User-Guided Device Driver Synthesis

Leonid Ryzhyk    Adam Walker    John Keys
Alexander Legg    Arun Raghunath    Michael Stumm
Mona Vij
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
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

Funded by a research grant from Intel
Observation

• Driver development is a mechanical task
• Can in principle be automated
Observation

- Driver development is a mechanical task
- Can in principle be automated
Driver Synthesis as a Game

- Driver synthesis can be formalised as a two-player game: *driver* vs *(device + OS)*
Motivation
Motivation

Addresses an important problem
Motivation

Addresses an important problem

A simple, neat idea
Motivation

Addresses an important problem

A simple, neat idea

One of few applications of FM to OS (beyond verification)
Motivation

Addresses an important problem

A simple, neat idea

One of few applications of FM to OS (beyond verification)

Considered impossible
driver.c

device spec
Driver Synthesis as a Game
Driver Synthesis as a Game

request:set_time(19:30:00)
Driver Synthesis as a Game

request:set_time(19:30:00)
write_hours(19)
Driver Synthesis as a Game

request:set_time(19:30:00)
write_hours(19)
write_minutes(30)
Driver Synthesis as a Game

request:set_time(19:30:00)
write_hours(19)
write_minutes(30)
write_seconds(00)
Driver Synthesis as a Game

request: set_time(19:30:00)
write_hours(19)
TICK
Driver Synthesis as a Game

request:set_time(19:30:00)
write_hours('19)
TICK

20:00:00
Driver Synthesis as a Game

request:set_time(19:30:00)
write_hours(19)
TICK
write_minutes(30)
write_seconds(00)
Driver Synthesis as a Game

request: set_time(19:30:00)

STOP

write_hours(19)

write_minutes(30)

write_seconds(00)

START
Driver Synthesis as a Game
Driver Synthesis as a Game

set_time(19:30:00)
Driver Synthesis as a Game

set_time(19:30:00)
Driver Synthesis as a Game
Driver Synthesis as a Game

STOP

9:59:59

set_time

9:59:59

TICK

10:00:00

write_hours

9:59:59

write_hours

19:59:59

TICK

20:00:00

write_minutes

19:59:59

write_minutes

19:30:59

TICK

20:30:00

write_seconds

19:30:59

write_seconds

19:30:00

write_seconds

19:30:00

STOP
Driver Synthesis as a Game
Driver Synthesis as a Game
Driver Synthesis as a Game

```
set_time
9:59:59
STOP
write_hours
9:59:59
STOP
write_minutes
19:59:59
START
write_hours
19:59:59
START
write_minutes
19:30:59
START
write_seconds
19:30:00
START
STOP
```
Driver Synthesis as a Game
Driver Synthesis as a Game
Termite Tool Demo
Push-Button Synthesis (SOSP'09)

• In theory:
  
  correct spec => correct implementation

• In practice: (based on our experience) taking control away from the developer is not a good idea
Push-Button Synthesis (SOSP'09)

- Choosing a preferred implementation method is hard (e.g., polling vs interrupts)
- Non-functional properties (power, performance, timing, etc.) are hard to enforce
- Achieving “nice” code structure is hard
User-Guided Synthesis

- The user is in control
  - can write arbitrary manual code or …
  - arbitrarily alter automatically generated code
- Synthesiser works as smart auto-complete
  - can generate a statement, a function, or even the whole driver (on demand)
  - never alters user code
  - completes synthesised+manual code to a correct implementation when possible
- The tool enforces correctness
Demo (continued)
Guided Synthesis
Scenario 1: Fully Automatic Synthesis

driver template

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

synthesised driver

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

drive template

send()
  ...}
}

receive()
  ...
}

partially synthesised driver

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

receive()
  ...
}

42
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 3: Verification

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

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

manually developed driver

√
Obtaining Specs for Driver Synthesis

- OS interface spec
- driver.c
- device spec
Obtaining Specs for Driver Synthesis

The synthesis paradox: developing the spec is harder than writing the driver “by hand”.

OS interface spec

device spec
Obtaining Specs for Driver Synthesis

OS specs are **generic**, i.e., made once for a **class** of devices
Obtaining Specs for Driver Synthesis

- OS interface spec
- device spec

Device specs obtained from hardware developers

- driver.c
## Synthesised Drivers

<table>
<thead>
<tr>
<th>Device</th>
<th>Synthesis time (s)</th>
<th>locs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time clock</td>
<td>74</td>
<td>56</td>
</tr>
<tr>
<td>IDE</td>
<td>71</td>
<td>94</td>
</tr>
<tr>
<td>STM32F103RB UART</td>
<td>309</td>
<td>74</td>
</tr>
<tr>
<td>exynos 5 UART</td>
<td>177</td>
<td>35</td>
</tr>
<tr>
<td>STM32F103RB I2C</td>
<td>39</td>
<td>119</td>
</tr>
<tr>
<td>exynos 5 I2C</td>
<td>40</td>
<td>77</td>
</tr>
<tr>
<td>webcam</td>
<td>190</td>
<td>113</td>
</tr>
<tr>
<td>SPI</td>
<td>15</td>
<td>27</td>
</tr>
</tbody>
</table>
## Synthesised Drivers

<table>
<thead>
<tr>
<th>Device</th>
<th>Synthesis time (s)</th>
<th>locs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Real-time clock</td>
<td>74</td>
<td>56</td>
</tr>
<tr>
<td>IDE</td>
<td>71</td>
<td>94</td>
</tr>
<tr>
<td>STM32F103RB UART</td>
<td>309</td>
<td>74</td>
</tr>
<tr>
<td>exynos 5 UART</td>
<td>177</td>
<td>35</td>
</tr>
<tr>
<td>STM32F103RB I2C</td>
<td>39</td>
<td>119</td>
</tr>
<tr>
<td>exynos 5 I2C</td>
<td>40</td>
<td>77</td>
</tr>
<tr>
<td>webcam</td>
<td>190</td>
<td>113</td>
</tr>
<tr>
<td>SPI</td>
<td>15</td>
<td>27</td>
</tr>
</tbody>
</table>
Scope and Limitations

- Focus on synthesising device control logic
  - Resource allocation, binding to OS interfaces, etc., must be written manually or synthesised using different techniques
- Sequential synthesis
  - Synchronisation synthesis as a separate step (jointly with CU Boulder and IST Austria)
- No DMA support
  - WiP
Summary

• Termite automates tedious driver development
• The user has full control over the source code, but Termite enforces correctness

https://github.com/termite2
http://termite2.org
Summary

- Termite automates tedious driver development
- The user has full control over the source code, but Termite enforces correctness
- Driver synthesis is less impossible than previously believed

https://github.com/termite2
http://termite2.org