The OKL4 Microvisor: Convergence Point of Microkernels and Hypervisors

Gernot Heiser, Ben Leslie

Open Kernel Labs and NICTA and UNSW
Sydney, Australia
Microkernels vs Hypervisors

- Hypervisors = “microkernels done right?” [Hand et al, HotOS ‘05]
  - Talks about “liability inversion”, “IPC irrelevance” …

- What’s the difference anyway?
What are Hypervisors?

> Hypervisor = “virtual machine monitor”
  > Designed to multiplex multiple virtual machines on single physical machine

> Invented in ‘60s to time-share with single-user OSes
> Re-discovered in ‘00s to work around broken OS resource management
What are Microkernels?

- Designed to minimise kernel code
  - Remove policy, services, retain mechanisms
  - Run OS services in user-mode
  - Software-engineering and dependability reasons
  - L4: \( \approx 10 \text{ kLOC} \), Xen \( \approx 100 \text{ kLOC} \), Linux: \( \approx 10,000 \text{ kLOC} \)

- IPC performance critical (highly optimised)
  - Achieved by API simplicity, cache-friendly implementation

- Invented 1970 [Brinch Hansen], popularised late ‘80s (Mach, Chorus)
What’s the Difference?

> Both contain all code executing at highest privilege level
  • Although hypervisor may contain user-mode code as well

> Both need to abstract hardware resources
  • Hypervisor: abstraction closely models hardware
  • Microkernel: abstraction designed to support wide range of systems

> What must be abstracted?
  • Memory
  • CPU
  • I/O
  • Communication
## What’s the difference?

<table>
<thead>
<tr>
<th>Resource</th>
<th>Hypervisor</th>
<th>Microkernel</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory</td>
<td>Virtual MMU (vMMU)</td>
<td>Address space</td>
</tr>
<tr>
<td>CPU</td>
<td>Virtual CPU (vCPU)</td>
<td>Thread or scheduler activation</td>
</tr>
<tr>
<td>I/O</td>
<td>• Simplified virtual device</td>
<td>• IPC interface to user-mode driver</td>
</tr>
<tr>
<td></td>
<td>• Driver in hypervisor</td>
<td></td>
</tr>
<tr>
<td></td>
<td>• Virtual IRQ (vIRQ)</td>
<td></td>
</tr>
<tr>
<td>Communication</td>
<td>Virtual NIC, with driver and network stack</td>
<td>High-performance message-passing IPC</td>
</tr>
</tbody>
</table>

- Just page tables in disguise
- Just kernel-scheduled activities
- Real Difference?
- Minimal overhead, Custom API
- Modelled on HW, Re-uses SW
Communication is critical for I/O

- Microkernel IPC is highly optimised
- Hypervisor inter-VM communication is frequently a bottleneck
Hypervisors vs Microkernels: Summary

> Fundamentally, both provide similar abstractions
> Optimised for different use cases

- Hypervisor designed for virtual machines
  - API is hardware-like to ease guest ports
- Microkernel designed for multi-server systems
  - seems to provide more OS-like abstractions
Hypervisors vs Microkernels: Drawbacks

**Hypervisors:**
> Communication is Achilles heel
  - More important than expected
    - Critical for I/O
  - Plenty attempts to improve in Xen

> Most hypervisors have big TCBs
  - Infeasible to achieve high assurance of security/safety
  - In contrast, microkernel implementations can be proved correct

**Microkernels:**
> Not ideal for virtualization
  - API not very effective
    - L4 virtualization performance close to hypervisor
    - effort much higher
  - Virtualization needed for legacy

> L4 model uses kernel-scheduled threads for more than exploiting parallelism
  - Kernel imposes policy
  - Alternatives exist, eg. K42 uses scheduler activations

Could we get the best of both?
Best of Both Worlds: The OKL4 Microvisor

- Microvisor = microkernel hypervisor
- Microkernel with a hypervisor API:
  - \( vCPU \): scheduled by microvisor on real CPU
    - Multiple \( vCPU \)s per VM for running on multiple cores
  - \( vMMU \): looks like a large TLB
    - caches mappings
  - \( vIRQ \): virtualize interrupts and provide asynchronous inter-VM signals
- Plus asynchronous \textit{channel} abstraction
  - for high-bandwidth, low-latency communication
- Small TCB: \( \approx 10 \text{ kLOC} \)
- Suitable for multi-server systems
But Liedtke Said: “IPC Shall Be Synchronous!”

- Early microkernels had poorly-performing semi-synchronous IPC
  - Asynchronous send, synchronous receive
  - Requires message buffering in kernel
  - Each message is copied twice!
But Liedtke Said: “IPC Shall Be Synchronous!”

- Early microkernels had poorly-performing semi-synchronous IPC
- Liedtke designed L4 IPC fully synchronously to avoid such overheads
  - Rendezvous model: each operation blocks until partner is ready
  - No buffering, message copied directly from sender to receiver
  - Requires kernel-scheduled threads to avoid blocking whole app!
In our experience, real-world embedded systems require asynchronous IPC
- Maps more naturally to hardware events
- In the past, we added asynchronous notification (lightweight signals) to L4 IPC

The OKL4 Microvisor only provides fully asynchronous communication:
- vIRQs as asynchronous signals
- Channels for asynchronous data transfer
  - Still single-copy, no kernel buffers!
  - Synchronised eg. by sending vIRQ
## Performance: Lmbench

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>null syscall</td>
<td>0.6 µs</td>
<td>0.96 µs</td>
<td>0.36 µs</td>
</tr>
<tr>
<td>read</td>
<td>1.14 µs</td>
<td>1.31 µs</td>
<td>0.17 µs</td>
</tr>
<tr>
<td>stat</td>
<td>4.73 µs</td>
<td>5.05 µs</td>
<td>0.32 µs</td>
</tr>
<tr>
<td>fstat</td>
<td>1.58 µs</td>
<td>2.24 µs</td>
<td>0.66 µs</td>
</tr>
<tr>
<td>open/close</td>
<td>9.12 µs</td>
<td>8.23 µs</td>
<td>-0.89 µs</td>
</tr>
<tr>
<td>select(10)</td>
<td>2.62 µs</td>
<td>2.98 µs</td>
<td>0.36 µs</td>
</tr>
<tr>
<td>sig handler</td>
<td>1.77 µs</td>
<td>2.05 µs</td>
<td>0.28 µs</td>
</tr>
<tr>
<td>pipe latency</td>
<td>41.56 µs</td>
<td>54.45 µs</td>
<td>12.89 µs</td>
</tr>
<tr>
<td>UNIX socket</td>
<td>52.76 µs</td>
<td>80.90 µs</td>
<td>28.14 µs</td>
</tr>
<tr>
<td>fork</td>
<td>1,106 µs</td>
<td>1,190 µs</td>
<td>84 µs</td>
</tr>
<tr>
<td>fork+execve</td>
<td>4,710 µs</td>
<td>4,933 µs</td>
<td>223 µs</td>
</tr>
<tr>
<td>system</td>
<td>7,583 µs</td>
<td>7,796 µs</td>
<td>213 µs</td>
</tr>
</tbody>
</table>

Beagle Board (500 MHz ARMv7)
## Performance: Netperf on Loopback Device

<table>
<thead>
<tr>
<th>Type</th>
<th>Benchmark</th>
<th>Native</th>
<th>Virtualized</th>
<th>Overhead</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>Throughput</td>
<td>651 [Mib/s]</td>
<td>630 [Mib/s]</td>
<td>3 %</td>
</tr>
<tr>
<td></td>
<td>CPU load</td>
<td>99 [%]</td>
<td>99</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td>12.5 [µ/KiB]</td>
<td>12.9 [µs/KiB]</td>
<td>3 %</td>
</tr>
<tr>
<td>UDP</td>
<td>Throughput</td>
<td>537 [Mib/s]</td>
<td>516 [Mib/s]</td>
<td>4 %</td>
</tr>
<tr>
<td></td>
<td>CPU load</td>
<td>99</td>
<td>99</td>
<td>0 %</td>
</tr>
<tr>
<td></td>
<td>Cost</td>
<td>15.2</td>
<td>15.8 [µs/KiB]</td>
<td>4 %</td>
</tr>
</tbody>
</table>

Beagle Board (500 MHz ARMv7)
Conclusions

- Classical hypervisors and microkernels both have significant drawbacks
  - Hypervisor inter-VM comms primitives are slow, hurts I/O
  - Also have huge TCBs
  - Microkernel APIs not ideal for virtualization

- Microvisor combines best of both
  - Virtualization-oriented API
  - Microkernel (i.e. small) code base
  - Fast communication channels for multi-server designs (stand-alone drivers)

- OKL microvisor shows excellent virtualization performance

- Suitability for multi-server designs still needs to be proven