# StreamPU: A DSEL and a Runtime for Efficient Execution of Streaming Applications on Multicore & Heterogeneous CPUs

Alpes Heterogeneous Computing Workshop, Annecy

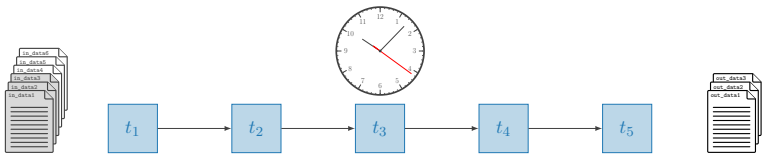
Adrien Cassagne – Sorbonne University, LIP6, CNRS

June, 2024

# Table of Contents

# Streaming Applications






- **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)
  - **High performance expectations**

- Examples of streaming applications
  - Media applications (OSI 6-7): **audio & video processing** in general
  - Network (OSI 3): **software routers**
  - Software-defined radio (OSI 1): smartphone base stations (**cloud-RAN**)

# Domain Specific Language



Domain Specific Language (DSL):

- Language **designed specifically for a class of applications**
- Pros:
  - **Very well suited** for a class of applications
  - Can be **specifically optimized**
- Cons:
  - Need to **re-write existing application** into the new language
  - Complicated to do things for what the DSL has not been intended for

Domain Specific Embedded Language (DSEL):

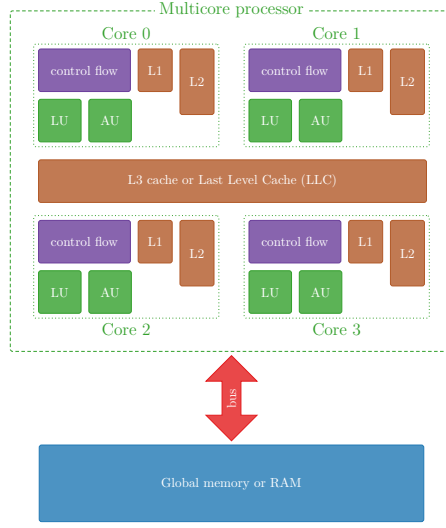
- **Embedded into an other main language**
- Pros:
  - **Simplify the porting** of applications
  - No need to write a compiler
- Cons:
  - Not as “pure” and “elegant” as a DSL... **over patching syndrome**

# Multicore CPU

1 Introduction & Context

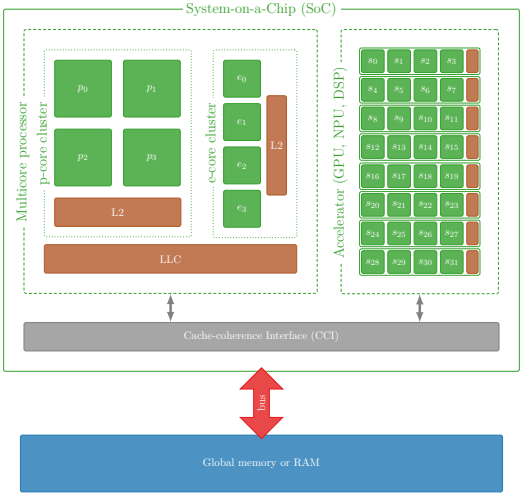
- Characteristics
  - Programmable ALUs
  - Grouped into cores
  - Memory affinities
- Well-spread
  - (Super-)computers
  - Smartphones
  - Embedded devices

➔ **Low cost for high performance!**



# Multicore & Heterogenous SoC






- Complex systems
  - Powerful and power efficient CPU cores
  - Specialized process units
  - Various sizes and types of on-chip memory
- Great opportunities
  - Unified global memory (shared pages)
  - Powerful, specialized and low power systems

➔ **Need for adapted software stack!**

# Table of Contents

2 DSEL for Streaming Applications

# Motivations

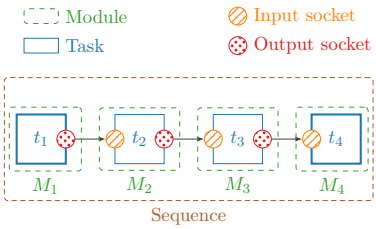


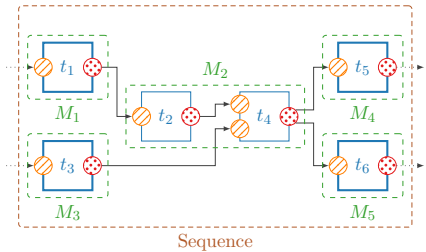
- Need for environment adapted to **streaming applications**:
  - Take advantage of multicore architectures
  - High throughput & low latency (for now, CPUs preferred to GPUs)
  - Manage energy consumption
- Proposed solution:
  - C++ Domain Specific Embedded Language (DSEL)
  - **Interpreted language**,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

  ➔ **StreamPU**

# Sockets, Tasks, Modules & Sequence



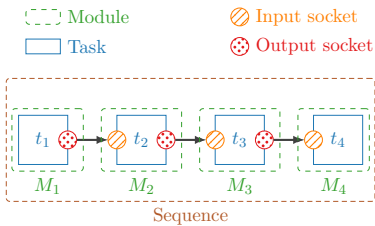



- **Directed graphs** are supported **to map a wide range of apps**
- A *sequence* is built from an initial and a final list of tasks
- Tasks execution order (scheduling) is determined by the user binding
- **States** are contained in *modules* (= C++ classes)
- One task execution is enough to run dependent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

# Simple Code Example



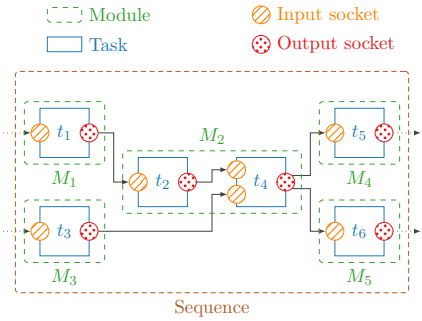



- On a module:
  - Use `("task_name")` functor to select a task
  - Use `["task_name::socket_name"]` operator to pick up a socket of a given task

```
1  // 1) create the module objects
2  M1 m1(); M2 m2(); M3 m3(); M4 m4();
3
4  // 2) bind the tasks
5  m2["t2::in"] = m1["t1::out"];
6  m3["t3::in"] = m2["t2::out"];
7  m4["t4::in"] = m3["t3::out"];
8
9  // 3) create the sequence (stop
10 //    automatically at t4 task)
11 runtime::Sequence seq(m1("t1"));
12
13 // 4) execute the sequence (tasks
14 //    graph is executed 100 times)
15 unsigned int exe_counter = 0;
16 seq.exec([&exe_counter]() {
17   return ++exe_counter >= 100;
18 });
```

# Task Graph Example



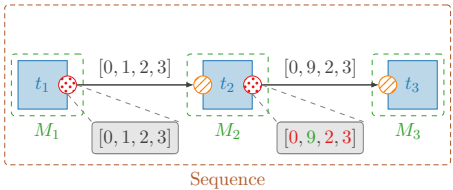


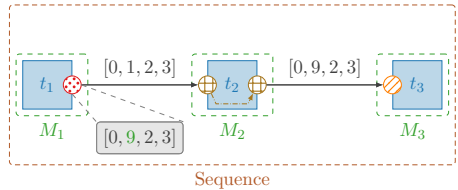

- Tasks are added to the graph in a **depth first traversal order**

```
1  // 1) create the module objects
2  M1 m1(); /* ... */ M7 m7();
3  std::vector<TYPE> some_data(SIZE);
4  // 2) bind the tasks
5  m1["t1::in" ] = some_data;
6  m3["t3::in" ] = some_data;
7  m2["t2::in" ] = m1["t1::out"];
8  m2["t4::in0"] = m2["t2::out"];
9  m2["t4::in1"] = m3["t3::out"];
10 m4["t5::in" ] = m2["t4::out"];
11 m5["t6::in" ] = m2["t4::out"];
12 m6["t7::in" ] = m4["t5::out"];
13 m7["t8::in" ] = m5["t6::out"];
14 // 3) create the sequence
15 std::vector<runtime::Task*>
16   first = { &m1("t1"), &m3("t3") },
17   last  = { &m4("t5"), &m5("t6") };
18 runtime::Sequence seq(first, last);
19 // 4) execute the sequence (no stop)
20 seq.exec([]() { return false; });
```

# Memory Allocations & Forward Socket





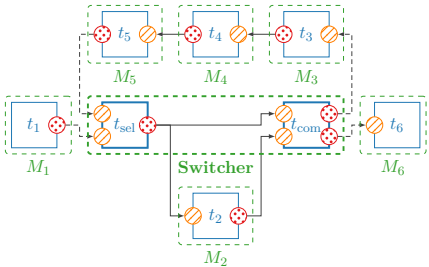




- Data are automatically allocated in the output sockets (see gray rectangles)
- Let's assume that $t_2$ only modify the second value of its input socket
  - "0", "2" and "3" are copied into $t_2$ output socket and "9" value replaces "1"
  - This is highly inefficient!

- 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!**

# Control Flow – Looping Illustrative Example






```
1 execute SS_1;
2 while execute SS_2 and
    not t_com.in_2 do
3   execute SS_3;
4 execute SS_4;
```

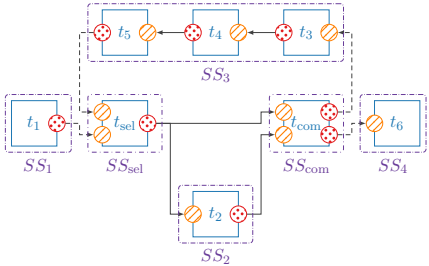
- Construct block: the **Switcher** module
  - *commute* task ($t_{\text{com}}$): **creates** exclusive execution paths
  - *select* task ($t_{\text{sel}}$): **joins** exclusive execution paths

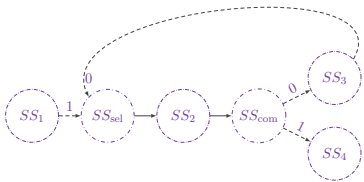
➔ Required in **turbo iterative receivers** or **recurent neural networks**

➔ **Nested loops are supported**

# While Loop – Code Example



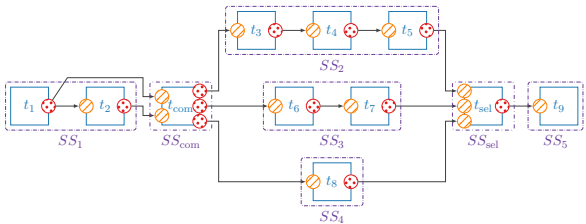



```
1  M1 m1(); /* ... */ M6 m6();
2  Switcher sw(2); // 2 exclusive paths
3
4  sw[ "select::in_data1"] = m1[     "t1::out"      ];
5  m2[     "t2::in"      ] = sw[ "select::out_data" ];
6  sw["commute::in_data" ] = sw[ "select::out_data" ];
7  sw["commute::in_ctrl" ] = m2[     "t2::out"      ];
8  // sub-seq. 3, executed if tcom::in2 = 0
9  m3[     "t3::in"      ] = sw["commute::out_data0"];
10 m4[     "t4::in"      ] = m3[     "t3::out"      ];
11 m5[     "t5::in"      ] = m4[     "t4::out"      ];
12 sw[ "select::in_data0"] = m5[     "t5::out"      ];
13 // sub-seq. 4, executed if tcom::in2 = 1
14 m6[     "t6::in"      ] = sw["commute::out_data1"];
15
16 runtime::Sequence seq(m1("t1"));
17 seq.exec([]() { return false; });
```

# Conditionnal Patterns – Switch-case Example






```
1 execute SS_1;
2 switch t_com.in_2 do
3     case 0 do
4         execute SS_2;
5     case 1 do
6         execute SS_3;
7     case 2 do
8         execute SS_4;
9 execute SS_5;
```

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

➔ Supports *if* and *if-then-else* patterns

➔ Static graph but **dynamic streams scheduling**

# Summary



Well-known streaming DSLs:

- StreamIt[a]: static scheduling, feedback loop, pipeline & fork-join parallelism
- Array-OL[b]: static scheduling, multi-stream data parallelism

[a] W. Thies, M. Karczmarek, and S. Amarasinghe. "StreamIt: A Language for Streaming Applications". In: *Compiler Construction*. Springer, 2002. DOI: 10.1007/3-540-45937-5_14.

[b] C. Glitia, P. Dumont, and P. Boulet. "Array-OL with Delays, a Domain Specific Specification Language for Multidimensional Intensive Signal Processing". In: *Springer Multidimensional Systems and Signal Processing* (2010). DOI: 10.1007/s11045-009-0085-4.

StreamPU[a] DSEL:

- Embedded in the C++ language
  - — Convenient for existing C++ apps
- JIT dataflow graph
  - — Can be modified at runtime
- Turing-complete control flow
  - — Dynamic streams scheduling
- Suited for task duration $> 1\mu s$

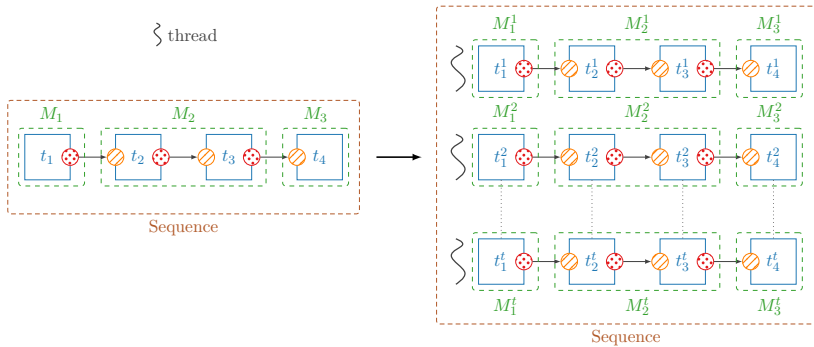
[a] A. Cassagne, R. Tajan, O. Aumage, D. Barthou, C. Leroux, and C. Jégo. "A DSEL for High Throughput and Low Latency Software-Defined Radio on Multicore CPUs". In: *Wiley Concurrency and Computation: Practice and Experience* (2023). DOI: 10.1002/cpe.7820.

# Table of Contents

3 Multi-threaded Runtime

# Sequence Replication

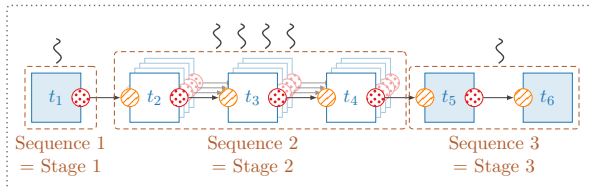
3 Multi-threaded Runtime




- **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

# Pipeline



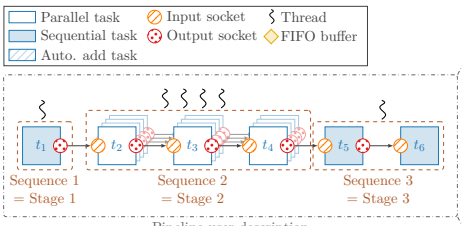


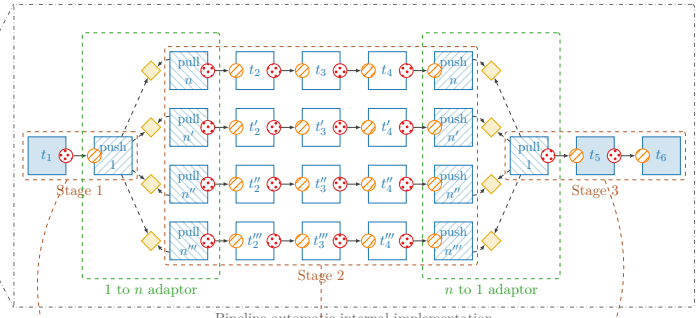

Pipeline

- Sequential tasks cannot be replicated ➔ **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**

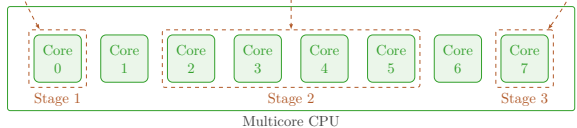
# Pipeline – The Big Picture

3 Multi-threaded Runtime




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**



# Summary



Well-known runtime systems like OpenMP and StarPU[1] focuses more on **data and tasks parallelisms**:

- Task graphs built according to the **tasks submission model**
- Pipeline and replication parallelism are not directly addressed

➔ **Not well-suited for streaming applications**

StreamPU runtime implementation is based on the **portable C++11 threads**:

- Efficient pipeline & replication parallel constructs
  - Round-robin & **zero-copy** distribution over the pipeline stages (= **low latency**)
- Take advantage of the **static task graph**
  - **Data binding** is used instead of tasks submission model (= **no overhead**)

[1] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. "StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures". In: *Wiley Concurrency and Computation: Practice and Experience* (2011). DOI: 10.1002/cpe.1631.

# Table of Contents

4 Evaluation on Real-world Applications

# Meteor Detection

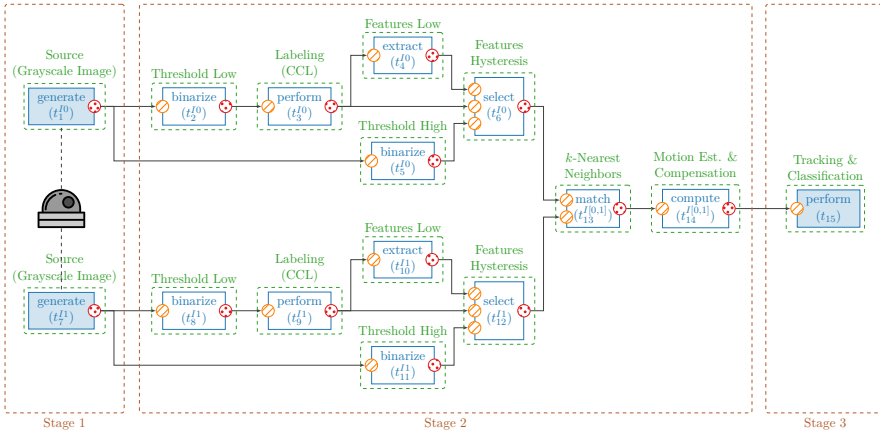
4 Evaluation on Real-world Applications


- A new computer vision application for **meteor detection**[1]
  - 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 (Paris's Observatory) joint-team

[1] C. Ciocan, M. Kandeepan, A. Cassagne, J. Vaubaillon, F. Zander, and L. Lacassagne. "Une nouvelle application de détection de météores robuste aux mouvements de caméra". In: *GRETSI*. 2023. DOI: 10.48550/arXiv.2309.06027.

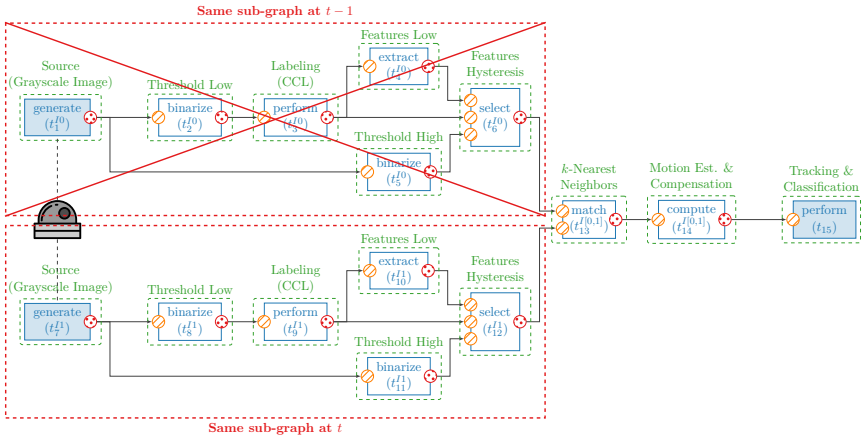
# Meteor Detection – Task Graph & Stages

4 Evaluation on Real-world Applications



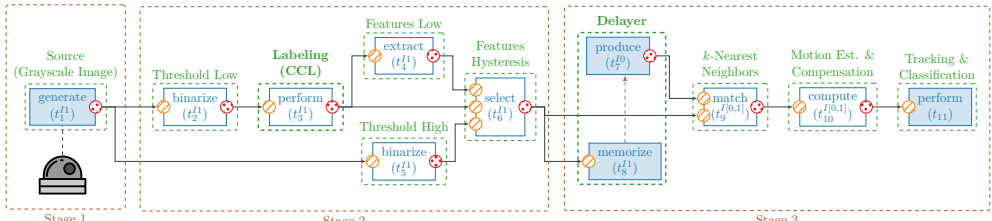
# Meteor Detection – Optimization (1)

4 Evaluation on Real-world Applications



# Meteor Detection – Optimization (2)





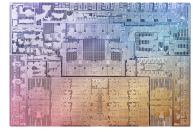

- **Delayer** module: **memorize** data at $t$ and **produce** it at $t + 1$
  - — **produce** task is triggered before **memorize** task (= stateful)
- Stage 2 takes **95% of the total time** in sequential and **can be replicated**
  - — Efficient **Light Speed Labeling** (LSL) algorithm[1] is used for labeling

[1] L. Lacassagne and B. Zavidovique. "Light Speed Labeling for RISC Architectures". In: *International Conference on Image Analysis and Processing*. IEEE, 2009. DOI: 10.1109/ICIP.2009.5414352.

# Meteor Detection – Testbed

## 4 Evaluation on Real-world Applications

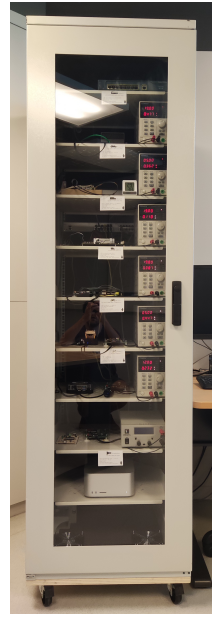


| Ref. | Name | Date | Proc. | CPUs | Freq. | RAM (Size & T/P) | |
|---|---|---|---|---|---|---|---|
| XU4 | Hardkernel Odroid-XU4 | 2016 | 28 nm | 4 × *LITTLE* ARMv7 Cortex-A7<br>4 × *Big* ARMv7 Cortex-A15 | 1.4 GHz<br>1.5 GHz | 2 GB | 3.5 GB/s |
| RPi4 | Raspberry Pi 4 model B | 2019 | 28 nm | 4 × *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 |
| M1U | Apple Silicon M1 Ultra | 2022 | 5 nm | 4 × *e-core* ARMv8 Icestorm<br>16 × *p-core* ARMv8 Firestorm | ≈ 2.0 GHz<br>≈ 3.0 GHz | 64 GB | 344.0 GB/s |

Specifications of the tested SoCs.

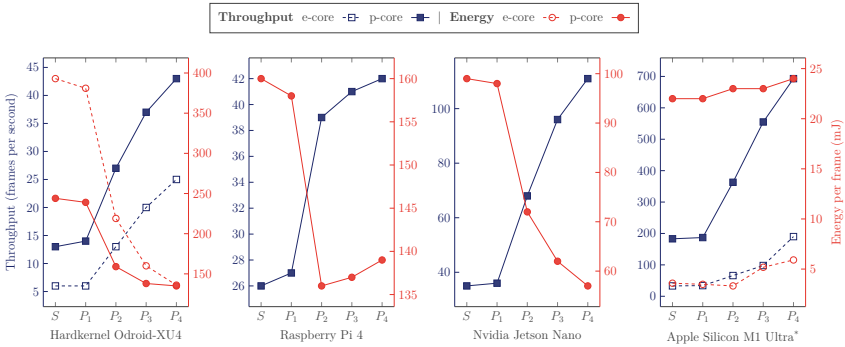
- Dedicated platform for energy measurement of embedded SoCs
  - — `https://monolithe.proj.lip6.fr`



The Monolithe.

# Meteor Detection – Performance

4 Evaluation on Real-world Applications

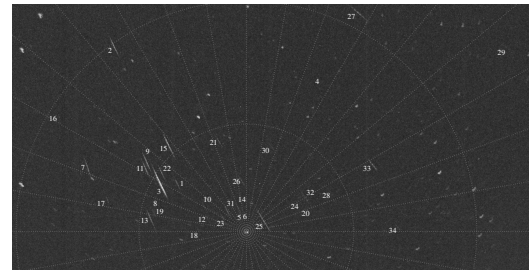



- Full HD Video ($1920 \times 1080$ pixels)
- $S$ = sequential, $P_i$ = pipeline with $i$ replications (= threads) in stage 2
  - XU4, RPi4 & Nano: consumption of the whole system (CPU, RAM, MB, ...)
  - M1U: consumption of the CPU only

# Meteor Detection – Summary



- Matches **real-time constraints** on SoCs[1]
  - $\geq$ 25 FPS, $\leq$ 10 Watts
- Contributed to the detection of **one of the largest meteor cluster** observed to date[2]
- Open source implementation
  - `https://github.com/alsoc/fmdt`



34 $\tau$-Herculids meteor cluster from aircraft.

[1] M. Kandeepan, C. Ciocan, A. Cassagne, and L. Lacassagne. "Parallélisation d'une nouvelle application embarquée pour la détection automatique de météores". In: *COMPAS*. 2023. DOI: 10.48550/arXiv.2307.10632.

[2] J. Vaubaillon et al. "A 2022 $\tau$-Herculid Meteor Cluster from an Airborne Experiment: Automated Detection, Characterization, and Consequences for Meteoroids". In: *Astronomy and Astrophysics* (2023). DOI: 10.1051/0004-6361/202244993.

# DVB-S2 Transceiver

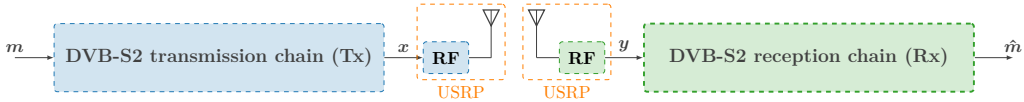
4 Evaluation on Real-world Applications

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

  ➔ **SDR on multicore and SIMD CPUs**

# DVB-S2 Transceiver – Setup & Objectives

4 Evaluation on Real-world Applications



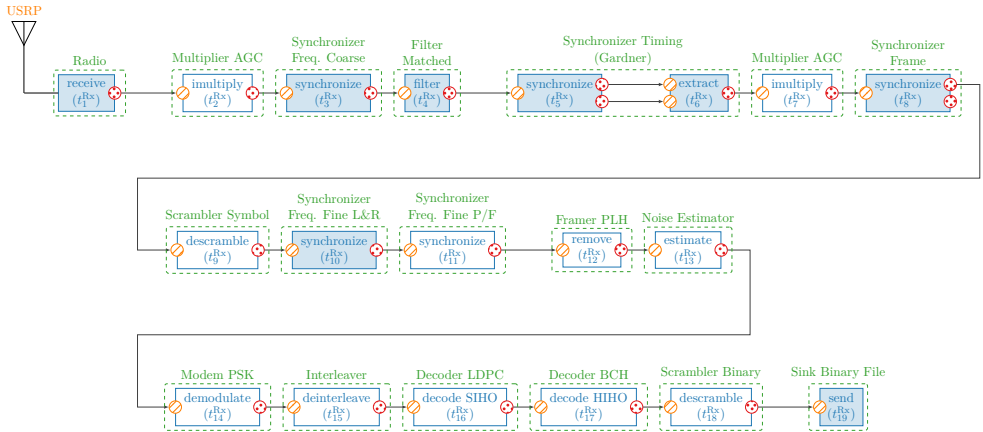

- 1x Middle class computer for the digital transmitter (Tx)
- 1x Server class computer for **the digital receiver (Rx)**
- 2x Universal Software Radio Peripherals (USRPs) N320 for the RF
- **Industrial real-time constraint**: 30 $\sim$ 50 Mb/s

| Config. | Modulation | Rate $R$ | $K_{\text{BCH}}$ | $K_{\text{LDPC}}$ | $N_{\text{LDPC}}$ | $\mathcal{T}_i$ (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).

# DVB-S2 Transceiver – Rx Task Graph





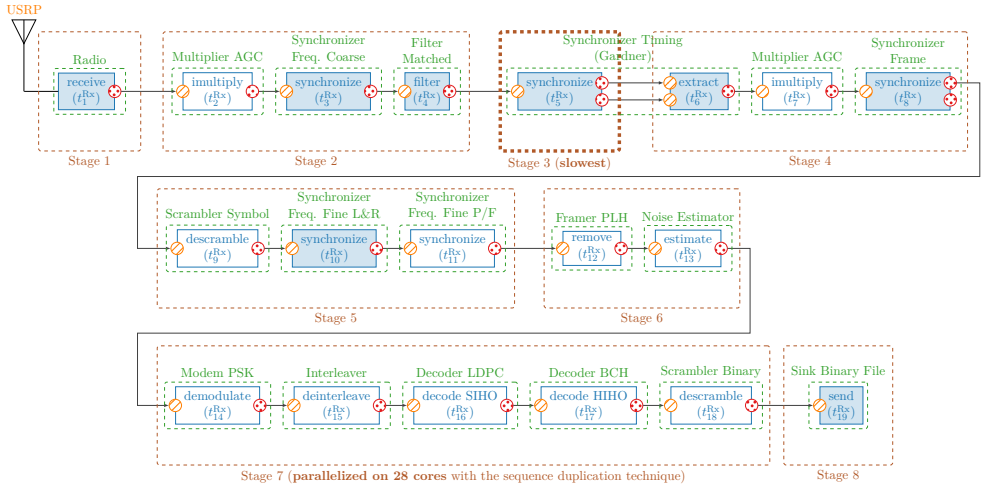
# DVB-S2 Transceiver – Rx Tasks Duration

4 Evaluation on Real-world Applications

| | Throughput (MS/s) | | | | Latency ($\mu s$) | | | Time |
|---|---|---|---|---|---|---|---|---|
| Stages and Tasks | Avg | Min. | Max. | $\mathcal{N}_{\text{Avg}}$ | Avg | Min. | Max. | (%) |
| Radio - *receive* ($t_1^{\text{Rx}}$) | 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* ($t_2^{\text{Rx}}$) | 864.41 | 420.05 | 935.71 | 367.45 | 619.71 | 572.49 | 1275.28 | 1.11 |
| Synch. Freq. Coarse - *synchronize* ($t_3^{\text{Rx}}$) | 1979.17 | 665.98 | 2237.38 | 841.32 | 270.66 | 239.42 | 804.35 | 0.48 |
| Filter Matched - *filter* ($t_4^{\text{Rx}}$) | 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* ($t_5^{\text{Rx}}$) | 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* ($t_6^{\text{Rx}}$) | 331.50 | 151.54 | 354.62 | 281.83 | 807.97 | 755.28 | 1767.48 | 1.44 |
| Multiplier AGC - *imultiply* ($t_7^{\text{Rx}}$) | 806.31 | 442.69 | 877.19 | 685.51 | 332.18 | 305.34 | 605.02 | 0.59 |
| Synch. Frame - *synchronize* ($t_8^{\text{Rx}}$) | 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* ($t_9^{\text{Rx}}$) | 1979.41 | 668.85 | 2649.55 | 1682.89 | 135.31 | 101.09 | 400.45 | 0.24 |
| Synch. Freq. Fine L&R - *synchronize* ($t_{10}^{\text{Rx}}$) | 1466.55 | 596.19 | 1741.72 | 1246.85 | 182.63 | 153.78 | 449.25 | 0.33 |
| Synch. Freq. Fine P/F - *synchronize* ($t_{11}^{\text{Rx}}$) | 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* ($t_{12}^{\text{Rx}}$) | 1148.07 | 427.71 | 1180.59 | 1008.60 | 225.77 | 219.55 | 606.02 | 0.40 |
| Noise Estimator - *estimate* ($t_{13}^{\text{Rx}}$) | 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* ($t_{14}^{\text{Rx}}$) | 46.07 | 42.12 | 46.28 | 40.47 | 5626.34 | 5600.83 | 6153.50 | 10.05 |
| Interleaver - *deinterleave* ($t_{15}^{\text{Rx}}$) | 1533.54 | 518.95 | 1582.97 | 1347.25 | 169.02 | 163.74 | 499.47 | 0.30 |
| Decoder LDPC - *decode SIHO* ($t_{16}^{\text{Rx}}$) | 166.15 | 69.12 | 171.59 | 164.21 | 1386.74 | 1342.74 | 3333.34 | 2.48 |
| Decoder BCH - *decode HIHO* ($t_{17}^{\text{Rx}}$) | 6.92 | 6.15 | 6.96 | 6.92 | 32905.37 | 32705.15 | 36998.15 | 58.79 |
| Scrambler Binary - *descramble* ($t_{18}^{\text{Rx}}$) | 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* ($t_{19}^{\text{Rx}}$) | 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 |

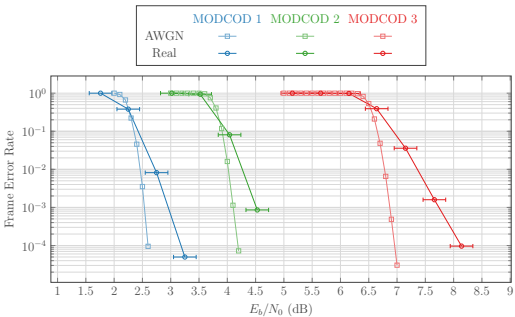
# DVB-S2 Transceiver – Rx Pipeline Stages

4 Evaluation on Real-world Applications

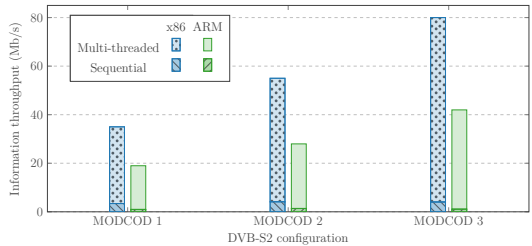


# DVB-S2 Transceiver – Rx Performance

4 Evaluation on Real-world Applications




Decoding performance.




Throughput performance.

- 2× Intel® Xeon™ Platinum 8168 x86 CPUs @ 2.70 GHz (35/48 cores used)
  - — **Matches the industrial real-time constraint** (from 35 to 80 Mb/s)
- 2 × Cavium ThunderX2® CN9975 v2.1 CPUs @ 2.00 GHz (52/56 cores used)

# DVB-S2 Transceiver – Summary



- DVB-S2: a digital **communication standard** for satellites
- Tested and validated on **real radios** (USRPs)[1]
  - **3.5 times faster** than the GNU Radio implementation
- **Used in industrial context** as an SDR demonstrator
- **Open source implementation** is available on GitHub
  - `https://github.com/aff3ct/dvbs2`

[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*. IEEE, 2021. DOI: 10.1109/ISTC49272.2021.9594063.

# Overall Summary



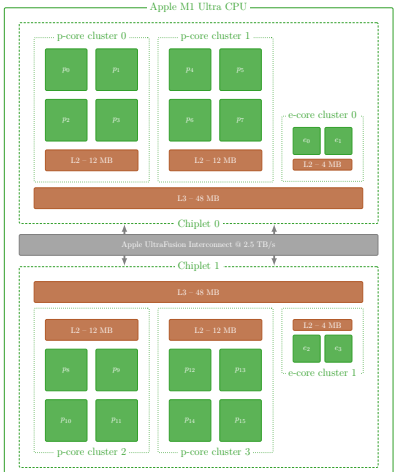
- **Real-world applications**
  - Computer vision on embedded and low power SoCs
  - Digital communications on middle class and server class computers
- Achieved **speedups** compared to sequential version
  - Meteor detection: from 1.6× to **3.8× on 4 cores**
  - DVB-S2 receiver: from 10× on 35 cores to **38× on 52 cores**
- StreamPU runtime
  - **Efficient on both low power and high-end multicore CPUs**
  - **Portable** over a large variety of architectures
  - Open source, tested and documented: `https://github.com/aff3ct/streampu`

# Table of Contents

5 Ongoing Works

# DVB-S2 Receiver on Heterogeneous SoCs

5 Ongoing Works



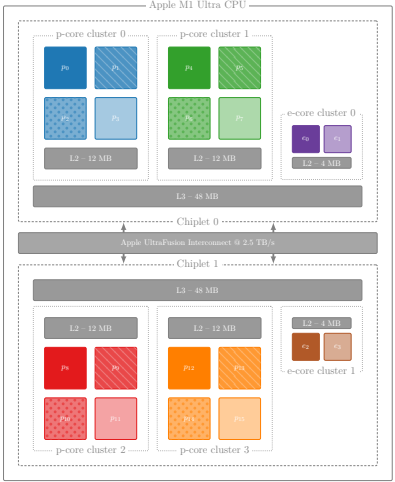

- Apple Silicon M1 Ultra
  - **Strongly heterogeneous architecture**
  - Running on Asahi Linux (kernel 6.6)
  - Thread pinning is enabled

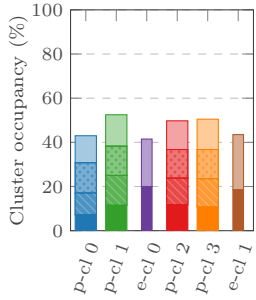
- Case study: DVB-S2 receiver
  - MODCOD 2 (QPSK, $R = 8/9$, $K = 14232$)
  - Fixed 13-stage task graph decomposition

➔ **Is it possible to take advantage of the p-cores and e-cores together?**

# DVB-S2 Pinning Strategies & Performance (1)

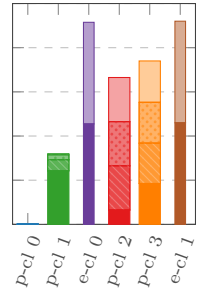
5 Ongoing Works



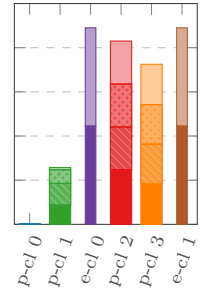

OS scheduling

T/P = 54.5 Mb/s
Power = 32 W
$\mathcal{E}$/fra = 8.0 mJ




Pinning for T/P

T/P = **56.0 Mb/s**
Power = 30 W
$\mathcal{E}$/fra = 7.3 mJ



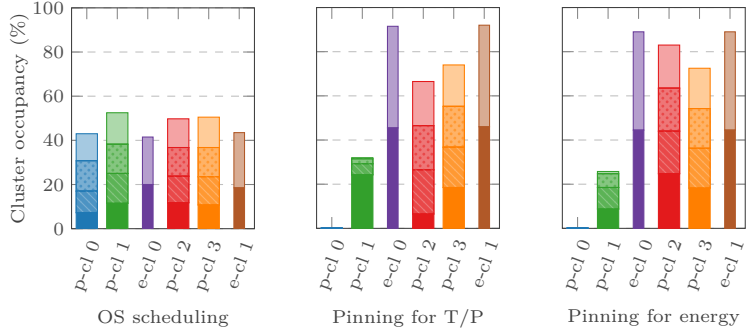

Pinning for energy

T/P = 53.6 Mb/s
Power = 26 W
$\mathcal{E}$/fra = **6.6 mJ**

# DVB-S2 Pinning Strategies & Performance (2)



- **Energy of entire system**
- Pinning strategies
  - Avoid thread migrations
  - **Maximize e-cores usage**
  - Pin threads according to pipeline stages locality
  - Put slowest stage on p-core
- Compared to OS scheduling
  - Throughput gain: +3%, Energy efficiency: +10%
  - Throughput gain: -1.5%, Energy efficiency: **+20%**

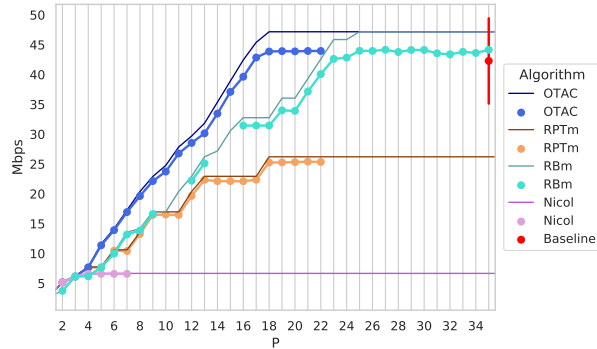



| OS scheduling | Pinning for T/P | Pinning for energy |
|---|---|---|
| T/P = 54.5 Mb/s | T/P = **56.0 Mb/s** | T/P = 53.6 Mb/s |
| Power = 32 W | Power = 30 W | Power = 26 W |
| $\mathcal{E}$/fra = 8.0 mJ | $\mathcal{E}$/fra = 7.3 mJ | $\mathcal{E}$/fra = **6.6 mJ** |

# Optimal Task Chains Pipeline Decomposition



- **OTAC: Optimal Algorithm for Task Chains Scheduling**[1]
- RPTm, RBm, Nicol: other scheduling algorithms
- $P$ is the number of CPU cores
- Solid lines: **Estimation**
- Solid lines with points: **Evaluation**
  - 2× Intel® Xeon™ Skylake Gold 6240 x86 CPUs @ 2.60 GHz (36 cores)




[1] D. Orhan, L. Lima Pilla, D. Barthou, A. Cassagne, O. Aumage, R. Tajan, C. Jégo, and C. Leroux. "OTAC: Optimal Scheduling for Pipelined and Replicated Task Chains for Software-Defined Radio". Preprint. 2023. URL: https://hal.science/hal-04228117.

# Table of Contents

# Conclusion

6 Conclusion & Future Works

- StreamPU: A new **DSL embedded into C++**
  - Interpreted but with a **low overhead**
  - **Facilitate the adaptation** of C/C++ existing codes
- Evaluated on **two real** streaming **classes of application**
  - **Computer vision**: a meteor detection chain
  - **Software-defined radio**: a DVB-S2 transceiver
- Suitable for various **multi-core CPUs**
  - Server-class computers
  - Embedded SoCs

# Future Works



- **Automatize thread pinning** on heterogeneous CPUs
- **Predict energy consumption** from pipeline configuration (stages, pinning)
- Manage **new types of process units like GPUs and NPUs** (SYCL?)
- **Integrate Rust language** to match **critical applications** requirements
- Adapt StreamPU for efficient **inference of deep neural networks**

# *Q&A*

*Thank you for listening!*
*Do you have any questions?*

# Bibliography

7 References


[1] W. Thies, M. Karczmarek, and S. Amarasinghe. “StreamIt: A Language for Streaming Applications”. In: *Compiler Construction*. Springer, 2002. DOI: 10.1007/3-540-45937-5_14.

[2] C. Glitia, P. Dumont, and P. Boulet. “Array-OL with Delays, a Domain Specific Specification Language for Multidimensional Intensive Signal Processing”. In: *Springer Multidimensional Systems and Signal Processing* (2010). DOI: 10.1007/s11045-009-0085-4.

[3] 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* (2023). DOI: 10.1002/cpe.7820.

[4] C. Augonnet, S. Thibault, R. Namyst, and P.-A. Wacrenier. “StarPU: A Unified Platform for Task Scheduling on Heterogeneous Multicore Architectures”. In: *Wiley Concurrency and Computation: Practice and Experience* (2011). DOI: 10.1002/cpe.1631.

[5] C. Ciocan, M. Kandeepan, A. Cassagne, J. Vaubaillon, F. Zander, and L. Lacassagne. “Une nouvelle application de détection de météores robuste aux mouvements de caméra”. In: *GRETSI*. 2023. DOI: 10.48550/arXiv.2309.06027.

# Bibliography

7 References


[6] L. Lacassagne and B. Zavidovique. “Light Speed Labeling for RISC Architectures”. In: *International Conference on Image Analysis and Processing*. IEEE, 2009. DOI: 10.1109/ICIP.2009.5414352.

[7] M. Kandeepan, C. Ciocan, A. Cassagne, and L. Lacassagne. “Parallélisation d’une nouvelle application embarquée pour la détection automatique de météores”. In: *COMPAS*. 2023. DOI: 10.48550/arXiv.2307.10632.

[8] 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 $\tau$-Herculid Meteor Cluster from an Airborne Experiment: Automated Detection, Characterization, and Consequences for Meteoroids”. In: *Astronomy and Astrophysics* (2023). DOI: 10.1051/0004-6361/202244993.

[9] 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*. IEEE, 2021. DOI: 10.1109/ISTC49272.2021.9594063.

# Bibliography

7 References

[10] D. Orhan, L. Lima Pilla, D. Barthou, A. Cassagne, O. Aumage, R. Tajan, C. Jégo, and C. Leroux. "OTAC: Optimal Scheduling for Pipelined and Replicated Task Chains for Software-Defined Radio". Preprint. 2023. URL: https://hal.science/hal-04228117.