

# **Returning data-flow to asynchronous programming**

**Matt Gilbert, Senior Staff Engineer, Qualcomm Technologies, Inc. 2018-04-16**

# Background

- Hardware design focuses on information flow (data and control): how do you compose the pieces of execution to balance speed, efficiency, and area?
- We self-impose asynchronicity to avoid accidental time travel between hardware timing boundaries.
- HW execution is modeled as concurrent asynchronous events, using publish/subscribe as the fundamental building block of composition (distributed state, à la actors).

# The problem

Applications composed of decoupled components, connected at runtime, lead to a callback nightmare -- e.g. how do you statically follow data-flow through the system?

# Our solution

Use the static information we have to reconstruct the decoupled callgraph.

Publish/subscribe library runtime connections are based on static information:

- Connections are type safe.
- Strong emphasis on connecting events to state transitions means little usage of dynamic string creation.

*Realization: we have enough static information to re-create a version, or multiple versions of the dynamic data-flow.*

# Basics

publish/subscribe communication is connected through a Registrar of connections.

Consumer:

```
1 //          signature          name
2 //          €                  €
3 reg.lookup<void(std::string)>("print channel").hook([] (const std::string &s) {
4     printf("%s", s.c_str());
5 });
```

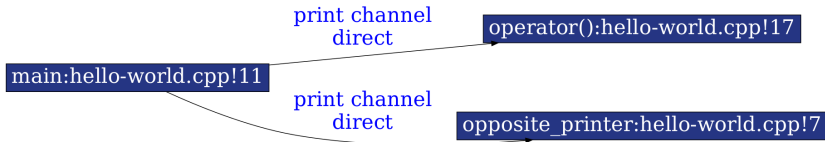
Producer:

```
1 //          signature          name
2 //          €                  €
3 auto print_channel = reg.lookup<void(std::string)>("print channel");
4 // deliver message now
5 print_channel("hello, world\n");
6 // deliver message in 1 cycle
7 // €
8 sched(1, print_channel, "hello, world\n");
```

# Example

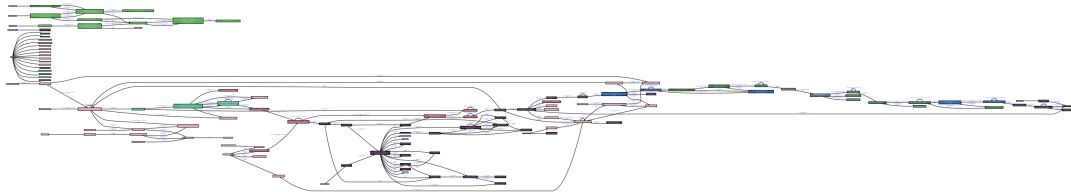
```
1 void opposite_printer(const std::string &s) {
2     std::cout << (s == "hello" ? "world" : "hello") << '\n';
3 }
4
5 int main() {
6     conduit::Registrar reg("reg");
7
8     // producer
9     auto print_channel = reg.lookup<void(std::string)>("print channel", "print_channel producer");
10
11    // first consumer
12    print_channel.hook([] (const std::string &s) {
13        std::cout << s << '\n';
14    });
15
16    // second consumer
17    print_channel.hook(opposite_printer);
18
19    print_channel("hello");
20 }
```

# Reconstructing the decoupled call-graph



```
print_channel producer -> reg.print channel(hello)
hello
world
```

# Real example





# Information recognized by the static analyzer

## Idioms:

- Synchronous and asynchronous connections between components.
- Concurrent state collection (events may happen 0 to N times).
- Channel merge (wait for N different events before triggering).
- Comment processing to allow better semantic descriptions of execution elements.

## State information:

- Non-const data members used in hook call-tree.

# Benefits

- Provides programmers another level of abstraction with which to describe the problem.
  - This is now part of our "modelers contract": descriptive problem decomposition must be reflected through the static analysis (used to bridge the gap between software model and HW implementer).
- Reinforces event  $\rightarrow$  state relationship.
- Helps identify concurrent data races.

# Conclusion

Static analysis combined with programming convention allows reconstruction of data-flow across asynchronous boundaries.



# Thank you

Follow us on: [f](#) [t](#) [in](#) [@](#)

For more information, visit us at:

[www.qualcomm.com](http://www.qualcomm.com) & [www.qualcomm.com/blog](http://www.qualcomm.com/blog)

Nothing in these materials is an offer to sell any of the components or devices referenced herein.

©2018 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

Qualcomm is a trademark of Qualcomm Incorporated, registered in the United States and other countries. Other products and brand names may be trademarks or registered trademarks of their respective owners.

References in this presentation to "Qualcomm" may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes Qualcomm's licensing business, QTL, and the vast majority of its patent portfolio. Qualcomm Technologies, Inc., a wholly-owned subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of Qualcomm's engineering, research and development functions, and substantially all of its product and services businesses, including its semiconductor business, QCT.