Outline

• Introduction
• Recovery Subsystem Architecture
• Device Drivers and Shadow Driver Design
• Shadow Driver Implementation
• Evaluation
• Conclusions
Introduction

• The goal of this report is to present a operating system mechanism—shadow driver to improve overall reliability when drivers fail.

Implementation of shadow driver’s results:
• masked device driver failures from applications to run normally during and after a driver failure
• Imposed minimal performance overhead
• Required no changes to existing applications and device drivers
• Integrated easily into an existing operating system
Figure 4: The Linux operating system with several device drivers and the driver recovery subsystem. New code components include the taps, the shadow manager and a set of shadow drivers, all built on top of the Nooks driver fault isolation subsystem.
Figure 1: A sample device driver. The device driver *exports* the services defined by the device’s class interface and *imports* services from the kernel’s interface.
1.2 Driver faults

- What time may device drivers crash?
  - The stream of requests from the kernel
  - Messages to and from the device
  - The kernel environment, which swap pages of memory, and interrupt the driver at arbitrary times
A driver bug would be triggered by some failures:

- **Deterministic failure:**
  - triggered by a sequence of configuration or I/O requests
  - recurred when the driver recovers, again causing a failure

- **Transient failure:**
  - triggered by additional inputs from the device or the operating system and occur infrequently
  - triggered by environmental factors that are unlikely to persist during recovery

- **Fail-stop:**
  - detected and stopped by the system before any OS, device, or application state is affected
2.1 Shadow Drivers

- **kernel agent** that improves reliability for a single device driver
- **Compensated** for and recovered from a failed driver
- **When a driver fails**, its shadow restores the driver to process I/O requests before the failure.
- **While the driver recovers**, the shadow driver services it requests.

- **Shadow drivers recover from two failures:**
  - Transient
  - Fail-stop
2.2 Shadow drivers execute in two modes:

- Passive mode
- Active mode

• **Passive mode:**
  - Used during *normal (non-faulting) operating*
  - Shadow driver monitors all communication between the kernel and the device driver
  - Monitoring is achieved via replicated procedure calls existing only to track the state of the driver as necessary for recovery
• Active mode:
  – Occurred during recovery from a failure
  – Shadow drivers in this mode perform three functions:
    a. “impersonates” the failed driver, intercepting and responding to calls from the kernel
    b. The kernel and higher-level applications continue operating in as normal a fashion as possible
    c. The shadow driver impersonates the kernel to restart the failed driver
  – Only the shadow driver is aware of the deception (application is unaware)
• Shadow drivers passive mode
• Shadow drivers active mode
3.1 Tap

Characteristic of Tap:

- Tap is T-junction placed between the kernel and its drivers.

- In passive mode, calls are replicated.
- In active mode, calls are redirected.
• Passive mode operation:
  – invoking the original driver
  – invoking the shadow driver with the parameters and results of the call

• Active mode operation:
  – terminating all communication between the driver and kernel
  – redirecting all invocations to corresponding interface in the shadow
4.1 Shadow Manager

- Recovery is supervised by the shadow manager.
- Shadow manager is a kernel agent that interfaces with and controls all shadow drivers.
- The shadow manager instantiates new shadow and injects taps into the call interfaces between the device driver and kernel.
- Received notification from the fault-isolation subsystem that a driver has stopped due to a failure.
- When a driver fails, the shadow manager transits its taps and shadow driver to active mode.
- When recovery ends, the shadow manager returns driver and taps to passive mode operation so the driver can resume service.
1.1 General infrastructure
Shadow Driver Implementation (2/5)

- Shadow driver implementation uses Nooks to provide these functions
- **Protection domains** use memory protection to trap driver faults and ensure the integrity of kernel
- **Interposes proxy procedures** on all communication between the device driver and kernel
- **Tracks kernel objects** used by drivers to perform garbage collection of kernel resources during recovery
- Shadow manager adds to receiving failure notifications from Nooks
2.1 Passive-Mode Monitoring

A shadow driver records several types of information:

- tracking requests made to the driver
- the state of each active connection for connection-oriented drivers
- a log of pending commands and arguments for request-oriented drivers
- Configuration and driver parameters to act in the driver’s place and reconfigure the driver to its pre-failure state when it is restarted, for example, to keep a log of `ioctl` calls
3.1 Active-Mode Recovery

- Shadow manager detects failure three steps of recovery:
  - Stopping the Failed Driver
  - Reinitializing the Driver
  - Transferring State to New Driver
4.1 Active-Mode Proxying of Kernel Requests

- The shadow driver’s response to a driver request depends on the driver class and request semantics

- **Shadow will take one of five actions:**
  1. Respond with information that it has recorded
  2. Silently drop the request
  3. Queue the request for later processing
  4. Block the request until the driver recovers
  5. Report that the driver is busy and the kernel or application should try again later
• Evaluation is divided into four kinds:
  ➢ Performance
  ➢ Fault tolerance
  ➢ Limitations
  ➢ Code size
**Testing environment**

- 3GHz Pentium 4 PC with 1GB of RAM and 80GB 7200 RPM IDE disk drive

<table>
<thead>
<tr>
<th>Class</th>
<th>Driver</th>
<th>Device</th>
</tr>
</thead>
<tbody>
<tr>
<td>Network</td>
<td>e1000</td>
<td>Intel Pro/1000 Gigabit Ethernet</td>
</tr>
<tr>
<td></td>
<td>pcnet32</td>
<td>AMD PCnet32 10/100 Ethernet</td>
</tr>
<tr>
<td></td>
<td>3c59x</td>
<td>3COM 3c509b 10/100 Ethernet</td>
</tr>
<tr>
<td></td>
<td>e100</td>
<td>Intel Pro/100 Ethernet</td>
</tr>
<tr>
<td></td>
<td>epic100</td>
<td>SMC EtherPower 10/100 Ethernet</td>
</tr>
<tr>
<td>Sound</td>
<td>audigy</td>
<td>SoundBlaster Audigy sound card</td>
</tr>
<tr>
<td></td>
<td>emu10k1</td>
<td>SoundBlaster Live! sound card</td>
</tr>
<tr>
<td></td>
<td>sb</td>
<td>SoundBlaster 16 sound card</td>
</tr>
<tr>
<td></td>
<td>es1371</td>
<td>Ensoniq sound card</td>
</tr>
<tr>
<td></td>
<td>cs4232</td>
<td>Crystal sound card</td>
</tr>
<tr>
<td></td>
<td>i810_audi</td>
<td>Intel 810 sound card</td>
</tr>
<tr>
<td>Storage</td>
<td>ide-disk</td>
<td>IDE disk</td>
</tr>
<tr>
<td></td>
<td>ide-cd</td>
<td>IDE CD-ROM</td>
</tr>
</tbody>
</table>

Table 1: **The three classes of shadow drivers and the Linux drivers tested.** We present results for the **boldfaced drivers only**, as the others behaved similarly.
### Testing application

<table>
<thead>
<tr>
<th>Device Driver</th>
<th>Application Activity</th>
</tr>
</thead>
</table>
| Sound (audigy driver) | • mp3 player (zinf) playing 128kb/s audio  
• audio recorder (audacity) recording from microphone  
• speech synthesizer (festival) reading a text file  
• strategy game *(Battle of Wesnoth)* |
| Network (e1000 driver) | • network send (netperf) over TCP/IP  
• network receive (netperf) over TCP/IP  
• network file transfer (scp) of a 1GB file  
• remote window manager (vnc)  
• network analyzer (ethereal) sniffing packets |
| Storage (ide-disk driver) | • compiler (make/gcc) compiling 788 C files  
• encoder (LAME) converting 90 MB file .wav to .mp3  
• database (mySQL) processing the Wisconsin Benchmark |

Table 2: The applications used for evaluating shadow drivers.
**Evaluation (4/11)**

- **Emulate Faults (Faults Injected)**

Table II. The Types of Faults Injected into Extensions and the Code Transformations Used to Emulate These Faults

<table>
<thead>
<tr>
<th>Fault type</th>
<th>Code transformation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Source fault</td>
<td>Change the source register</td>
</tr>
<tr>
<td>Destination fault</td>
<td>Change the destination register</td>
</tr>
<tr>
<td>Pointer fault</td>
<td>Change the address calculation for a memory instruction</td>
</tr>
<tr>
<td>Interface fault</td>
<td>Use existing value in register instead of passed parameter</td>
</tr>
<tr>
<td>Branch fault</td>
<td>Delete a branch instruction</td>
</tr>
<tr>
<td>Loop fault</td>
<td>Invert the termination condition of a loop instruction</td>
</tr>
<tr>
<td>Text fault</td>
<td>Flip a bit in an instruction</td>
</tr>
<tr>
<td>NOP fault</td>
<td>Elide an instruction</td>
</tr>
</tbody>
</table>
Evaluation (5/11)

- **Performance**

All applications, performance of the system with shadow drivers averaged 99% of the system without, and was never worse than 97%. 
Evaluation (6/11)

• Performance

Network transmits 45,000 packets per second, causing 45,000 domain crossings.
CPU utilization increases from 28% to 57%.
 Fault tolerance

<table>
<thead>
<tr>
<th>Device Driver</th>
<th>Application Activity</th>
<th>Application Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Sound (audigy driver)</strong></td>
<td>mp3 player, audio recorder, speech synthesizer, strategy game</td>
<td>CRASH</td>
</tr>
<tr>
<td><strong>Network (e1000 driver)</strong></td>
<td>network file transfer, remote window manager, network analyzer</td>
<td>CRASH</td>
</tr>
<tr>
<td><strong>IDE (ide-disk driver)</strong></td>
<td>compiler, encoder, database</td>
<td>CRASH</td>
</tr>
</tbody>
</table>

Table 3: The observed behavior of several applications following the failure of the device drivers on which they rely. There are three behaviors: a checkmark (✓) indicates that the application continued to operate normally; CRASH indicates that the application failed completely (i.e., it terminated); MALFUNCTION indicates that the application continued to run, but with abnormal behavior.
Evaluation (8/11)

• Limitations

Figure 7: Results of fault-injection experiments on Linux-SD. We show (1) the percentage of failures that are automatically detected by the fault isolation subsystem, and (2) the percentage of failures that shadow drivers successfully recovered. The total number of failures experienced by each application is shown at the top of the chart.
Evaluation (9/11)

• Limitations for recovery (1/2)
  ➢ Shadow drivers rely on dynamic unloading and reloading of device drivers
  ➢ Shadow drivers rely on explicit communication between the device driver and kernel
  ➢ Shadow drivers assume the driver failure does not cause irreversible side effects
    □ Example: writing bad data on a disk
  ➢ Shadow drivers may not be suitable for applications with real-time demands
Evaluation (10/11)

• Limitations for recovery (2/2)
  ➢ Non-fail-stop Failures
    ➢ System did not detect application hangs caused by I/O requests that never completed
    ➢ System did not detect errors in the interactions between the device and the driver
      - Example: incorrect copying sound data to a sound card
    ➢ System did not detect certain bad parameters
  
  ➢ Non-transient Failures
    ➢ Deterministic failures may recur during recovery
Evaluation(11/11)

- **Code size**

<table>
<thead>
<tr>
<th>Driver Class</th>
<th>Shadow Driver Lines of Code</th>
<th>Device Driver Shadowed Lines of Code</th>
<th>Class Size # of Drivers</th>
<th>Class Size Lines of Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sound</td>
<td>666</td>
<td>7,381 (audigy)</td>
<td>48</td>
<td>118,981</td>
</tr>
<tr>
<td>Network</td>
<td>198</td>
<td>13,577 (e1000)</td>
<td>190</td>
<td>264,500</td>
</tr>
<tr>
<td>Storage</td>
<td>321</td>
<td>5,358 (ide-disk)</td>
<td>8</td>
<td>29,000</td>
</tr>
</tbody>
</table>

Table 4: *Size and quantity of shadows and the drivers they shadow.*
Conclusions

• Shadow drivers improve system reliability:
  – Concealing driver’s failures from both OS and applications
  – Efficiently imposed little performance degradation
  – Transparently required no code changes to existing drivers
  – recovery for an entire class of device drivers