Automatic Synthesis of High-Assurance Device Drivers

Leonid Ryzhyk

- Project Overview
- WP1: Guided Sequential Synthesis
Project members

- **NICTA (Sydney, Australia)**
  - PI: Gernot Heiser
- **University of Toronto**
  - Co-PI's: Michael Stumm, Leonid Ryzhyk
- **University of Colorado Boulder**
  - PI: Pavol Cerny
- **Imperial College London**
  - PI: Alastair Donaldson
Motivation

● The joys of driver development
  ● Drivers are hard to write
  ● … and even harder to debug
  ● They often delay product delivery
  ● … and are the most common source of OS failures
Can We Fix Drivers?

Static analysis

Model checking

Runtime verification

Software isolation

The drivers problem

Hardware isolation

Symbolic verification

DSL's
Can We Fix Drivers?

• Lots of research, but only limited practical impact:
  • SLAM
  • User-level driver frameworks in Linux and Windows
  • Register description languages
We are going about it the wrong way!

- Driver as a C program:
  - 1000's lines of code
  - Extensive use of bit-level arithmetic
  - Extensive use of pointers and dynamic memory allocation
  - Event-driven logic
  - Concurrency
What Drivers Actually Do

• The device provides a service (e.g., storage or communication)
• The OS wants to use the service
• The driver translates OS requests into device commands (kind of like RPC)
  • Every bit of every register must be read and written correctly and in the right order
  • Memory buffers must be allocated and formatted, and later recycled
  • OS resources must be reserved for each operation (timers, physical buffers, interrupts, locks, etc)
• This translation is tedious and error-prone, but largely mechanical
What Drivers Don't Do

• Drivers rarely perform complex computation and data transformation
  • If they do, this functionality can be encapsulated in a separate module
Perfect Target for Automation!

- Largely mechanical task
- Tedious and error-prone
- Determined by input specifications
Formal OS interface spec

Formal device spec

driver.c
Proposal Overview

• Current driver development methodology is beyond fixing

• We propose to re-think driver development practices with the goal of achieving:
  • Strong correctness guarantees
  • Reduced development and maintenance effort

• Not a theoretical exercise!
  • The goal is to synthesise and verify drivers for complex real-world devices (network, storage, audio, etc.)
Work Packages

- WP1 (University of Toronto, NICTA)
  - Sequential synthesis
- WP2 (University of Colorado Boulder)
  - Concurrent synthesis
- WP3 (Imperial College)
  - Automatic verification
Work Packages

WP1 (sequential synthesis)

WP2 (concurrent synthesis)

WP3 (verification)

Input specs

Sequential driver

Concurrent driver
Work Packages

- Work packages are largely independent
- Individual WPs have the potential to produce valuable scientific and practical results
- Together they have the potential to solve the drivers problem
Work Package 1: Guided Sequential Synthesis
Formal OS interface spec

Formal device spec

driver.c
Where Do Specifications Come from?

- A device spec can be as complex as the driver
- Use existing device specifications developed by hardware designers

Formal OS interface spec

Formal device spec

driver.c
Hardware Design Workflow

1. Informal specification
2. High-level model
3. Register-transfer-level description
4. netlist
Hardware Design Workflow

- Informal specification
- High-level model
- Register-transfer-level description
- netlist

- Low-level description: registers, gates, wires.
- Cycle-accurate
- Precisely models internal device architecture and interfaces
Hardware Design Workflow

- Informal specification
- Captures external behaviour
- Abstracts away structure and timing
- Abstracts away the low-level interface

High-level model

- Register-transfer-level description

netlist

```c
bus_write(u32 addr, u32 val) {
    ...
}
```
Driver synthesis as controller synthesis

Driver = controller

OS requests = control objective

send() - send a network packet
Driver synthesis as controller synthesis

Driver = controller

OS requests = control objective

send() - send a network packet

Packet has been sent
Game theory

- Game theory
  - Provides a theoretical framework for verification and synthesis of reactive systems
  - Provides a classification of games
  - Complexity bounds for various types of games
  - Algorithms for finding winning strategies
Example: trivial network adapter

- **off**
  - write(ctl,1)
  - write(ctl,0)

- **on**
  - write(dat,...)

- **bsy**
  - write(ctl,1)

- **done**

- **send**

- uncontrollable transition

- controllable transition
Computing the winning set

- **INIT**
- **on**
  - `write(ctl,0)`
  - `write(ctl,1)`
  - `write(dat,...)`
- **bsy**
  - `write(ctl,1)`
- **done**
  - `send`
- **GOAL**
Computing the winning set

\[
\text{write}(\text{ctl}, 0) \\
\text{write}(\text{ctl}, 1) \\
\text{write}(\text{dat}, \ldots) \\
\text{write}(\text{ctl}, 1) \\
\text{send} \\
\text{Cpre}\{\text{done}\} = \{\text{bsy}\}
\]

\[
\text{INIT} \xrightarrow{\text{write}(\text{ctl}, 0)} \text{off} \xrightarrow{\text{write}(\text{ctl}, 1)} \text{on} \xrightarrow{\text{write}(\text{dat}, \ldots)} \text{bsy} \xrightarrow{\text{write}(\text{ctl}, 1)} \text{done} \xrightarrow{\text{send}} \text{GOAL}
\]
Computing the winning set

- **INIT**: On transition to the initial state, write(ctl,0) or write(ctl,1) can be done.

- **off**
  - write(ctl,1): On transition to on.
  - write(ctl,0): ~

- **on**
  - write(dat,...): On transition to bsy.

- **bsy**
  - Cpre({done,bsy}) = {bsy,on}
  - send: On transition to done.
  - write(ctl,1): On transition to GOAL.

- **done**
  - Cpre({done}) = {bsy}

- **GOAL**: On transition from bsy.
Computing the winning set

\[
\begin{align*}
\text{Cpre}\{\text{done,bsy,on}\} &= \{\text{off,done,bsy,on}\} \\
\text{Cpre}\{\text{done,bsy}\} &= \{\text{bsy,on}\} \\
\text{Cpre}\{\text{done}\} &= \{\text{bsy}\}
\end{align*}
\]
OS specification

Game objective: The driver must be in state 0 infinitely often (aka Büchi objective)
Winning strategy

- off 0 → xmit → off 1
- write(ctl,1) → on 1
- write(dat,...) → bsy 1
- write(ctl,1) → send
- xmit
- done 0 → xmit → done 1
- xmit_complete
- done 2
Challenges

1. State explosion
2. Support for DMA
3. Synthesis with imperfect information
Challenges: State Explosion

- Every bit in every device register doubles the size of the system state space
  - e.g., $2^{320}$ states in a simplified IDE controller model
- Classical game theory algorithms do not scale well
Tackling State Explosion: Predicate Abstraction

- $x$ (32 bits) - current device configuration
- $y$ (32 bits) - new configuration requested by the OS
- Total state space: $2^{64}$ states
- Introducing predicate: $x=y$
- The predicate can be represented with a single boolean variable (2 states)
- Naive abstraction algorithm reduces IDE state space to $2^{48}$ states
Tackling State Explosion: Symbolic Algorithms

- Even after abstraction the state space is too large to explore explicitly
- Symbolic data structures allow representing and manipulating large state spaces compactly
- Common symbolic representations:
  - Binary Decision Diagrams (BDD)
    - BDD encoding of abstracted IDE spec consists of ~3000 BDD nodes
  - SAT formulas
Challenges: DMA

- Synthesising drivers for DMA-capable devices
- The entire RAM is now part of the state space
Synthesis for DMA

1. Typed view of memory

DMA circular buffer:

OS request queue:

2. Predicates over in-memory data structures

\[ \forall_{i=0}^{l-1} \text{queue}[i] = \text{buffer}[i] \]
Challenges: Imperfect Information

- The basic synthesis algorithm assumes complete knowledge of device state
- In reality, device-internal state is invisible to the driver
  - Status registers are used to determine (relevant parts of) the state
Tackling Imperfect Information

• In synthesis, we must reason about **sets of possible states** rather than individual states => further exponential state explosion

• In practice, only a few bits of unobservable state are relevant to the driver

• Heuristically discover those bits and perform subset construction only on them
Is It Going to Work?

• NICTA & Intel have built a prototype implementation of a driver synthesis tool
  • Simplistic abstraction algorithm
  • Symbolic algorithms
  • Rudimentary support for DMA and partial information
  • DML frontend
Successfully synthesised drivers

IDE disk controller

Asix AX88772 USB-to-Eth adapter

W5100 Eth shield

SD host controller
Lessons Learned

• Automatic driver synthesis is possible
• High-level hardware models are suitable inputs for driver synthesis
• Abstraction and symbolic algorithms are the way to go
Lessons Learned

• There are areas where human expertise is essential:
  • Functionality
  • Correctness
  • Readability
  • Performance

• The “all or nothing” approach to synthesis will not yield satisfactory drivers
Don't Fire Your Driver Developers Yet!
Guided Synthesis

- The user has complete control over synthesised source code
- The user communicates their decisions to the tool via source code
- User errors can lead to synthesis failures, but not to an incorrect driver
Guided Synthesis
Scenario 1: Fully Automatic Synthesis

send()
{
    ...
}
receive()
{
    ...
}

driver template

send()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,snd);
}
receive()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,rcv);
}
synthesised driver
Guided Synthesis
Scenario 2: Hybrid Approach

send()
{
    ...
}

receive()
{
    ...
}

empty driver template

send()
{
    write(ctl, flags);
    ...
}

receive()
{
    ...
}

partially synthesised driver


Guided Synthesis
Scenario 2: Hybrid Approach

send(){
    write(ctl,0);
    ...
}
receive(){
    ...
}

modified driver template

send(){
    write(ctl,flags);
    ...
}
receive(){
    ...
}

partially synthesised driver
Guided Synthesis
Scenario 2: Hybrid Approach

send()
{
    write(ctl,0);
    ...
}
receive()
{
    ...
}
modified driver template

send()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,snd);
}
receive()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,rcv);
}
synthesised driver
Guided Synthesis
Scenario 2: Hybrid Approach

send()
{
    write(ctl,0);
}

receive()
{
    ...
}

modified driver template

send()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,snd);
}

receive()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,rcv);
}

synthesised driver
Guided Synthesis
Scenario 3: Verification

manually developed driver

send()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,snd);
}

receive()
{
    write(ctl,flags);
    write(irq_en,0xff);
    write(cmd,rcv);
}

√
Conclusions

- The promise of automatic device-driver synthesis:
  - correct-by-construction device drivers at a fraction of the cost of manual development
  - practical alternative to traditional driver development

We thank Intel for the opportunity to carry out this research!