AFF3CT a DSEL dedicated to Real-time Streaming Applications on Multi-core CPUs

Sorbonne Université – Master SESI – MU5IN60 – Parallel Programming

Adrien Cassagne

October 16, 2023

Table of Contents

What is a Streaming Application?

- **Characteristics** of streaming applications
  - Large streams of data (= **virtually infinite sequence of input data**)
  - Some **data independent** batches of task
  - **Stable computation pattern** (almost the same computations for each stream)
  - Occasional modification of the stream structure
  - **High performance expectations**

What is a Streaming Application?

- **Characteristics** of streaming applications
  - Large streams of data (= **virtually infinite sequence of input data**)
  - Some **data independent** batches of task
  - **Stable computation pattern** (almost the same computations for each stream)
  - Occasional modification of the stream structure
  - **High performance expectations**
- Examples of streaming applications
  - Media applications (OSI 6-7): **audio & video processing** in general
    - **Deep Neural Networks** (DNNs): models have a fixed pattern of layers
  - Network (OSI 3): **software routers**
  - Software-defined radio (OSI 1): smartphone base stations (**cloud-RAN**)

What is a Streaming Application?

- **Characteristics** of streaming applications
  - Large streams of data (= **virtually infinite sequence of input data**)
  - Some **data independent** batches of task
  - **Stable computation pattern** (almost the same computations for each stream)
  - Occasional modification of the stream structure
  - **High performance expectations**
- Examples of streaming applications
  - Media applications (OSI 6-7): **audio & video processing** in general
    - **Deep Neural Networks** (DNNs): models have a fixed pattern of layers
  - Network (OSI 3): **software routers**
  - Software-defined radio (OSI 1): smartphone base stations (**cloud-RAN**)
- Examples that are **NOT** streaming applications
  - **Image processing**: because there is only one stream (= one frame)
  - Direct/indirect **linear algebra solvers**
  - In general: **iterative schemes where the result from $t-1$ is the input at t**

What is a DSEL?

- Domain Specific Language (DSL)
  - Language **designed specifically for a class of applications**

What is a DSEL?

- Domain Specific Language (DSL)
  - Language **designed specifically for a class of applications**
  - Pros:
    - **Very well suited** for a class of applications
    - Efficient way to write new program
    - Can be **specifically optimized**, generally outperform standard compilers
  - Cons:
    - Need to **re-write existing application** into the new language
    - Complicated when trying to do something that the language has not been made for

What is a DSEL?

- Domain Specific Language (DSL)
  - Language **designed specifically for a class of applications**
  - Pros:
    - **Very well suited** for a class of applications
    - Efficient way to write new program
    - Can be **specifically optimized**, generally outperform standard compilers
  - Cons:
    - Need to **re-write existing application** into the new language
    - Complicated when trying to do something that the language has not been made for
- Domain Specific Embedded Language (DSEL)
  - **Embedded into an other main language**

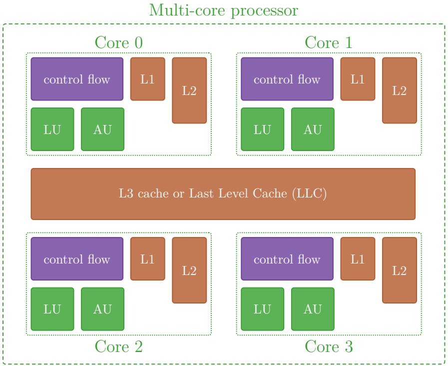
What is a DSEL?

- Domain Specific Language (DSL)
  - Language **designed specifically for a class of applications**
  - Pros:
    - **Very well suited** for a class of applications
    - Efficient way to write new program
    - Can be **specifically optimized**, generally outperform standard compilers
  - Cons:
    - Need to **re-write existing application** into the new language
    - Complicated when trying to do something that the language has not been made for

- Domain Specific Embedded Language (DSEL)
  - **Embedded into an other main language**
  - Pros:
    - **Simplify the porting** of applications from the main language to the DSEL
    - Take advantage of general compilers efficiency
    - **Possible workaround solutions** for unexpected usage
  - Cons:
    - Not as "pure" and "beautiful" as a DSL... **over patching syndrome**

What is a Multi-core CPU?

- Characteristics
  - Programmable ALUs
  - Grouped into cores
  - Memory affinities



What is a Multi-core CPU?

- Characteristics
  - Programmable ALUs
  - Grouped into cores
  - Memory affinities
- Well-spread
  - (Super-)computers
  - Smartphones
  - Embedded devices
  - ➔ Low cost for high performance!

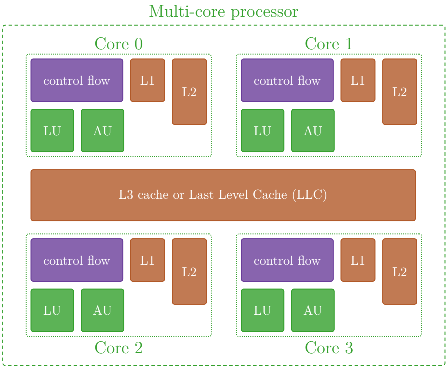


Table of Contents

Motivations

- Need for environment adapted to **streaming applications**:
  - Take advantage of multi-core architectures
  - High throughput & low latency (CPUs preferred to GPUs)
  - Manage energy consumption

Motivations

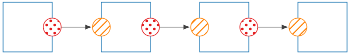
- Need for environment adapted to **streaming applications**:
  - Take advantage of multi-core architectures
  - High throughput & low latency (CPUs preferred to GPUs)
  - Manage energy consumption
- Proposed solution:
  - C++ Domain Specific Embedded Language (DSEL)
  - **Interpreted language**, meta-programming technique is avoided
  - **Synchronous DataFlow** (SDF) model but **stateful**

Sockets, Tasks, Modules & Sequence

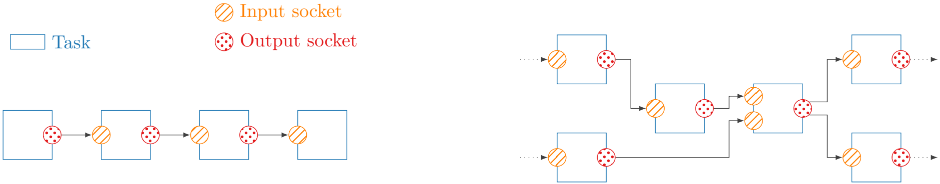
Task

Input socket

Output socket

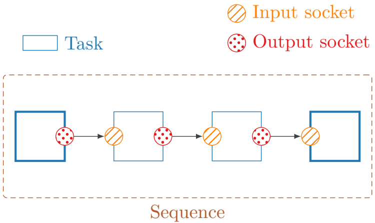


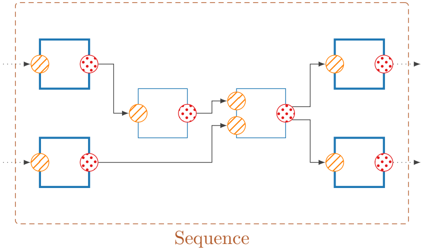
Sockets, Tasks, Modules & Sequence



- **Directed graphs** are supported **to map a wide range of apps**

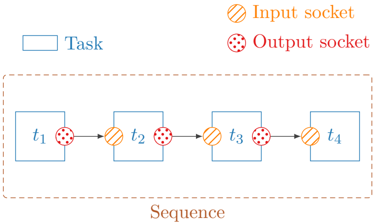
Sockets, Tasks, Modules & Sequence

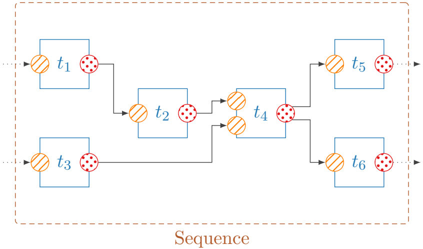




- **Directed graphs** are supported **to map a wide range of apps**
- A *sequence* is built from an initial and a final list of tasks

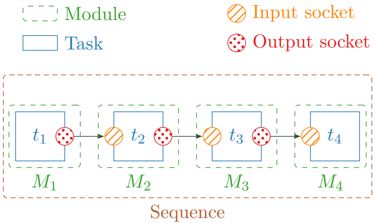
Sockets, Tasks, Modules & Sequence

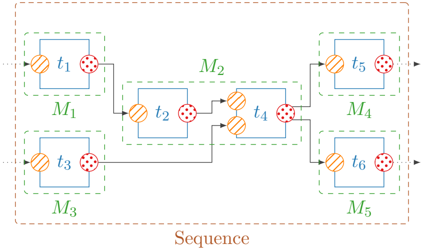




- **Directed graphs** are supported **to map a wide range of apps**
- A *sequence* is built from an initial and a final list of tasks
- Tasks execution order (scheduling) is determined by the user binding

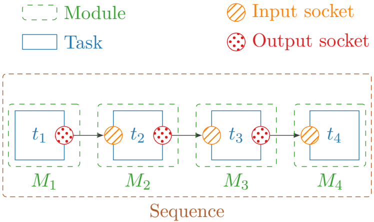
Sockets, Tasks, Modules & Sequence

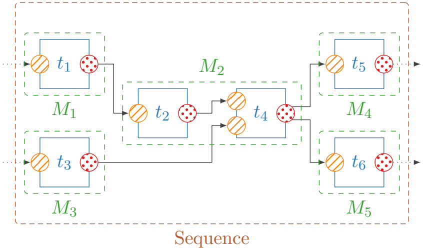




- **Directed graphs** are supported **to map a wide range of apps**
- A *sequence* is built from an initial and a final list of tasks
- Tasks execution order (scheduling) is determined by the user binding
- **States** are contained in *modules* (= C++ classes)

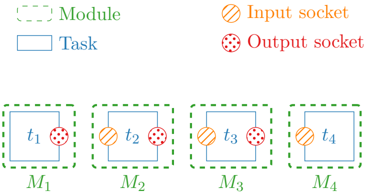
Sockets, Tasks, Modules & Sequence





- **Directed graphs** are supported **to map a wide range of apps**
- A *sequence* is built from an initial and a final list of tasks
- Tasks execution order (scheduling) is determined by the user binding
- **States** are contained in *modules* (= C++ classes)
- One task execution is enough to run dependent tasks (**single rate SDF**)

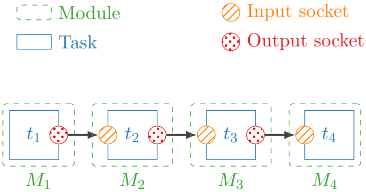
Simple Code Example




```
// 1) create the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4();
```

- On a module:
  - Use `("task_name")` functor to select a task
  - Use `["task_name::socket_name"]` operator to pick up a socket of a given task

Simple Code Example

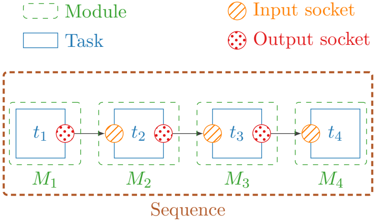



```
// 1) create the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4();

// 2) bind the tasks
m2["t2::in"] = m1["t1::out"];
m3["t3::in"] = m2["t2::out"];
m4["t4::in"] = m3["t3::out"];
```

- On a module:
  - Use ("task_name") functor to select a task
  - Use ["task_name::socket_name"] operator to pick up a socket of a given task

Simple Code Example

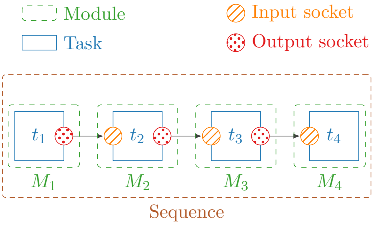
2 Domain Specific Embedded Language




```
// 1) create the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4();

// 2) bind the tasks
m2["t2::in"] = m1["t1::out"];
m3["t3::in"] = m2["t2::out"];
m4["t4::in"] = m3["t3::out"];

// 3) create the sequence (stop
//    automatically at t4 task)
runtime::Sequence seq(m1("t1"));
```

- On a module:
  - Use `("task_name")` functor to select a task
  - Use `["task_name::socket_name"]` operator to pick up a socket of a given task

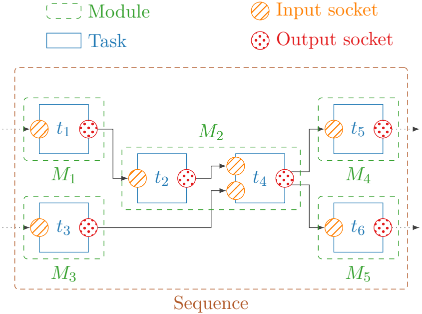
Simple Code Example



- On a module:
  - Use `("task_name")` functor to select a task
  - Use `["task_name::socket_name"]` operator to pick up a socket of a given task

```
// 1) create the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4();

// 2) bind the tasks
m2["t2::in"] = m1["t1::out"];
m3["t3::in"] = m2["t2::out"];
m4["t4::in"] = m3["t3::out"];

// 3) create the sequence (stop
//    automatically at t4 task)
runtime::Sequence seq(m1("t1"));

// 4) execute the sequence (tasks
//    graph is executed 100 times)
unsigned int exe_counter = 0;
seq.exec([&exe_counter]() {
  return ++exe_counter >= 100;
});
```

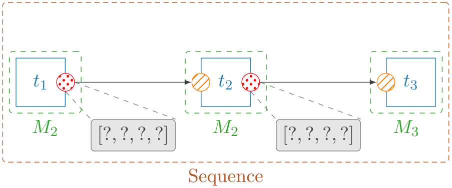
Graph Code Example



- Tasks are added to the graph in a **depth first traversal order**

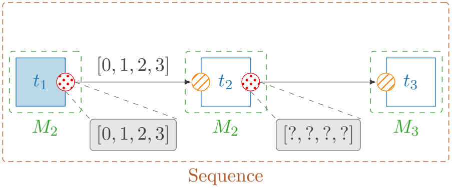
```cpp
// 1) create the module objects
M1 m1(); /* ... */ M7 m7();
std::vector<TYPE> some_data(SIZE);
// 2) bind the tasks
m1["t1::in" ] = some_data;
m3["t3::in" ] = some_data;
m2["t2::in" ] = m1["t1::out"];
m2["t4::in0"] = m2["t2::out"];
m2["t4::in1"] = m3["t3::out"];
m4["t5::in" ] = m2["t4::out"];
m5["t6::in" ] = m2["t4::out"];
m6["t7::in" ] = m4["t5::out"];
m7["t8::in" ] = m5["t6::out"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &m1("t1"), &m3("t3") },
  last  = { &m4("t5"), &m5("t6") };
runtime::Sequence seq(first, last);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

Memory Allocations & Forward Socket



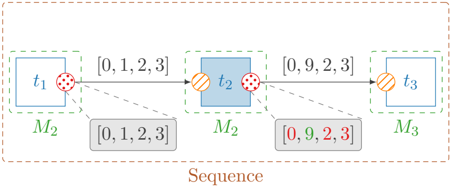
- Data are automatically allocated in the output sockets (see gray rectangles)

Memory Allocations & Forward Socket



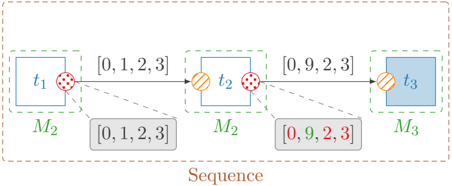
- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that t_2 only modify the second value of its input socket

Memory Allocations & Forward Socket



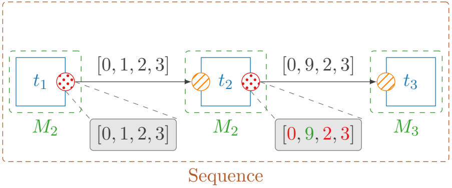
- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that t_2 only modify the second value of its input socket
  - "0", "2" and "3" are copied into t_2 output socket and "9" value replaces "1"

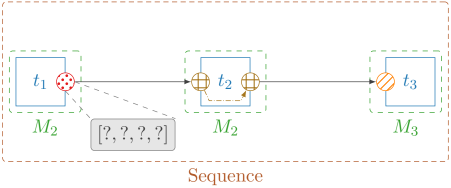
Memory Allocations & Forward Socket



- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that t_2 only modify the second value of its input socket
  - "0", "2" and "3" are copied into t_2 output socket and "9" value replaces "1"
  - This is highly inefficient!

Memory Allocations & Forward Socket

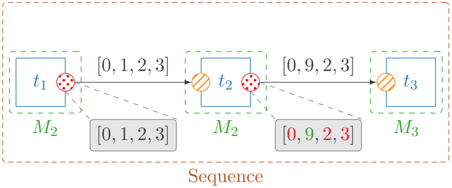


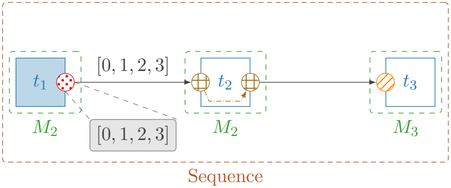


- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that t_2 only modify the second value of its input socket
  - "0", "2" and "3" are copied into t_2 output socket and "9" value replaces "1"
  - This is highly inefficient!

- Forward socket: at the same time an input and output socket (read+write)
  - There is NO data allocation

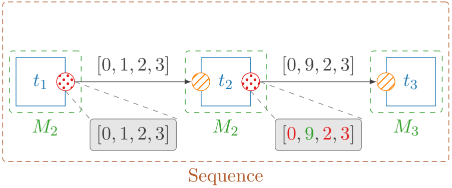
Memory Allocations & Forward Socket

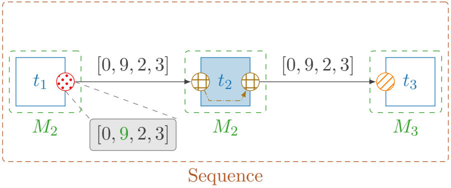




- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that t_2 only modify the second value of its input socket
  - "0", "2" and "3" are copied into t_2 output socket and "9" value replaces "1"
  - This is highly inefficient!
- Forward socket: at the same time an input and output socket (read+write)
  - There is NO data allocation
- We propose a new implementation of t_2 with a forward socket

Memory Allocations & Forward Socket

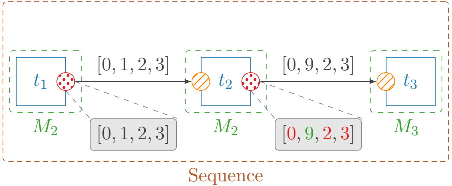


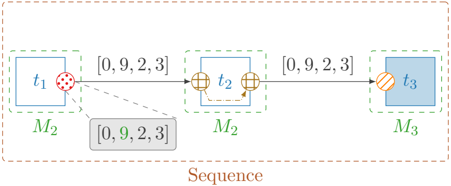


- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that t_2 only modify the second value of its input socket
  - "0", "2" and "3" are copied into t_2 output socket and "9" value replaces "1"
  - This is highly inefficient!

- Forward socket: at the same time an input and output socket (read+write)
  - There is NO data allocation
- We propose a new implementation of t_2 with a forward socket
  - t_1 output socket is modified in-place ("1" becomes "9")

Memory Allocations & Forward Socket

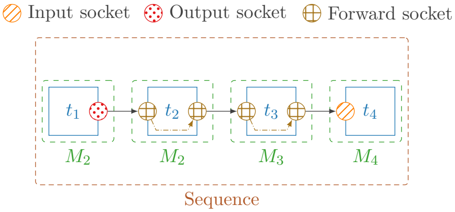




- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that t_2 only modify the second value of its input socket
  - "0", "2" and "3" are copied into t_2 output socket and "9" value replaces "1"
  - This is highly inefficient!

- Forward socket: at the same time an input and output socket (read+write)
  - There is NO data allocation
- We propose a new implementation of t_2 with a forward socket
  - t_1 output socket is modified in-place ("1" becomes "9")
  - This is efficient and cache-friendly!

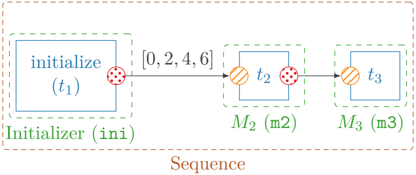
Forward Socket – Code Example



- Forward sockets are dangerous because they modify output socket values
  - The tasks execution order is very important because it can change the result!
- Here t_2 and t_3 have a forward socket
  - The same socket name (`fwd`) is used for input/output socket binding

```cpp
// 1) create the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4();

// 2) bind the tasks
m2["t2::fwd"] = m1["t1::out"];
m3["t3::fwd"] = m2["t2::fwd"];
m4["t4::in" ] = m3["t3::fwd"];

// 3) create the sequence (stop
//    automatically at t4 task)
runtime::Sequence seq(m1("t1"));

// 4) execute the sequence (tasks
//    graph is executed 100 times)
unsigned int exe_counter = 0;
seq.exec([&exe_counter]() {
  return ++exe_counter >= 100;
});
```

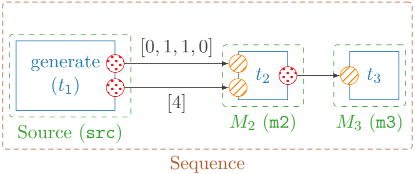
Basic Module – Initializer



- Always ouput the sames values
- Can be useful to initilize the memory
- `set_init_data` method is used to pass the data to the module (before the sequence execution)

```
// 1) create the module objects
Initializer<uint8_t> ini(4);
M2 m2(); M3 m3();
// 2) set init vals of the initializer
std::vector<uint8_t> init_data{0, 2, 4, 6};
ini.set_init_data(init_data)
// 3) bind the tasks
m2["t2::in" ] = ini["initialize::out"];
m3["t3::in" ] =  m2["t2::out"];
// 4) create the sequence
runtime::Sequence seq(ini("initialize"));
// 5) execute the sequence (no stop)
seq.exec([]() { return false; });
```

Basic Module – Source

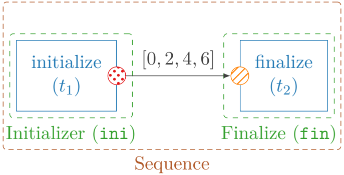



```
// 1) create the module objects
bool auto_reset = false;
Source_user_binary<uint8_t>
  src(4, "vid.mp4", auto_reset);
M2 m2(); M3 m3();
// 3) bind the tasks
m2["t2::in0"] = src["generate::out_data"];
m2["t2::in1"] = src["generate::out_count"];
m3["t3::in" ] = m2["t2::out"];
// 4) create the sequence
runtime::Sequence seq(src("generate"));
// 5) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- Available implementations:
  - `Source_user`: read data from an ASCII file
  - `Source_user_binary`: read data from a binary file
  - `Source_random`: generate 0 and 1 randomly
  - `Source_AZCW`: All Zeros Code Word → 0
- Read the bits 4 by 4 (frame size = 4)

- Outputs binary values (0 or 1)
- If `auto_reset = false` stops the sequence when EOF
- `out_count` socket returns # of read bits

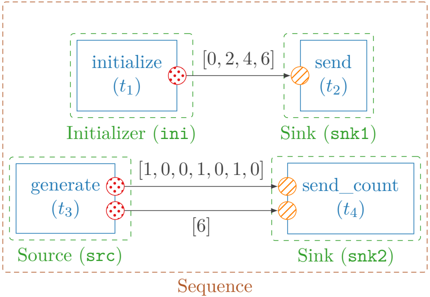
Basic Module – Finalizer



- Used to retrieve data from the graph
- Can be read
  - At the end of the sequence execution
  - At the end of a stream execution (see code example)

```cpp
// 1) create the module objects
Initializer<uint8_t> ini(4);
Finalizer<uint8_t> fin(4);
// 2) set init vals of the initializer
std::vector<uint8_t> init_data{0, 2, 4, 6};
ini.set_init_data(init_data)
// 3) bind the tasks
fin["finalize::in"] = ini["initialize::out"];
// 4) create the sequence
runtime::Sequence seq(ini("initialize"));
// 5) execute the sequence (10 times)
unsigned int exe_counter = 0;
seq.exec([&exe_counter]() {
  // get the date from the finalizer
  const std::vector<uint8_t> &final_data =
    fin.get_final_data()[0];
  // for each stream, print: "0,2,4,6,\n"
  for (const uint8_t &val : final_data)
    std::cout << val << ",";
  std::cout << std::endl;
  return ++exe_counter >= 10;
});
```

Basic Module – Sink

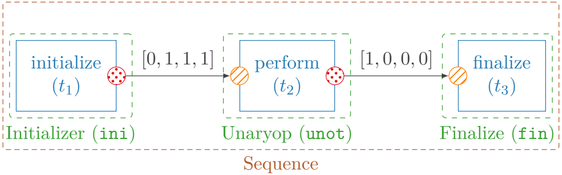


- Write data into a binary file
- Two available task variants
  - `send`: simply write the input socket
  - `send_count`: write only the # of bits given in the `in_count` input socket

```
// 1) create the module objects
Initializer<uint8_t> ini(4);
Source_user_binary<uint8_t>
  src(7, "vid.mp4", false);
Sink_user_binary<uint8_t> snk1(4, "out1.bin"),
                          snk2(7, "out2.bin");
// 2) set init vals of the initializer
ini.set_init_data({0, 2, 4, 6})
// 3) bind the tasks
snk1["send::in_data"] = ini["initialize::out"];
snk2["send_count::in_data"] = src["generate::out_data"];
snk2["send_count::in_count"] = src["generate::out_count"];
// 4) create the sequence
std::vector<runtime::Task*>
  first = { &ini("initialize"), &src("generate") },
runtime::Sequence seq(first);
// 5) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- `send_count` task: `in_data` socket contains 7 bits but only the 6 first bits are written

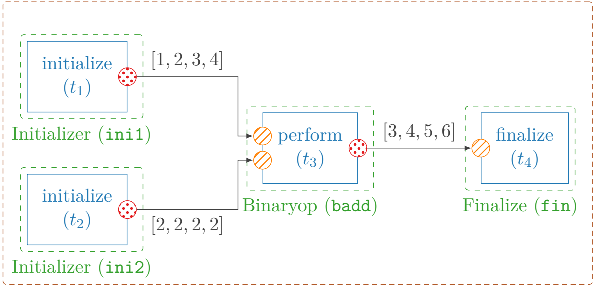
Basic Module – Unaryop




```cpp
// 1) create the module objects
Initializer<uint8_t> ini(4);
Unaryop_not<uint8_t> unot(4);
Finalizer<uint8_t> fin(4);
// 2) set init vals of the initializer
std::vector<uint8_t> init_data{0, 1, 1, 1};
ini.set_init_data(init_data)
// 3) bind the tasks
unot["perform::in"] = ini["initialize::out"];
fin["finalize::in"] = unot["perform::out"];
// 4) create the sequence
runtime::Sequence seq(ini("initialize"));
// 5) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- Operations involving one input data to produce one output
  - `Unaryop_abs`: absolute value
  - `Unaryop_not`: bitwise not (`~d` in C)
  - `Unaryop_not_abs`: `abs(~d)`
  - `Unaryop_sign`: return 1 if the number is negative, 0 otherwise (`d<0 ? 1 : 0`)

Basic Module – Binaryop



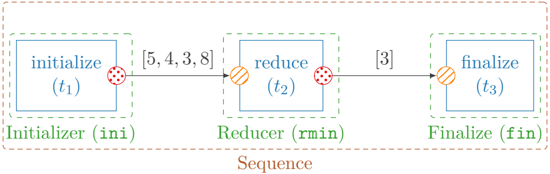
Sequence

```
// 1) create the module objects
Initializer<uint8_t> ini1(4), ini2(4);
Binaryop_add<uint8_t> bapp(4);
Finalizer<uint8_t> fin(4);
// 2) set init vals of the initializer
ini1.set_init_data({1, 2, 3, 4});
ini2.set_init_data({2, 2, 2, 2});
// 3) bind the tasks
badd["perform::in0"] = ini1["initialize::out"];
badd["perform::in1"] = ini2["initialize::out"];
fin["finalize::in"] = badd["perform::out"];
// 4) create the sequence
std::vector<runtime::Task*>
  first = { &ini1("initialize"),
            &ini2("initialize") },
runtime::Sequence seq(first);
// 5) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- Operations involving two inputs data to produce one output
  - Arithmetic: add, sub, mul, div, ...
  - Bitwise: and, or, xor, ...
  - Comparison: eq, neq, ge, le, ...

- Most common operations are supported!

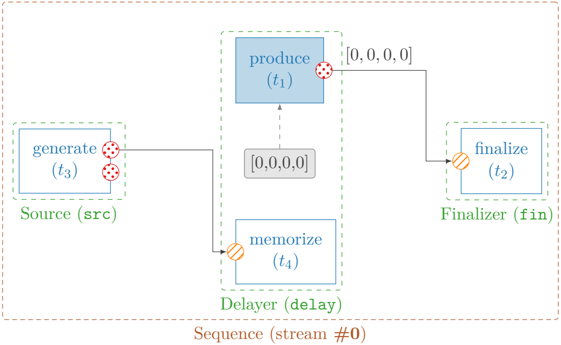
Basic Module – Reducer



- Operations involving a vector of inputs and produce one output
  - `Reducer_add`, `Reducer_mul`, `Reducer_or`, `Reducer_and`, `Reducer_min`, `Reducer_max`

```
// 1) create the module objects
Initializer<uint8_t> ini(4);
Reducer_min<uint8_t> rmin(4);
Finalizer<uint8_t> fin(1);
// 2) set init vals of the initializer
std::vector<uint8_t> init_data{5, 4, 3, 8};
ini.set_init_data(init_data)
// 3) bind the tasks
rmin["reduce::in"] = ini["initialize::out"];
fin["finalize::in"] = rmin["reduce::out"];
// 4) create the sequence
runtime::Sequence seq(ini("initialize"));
// 5) execute the sequence (no stop)
seq.exec([]() { return false; });
```

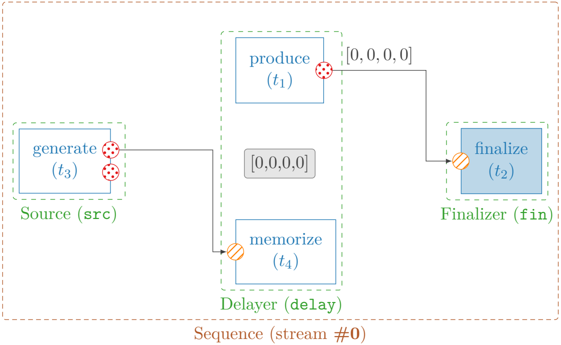
Basic Module – Delayer




```cpp
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

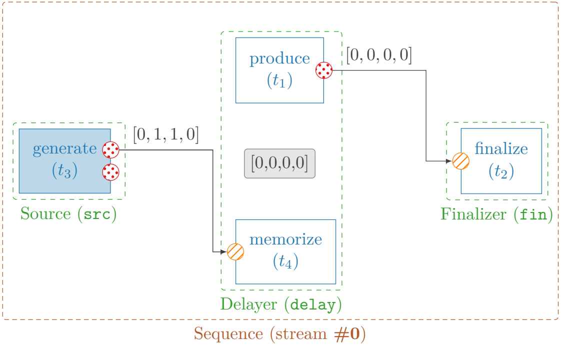
Basic Module – Delayer




```
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

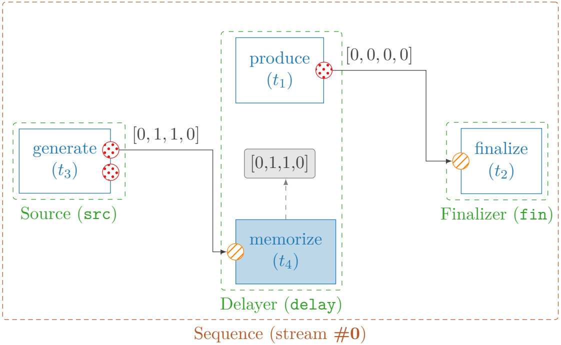
Basic Module – Delayer




```cpp
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

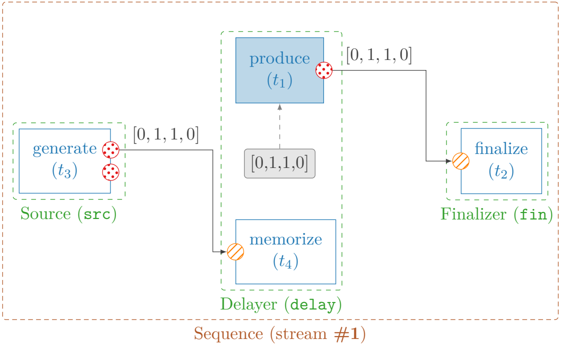
Basic Module – Delayer




```
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

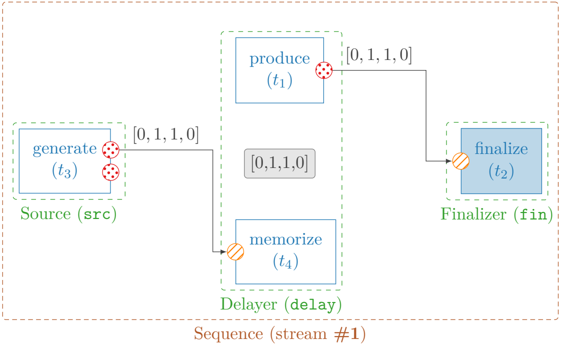
Basic Module – Delayer




```cpp
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

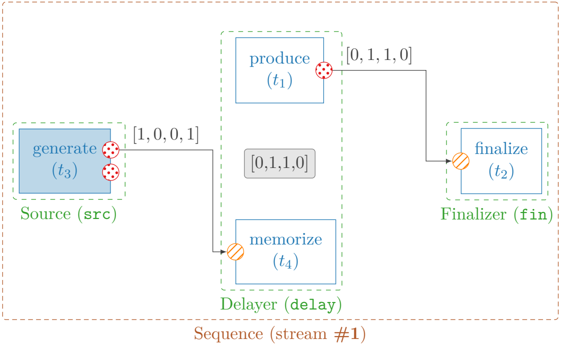
Basic Module – Delayer




```cpp
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

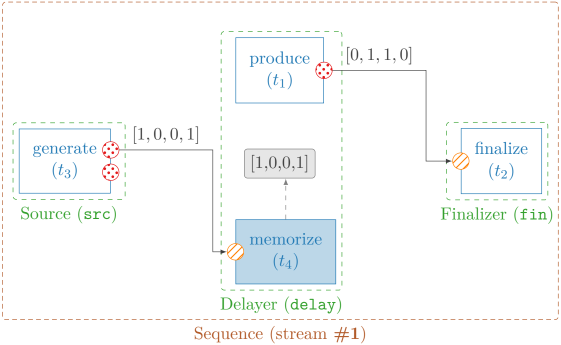
Basic Module – Delayer




```cpp
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

Basic Module – Delayer




```
// 1) create the module objects
Source_user_binary<uint8_t>
  src(4, "vid.mp4", false);
uint8_t init_val = 0; // buff val for stream #0
Delayer<uint8_t> delay(4, init_val);
Finalizer<uint8_t> fin(4);
// 2) bind the tasks
fin["finalize::in"] = delay["produce::out"];
delay["memorize::in"] = src["generate::out_data"];
// 3) create the sequence
std::vector<runtime::Task*>
  first = { &delay("produce"), &src("generate") },
runtime::Sequence seq(first);
// 4) execute the sequence (no stop)
seq.exec([]() { return false; });
```

- To memorize data from the previous stream
- `Delayer` module contains 2 tasks
  - `produce` needs to be called first
  - Then, `memorize` updates the shared buffer

Task and Module Implementation

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
```

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
```

Task and Module Implementation

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
```

Task and Module Implementation

Stateless task example (min. reduction):

```cpp
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
```

Task and Module Implementation

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    /* ... */
});
```

Task and Module Implementation

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
});
```

Task and Module Implementation

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
});
```

Task and Module Implementation

Stateless task example (min. reduction):

```cpp
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Stateful task example (basic counter):

```
class Counter : public Module {
  private:
  int cnt; // inner data => stateful
};
```

Task and Module Implementation

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Stateful task example (basic counter):

```
class Counter : public Module {
  private:
  int cnt; // inner data => stateful
  int get_value() { return this->cnt; } // RO method
  void increment() { this->cnt++; } // W method
};
```

Task and Module Implementation

2 Domain Specific Embedded Language

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Stateful task example (basic counter):

```
class Counter : public Module {
  private:
  int cnt; // inner data => stateful
  int get_value() { return this->cnt; } // RO method
  void increment() { this->cnt++; } // W method
  public:
  Counter() : Module(), cnt(0) { // constructor
    /* ... */
  }
};
```

Task and Module Implementation

2 Domain Specific Embedded Language

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Stateful task example (basic counter):

```
class Counter : public Module {
  private:
  int cnt; // inner data => stateful
  int get_value() { return this->cnt; } // RO method
  void increment() { this->cnt++; } // W method
  public:
  Counter() : Module(), cnt(0) { // constructor
    set_name("Counter");
    Task &t = create_tsk("get_val");
    size_t so = create_sck_out<int>(t, "o", 1);
    create_codelet(t,
      [so](Module &m, Task &tsk) -> int {
        /* ... */
    });
  }
};
```

Task and Module Implementation

2 Domain Specific Embedded Language

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Stateful task example (basic counter):

```
class Counter : public Module {
  private:
  int cnt; // inner data => stateful
  int get_value() { return this->cnt; } // RO method
  void increment() { this->cnt++; } // W method
  public:
  Counter() : Module(), cnt(0) { // constructor
    set_name("Counter");
    Task &t = create_tsk("get_val");
    size_t so = create_sck_out<int>(t, "o", 1);
    create_codelet(t,
      [so](Module &m, Task &tsk) -> int {
        int* po = tsk[so].get_dataptr<int>();
        // cast 'm' into 'Counter' class type
        Counter &mc = static_cast<Counter&>(m);
        // write value in the output socket
        *po = mc.get_value();
        mc.increment(); // increment internal counter
        return runtime::status_t::SUCCESS;
    });
  }
};
```

Task and Module Implementation

Stateless task example (min. reduction):

```
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Stateful task example (basic counter):

```
class Counter : public Module {
  private:
  int cnt; // inner data => stateful
  int get_value() { return this->cnt; } // RO method
  void increment() { this->cnt++; } // W method
  public:
  Counter() : Module(), cnt(0) { // constructor
    set_name("Counter");
    Task &t = create_tsk("get_val");
    size_t so = create_sck_out<int>(t, "o", 1);
    create_codelet(t,
      [so](Module &m, Task &tsk) -> int {
        int* po = tsk[so].get_dataptr<int>();
        // cast 'm' into 'Counter' class type
        Counter &mc = static_cast<Counter&>(m);
        // write value in the output socket
        *po = mc.get_value();
        mc.increment(); // increment internal counter
        return runtime::status_t::SUCCESS;
    });
  }
};
```

Wrap Existing C-like Code into a Module

- Let's assume we have the following C declarations

```c
// a structure used by the 'compute' function
typedef struct {
  const size_t size; // RO data size
  const float limit; // RO data
  float *tmp_array; // tmp data used in 'compute' func
} inner_data_t;

// 'compute' function takes inner data 'idat', read
// 'in_data' and produce 'out_data' , the size of
// 'in_data' and 'out_data' is contained in 'idat->size'
void compute(inner_data_t* idat, const float *in_data,
                                 int8_t *out_data);
```

- The code on the right shows how the C code can be wrapped in an module named `My_module`

```cpp
class My_module : public Module {
  private:
  inner_data_t idat; // inner data for 'compute' func
  public:
  My_module(const size_t size, const float limit)
  : Module(), idat({size, limit, new float[size]}) {
    set_name("My_module");
    Task &t = create_tsk("compute");
    size_t si =
      create_sck_in<float>(t, "in", this->idat.size);
    size_t so =
      create_sck_out<int8_t>(t, "out", this->idat.size);
    create_codelet(t,
      [si,so](Module &m, Task &tsk) -> int {
        My_module &m = static_cast<My_module&>(m);
        // call the C 'compute' function with its params
        compute(m.idat, tsk[si].get_dataptr<float>(),
                        tsk[so].get_dataptr<int8_t>());
        return runtime::status_t::SUCCESS;
    });
  }
  virtual ~My_module() { delete[] this->idat.tmp_array; };
};
```

Stop the Overall Execution from a Module


```cpp
class My_module : public Module,
                  public tools::Interface_is_done {
  private:
  inner_data_t idat; // inner data for 'compute' func
  public:
  My_module(const size_t size, const float limit)
  : Module(), idat({size, limit, new float[size]}) {
    set_name("My_module");
    Task &t = create_tsk("compute");
    size_t si = create_sck_in<float>(t, "in", this->idat.size);
    size_t so = create_sck_out<int8_t>(t, "out", this->idat.size);
    create_codelet(t,
      [si,so](Module &m, Task &tsk) -> int {
        My_module &m = static_cast<My_module&>(m);
        compute(m.idat, tsk[si].get_dataptr<float>(),
                        tsk[so].get_dataptr<int8_t>());
        return runtime::status_t::SUCCESS;
    });
    // stop if sum of vals in 'tmp_array' > limit
    virtual bool is_done() const {
      float sum = 0.f;
      for (auto &v : this->idat.tmp_array)
        sum += v;
      return sum > this->idat.limit;
    }
  }
};
```

- Need to inherit from the `tools::Interface_is_done` class
- And implement `is_done()` method (= interface function)
- At the end of a stream, the sequence automatically calls `is_done()` on all the modules
  - If at least one module returns `true`, then the sequence stops

Execute a Task Outside a Sequence


```cpp
int main(int argc, char** argv) {
  const size_t size = 10; const float limit = 100.f;
  My_module mymod(size, limit);

  std::vector<float> in_data(size);
  // initialize 'in_data' to [0,1,2,3,4,5,6,7,8,9]
  std::iota(init_data.begin(), init_data.end(), 0.f);
  std::vector<int8_t> out_data(size);

  // attach the previously allocated buffers to the
  // 'in' and 'out' sockets of the 'compute' task
  mymod["compute::in"].bind(in_data);
  mymod["compute::out"].bind(out_data);
  // execute the 'compute' function
  mymod("compute").exec();

  // display the result contained in the 'out_data'
  // vector
  std::cout << "out_data = [";
  for (auto v : out_data)
    std::cout << v << ",";
  std::cout << "]" << std::endl;

  return 0;
}
```

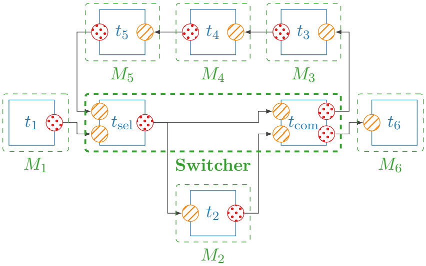
- A task can be interfaced with a “classic” C/C++ code
  - It can be useful to check if the module and the task have been correctly wrapped
  - Enable to execute the code even if all the functions are not wrapped into tasks

- The user must take charge of
  - Memory allocations (supports 1D C array and C++ `std::vector`)
  - Bindings of the allocated data onto the task’s sockets
  - Explicit task execution

Table of Contents

3 DSEL: Control Flow

While Loop

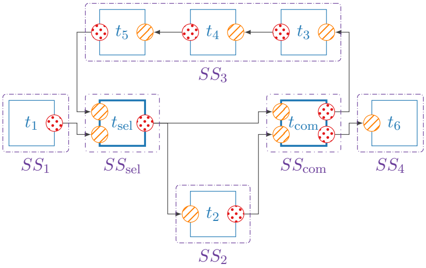


execute SS_1;
while execute SS_2 **and** **not** $t_{\text{com}}.\text{in}_2$ **do**
  execute SS_3;
execute SS_4;

- Construct block: the **Switcher** module

While Loop

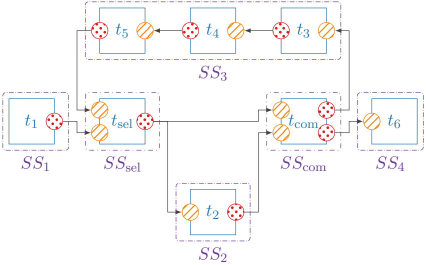
3 DSEL: Control Flow




```
execute SS_1;
while execute SS_2 and
  not t_com.in_2 do
    execute SS_3;
execute SS_4;
```

- Construct block: the **Switcher** module
  - *commute* task (t_{com}): **creates** exclusive execution paths
  - *select* task (t_{sel}): **joins** exclusive execution paths

While Loop

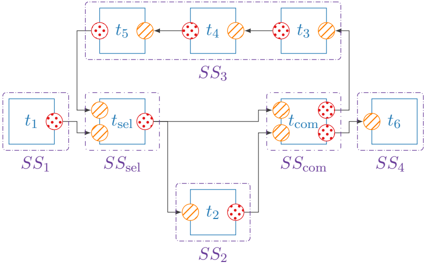


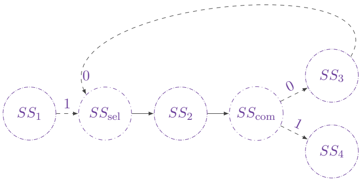
execute SS_1;
while execute SS_2 **and** **not** $t_{com}.\text{in}_2$ **do**
  execute SS_3;
execute SS_4;

- Construct block: the **Switcher** module
  - *commute* task (t_{com}): **creates** exclusive execution paths
  - *select* task (t_{sel}): **joins** exclusive execution paths

➔ Required in **turbo iterative receivers**

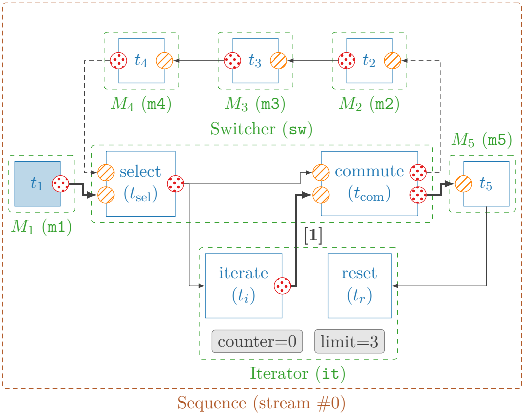
While Loop – Code Example





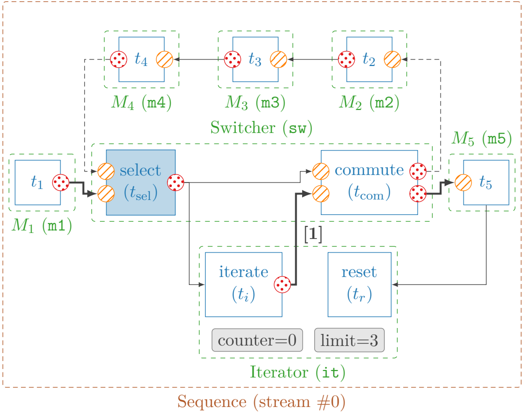

```
M1 m1(); /* ... */ M6 m6();
Switcher sw(2); // 2 exclusive paths

sw[ "select::in_data1"] = m1[      "t1::out"        ];
m2[      "t2::in"       ] = sw[ "select::data_out" ];
sw["commute::in_data" ] = sw[ "select::data_out" ];
sw["commute::in_ctrl" ] = m2[      "t2::out"        ];
// sub-seq. 3, executed if tcom::in2 = 0
m3[      "t3::in"       ] = sw["commute::data_out0"];
m4[      "t4::in"       ] = m3[      "t3::out"        ];
m5[      "t5::in"       ] = m4[      "t4::out"        ];
sw[ "select::in_data0"] = m5[      "t5::out"        ];
// sub-seq. 4, executed if tcom::in2 = 1
m6[      "t6::in"       ] = sw["commute::out_data1"];

Sequence seq(m1("t1"));
seq.exec([]() { return false; });
```

For Loop – Module Iterator



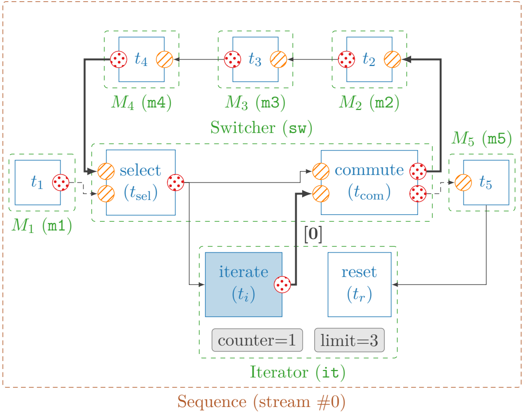

```
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5();
Switcher sw(2); // 2 exclusive paths
Iterator it(3); // 3 iterations in the loop

sw[ "select::in_data1"] = m1[     "t1::out"      ];
// special binding between a socket and a task
it("iterate")           = sw[ "select::data_out" ];
sw["commute::in_data" ] = sw[ "select::data_out" ];
sw["commute::in_ctrl" ] = it["iterate::out"      ];
m2[       "t2::in"    ] = sw["commute::out_data0"];
m3[       "t3::in"    ] = m2[     "t2::out"      ];
m4[       "t4::in"    ] = m3[     "t3::out"      ];
sw[ "select::in_data0"] = m4[     "t4::out"      ];
m5[       "t5::in"    ] = sw["commute::out_data1"];
// all the tasks have an implicit out 'status' socket
it(  "reset")           = m5[     "t5::status"   ];

Sequence seq(m1("t1"));
seq.exec([]() { return false; });
```

For Loop – Module Iterator



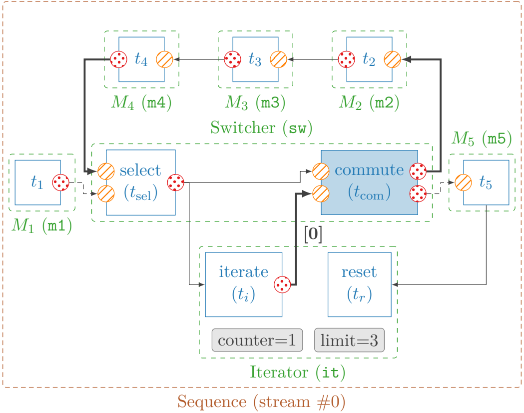
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1

For Loop – Module Iterator



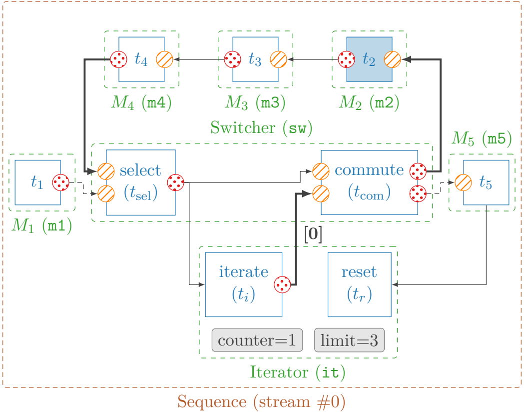
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



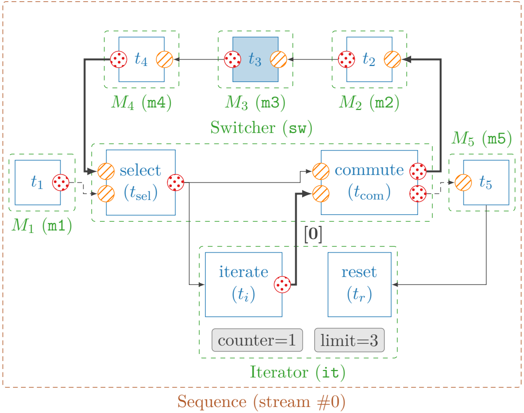
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



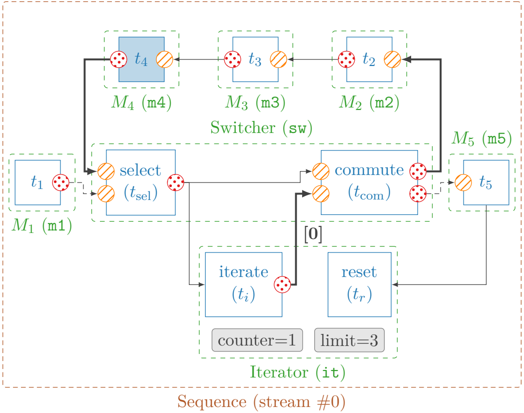
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



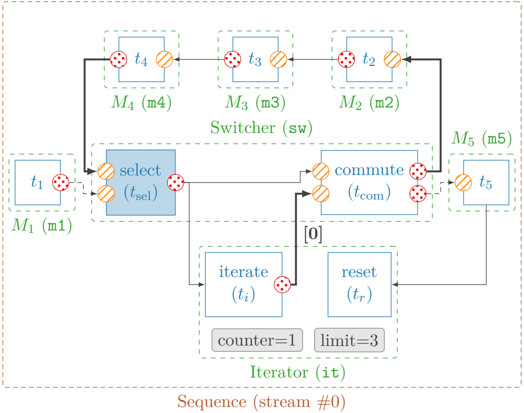
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



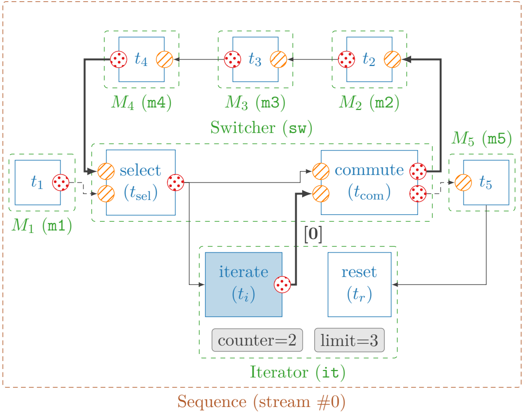
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



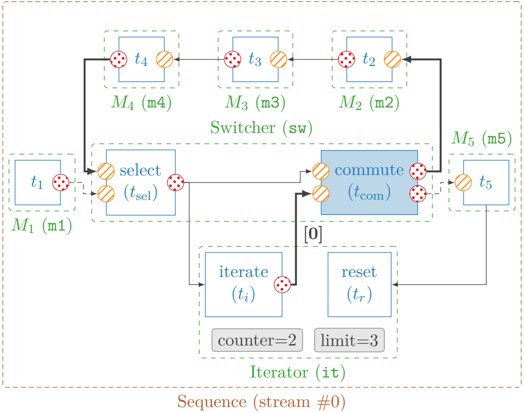
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



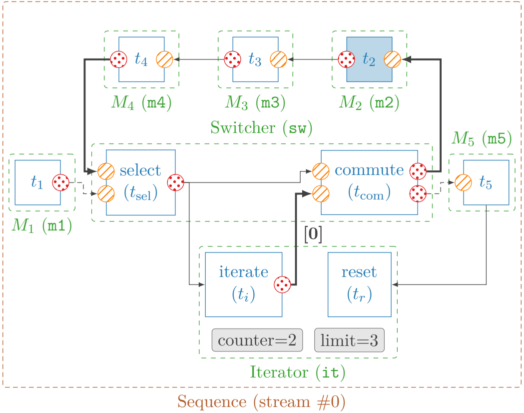
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



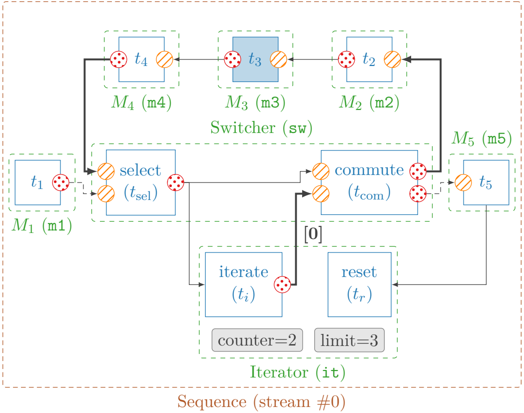
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator
3 DSEL: Control Flow



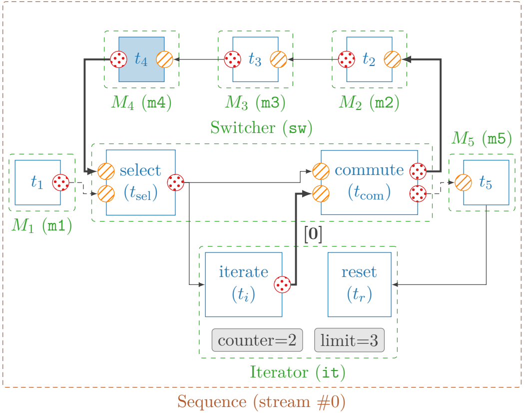
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



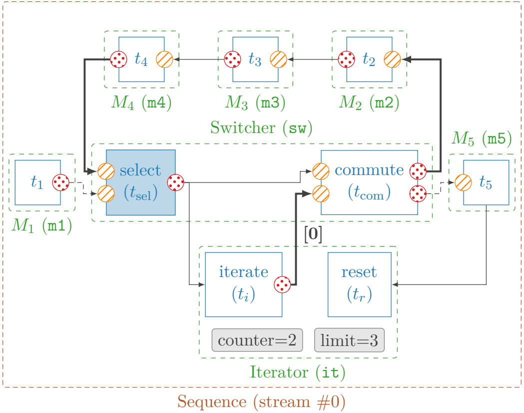
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



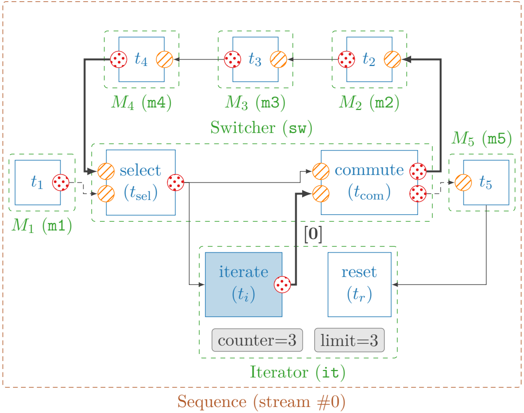
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



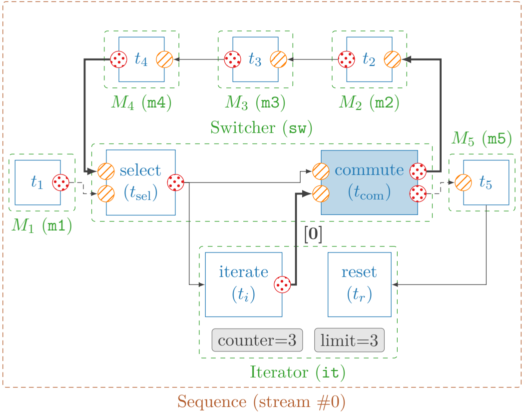
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

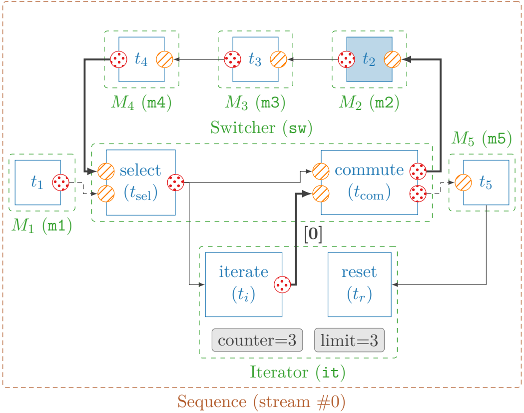
For Loop – Module Iterator



- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

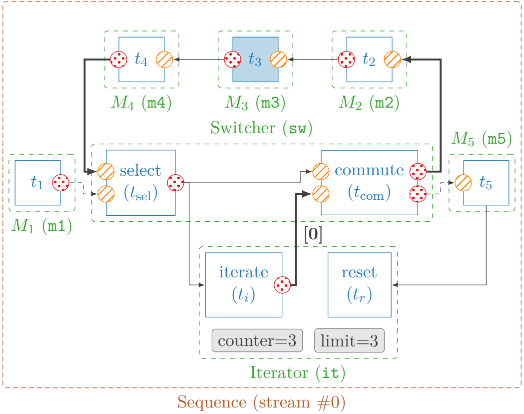
For Loop – Module Iterator

3 DSEL: Control Flow



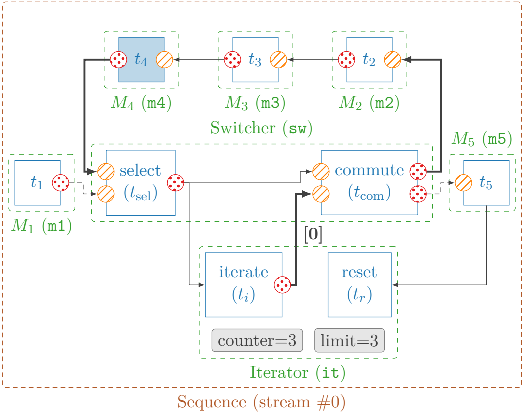
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



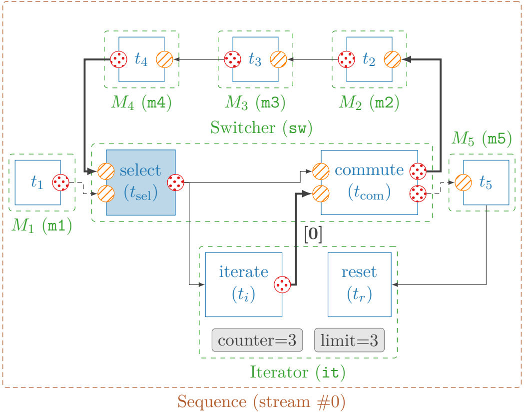
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



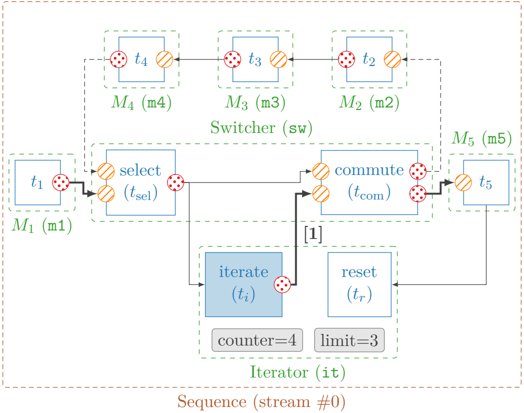
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



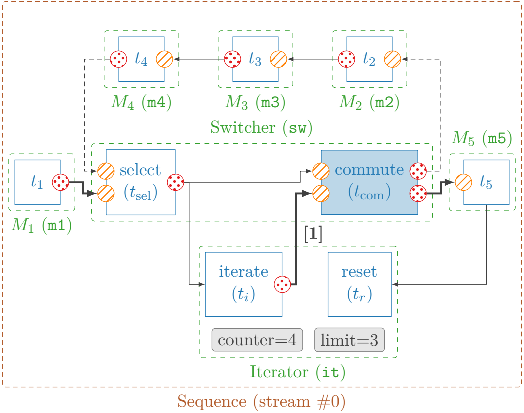
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0

For Loop – Module Iterator



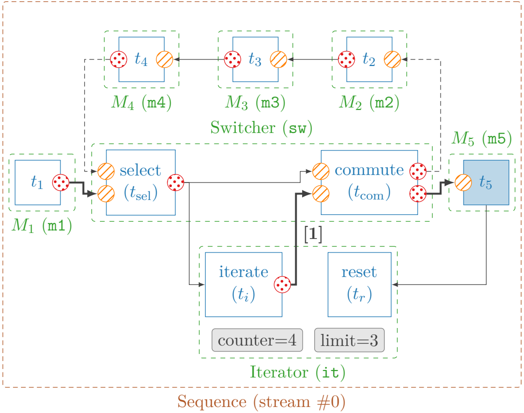
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0
  - When `iterate` is executed for the fourth time `counter > limit` returns 1 and the path changes to 1

For Loop – Module Iterator



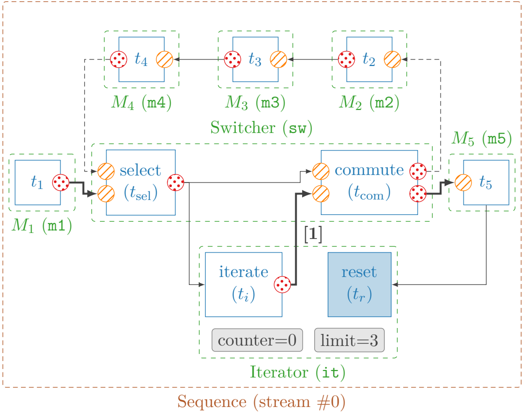
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0
  - When `iterate` is executed for the fourth time `counter > limit` returns 1 and the path changes to 1

For Loop – Module Iterator



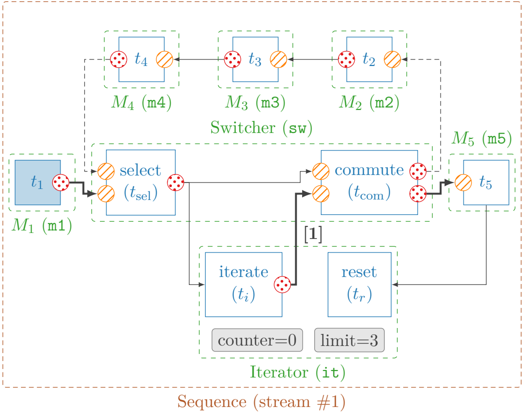
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0
  - When `iterate` is executed for the fourth time `counter > limit` returns 1 and the path changes to 1

For Loop – Module Iterator



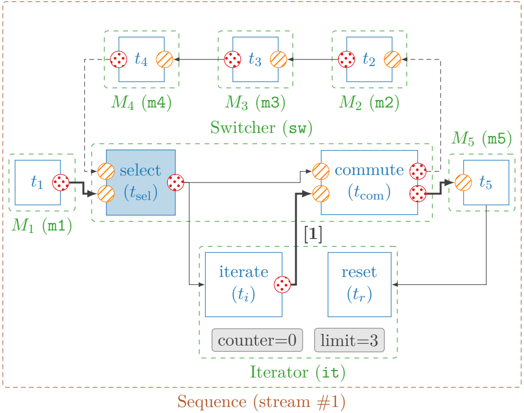
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0
  - When `iterate` is executed for the fourth time `counter > limit` returns 1 and the path changes to 1
  - At the end we need to call the `reset` task of the `Iterator` module to be ready for the next stream (= stream #1)

For Loop – Module Iterator

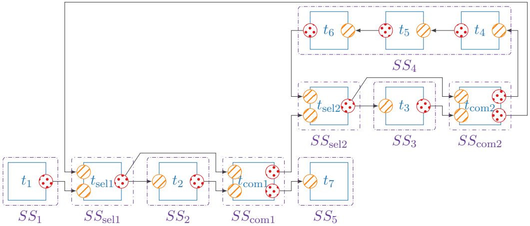


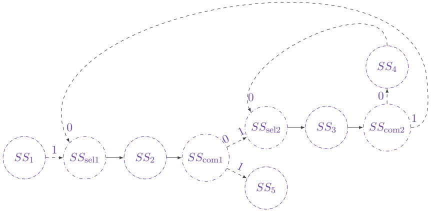
- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0
  - When `iterate` is executed for the fourth time `counter > limit` returns 1 and the path changes to 1
  - At the end we need to call the `reset` task of the `Iterator` module to be ready for the next stream (= stream #1)

For Loop – Module Iterator



- The output socket of the `iterate` task is bound to the input `in_ctrl` socket of the `commute` task
  - The path is initialized to 1
  - When `iterate` is executed for the first time `counter > limit` returns 0 and the path changes to 0
  - When `iterate` is executed for the fourth time `counter > limit` returns 1 and the path changes to 1
  - At the end we need to call the `reset` task of the `Iterator` module to be ready for the next stream (= stream #1)

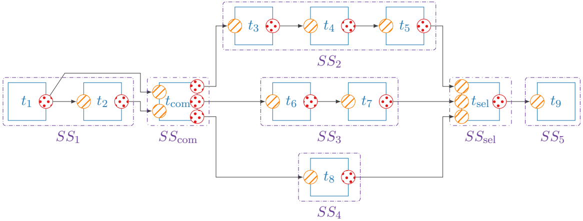




execute SS_1;
while execute SS_2 **and not** $t_{\mathrm{com1}}.\mathrm{in}_2$ **do**
  while execute SS_3 **and not** $t_{\mathrm{com2}}.\mathrm{in}_2$ **do**
    execute SS_4;
execute SS_5;

- **Nested** loops are also possible
- **Do while** loop can be implemented
- **For** loop is a variant of While loop

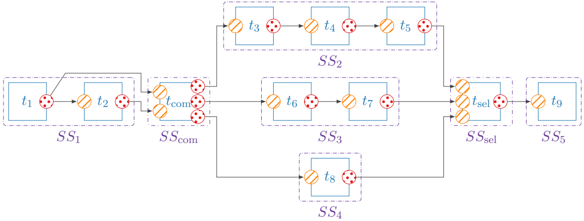
Switch




```
execute SS_1;
switch t_com.in_2 do
    case 0 do
        execute SS_2;
    case 1 do
        execute SS_3;
    case 2 do
        execute SS_4;
execute SS_5;
```

- Uses the same **Switcher** module as for the loops
  - Positions of the *commute* task (t_{com}) and *select* task (t_{sel}) are switched
  - The number of paths is determined by the system designer

Switch

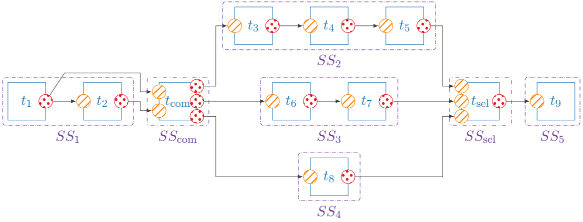


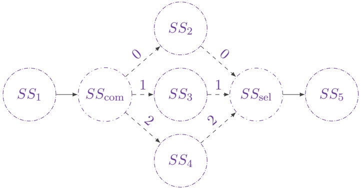
execute SS_1;
switch $t_{com}.\text{in}_2$ **do**
  case *0* **do**
    execute SS_2;
  case *1* **do**
    execute SS_3;
  case *2* **do**
    execute SS_4;
execute SS_5;

- Uses the same **Switcher** module as for the loops
  - Positions of the *commute* task (t_{com}) and *select* task (t_{sel}) are switched
  - The number of paths is determined by the system designer

➔ Static graph but **dynamic frames scheduling**

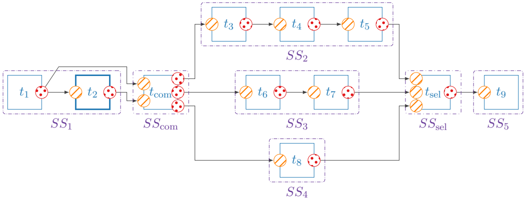
Switch – Code Example






```
Switcher sw(3); // 3 exclusive paths

sw["commute::in_data" ] = m1[     "t1::out"        ];
m2[      "t2::in"       ] = m1[     "t1::out"        ];
sw["commute::in_ctrl" ] = m2[     "t2::out"        ];
// sub-seq. 2, executed if commute::in_ctrl = 0
m3[      "t3::in"       ] = sw["commute::out_data0"];
m4[      "t4::in"       ] = m3[     "t3::out"        ];
m5[      "t5::in"       ] = m4[     "t4::out"        ];
// sub-seq. 3, executed if commute::in_ctrl = 1
m6[      "t6::in"       ] = sw["commute::out_data1"];
m7[      "t7::in"       ] = m6[     "t6::out"        ];
// sub-seq. 4, executed if commute::in_ctrl = 2
m8[      "t8::in"       ] = sw["commute::out_data2"];
// merge exclusive paths
sw[ "select::in_data0"] = m5[     "t5::out"        ];
sw[ "select::in_data1"] = m7[     "t7::out"        ];
sw[ "select::in_data2"] = m8[     "t8::out"        ];
// last task binding
m6[      "t6::in"       ] = sw[ "select::out_data" ];
```

Switch – Controller



- t_2 task controls the path selection of the `commute`
- Some predefined "Controllers" are available
  - `Controller_static`: always take the same path during the sequence execution
  - `Controller_limit`: change the path after a fixed number of executions
  - `Controller_cyclic`: change the path cyclically (ex: 0, 1, 2, 0, 1, 2, ...)
- Controllers are intended to be combined with the switch pattern (and not the loop pattern), thus there is only one path taken per stream
  - Except if the switch is in a loop of course...

Evaluation of the Runtime Overhead

Notations:

- $\mathcal{C}$: compute task
- $\mathcal{I}$: control task
- $\mathcal{S}_{\text{sel}}$: select task
- $\mathcal{S}_{\text{com}}$: commute task

Micro-benchmarks:

- MB_1: simple chain
- MB_2: for loop
- MB_3: nested loops
- MB_4: switch

Evaluation of the Runtime Overhead

Notations:

- $\mathcal{C}$: compute task
- $\mathcal{I}$: control task
- $\mathcal{S}_{\text{sel}}$: select task
- $\mathcal{S}_{\text{com}}$: commute task

Micro-benchmarks:

- MB_1: simple chain
- MB_2: for loop
- MB_3: nested loops
- MB_4: switch

			Overhead								
			$\mathcal{C}$ tasks		$\mathcal{S}_{\text{sel}}$ tasks		$\mathcal{S}_{\text{com}}$ tasks		$\mathcal{I}$ tasks		Other
Label	# streams	Run time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Time (ms)
MB_1	375000	4656.45	1125000	151.86	–	–	–	–	–	–	4.59
MB_2	37500	4744.08	1125000	151.86	412500	24.75	412500	33.00	412500	28.88	5.59
MB_3	37500	4777.03	1125000	151.86	562500	33.75	562500	45.00	562500	39.38	7.04
MB_4	562500	4784.88	1125000	151.86	562500	33.75	562500	45.00	562500	39.38	14.89

Execution of 1 125 000 $\mathcal{C}$ tasks of 4 μs each. Theoretical execution time is 4500 ms for each micro-benchmark (on Intel® Core™ i5-8250U @ 1.60 GHz, 15-Watt TDP).

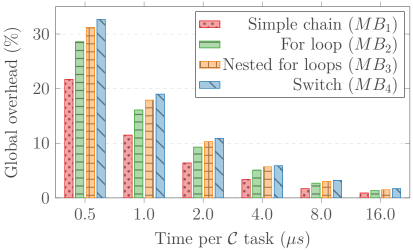
Evaluation of the Runtime Overhead

Notations:

- $\mathcal{C}$: compute task
- $\mathcal{I}$: control task
- $\mathcal{S}_{\text{sel}}$: select task
- $\mathcal{S}_{\text{com}}$: commute task

Micro-benchmarks:

- MB_1: simple chain
- MB_2: for loop
- MB_3: nested loops
- MB_4: switch



			Overhead								
			$\mathcal{C}$ tasks		$\mathcal{S}_{\text{sel}}$ tasks		$\mathcal{S}_{\text{com}}$ tasks		$\mathcal{I}$ tasks		Other
Label	# streams	Run time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Exec.	Time (ms)	Time (ms)
MB_1	375000	4656.45	1125000	151.86	–	–	–	–	–	–	4.59
MB_2	37500	4744.08	1125000	151.86	412500	24.75	412500	33.00	412500	28.88	5.59
MB_3	37500	4777.03	1125000	151.86	562500	33.75	562500	45.00	562500	39.38	7.04
MB_4	562500	4784.88	1125000	151.86	562500	33.75	562500	45.00	562500	39.38	14.89

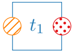
Execution of 1 125 000 $\mathcal{C}$ tasks of 4 μs each. Theoretical execution time is 4500 ms for each micro-benchmark (on Intel® Core™ i5-8250U @ 1.60 GHz, 15-Watt TDP).

Table of Contents

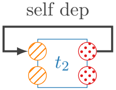
4 DSEL: Multi-threaded Runtime

Stateless and Stateful Tasks

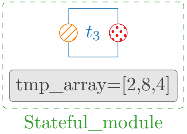
- Stateless task:
  - A task that has no internal state or side effect
  - This type of task is straight-forward to replicate (= parallelize)
  - All the input and output data are defined through sockets



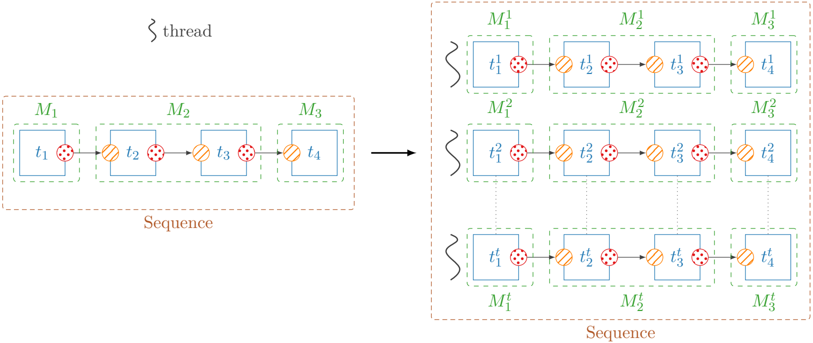
- Stateful task with data dependency between 2 consecutive executions:
  - By definition, only one instance of this type of task can be ran in parallel because the execution x needs the data of the execution $x - 1$
  - In AFF3CT, when there is this type of dependency, we do not use sockets but internal module memory instead



- Stateful task without data dependency:
  - This type of task will read/write inner data (not exposed in sockets)
  - However, if this data is carefully replicated, multiple instances of this type of task can run in parallel

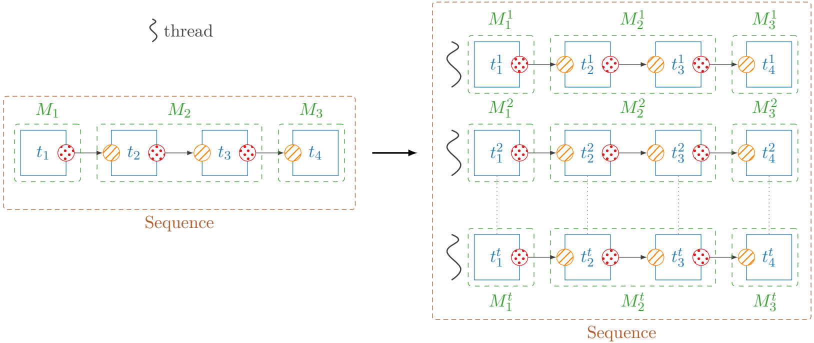


Sequence Replication



- **Automatic replication** of the modules
  - Execute the tasks on different threads: preserves **data locality**
  - Stateful model: user sometimes needs to implement a `deep_copy()` method

Sequence Replication



- **Automatic replication** of the modules
  - Execute the tasks on different threads: preserves **data locality**
  - Stateful model: user sometimes needs to implement a `deep_copy()` method
    - ➔ Based on the “clone” design pattern

Module Clone

- By default, a stateful task cannot be replicated and will throw an exception during the automatic clone process
- AFF3CT cannot know if the inner data represents data dependency or not
- The developer of the module needs to confirm that the data can be replicated
  - Depending on the type of data (heap data pointer or local data) the developer need to explicit the data replication
  - In theory, a read only pointer does not need to be copied (but sometime it is simpler to copy it: think about which module will make the free...)
  - Whereas, a read/write pointer will require a new allocation for the new replicated task
- Naturally, for “stateless modules”, the replication will work by default

Module Clone – Simple Example

Let us assume we have this module:

```cpp
class My_module1 : public Module {
  private:
  // this two members need to be copied if
  // we want to replicate the module
  const float val1;
  int tmp[12];
  public:
  My_module(const float val1) : Module(), val1(val1) {
    set_name("My_module1");
    Task &t = create_tsk("compute");
    size_t si = create_sck_in<float>(t, "in", 12);
    size_t so = create_sck_out<int8_t>(t, "out", 12);
    create_codelet(t,
      [si,so](Module &m, Task &tsk) -> int {
        /* ... */
        return runtime::status_t::SUCCESS;
    });
  }
};
```

To make it replicable we need to add:

```cpp
class My_module1 : public Module {
  private:
  const float val1;
  int tmp[12];
  public:
  My_module(const float val1) : Module(), val1(val1) {
    set_name("My_module1");
    Task &t = create_tsk("compute");
    /* ... */
  }
  virtual My_module1* clone() const {
    // this line invokes the default copy constructor
    // of 'My_module1' class, and the 'val1' and 'tmp'
    // members are automatically copied
    auto m = new My_module1(*this);
    // the 'deep_copy' method comes from the 'Module'
    // class, this is required to copy the sockets
    m->deep_copy(*this);
    // return a new instance of My_module1 class
    return m;
  }
};
```

Module Clone – Less Simple Example

Let us consider a slightly different case:

```
class My_module2 : public Module {
  private:
  // this two members need to be copied if
  // we want to replicate the module
  const size_t size;
  int* tmp; // <- here the default copy constructor
            //    will only copy the pointer /!\
  public:
  My_module2(const size_t size) : Module(), size(size) {
    this->tmp = new int[size];
    set_name("My_module2");
    Task &t = create_tsk("compute");
    size_t si = create_sck_in<float>(t, "in", size);
    size_t so = create_sck_out<int8_t>(t, "out", size);
    /* ... */
  }
  virtual ~My_module2() { delete[] this->tmp; };
};
```

To make it replicable we need to add:

```
class My_module2 : public Module {
  private:
  const size_t size;
  int* tmp;
  public:
  /* same constructor & destructor methods */
  virtual My_module2* clone() const {
    auto m = new My_module2(*this);
    m->deep_copy(*this);
    return m;
  }
  // in this case, overloading the 'deep_copy'
  // method is required
  virtual void deep_copy(const My_module2& m) {
    Module::deep_copy(m); // <- do not forget!
    // allocate a new tmp buffer
    this->tmp = new int[this->size];
    // copy the values from the initial tmp into
    // the new one ('this' is the new one!)
    std::copy(m.tmp.begin(), m.tmp.end(),
              this->tmp.begin());
  }
};
```

Module Clone – Less Simple Example

Let us consider a slightly different case:

```cpp
class My_module2 : public Module {
  private:
  // this two members need to be copied if
  // we want to replicate the module
  const size_t size;
  int* tmp; // <- here the default copy constructor
            //    will only copy the pointer /!\
  public:
  My_module2(const size_t size) : Module(), size(size) {
    this->tmp = new int[size];
    set_name("My_module2");
    Task &t = create_tsk("compute");
    size_t si = create_sck_in<float>(t, "in", size);
    size_t so = create_sck_out<int8_t>(t, "out", size);
    /* ... */
  }
  virtual ~My_module2() { delete[] this->tmp; };
};
```

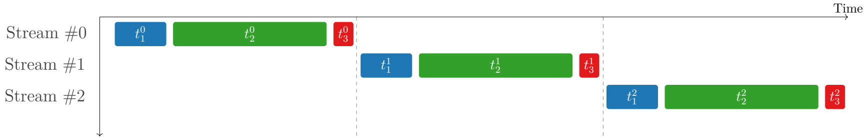
- We could have overloaded the copy constructor but it is more explicit to have a dedicated `deep_copy` method

To make it replicable we need to add:

```cpp
class My_module2 : public Module {
  private:
  const size_t size;
  int* tmp;
  public:
  /* same constructor & destructor methods */
  virtual My_module2* clone() const {
    auto m = new My_module2(*this);
    m->deep_copy(*this);
    return m;
  }
  // in this case, overloading the 'deep_copy'
  // method is required
  virtual void deep_copy(const My_module2& m) {
    Module::deep_copy(m); // <- do not forget!
    // allocate a new tmp buffer
    this->tmp = new int[this->size];
    // copy the values from the initial tmp into
    // the new one ('this' is the new one!)
    std::copy(m.tmp.begin(), m.tmp.end(),
              this->tmp.begin());
  }
};
```

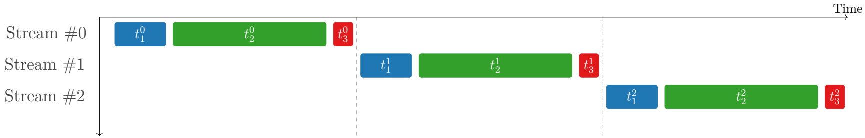
Pipeline Strategy

- Let us assume we have 3 stateful task t_1^s, t_2^s and t_3^s we cannot replicate because of data dependency (s is the stream id)
- Here is the corresponding Gantt chart without pipelining:

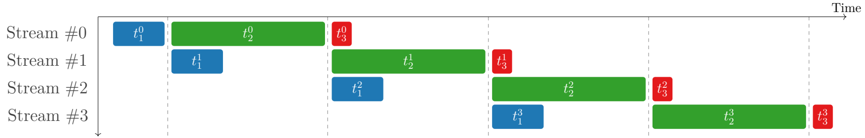


Pipeline Strategy

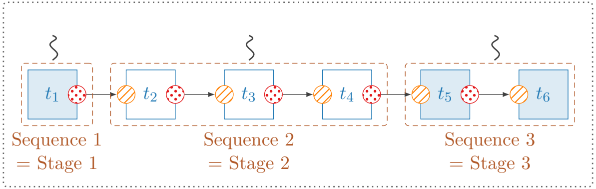
- Let us assume we have 3 stateful task t_1^s, t_2^s and t_3^s we cannot replicate because of data dependency (s is the stream id)
- Here is the corresponding Gantt chart without pipelining:



- And here is the corresponding Gantt chart with pipelining:



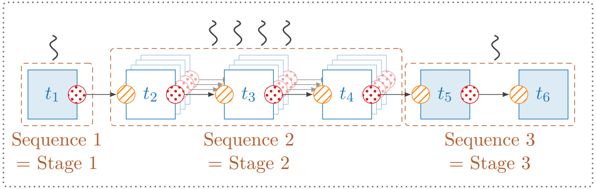
Pipeline



Pipeline

- Sequential tasks cannot be duplicated ➔ **Pipeline strategy**
  - From now, stateful tasks that cannot be replicated will be represented by light blue filled boxes (ex.: t_1, t_5 and t_6 here)
- A pipeline is composed of a sequence list

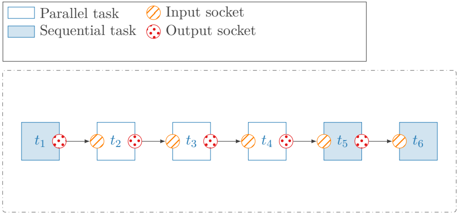
Pipeline



Pipeline

- Sequential tasks cannot be duplicated ➔ **Pipeline strategy**
  — From now, stateful tasks that cannot be replicated will be represented by light blue filled boxes (ex.: t_1, t_5 and t_6 here)
- A pipeline is composed of a sequence list
- The **sequence replication** technique is still possible in **parallel stages**

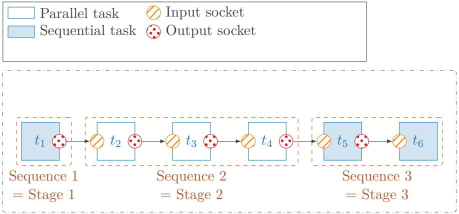
Pipeline – The Big Picture



Pipeline user description

1. Describe the app in a **directed graph of tasks**

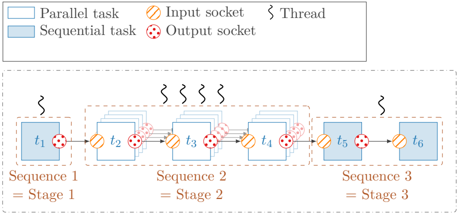
Pipeline – The Big Picture



Pipeline user description

1. Describe the app in a **directed graph of tasks**
2. Group tasks in **pipeline stages**

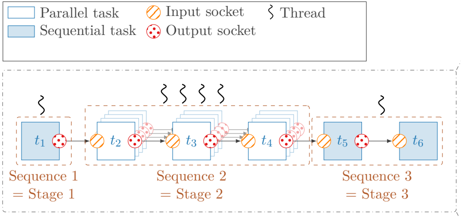
Pipeline – The Big Picture



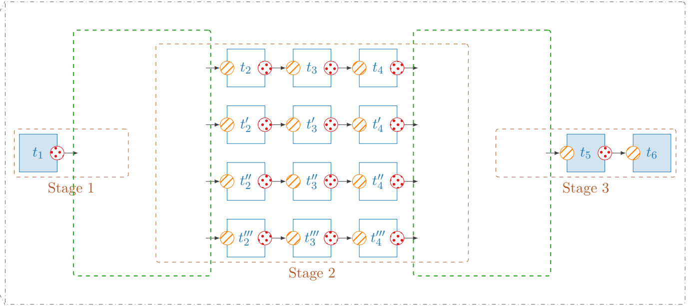
Pipeline user description

1. Describe the app in a **directed graph of tasks**
2. Group tasks in **pipeline stages**
3. Select the **number of threads** per stage

Pipeline – The Big Picture



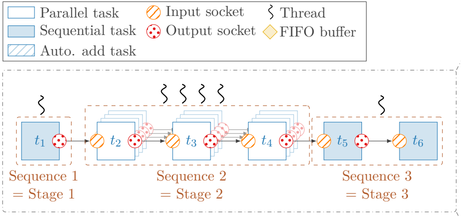
Pipeline user description



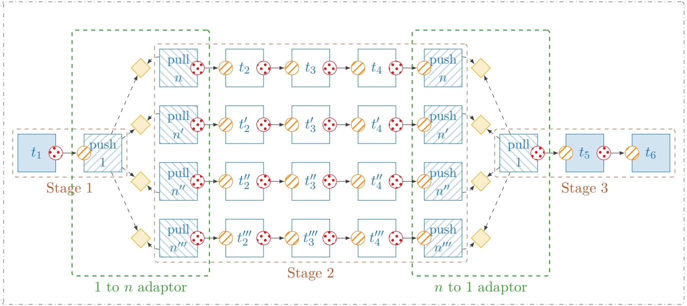
Pipeline automatic internal implementation

1. Describe the app in a **directed graph of tasks**
2. Group tasks in **pipeline stages**
3. Select the **number of threads** per stage

Pipeline – The Big Picture



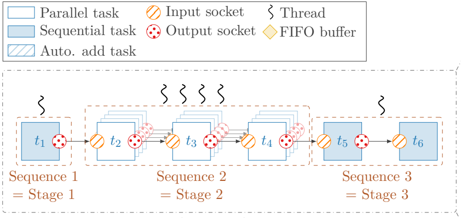
Pipeline user description



Pipeline automatic internal implementation

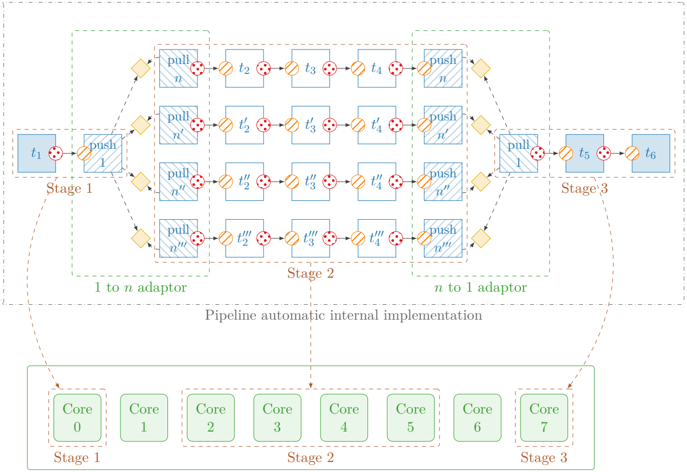
1. Describe the app in a **directed graph of tasks**
2. Group tasks in **pipeline stages**
3. Select the **number of threads** per stage

Pipeline – The Big Picture



Pipeline user description

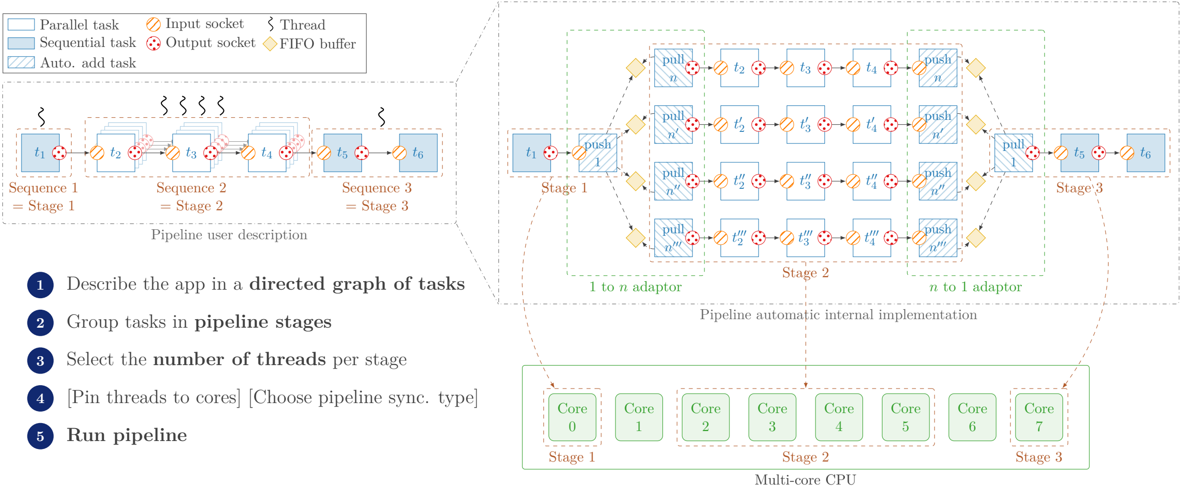
1. Describe the app in a **directed graph of tasks**
2. Group tasks in **pipeline stages**
3. Select the **number of threads** per stage
4. [Pin threads to cores] [Choose pipeline sync. type]



Pipeline automatic internal implementation

Multi-core CPU

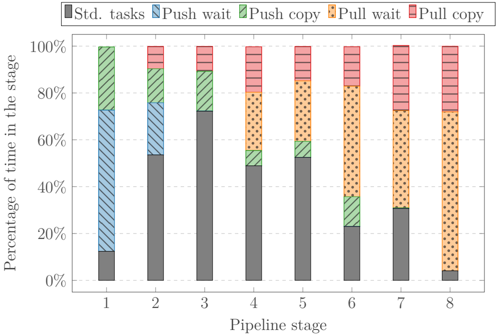
Pipeline – The Big Picture



1. Describe the app in a **directed graph of tasks**
2. Group tasks in **pipeline stages**
3. Select the **number of threads** per stage
4. [Pin threads to cores] [Choose pipeline sync. type]
5. **Run pipeline**

Pipeline – Data Copy VS Copy-less

Benchmarks on a radio application (DVB-S2 receiver, MODCOD 2):

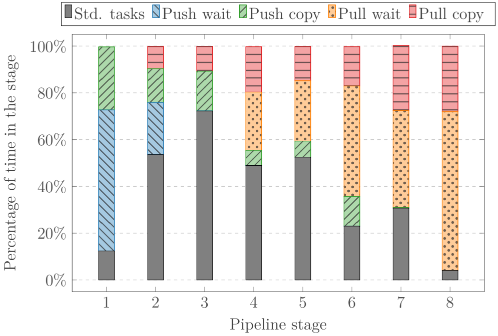


Percentage of time per stage with copy.

- Stage throughput: **40 Mb/s**

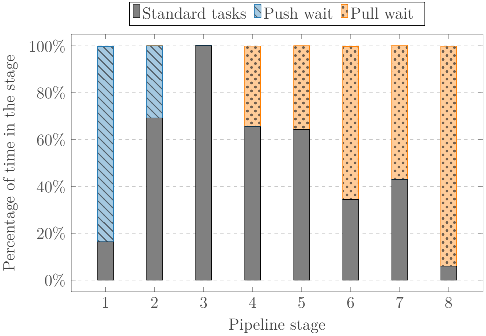
Pipeline – Data Copy VS Copy-less

Benchmarks on a radio application (DVB-S2 receiver, MODCOD 2):



Percentage of time per stage with copy.

- Stage throughput: **40 Mb/s**

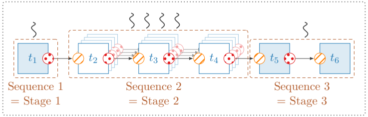


Percentage of time per stage w/o copy.

- Stage throughput: **55 Mb/s**

Pipeline – Source Code Example

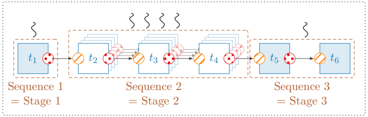
4 DSEL: Multi-threaded Runtime



Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
```

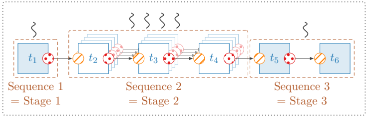
Pipeline – Source Code Example



Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
```

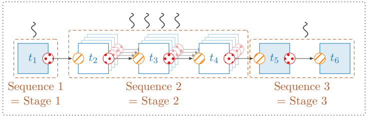
Pipeline – Source Code Example



Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(m1("t1"),        // first task for stages verification
  { { { m1("t1") }, { m1("t1") } },    // first & last tasks of stage 1
    { { m2("t2") }, { m4("t4") } },    // first & last tasks of stage 2
    { { m5("t5") }, { m6("t6") } } });// first & last tasks of stage 3
```

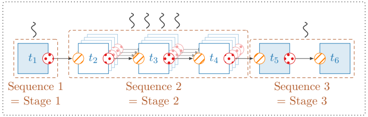
Pipeline – Source Code Example



Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(m1("t1"),          // first task for stages verification
  { { { m1("t1") }, { m1("t1") } },      // first & last tasks of stage 1
    { { m2("t2") }, { m4("t4") } },      // first & last tasks of stage 2
    { { m5("t5") }, { m6("t6") } } },    // first & last tasks of stage 3
  {   1,           4,          1     }); // number of threads per stage
```

Pipeline – Source Code Example

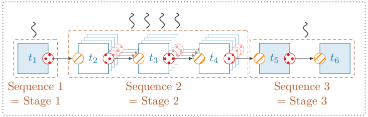


Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(m1("t1"),        // first task for stages verification
  { { { m1("t1") }, { m1("t1") } },   // first & last tasks of stage 1
    { { m2("t2") }, { m4("t4") } },   // first & last tasks of stage 2
    { { m5("t5") }, { m6("t6") } } }, // first & last tasks of stage 3
  {   1,           4,           1    },  // number of threads per stage
  { { 0 }, { 2, 3, 4, 5 }, { 7 }, }); // threads pinning on the CPU cores
```

Pipeline – Source Code Example

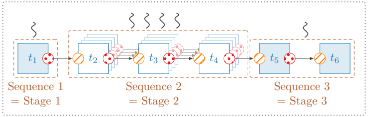
4 DSEL: Multi-threaded Runtime



Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(m1("t1"),        // first task for stages verification
  { { { m1("t1") }, { m1("t1") } },    // first & last tasks of stage 1
    { { m2("t2") }, { m4("t4") } },    // first & last tasks of stage 2
    { { m5("t5") }, { m6("t6") } } }, // first & last tasks of stage 3
  {   1,           4,           1    }, // number of threads per stage
  { { 0 }, { 2, 3, 4, 5 }, { 7 }, },   // threads pinning on the CPU cores
  {       3,               12        }); // size of the sync. buffers between stages
```

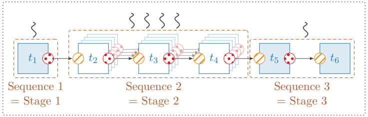
Pipeline – Source Code Example



Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(m1("t1"),          // first task for stages verification
  { { { m1("t1") }, { m1("t1") } },      // first & last tasks of stage 1
    { { m2("t2") }, { m4("t4") } },      // first & last tasks of stage 2
    { { m5("t5") }, { m6("t6") } } },    // first & last tasks of stage 3
  {   1,          4,          1    },    // number of threads per stage
  { { 0 }, { 2, 3, 4, 5 }, { 7 }, },     // threads pinning on the CPU cores
  {      3,              12        },    // size of the sync. buffers between stages
  {      true,           false     });   // active/passive waiting between stages
```

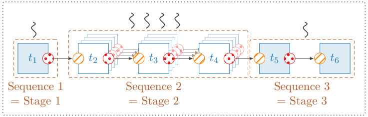
Pipeline – Source Code Example



Pipeline

```
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(m1("t1"),        // first task for stages verification
  { { { m1("t1") }, { m1("t1") } },    // first & last tasks of stage 1
    { { m2("t2") }, { m4("t4") } },    // first & last tasks of stage 2
    { { m5("t5") }, { m6("t6") } } },  // first & last tasks of stage 3
  {   1,           4,           1    },  // number of threads per stage
  { { 0 }, { 2, 3, 4, 5 }, { 7 }, },   // threads pinning on the CPU cores
  {      3,              12          },  // size of the sync. buffers between stages
  {     true,           false        }); // active/passive waiting between stages
// 4) execution of the pipeline, it is indefinitely executed in loop
pip.exec([]() { return false; });
```

Pipeline – Source Code Example



Pipeline

```cpp
// 1) creation of the module objects
M1 m1(); M2 m2(); M3 m3(); M4 m4(); M5 m5(); M6 m6();
// 2) binding of the tasks
m2["t2::in"] = m1["t1::out"]; m3["t3::in"] = m2["t2::out"]; m4["t4::in"] = m3["t3::out"];
m5["t5::in"] = m4["t4::out"]; m6["t6::in"] = m5["t5::out"];
// 3) creation of the pipeline (= sequences and pipeline analyses)
runtime::Pipeline pip(m1("t1"),         // first task for stages verification
  { { { m1("t1") }, { m1("t1") } },    // first & last tasks of stage 1
    { { m2("t2") }, { m4("t4") } },    // first & last tasks of stage 2
    { { m5("t5") }, { m6("t6") } } }, // first & last tasks of stage 3
  {   1,           4,          1    }, // number of threads per stage
  { { 0 }, { 2, 3, 4, 5 }, { 7 }, }, // threads pinning on the CPU cores
  {      3,              12         }, // size of the sync. buffers between stages
  {     true,           false      }); // active/passive waiting between stages
// 4) execution of the pipeline, it is indefinitely executed in loop
pip.exec([]() { return false; });
```

Summary

- A **domain specific embedded language** for **streaming applications**
- Based on single rate **synchronous dataflow** but with **runtime paths**
- **Pragmatic design**: guided by existing C/C++ codes
- Target **real-time implementations** on multi-core CPUs
- **Multi-threaded runtime**: tasks replication and pipeline
- Recently accepted in a **computer science journal**[1]
- **Open source** and available on GitHub[2]

[1] A. Cassagne, R. Tajan, O. Aumage, D. Barthou, C. Leroux, and C. Jégo. "A DSEL for High Throughput and Low Latency Software-Defined Radio on Multicore CPUs". In: *Wiley Concurrency and Computation: Practice and Experience (CCPE)* (2023). DOI: 10.1002/cpe.7820.

[2] AFF3CT-core DSEL repository: https://github.com/aff3ct/aff3ct-core

Table of Contents

5 DSEL: Miscellaneous

Output Status Socket

- For all the tasks, an output socket named "status" is automatically created
  - Contain the value of the returned integer in the task codelet!
  - 0 value ➔ everything is going well
  - Other values ➔ error code
- `runtime::status_t` enumerate defines some basic behaviors
  - `SUCCESS = 0`
  - `FAILURE = 1`
  - `FAILURE_STOP = -1`
  - `UNKNOWN = - 2`
- Type is `int32_t`

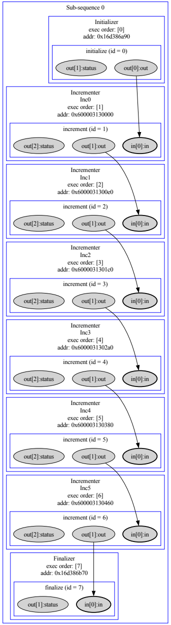
```cpp
// create a stateless module
Stateless min32();
// set module name
min32.set_name("Minimum32");
// create a task for the 'min32' module
Task &t = min32.create_tsk("find_min");
// create in/out sockets for the task
size_t si = min32.create_sck_in<int>(t, "in", 32);
size_t so = min32.create_sck_out<int>(t, "out", 1);
// define the code to execute when the
// 'find_min' task is called
min32.create_codelet(t,
  [si, so](Module &m, runtime::Task &tsk) -> int {
    // get in/out data pointers
    const int* pi = tsk[si].get_dataptr<const int>();
    int* po = tsk[so].get_dataptr<int>();
    // compute the minimum of 32 elements
    *po = pi[0];
    for (int i = 1; i < 32; i++)
      *po = std::min(*po, pi[i]);
    return runtime::status_t::SUCCESS;
});
```

Display Tasks Graph for Debugging

- Tasks bindings can be a tedious phase
  - Error messages are not always easy to understand
  - Sometimes the code executes but we don't have the expected result
- A good start is always to check if the inner tasks graph is correct

```
Initializer<uint8_t> ini(12); Finalizer<uint8_t> fin(12);
Incrementer<uint8_t> inc1(12), inc2(12), inc3(12), inc4(12), inc5(12);
// bindings
inc1["increment::in"] = ini ["initialize::out"];
inc2["increment::in"] = inc1[ "increment::out"];
inc3["increment::in"] = inc2[ "increment::out"];
inc4["increment::in"] = inc3[ "increment::out"];
inc5["increment::in"] = inc4[ "increment::out"];
fin [ "finalize::in"] = inc5[ "increment::out"];
// create a sequence (you could do the same with a pipeline)
runtime::Sequence seq(ini("initialize"));
std::ofstream file("graph.dot"); // open the 'graph.dot' file in write only mode
seq.export_dot(file); // write the sequence graph in the 'dot' format
```

- Convert to PDF: `dot -Tpdf -O graph.dot` ➔ `graph.dot.pdf`



Get Modules and Tasks – Part 1

- It is common to want to apply something to all (or a subset of) the modules/tasks of a same sequence or pipeline
  - We can do this from the modules we declared but this is dangerous: if you performed a replication you do not have the references on the newly replicated tasks!
  - To overcome this problem, sequence and pipeline come with the `get_modules*<Module_name>()` methods

```cpp
inc1["increment::in"] = ini ["initialize::out"];
inc2["increment::in"] = inc1[ "increment::out"]; inc3["increment::in"] = inc2[ "increment::out"];
inc4["increment::in"] = inc3[ "increment::out"]; inc5["increment::in"] = inc4[ "increment::out"];
fin [ "finalize::in"] = inc5[ "increment::out"];
// create a sequence (you could do the same with a pipeline)
runtime::Sequence seq(ini("initialize"), 4); // 4 means 4 threads => the sequence is replicated 4 times!
// get all the modules contained in the sequence
for (auto& mdl : seq.get_modules<module::Module>(false)) // for each module
  for (auto& tsk : mdl->tasks) // for each task of the current module
    tsk->set_debug(true); // enable the debug mode on the task
```

Get Modules and Tasks – Part 2

- Sometimes when want something more specific
  - Let us assume we want to initialize all the “Initializer” of a sequence that has been replicated 4 times

```cpp
Initializer<uint8_t> ini(2048); Finalizer<uint8_t> fin(2048);
Incrementer<uint8_t> inc1(2048), inc2(2048), inc3(2048), inc4(2048), inc5(2048);

// bindings
inc1["increment::in"] = ini ["initialize::out"];
inc2["increment::in"] = inc1[ "increment::out"]; inc3["increment::in"] = inc2[ "increment::out"];
inc4["increment::in"] = inc3[ "increment::out"]; inc5["increment::in"] = inc4[ "increment::out"];
fin [ "finalize::in"] = inc5[ "increment::out"];

// create a sequence (you could do the same with a pipeline)
runtime::Sequence seq(ini("initialize"), 4); // 4 means 4 threads => the sequence is replicated 4 times!

std::vector<uint8_t> init_vals(2048);
// set 0, 1, 2, ..., 2047
std::iota(init_vals.begin(), init_vals.end(), 0);
// get all the 'Initializer<uint8_t>'' contained in the sequence
for (auto& mdl : seq.get_modules<Initializer<uint8_t>>(false)) // for each 'Initializer<uint8_t>' module
  mdl->set_init_data(init_vals); // 4 instances of ''Initializer<uint8_t>' class will be initialized
```

Print Tasks Statistics

- AFF3CT comes with integrated performance measurements
  - Number of task executions
  - Task latency (average, minimum and maximum)

```
Initializer<uint8_t> ini(2048); Finalizer<uint8_t> fin(2048);
Incrementer<uint8_t> inc1(2048), inc2(2048), inc3(2048), inc4(2048), inc5(2048);
// bindings
inc1["increment::in"] = ini ["initialize::out"];
inc2["increment::in"] = inc1[ "increment::out"]; inc3["increment::in"] = inc2[ "increment::out"];
inc4["increment::in"] = inc3[ "increment::out"]; inc5["increment::in"] = inc4[ "increment::out"];
fin [ "finalize::in"] = inc5[ "increment::out"];
// create a sequence (you could do the same with a pipeline)
runtime::Sequence seq(ini("initialize"));
// enable the statistics collection for each task of the sequence
for (auto& mdl : seq.get_modules<module::Module>(false))
  for (auto& tsk : mdl->tasks)
    tsk->set_stats(true); // enable the statistics
// execute the sequence (tasks graph is executed 100000 times)
unsigned int exe_counter = 0;
seq.exec([&exe_counter]() { return ++exe_counter >= 100000; });
// print the tasks statistics
const bool ordered = true, display_throughput = false;
tools::Stats::show(seq.get_modules_per_types(), ordered, display_throughput);
```

Print Tasks Statistics – Output Example


```
# -------------------------------------------||------------------------------||--------------------------------
#        Statistics for the given task       ||       Basic statistics       ||        Measured latency
#     ('*' = any, '-' = same as previous)    ||          on the task         ||
# -------------------------------------------||------------------------------||--------------------------------
# -------------|-------------------|---------||----------|----------|--------||----------|----------|----------
#       MODULE |              TASK |   TIMER ||    CALLS |     TIME |   PERC || AVERAGE |  MINIMUM |  MAXIMUM
#              |                   |         ||          |      (s) |    (%) ||     (us) |     (us) |     (us)
# -------------|-------------------|---------||----------|----------|--------||----------|----------|----------
#         Inc3 |         increment |       * ||   100000 |     0.50 |  16.67 ||     5.02 |     5.00 |    31.00
#         Inc5 |         increment |       * ||   100000 |     0.50 |  16.67 ||     5.02 |     5.00 |    54.00
#         Inc4 |         increment |       * ||   100000 |     0.50 |  16.67 ||     5.02 |     5.00 |    38.00
#         Inc0 |         increment |       * ||   100000 |     0.50 |  16.66 ||     5.02 |     5.00 |    32.00
#         Inc1 |         increment |       * ||   100000 |     0.50 |  16.66 ||     5.02 |     5.00 |    37.00
#         Inc2 |         increment |       * ||   100000 |     0.50 |  16.66 ||     5.02 |     5.00 |    64.00
#    Finalizer |          finalize |       * ||   100000 |     0.00 |   0.00 ||     0.00 |     0.00 |    13.00
#  Initializer |        initialize |       * ||   100000 |     0.00 |   0.00 ||     0.00 |     0.00 |     9.00
# -------------|-------------------|---------||----------|----------|--------||----------|----------|----------
#        TOTAL |                 * |       * ||   100000 |     3.01 | 100.00 ||    30.12 |    30.00 |   278.00
```

Apple M1 Pro CPU, buffers size of 2048 bytes, 100000 streams, 1 thread and each "Incrementer" sleeps for 5 μs (`test-simple-chain -p -d 2048 -e 100000 -t 1 -s 5`).

Inter-frame

TODO :-)

The `set_n_frames()` method allows to process more than one frame per task execution. This can enable data parallelism inside the tasks (simple `#pragma omp paraller for`, or more complex reordering for SIMD efficiency).

Early Exit in Stream

TODO :-)

Throwing the `tools::processing_aborted` exception during the execution of a task will cancel the current stream at this step and the next stream will start from the beginning.

Force Dependencies between Tasks

- Sometimes we want to force the execution order between two tasks and data dependency from the binding is not enough to do it
  - This is especially the case when using forward socket!
- A solution to force a dependency is to add a “fake” input socket to the task that need to execute after, and to bind the output `status` socket of the first task to the “fake” input socket of the second task ;-)

Table of Contents

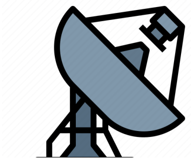
Section 6.1

DVB-S2 Standard Application

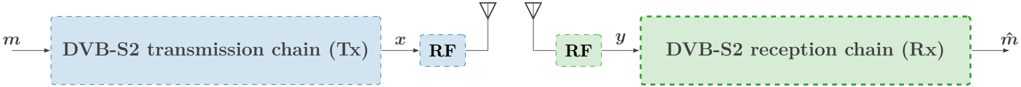
Presentation & Context

- **Ground-satellite communications**
- Video transmission: **DVB-S2 standard**
- **Ground station** side implementation
- Need for flexibility

  → **SDR on multi-core and SIMD CPUs**

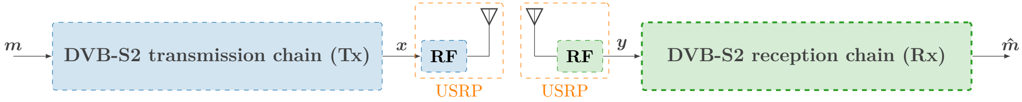


Setup & Objectives



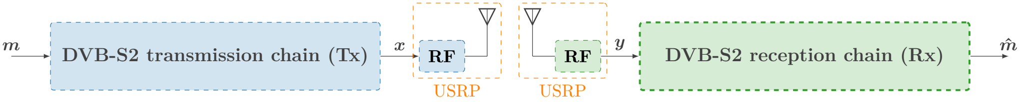
- 1x Middle class computer for the digital transmitter (Tx)
- 1x Server class computer for **the digital receiver (Rx)**

Setup & Objectives



- 1x Middle class computer for the digital transmitter (Tx)
- 1x Server class computer for **the digital receiver (Rx)**
- 2x Universal Software Radio Peripherals (USRPs) N320 for the RF

Setup & Objectives



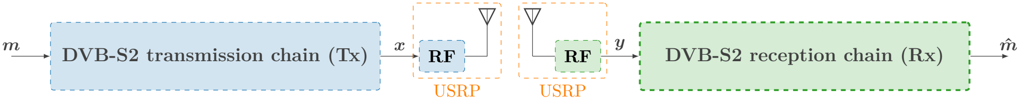
- 1x Middle class computer for the digital transmitter (Tx)
- 1x Server class computer for **the digital receiver (Rx)**
- 2x Universal Software Radio Peripherals (USRPs) N320 for the RF

Config.	Modulation	Rate R	K_{BCH}	K_{LDPC}	N_{LDPC}	$\mathcal{T}_i$ (Rx, Seq.)
MODCOD 1	QPSK	3/5	9552	9720	16200	3.4 Mb/s
MODCOD 2	QPSK	8/9	14232	14400	16200	4.1 Mb/s
MODCOD 3	8-PSK	8/9	14232	14400	16200	4.0 Mb/s

Selected DVB-S2 configurations (MODCOD).

Setup & Objectives

6 Applications



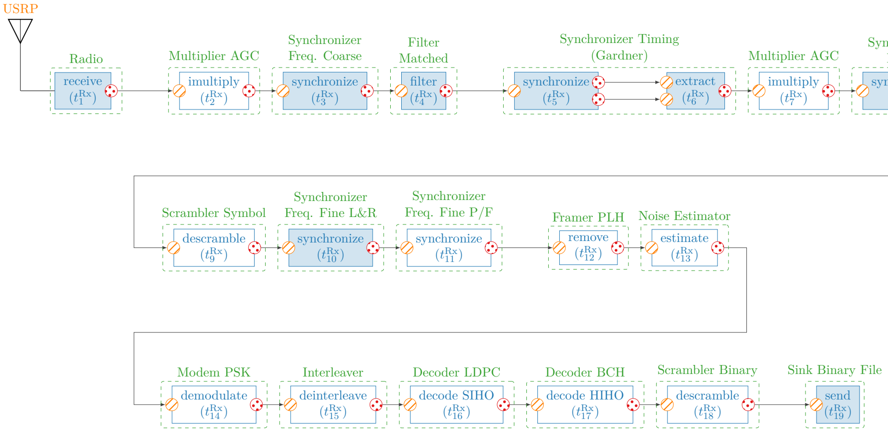
- 1x Middle class computer for the digital transmitter (Tx)
- 1x Server class computer for **the digital receiver (Rx)**
- 2x Universal Software Radio Peripherals (USRPs) N320 for the RF
- **Industrial real-time constraint**: 30 $\sim$ 50 Mb/s

Config.	Modulation	Rate R	K_{BCH}	K_{LDPC}	N_{LDPC}	$\mathcal{T}_i$ (Rx, Seq.)
MODCOD 1	QPSK	3/5	9552	9720	16200	3.4 Mb/s
MODCOD 2	QPSK	8/9	14232	14400	16200	4.1 Mb/s
MODCOD 3	8-PSK	8/9	14232	14400	16200	4.0 Mb/s

Selected DVB-S2 configurations (MODCOD).

Receiver: Tasks Graph

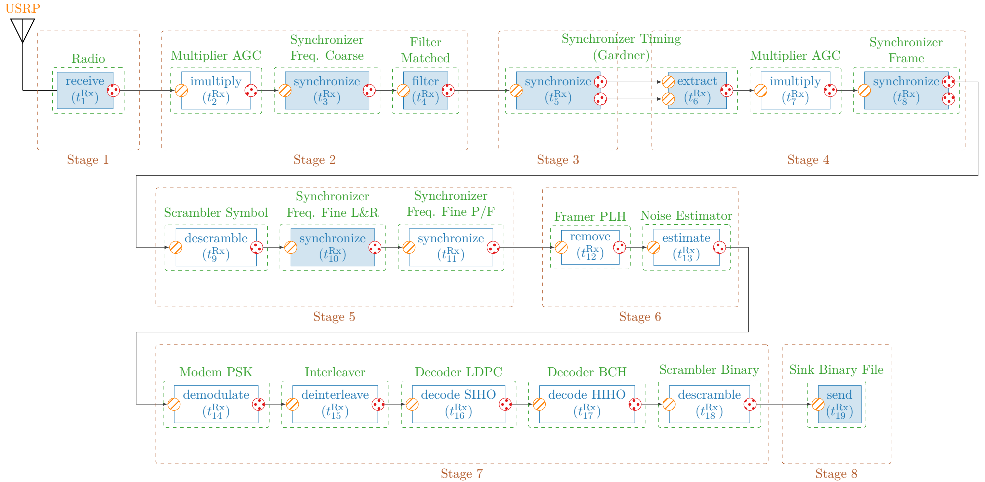
6 Applications



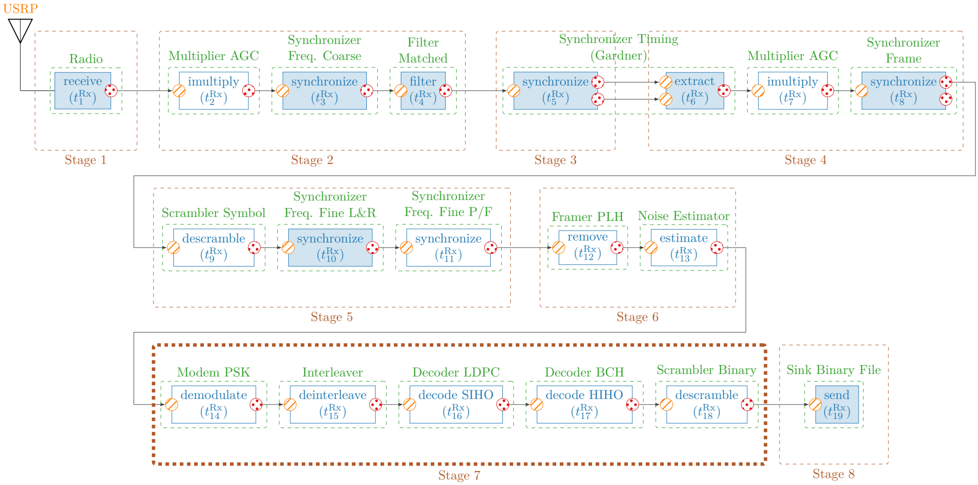
Receiver: Sequential Tasks (MODCOD 2)

	Throughput (MS/s)				Latency (μs)			Time
Stages and Tasks	Avg	Min.	Max.	$\mathcal{N}_{\text{Avg}}$	Avg	Min.	Max.	(%)
Radio - *receive* (t_1^{Rx})	1015.86	234.20	1093.98	431.83	527.32	489.66	2287.24	0.94
Stage 1	1015.86	234.20	1093.98	431.83	527.32	489.66	2287.24	0.94
Multiplier AGC - *imultiply* (t_2^{Rx})	864.41	420.05	935.71	367.45	619.71	572.49	1275.28	1.11
Synch. Freq. Coarse - *synchronize* (t_3^{Rx})	1979.17	665.98	2237.38	841.32	270.66	239.42	804.35	0.48
Filter Matched - *filter* (t_4^{Rx})	273.85	121.60	275.25	116.41	1956.08	1946.13	4405.09	3.49
Stage 2	188.19	82.61	194.22	80.00	2846.45	2758.04	6484.72	5.08
Synch. Timing - *synchronize* (t_5^{Rx})	130.38	58.97	131.31	55.42	4108.52	4079.39	9084.64	7.34
Stage 3	130.38	58.97	131.31	55.42	4108.52	4079.39	9084.64	7.34
Synch. Timing - *extract* (t_6^{Rx})	331.50	151.54	354.62	281.83	807.97	755.28	1767.48	1.44
Multiplier AGC - *imultiply* (t_7^{Rx})	806.31	442.69	877.19	685.51	332.18	305.34	605.02	0.59
Synch. Frame - *synchronize* (t_8^{Rx})	187.50	120.17	193.25	159.41	1428.51	1386.01	2228.76	2.55
Stage 4	104.27	58.21	109.47	88.65	2568.66	2446.63	4601.26	4.58
Scrambler Symbol - *descramble* (t_9^{Rx})	1979.41	668.85	2649.55	1682.89	135.31	101.09	400.45	0.24
Synch. Freq. Fine L&R - *synchronize* (t_{10}^{Rx})	1466.55	596.19	1741.72	1246.85	182.63	153.78	449.25	0.33
Synch. Freq. Fine P/F - *synchronize* (t_{11}^{Rx})	132.40	62.59	140.88	112.56	2022.98	1901.24	4279.30	3.61
Stage 5	114.42	52.22	124.22	97.27	2340.92	2156.11	5129.00	4.18
Framer PLH - *remove* (t_{12}^{Rx})	1148.07	427.71	1180.59	1008.60	225.77	219.55	606.02	0.40
Noise Estimator - *estimate* (t_{13}^{Rx})	626.12	151.24	656.09	550.06	413.98	395.07	1713.87	0.74
Stage 6	405.16	111.73	421.72	355.94	639.75	614.62	2319.89	1.14
Modem PSK - *demodulate* (t_{14}^{Rx})	46.07	42.12	46.28	40.47	5626.34	5600.83	6153.50	10.05
Interleaver - *deinterleave* (t_{15}^{Rx})	1533.54	518.95	1582.97	1347.25	169.02	163.74	499.47	0.30
Decoder LDPC - *decode SIHO* (t_{16}^{Rx})	166.15	69.12	171.59	164.21	1386.74	1342.74	3333.34	2.48
Decoder BCH - *decode HIHO* (t_{17}^{Rx})	6.92	6.15	6.96	6.92	32905.37	32705.15	36998.15	58.79
Scrambler Binary - *descramble* (t_{18}^{Rx})	91.11	47.74	91.73	91.11	2499.41	2482.41	4770.24	4.47
Stage 7	5.35	4.40	5.38	5.35	42586.88	42294.87	51754.70	76.09
Sink Binary File - *send* (t_{19}^{Rx})	1838.31	25.30	2100.47	1838.31	123.87	108.41	9001.34	0.22
Stage 8	1838.31	25.30	2100.47	1838.31	123.87	108.41	9001.34	0.22
Total	4.09	2.51	4.14	4.09	55742.37	54947.73	90662.79	99.57

Receiver: Stages Decomposition

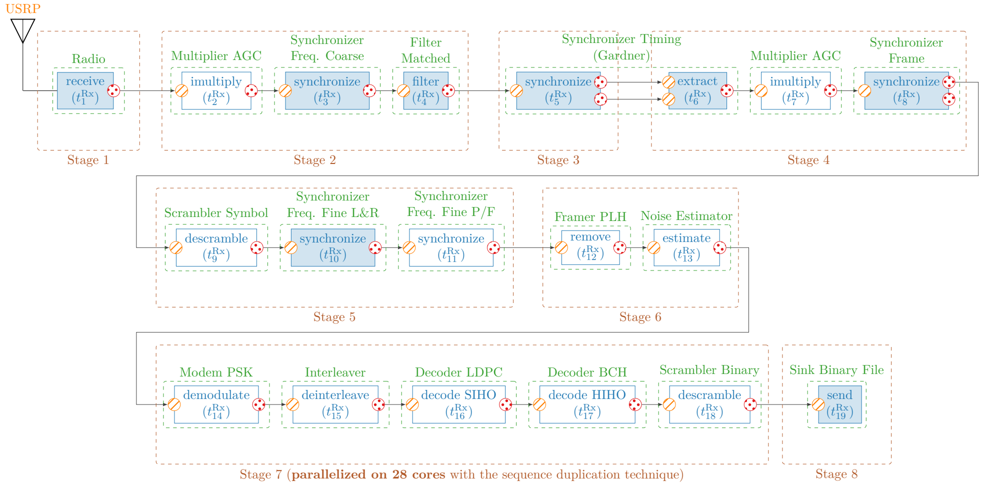


Receiver: Stages Decomposition



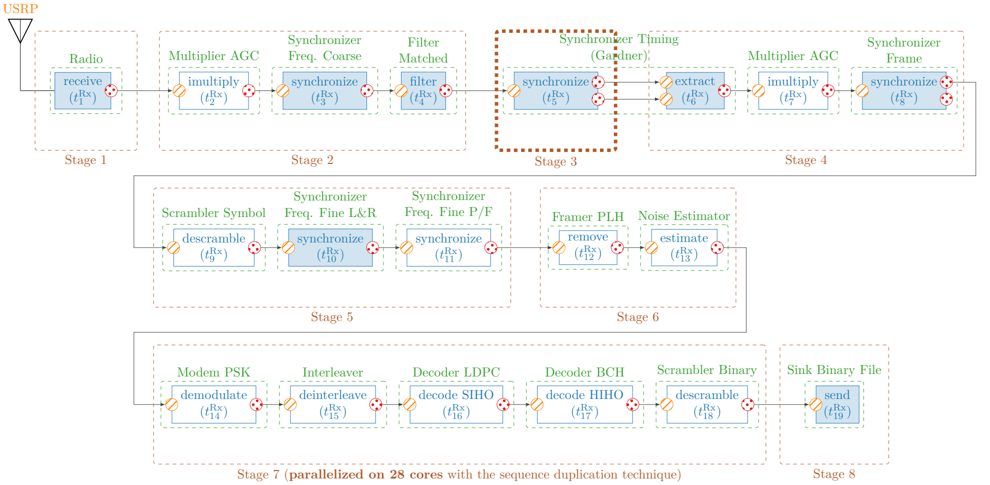
Receiver: Stages Decomposition

6 Applications

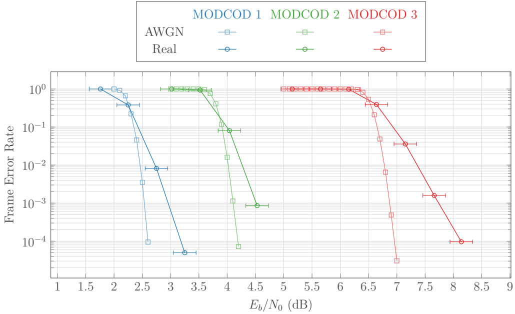


Receiver: Stages Decomposition

6 Applications

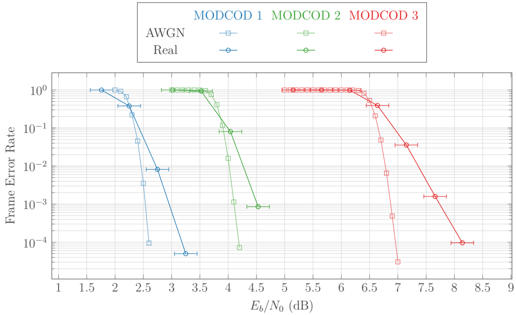


Receiver: Performance

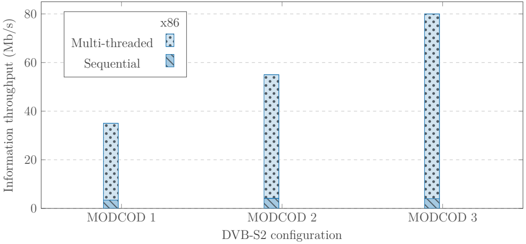


Decoding performance.

Receiver: Performance



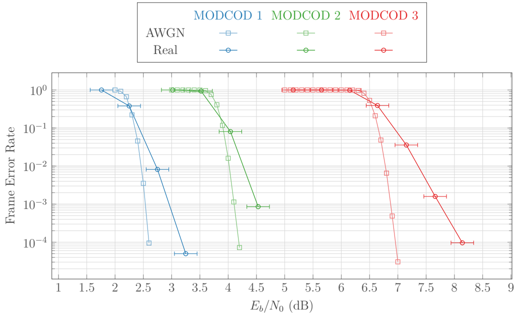
Decoding performance.



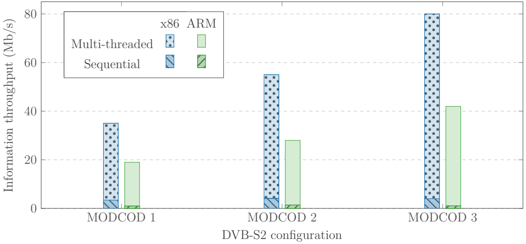
Throughput performance.

- 2× Intel® Xeon™ Platinum 8168 x86 CPUs @ 2.70 GHz (28/48 cores used)
  — **Matches the industrial real-time constraint** (from 35 to 80 Mb/s)

Receiver: Performance



Decoding performance.



Throughput performance.

- 2× Intel® Xeon™ Platinum 8168 x86 CPUs @ 2.70 GHz (28/48 cores used)
  — **Matches the industrial real-time constraint** (from 35 to 80 Mb/s)
- 2 × Cavium ThunderX2® CN9975 v2.1 CPUs @ 2.00 GHz (40/56 cores used)

Summary

- DVB-S2: a digital **communication standard** for satellites
- Tested and validated on **real radios** (USRPs)
- **3.5 times faster** than the GNU Radio implementation
- Published in the proceedings of a **signal processing conference**[1]
- **Used in industrial context** as an SDR demonstrator
- **Open source implementation** is available on GitHub[2]

[1] A. Cassagne, M. Léonardon, R. Tajan, C. Leroux, C. Jégo, O. Aumage, and D. Barthou. "A Flexible and Portable Real-time DVB-S2 Transceiver using Multicore and SIMD CPUs". In: *International Symposium on Topics in Coding (ISTC)*. IEEE, Sept. 2021. DOI: 10.1109/ISTC49272.2021.9594063.

[2] DVB-S2 SDR transceiver repository: https://github.com/aff3ct/dvbs2

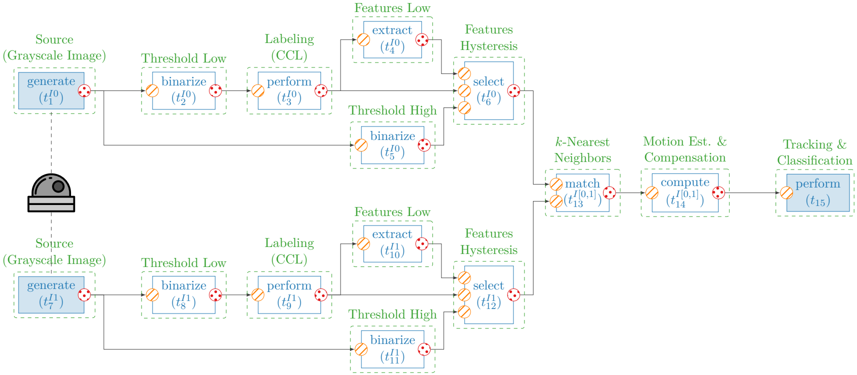
Section 6.2

Meteor Detection Chain Application

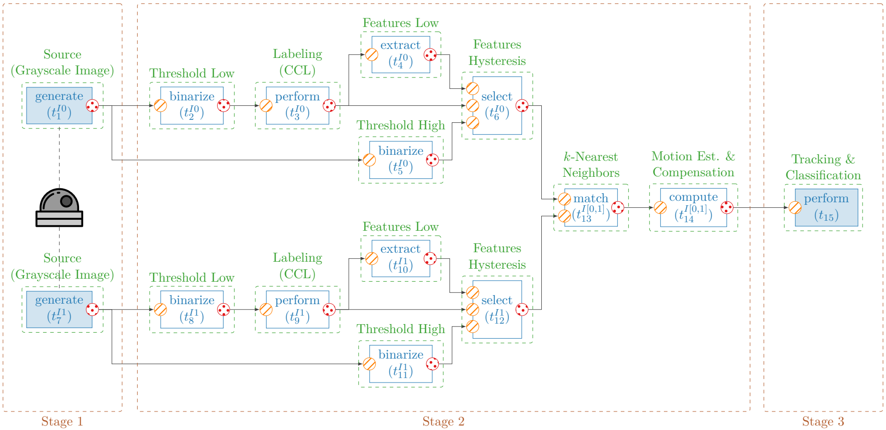
Presentation & Context

- A new application for **meteor detection**
  - Robust to **camera movements**
  - For **low power** embedded SoCs
- For **airborne observations**
  - Aircraft campaigns
  - “Weather” balloon
- Real-time constraints: $\geq$ 25 FPS, $\leq$ 10 Watts
- LIP6 ALSoC & IMCCE joint-team

Tasks Graph & Pipeline Stages

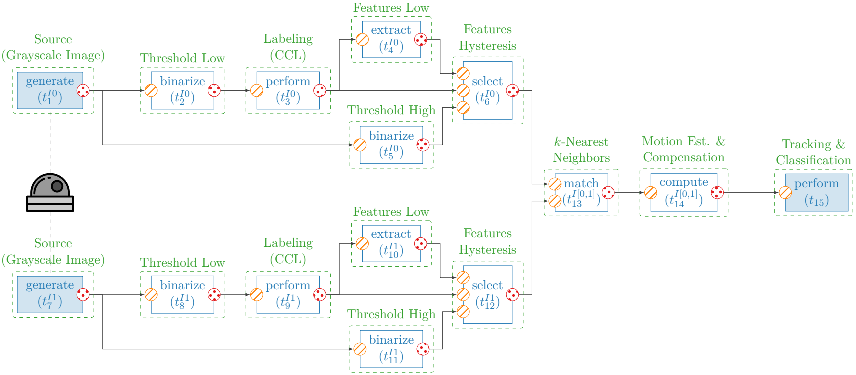


Tasks Graph & Pipeline Stages

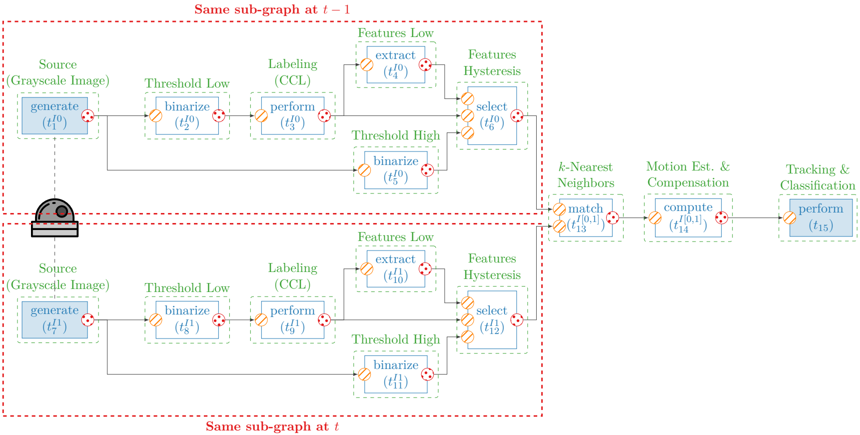


Tasks Graph Optimization (1)

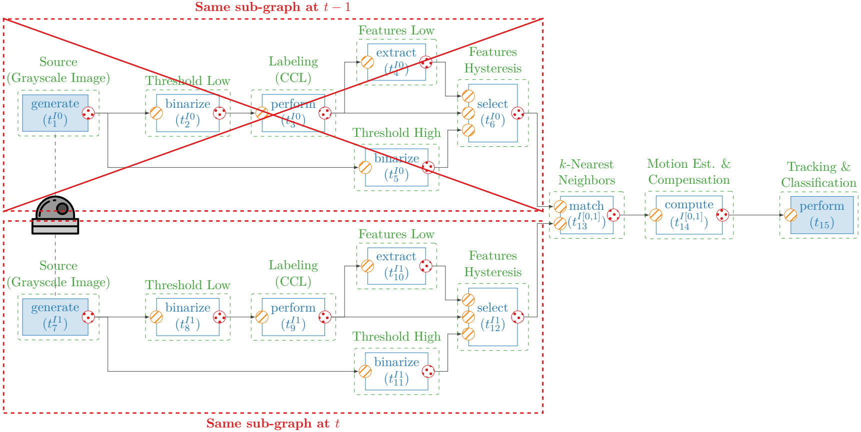
6 Applications



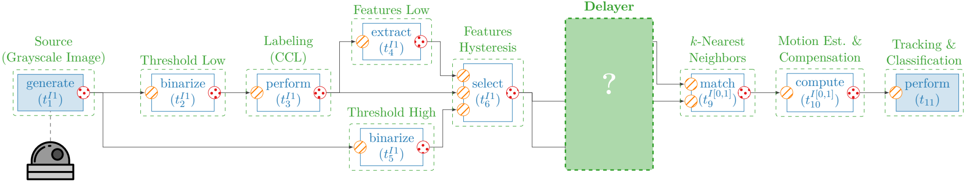
Tasks Graph Optimization (1)



Tasks Graph Optimization (1)

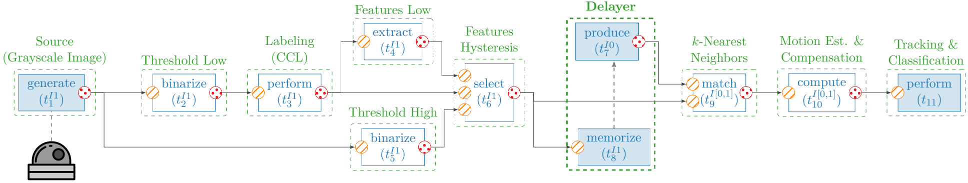


Tasks Graph Optimization (2)



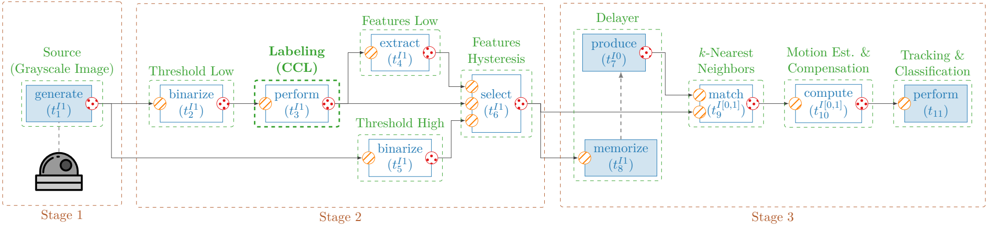
- **Delayer** module: **memorize** data at t and **produce** it at $t+1$

Tasks Graph Optimization (2)



- **Delayer** module: **memorize** data at t and **produce** it at $t+1$
  - **produce** task is triggered before **memorize** task
  - Stateful tasks with a **common internal buffer**

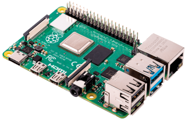
Tasks Graph Optimization (2)

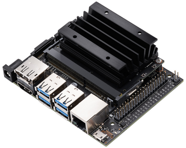


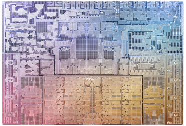
- **Delayer** module: **memorize** data at t and **produce** it at $t + 1$
  - **produce** task is triggered before **memorize** task
  - Stateful tasks with a **common internal buffer**
- Stage 2 takes more than **95% of the total time**
  - Efficient **Light Speed Labeling** (LSL) algorithm is used for labeling

Testbed

6 Applications

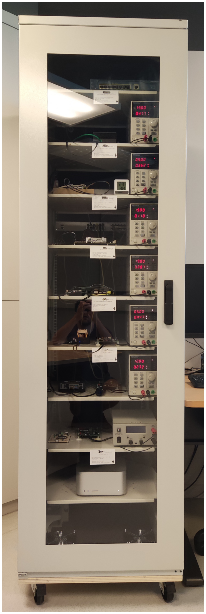




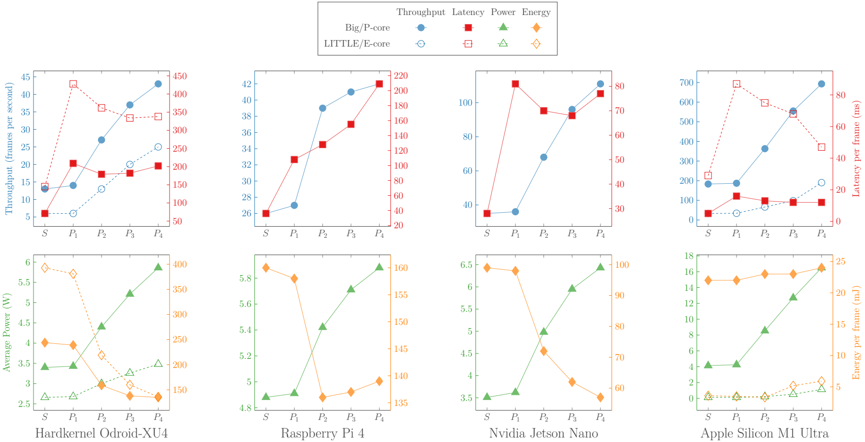


Ref.	Name	Date	Proc.	CPUs	Freq.	RAM (Size & T/P)	
XU4	Hardkernel Odroid-XU4	2016	28 nm	4 × *LITTLE* ARMv7 Cortex-A7 4 × *Big* ARMv7 Cortex-A15	1.4 GHz 1.5 GHz	2 GB	3.5 GB/s
RPi4	Raspberry Pi 4 model B	2019	28 nm	4 × *Big* ARMv8 Cortex-A72	1.5 GHz	8 GB	3.9 GB/s
Nano	Nvidia Jetson Nano	2019	20 nm	4 × *Big* ARMv8 Cortex-A57	≈ 1.5 GHz	4 GB	9.0 GB/s
M1	Apple Silicon M1 Ultra	2022	5 nm	4 × *E-core* ARMv8 Icestorm 16 × *P-core* ARMv8 Firestorm	≈ 2.0 GHz ≈ 3.0 GHz	64 GB	344.0 GB/s

Specifications of the tested SoCs.



Performance on Full HD Video (1920×1080)



Summary
6 Applications

- Application of **interest for astronomers**
- All the targeted SoCs match **real-time constraints** ($\geq$ 25 FPS, $\leq$ 10 Watts)
- Used in an **astronomy and astrophysics journal paper**[1]
- **Optimizations and parallelization** has been presented at COMPAS[2]
- **Open source implementation** is available on GitHub[3]

---

[1] J. Vaubaillon, C. Loir, C. Ciocan, M. Kandeepan, M. Millet, A. Cassagne, L. Lacassagne, P. da Fonseca, F. Zander, D. Buttsworth, S. Loehle, J. Tóth, S. Gray, A. Moingeon, and N. Rambaux. "A 2022 τ-Herculid Meteor Cluster from an Airborne Experiment: Automated Detection, Characterization, and Consequences for Meteoroids". In: *Astronomy and Astrophysics (A&A)* (Feb. 2023). DOI: `10.1051/0004-6361/202244993`.

[2] Mathuran Kandeepan, Clara Ciocan, Adrien Cassagne, and Lionel Lacassagne. "Parallélisation d'une nouvelle application embarquée pour la détection automatique de météores". In: *Conférence d'informatique en Parallélisme, Architecture et Système (COMPAS)*. Annecy, France, July 2023. DOI: `10.48550/arXiv.2307.10632`.

[3] FMDT repository: `https://github.com/alsoc/fmdt`

Q&A

Thank you for listening!
Do you have any questions?

Bibliography

[1] A. Cassagne, R. Tajan, O. Aumage, D. Barthou, C. Leroux, and C. Jégo. “A DSEL for High Throughput and Low Latency Software-Defined Radio on Multicore CPUs”. In: *Wiley Concurrency and Computation: Practice and Experience (CCPE)* (2023). DOI: 10.1002/cpe.7820.

[2] A. Cassagne, M. Léonardon, R. Tajan, C. Leroux, C. Jégo, O. Aumage, and D. Barthou. “A Flexible and Portable Real-time DVB-S2 Transceiver using Multicore and SIMD CPUs”. In: *International Symposium on Topics in Coding (ISTC)*. IEEE, Sept. 2021. DOI: 10.1109/ISTC49272.2021.9594063.

[3] J. Vaubaillon, C. Loir, C. Ciocan, M. Kandeepan, M. Millet, A. Cassagne, L. Lacassagne, P. da Fonseca, F. Zander, D. Buttsworth, S. Loehle, J. Tóth, S. Gray, A. Moingeon, and N. Rambaux. “A 2022 τ-Herculid Meteor Cluster from an Airborne Experiment: Automated Detection, Characterization, and Consequences for Meteoroids”. In: *Astronomy and Astrophysics (A&A)* (Feb. 2023). DOI: 10.1051/0004-6361/202244993.

[4] Mathuran Kandeepan, Clara Ciocan, Adrien Cassagne, and Lionel Lacassagne. “Parallélisation d’une nouvelle application embarquée pour la détection automatique de météores”. In: *Conférence d’informatique en Parallélisme, Architecture et Système (COMPAS)*. Annecy, France, July 2023. DOI: 10.48550/arXiv.2307.10632.