>
</tr>
<tr>
<td>BigInOut</td>
<td>A procedure taking and returning one 200-byte argument</td>
<td>219</td>
<td>227</td>
<td>636</td>
</tr>
</tbody>
</table>

Times in µsec

From [Bershad90]

- ~25% of remaining overhead due to TLB misses after context switches
- (Caveat: Firefly doesn’t support pairwise shared memory; implementation uses global shared memory, so less safety)
LRPC performance on multiprocessors

- Scales well on multiprocessors

- Poor performance of RPC due to global lock
Lightweight RPC: Conclusions

- Optimize the common cases: Local RPC calls

- ~3x speed-up over conventional RPC mechanism
  - Impact on speed of apps and overall system?
  - Is MP optimization useful in practice? (how often are idle CPUs available?)
  - Additional bind-time overhead (allocating shared a-stacks)?