SCIENCES

Electronics Engineering, Field Director – Francis Balestra

Design Methodologies and Architecture, Subject Head – Ahmed Jerraya

Multi-Processor System-on-Chip 2

Applications

Coordinated by

Liliana Andrade
Frédéric Rousseau

Color Section

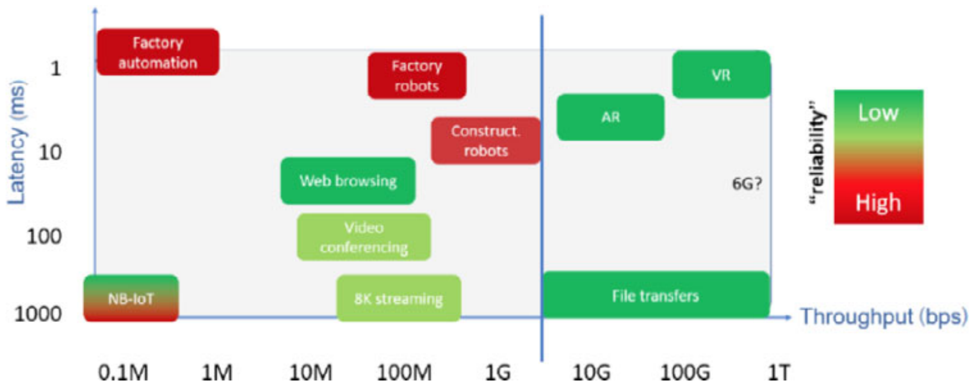


Figure 1.1. *Application mapping on the rate–latency plane with regard to the reliability requirement (Fettweis et al. 2019)*

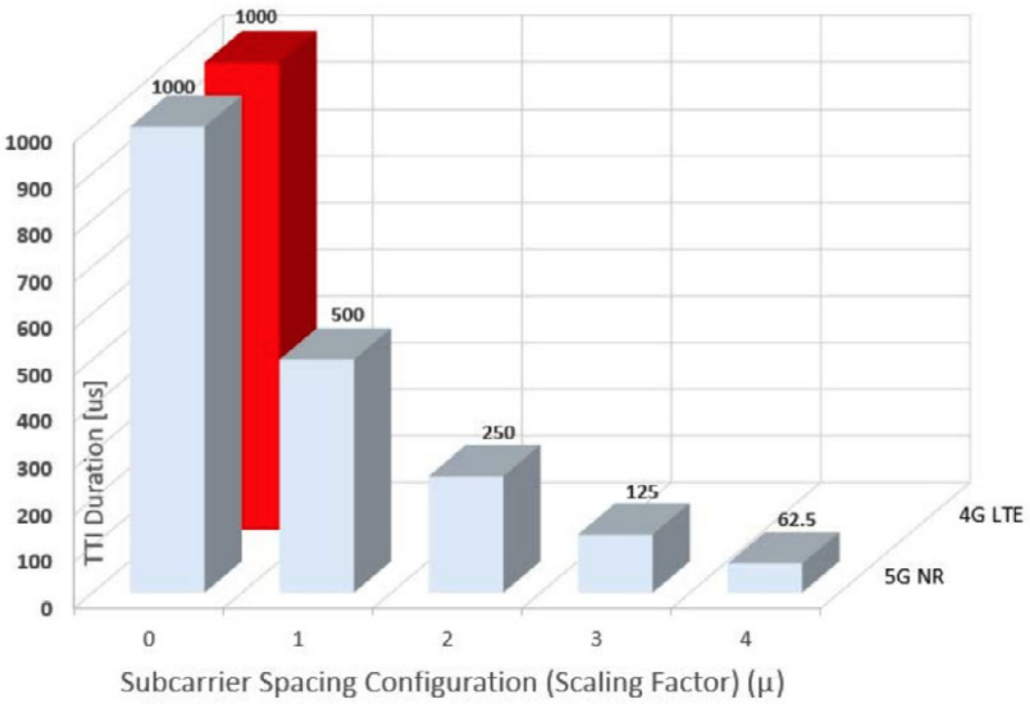


Figure 1.2. *Comparing 14 OFDM symbols' TTI duration of 4G and 5G*

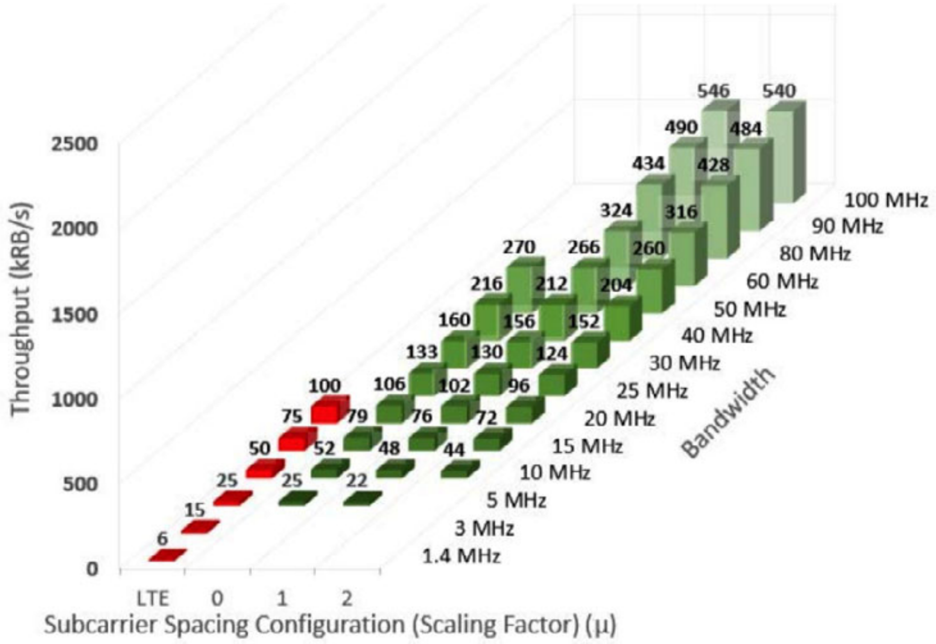


Figure 1.3. *Processing load in kRB/s for 5G NR FR1 (Damjancevic et al. 2019)*

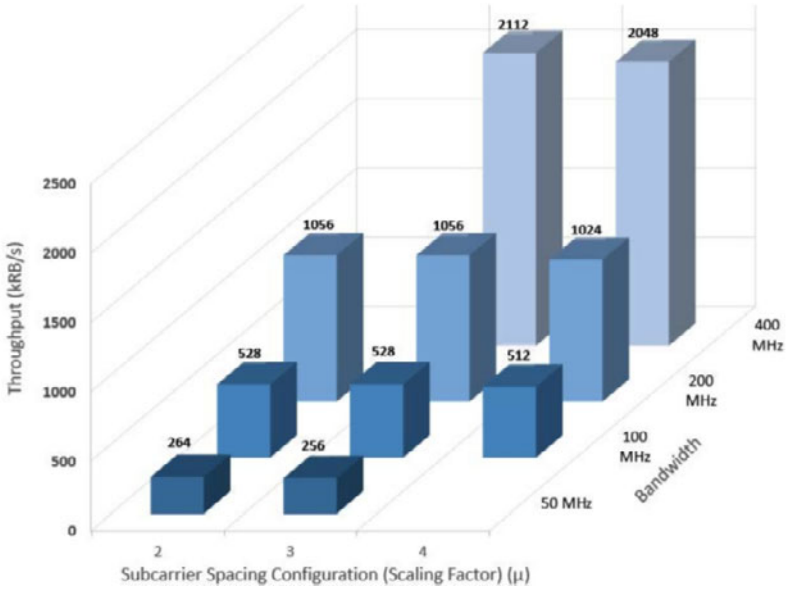


Figure 1.4. *Processing load in kRB/s for 5G NR FR2*

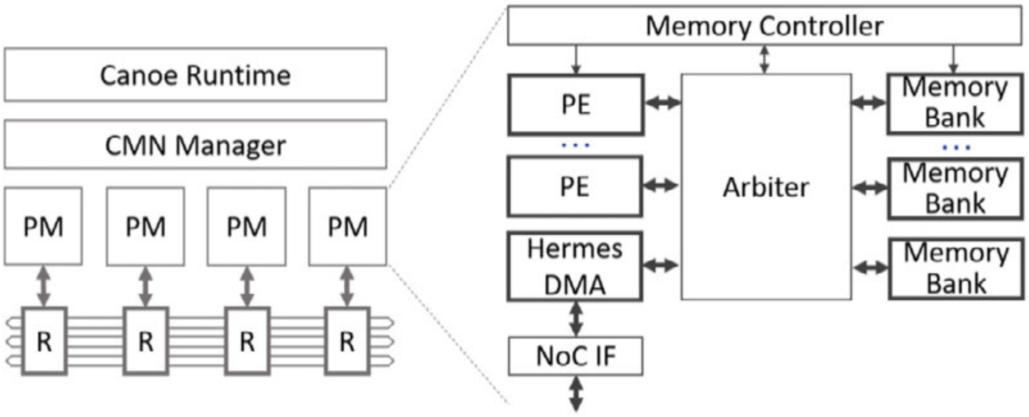


Figure 1.5. *Tiled "Kachel" MPSoC with decentralized tightly coupled memories*

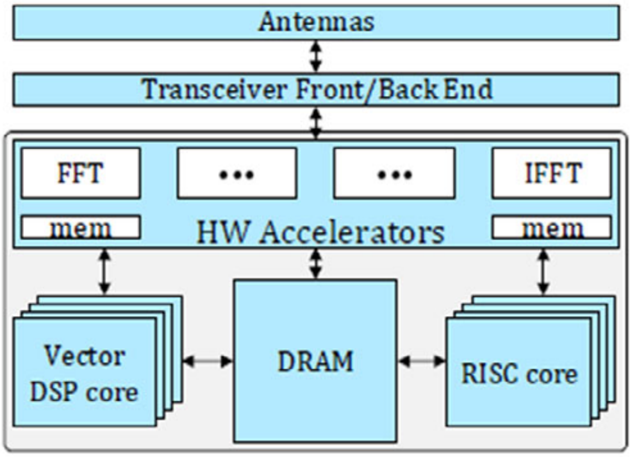


Figure 1.6. *Heterogeneous MPSoC with a central shared memory architecture*

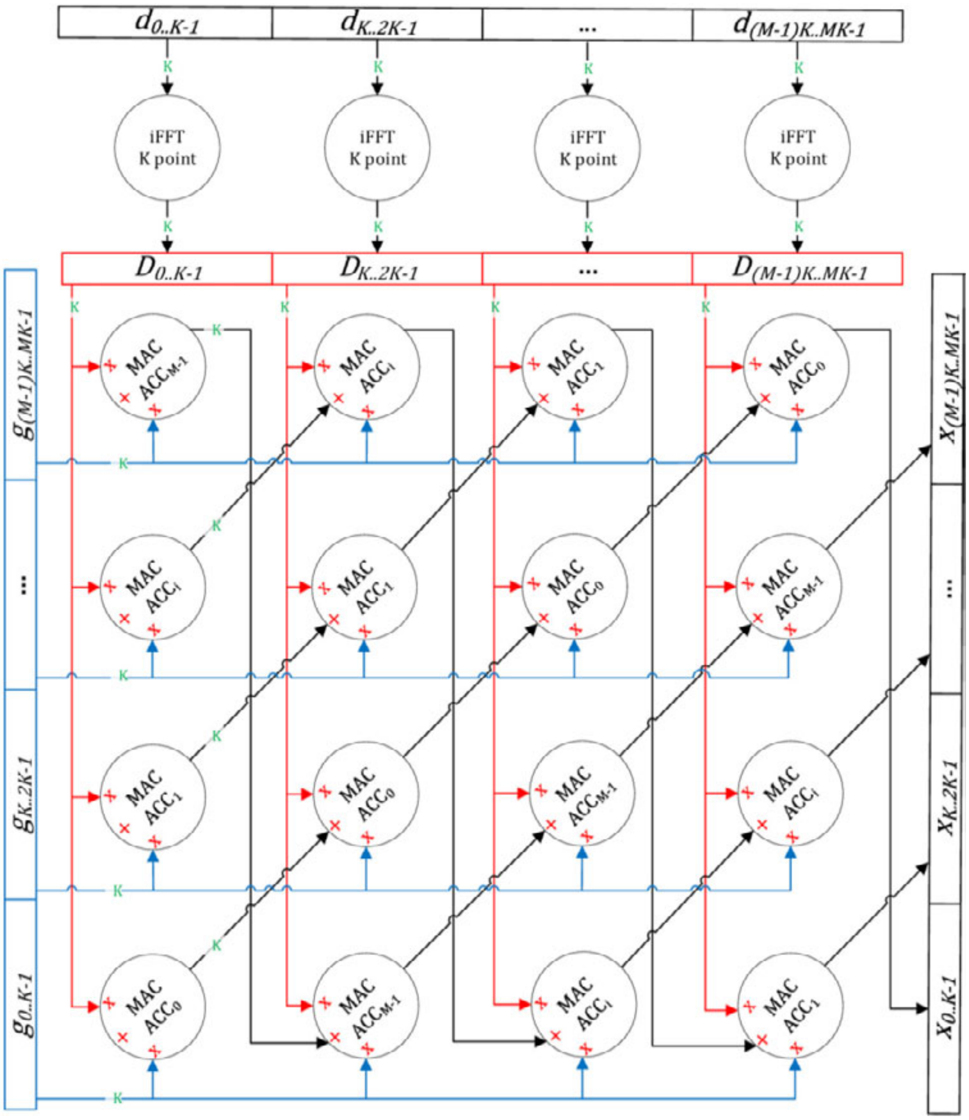


Figure 1.7. *GFDM processing dataflow diagram*

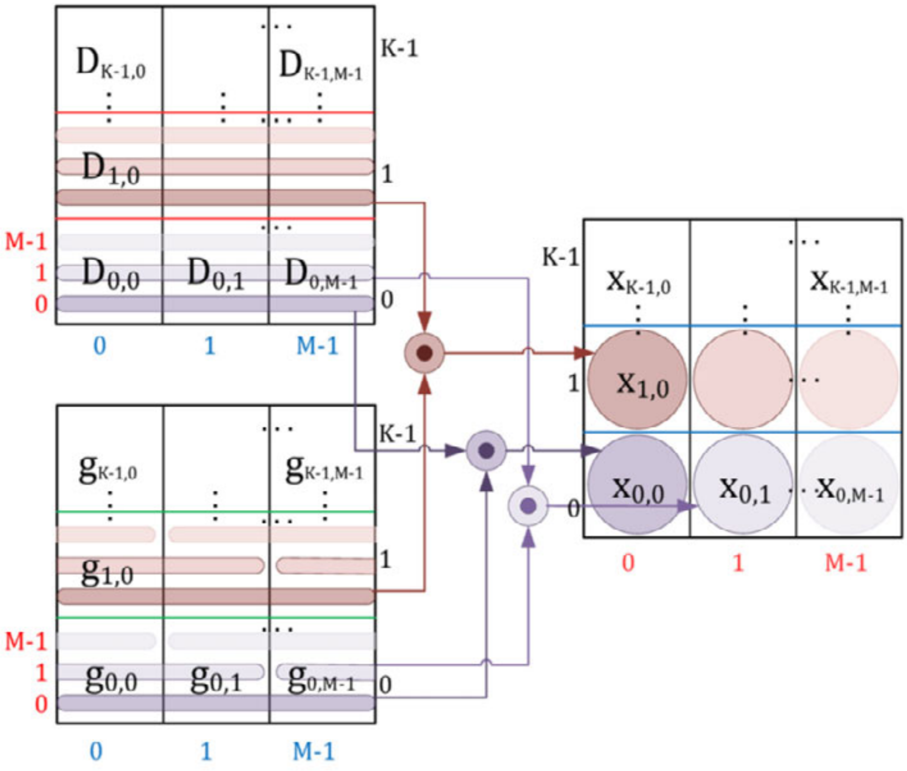


Figure 1.8. *Visualization of time-domain GFDM filtering*

```
input : IDFT output D and filter coefficients g
output: GFDM signal x
// Assume complex operations *,+=
for $l \leftarrow M-1$ to 0 do
    for $i \leftarrow 0$ to $K-1$ do
        $ACC$ = 0;
        for $m \leftarrow 0$ to $l$ do
            // split in 2 loops to avoid modulo operations
            $ACC += D[mK+i] * g[(M-1+m-l)K+i]$;
        end
        for $m \leftarrow l+1$ to $M-1$ do
            $ACC += D[mK+i] * g[(m-1-l)K+i]$;
        end
        $x[(M-1-l)K+i] = ACC$;
    end
end
```

Figure 1.9. *GFDM pseudo-code*

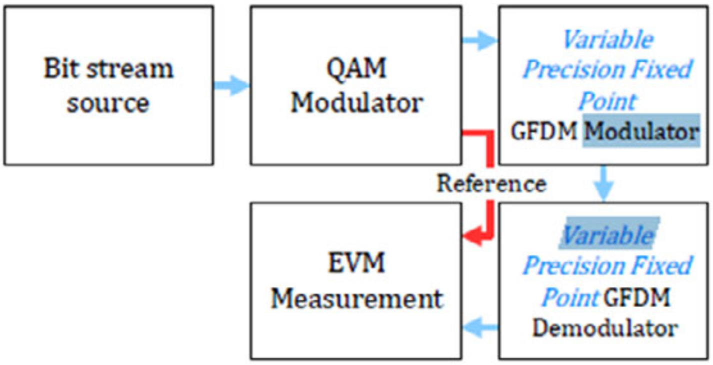

Bit stream source
QAM Modulator
Variable Precision Fixed Point GFDM Modulator
Reference
EVM Measurement
Variable Precision Fixed Point GFDM Demodulator

Figure 1.10. *Precision test bed set up*

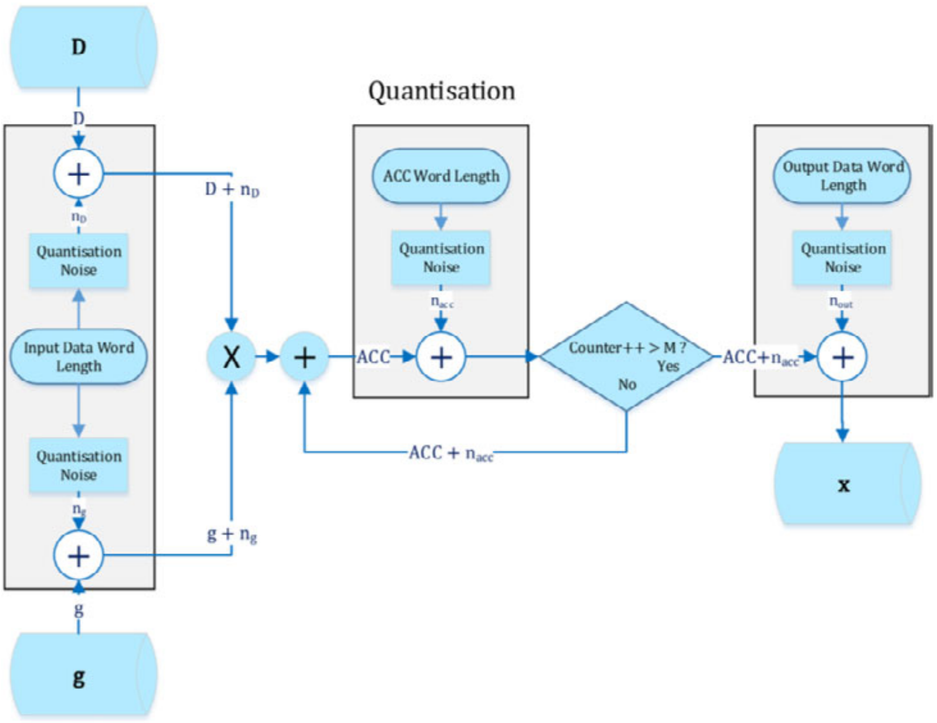


Figure 1.11. *Varied precision quantization of GFDM*

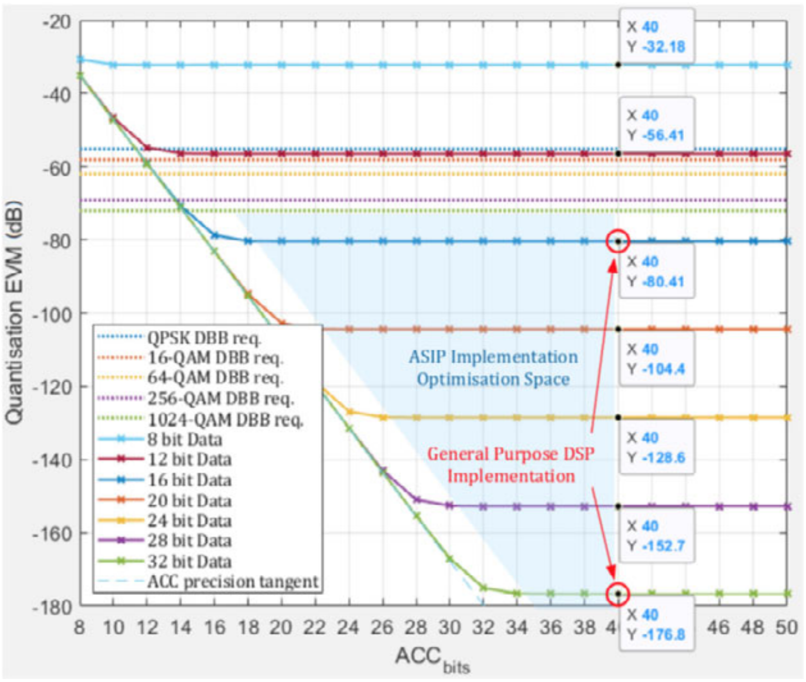


Figure 1.12. *GFDM EVM for varied data and ACC complex bit-lengths compared to adjusted 3GPP EVM DBB requirements (3GPP 2018b, 2019a)*

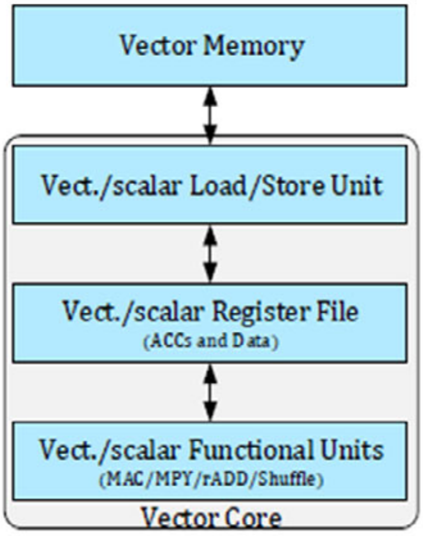

Vector Memory
Vect./scalar Load/Store Unit
Vect./scalar Register File
(ACCs and Data)
Vect./scalar Functional Units
(MAC/MPY/rADD/Shuffle)
Vector Core

Figure 1.13. *vDSP simplified HW block diagram*

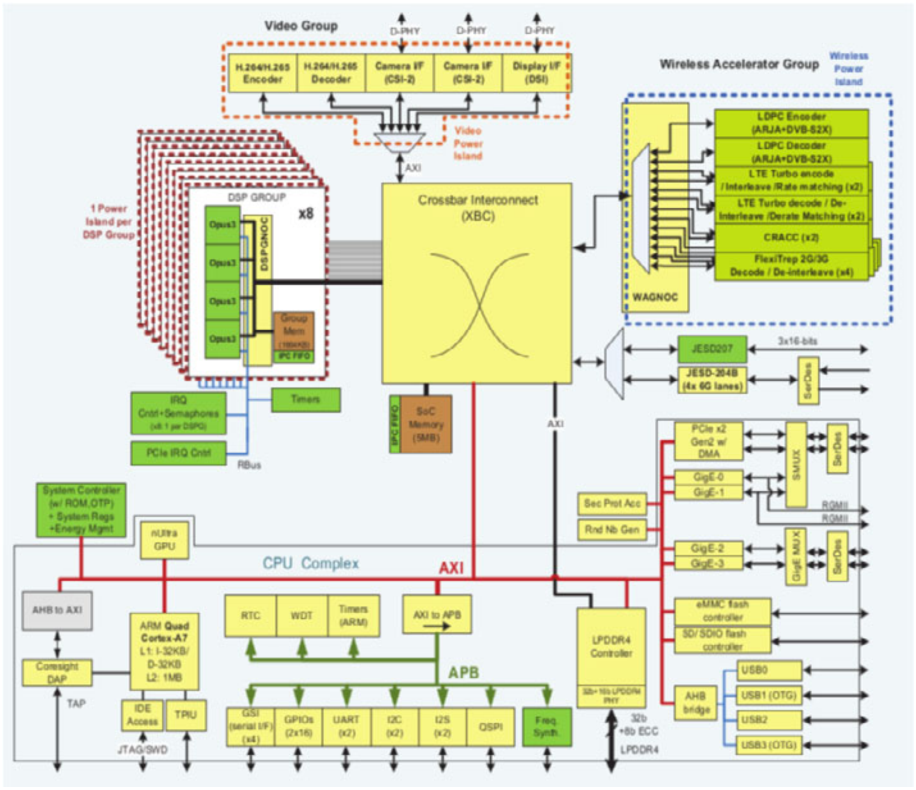


Figure 2.1. *State-of-the-art commercial system-on-chip baseband architecture*

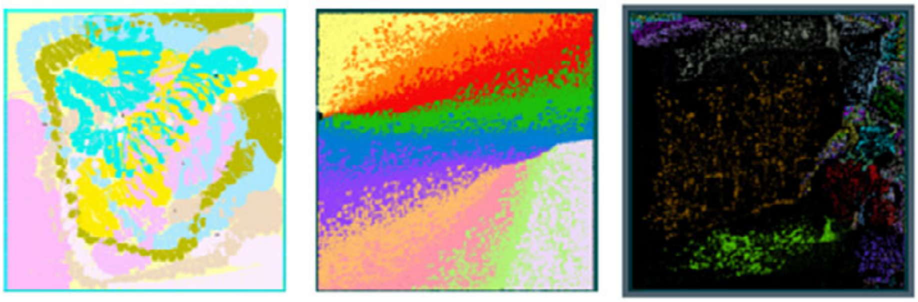


Figure 2.2. *Left:* 306Gbit/s *turbo decoder. Middle:* 288 Gbit/s *LDPC decoder. Right:* 764Gbit/s *polar decoder*

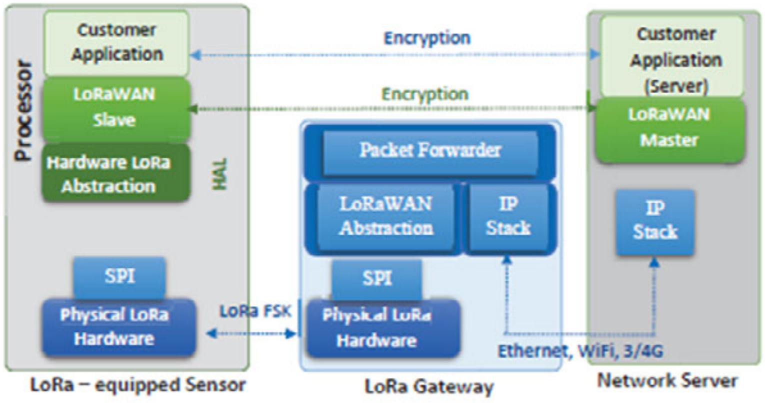


Figure 3.1. *Security in LoRaWAN*

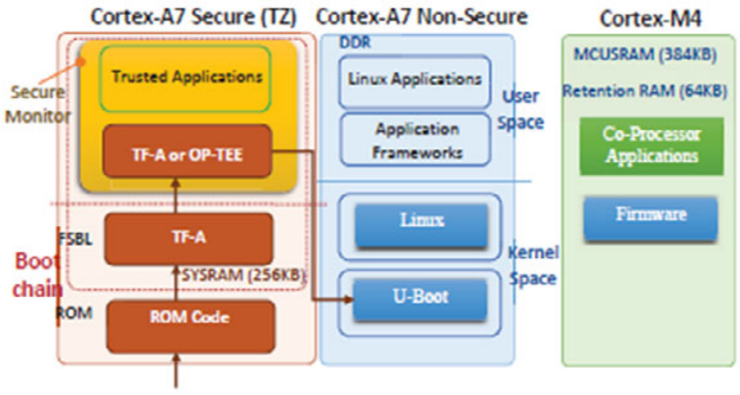


Figure 3.2. *Boot process in an STM32MP1 device*

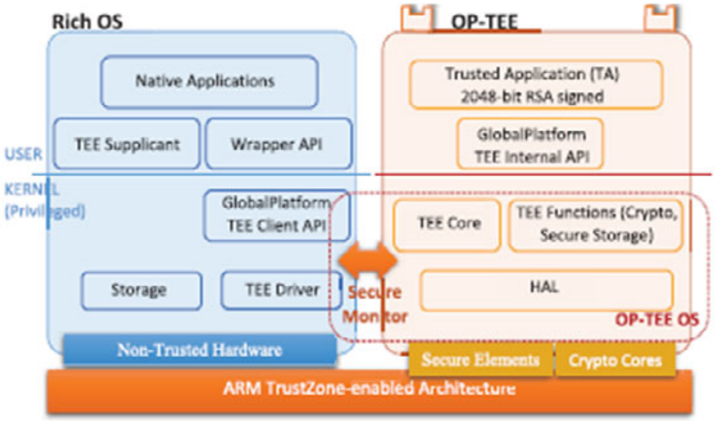


Figure 3.3. *Execution environments in OP-TEE enabled organization based on ARM TrustZone architecture*

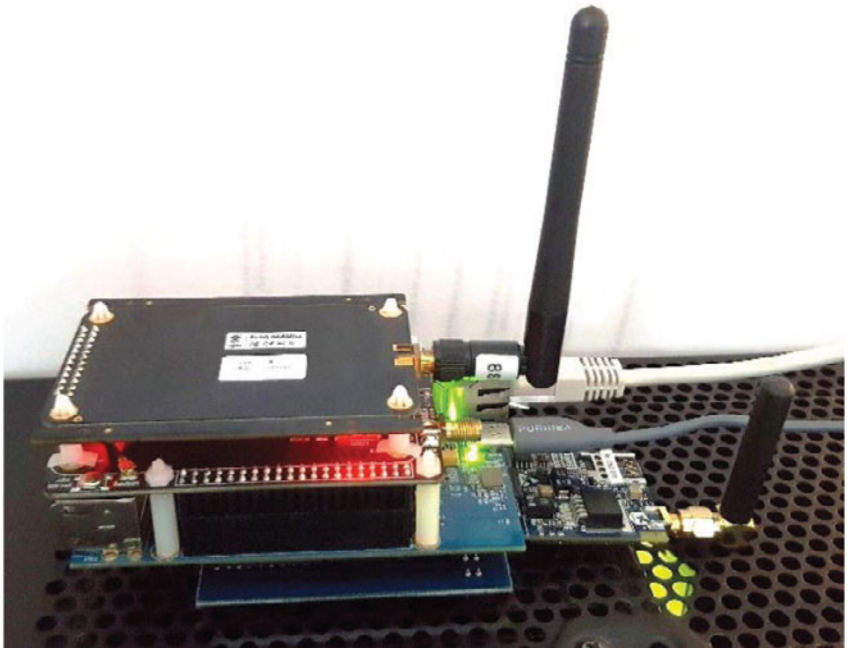


Figure 3.4. LoraWAN gateway using an RAK831 RF with a GPS (top two shields), the STM32MP1 and a 3G modem (bottom shield) when a wired connection is not available

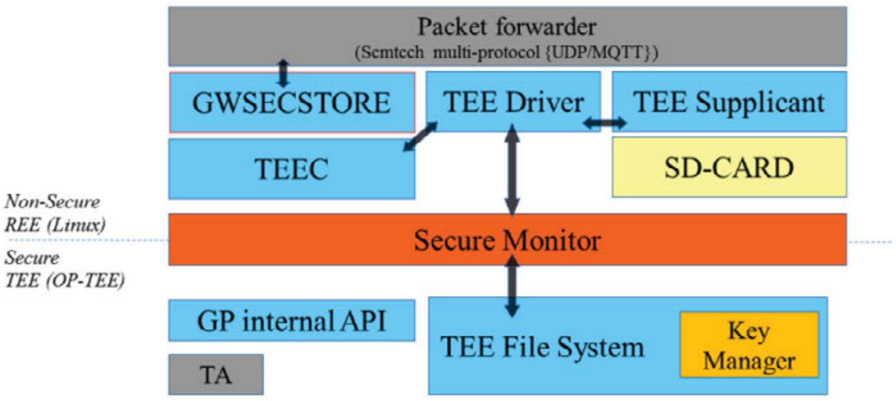


Figure 3.5. *Execution of gateway packet forwarder in OP-TEE enabled organization based on the STM32MP1 platform*

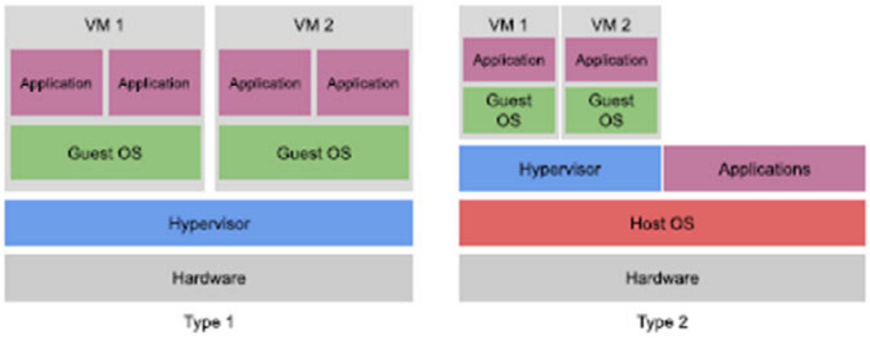


Figure 4.1. *Hypervisor types*

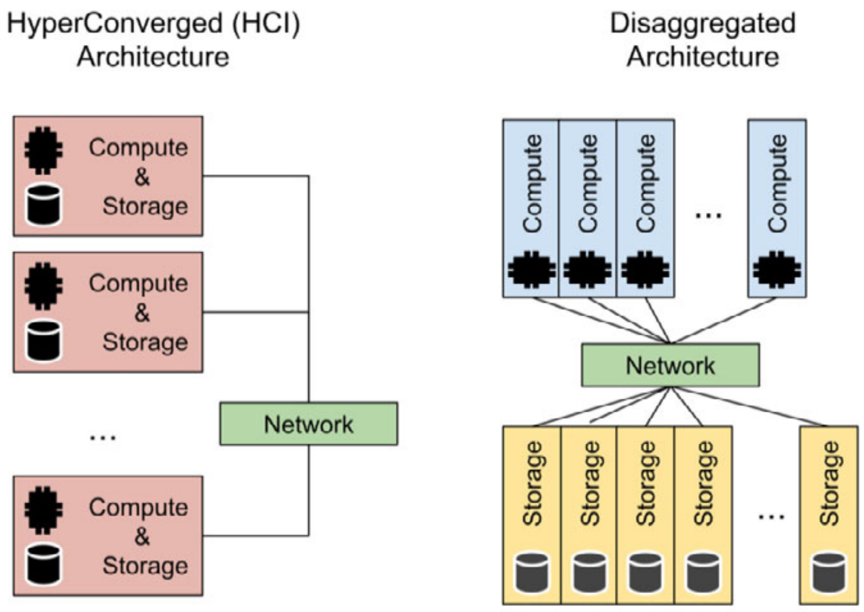


Figure 4.2. *Hyperconverged versus disaggregated architectures*

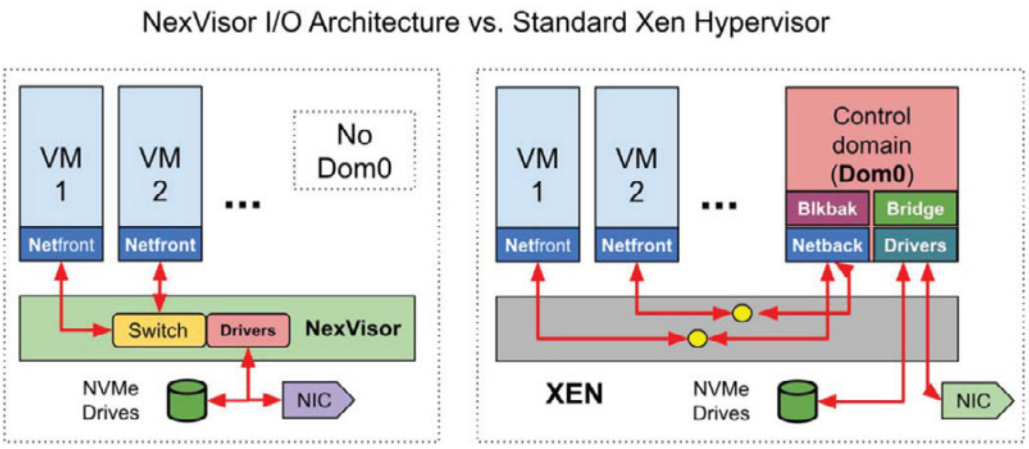


Figure 4.3. *Comparison of the NexVisor I/O architecture to standard Xen*

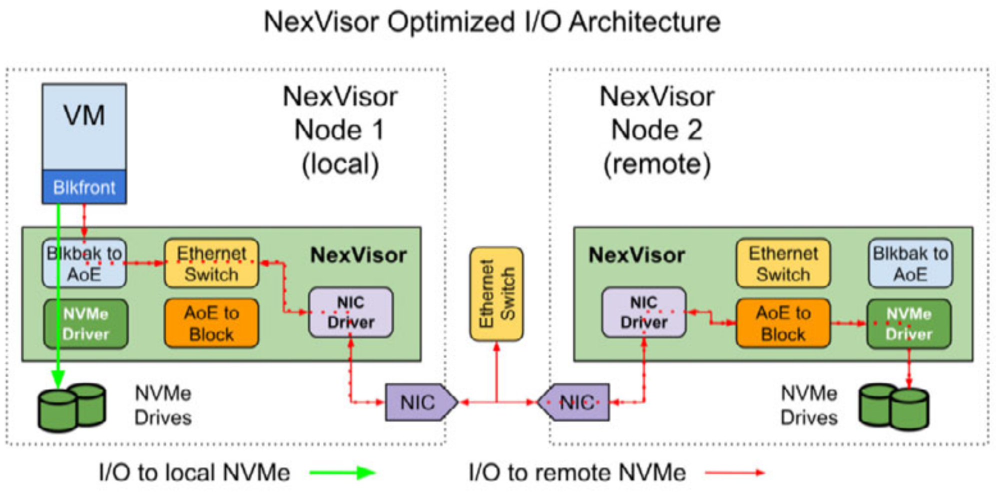


Figure 4.4. *Optimized I/O datapath operations in the NexVisor for local and remote NVMes over the network*

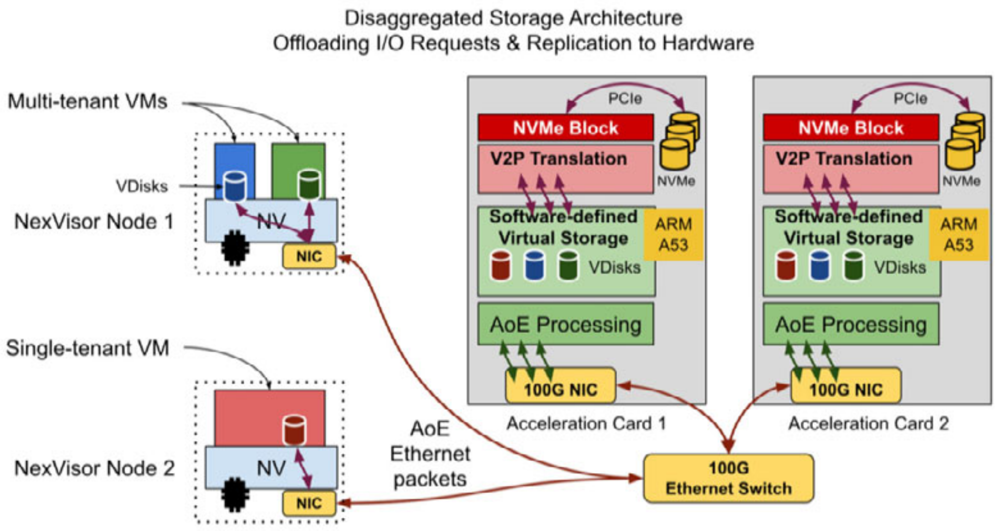


Figure 4.5. *High-level view of disaggregated storage architecture, showing the I/O requests originating from the VMs through the NexVisors and through the ethernet network to the acceleration cards*

```
      0                   1                   2                   3
      0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   0 |                    Ethernet Destination Addr                  |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   4 |   Ethernet Destination Addr   |      Ethernet Source Addr     |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
   8 |                      Ethernet Source Addr                     |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  12 |     Ethernet Type (0x88A2)    |  Ver  | Flags |     Error     |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  16 |             Major             |     Minor     |    Command    |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  20 |                              Tag                              |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
  24 |                              Arg                              |
     +-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 4.6. *ATA over Ethernet (AoE) header format (Hopkins and Coile 2009)*

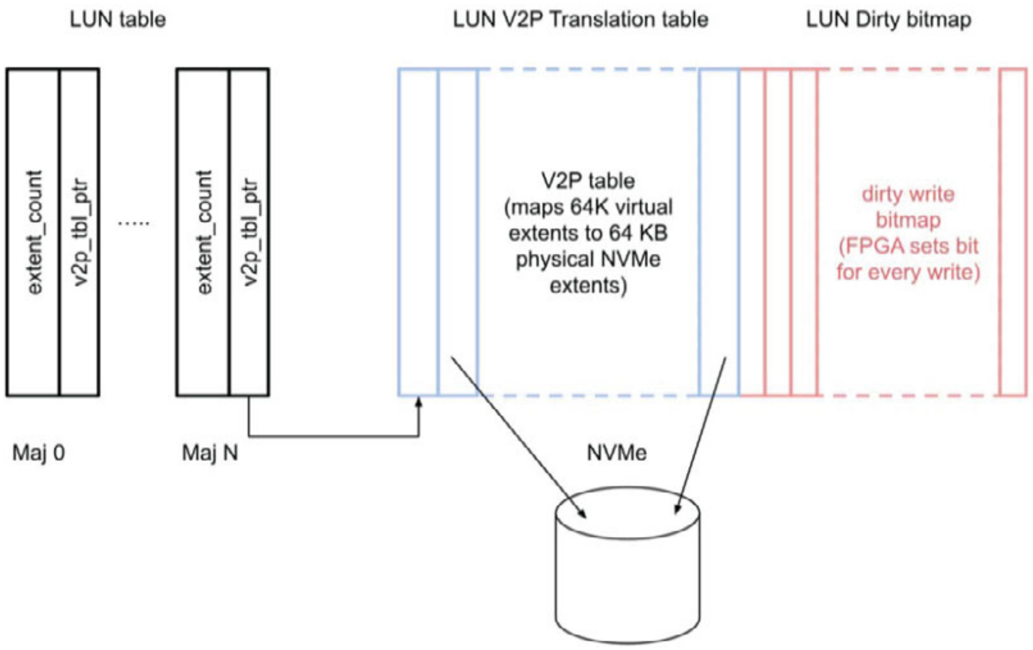


Figure 4.7. *Storage virtualization data structures used by the accelerated datapath*

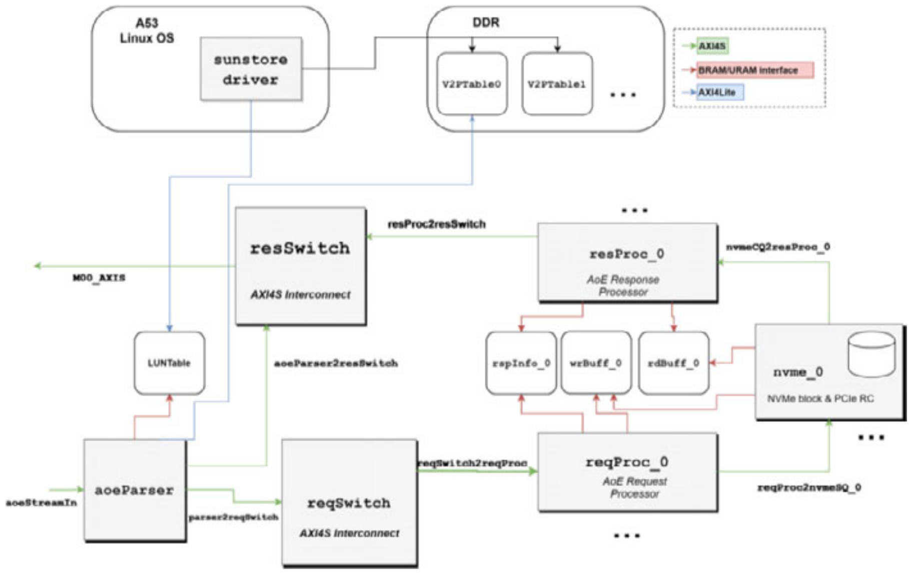


Figure 4.8. *Hardware architecture for the disaggregated storage acceleration card*

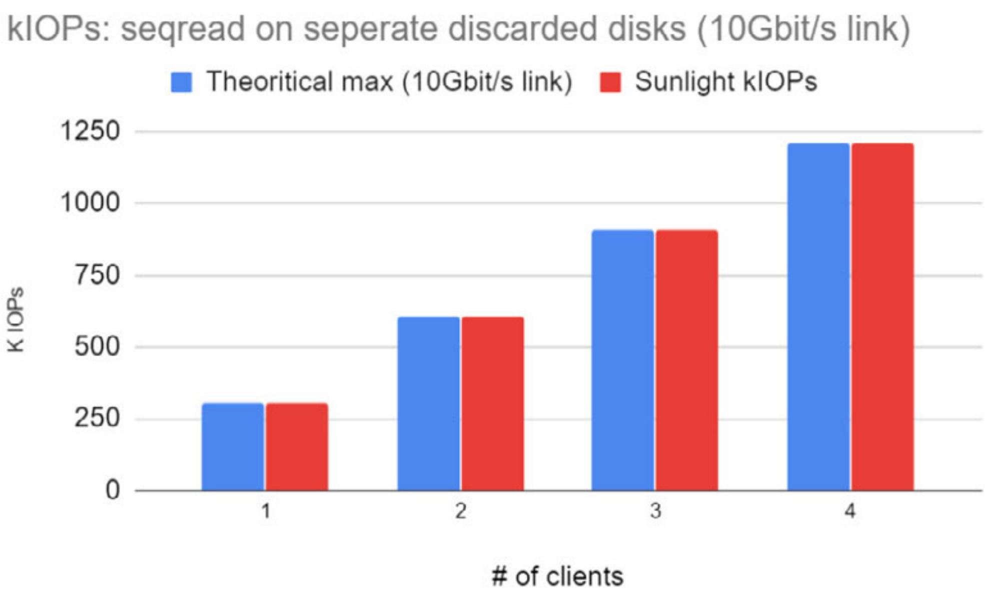


Figure 4.9. *Thousands of sequential read I/Os per second, using fio on four VM clients, each reading on a separate NVMe drive*

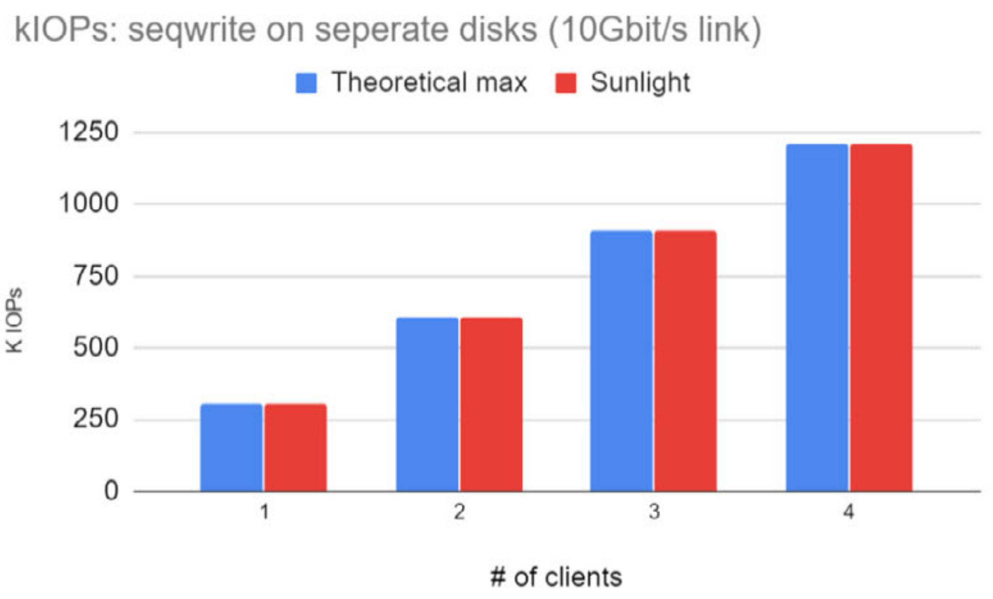


Figure 4.10. *Thousands of sequential write I/Os per second, using fio on four VM clients, each writing to a separate NVMe drive*

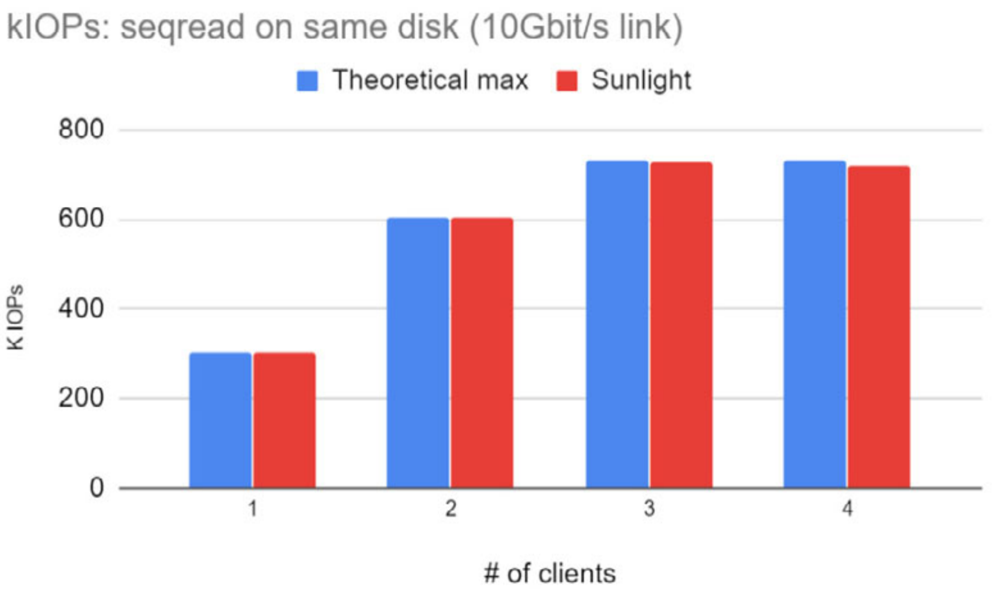


Figure 4.11. *Thousands of sequential read I/Os per second, using fio on four VM clients, all reading from the same NVMe drive*

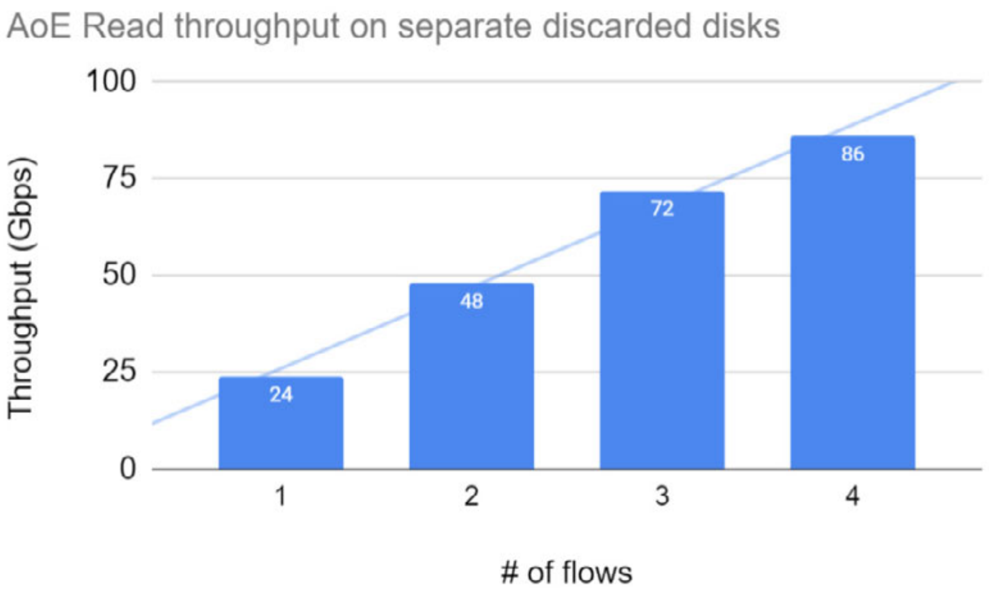


Figure 4.12. *AoE read throughput scaling for one to four client flows*

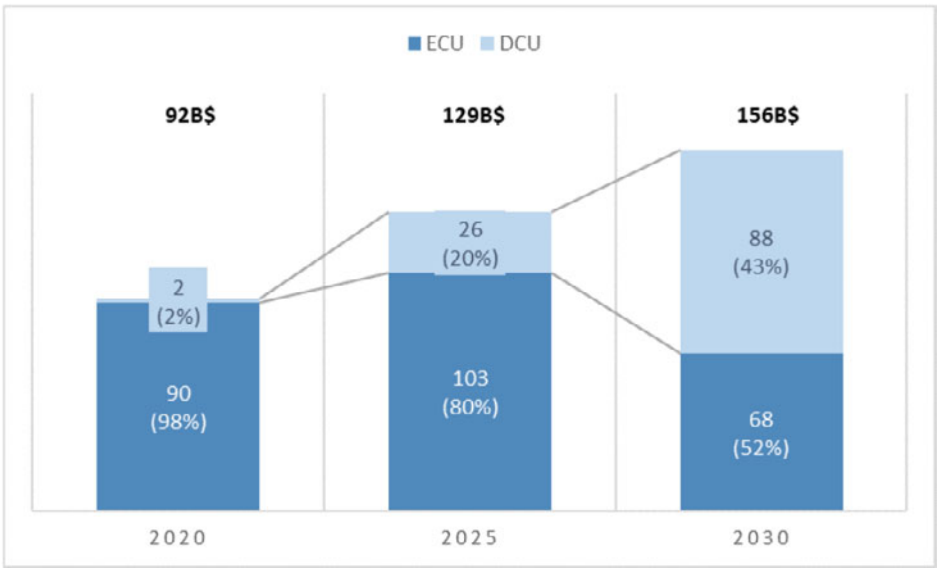
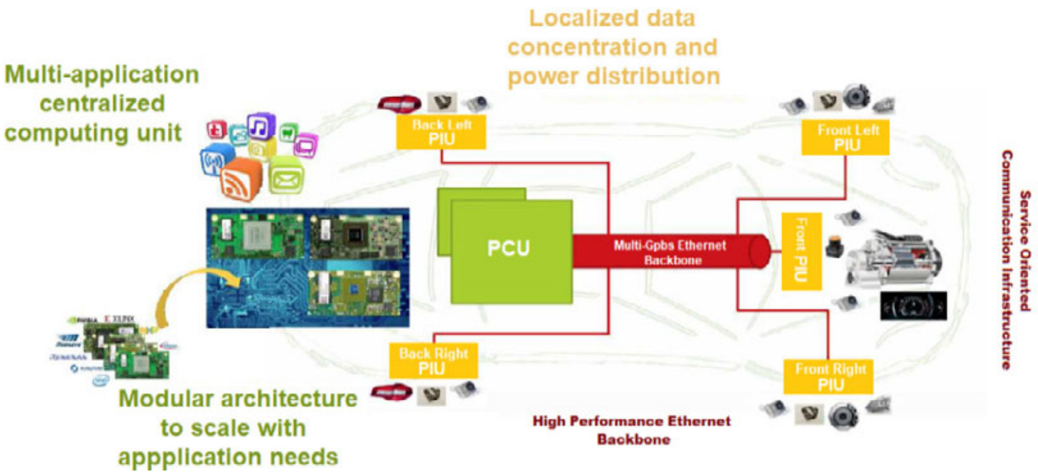
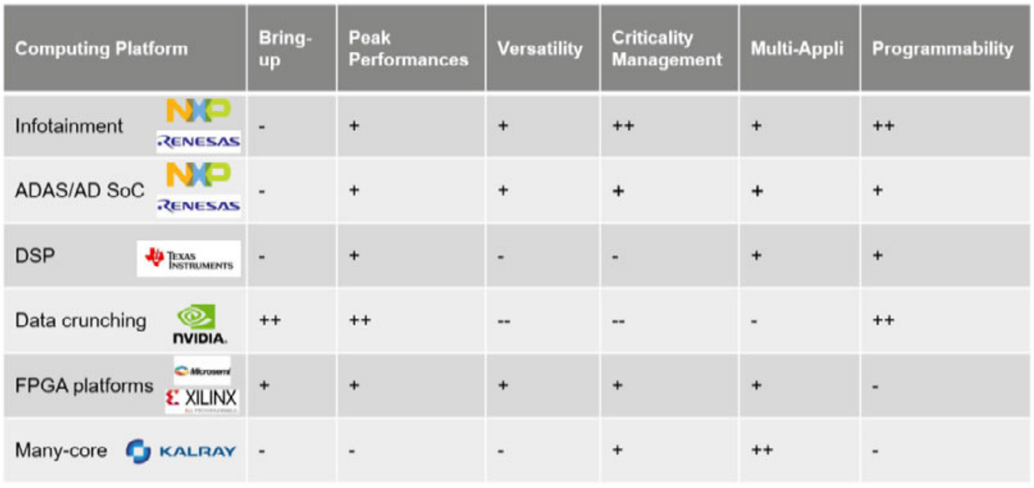


Figure 5.1. *Overall ECU/DCU costs (B$) evolution and breakdown between standard ECU and DCU from 2020 to 2030 (McKinsey and Company 2019)*



Multi-application centralized computing unit
Localized data concentration and power distribution
Back Left PIU
Front Left PIU
PCU
Multi-Gpbs Ethernet Backbone
Front PIU
Back Right PIU
Front Right PIU
Modular architecture to scale with appplication needs
High Performance Ethernet Backbone
Service Oriented Communication Infrastructure

Figure 5.2. *Overview of the FACE PCU and PIU infrastructure*



Computing Platform	Bring-up	Peak Performances	Versatility	Criticality Management	Multi-Appli	Programmability
Infotainment NXP RENESAS	-	+	+	++	+	++
ADAS/AD SoC NXP RENESAS	-	+	+	+	+	+
DSP Texas Instruments	-	+	-	-	+	+
Data crunching NVIDIA.	++	++	--	--	-	++
FPGA platforms Microsemi XILINX	+	+	+	+	+	-
Many-core KALRAY	-	-	-	+	++	-

Figure 5.3. *Synthetic result of the hardware benchmark*

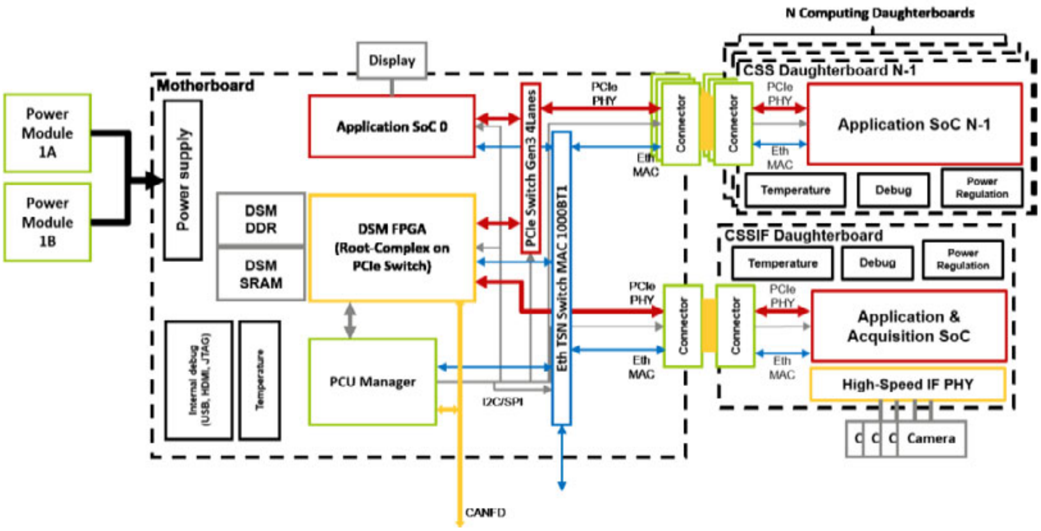


Figure 5.4. *Overview of the FACE PCU structure*

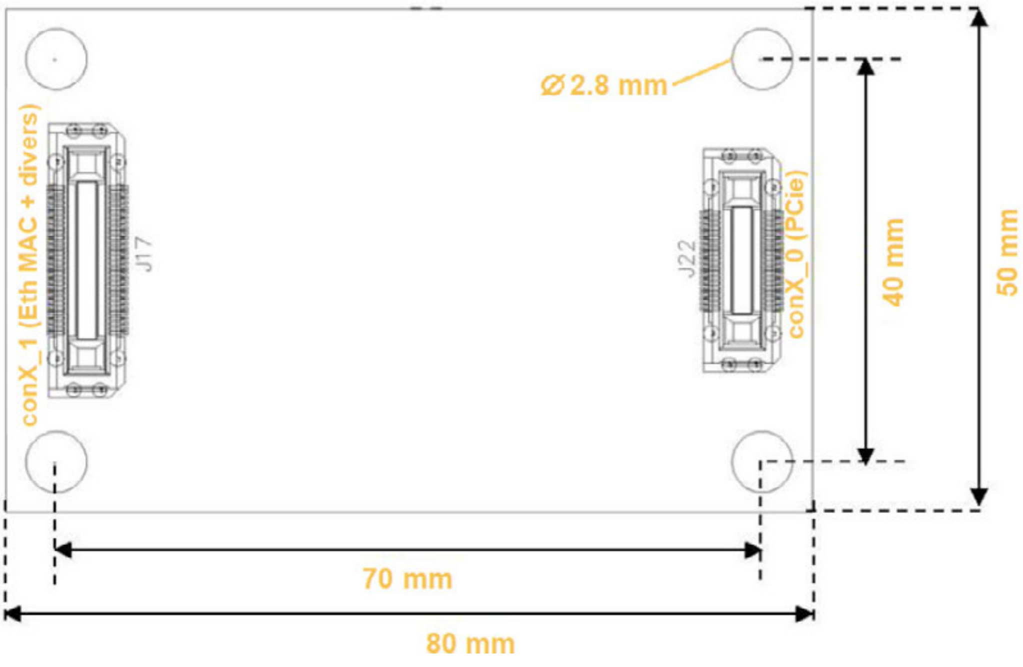


Figure 5.5. *Daughterboard physical form factor*

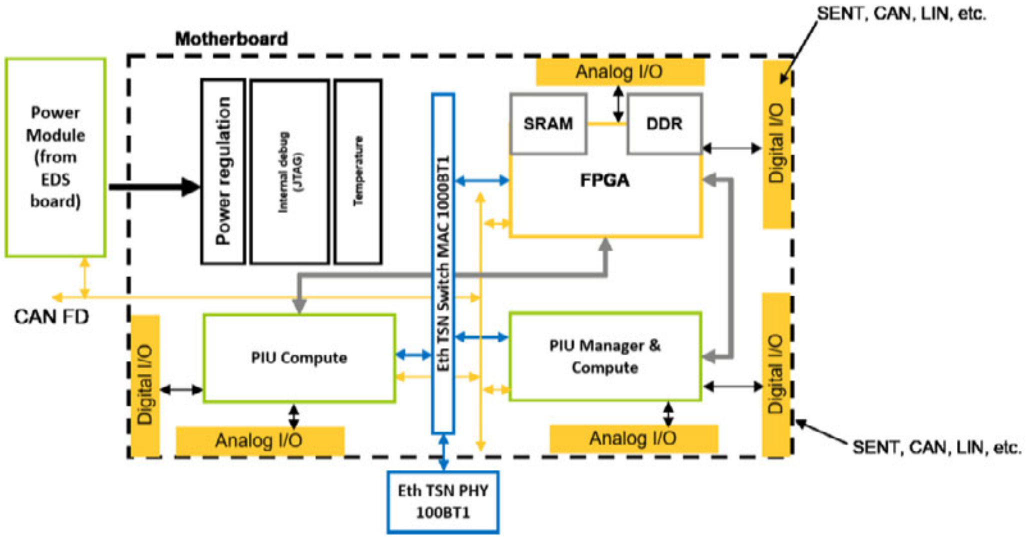


Figure 5.6. *Overview of the FACE PIU structure*

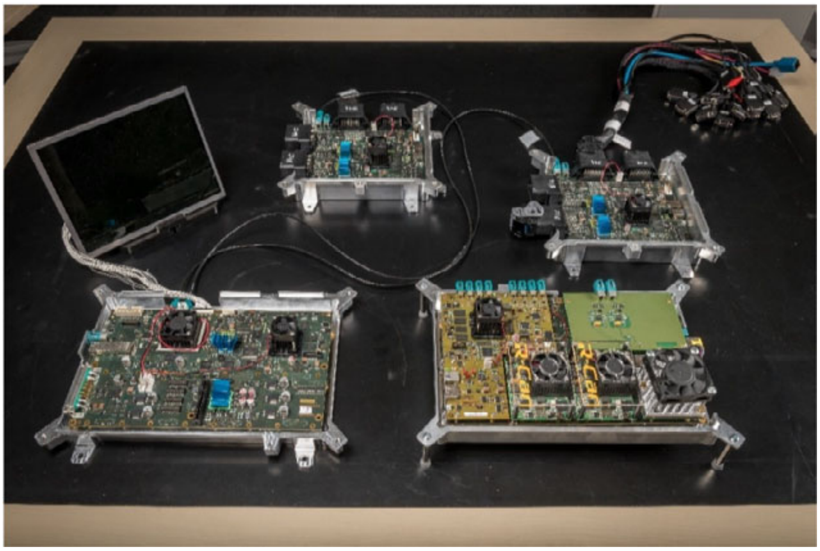


Figure 5.7. *Example of hardware setup of the FACE platform. PCU front and back sides are on the bottom of the photo. PCU is connected through 100Base-T1 Ethernet TSN to two PIUs. Harness to connect heterogeneous sensors and actuators is in the right-top corner of the photo*

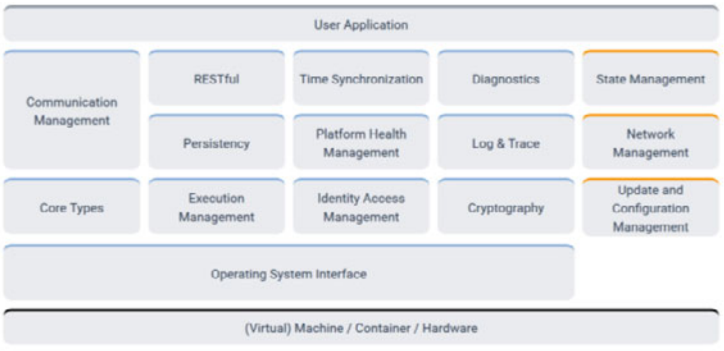


Figure 5.8. *Illustration of AUTOSAR adaptive platform software architecture*

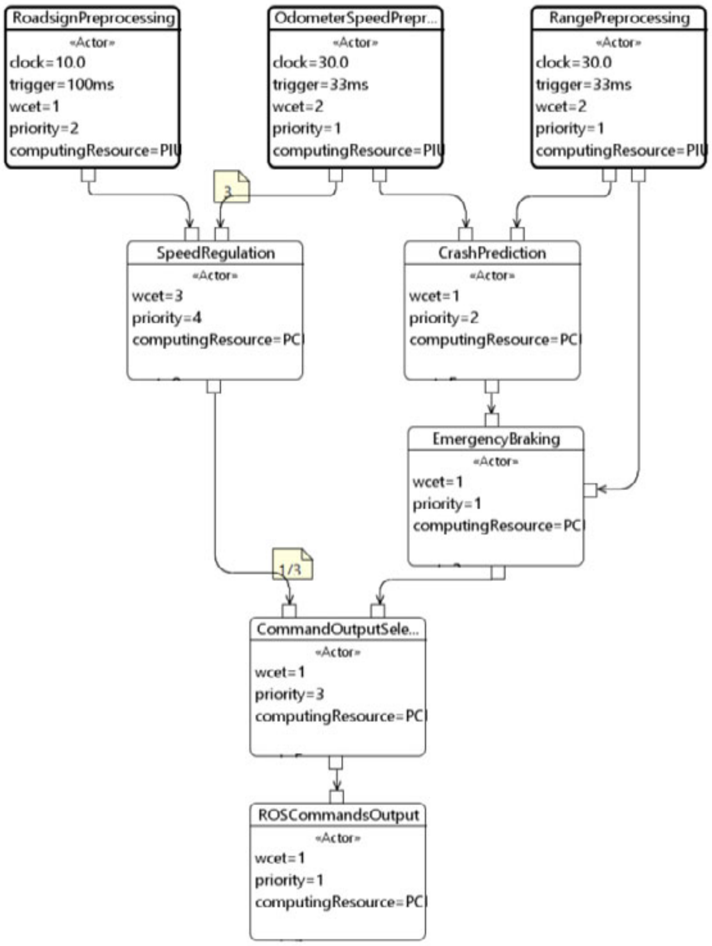


Figure 5.9. *ADAS polygraph model*

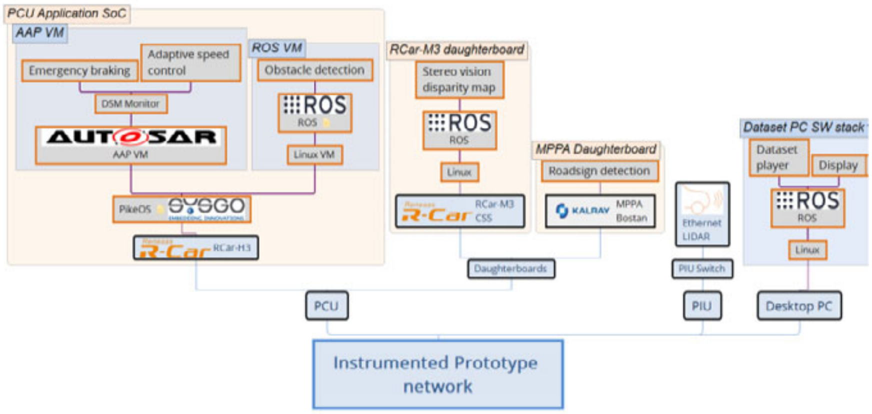


Figure 5.10. *The FACE instrumented prototype setup*

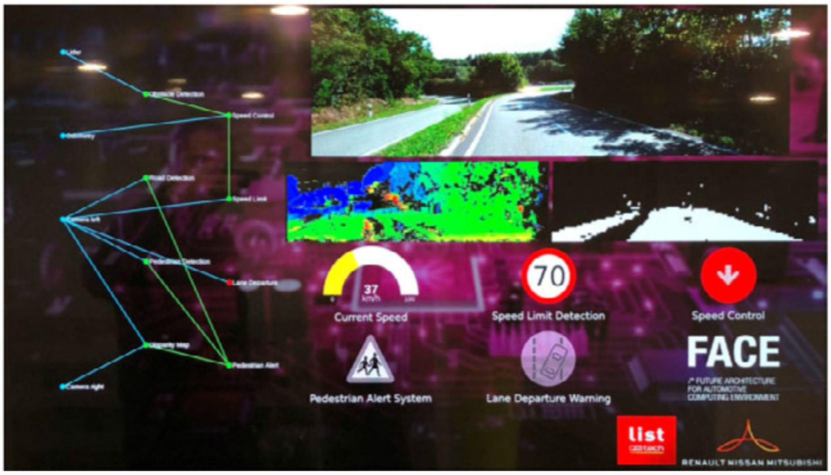


Figure 5.11. *Use case interface*

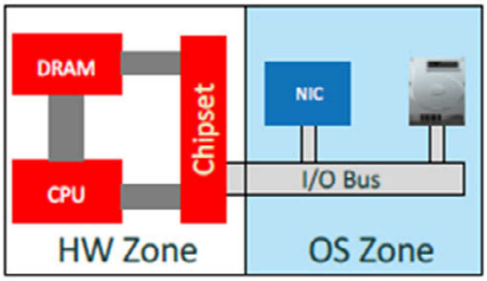


Figure 6.1. *The anatomy of a desktop in the 1980s*

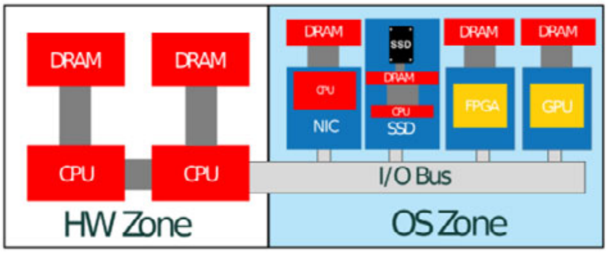


Figure 6.2. *The anatomy of a modern dual-socket server blade*

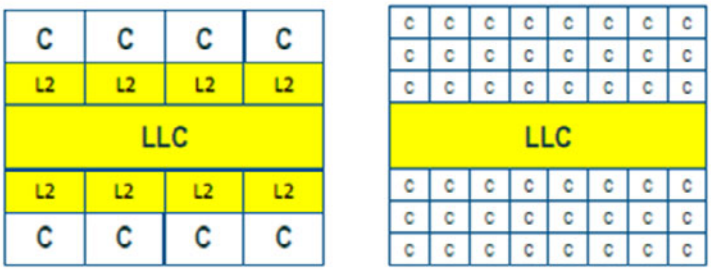


Figure 6.3. *Chip organization in an x86-based CPU (left) and a custom many-core CPU (right)*

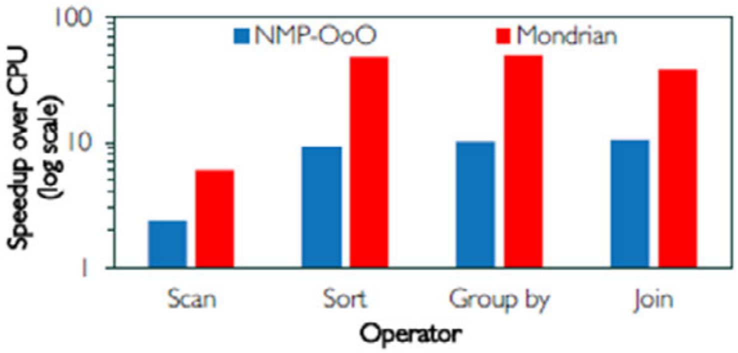


Figure 6.4. *Speedup of near-memory processing: many-core OoO CPU and Mondrian with streaming algorithms*

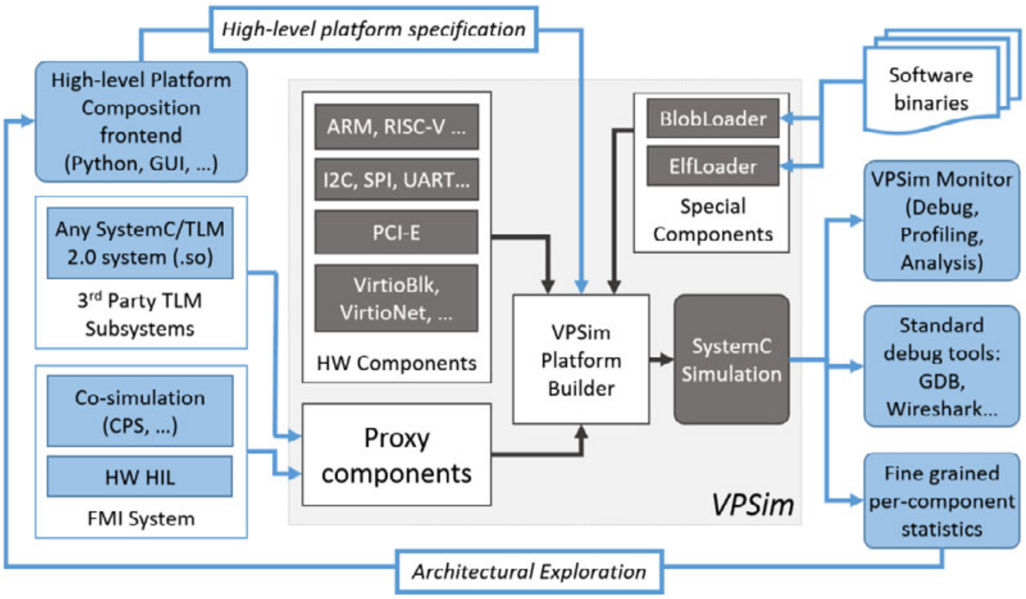

High-level platform specification
High-level Platform Composition frontend (Python, GUI, ...)
Any SystemC/TLM 2.0 system (.so)
3rd Party TLM Subsystems
Co-simulation (CPS, ...)
HW HIL
FMI System
ARM, RISC-V ...
I2C, SPI, UART...
PCI-E
VirtioBlk, VirtioNet, ...
HW Components
Proxy components
VPSim Platform Builder
SystemC Simulation
BlobLoader
ElfLoader
Special Components
VPSim
Software binaries
VPSim Monitor (Debug, Profiling, Analysis)
Standard debug tools: GDB, Wireshark...
Fine grained per-component statistics
Architectural Exploration

Figure 7.1. *Overview on the operation of VPSim*

```python
import pyvp
import sys
import tty
import select
import os

class PL011Uart:
    def __init__(self, **kwargs):
        self.base=kwargs["base"]
        self.irq=kwargs["irq"]
        # ... other initializations

        print("Initialized device pl011... (base=%s,irq=%s)" %
            (self.base,self.irq))

    def read(self, addr, size):
        off = addr - self.base
        self.update_status()
        if off == 0x00:
            return self.get_rx()
        elif off == 0x18:
            return self.get_status()
        elif off == 0x3C:
            return self.get_interrupt_unmasked()
        elif off == 0x40:
            return self.get_interrupt_masked()
        else:
            # other cases...

    def write(self, addr, value, size):
        off=addr-self.base
        if off == 0x00:
            self.backend_writechar(chr(value))
        elif off == 0x38:
            self.update_interrupt_enable(value)
        elif off == 0x44:
            self.updadte_interrupt_disable(value)
        else:
            # other cases...

        self.update_status()
        self.check_interrupt()

    def check_interrupt(self):
        if self.shouldInterrupt():
            pyvp.interrupt(self, self.irq, 1)
        else:
            pyvp.interrupt(self, self.irq, 0)

    def loop(self):
        while True:
            pyvp.wait(self, int(self.nsPerChar))
            self.update_status()
            self.check_interrupt()
```

Figure 7.2. *Simplistic implementation of the PL011 UART in Python*

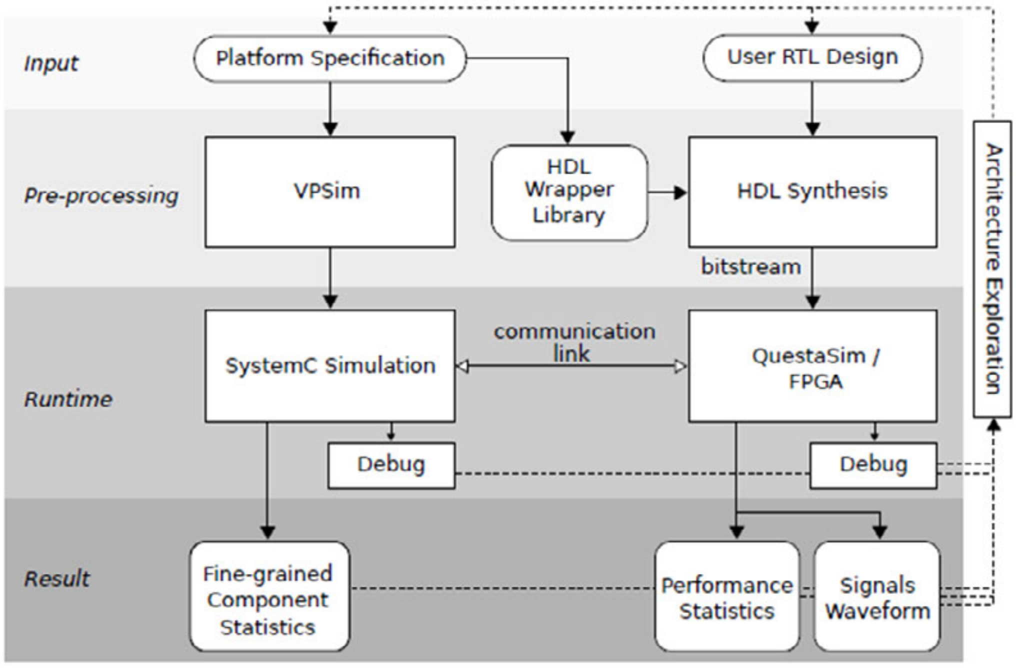


Figure 7.3. *System validation flow using the hybrid prototyping solution*

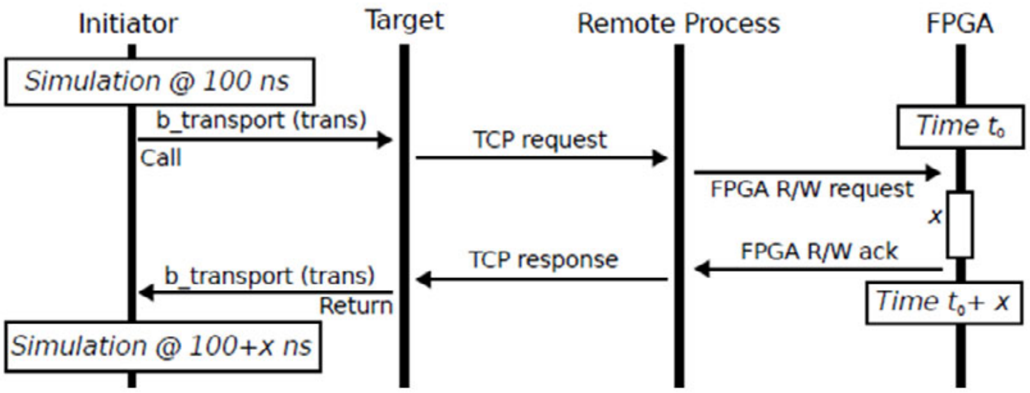


Figure 7.4. *Synchronization mechanism between VPSim and FPGA in TLM R/W*

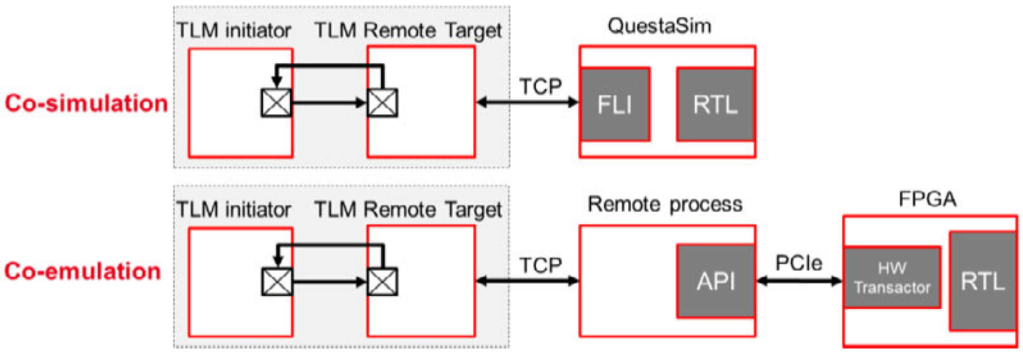


Figure 7.5. *Communication scheme in VPSim during co-simulation and co-emulation*

```
'can': [
 {
  'name': 'RenesasRcarH3CAN',
  'base': 0xe6c30000,
  'size': 0x1000,
  'irq': 186,

  'fmi_values': [
   {
    'name': 'speed',
    'can_id': 0x10,
    'causality': 'input',
   },
   {
    'name': 'accel',
    'can_id': 0x11,
    'causality': 'output',
   },
   {
    'name': 'brake',
    'can_id': 0x12,
    'causality': 'output',
   },
  ],
 },
],
```

Figure 7.6. *Example of FmiValue declaration*

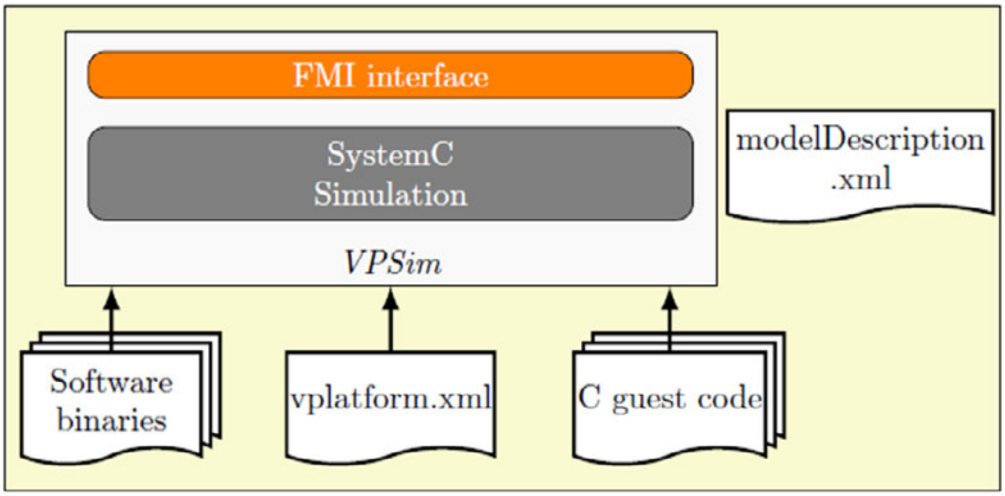


Figure 7.7. *Structure of the generated virtual platform FMU*

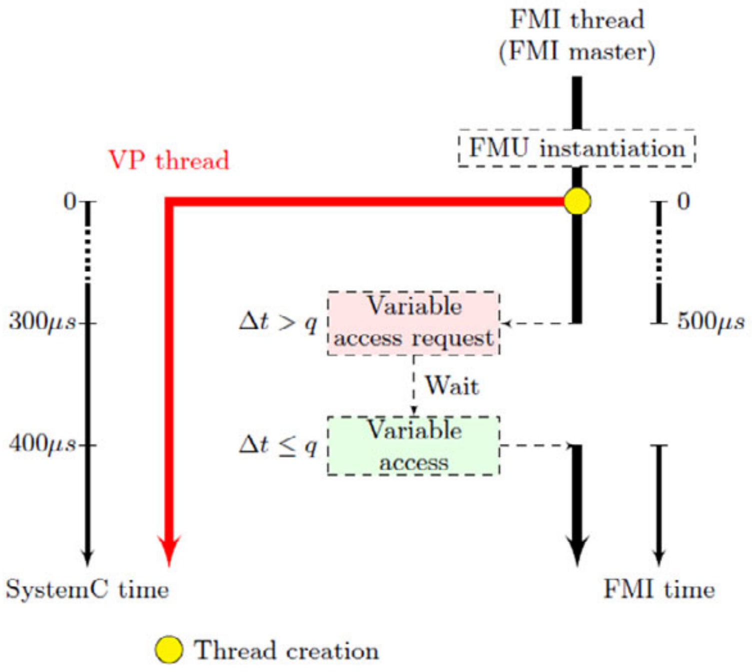


Figure 7.8. *Parallel implementation of the virtual platform FMU*

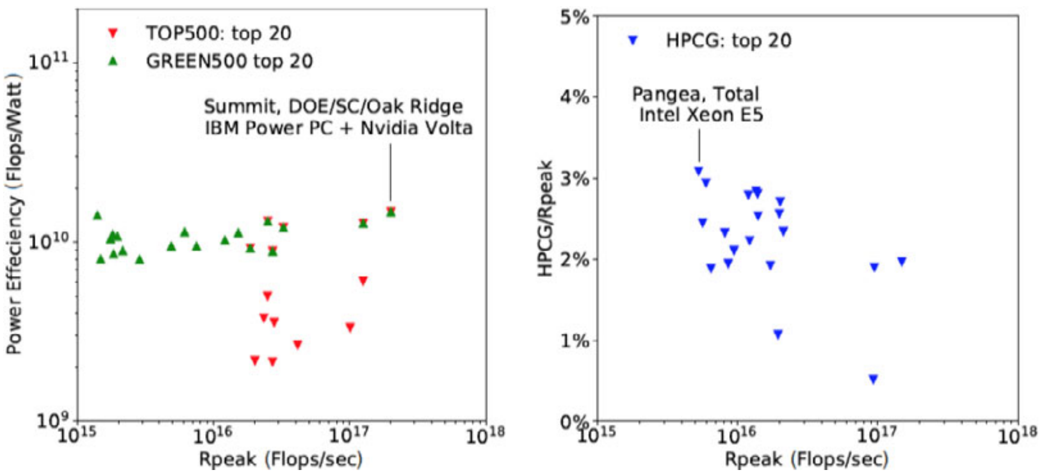


Figure 8.1. *The top 20 of the TOP500 and GREEN500 supercomputers (left) and the top-20 of the HPCG supercomputers (right)*

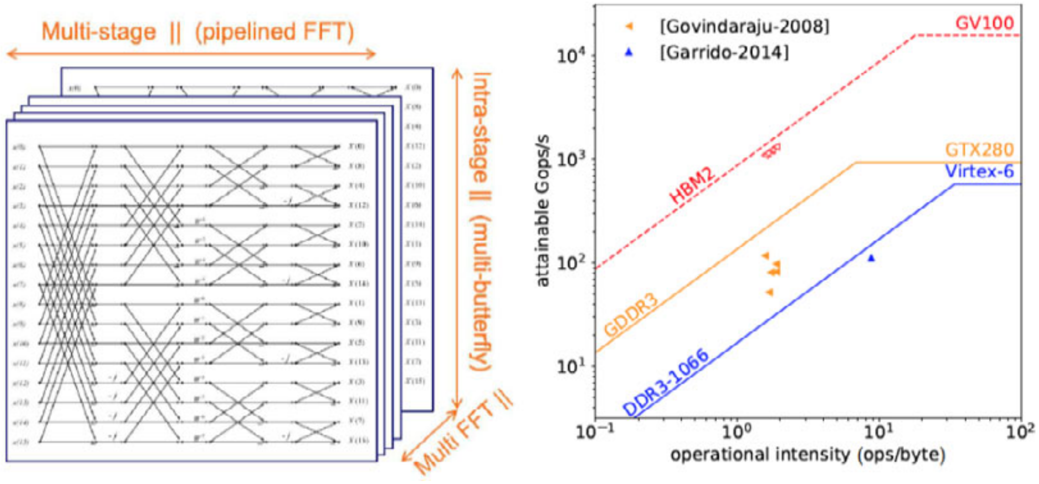


Figure 8.2. *The three axes of FFT parallelism (left) and various machine rooflines with FFT performances*

```
G   = Graph()
G.n = [Node() for i in range(7)]
G.e = Path (G.n)
G.f = Edge (G.n[1], G.n[5])
G.e[1].init(D=3)                # used only for the second simulation
```

Figure 8.3. *Homogeneous flow graph G comprising seven nodes and seven edges*

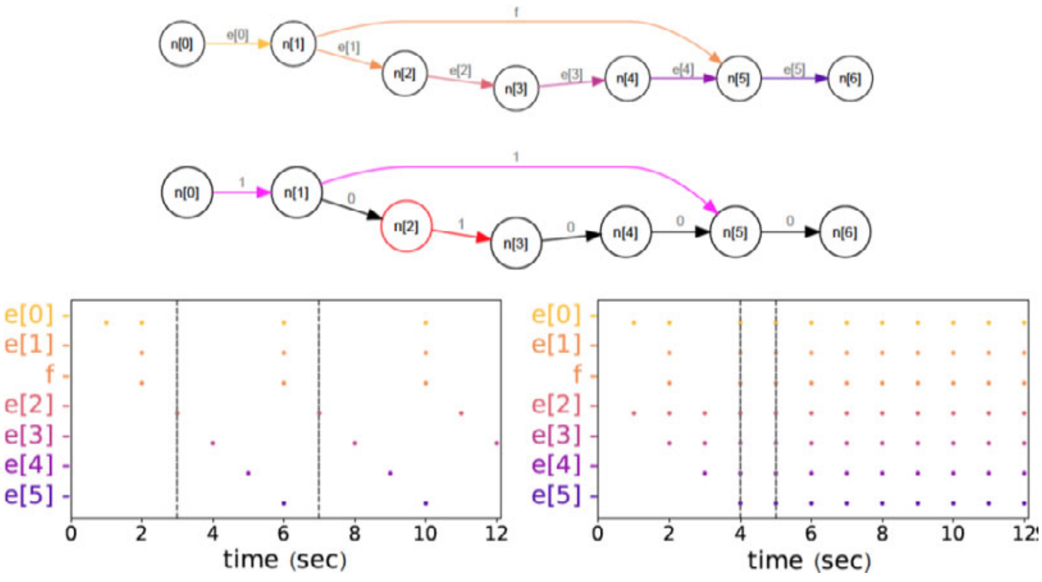


Figure 8.4. *Flow graph G (top), its state after three cycles (mid), its flow (event timing) during 12 cycles (left) and its flow with three initial tokens on edge* e[1] *(right)*

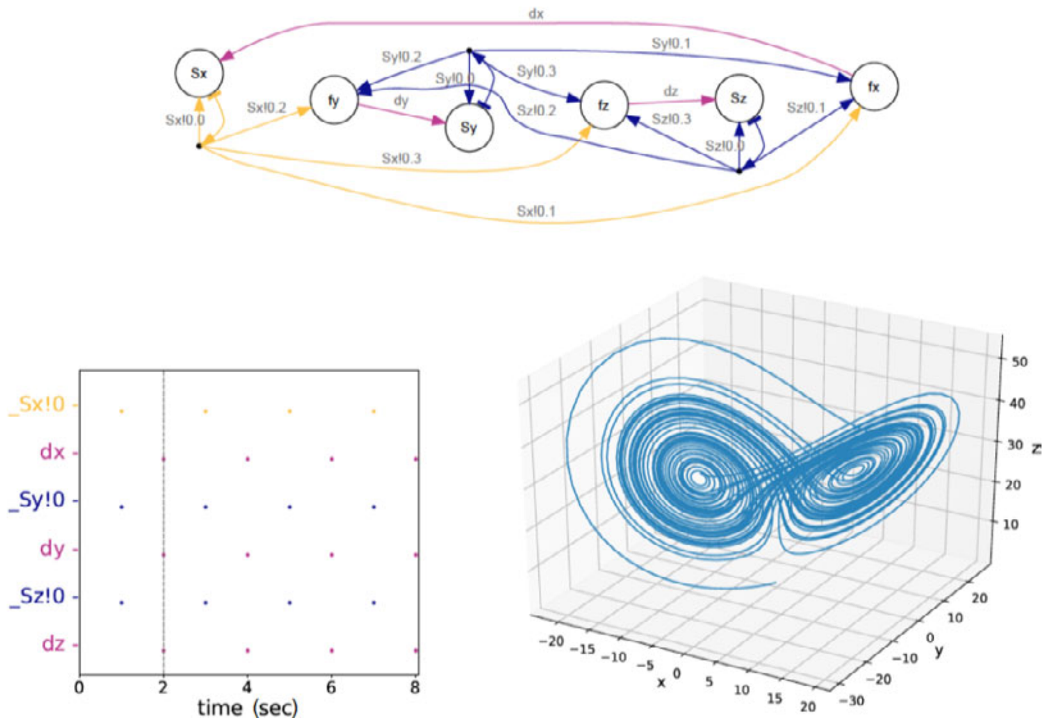


Figure 8.5. *Graph Glorenz (top), its flow during the first eight cycles (left) and the* x, y, z *data produced by nodes* fx, fy, fz, *respectively (right)*

```
sigma,rho,beta, dt= 10, 28, 8/3. 0.01
fo              = [lambda x: sigma*(x[1] - x[0]),
                   lambda x: (rho-x[2])*x[0] - x[1],
                   lambda x: x[0]*x[1] - beta*x[2]]

Glorenz         = Graph(); G=Glorenz
G.Sx,G.Sy,G.Sz= [Node(I=0, fo= lambda x: x[0]+dt*x[1])
                   for i in range(3)]
G.fx,G.fy,G.fz= [Node(I=[G.Sx.O[0],G.Sy.O[0],G.Sz.O[0]],fo=fo[i])
                   for i in range(3)]
G.dx,G.dy,G.dz= [Edge(G.fx,G.Sx),Edge(G.fy,G.Sy),Edge(G.fz,G.Sz)]

for i,n in enumerate([G.Sx, G.Sy, G.Sz]):
    n.I[0].init(D=1, S=0, x=[-0.15, -0.2, 0.2][i])
for e in (G.dx, G.dy, G.dz): e.init(D=1, S=0, x=0)
```

Figure 8.6. *Definition graph* Glorenz

```
N      = 11
GMA    = Graph(rate=50e3);   G= GMA
G.mp3  = MP3(mp3='../../../audio/war.mp3', tmin=37.5, rate=44100)
fo     = lambda x: np.int16((x[0][N]-x[0][0])/N +x[1])
G.ma   = Node(I=[G.mp3,0], fo=fo)
G.ma.I[0].init(x=[0 for i in range(N)])
G.ma.I[1].init(D=1, x=[0])          # accumulator, initialized to 0
G.snk  = Node(I=G.ma.O[0])
```

Figure 8.7. *Definition of graph* GMA

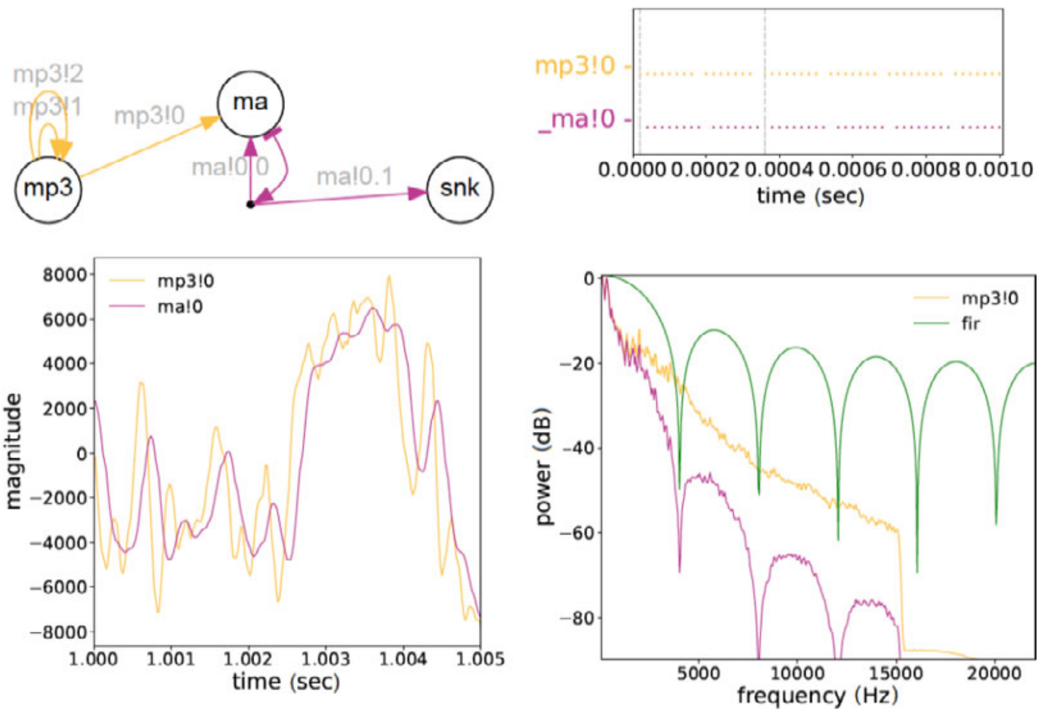


Figure 8.8. *Node* ma *computes the moving average over a window N = 11 samples. The flow shows incidental gaps due to flow throttling by the* mp3 *source. The two bottom plots show the signal smoothening in time and frequency domain*

```
set_fsm([Repeat(L, iter=Rule(I=[1,0,1], O=[0,1,1], s=0), exit=1),
         Fire(Rule(I=[1,1,1], O=[1,1,1], s=1))])
```

Figure 8.9. *An example of an FSM for a cyclo-static dataflow node*

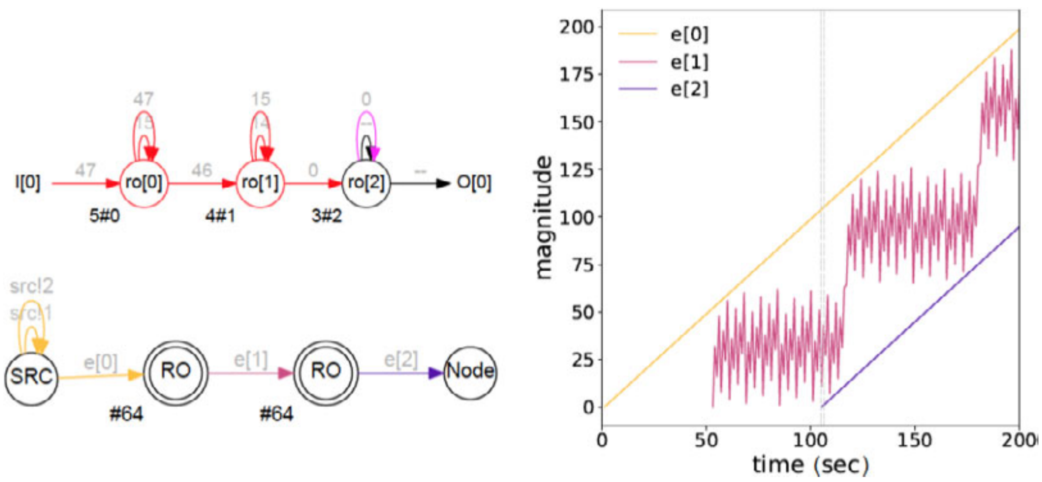


Figure 8.10. *Subgraph RO (left, top) comprises three CSDF nodes* ro[i] *and performs an N = 64 FFT bit-reversal. As FFT bit-reversal is its own inverse, a cascade of two* ROs *performs the identity, as shown for a linear input sequence (right)*

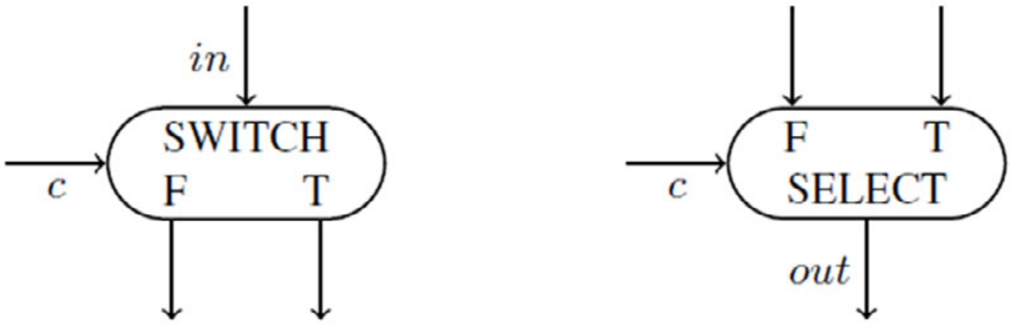


Figure 8.11. *SWITCH and SELECT nodes as used in Boolean Dataflow*

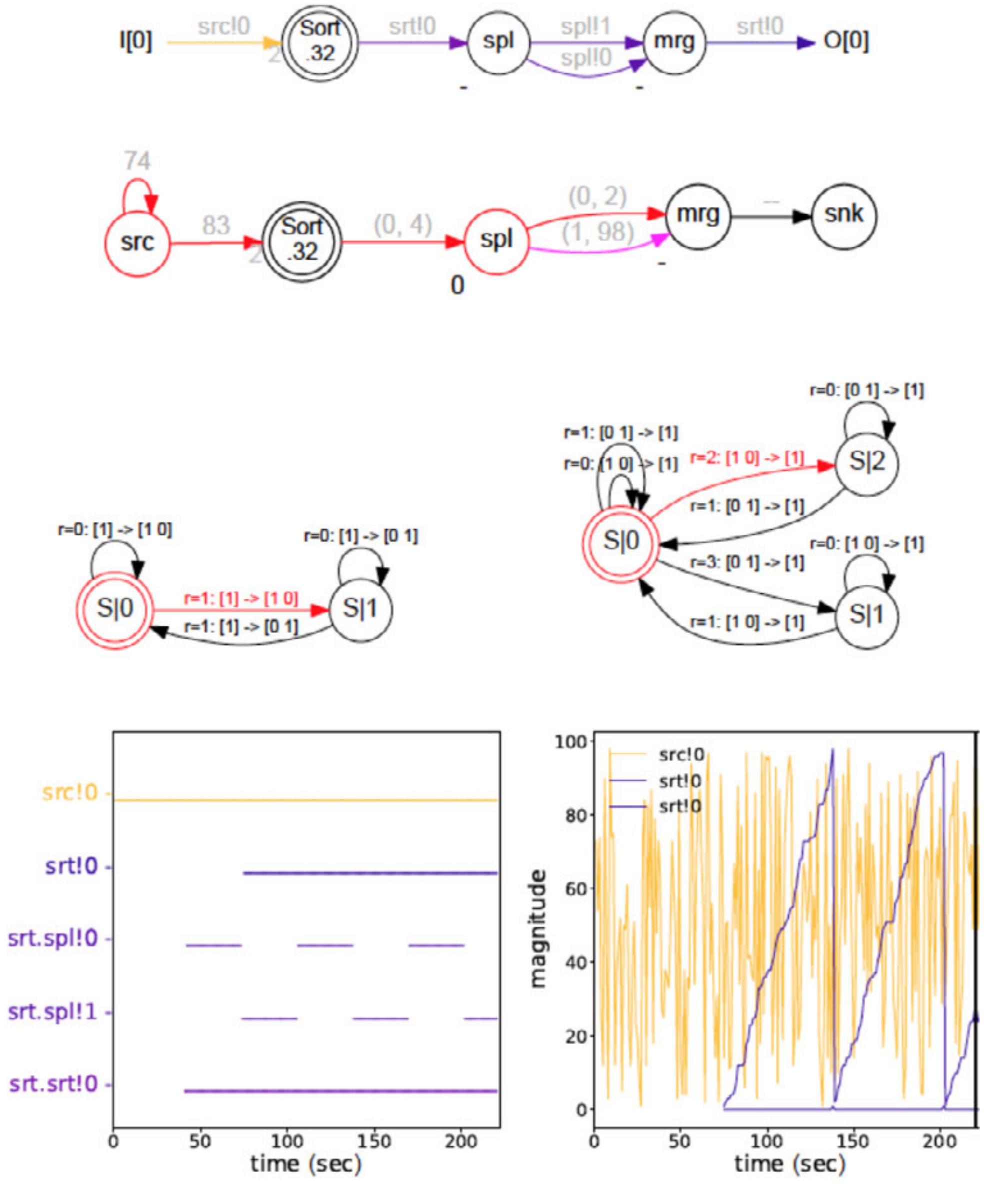


Figure 8.12. *Recursive dataflow graph* Sort.64 *(top),* Sort.64 *connected to a random-number source* src *and* snk *(one-from-top), the FSMs of nodes* spl *and* mrg *(mid-left and right) and the flow and data plots for* Sort.64 *(bottom)*

```
class Split(Node):
    def __init__(self, N, I=[]):
        super(Split, self).__init__(I=I)
        pass0 = lambda s: Rule(I=(1,), O=(1,0), s=s)
        pass1 = lambda s: Rule(I=(1,), O=(0,1), s=s)
        self.set_fsm([Select([pass0(0), pass0(1)]),
                      Select([pass1(1), pass1(0)])])
        self.set_fs([lambda x: x[0], lambda x: x[0]])
        self.set_fo([lambda x: x,    lambda x: x])
```

Figure 8.13. *Definition of node class* Split

```
class IMG(Node):
    def __init__(self, N I=[]):
        super(IMG, self).__init__(I=I)
        rules= [Rule(I=[int(j==i) for j in range(N)],O=[1,],s=0)
                for i in range(N)]
        self.set_fsm([Choice(rules)])
        self.set_fo([lambda x, r: x[r]])
```

Figure 8.14. *Definition of node class* IMG

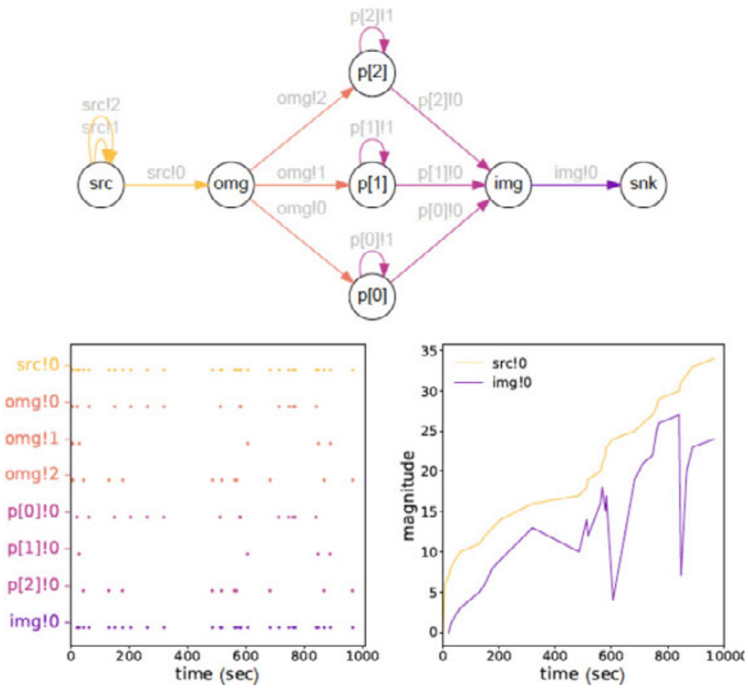


Figure 8.15. *A toy processor farm: dataflow graph (top), flow plot (left) and data plot (right). Node src produces the natural numbers, in order*

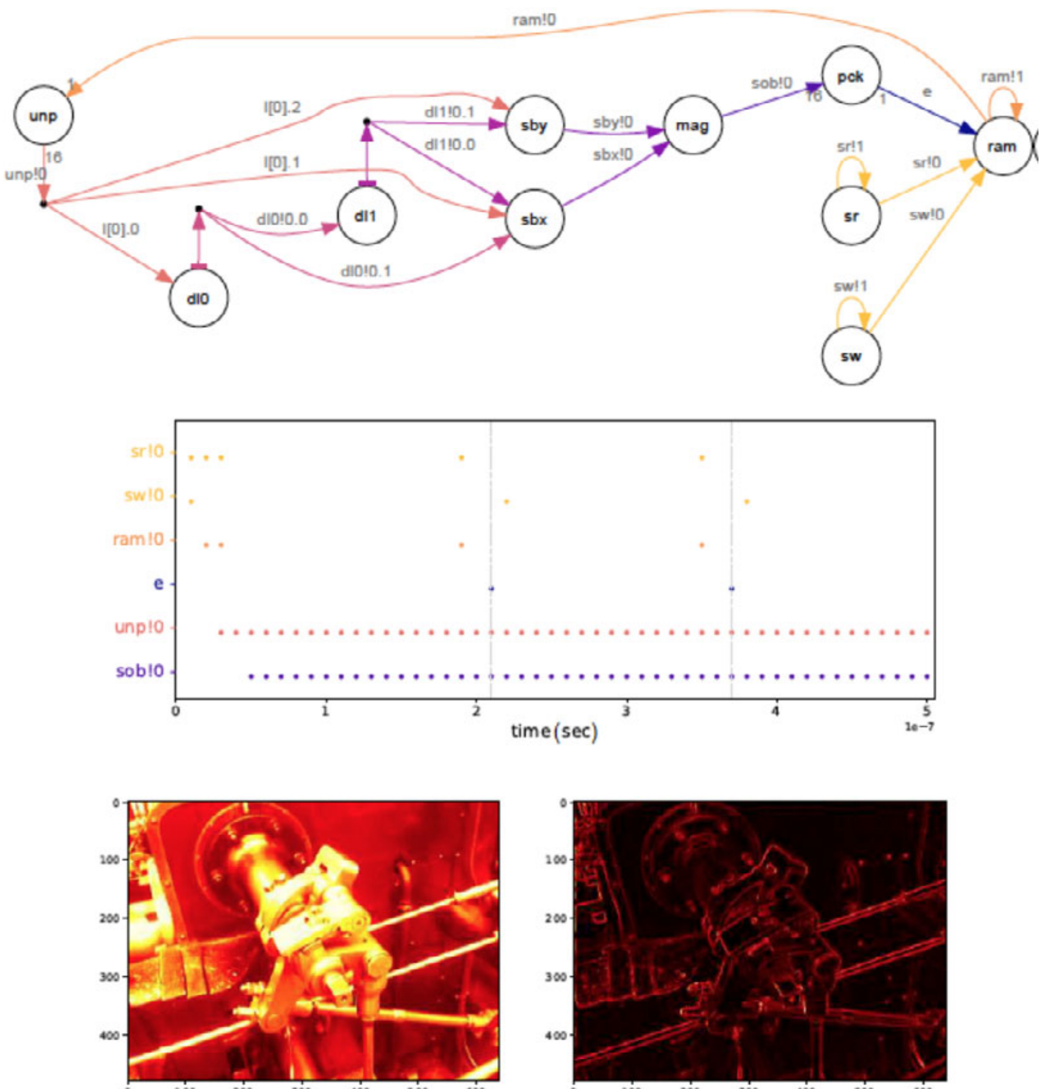


Figure 8.16. *Dataflow graph* Sobel *with I/O to* RAM *node* ram *(top), 50μsec flow plot (middle) and example input+output images (bottom)*

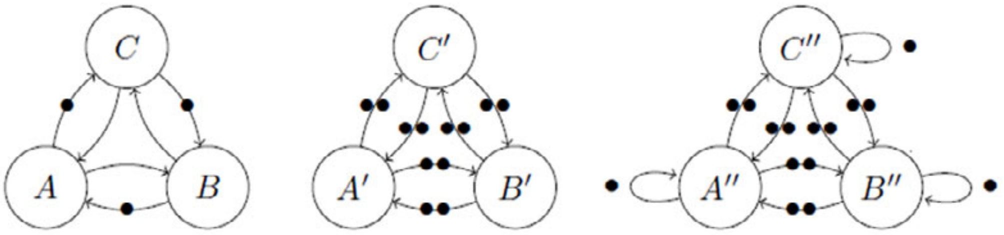


Figure 8.17. *Three tricycle dataflow graphs. Node execution times equal 1*

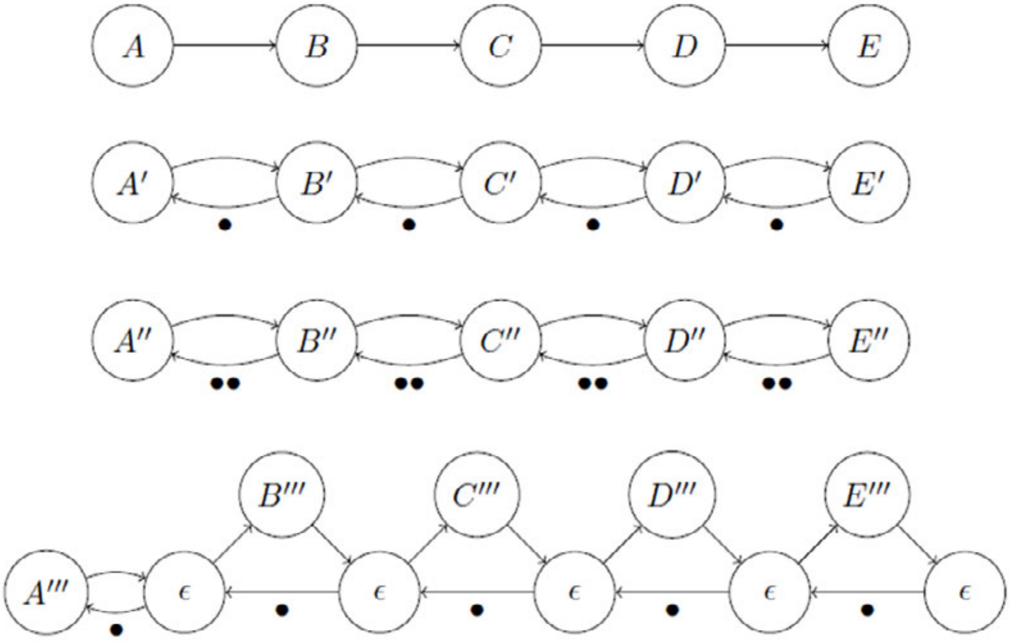


Figure 8.18. *A dataflow pipeline with four different back-pressure schemes. Nodes A to E have an execution time of 1 and self-loops to bound the node's throughput to 1 (not drawn). ϵ-nodes have an execution time of $\epsilon \to 0$*

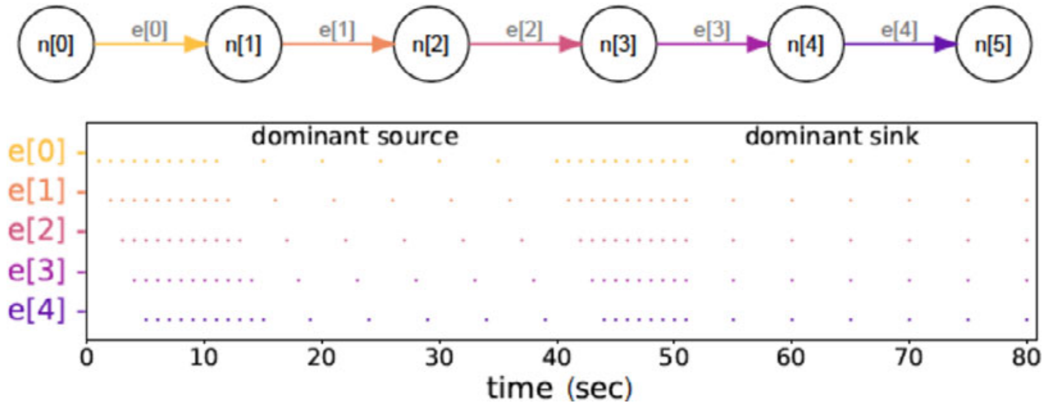


Figure 8.19. *A six-node pipeline starts with all nodes running self-timed with $t_n = 1$. After 11 firings, source node n[0] is throttled to $t_{n[o]} = 5$. At time=40, the source throttle is switched off, and at time=50, the sink node is throttled to $t_{n[5]} = 5$*

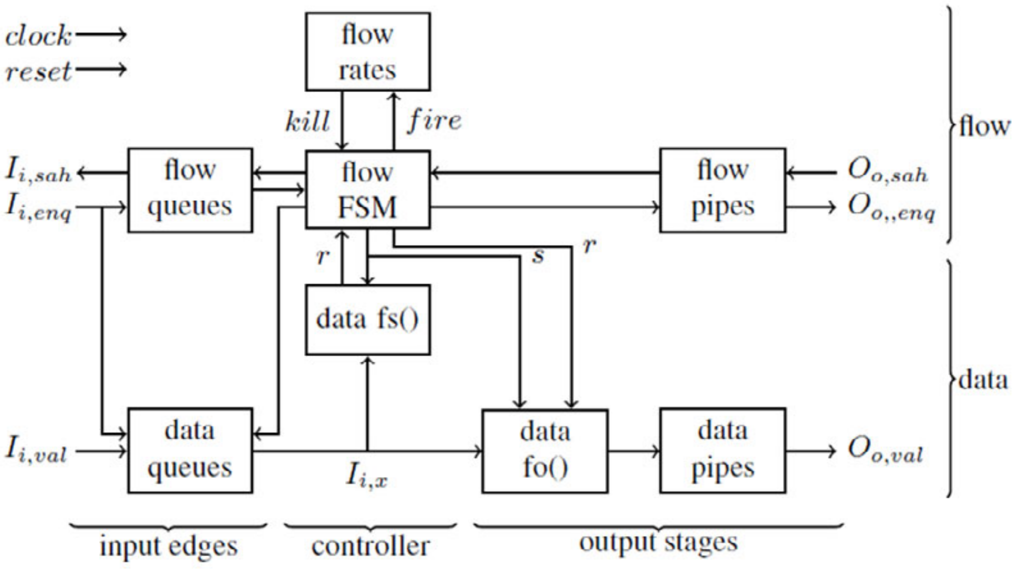


Figure 8.20. *Verilog template for a StaccatoLab node, including input edges*

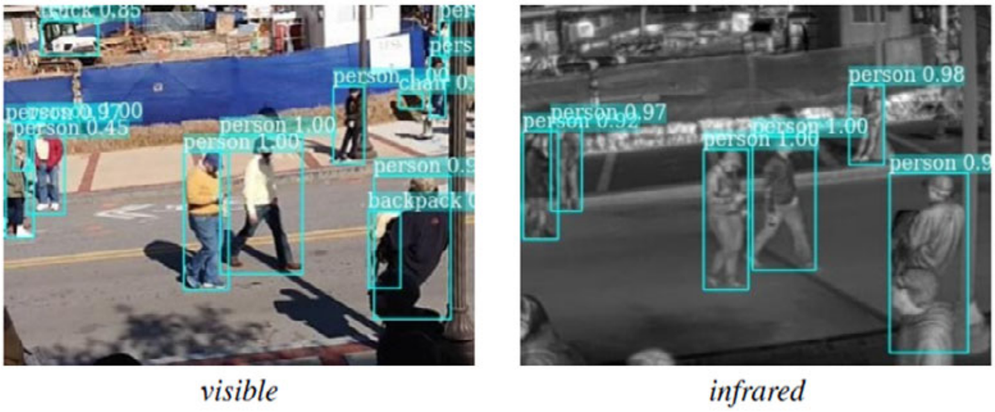


Figure 9.1. *Visible and infrared images and associated classification from the CAMEL dataset (https://camel.ece.gatech.edu)*

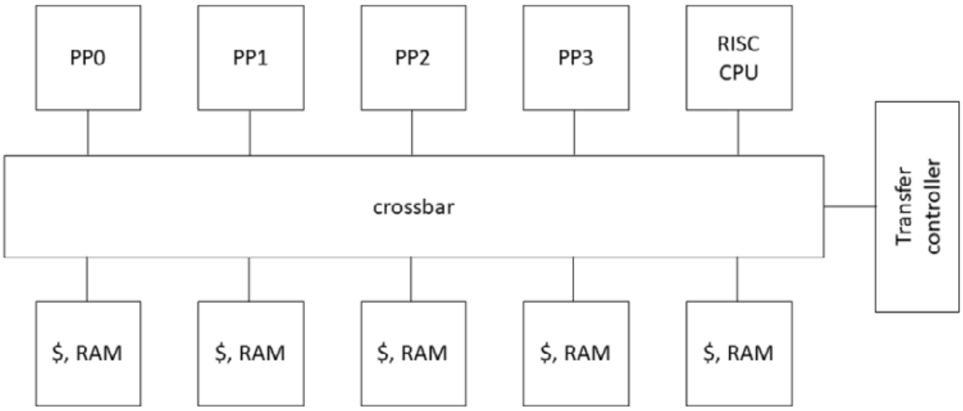


Figure 9.2. *The TI MVP*

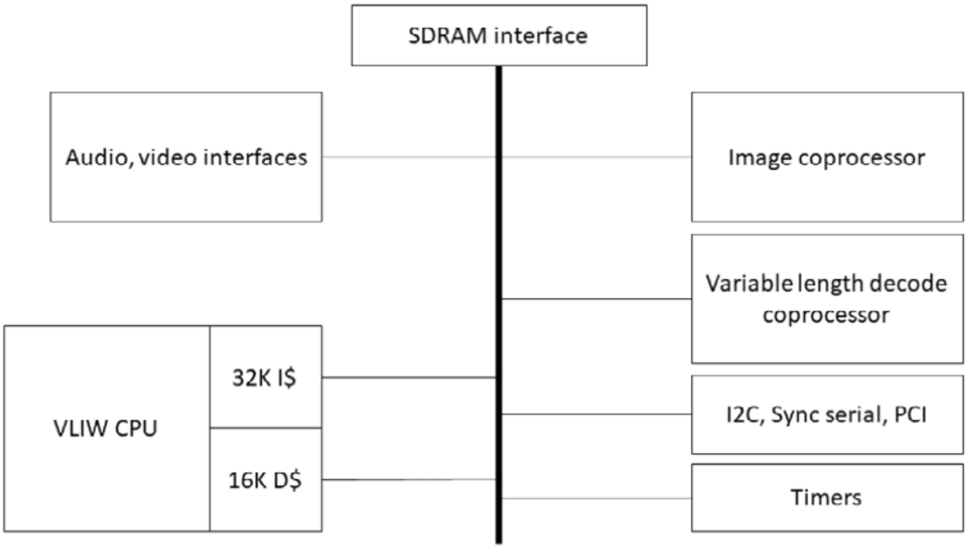


Figure 9.3. *The Trimedia TM-1*

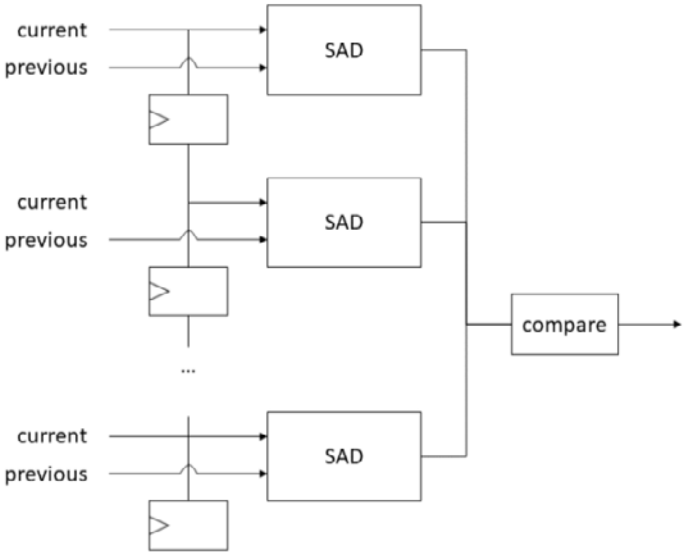


Figure 9.4. *Motion compensation accelerator*

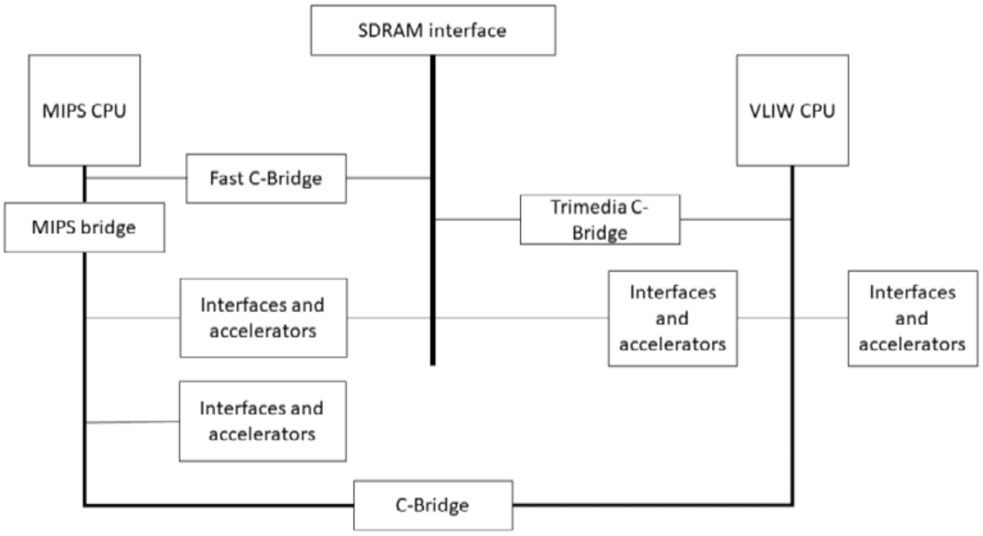

SDRAM interface
MIPS CPU
VLIW CPU
Fast C-Bridge
Trimedia C-Bridge
MIPS bridge
Interfaces and accelerators
Interfaces and accelerators
Interfaces and accelerators
Interfaces and accelerators
C-Bridge

Figure 9.5. *The Philips Viper*

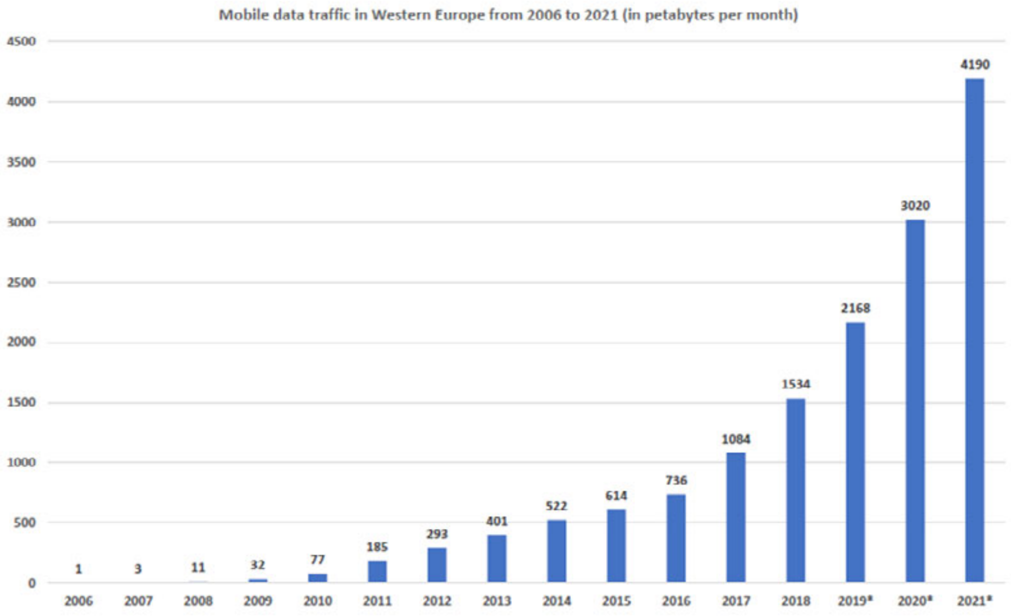


Figure 10.1. *Mobile data traffic in western Europe from 2006 to 2021 (2019-2021 are projections), adapted from Page et al. (2010) and Cisco Systems (2017)*

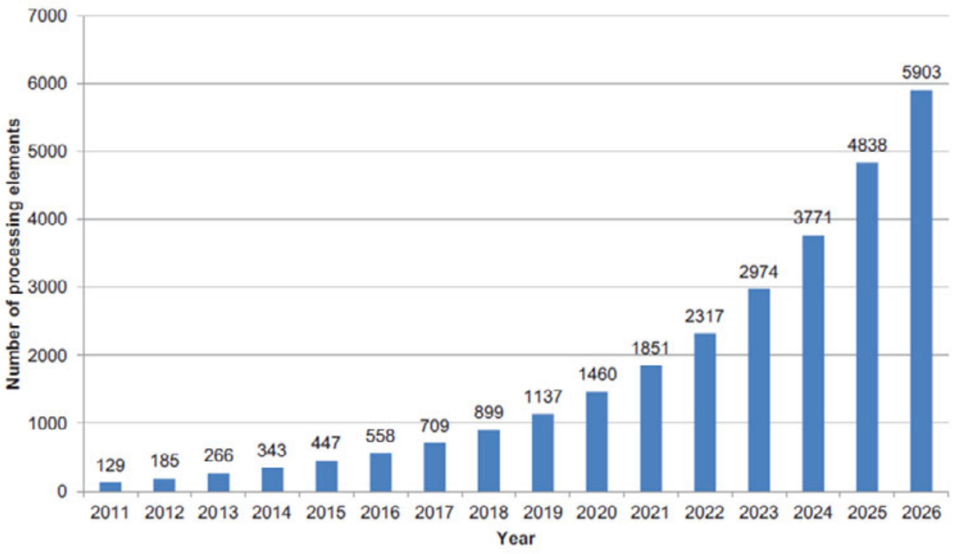


Figure 10.2. *SOC consumer portable design trends (adapted from International Technology Roadmap for Semiconductors (2011)). The processing elements are referred to cores or parallel execution units in CPUs, GPUs, accelerators, etc.*

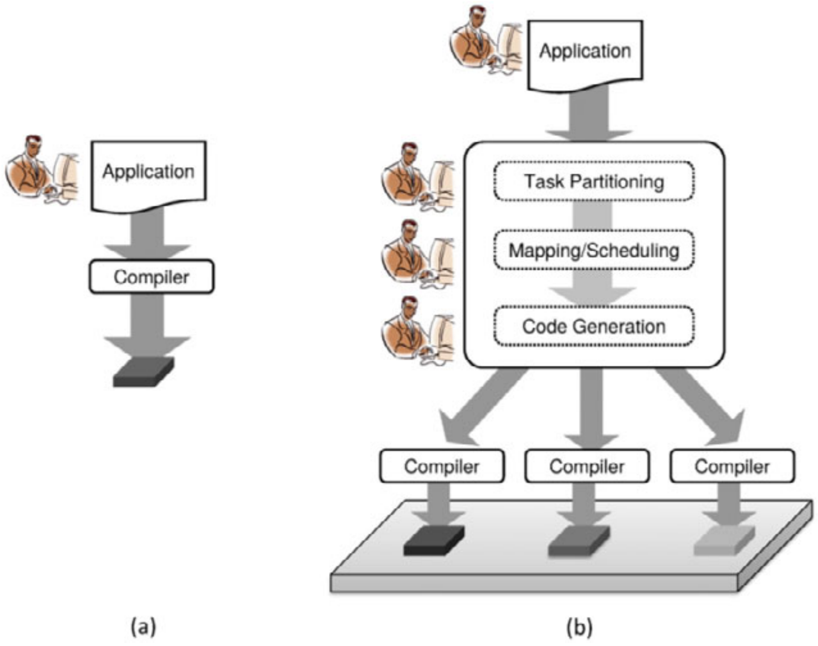


Figure 10.3. *Programming flow. (a) Uni-processor: compilers perform an end-to-end translation from parsing source code, target independent optimization and targetdependent optimization to final binary generation. (b) MPSoC: no compilation framework for MPSoCs is in place. The process (task partitioning,mapping/scheduling and code generation) is manual*

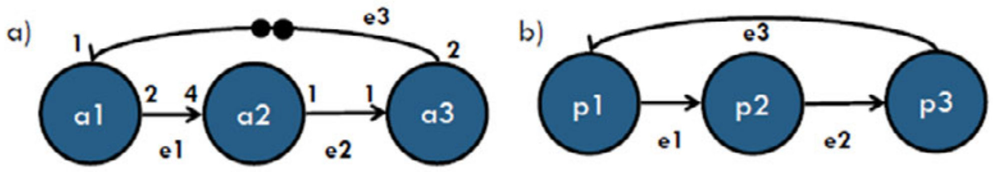


Figure 10.4. *a) Example of an SDF graph. Actors have fixed consumption and production rates. Initial tokens are represented as dots on the channels. b) Example of a process network. The abstract specification expresses that nodes can communicate if there is a channel that connects them, but does not specify how this communication takes place*

```
typedef struct { int i; double d; } my_struct_t;
typedef union { float f; short s[4]; } my_union_t;

__PNchannel char B[3][3];
__PNchannel my_struct_t C;
__PNchannel my_union_t D;
__PNchannel int A = {1, 2, 3}; /* Initial channel tokens */
```

Figure 10.5. *CPN example code: channel declaration*

```
/* Run Length Decoding, e.g. 4A2B5C3D -> AAAABBCCCCCDDD */
__PNkpn RLD __PNin(int EncIn) __PNout(int DecOut)
{
    int count, i;
    while (1) {
      __PNin(EncIn) /* read a token (# of appearances) from EncIn,
                       e.g. 4 */
         count = EncIn;
      __PNin(EncIn) /* read a token (data itself) from EncIn,
                       e.g. A */
         for (i = 0; i < count; ++i) /* write data to DecOut,
                                        e.g 4 times of A */
            __PNout(DecOut)
               DecOut = EncIn;
    }
}
```

Figure 10.6. *CPN example code: KPN process template (run-length decoding)*

```
__PNsdf Add __PNin(int a, int b) __PNout(int sum) {
  /* initialization code could be placed here */
  __PNloop { /* infinite loop */
    /* a and b are read from the channel implicitly */
    sum = a + b; /* channel variables are accessible in C code */
    /* sum is written to the channel implicitly */
  }
}
```

Figure 10.7. *CPN example code: SDF process template (adder)*

```
 __PNchannel int decoder1_input = {4,'A',2,'B',5,'C',3,'D'};
 __PNchannel int decoder2_input = {3,'E',5,'F',4,'G',2,'H'};

 __PNchannel int decoder1_output, decoder2_output;
 __PNchannel int add_output;

 __PNprocess decoder1 = RLD __PNin(decoder1_input)
    __PNout(decoder1_output);
 __PNprocess decoder2 = RLD __PNin(decoder2_input)
    __PNout(decoder2_output);
 __PNprocess add = Add __PNin(decoder1_output, decoder2_output)
    __PNout(add_output);
```

Figure 10.8. *CPN example code: process instantiation*

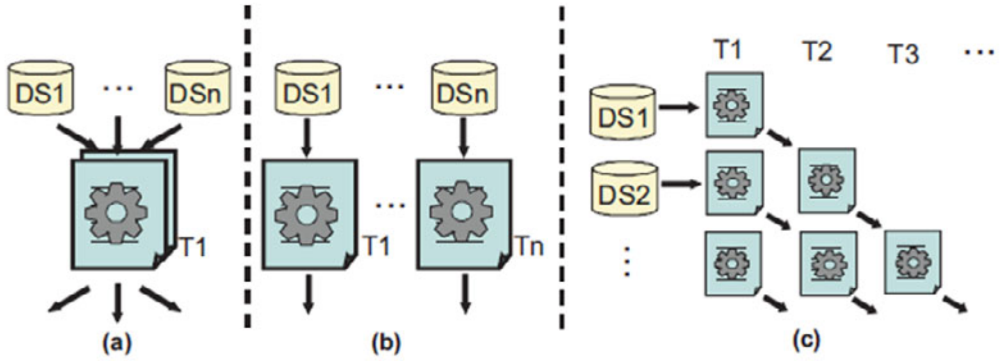


Figure 10.9. *Types of parallelism: (a) DLP, (b) TLP and (c) PLP (DS stands for data set and T stands for task)*

```
__PNchannel int DS1, DS2, DSN;
__PNchannel int Result1, Result2, ResultN;
__PNprocess T_1 = SameTaskT1 __PNin(DS1) __PNout(Result1);
__PNprocess T_2 = SameTaskT1 __PNin(DS2) __PNout(Result2);
__PNprocess T_N = SameTaskT1 __PNin(DSN) __PNout(ResultN);
```

Figure 10.10. *CPN example code: DLP*

```
__PNchannel int DS1, DS2, DSN;
__PNchannel int Result1, Result2, ResultN;
__PNprocess T_1 = Task1 __PNin(DS1) __PNout(Result1);
__PNprocess T_2 = Task2 __PNin(DS2) __PNout(Result2);
__PNprocess T_N = TaskN __PNin(DSN) __PNout(ResultN);
```

Figure 10.11. *CPN example code: TLP*

```
__PNchannel pixels DS, intermediate1, intermediate2,
   intermediate3, output;
__PNprocess source = ReadImage __PNout(DS);
__PNprocess T_1 = PipelineTask1 __PNin(DS)
   __PNout(intermediate1);
__PNprocess T_2 = PipelineTask2 __PNin(intermediate1)
   __PNout(intermediate2);
__PNprocess T_3 = PipelineTask3 __PNin(intermediate2)
   __PNout(intermediate3);
__PNprocess T_N = PipelineTaskN __PNin(intermediate3)
   __PNout(output);
__PNprocess sink = WriteImage __PNin(output);
```

Figure 10.12. *CPN example code: PLP*

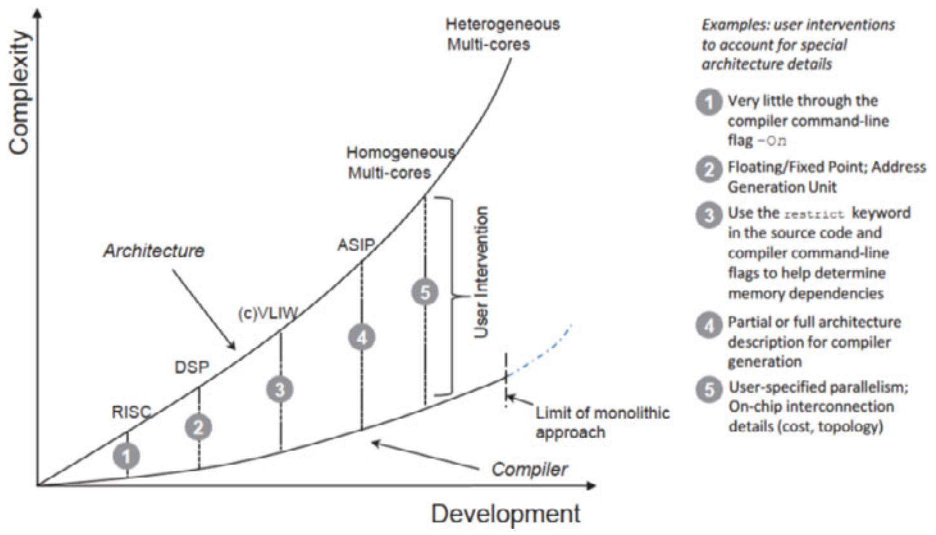


Figure 10.13. *Evolution of compiler development versus architecture development*

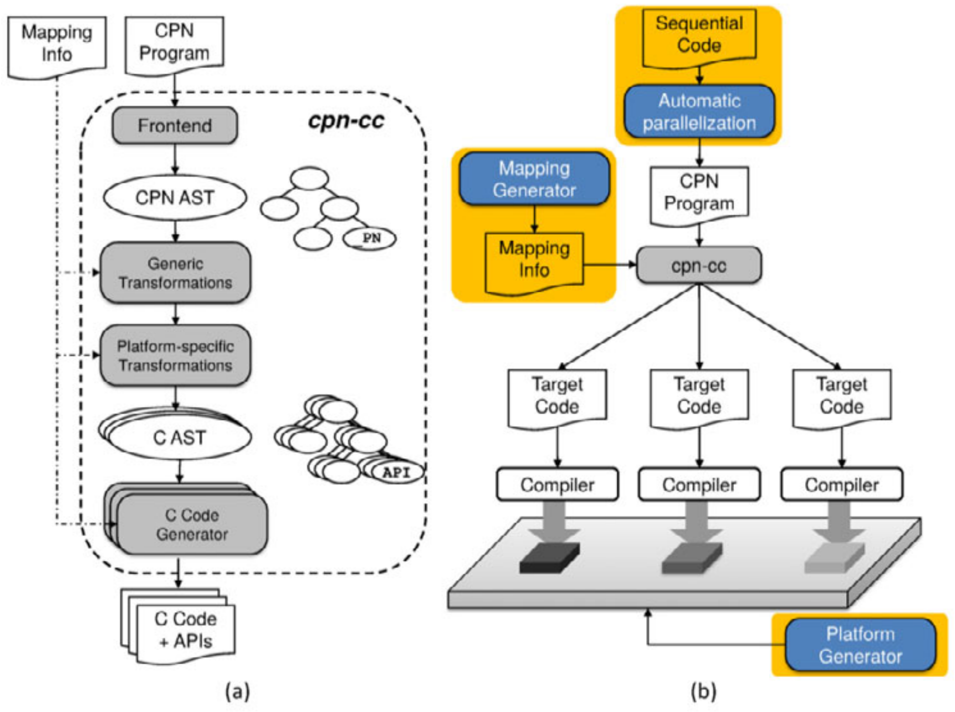
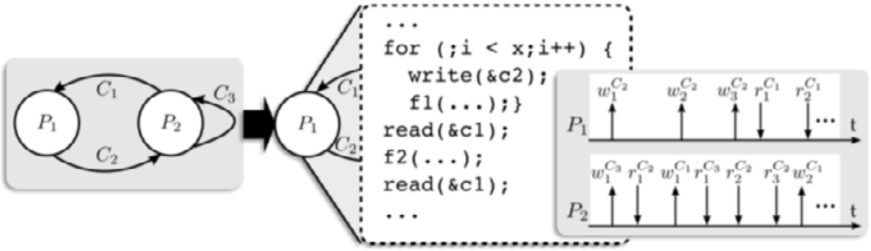


Figure 10.14. *(a) Structure of the source-to-source compiler cpn-cc. (b) An example of a complete compiler framework for a three-processor heterogeneous MPSoC: the parts with a shaded background are optional to the core compiler framework*


P_1
P_2
C_1
C_2
C_3
$w_1^{C_2}$ $w_2^{C_2}$ $w_3^{C_2}$ $r_1^{C_1}$ $r_2^{C_1}$ ...
$w_1^{C_3}$ $r_1^{C_2}$ $w_1^{C_1}$ $r_1^{C_3}$ $r_2^{C_2}$ $r_3^{C_2}$ $w_2^{C_1}$ ...
t

```
...
for (;i < x;i++) {
  write(&c2);
  f1(...);}
read(&c1);
f2(...);
read(&c1);
...
```

Figure 10.15. *Illustration of process traces. Example process network with a sample code of the logic of process P1, alongside a hypothetical trace of read and write events that may result from the process inner control flow*

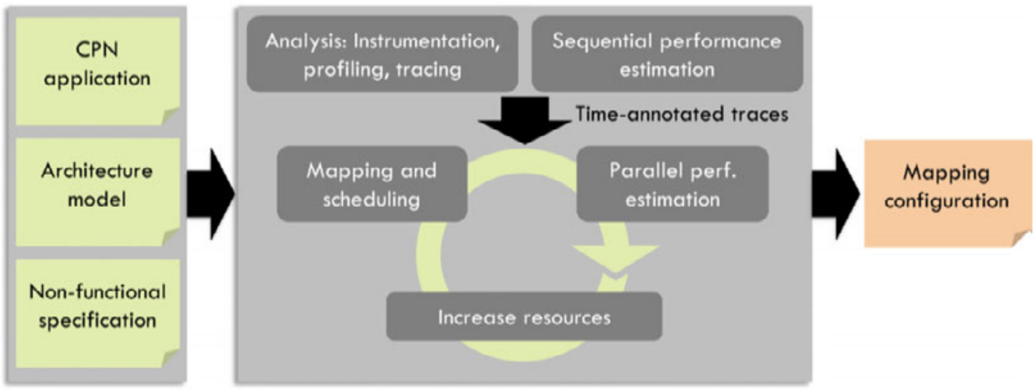

CPN application
Architecture model
Non-functional specification
Analysis: Instrumentation, profiling, tracing
Sequential performance estimation
Time-annotated traces
Mapping and scheduling
Parallel perf. estimation
Increase resources
Mapping configuration

Figure 10.16. *Overview of the mapping flow*

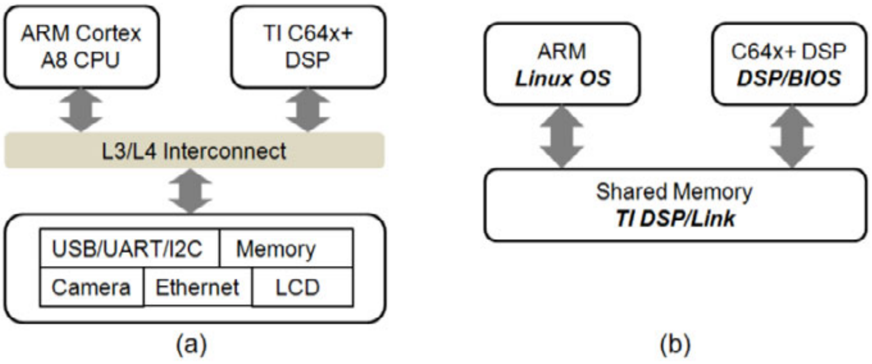


Figure 10.17. *(a) OMAP3530 block diagram. (b) Overview of TI OMAP software*

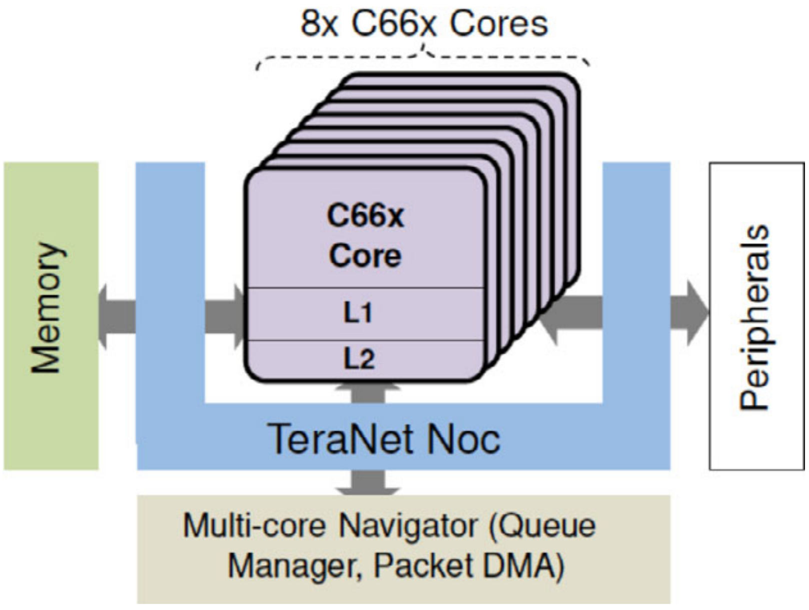
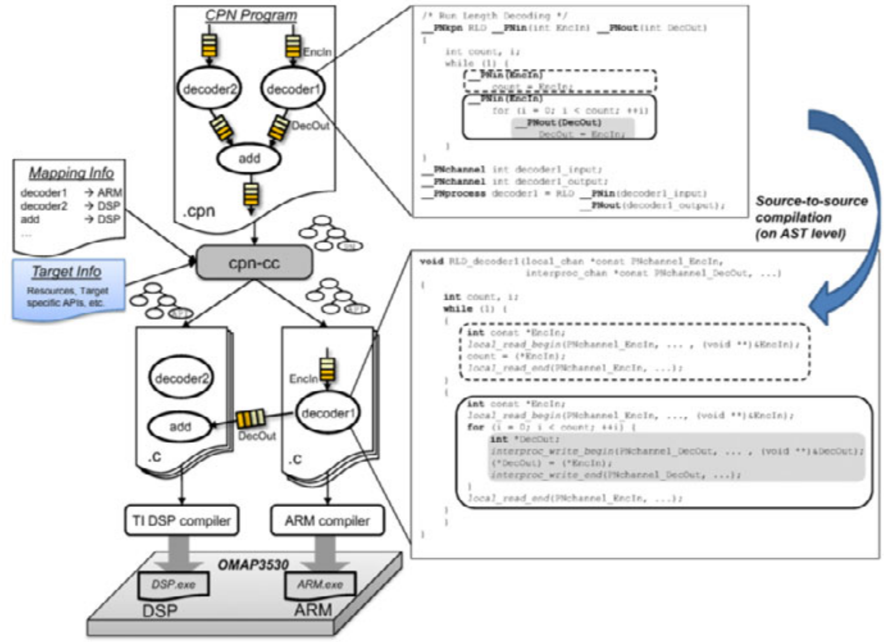


Figure 10.18. *TI C6678 block diagram*



```
/* Run Length Decoding */
__PNkpn RLD __PNin(int EncIn) __PNout(int DecOut)
{
    int count, i;
    while (1) {
        __PNin(EncIn)
            count = EncIn;
        __PNin(EncIn)
            for (i = 0; i < count; ++i)
                __PNout(DecOut)
                    DecOut = EncIn;
    }
}
__PNchannel int decoder1_input;
__PNchannel int decoder1_output;
__PNprocess decoder1 = RLD __PNin(decoder1_input)
                           __PNout(decoder1_output);
```

```
void RLD_decoder1(local_chan *const PNchannel_EncIn,
                  interproc_chan *const PNchannel_DecOut, ...)
{
    int count, i;
    while (1) {
    {
        int const *EncIn;
        local_read_begin(PNchannel_EncIn, ... , (void **)&EncIn);
        count = (*EncIn);
        local_read_end(PNchannel_EncIn, ...);
    }
    {
        int const *EncIn;
        local_read_begin(PNchannel_EncIn, ..., (void **)&EncIn);
        for (i = 0; i < count; ++i) {
            int *DecOut;
            interproc_write_begin(PNchannel_DecOut, ... , (void **)&DecOut);
            (*DecOut) = (*EncIn);
            interproc_write_end(PNchannel_DecOut, ...);
        }
        local_read_end(PNchannel_EncIn, ...);
    }
    }
}
```

Figure 10.19. *A complete compilation framework for OMAP3530 using MAPS*

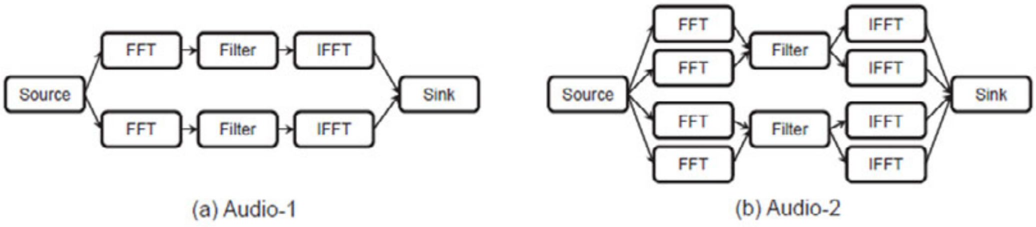


Figure 10.20. *Benchmark: digital audio filter*

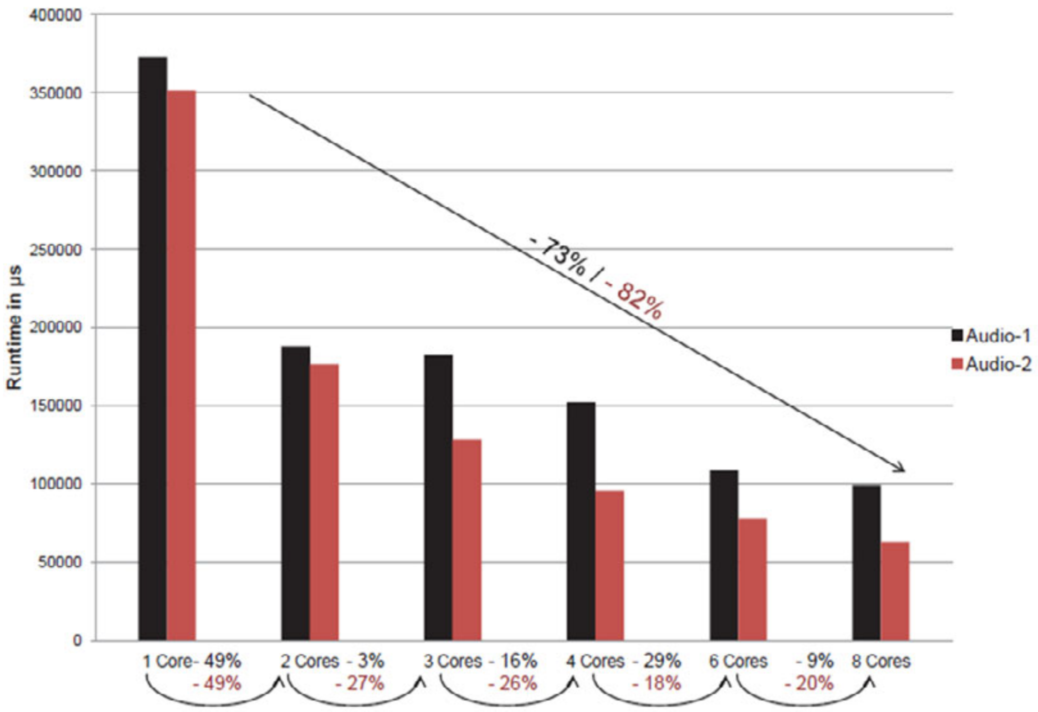


Figure 10.21. *Results of the audio filter (the percentage indicates the reduction of runtime between two mappings)*

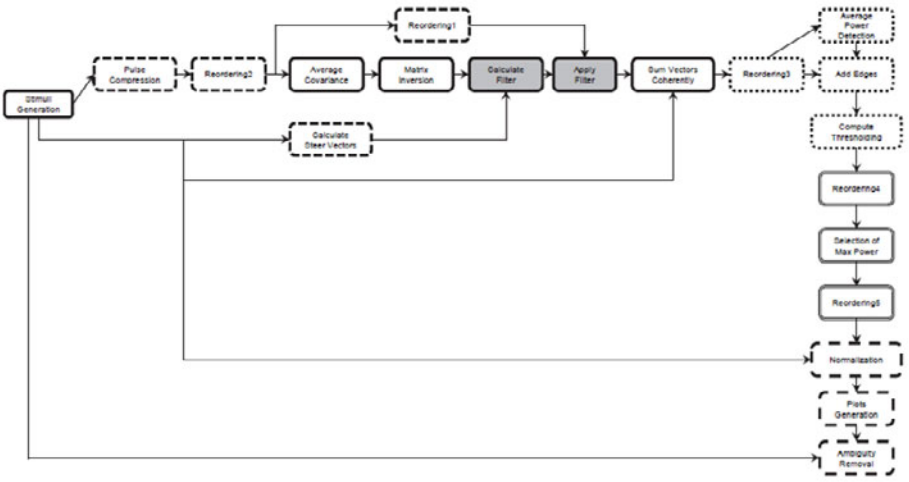


Figure 10.22. *Benchmark: radar application*

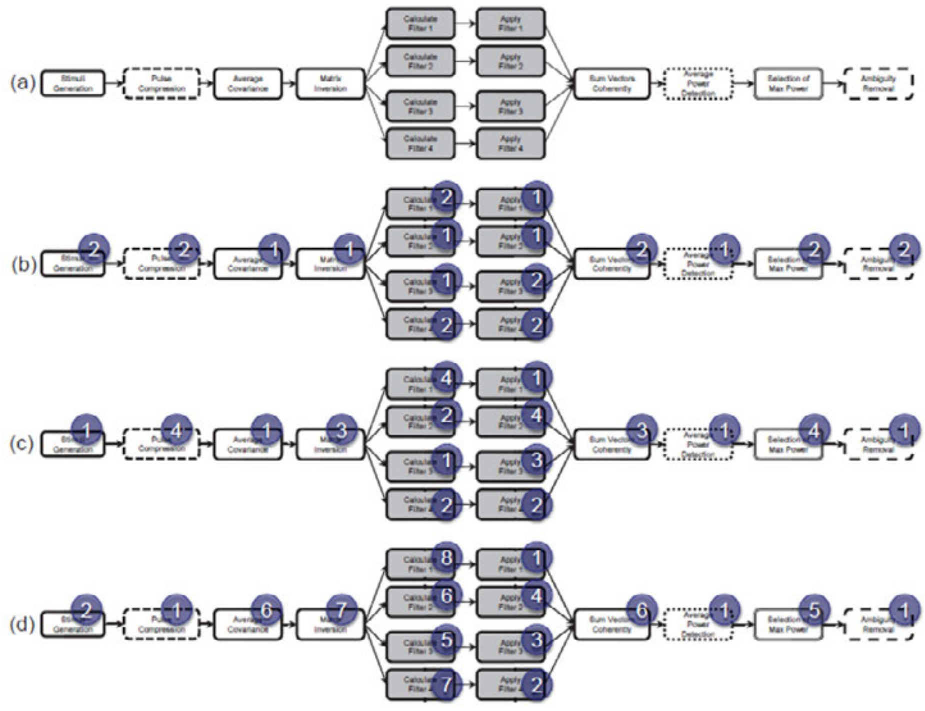


Figure 10.23. *Mapping results of the radar application: improved version. (a) Process network topology (only edges that communicate major data traffic are plotted for the sake of clarity. (b) Mapping onto two cores. (c) Mapping onto four cores. (d) Mapping onto eight cores*

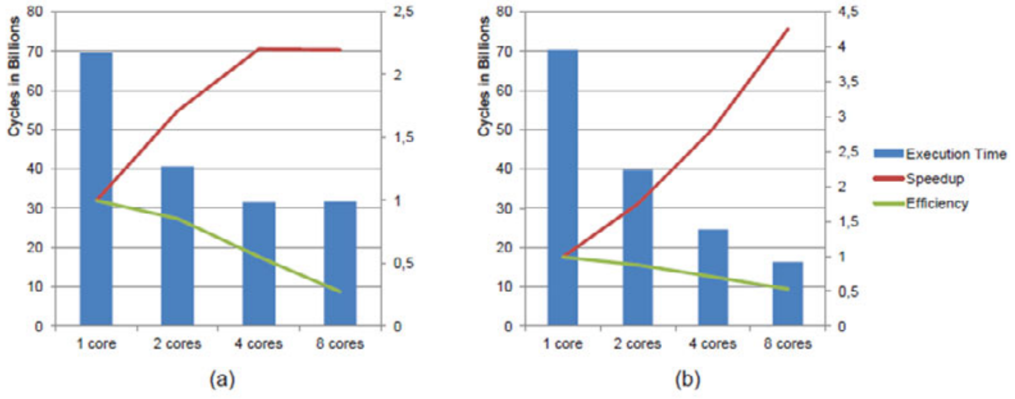


Figure 10.24. *Results of the radar applications: performance. (a) Initial version and (b) improved version.* $Speedup = \frac{Execution\ \ Time}{Sequential\ \ Time}$; $Efficiency = \frac{Speedup}{Number\ of\ cores\ used}$