An Efficient End-host Architecture for Cluster Communication Services

Xin Qi, Gabriel Parmer and Richard West
Introduction

- General-purpose systems provide a general set of abstractions that allow for a good combination of
  - Fairness between processes
  - Simple abstraction of the base hardware to all application processes

- With generality, fine grained control is sacrificed
  - Copying of data via kernel for network stack
  - Networking core cannot be easily extended with new protocols

- High performance applications may demand more than these generic abstractions can provide
Contributions

- High performance networking using “user-level sandboxing”
  - Zero copy
  - Eliminates scheduling overheads
- Safe abstraction
  - Kernel controls access rights to user-level sandbox services
  - Regulated access to I/O devices is guaranteed
- Example usages:
  - Efficient middleware routing of high bandwidth/low latency data streams
  - A proxy server handing remote procedure calls
User-level Sandboxing

- Provides user-level environment for the execution of service extensions
- Separate kernel from app-specific code
- Use only page-level hardware protection
  - Approach does not require specific hardware protection features, such as *segmentation* and *tagged TLBs*
- Extension code only activated by the kernel via upcalls
- Sandbox extensions can be executed in the context of any process to avoid scheduling overheads
Hardware Support for Memory-safe Extensions

- Process-private address space
- Sandbox region (shared virtual address space)
- Kernel Level
- User Level

Extension for P2

Kernel events make sandbox region user-level accessible

E.g., user-level network stack
User-level Sandboxing Implementation

- Modify address spaces of all processes to contain one or more shared pages of virtual addresses
  - Shared pages used for sandbox
    - Normally inaccessible at user-level
    - Kernel upcalls toggle sandbox page protection bits & perform TLB invalidation on corresponding page(s)

- Current x86 approach
  - 2x4MB superpages
  - Modified dietlibc supports most normal functionality
  - ELF loader to map code into sandbox
  - Support sandboxed threads that can block on syscalls
Hardware Support for Memory-safe Extensions

- Private address space
- Sandbox public area
- Protected area
- Extension Code + read-only data
- Mapped Data
- Extension Stacks
- Sandbox public area
- Protected area

4MB 4MB
Invoking Sandbox Extensions

- Fast Upcalls
  - Leverage SYSEXIT/SYSENTER on x86
    - Support Traditional IRET approach also

- Kernel Events
  - Generic interface supports delivery of events to specific extensions
  - Each extension has its own stack & thread structure
  - Events can be queued -- like POSIX.4 (real time) signals
User-level Networking

- Issues involved in building a high performance customizable user-level networking stack:
  - Memory Management
    - A slab allocator that has a-priori knowledge of objects such as packet descriptors
  - Kernel Bypassing
    - An abstraction for passing control to an extension for interrupt time (asynchronous) processing
  - NIC interaction
    - Support DMA transfers of packet data to / from sandboxed extension code
User-level Networking

- UML (User Mode Linux) used as the basis for network service extensions, by providing:
  - memory allocation
  - a modular device interface
  - a fully functional, modular networking stack

- A well defined set of communication channels between kernel and sandbox to pass memory location for packet arrival and transmission
  - High speed DMA to user level is only possible because sandbox exists in every process virtual address space
Demultiplexing packets

Some technologies rely on programmable NICs which have a-priori knowledge of the destination of incoming packets.

All incoming packets must still be allocated and transferred to sandbox area.

A light-weight classifier can be written either in the kernel or sandbox.

Sandbox networking scheme is not intended for efficient processing of all packets.

We focus on efficient communication for sandbox extensions.
User-level Networking
--(Asynchronous Mode)
User-level Networking
--(Synchronous Mode)
Experimental Results

- Example customized service extension: Relay Socket
  - To bind a pair of sockets together
  - For efficient forwarding of packets at transport layer

- UDP Forwarding
  - Comparison of networking implementations
  - Transfer time jitter

- TCP Forwarding
Experiment Environment

A

wget Iperf

B

Socket Relayer

Sandbox Region

Sandbox Relayer

Kernel Relayer

C

httpd Iperf

User Level

Kernel Level

Control / Data Channels
UDP forwarding (1/2)

- **UML in user process vs. UML in sandbox**
  - An improvement of 130% with no background threads
  - With more background threads, sandbox agent does not suffer scheduling delays and therefore maintains high throughput
User-level vs. Kernel-level vs. Sandbox Networking

- Sandbox networking is comparable to kernel approach with no background threads
- Throughput remains constant irrespective of background threads
Low jitter is important for QoS-constrained (e.g. multimedia) applications.

Near constant jitter is demonstrated by the sandboxed networking scheme.

- other two approaches show larger and more variable jitter as the number of background threads increases.
**TCP Forwarding**

- **UML in user space vs. sandbox**
  - Using *wget* to get 1GB file from Apache server via intermediate node
  - 30% improvements in throughput using SCHED_OTHER
  - Prioritizing the sandbox thread using SCHED_RR yields more than 50% higher throughput irrespective of background threads
## Microbenchmarks

<table>
<thead>
<tr>
<th>Operation</th>
<th>Cost in CPU Cycles</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null Fast Upcall</td>
<td>1370</td>
</tr>
<tr>
<td>Sandbox Packet Processing time</td>
<td>6360</td>
</tr>
<tr>
<td>Kernel Packet Processing time</td>
<td>4800</td>
</tr>
</tbody>
</table>
Conclusions

- Efficient networking stack in a “user-level sandbox”
  - Higher throughput and lower jitter than traditional middleware services implemented in process-private address spaces
  - In many cases, our architecture enables user-level services to outperform equivalent kernel-based services that require scheduling

- User-level sandboxing scheme allows extension code to:
  - Safely and efficiently access lower-level abstractions (e.g., interrupt time execution, network hardware)
  - Execute without scheduling process-private address spaces
  - Easy to debug and implement new services.
Future Work

- Type safe language support / software-based fault isolation
- Binary rewriting techniques to avoid patching host kernel for sandbox support