Do Microkernels Suck?

Gernot Heiser
UNSW, NICTA and Open Kernel Labs
• Talk by Christoph Lameter: “Extreme High Performance Computing or Why Microkernels Suck”

• Contents:
  – This is how we got Linux to scale to 1000's of CPUs
    ‣ clearly knows what he's talking about
    ‣ no need to add to this...
  – This is why microkernels can't do the same
    ‣ clearly hasn't got a clue about microkernels

• I'll explain...
Summary of Paper

• Look, we've scaled Linux to 1000 processors [with a little help of billions of $$ from IBM, HP, SGI, ...], microkernels [developed mostly by cash-strapped universities] haven't done the same, obviously they suck

• Equivalent statement in 1998: Look, Windows has drivers for zillions of devices, Linux doesn't, hence Linux sux.

• Very scientific approach, right?

• OK, I'm exaggerating somewhat, but let's see what it really says...
Common Misconceptions

• Microkernel-based systems are less reliable, as failure of one component makes whole system fail

• Wrong!
  – Counter example: QNX High Availability Toolkit (sold commercially since 2001)
  – More recent counter example: Minix 3, which is open source — check it out for yourself

• Were reliability matters most, microkernels are used
  – aerospace, automotive, medical devices...
A Voice from the Coal Face

• “NTFS-3G is a user/hybrid-space driver”
• “Similar functionality and performance on commodity hardware as in-kernel file systems”
• “The invested effort and resource were only a fraction of what is usually needed, besides other benefits.”
• “The empirical learnings keep being highly instructive, refuting widely believed folklore”

Szaka Szabolcs, leader of NTFS-3G, http://ntfs-3g.org
Common Misconceptions

• Microkernel relies on IPC, IPC requires expensive message queue operations, hence IPC is costly

• Wrong!
  - Counter example: L4, since 1993 (publ in SOSP)
  - L4 runs in 10s of millions of mobile phone
  - OS performance is critical for cell-phone baseband processing
  - L4 expected to run on 250M mobile devices within a year

• Why the sudden popularity?
  - it's fast
  - it's small
  - it enables fault containment
Let's Look at IPC

- IPC is used to obtain system service
  - IPC performance is important
Intrinsic Difference Syscall vs IPC

- Syscall: 2 mode switches (user\rightarrow kernel, kernel\rightarrow user)
- IPC: 2 mode switches + 1 context switch
- Server invocation needs 2 IPCs
  - extra cost is 2 mode switches, 2 context switches
- This is the inherent microkernel overhead!
  - it is wrong to think that IPC was used inside the system a lot (replacing function calls)
- Is it significant?
  - depends on the ratio between overhead and total cost of service obtained
  - it's a killer for the null system call
  - it's irrelevant for most others
Actual L4 IPC Cost [cycles]

<table>
<thead>
<tr>
<th>Architecture</th>
<th>Intra address space</th>
<th>Inter address space</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pentium</td>
<td>113</td>
<td>305</td>
</tr>
<tr>
<td>AMD-64</td>
<td>125</td>
<td>230</td>
</tr>
<tr>
<td>Itanium</td>
<td>36</td>
<td>36</td>
</tr>
<tr>
<td>MIPS64</td>
<td>109</td>
<td>109</td>
</tr>
<tr>
<td>ARM Xscale</td>
<td>170</td>
<td>180</td>
</tr>
</tbody>
</table>

How do a couple hundred cycles compare to the typical Linux system call???
Sort-of Extreme Example: Linux on L4

- Cops the full microkernel overhead
- Doesn't get any of the microkernel benefits
- How does it perform?
<table>
<thead>
<tr>
<th>ReAIM Benchmark</th>
<th>Native</th>
<th>Virtualised</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Task</td>
<td>45.2</td>
<td>43.6</td>
<td>0.96</td>
</tr>
<tr>
<td>2 Tasks</td>
<td>23.6</td>
<td>22.6</td>
<td>0.96</td>
</tr>
<tr>
<td>3 Tasks</td>
<td>15.8</td>
<td>15.3</td>
<td>0.97</td>
</tr>
</tbody>
</table>

Native Linux vs Linux virtualized on L4
on Xscale PXA255 @ 400MHz
Not everything in L4 fully optimised yet (fork/exec)
## Lmbench microbenchmarks

### Lmbench latencies in microseconds, smaller is better

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>lat_proc procedure</td>
<td>0.21</td>
<td>0.21</td>
<td>0.99</td>
</tr>
<tr>
<td>lat_proc fork</td>
<td>5679</td>
<td>8222</td>
<td>0.69</td>
</tr>
<tr>
<td>lat_proc exec</td>
<td>17400</td>
<td>26000</td>
<td>0.67</td>
</tr>
<tr>
<td>lat_proc shell</td>
<td>45600</td>
<td>68800</td>
<td>0.66</td>
</tr>
</tbody>
</table>

### Lmbench bandwidths, MB/s, larger is better

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>bw_file_rd 1024 io_only</td>
<td>38.8</td>
<td>26.5</td>
<td>0.68</td>
</tr>
<tr>
<td>bw_mmap_rd 1024 mmap_only</td>
<td>106.7</td>
<td>106</td>
<td>0.99</td>
</tr>
<tr>
<td>bw_mem 1024 rd</td>
<td>416</td>
<td>412.4</td>
<td>0.99</td>
</tr>
<tr>
<td>bw_mem 1024 wr</td>
<td>192.6</td>
<td>191.9</td>
<td>1</td>
</tr>
<tr>
<td>bw_mem 1024 rdwr</td>
<td>218</td>
<td>216.5</td>
<td>0.99</td>
</tr>
<tr>
<td>bw_pipe</td>
<td>7.55</td>
<td>20.64</td>
<td>2.73</td>
</tr>
<tr>
<td>bw_unix</td>
<td>17.5</td>
<td>11.6</td>
<td>0.66</td>
</tr>
</tbody>
</table>

Native Linux vs Linux virtualized on L4

on Xscale PXA255 @ 400MHz

Not everything in L4 fully optimised yet (fork/exec)
## Lmbench Context Switching

Native Linux vs Linux virtualized on L4 on Xscale PXA255 @ 400MHz

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Ratio</th>
</tr>
</thead>
<tbody>
<tr>
<td>lmbench latencies in microseconds, smaller is better</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>lat_ctx -s 0 1</td>
<td>11</td>
<td>20</td>
<td>0.55</td>
</tr>
<tr>
<td>lat_ctx -s 0 2</td>
<td>262</td>
<td>5</td>
<td>52.4</td>
</tr>
<tr>
<td>lat_ctx -s 0 10</td>
<td>298</td>
<td>45</td>
<td>6.62</td>
</tr>
<tr>
<td>lat_ctx -s 4 1</td>
<td>48</td>
<td>58</td>
<td>0.83</td>
</tr>
<tr>
<td>lat_ctx -s 4 10</td>
<td>419</td>
<td>203</td>
<td>2.06</td>
</tr>
<tr>
<td>lat_fifo</td>
<td>509</td>
<td>49</td>
<td>10.39</td>
</tr>
<tr>
<td>lat_pipe</td>
<td>509</td>
<td>49</td>
<td>10.39</td>
</tr>
<tr>
<td>lat_unix</td>
<td>1015</td>
<td>77</td>
<td>13.18</td>
</tr>
<tr>
<td>lat_syscall null</td>
<td>0.8</td>
<td>4.8</td>
<td>0.17</td>
</tr>
</tbody>
</table>
How Can Virtual be Faster than Real?

• It's a **microkernel**!
  - Complete kernel is about 10–11kloc!

• Linux is **big**!
  - 100s of kloc not counting drivers, file systems etc

• ARM MMU is quirky, needs a lot of effort to optimise
  - **much easier to optimize a small code base**

• Of course, the same can be achieved with Linux
  - in fact, we did it and offered patches upstream
  - maintainers didn't take — who cares about factor of 50!
  - Snapgear is running our patches in their modems
Lameter myth: IPC is needed across nodes inside a microkernel OS, and on NUMA this causes problems allocating the message queues NUMA-friendly.

Whom you gonna call — local or remote OS???
Multiprocessor Scalability

- syscall slowdown vs # CPUs
- compare against several commercial systems
- only one system scales (constant slowdown)
- which is it?
What's the story?

- **Tornado microkernel** scales perfectly to 16p
  - this is 1999! [Gamsa et al, 3rd OSDI]
  - done by a small group at Univ of Toronto
  - Tornado is predecessor of IBM's K42

- How far did Linux scale in 1999?

- How far would Linux scale **today** on the same benchmarks?
  - Note: the benchmarks show **concurrent** ops on all CPUs
    - page faults, fstats, thread creation
Synchronization Claims

- “Microkernel isolation limits synchronization methods”
- “Data structures have to be particular to subsystems”
- “Linux would never have been able to scale to these extremes with a microkernel approach because of the rigid constraints that strict microkernel designs place on the architecture of operating systems”

**This is simply wrong (repeating doesn't make it right)**
- synchronisation in a well-designed system is local to subsystems
- there is no reason why subsystems can't share memory, even if microkernel-based
OS Scalability Principles

- OS must not impose synchronisation overhead except as forced by user code
- Then user code scalable $\Rightarrow$ system scalable
- What does this mean?
  - keep data structures local
  - process system calls on the caller's CPU
  - only involve other CPUs if the caller explicitly asks for it!
    - creating/killing/signalling a thread on another CPU
    - invoking a synchronisation system call
    - unmap pages
- If this is done, you get a scalable OS
  - even if the apps actually perform system calls
  - user pays what user asks for...
• Hey, I can do this cool thing but you can't
  - How do you know if you don't understand me?

• Linux is cool
  - but this doesn't mean it is perfect for everything
  - nor does it mean Linux will remain as is forever

• Same is true for microkernels