Software Support for Improved Driver Reliability

Michael Swift
University of Wisconsin--Madison
The (old) elephant in the room

- Device drivers are:
  - The majority of kernel code written
  - A large fraction of kernel code in memory
  - Unreliable
    - 89% of Windows XP crashes are from drivers
    - Linux drivers had 7x bug density of other kernel code
  - Difficult to write, test, and maintain
Driver programming has not changed much

- **Unix Version 3, 1973**
  - (dn.c, a DN-11 modem)

```c
void dnwrite(dev) {
    struct dn *dp;
    register struct dn *rdp;
    int c;
    dp = &DNADDR->dn11[dev.d_minor];
    for(;;) {
        while (((rdp = dp)-
                &dn_stat&DONE))==0)
            sleep(DNADDR, DNPRI);
        rdp->dn_stat =& ~DONE;
        if (rdp->dn_reg&(PW|ACR)) {
            u.u_error = EIO;
            return;
        }
        if (rdp->dn_stat&DSS) return;
        rdp = dp;
        rdp->dn_reg = c-'0';
        rdp->dn_stat |= DPR;
    }
}
```

- **Linux 2.6.23, 2007**
  - esp.c, a serial port driver

```c
static int rs_write(struct tty_struct *tty, const unsigned char *buf, int count) {
    int c, t, ret = 0;
    struct esp_struct *info = (struct esp_struct *)tty->driver_data;
    unsigned long flags;
    while (1) {
        c = count;
        t = ESP_XMIT_SIZE - info->xmit_cnt - 1;
        memcpy(info->xmit_buf + info->xmit_head, buf, c);
        info->xmit_head = (info->xmit_head + c) & (ESP_XMIT_SIZE-1);
        ...
    }
    serial_out(info, UART_ESI_CMD1, ESI_SET_SRV_MASK);
    serial_out(info, UART_ESI_CMD2, info->IER);
}
```
Everything else has changed

- **Unix Version 3, 1973**
  - 16 drivers
  - 36 KB of driver code
  - Written by Dennis Ritchie

- **Linux 2.6.31.6, 2009**
  - 4254 driver variations
  - 204 MB of driver code
  - 4.5 million lines of code
  - Written by > 374 people
  - ~1400 person-years of effort
Writing quality drivers is hard

- Kernel programming is hard
  - Few tools
  - Hard to debug
- Many rules to follow
  - Which locks can be used
  - Which memory can be touched
  - Which order operations must follow
- Hardware unreliability
  - May fail independently and transiently

- USB Homer Simpson doll was an epiphany for Microsoft
Software Support for Driver Reliability

- **State of the art:**
  - Driver isolation: allow existing drivers to fail and clean up
    - Nooks
    - SafeDrive
  - Driver architecture: new designs for new drivers to reduce faults and tolerate failures
    - Minix 3
    - Windows UMDF, KMDF
    - UNSW Dingo & Termite

- **Our approach: tool + runtime**
  - **Carburizer**: detect and tolerate device failures in software
  - **Decaf Drivers**: simplify driver development through language
Outline

- Introduction
- **Carburizer**
  - Hardening Drivers
  - Reporting Errors
- Decaf Drivers
- Conclusions
Current state of OS-hardware interaction

- Many device drivers assume device perfection
  - Common Linux network driver: 3c59x .c

While (ioread16(ioaddr + Wn7_MasterStatus)) & 0x8000)

; 

HANG!

Hardware dependence bug: device malfunction can crash the system
Current state of OS-hardware interaction

- Hardware dependence bugs occur across driver classes
- Manifest as hangs, crashes, incorrect behavior

```c
void hptitop_iop_request_callback(...) {
    arg = readl(...);
    ...
    if (readl(&req->result) == IOP_SUCCESS) {
        arg->result = HPT_IOCTL_OK;
    }
}
```

Highpoint SCSI driver(hptiop.c)

*Code simplified for presentation purposes*
How do the hardware bugs manifest?

- **Drivers often trust hardware to always work correctly**
  - Drivers use device data in critical control and data paths
  - Drivers do not report device malfunctions to system log
  - Drivers do not detect or recover from device failures
An example: Windows servers

- Transient hardware failures caused 8% of all crashes and 9% of all unplanned reboots \(^1\)
  - Systems work fine after reboots
  - Vendors report returned device was faultless

- Existing solution is hand-coded hardened driver:
  - Crashes reduced from 8% to 3%

\(^1\) Fault resilient drivers for Longhorn server, May 2004. Microsoft Corp.
Carburizer

- **Goal:** Tolerate hardware device failures in software through hardware failure detection and recovery

- **Static analysis tool** - analyze and insert code to:
  - Detect and fix hardware dependence bugs
  - Detect and generate missing error reporting information

- **Runtime**
  - Handle interrupt failures
  - Transparently recover from failures
Outline

- Introduction
- Carburizer
  - Hardening Drivers
  - Reporting Errors
- Decaf Drivers
- Conclusions
Hardware unreliability

- Sources of hardware misbehavior:
  - Device wear-out, insufficient burn-in
  - Bridging faults
  - Electromagnetic radiation
  - Firmware bugs

- Result of misbehavior:
  - Corrupted/stuck-at inputs
  - Timing errors/unpredictable DMA
  - Interrupt storms/missing interrupts
Vendor recommendations for driver developers

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Recommended by</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Intel</td>
</tr>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Read once&amp; CRC data</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>●</td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td>●</td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>●</td>
</tr>
<tr>
<td>Recovery</td>
<td>Handle all failures</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>●</td>
</tr>
</tbody>
</table>

Goal: *Automatically* implement as many recommendations as possible in commodity drivers
Carburizer architecture

Compile-time components
- Carburizer
- Compiler
- Driver

Run-time components
- Kernel Interface
- Hardened Driver Binary
- Faulty Hardware
- Carburizer Runtime

Example code snippet:
```
if (c==0) {
    print("Driver init");
}
```
Outline

- Introduction
- Carburizer
  - Hardening Drivers
    - Finding sensitive code
    - Repairing code
  - Reporting Errors
- Decaf Drivers
- Conclusions
Hardening drivers

- Goal: Remove hardware dependence bugs
  - Find driver code that uses data from device
  - Ensure driver performs validity checks
- Carburizer detects and fixes hardware bugs from
  - Infinite polling
  - Unsafe static/dynamic array reference
  - Unsafe pointer dereferences
  - System panic calls
Hardening drivers

- Finding sensitive code
  - First pass: Identify tainted variables
Finding sensitive code

First pass: Identify tainted variables

```c
int test () {
    a = read1();
    b = inb();
    c = b;
    d = c + 2;
    return d;
}

int set() {
    e = test();
}
```

Tainted Variables:
- a
- b
- c
- d
- test()
- e
Detecting risky uses of tainted variables

- Finding sensitive code
  - Second pass: Identify risky uses of tainted variables

- Example: Infinite polling
  - Driver waiting for device to enter particular state
  - Solution: Detect loops where all terminating conditions depend on tainted variables
Example: Infinite polling

Finding sensitive code

```c
static int amd8111e_read_phy(...)
{
  ...
  reg_val = readl(mmio + PHY_ACCESS);
  while (reg_val & PHY_CMD_ACTIVE)
    reg_val = readl(mmio + PHY_ACCESS)
  ...
}
```

AMD 8111e network driver(amd8111e.c)
Not all bugs are obvious

```c
while (DAC960_PD_StatusAvailableP(ControllerBaseAddress)) {
    DAC960_V1_CommandIdentifier_T CommandIdentifier = DAC960_PD_ReadStatusCommandIdentifier(ControllerBaseAddress);
    DAC960_Command_T *Command = Controller->Commands [CommandIdentifier-1];
    DAC960_V1_CommandMailbox_T *CommandMailbox = &Command->V1.CommandMailbox;
    DAC960_V1_CommandOpcode_T CommandOpcode = CommandMailbox->Common.CommandOpcode;
    Command->V1.CommandStatus = DAC960_PD_ReadStatusRegister(ControllerBaseAddress);
    DAC960_PD_AcknowledgeInterrupt(ControllerBaseAddress);
    DAC960_PD_AcknowledgeStatus(ControllerBaseAddress);
    switch (CommandOpcode) {
        case DAC960_V1_Enquiry_Old:
            DAC960_P_To_PD_TranslateReadWriteCommand(CommandMailbox);
            ...
    }
}
```

DAC960 Raid Controller(DAC960.c)
Detecting risky uses of tainted variables

- Example II: Unsafe array accesses
  - Tainted variables used as array index into static or dynamic arrays
  - Tainted variables used as pointers
Example: Unsafe array accesses

Unsafe array accesses

```c
static void __init attach_pas_card(...) {
  if ((pas_model = pas_read(0xFF88))) {
    ...
    sprintf(temp, "%s rev %d",
           pas_model_names[(int) pas_model], pas_read(0x2789));
    ...
  }
}
```

Pro Audio Sound driver (pas2_card.c)
Analysis results over the Linux kernel

- Analyzed drivers in 2.6.18.8 Linux kernel
  - 6300 driver source files
  - 2.8 million lines of code
  - 37 minutes to analyze and compile code

- Additional analyses to detect existing validation code
### Analysis results over the Linux kernel

<table>
<thead>
<tr>
<th>Driver class</th>
<th>Infinite polling</th>
<th>Static array</th>
<th>Dynamic array</th>
<th>Panic calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>net</td>
<td>117</td>
<td>2</td>
<td>21</td>
<td>2</td>
</tr>
<tr>
<td>scsi</td>
<td>298</td>
<td>31</td>
<td>22</td>
<td>121</td>
</tr>
<tr>
<td>sound</td>
<td>64</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>video</td>
<td>174</td>
<td>0</td>
<td>22</td>
<td>22</td>
</tr>
<tr>
<td>other</td>
<td>381</td>
<td>9</td>
<td>57</td>
<td>32</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>860</strong></td>
<td><strong>43</strong></td>
<td><strong>89</strong></td>
<td><strong>179</strong></td>
</tr>
</tbody>
</table>

Many cases of poorly written drivers with hardware dependence bugs
Repairing drivers

- Hardware dependence bugs difficult to test
- Carburizer automatically generates repair code
  - Inserts timeout code for infinite loops
  - Inserts checks for unsafe array/pointer references
  - Replaces calls to panic() with recovery service
  - Triggers generic recovery service on device failure
Carburizer automatically fixes infinite loops

```c
timeout = rdstcll(start) + (cpu/khz/HZ)*2;
reg_val = readl(mmio + PHY_ACCESS);
while (reg_val & PHY_CMD_ACTIVE) {
    reg_val = readl(mmio + PHY_ACCESS);
    if (_cur < timeout)
        rdstcll(_cur);
    else
        __recover_driver();
}
```

*Code simplified for presentation purposes*
Carburizer automatically adds bounds checks

```c
static void __init attach_pas_card(...) {
    if ((pas_model = pas_read(0xFF88))) {
        ...
        if ((pas_model < 0) || (pas_model >= 5))
            __recover_driver();
        .
        sprintf(temp, "%s rev %d",
            pas_model_names[(int) pas_model], pas_read(0x2789));
    }
}
```

*Code simplified for presentation purposes*
Runtime fault recovery

- Low cost transparent recovery
  - Based on shadow drivers
  - Records state of driver
  - Transparent restart and state replay on failure
- Independent of any isolation mechanism (like Nooks)
Experimental validation

- Synthetic fault injection on network drivers
  - Modify I/O routines to return modified values

Results

<table>
<thead>
<tr>
<th>Device/Driver</th>
<th>Original Driver</th>
<th>Carburized Driver</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Behavior</td>
<td>Detection</td>
</tr>
<tr>
<td>3COM 3C905</td>
<td>CRASH</td>
<td>None</td>
</tr>
<tr>
<td>DEC DC 21x4x</td>
<td>CRASH</td>
<td>None</td>
</tr>
</tbody>
</table>

Carburizer failure detection and transparent recovery work well for transient device failures.
Outline

- Introduction
- Carburizer
  - Hardening Drivers
  - Reporting Errors
- Decaf Drivers
- Conclusions
Reporting errors

- Drivers often fail silently and fail to report device errors
  - Drivers should proactively report device failures
  - Fault management systems require these inputs

- Driver already detects failure but does not report them

- Carburizer analysis performs two functions
  - Detect when there is a device failure
  - Report unless the driver is already reporting the failure
Detecting driver detected device failures

- Detect code that depends on tainted variables
  - Perform unreported loop timeouts
  - Returns negative error constants
  - Jumps to common cleanup code

```c
while (ioread16 (regA) == 0x0f) {
    if (timeout++ == 200) {
        sys_report("Device timed out %s.\n", mod_name);
        return (-1);
    }
}
```

Reporting code added by Carburizer
Detecting existing reporting code

Carburizer detects function calls with string arguments

```c
static u16 gm_phy_read(...) {
    ...
    if (__gm_phy_read(...))
        printk(KERN_WARNING "\%s: ...\n", ...);
}
```

Carburizer detects existing reporting code

SysKonnect network driver (skge.c)
Evaluation

- Manual analysis of drivers of different classes

<table>
<thead>
<tr>
<th>Driver</th>
<th>Class</th>
<th>Driver detected device failures</th>
<th>Carburizer reported failures</th>
</tr>
</thead>
<tbody>
<tr>
<td>ens1371</td>
<td>sound</td>
<td>10</td>
<td>9</td>
</tr>
</tbody>
</table>

Carburizer *automatically* improves the fault diagnosis capabilities of the system

- No false positives
- Fixed **1135** cases of unreported timeouts and **467** cases of unreported device failures in Linux drivers
## Carburizer Summary

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>●</td>
<td>●</td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Stuck interrupt</td>
<td></td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>Lost request</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Avoid excess delay in OS</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>Recovery</td>
<td>Handle all failures</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>●</td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
</tr>
</tbody>
</table>
## Carburizer Summary

<table>
<thead>
<tr>
<th>Recommendation</th>
<th>Summary</th>
<th>Intel</th>
<th>Sun</th>
<th>MS</th>
<th>Linux</th>
<th>Carburizer Ensures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Validation</td>
<td>Input validation</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Read once &amp; CRC data</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DMA protection</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td>Timing</td>
<td>Infinite polling</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Unexpected events</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Reporting</td>
<td>Report all failures</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td>Recovery</td>
<td>Handle all failures</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Cleanup correctly</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td></td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Do not crash on failure</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
<tr>
<td></td>
<td>Wrap I/O memory access</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
<td>•</td>
</tr>
</tbody>
</table>

Carburizer improves system reliability by *automatically* ensuring that hardware failures are tolerated in software.
Outline

- Introduction
- Carburizer
- Decaf Drivers
  - Intuition
  - Driver Slicer
  - Runtime
  - Evaluation
- Conclusions
Intuition

- Kernel programming is difficult and leads to driver unreliability
- For compatibility and performance, some driver tasks should remain in the kernel.
- Many driver tasks need not
  - Initialization/shutdown
  - Configuration
  - Error handling
- Thesis: there are better languages than C and better places than the kernel to run this code.
Why change drivers?

- Kernel environment is brittle
  - Sensitive to pointer problems
  - Difficult/cumbersome memory management
  - Little support for error handling
  - Difficult to debug
## Kernel vs. Java development

<table>
<thead>
<tr>
<th>Feature</th>
<th>Kernel</th>
<th>Java</th>
</tr>
</thead>
<tbody>
<tr>
<td>Memory management</td>
<td>Manual</td>
<td>Garbage collection</td>
</tr>
<tr>
<td>Type safety</td>
<td>Limited</td>
<td>Extensive</td>
</tr>
<tr>
<td>Debugging</td>
<td>Few tools / difficult</td>
<td>Many tools / easier</td>
</tr>
<tr>
<td>Data structure library</td>
<td>Subset of libc</td>
<td>Java class library</td>
</tr>
<tr>
<td>Error handling</td>
<td>Return values</td>
<td>Exceptions</td>
</tr>
</tbody>
</table>
How much code can potentially be moved from the kernel?

Up to 1.8 million lines of code

Potential for change

Portion of code removable in Linux 2.6.27

- Network (89)
- SCSI (33)
- Sound (175)

Driver Class (# of drivers in class)
Decaf Drivers

- *Decaf Drivers* execute most driver code in user mode Java
  - Performance critical code left in kernel
- The *Decaf System* provides support for
  1. Migrating driver code into a modern language (Java)
  2. Executing drivers with high performance
  3. Evolving drivers over time
Existing Driver Architecture

- Little error checking at compile or runtime
- No rich data structure library
- Few debugging aids

Diagram:
- Application
- Kernel
- Driver
- Device
Decaf Architecture

Application

Kernel

User-Level Driver

Decaf Driver (Java)

Decaf Runtime/XPC

Nuclear Runtime/XPC

Driver

Device
Decaf Architecture

Application

Kernel

Nuclear Runtime/XPC

Driver Nucleus

Decaf Runtime/XPC

Decaf Driver (Java)

Driver Library (C)

User-Level Driver

Device
Creating Decaf Drivers

1. Annotate it
2. Run DriverSlicer to split the driver into a Driver Nucleus and Library
3. Migrate code from the Driver Library into the Decaf Driver
Splitting a driver

Goal: separate critical code from the rest

1. Low latency requirements
2. High bandwidth requirements
3. High priority requirements

Solution: leverage standard driver interfaces

1. Identify critical root functions for a driver from driver interface definition
2. Expand transitively through call graph
3. Identify all entry point functions where control passes between the driver library and nucleus
Splitting drivers

- Legacy device driver
- Splitter
- Code generator
- Driver Library
- Marshaling annotations
- User
- Kernel
- Marshaling
- Driver Nucleus
Generating marshaling code

- Goal: generate code for entry point functions to pass data structures between kernel and user

- Problems:
  - Kernel structures are highly linked
  - Types defined incompletely in C
Marshaling linked structures

- Solution: only copy fields actually accessed
  - Identify which fields are accessed from each entry point
  - Generate unique code for each entry point
Field analysis example

**net_device before:**

```c
struct net_device {
    char       name[IFNAMSIZ];
    struct hlist_node name_hlist; /* shared mem end
    unsigned long mem_start;    /* shared mem start
    unsigned long mem_end;      /* device I/O
    unsigned int  irq;          /* device IRQ number
    unsigned char if_port;      /* Selectable AUI, TP,...*/
    unsigned char dma;          /* DMA channel
    unsigned long state;        /* device I/O address */
    struct net_device *next;    /* interface flags (a la BSD) */
    int (*init)(struct net_device *dev); /* invisible to userspace. */
    unsigned long features;     /* How much padding added */
    unsigned long trans_start;  /* interface MTU value */
    unsigned short flags;       /* interface hardware */
    unsigned short gflags;      /* for shared network cards */
    unsigned short padded;      /* hardware hdr length*/
    struct net_device *master;  /* Multicast mac */
    unsigned short mtu;         /* Number of installed */
    int ifindex;                /* interface hardware */
    int iflink;                 /* AppleTalk link */
    int *ip_ptr;                /* IPv4 specific data */
    void *dn_ptr;               /* DECnet */
    void *ip6_ptr;              /* IPv6 specific */
    int *ec_ptr;                /* Econet specific data */
    void *ax25_ptr;             /* AX.25 specific data */
    struct dev_mc_list *mc_list; /* Multicast mac */
    int mc_count;               /* Number of installed */
    int promiscuity;            /* AppleTalk link */
    int allmulti;               /* IPv4 specific data */
    int *atalk_ptr;             /* Econet specific data */
    int *poll_list;             /* AX.25 specific data */
    unsigned long dev_id;       /* IPv6 specific */
    struct list_head poll_list; /* for shared */
    struct list_head poll_list; /* for shared */
};
```
Marshaling incomplete types

Extend C with 7 marshaling annotations:
- Nullterm
- Array
- Combolock
- Opaque
- Sentinel
- Storefield
- Container

Guides programmers in placing annotations
Annotation example

```c
struct pcnet32 private {
    const char * name;
    int rx_ring_size;
    struct pcnet32_rx_head * rx_ring;
    spinlock_t lock; ...
}
```

Problem Pointers

- Problem lock
- Problem Pointers
struct pcnet32 private {
    const char * Nullterm name;
    int rx_ring_size;
    struct pcnet32_rx_head *
        Array(rx_ring_size) rx_ring;
    spinlock_t Combolock lock; ...
}
Java access to kernel data and functions

Problem
- Different type systems
- No easy conversion

Leverage: RPC systems solve this problem

Phase one: DriverSlicer emits XDR
- Extracts all data structure definitions and typedefs
- Converts these definitions to an XDR specification

Phase two: `rpcgen` and `jrpcgen` generate code
- Create Java classes with public fields
- Support features like recursive data structures
Phase 1: Example

```
struct e1000_adapter { ...
    struct e1000_rx_ring test_rx_ring;
    uint32_t * __attribute__((exp(PCI_LEN))) config_space;
    int msg_enable;
    ...
};
```

```
typedef unsigned int uint32_t;

struct uint32_256_t {
    uint32_t array_256[256];
};

typedef struct uint32_t_256 *uint32_t_256_ptr;

struct e1000_adapter { ...
    struct e1000_rx_ring test_rx_ring;
    uint32_t_256_ptr config_space;
    int msg_enable;
    ...
};
```

Original C code

Automatically-generated XDR Definition
Phase 2: Continued

typedef unsigned int uint32_t;
struct uint32_256_t {
    uint32_t array_256[256];
};
typedef struct uint32_t_256 *uint32_t_256_ptr;
struct e1000_adapter {
    struct e1000_rx_ring test_rx_ring;
    uint32_t_256_ptr config_space;
    int msg_enable;
    ... }

public class e1000_adapter {
    public e1000_rx_ring test_rx_ring;
    public uint32_t_256_ptr config_space;
    public int msg_enable;
    ... 
    public e1000_adapter () {
    public e1000_adapter(XdrDecStream xdr) {
    public void xdrEncode(XdrEncStream xdr) {
    public void xdrDecode(XdrDecStream xdr) {
}
DriverSlicer Summary

- **Splitter**
  - Identifies kernel code from **critical root functions**
  - Identifies driver library/nucleus entry points

- **Marshaler**
  - Generates code to marshal/unmarshal structures
  - Identifies which fields are accessed in user mode

- **Java Conversion**
  - C → XDR conversion
  - XDR code generation
Outline

- Introduction
- Overview
- Design and Implementation
- Evaluation
  - Conversion effort
  - Performance analysis
  - Benefits of Decaf Drivers
    - Case study of E1000 gigabit network driver
- Conclusion
## Conversion Effort

<table>
<thead>
<tr>
<th>Driver</th>
<th>Original Lines of Code</th>
<th>Annotations</th>
<th>Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>Driver Nucleus</td>
</tr>
<tr>
<td>e1000</td>
<td>14,204</td>
<td>64</td>
<td>46</td>
</tr>
<tr>
<td>8139too</td>
<td>1,916</td>
<td>17</td>
<td>12</td>
</tr>
<tr>
<td>ens1371</td>
<td>2,165</td>
<td>18</td>
<td>6</td>
</tr>
<tr>
<td>psmouse</td>
<td>2,448</td>
<td>17</td>
<td>15</td>
</tr>
<tr>
<td>uhci-hcd</td>
<td>2,339</td>
<td>94</td>
<td>68</td>
</tr>
</tbody>
</table>
Results: Relative Performance

Decaf / Native Perf.

8139too tx  8139too rx  E1000 tx  E1000 rx  ens1371  uhci-hcd  psmouse
Results: CPU Utilization

- E1000: Core 2 Quad 2.4Ghz, 4GB RAM
- All others: Pentium D 3.0Ghz, 1GB RAM

- One XPC call every two seconds
- No XPC
- 15 XPC calls on playback start/end
- No XPC
Experience Rewriting Drivers

- **Step one: initial conversion**
  - Largely mechanical: syntax is similar
  - Leaf functions first, then remainder

- **Step two: use Java language features**
  - Example benefit: E1000 exception handling
Java Error Handling

Original C, e1000_hw.c

```c
if(hw->ffe_config_state == e1000_ffe_config_active) {
    ret_val = e1000_read_phy_reg(hw, 0x2F5B, &phy_saved_data);
    if(ret_val) return ret_val;

    ret_val = e1000_write_phy_reg(hw, 0x2F5B, 0x0003);
    if(ret_val) return ret_val;

    msec_delay_irq(20);
    ret_val = e1000_write_phy_reg(hw, 0x0000, IGP01E1000_IEEE_FORCE_GIGA);
    if(ret_val) return ret_val;
}
```

- Easy to miss an error condition
Java Error Handling

Java, e1000_hw.java

```java
if(hw.ffe_config_state.value == e1000_ffe_config_active) {
    e1000_read_phy_reg(0x2F5B, phy_saved_data);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
    e1000_write_phy_reg((short) 0x2F5B, (short) 0x0003);
    DriverWrappers.Java_msleep (20);
    e1000_write_phy_reg((short) 0x0000,
        (short) IGP01E1000_IEEE_FORCE_GIGA);
}
```

- Uncovered at least 28 cases of ignored error conditions
- Resulting code 8% shorter *overall*
Decaf Summary

- Decaf Drivers simplify driver programming
  - Provide a migration path from C to Java
  - Allow driver code to run in user mode
  - Support continued driver and kernel evolution
  - Offer excellent performance
Conclusions

- Large-scale changes to driver architecture or development practices take time
- Drivers can be improved through program-analysis tools and a small amount of runtime code
  - Carburizer: detect/fix hardware dependency bugs
  - Decaf drivers: migrate code to Java
- Runtime costs can be low, by focusing on uncommon cases
Questions?
Reliability

- Kernel code is brittle
- 89% of XP crashes are from drivers
- Linux drivers had 7x bug density of other kernel code
Driver Evolution

- Example: E1000 network driver 2.6.18.1 to 2.6.27
  - e1000_adapter structure needs additional members
- Existing marshaling code does not transfer new fields automatically
- Solution: the driver is the specification
  1) Add new member definitions to original e1000.h
  2) Re-run DriverSlicer
  3) Use variables in Driver Nucleus or Decaf Driver
What has changed?

Driver Classes in Unix, 1973
1. Character
2. Disk
3. Tape
4. Printer
5. TTY

Driver/Device Classes in Windows Vista/7
1. 1394 device
2. Auxiliary display (SideShow)
3. Bluetooth L2CAP
4. Bluetooth Radio Frequency Communications (RFCOMM)
5. Cell phone, PDA, portable media player*
6. Digital camera
7. Display adapter
8. Human input device (HID)
9. Keyboard/Mouse filter
10. Modem, cable modem
11. Network Transport Driver Interface (TDI) client
12. Network-connected device*
13. Printer
14. Scanner
15. Secure digital (SD)
16. Serial and parallel devices (legacy)
17. Smart card device
18. USB device
19. Video capture