Scheduling of CAL actor networks based on dynamic code analysis

Mickaël Raulet
INSA Rennes, France

Jani Boutellier, Olli Silvén
University of Oulu, Finland

Motivation

- Describing computer programs can be done at different levels of abstraction

Assembly | C | RVC-CAL | Functional languages

low — level of abstraction — high

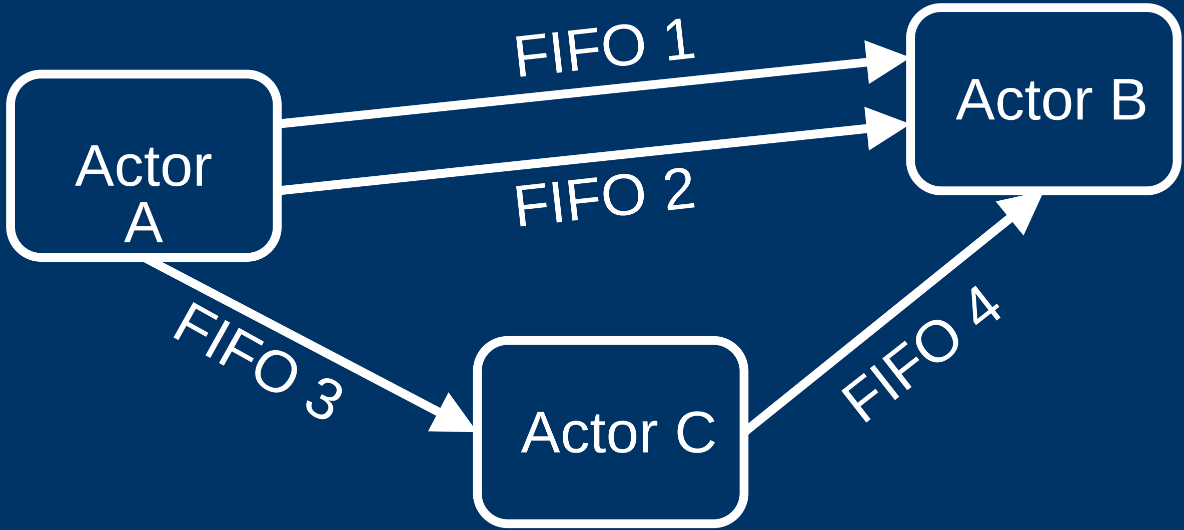
implementation efficiency

reusability
analyzability
verifiability
productivity

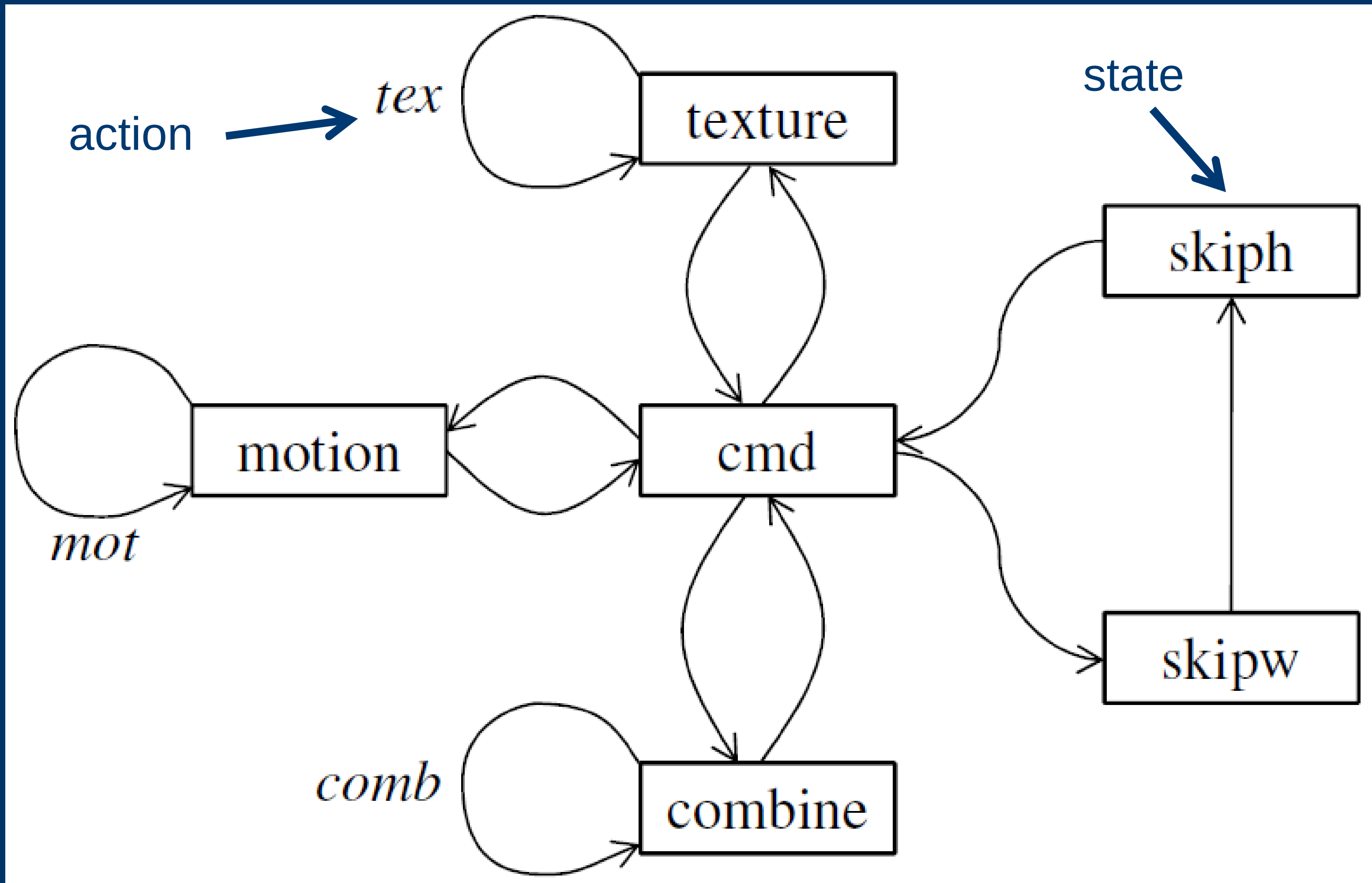
The RVC-CAL language

- A dataflow language that is a subset of the CAL language originally developed at UC Berkeley
- The RVC-CAL language has been standardized by ISO (ISO/IEC23001-4) in 2009

The RVC-CAL language



The RVC-CAL language



The RVC-CAL language

The main differences between the RVC-CAL and traditional dataflow models of computation:

•Allows conditional execution

+ Makes the language applicable to a wider set of applications

- Makes the language harder to analyze for humans and compilers

Topic of this work

The main point of our work is to improve the efficiency of programs written in RVC-CAL

Method of this work

- In RVC-CAL, each dataflow actor runs completely independently
- Basically this is good, as it improves the modularity of the language
- In practice, the actors within a program are very dependent on each other's behaviour
- We try to automatically discover these interdependencies and optimize the implementation with this information

Method of this work

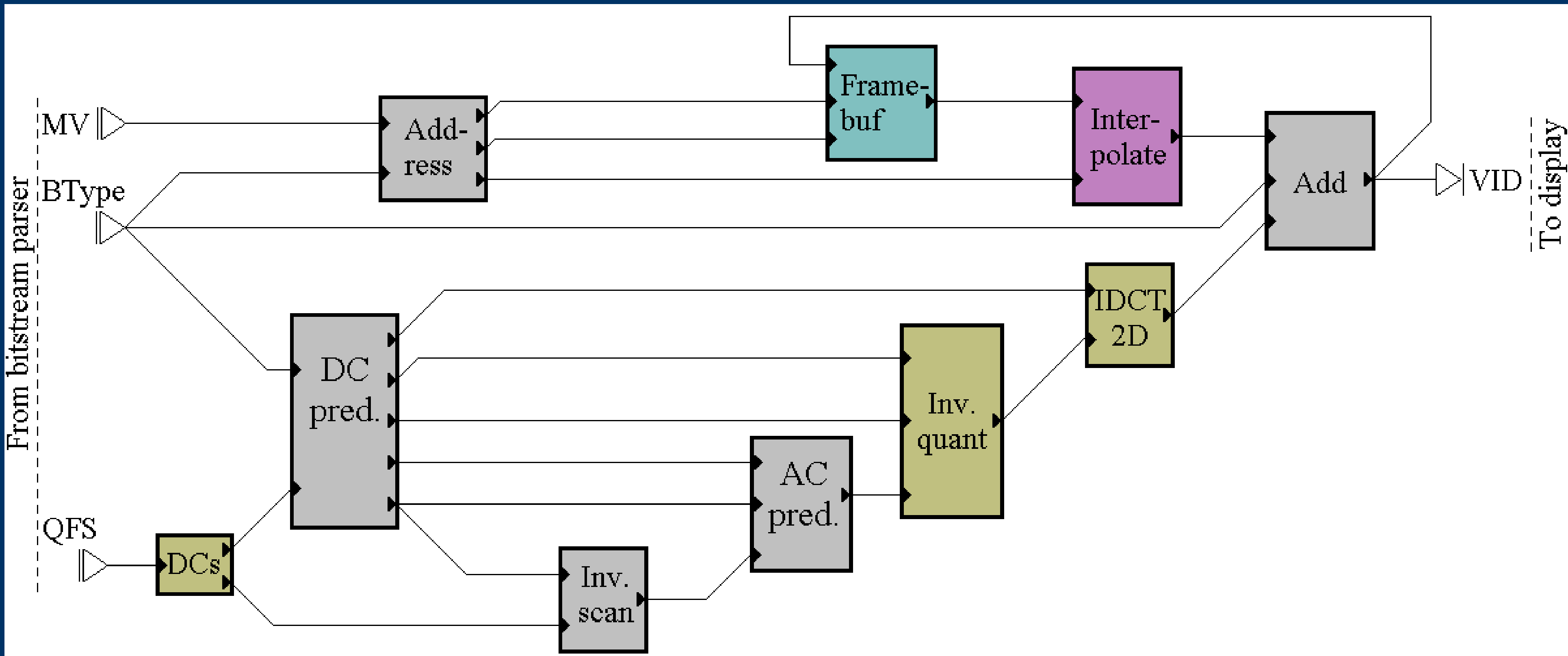
- Our approach is based on *dynamic program analysis*
- In dynamic analysis the behaviour of the program is examined *as it is running*
- Based on information acquired from analysis, a new, more efficient version of the program can be generated

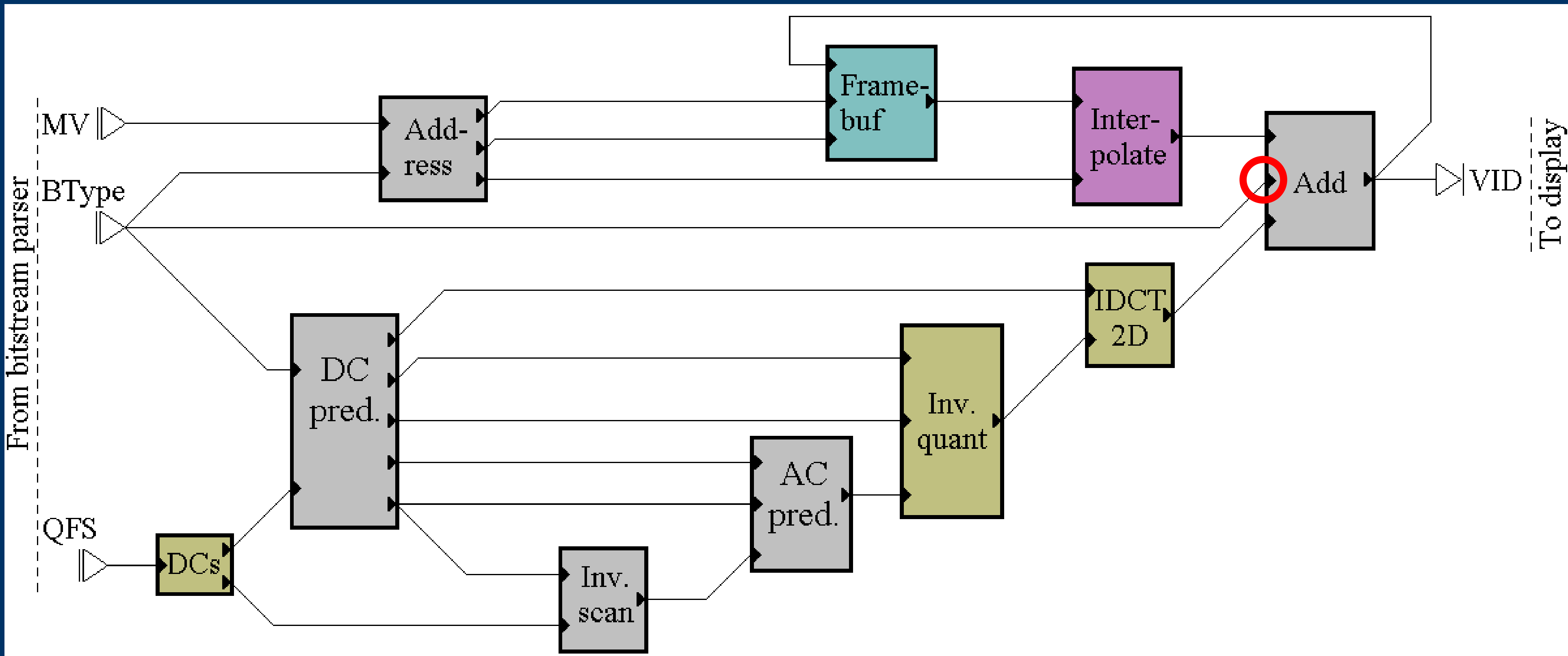
1. **Finding the data dependencies**
2. **Detecting the strands**
3. **Detecting the actor signatures**
4. **Code generation**

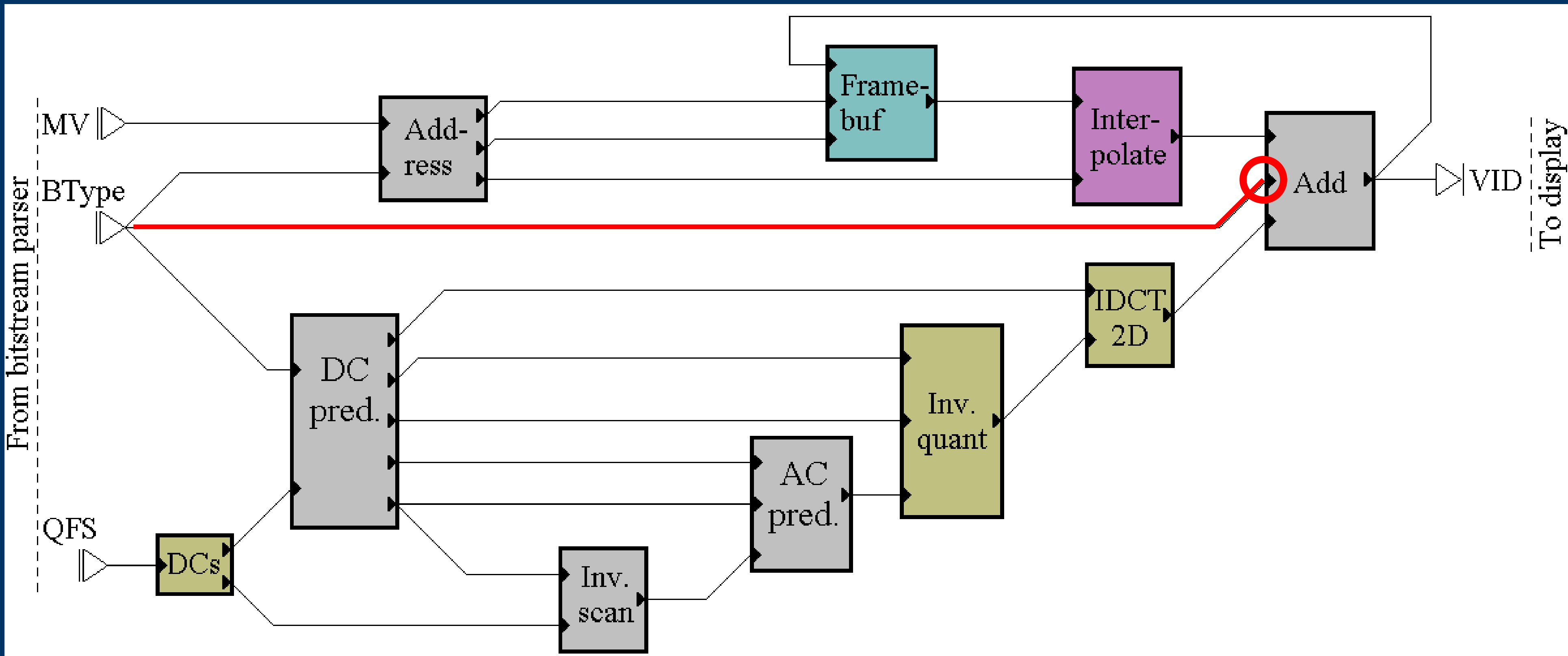
Finding the data dependencies

- The first step in our approach is to automatically find the signals in the network that cause conditional execution (control signals)
- The detection rule for these signals is

> If the value of data incoming from FIFO *f* affects the behaviour of an actor, *f* is a control signal


From bitstream parser
MV
BType
QFS
Add-
ress
Frame-
buf
Inter-
polate
Add
VID
To display
DC
pred.
IDCT
2D
Inv.
quant
AC
pred.
DCs
Inv.
scan


From bitstream parser
MV
BType
QFS
Add-
ress
Frame-
buf
Inter-
polate
Add
VID
To display
DC
pred.
IDCT
2D
Inv.
quant
AC
pred.
DCs
Inv.
scan

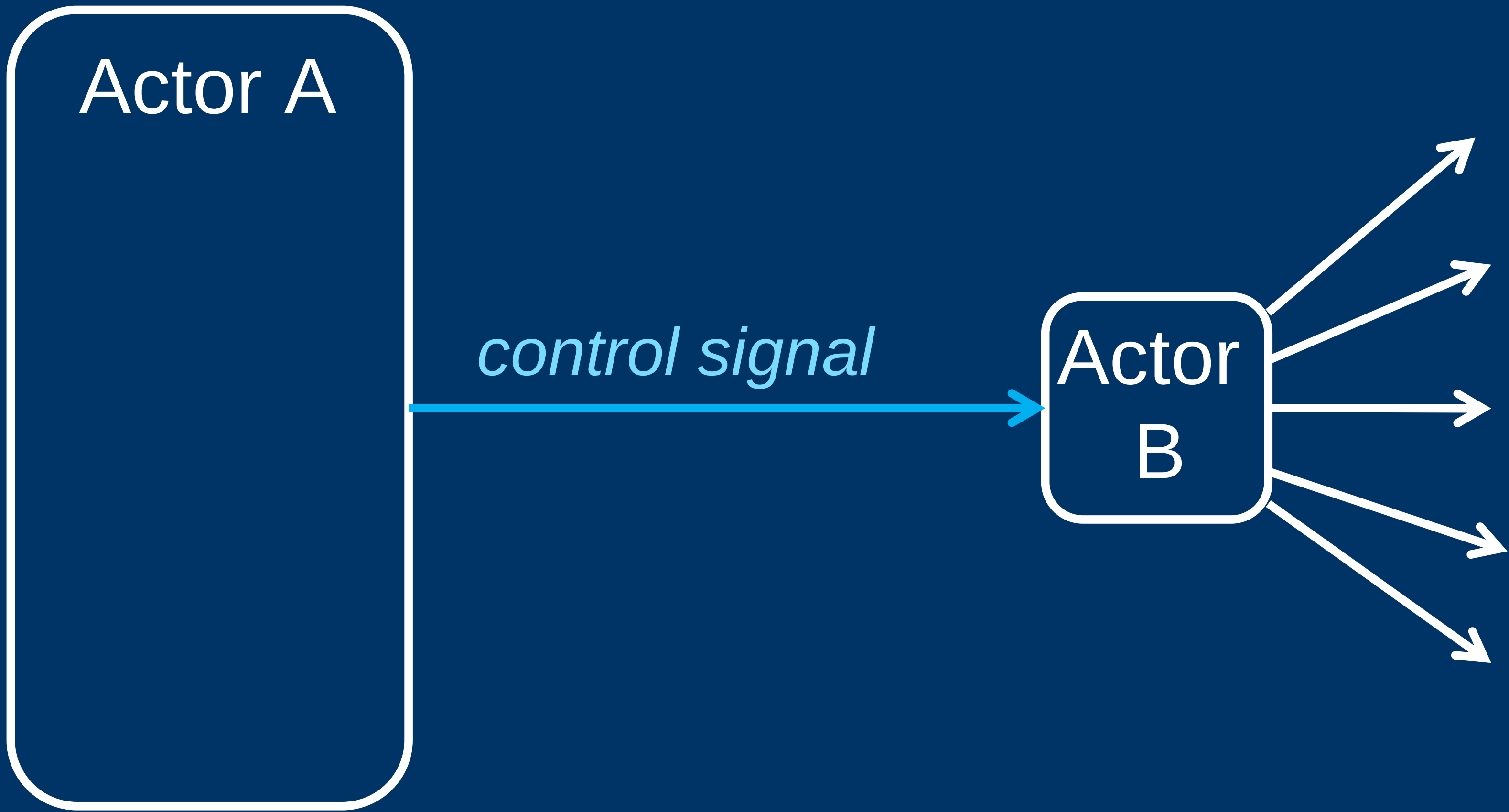

From bitstream parser
MV
BType
QFS
Add-
ress
Frame-
buf
Inter-
polate
Add
VID
To display
IDCT
2D
DC
pred.
Inv.
quant
AC
pred.
DCs
Inv.
scan

1. Finding the data dependencies
2. **Detecting the strands**
3. Detecting the actor signatures
4. Code generation

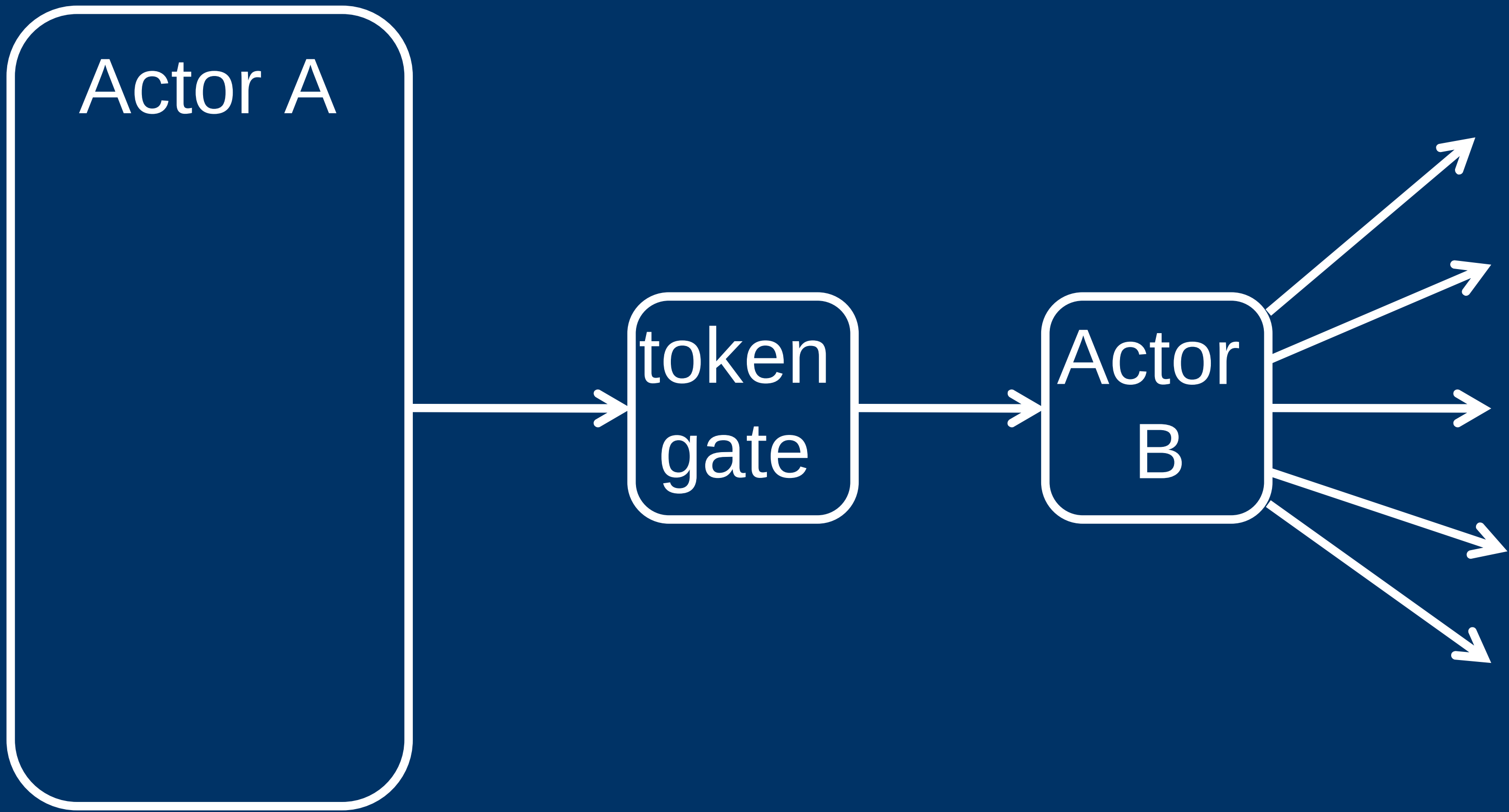
Detecting the strands

- Knowing the control signals in the actor network, we want to express the behaviour of the network as a function of the control signal tokens
- To be able to observe and control the network behaviour, we insert special actors named *token gates* to control signals

Detecting the strands



Detecting the strands



Detecting the strands

- A *strand* is a sequence of actor invocations. Each *value* coming through the token gate invokes 1 strand at run-time
- The strands can be detected automatically with the help of token gating:

1) Let a token throught the gate and observe its value
2) record the set of actors that it invokes

Detecting the strands

- However, this is not enough
- Generally, actors can behave in many different ways for each value passing through the token gate
- Therefore, we also need to find all the different actor behaviours for each strand

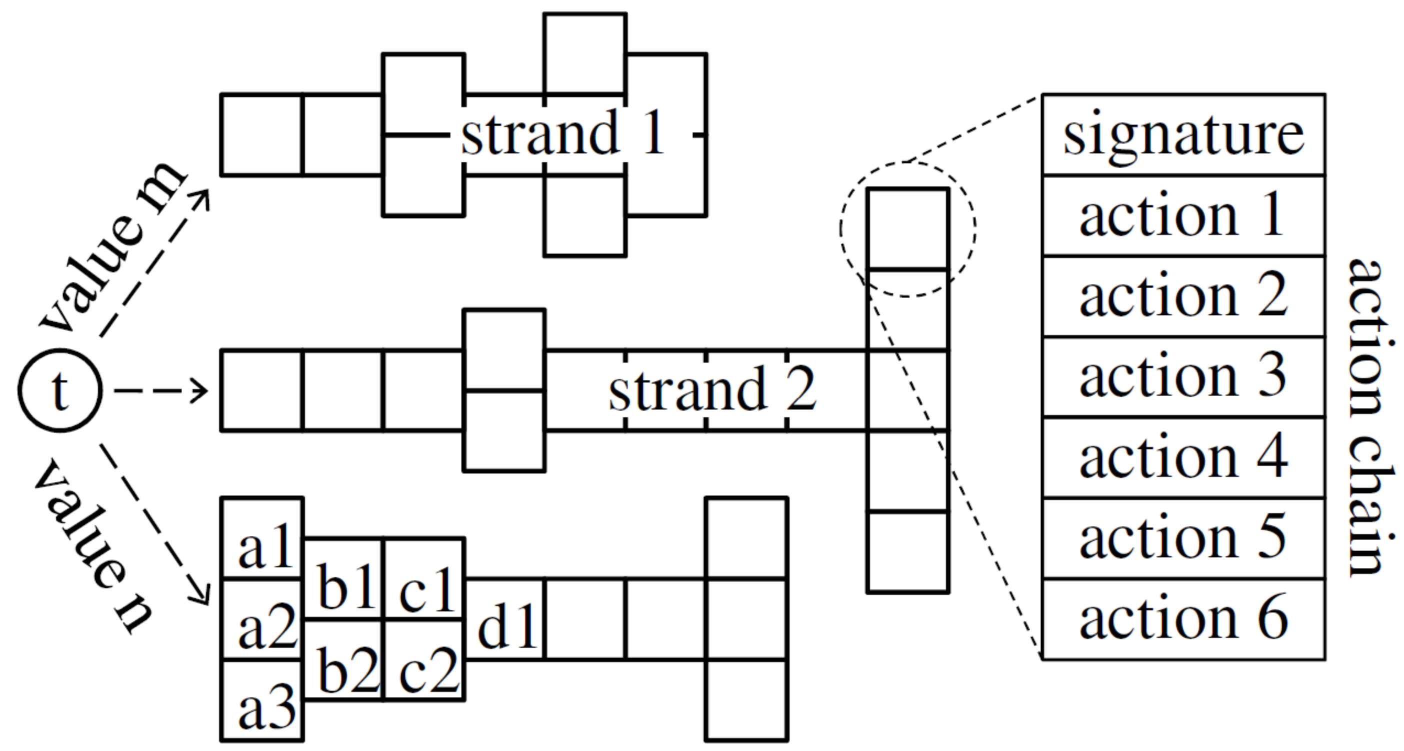
1. Finding the data dependencies
2. Detecting the strands
3. **Detecting the actor signatures**
4. Code generation

Detecting the actor signatures

- The behaviour of a CAL actor can be fully predicted *before* its execution by looking at the following properties
  1. Values of state variables
  2. Number of tokens at input ports
  3. Value of tokens at input ports
- These form the *signature* of the actor

Detecting the actor signatures

- At the network analysis stage, the actor signatures are recorded before letting the actor execute
- For every signature, the sequence of executed actions is recorded



t
value m
value n
strand 1
strand 2
a1
a2
a3
b1
b2
c1
c2
d1
signature
action 1
action 2
action 3
action 4
action 5
action 6
action chain

1. Finding the data dependencies
2. Detecting the strands
3. Detecting the actor signatures
4. **Code generation**

Code generation

- Now we have modeled the functionality of the application with gate token values and actor signatures
- Next, we generate the C code of a token-gated run-time program
- Make a switch-statement for each token gate value and signature

Results

MPEG-4 part 2 decoders

”MVG”	2.11x speedup
”RVC”	1.14x speedup
”Serial”	1.33x speedup
”Xilinx”	1.20x speedup

Conclusion

- We have presented a fully automated approach to speed up implementations of programs written in RVC-CAL
- The average speedup provided by our approach is 1.5x on the used set of RVC-CAL networks

Directions for future work

- Based on the lessons learned from the dynamic analysis approach, a static analysis approach could be developed
- Improving the code generation would provide better speedups
- Make the method applicable to programs with several data dependencies

Thanks for your attention.

Questions?