FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Diplomarbeitspräsentation

Language Support for Linux Device Driver Programming

Master- /Diplomstudium:
Software Engineering and Internet Computing

Günter Anton Khyo

Technische Universität Wien
Institut für Computersprachen
Arbeitsbereich: Programmiersprachen und Übersetzer
BetreuerIn: Ao.Univ.-Prof. Dr. M. Anton Ertl

The Device Driver Reliablity Problem

Drivers contain 3 to 7 times more bugs per LOC than any other kernel component [1]

The Linux driver tree is huge, featuring over 7.4 Million SLOC!

Drivers execute in privileged mode => every device driver has the potential to crash the entire system.

Thesis Objectives

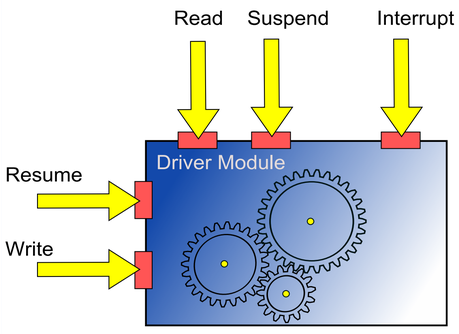
1. Improve the Reliability of Linux Device Drivers, without changing the way drivers are written (we do not want to port 7.4 MSLOC).
2. Identify Key Problems in Driver Development
3. Provide compiler and language support for essential programming aspects

Why are Device Drivers buggy?

We can identify two Root Causes

1. Behavior of devices poorly described by informal, incomplete and inconsistent specifications
2. My Focus: Drivers implemented with Highly Complex Programming Model

1. Concurrency model is too complex



Driver code is highly concurrent

- Multiple requests occur at the same time
- Hotplug and Powermgmt-Events, Interrupts, I/O Requests,...
- One seperate execution trace for each event!
- Blocking operations in interrupts lead to deadlocks!

A typical NIC driver has over 20 entry points!

Parallel code paths induce many side-effects and interdependencies

✗ Race conditions and deadlocks constitute 19% of all driver faults [3]

2. Hardware I/O is error prone

I/O code consists of low-level bit arithmetic

✗ C compiler does not check consistency on hardware I/O operations

✗ Minor programming slips/typos break I/O code

3. No separation between OS-specific and device-specific logic

✗ API evolutions break drivers [2]

✗ Drivers contain up to 20% copy&pasted code [4]

Experimental Results

2 Linux Drivers have been ported to CiD ...

	8139C+ NIC Driver	USB Mass Storage Driver
Concurrency and Synchronization		
Conflicting Access Patterns Inferred	600	512
Race Condition Reports (of synchronized driver code)	40	110
Inferred atomic ops	N/A - none in driver	8
Critical sections / Locks inferred	18 / 1	14 / 3
Hardware I/O		
Superfluous Byte Order Conv.	0	3
Code Statistics		
Code Reduction	14%	0%

✓ All locks are correctly inferred and balanced

✓/✗ Code size of NIC driver reduced by 14% ...
but no reduction for mass storage driver achieved

✓/✗ Race condition detection very accurate for NIC driver ...
but compiler has a hard time with obscure mass storage code

→ Currently, register-oriented devices (8139C+) are best supported by CiD, but message-oriented drivers (mass storage) need support for asynchronous functions and coordination patterns.

Making Drivers Robust with Language Extensions

Introducing CiD (**C** for **D**rivers)

A subset of C with built-in support for concurrency, hardware I/O and code reuse

1. Support for Concurrency

Built-in support for locking - No Hassle with lock types and instances

```
request interrupt irq_handler(...) {
   synchronized {
      foo(shared_data);
   }
}
```

→

```
spinlock_t lock;
irqreturn_t irq_handler (...) {
   spin_lock(&lock);
      foo(shared_data);
   spin_unlock(&lock);
}
```

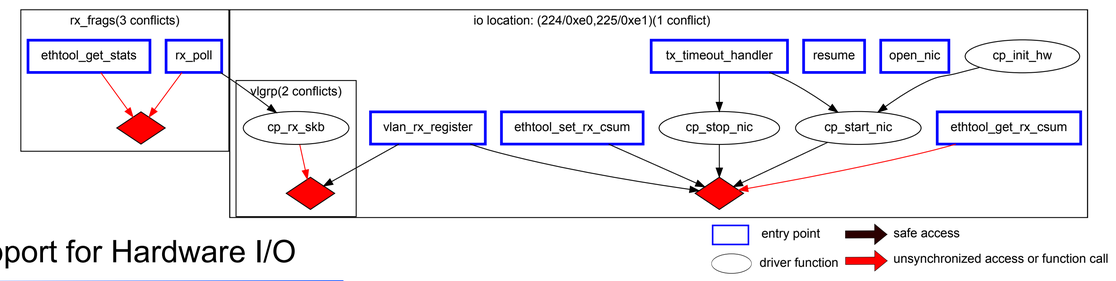
Built-in support for atomic expressions

```
int open_count<atomic>;
open_count = open_count + 1;
```

→

```
atomic_t open_count;
atomic_inc(&open_count);
```

✓ The CiD Compiler assists the programmer with detection of data-flow Races



2. Support for Hardware I/O

Introducing descriptor types ("portable bitfields") and register files (see thesis)

bit→ ↓byte	7	6	5	4	3	2	1	0
0	Operation code = 2Ah							
1	LUN			DPO	FUA	EBP	Rsrvd	RelAdr
2–5	LBA							
6	Reserved							
7–8	Transfer length							
9	Control							

→

```
descriptor SCSI_Write10 {
   0  :   unsigned int(8) op_code= 0x2A;
   1  :   unsigned int(3) LUN: 7..5;
          bit DPO:     4;
          bit FUA:     3;
          bit EBP:     2;
          _   :        1; /* Reserved */
          bit RelAdr:  0;
   5..2: unsigned int(32) LBA;
   6    : _; /* Reserved */
   8..7: unsigned int(16) TransferLength;
      9: unsigned int(8)  Control;
}
```

CiD Compiler performs:

✓ Consistency checks on descriptor and register-file layouts
Are all bits accounted for? Are there overlapping regions?
✓ Bit arithmetic for accessing individual fields
✓ Automatic byte order conversion on field access

3. Code Templates

✓ Seperate device-specific code from device-class code

```
request int PCINET.probe(…)
{
   return setup_device();
}
```

→

1. Setup PCI Device
2. Claim PCI Address Space
3. Initialize DMA
   $:code$
4. Register Network Device

References

[1] A. Chou et al., An Empirical Study of Operating System Errors, Proc. 18th ACM Symposium on Operating Systems Principles, 2001
[2] Y. Padioleau et al., Understanding Collateral Evolution in Linux Device Drivers, In Proc. of EuroSys 2006
[3] L. Ryzhyk et al., Dingo: Taming device drivers, Proc. of EuroSys 2009
[4] Zhenim Li et al., CP-Miner: A Tool for Finding Copy-paste and Related Bugs in Operating System Code IEEE Transactions on Software Engineering, 2004

Kontakt: e0326024@student.tuwien.ac.at