A Declarative Language Approach to Device Configuration

Adrian Schüpbach  Andrew Baumann  Timothy Roscoe  Simon Peter
Systems Group, ETH Zurich
This talk is about

- Hardware resource configuration is harder than you think
  1. The idealized problem is complex
  2. In practice there are many exceptions and quirks
- We apply high-level languages to deal with hardware configuration
  - Approach
  - Evaluation
Hardware configuration is surprisingly complex!

- Allocate hardware resources to devices
  - Physical address ranges
  - RAM buffers
  - Interrupt lines
  - ...

- These resources are limited

- The problem is constrained in multiple ways

- Hardware in reality does not fit the specifications, it often has bugs
Example: PCI bus configuration

- Tree with multiple children per node
  - Inner nodes: PCI bridges
  - Leaves: devices
  - PCI bridge hierarchy translates physical addresses on device requests
  - Base address registers (BARs) define base address
Example: PCI bus configuration

- Tree with multiple children per node
  - Inner nodes: PCI bridges
  - Leaves: devices
  - PCI bridge hierarchy translates physical addresses on device requests
  - Base address registers (BARs) define base address
Example: PCI bus configuration

- Tree with multiple children per node
  - Inner nodes: PCI bridges
  - Leaves: devices
  - PCI bridge hierarchy translates physical addresses on device requests
  - Base address registers (BARs) define base address

physical addresses

root bridge

bridge 1 bridge 2 bridge 3 bridge 4 bridge 5 bridge 6 bridge 7

d1 d2 d3 d4 d5 d6 d7
Example: PCI bus configuration

- Tree with multiple children per node
  - Inner nodes: PCI bridges
  - **Leaves: devices**
  - PCI bridge hierarchy translates physical addresses on device requests
  - Base address registers (BARs) define base address

```
0000000000000000
0000000000000000
1111111111111111
1111111111111111
```

```
0000000000000000
0000000000000000
1111111111111111
1111111111111111
```

```
0000000000000000
0000000000000000
1111111111111111
1111111111111111
```

```
0000000000000000
0000000000000000
1111111111111111
1111111111111111
```

```
0000000000000000
0000000000000000
1111111111111111
1111111111111111
```

```
0000000000000000
0000000000000000
1111111111111111
1111111111111111
```

```
0000000000000000
0000000000000000
1111111111111111
1111111111111111
```

```
```
Example: PCI bus configuration

- Tree with multiple children per node
  - Inner nodes: PCI bridges
  - Leaves: devices
  - **PCI bridge hierarchy translates physical addresses on device requests**
  - Base address registers (BARs) define base address

```
root bridge
```

```
bridge 1
bridge 2
bridge 3
bridge 4
bridge 5
bridge 6
bridge 7

d1
   /
   
   d2
   /  
   
   d3   d4
   /    /
   
   d5    d6    d7
```
Example: PCI bus configuration

- Tree with multiple children per node
  - Inner nodes: PCI bridges
  - Leaves: devices
  - PCI bridge hierarchy translates physical addresses on device requests
  - **Base address registers (BARs) define base address**
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

1. Uninitialized PCI bus
The Problem
Hardware resource allocation in PCI

In theory, apply the following rules.

2. All devices should be configured
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

3. No overlapping of siblings must occur
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

3. No overlapping of siblings must occur
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

4. Device addresses have to be naturally aligned
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

4. Device addresses have to be naturally aligned
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

4. Device addresses have to be naturally aligned
The Problem
Hardware resource allocation in PCI

In theory, apply the following rules.

5. Children have to be within their parent bridges address range
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

5. Children have to be within their parent bridges address range
The Problem
Hardware resource allocation in PCI

In theory, apply the following rules.

6. The PCI tree has to fit within available address range
The Problem
Hardware resource allocation in PCI

In **theory**, apply the following rules.

6. The PCI tree has to fit within available address range
The Problem
Hardware resource allocation in PCI

In theory, apply the following rules.

6. The PCI tree has to fit within available address range
The Problem

Hardware resource allocation in PCI

In *theory*, apply the following rules.

6. The PCI tree has to fit within available address range
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules.

6. The PCI tree has to fit within available address range

physical addresses

0  8  16  24  32  40  48  56  64  72  80  88  96

root bridge

bridge 1

bridge 3

bridge 2

bridge 1

bridge 2

bridge 3

0  8  4840322416

physical addresses

968880726456

bridge 1

bridge 2

bridge 3

d3
sz 32

d1
sz 4

d7
sz 4

d2
sz 16

d4
sz 4

d5
sz 8

d6
sz 4
The Problem

Hardware resource allocation in PCI

In theory, apply the following rules. Until here idealized problem.

6. The PCI tree has to fit within available address range

![Diagram showing physical addresses and PCI tree structure]
The Problem
Hardware resource allocation in PCI

But in practice handle also special cases.

7. Some devices have fixed address requirements

```
physical addresses
0  8  16  24  32  40  48  56  64  72  80  88  96

root bridge
bridge 1
bridge 2
bridge 3

d3
sz 32
fixed address 0
d2
sz 16
d1
sz 4
sz 4
d7

bridge 3
d6
sz 4
sz 4

d4
sz 4

bridge 2
d5
sz 8

bridge 1
```

7th March 2011
Systems Group | Department of Computer Science | ETH Zürich
The Problem
Hardware resource allocation in PCI

But in practice handle also special cases.

7. Some devices have fixed address requirements
The Problem

Hardware resource allocation in PCI

But in practice handle also special cases.

7. Some devices have fixed address requirements

![Physical addresses diagram](image)
The Problem
Hardware resource allocation in PCI

But in practice handle also special cases.

7. Some devices have fixed address requirements
The Problem

Hardware resource allocation in PCI

But in **practice** handle also special cases.

7. Some devices have fixed address requirements
The Problem

Hardware resource allocation in PCI

But in **practice** handle also special cases.

7. Some devices have fixed address requirements

![Diagram of PCI resource allocation](image)
The Problem
Hardware resource allocation in PCI

But in **practice** handle also special cases.

8. Physical memory regions have holes

![Diagram showing physical addresses and bridges with holes and sizes]

- Physical memory regions have holes.
The Problem
Hardware resource allocation in PCI

But in practice handle also special cases.

8. Physical memory regions have holes

physical addresses

```
0  8  16  24  32  40  48  56  64  72  80  88  96
```

root bridge

bridge 1

bridge 2

bridge 3

d1 d7
sz 4 sz 4

d2 sz 16

d3 sz 32

d5 d4 d6
hole sz 8 sz 4 sz 4

7th March 2011
The Problem

Hardware resource allocation in PCI

But in practice handle also special cases.

8. Physical memory regions have holes
The Problem

Hardware resource allocation in PCI

But in practice handle also special cases.

8. Physical memory regions have holes

![Physical Memory Regions Diagram]
Quirks (some of the 3000 LOCs in Linux’s quirks.c)

/*Following the PCI ordering rules is optional on the AMD762. I’m not sure what the designers were smoking but let’s not inhale...
To be fair to AMD, it follows the spec by default, its BIOS people who turn it off! */
Quirks (some of the 3000 LOCs in Linux’s quirks.c)

- /*Following the PCI ordering rules is optional on the AMD762. I’m not sure what the designers were smoking but let’s not inhale... To be fair to AMD, it follows the spec by default, its BIOS people who turn it off! */

- This card decodes and responds to addresses not apparently assigned to it. We force a larger allocation to ensure that nothing gets put too close to it.
Quirks (some of the 3000 LOCs in Linux’s quirks.c)

- /*Following the PCI ordering rules is optional on the AMD762. I’m not sure what the designers were smoking but let’s not inhale...
To be fair to AMD, it follows the spec by default, its BIOS people who turn it off! */

- This card decodes and responds to addresses not apparently assigned to it. We force a larger allocation to ensure that nothing gets put too close to it.

- S3 868 and 968 chips report region size equal to 32M, but they decode 64M.
Quirks (some of the 3000 LOCs in Linux’s quirks.c)

- /*Following the PCI ordering rules is optional on the AMD762. I’m not sure what the designers were smoking but let’s not inhale... To be fair to AMD, it follows the spec by default, its BIOS people who turn it off! */
- This card decodes and responds to addresses not apparently assigned to it. We force a larger allocation to ensure that nothing gets put too close to it.
- S3 868 and 968 chips report region size equal to 32M, but they decode 64M.
- The first BAR is the location of the IO APIC...we must not touch this (and it’s already covered by the fixmap), so forcibly insert it into the resource tree. The next five BARs all seem to be rubbish, so just clean them out.
Quirks (some of the 3000 LOCs in Linux’s quirks.c)

- /*Following the PCI ordering rules is optional on the AMD762. I’m not sure what the designers were smoking but let’s not inhale...To be fair to AMD, it follows the spec by default, its BIOS people who turn it off! */

- This card decodes and responds to addresses not apparently assigned to it. We force a larger allocation to ensure that nothing gets put too close to it.

- S3 868 and 968 chips report region size equal to 32M, but they decode 64M.

- the first BAR is the location of the IO APIC...we must not touch this (and it’s already covered by the fixmap), so forcibly insert it into the resource tree, The next five BARs all seem to be rubbish, so just clean them out
What do people do today?

- Linux uses BIOS allocation and runs fixup procedure
  - Configure missing devices
  - Allocate address range from bridge, or fail if bridge does not have enough free address range
- Windows Vista, Server 2008: PCI Multi-Level Rebalance
  - Can move bridges to a place with bigger free space
- IBM US patent 5,778,197, 1998: Method for allocating system resources in a hierarchical bus structure
  - Recursive bottom-up algorithm to allocate resources
What we wanted to try

- Express allocation problem as constraint logic program (CLP) in high-level language
- Explore modern techniques to configure hardware
- Separate allocation computation from register access

- Why CLP?
  - Allows constraining variables before assigning concrete values
  - Natural way to implement allocation rules
  - Naturally express hardware constraints and limitations
  - Handle quirks in a clean way, not ad-hoc
  - Leads to platform independence and portability

- We use ECLiPS\textsuperscript{e}: Prolog + constraint extensions
How does CLP work?

1. Create tree data structure which matches the PCI tree
2. Create Base and Size variables in every node in the data structure
3. Apply constraints to these variables
4. Instantiate the variables with concrete values representing PCI base addresses
Nonoverlap rule: siblings must not overlap

Code written in ECL\textsuperscript{i}PS\textsuperscript{e}

\begin{verbatim}
nonoverlap(Tree) :-
    % collect direct children of this root in ChildList
t(_,Children) = Tree,
maplist(root,Children,ChildList),
% if there are direct children...
( not ChildList=[] ->
    % determine base and size of each child
    maplist(base,ChildList,Bases),
    maplist(size,ChildList,Sizes),
    % constrain the regions they define not to overlap
    disjunctive(Bases,Sizes)
    ; true
),
% recurse on all children
( foreach(El, Children) do nonoverlap(El) ).
\end{verbatim}
Quirk: do not move BARs pointing to IOAPICs

Code written in ECLiPSe

```
keep_ioapic_bars(_, []).  
keep_ioapic_bars(Buselements, [H|IOAPICList]) :-
    ( % get the base of the first IOAPIC
        range(B, _) = H,
        % check if a BAR with the same original base exists
        bar(addr(Bus, Dev, Fun), _, OrigBase, _, _, _, _),
        OrigBase =:= B ->
        % if found, keep the device at its original address
        keep_orig_addr(Buselements, _, _, _, _, Bus, Dev, Fun);
        true
    ),
    % iterate on the IOAPIC list
    keep_ioapic_bars(Buselements, IOAPICList).
```
Implementation

- We program the PCI bus in our research operating system Barrelfish like this

- We use ECL\textsuperscript{i}PS\textsuperscript{e}-CLP engine to run the algorithm
  - Starts early in the operating system boot sequence
  - Uses a RAM disk to load everything necessary
  - Is self-contained
Boot sequence

OS startup procedure

PCI bus
Boot sequence

OS startup procedure \(\xrightarrow{\text{starts}}\) CLP engine

PCI bus
Boot sequence

- OS startup procedure
- CLP engine
- PCI driver

PCI bus
Boot sequence

OS startup procedure
start

CLP engine
start

PCI driver

1. scan
hardware information

PCI bus
Boot sequence

1. scan hardware information
2. hw information
PCI driver

1. scan hardware information

OS startup procedure

CLP engine

PCI bus
Boot sequence

OS startup procedure

CLP engine

PCI driver

PCI bus

1. scan
   hardware
   information

2. hw information
3. call algorithm

1. scan
   hardware
   information
Boot sequence

1. scan hardware information
2. hw information
3. call algorithm
4. get the result

OS startup procedure
CLP engine
PCI driver

1. scan hardware information

PCI bus
Boot sequence

1. scan hardware information
2. hw information
3. call algorithm
4. get the result
5. program PCI registers

OS startup procedure
CLP engine
PCI driver
PCI bus

1. scan hardware information
2. hw information
3. call algorithm
4. get the result
5. program PCI registers
Evaluation

➤ We care about
  ➤ Correctness of the allocation
  ➤ Maintainability of the code
  ➤ Performance not the primary focus

➤ We used nine different real hardware systems for the evaluation
## Evaluation

<table>
<thead>
<tr>
<th>C LOC</th>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>235</td>
<td>31</td>
</tr>
<tr>
<td>Data structure</td>
<td>817</td>
<td>31</td>
</tr>
<tr>
<td>Algorithm</td>
<td>360</td>
<td>224</td>
</tr>
<tr>
<td>Interrupts</td>
<td>660</td>
<td>28</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>109</td>
<td>283</td>
</tr>
<tr>
<td>Total</td>
<td>2181</td>
<td></td>
</tr>
</tbody>
</table>

Table: LOC our approach

---

7th March 2011
Systems Group | Department of Computer Science | ETH Zürich
Evaluation

<table>
<thead>
<tr>
<th>Category</th>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>897</td>
<td></td>
</tr>
<tr>
<td>Data structure</td>
<td>1686</td>
<td></td>
</tr>
<tr>
<td>Resource management</td>
<td>706</td>
<td></td>
</tr>
<tr>
<td>ACPI</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td>521</td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3976</strong></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>Data structure</td>
<td>817</td>
<td>31</td>
</tr>
<tr>
<td>Algorithm</td>
<td></td>
<td>224</td>
</tr>
<tr>
<td>ACPI</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td>660</td>
<td>28</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2181</strong></td>
<td><strong>283</strong></td>
</tr>
</tbody>
</table>

Table: LOC Linux

▶ Do not move a device:

Table: LOC our approach
Evaluation

<table>
<thead>
<tr>
<th></th>
<th>C LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>897</td>
</tr>
<tr>
<td>Data structure</td>
<td>1686</td>
</tr>
<tr>
<td>Resource management</td>
<td>706</td>
</tr>
<tr>
<td>ACPI</td>
<td>121</td>
</tr>
<tr>
<td>Interrupts</td>
<td>521</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>45</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>3976</strong></td>
</tr>
</tbody>
</table>

Table: LOC Linux

- Do not move a device: call `keep_orig_addr()`

<table>
<thead>
<tr>
<th></th>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>Data structure</td>
<td>817</td>
<td>31</td>
</tr>
<tr>
<td>Algorithm</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACPI</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td>660</td>
<td>28</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2181</strong></td>
<td><strong>284</strong></td>
</tr>
</tbody>
</table>

Table: LOC our approach
Evaluation

<table>
<thead>
<tr>
<th>C LOC</th>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>897</td>
<td></td>
</tr>
<tr>
<td>Data structure</td>
<td>1686</td>
<td></td>
</tr>
<tr>
<td>Resource management</td>
<td>706</td>
<td></td>
</tr>
<tr>
<td>ACPI</td>
<td>121</td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td>521</td>
<td></td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>45</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>3976</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>235</td>
</tr>
<tr>
<td>Data structure</td>
<td>817</td>
</tr>
<tr>
<td>Algorithm</td>
<td></td>
</tr>
<tr>
<td>ACPI</td>
<td>360</td>
</tr>
<tr>
<td>Interrupts</td>
<td>660</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>109</td>
</tr>
<tr>
<td>Total</td>
<td>2181</td>
</tr>
</tbody>
</table>

Table: LOC Linux

- Do not move a device: call `keep_orig_addr()`
- IOAPIC appears as BAR:

Table: LOC our approach

7th March 2011
Systems Group | Department of Computer Science | ETH Zürich
## Evaluation

<table>
<thead>
<tr>
<th>C LOC</th>
<th></th>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>897</td>
<td>Register access</td>
<td>235</td>
</tr>
<tr>
<td>Data structure</td>
<td>1686</td>
<td>Data structure</td>
<td>817</td>
</tr>
<tr>
<td>Resource management</td>
<td>706</td>
<td>Algorithm</td>
<td>239</td>
</tr>
<tr>
<td>ACPI</td>
<td>121</td>
<td>ACPI</td>
<td>360</td>
</tr>
<tr>
<td>Interrupts</td>
<td>521</td>
<td>Interrupts</td>
<td>660</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>45</td>
<td>Miscellaneous</td>
<td>109</td>
</tr>
<tr>
<td>Total</td>
<td>3976</td>
<td>Total</td>
<td>2181</td>
</tr>
</tbody>
</table>

### Table: LOC Linux

- Do not move a device: call `keep_orig_addr()`
- IOAPIC appears as BAR: implement `keep_ioapic_bars()`

### Table: LOC our approach
Evaluation

<table>
<thead>
<tr>
<th></th>
<th>C LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>897</td>
</tr>
<tr>
<td>Data structure</td>
<td>1686</td>
</tr>
<tr>
<td>Resource management</td>
<td>706</td>
</tr>
<tr>
<td>ACPI</td>
<td>121</td>
</tr>
<tr>
<td>Interrupts</td>
<td>521</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>45</td>
</tr>
<tr>
<td>Total</td>
<td>3976</td>
</tr>
</tbody>
</table>

Table: LOC Linux

- Do not move a device: call `keep_orig_addr()`
- IOAPIC appears as BAR: implement `keep_ioapic_bars()`
- Additional requirements to handle quirks easy to apply

<table>
<thead>
<tr>
<th></th>
<th>C LOC</th>
<th>CLP LOC</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register access</td>
<td>235</td>
<td></td>
</tr>
<tr>
<td>Data structure</td>
<td>817</td>
<td>31</td>
</tr>
<tr>
<td>Algorithm</td>
<td>239</td>
<td></td>
</tr>
<tr>
<td>ACPI</td>
<td>360</td>
<td></td>
</tr>
<tr>
<td>Interrupts</td>
<td>660</td>
<td>28</td>
</tr>
<tr>
<td>Miscellaneous</td>
<td>109</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>2181</td>
<td>298</td>
</tr>
</tbody>
</table>

Table: LOC our approach
Evaluation

Memory consumption and performance

- ECL\textsuperscript{1PS\textsuperscript{e}} is about 16242 LOCs of C
- Solver executable (statically linked): 1.5MB
- 600kB RAM disk
- 60MB dynamically allocated RAM buffers
- Execution time in the range of 2ms to 36ms
Conclusion

- PCI configuration in the real world is a hard, irregular problem
- Declarative languages
  - Tradeoff CPU cycles and memory footprint for simpler code
  - Facilitate handling quirks and other hardware bugs
- We think it is a promising approach for dealing with a large, diverse, and evolving hardware base

Download:
http://www.barrelfish.org
## Changes to quirks.c

Kernel 2.6.36, 2005-2010

<table>
<thead>
<tr>
<th>#commits</th>
<th>Year</th>
</tr>
</thead>
<tbody>
<tr>
<td>26</td>
<td>2005</td>
</tr>
<tr>
<td>47</td>
<td>2006</td>
</tr>
<tr>
<td>49</td>
<td>2007</td>
</tr>
<tr>
<td>43</td>
<td>2008</td>
</tr>
<tr>
<td>42</td>
<td>2009</td>
</tr>
<tr>
<td>23</td>
<td>2010</td>
</tr>
</tbody>
</table>
Code examples

Keep original address

\[
\text{keep\textunderscore orig\textunderscore addr}([], _, _, _, _, _, _).
\]

\[
\text{keep\textunderscore orig\textunderscore addr}([\text{H}|\text{Tl}], \text{Cl}, \text{SubCl}, \text{PIf}, \text{Bs}, \text{Dv}, \text{Fn}) : -
\]

\[
(\text{if this is a device BAR...}
\text{buselement}(\text{device}, \text{addr(Bs,Dv,Fn)}, \text{BARNr,Base,_,_,_,_,_,_})
= \text{H},
\% and its device is in the required class...
\text{device(_,addr(Bs,Dv,Fn),_,_,Cl, SubCl, PIf,_)},
\% lookup the original base address of the BAR
\text{bar(addr(Bs,Dv,Fn),BARNr,OrigBase,_,_,_,_,_)} ->
\% constrain the Base to equal its original value
\text{Base $= OrigBase}
; \text{true}
),
\]

\[
\text{keep\textunderscore orig\textunderscore addr}(\text{Tl}, \text{Cl}, \text{SubCl}, \text{PIf}, \text{Bs}, \text{Dv}, \text{Fn}).
\]
Discussion

Advantages

▶ Policy/mechanism separation
▶ Handle special cases completely in ECL\textsuperscript{i}PS\textsuperscript{e}
▶ General data entries
▶ Late-binding of algorithm
▶ Platform-independence
Discussion

Advantages

▶ Policy/mechanism separation
▶ Handle special cases completely in ECL^i^P^S^e
▶ General data entries
▶ Late-binding of algorithm
▶ Platform-independence

Disadvantages

▶ Increased resource usage
▶ Large code base
▶ Boot sequence
▶ Learning curve
▶ Need sometimes to understand how solver works
Space consumption

- artificial experiment
  - add more and more devices
  - sum of address space requests of all devices fill available range
  - monitor behaviour of postorder algorithm and CLP algorithm
Space consumption

![Graph showing space consumption]

- Root size (max)
- Device sum
- CLP
- Postorder sorted ascending

Physical address size required [bytes]

Fill rate (device sum / max available size) [%]
Valid and invalid configurations

Valid configurations

Invalid configurations
Valid and invalid configurations

Valid configurations

Example 1

Invalid configurations

physical addresses
root bridge
bridge 1
d1
Valid and invalid configurations

Valid configurations

Example 1

Invalid configurations
Valid and invalid configurations

Valid configurations

Example 1

physical addresses
root bridge
bridge 1
d1

Example 2

physical addresses
root bridge
bridge 1
bridge 2
bridge 3
d1
d2

Invalid configurations

physical addresses
root bridge
bridge 1
d1
Valid and invalid configurations

Valid configurations

Example 1

physical addresses
root bridge

bridge 1

d1

Example 2

physical addresses
root bridge

bridge 1

bridge 2

bridge 3

d1
d2

Invalid configurations

physical addresses
root bridge

bridge 1

d1

bridge 2

bridge 3

d1
d2
Why is it difficult?

- placement of child depends on placement of parent
- permutation of siblings possible at every level
- natural address alignment: big gaps possible
  - bad for resource utilization
  - good for hotplug
- fixed address requirements influence placing of parent bridges and siblings
- finding reasonable tree permutation is hard
  - changing order of bridges causes children to move as well
  - children can also be permuted
Why is it difficult?

Example

- Bridge 1
- Bridge 2
- Bridge 3

Physical addresses:
- d1
- d2
- d3
- d4
- d5
- d6
- d7

Root bridge
Why is it difficult?

Example

7th March 2011
Why is it difficult?

Example
Why is it difficult?

Example
Why is it difficult?

Example
Why is it difficult?

Example

![Diagram of bridges and physical addresses]
Why is it difficult?

Example
The Problem

▸ **In theory**, find a valid allocation of address ranges to devices, such that
  ▸ All devices and bridges are configured
  ▸ No overlapping of siblings occurs
  ▸ Addresses are aligned to device specific boundaries
  ▸ Children are within their parent bridge’s address window
  ▸ Complete PCI tree fits within available physical address space
The Problem

- **In theory**, find a valid allocation of address ranges to devices, such that
  - All devices and bridges are configured
  - No overlapping of siblings occurs
  - Addresses are aligned to device specific boundaries
  - Children are within their parent bridge’s address window
  - Complete PCI tree fits within available physical address space

- **But in practice** also
  - Certain devices can only have (partially) fixed addresses
  - Some bridges must be programmed with predefined values
  - Some physical regions have “holes” that can’t be used
  - “Quirks”
Implementation

- We use ECL\textsuperscript{i}PS\textsuperscript{e}-CLP to implement the algorithm
  - Prolog + constraint extensions
- We use a real system: Barrelfish
  - New operating system for heterogeneous manycore systems
  - Implemented from scratch \(\rightarrow\) lots of freedom to try out ideas
- Implementation done in the system knowledge base (SKB)
  - User-space service containing ECL\textsuperscript{i}PS\textsuperscript{e}
  - Contains data base with hardware facts in Prolog form
  - Uses RAM disk to access ECL\textsuperscript{i}PS\textsuperscript{e} code and is self-contained
Boot sequence in Barrelfish
Boot sequence in Barrelish
Boot sequence in Barrelfish
Boot sequence in Barrelfish

- CPU Driver (Kernel)
  - CPU
  - SKB
  - PCI bus

 starts

7th March 2011
Boot sequence in Barrelfish

- CPU
- CPU Driver (Kernel)
- Init
- SKB
- PCI driver
- PCI bus

- Starts from
- Starts
- Starts
Boot sequence in Barrelfish

1. scan
   hardware
   information

CPU Driver
(Kernel)

PCI driver

PCI bus

1. scan
   hardware
   information

SKB

starts

starts

starts
Boot sequence in Barrelfish

1. scan hardware information
   - CPU Driver (Kernel)
   - SKB
   - PCI driver
2. hw information
   - PCI bus

- CPU
- starts
- starts
- starts
Boot sequence in Barrelfish

1. scan
   hardware
   information

CPU Driver
(Kernel)

CPU

Init

starts

SKB

starts

2. hw information

3. call algorithm

PCI driver

1. scan
   hardware
   information

PCI bus
Boot sequence in Barrelfish

1. scan hardware information
2. hw information
3. call algorithm
4. get the result

CPU Driver (Kernel)

Init

SKB

PCI driver

PCI bus

1. scan hardware information

7th March 2011
Systems Group | Department of Computer Science | ETH Zürich
Boot sequence in Barrelfish

1. scan hardware information
2. hw information
3. call algorithm
4. get the result
5. program PCI registers

- Init
- SKB
- PCI driver
- CPU Driver (Kernel)
- CPU
- PCI bus