Microdrivers
A New Architecture for Device Drivers

Vinod Ganapathy       Arini Balakrishnan
Michael Swift         Somesh Jha

Computer Sciences Department
University of Wisconsin-Madison

HotOS XI, 8th May 2007
Introduction

- Device drivers account for a large fraction of kernel code
  - Over 70% in Linux [Chou et al., 2001]

- Buggy device drivers are a major source of reliability problems
  - Account for over 85% of Windows XP crashes [Orgovan and Tricker, 2003]
Challenging to write and debug

Device drivers are hard to get right

- Writing a device driver
  - Must handle asynchronous events
  - Must obey kernel programming rules
- Debugging a device driver
  - Non-reproducible failures
  - Fewer advanced development tools
- Many drivers written by non-kernel experts
Macroknernels

Poor fault isolation
User-space device drivers

- Applications
- Kernel
- Runtime
- Device
- Driver
- Driver

The diagram illustrates the relationship between different components in a user-space device driver system.
But…

😊 Poor performance

- Limited by existing kernel/driver interface [Van Maren, 1999]
  - Written expecting local procedure calls

😊 Incompatible with commodity OS

- New interfaces (e.g., new system calls)
- New device drivers [Chubb 2004, Leslie et al., 2005]
Best of both worlds: Microdrivers

Split device driver functionality

Applications

Kernel

Userdriver

Runtime

Microdriver

Runtime

Kerndriver

Startup, shutdown, device configuration

Performance-critical functionality

Device

Rare case
How to produce a microdriver?
Use program analysis & transformation

Traditional device driver → **Splitter and Code generator** → Userdriver → Runtime → Runtime Kerndriver
How big is the kernel driver?

- Studied 455 drivers from Linux-2.6.18
- Identified **critical functions**
  - Interrupt handlers
  - Tasklets, bottom-halves
  - Supply/receive data to/from the device
  - Plus the functions that they transitively call
- Analysis is automatic
  - Uses the call-graph of the device driver
HotOS XI Microdrivers: A New Architecture for Device Drivers

e100_xmit_frame

e100_exec_cmd
e100_xmit_frame

e100_exec_cmd
In-kernel driver code is reduced

<table>
<thead>
<tr>
<th>Category</th>
<th>Non-critical</th>
<th>Critical</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network (134)</td>
<td>72.2</td>
<td>27.8</td>
</tr>
<tr>
<td>SCSI (49)</td>
<td>73.9</td>
<td>26.1</td>
</tr>
<tr>
<td>Sound (272)</td>
<td>92.2</td>
<td>7.8</td>
</tr>
<tr>
<td>Average</td>
<td>79.4</td>
<td>20.6</td>
</tr>
</tbody>
</table>
Mechanics
Architecture of a microdriver

Responsibilities
- Communication
- Object tracking
- Recovery
Communication

- Mechanisms for control and data transfer
- Control transfer:
  - LRPC [Bershad et al., 1990], Nooks XPC [Swift et al., 2003]
  - Stubs in kerndriver for userdriver functions
  - Upcall and downcall mechanism
- Data transfer:
  - Copy function arguments
  - Copy shared data structures
- Synchronization done by object tracker
Object tracking

Synchronize shared data structures

Kerndriver

Userdriver

Upcall

<table>
<thead>
<tr>
<th>0x128</th>
<th>0x512</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x196</td>
<td>0x248</td>
</tr>
<tr>
<td>0x202</td>
<td>0x296</td>
</tr>
</tbody>
</table>
Object tracking: Key challenges

Memory objects with special semantics

- Challenging cases
  - Locks
  - Device memory and registers
  - DMA memory

- Solution
  - Userdriver must synchronize and update version seen by the kerndriver
Recovery

- Detect and recover from failed userdriver
  - Ideally transparent to applications
- Detection done at interface
  - Parameter checks and timeouts
- Recovery – compatible with prior work
  - Shadow driver mechanism [Swift et al., 2004]
  - SafeDrive recovery mechanism [Zhou et al., 2006]
Benefits
Improved fault isolation

Fewer lines of in-kernel driver code
Good common-case performance

User/kernel crosses are infrequent
Compatibility with commodity OS

Kernel/driver interface is unchanged
Compatibility with commodity OS
Can coexist with traditional drivers

Diagram showing the interaction between different components such as Applications, Kernel, Runtime, Userdriver, Driver, and Device.
Ease of driver development

More tools available for driver development

Kernel

Applications

Userdriver

Runtime

Kerndriver

Device

gdb, gprof, valgrind, ...

Runtime
Pragmatics
Generating a microdriver

Traditional device driver

Splitter

Code generator

User

Marshaling annotations

Userdriver

Runtime

Kern

Runtime Kerndriver
Marshaling annotations

Need to serialize complex data structures

- char *nullterm string;
- struct net_device *list next;
- struct pcnet32_rx_head *array(“rx_ringsize”) rx_ring;

- Program analysis algorithms to infer need for annotations
Performance of a microdriver

- e1000 device driver from Linux-2.6.18
  - Intel PRO/1000 gigabit network adapter
- Methodology
  - Split into kerndriver and userdriver
  - Ran both halves in the kernel
  - Used delays to simulate user/kernel crosses
  - Infrastructure is still in construction!
- Testbed
  - Dual-core 3Ghz Pentium-D machine.
TCP-send performance

- Relative performance
  - Throughput
  - CPU

- 26% rise
- 6.8% drop

- 60 million machine cycles

Delay to simulate cost of user/kernel crossing (in microseconds)
Open questions

- Will microdrivers improve system reliability in practice?
  - Where are most of the bugs – in the kerndriver or in the userdriver?

- Will the transition to microdrivers expose otherwise latent bugs in device drivers?
Microdrivers …

... reduce the amount of code in the kernel

... improve fault isolation

... have good common-case performance

... are compatible with commodity OSes

... permit the use of user-mode tools for driver development

... can be generated largely automatically from existing drivers
Microdrivers
A New Architecture for Device Drivers

Vinod Ganapathy  vg@cs.wisc.edu
Arini Balakrishnan  arinib@cs.wisc.edu
Michael Swift  swift@cs.wisc.edu
Somesh Jha  jha@cs.wisc.edu

Computer Sciences Department
University of Wisconsin-Madison