ALOE Session 7: Computing Resource Management Framework

Vuk Marojevic, Ismael Gomez, Antoni Gelonch
Universitat Politècnica de Catalunya

June 2011

1. Objective

This session presents ALOE's computing resource management framework. You will learn how to use the framework, change its parameters, and create custom waveform and platform models.

2. Overview

- Introduction to computing resource management
- ALOE's computing system modeling and management
- Download and use the computing resource management framework
- Create custom application and platform models
- Exercises

3. Requirements

- C compiler
- Basic C programming skills

4. Versions

We continuously evolve the ALOE framework and tools. Consult http://flexnets.upc.edu/trac/wiki/ResourceManagement for the latest version of the

computing resource management framework. This session does not require the download and installation of ALOE.

5. Procedure

5.1. Introduction

In the early 90s Mitola envisaged radio transmitters and receivers (transceivers) that implement the entire signal processing chain in software. He coined this vision *software radio*. Software radio describes multistandard, multiservice, and multiband radio systems, which are software-reconfigurable. Software-defined radio (SDR) is a generalization of Mitola's software radio, as it describes transceivers that implement part of their physical layer processing in software.

SDR introduces flexibility to wireless systems: It permits adjusting or switching a terminal's radio access technology (RAT) implementation for adapting to changes in the heterogeneous radio environment. SDR platforms stand for software-programmable computing equipment, including handset transceivers, base stations, and core networks. SDR applications or waveforms refer to RAT-specific digital signal processing chain. Reconfiguring an SDR platform to execute another SDR application can then change radio communications link characteristics or even the entire radio standard. Dynamic RAT switches or modifications during communications sessions are also envisaged.

SDR presents a hard real-time computing challenge. The computing constraints increase as wireless systems evolve. Therefore, the flexibility of SDR terminals and network elements is a function of the computing resource managers, which need to continuously track the states of the computing resources and assign them properly.

An SDR application is the part of an SDR transceiver that is implemented in software. It consists of a set of concurrent processes that continuously process and propagate real-time data. Such a processing chain is not specifically tailored but rather executable on any general-purpose platform with sufficient computing capacity. Because of the similarities between future SDR applications and platforms and today's general-purpose computing applications and platforms, we consider general-purpose computing methods practical for SDR systems. We particularly think that the introduction of appropriate mapping and scheduling techniques will leverage the design of SDR platforms and applications. *Mapping* (*matching* in heterogeneous computing literature) describes the process of assigning software modules to hardware resources, whereas *scheduling* determines the execution times of these modules. Mapping and scheduling are two complementary computing resource management methods that facilitate a dynamic computing resource allocation under the given constraints.

5.2. ALOE's computing resource management framework

ALOE features a computing resource management (CRM) framework that permits a flexible and dynamic management of different types of computing system constraints and objectives. ALOE's CRM framework consists of several modules (Fig. 1). The basis for the resource management is ALOE's time management (Section 5.2.1).

The computing system modeling (Section 5.2.2) creates suitable models of SDR platforms and applications, capturing all relevant computing resources and requirements.

The computing resource management (Section 5.2.3) is based on the use of a general-purpose mapping algorithm and external cost functions. It manages (allocates, keeps track of, and updates) the available computing resources as a function of the computing requirements and management policy.

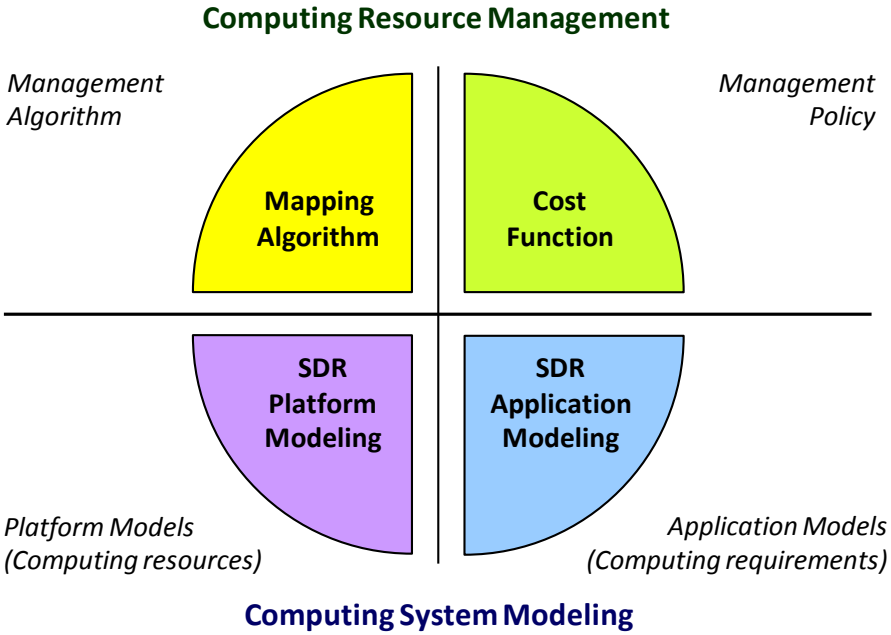

Figure 1 - CRM organization.

5.2.1. Time management

Metrics

An SDR platform represents an SDR mobile terminal or an SDR network element. These platforms comprise a few or many heterogeneous processing devices, such as FPGAs, DSPs, and GPPs, which communicate with each other. An FPGA's prime resource is the logic area for parallel processing, which can be converted to multiply-accumulate operations (MACs) per time unit when using well-defined benchmarks (filter, FFT, and so forth). DSP, GPP, and MP-SoC performances are typically given in million instructions per second (MIPS).

The processing powers and the inter-processor bandwidths are the primary resources of SDR platforms. We consider million operations per second (MOPS) as the basic unit for characterizing the processing powers and mega-bits per second (Mbps) for the

(inter-)processor communication capacities. We correspondingly apply the same metrics for capturing SDR applications' processing and data flow requirements. The implicit timing requirements need to be specified as a function of the radio link timing requirements.

Time-Slot Division and Pipelining

We consider processing time as just another limited computing resource. MOPS and Mbps embed this critical resource and thus permit an implicit time management. In continuation we discuss two mechanisms that ease the computing resource management.

Data that is transmitted or received over the wireless link needs to be processed for as long as there is data to transmit or receive. An SDR application will execute during the entire user session or until it is exchanged by another one. We thus propose breaking up the continuous execution into periodic executions by dividing the computing resource time in equidistant computing time slots and the SDR application in pipelining stages. Figure 2 illustrates this.

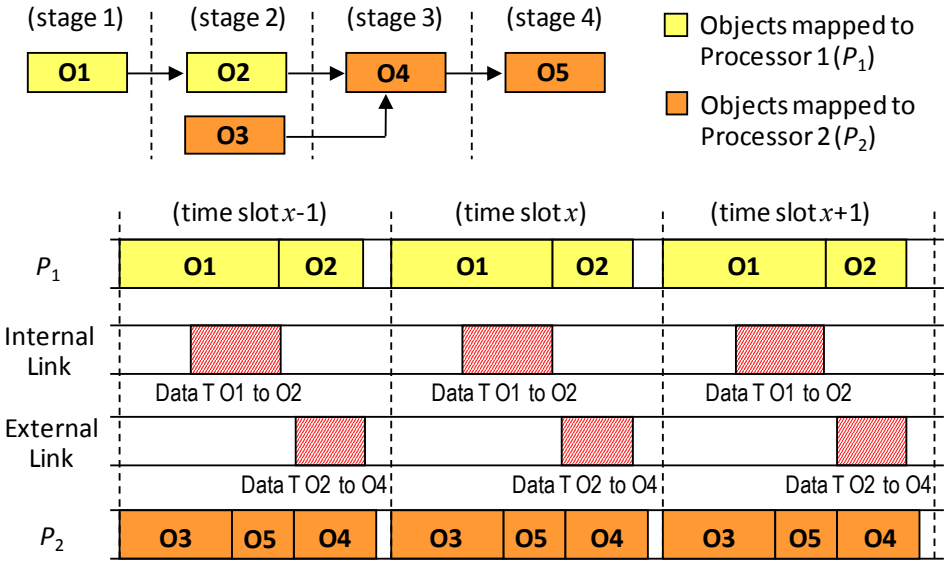

Figure 2 - Time slots and pipelining.

The pipelined execution of an SDR application establishes that, in any time slot, all SDR functions process and propagate some part of the data. That is, the same processing and data transfers repeat each time slot on a different data portion (Figure 2). We can then derive the new units million operations per time slot (MOPTS) and mega-bits per time slot (MBPTS) as $t \cdot$ MOPS and $t \cdot$ Mbps, where t is the time slot duration that is specified as a function of the latency requirements and the number of pipelining stages (see next subsection). MOPTS and MBPTS are the basic units for the SDR computing system modeling (Section 5.2.2).

Scheduling: Meeting the Real-Time Computing Constraints

The computing resource management facilities and our computing system modeling permit mapping an SDR application to an SDR platform on time slot basis, that is, the

assignment of computing requirements to computing resources within a single time slot frame.

We assume that coprocessors facilitate the concurrent data processing and data propagation on all processor's in- and outputs. Since repetitive operations on data samples and continuous outputs, often one per execution cycle, characterize digital signal processing, we may further assume that the software and hardware facilitate the immediate propagation of processed data samples. The SDR framework finally needs to manage the synchronized execution on all processors and provide pipelining and buffering mechanisms, among others, for the proper and timely data delivery. ALOE provides these mechanisms.

On the basis of a feasible mapping—a mapping that reserves no more than 100 % of any available computing resource—and under the above assumptions, the usually complex scheduling process can then be simplified to *N* independent local scheduling tasks. A processor's local scheduler need to organize the execution sequence of the corresponding SDR functions and their data transfers within the time-slot boundaries. A feasible mapping and a suitable scheduling ensures that the input data of any SDR application's module or set of modules is processed according to its arrival rate and without excessive data buffering, meeting the minimum bit rate requirement. The time slot duration provides direct control over the pipelining latency [2]; scheduling is a complementary tool for controlling the processing latency.

5.2.2. Computing system modeling

The SDR computing system modeling consists of the platform modeling and the application modeling. The platform modeling characterizes SDR platforms and their computing resources, whereas the application modeling abstracts SDR applications and their computing requirements. We identify four relevant types of computing resources: processing, bandwidth, memory and energy resources. The computing requirements correspondingly include the processing, dataflow, memory and energy demands. The following models address processing and interprocessor bandwidth capacities and requirements. These models are the basis for the computing resource management (Section 5.2.3).

Platform Modeling (Computing Resources)

The framework assumes an SDR platform model consisting of *N* interconnected processors P_1, P_2, ..., P_N. The *device model*

$$\boldsymbol{C} = (C_1, C_2, \ldots, C_N)\ \text{[MOPTS]} \tag{1}$$

is an *N*-element vector that captures the distributed processing capacities. The *communication model*,

$$\boldsymbol{Bx} = \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1N} \\ B_{21} & B_{22} & \cdots & B_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{N1} & B_{N2} & \cdots & B_{NN} \end{pmatrix} = \begin{pmatrix} \infty & B_{12} & \cdots & B_{1N} \\ B_{21} & \infty & \cdots & B_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ B_{N1} & B_{N2} & \cdots & \infty \end{pmatrix} \text{[MBPTS]}, \tag{2}$$

is an N times N matix, specifying the inter- and intraprocessor bandwidth capacities. B_{xy} indicates the available bandwidth for moving data from processor P_x to processor P_y. We assume that interprocessor bandwidths can be modeled as if they were infinite.

Bx thus informs about the communication topology (interconnectivity network) and the communication resources (bandwidths), assuming a network that consists of unidirectional communication links between each pair of processors. Additional information is needed for modeling shared links. We therefore suggest a more general communication modeling that distinguishes between the communication topology (3) and the communication resources (4).

$$\boldsymbol{I} = \begin{pmatrix} I_{11} & I_{12} & \cdots & I_{1N} \\ I_{21} & I_{22} & \cdots & I_{2N} \\ \vdots & \vdots & \ddots & \vdots \\ I_{N1} & I_{N2} & \cdots & I_{NN} \end{pmatrix} \quad (3)$$

represents the logical interconnection model, where $I_{uv} \in 1, 2, \ldots, N{\cdot}N$ is a unique numerical label of the logical link between P_u and P_v. A logical link corresponds to a directed (unidirectional) communication line between a pair of processors. These logical links map to physical link bandwidths

$$\boldsymbol{B} = (B_1, B_2, \ldots, B_N, B_{N+1}, \ldots, B_{N{\cdot}N}) = (\infty, \infty, \ldots, \infty, B_{N+1}, \ldots, B_{N{\cdot}N}) \text{ [MBPTS]}. \quad (4)$$

B_u, where $u = I_{32}$ for instance, is the maximum bandwidth that is available for the directed data transfer from the local data memory of processor P_3 to the local data memory of processor P_2. It would be zero if the physical link is unavailable or nonexistent. The first N elements of ***B***, B_1 to B_N, capture the processor-internal communication resources of processor P_1 to P_N; hence, $I_{uu} \in 1, 2, \ldots, N$. Since processor-internal data movements are typically orders of magnitude faster than processor-external data transfers, we can label logical links so that $B_1 \geq B_2 \geq B_3 \geq \ldots \geq B_{N{\cdot}N}$. Unused elements of ***B*** are filled with 0s [1].

The CRM framework accepts both models, but internally uses ***I*** and ***B***, which can be generated from ***Bx*** for certain interprocessor communication topologies: dedicated links between processor pairs (full- or half-duplex) or a single shared bus per platform.

Application Modeling (Computing Requirements)

SDR applications or waveforms are digital signal processing chains that consists of the M SDR functions $f_1, f_2, \ldots, f_M$. Filters, equalizers, or decoders are example SDR functions. Without loss of generally, we consider an SDR function as an indivisible software process. That is, any SDR function will execute without preemption on a single processor. The definition of processing blocks remains at the modeling level. Two different models of the same waveform are then considered as two waveforms by the computing resource manager.

The *function model*,

$$\boldsymbol{c} = (c_1, c_2, \ldots, c_M) \text{ [MOPTS]}, \quad (5)$$

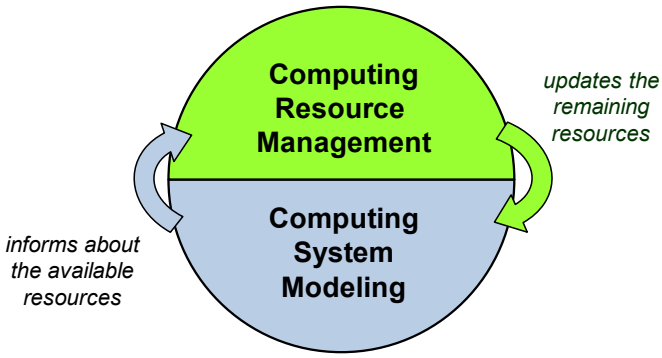

Figure 3 - Management interactions.

provides their processing requirements, whereas the *dataflow model*,

$$\boldsymbol{b} = \begin{pmatrix} b_{11} & b_{12} & \cdots & b_{1M} \\ b_{21} & b_{22} & \cdots & b_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ b_{M1} & b_{M2} & \cdots & b_{MM} \end{pmatrix} \begin{pmatrix} 0 & b_{12} & \cdots & b_{1M} \\ 0 & 0 & \cdots & b_{2M} \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \cdots & 0 \end{pmatrix} \text{[MBPTS]}, \quad (6)$$

indicates the precedence constraints between SDR functions as well as the data flow bandwidth requirements: $b_{xy} > 0$ indicates a data dependency between f_x and f_y. More precisely, process f_x sends data to process f_y and requires a bandwidth of b_{12} MBPTS for this data transfer. Since modeling SDR applications as directed acyclic graphs (DAGs), we can apply the logical numbering principle (if f_x sends data to f_y, then x<y) and ***b*** becomes a strictly upper diagonal matrix [1].

5.2.3. Computing resource management

The computing resource management relies on the computing system modeling. Figure 3 illustrates this relation. The framework features different mapping algorithms. All algorithms are general-purpose as they allow applying different cost functions or optimization criteria.

A mapping algorithm distributes the application modules among the limited and distributed computing resources. Consider, for example, the application model of Figure 8b in the appendix and the platform model of Figure 8c. The computing resource management problem here consists of allocating processor resources and communication facilities to the 24 applications modules and their data flows under the given computing constraints (limited resources and hard real-time requirements).

The t_w-mapping was introduced in [2] and is fully described in [1]. The following subsection provides a summary. The g_w-mapping is an extended or parameterized greedy algorithm. The generally low computing complexity makes the g_w-mapping applicable for large-scale computing resource management problems. It also serves as a baseline algorithm for evaluating the t_w-mapping results [1].

The *opt*-mapping does an exhaustive search over the entire solution space for finding the mapping of minimum cost for the given problem and cost function. The long execution times for reasonable problem sizes limit the applicability of the *opt*-

mapping. The problem size is defined by the number of processors (*N*) and processes (*M*): There are N^M different mappings of *M* processes to *N* processors. Table 1 indicates some mappings of 4 processes to 2 processors.

Table 1 - The different mappings solutions for N = 2 processors and M = 4 processes.

Mapping	Digital representation
$f_1, f_2, f_3, f_4 \rightarrow P_1$	0 0 0 0
$f_1, f_2, f_3 \rightarrow P_1$ $f_4 \rightarrow P_2$	0 0 0 1
$f_1, f_2, f_4 \rightarrow P_1$ $f_3 \rightarrow P_2$	0 0 1 0
$f_1, f_2 \rightarrow P_1$ $f_3, f_4 \rightarrow P_2$	0 0 1 1
$f_1, f_3, f_4 \rightarrow P_1$ $f_2 \rightarrow P_2$	0 1 0 0
$f_1, f_3 \rightarrow P_1$ $f_2, f_4 \rightarrow P_2$	0 1 0 1
$f_1 \rightarrow P_1$ $f_2, f_3, f_4 \rightarrow P_2$	0 1 1 1
$f_2, f_3, f_4 \rightarrow P_1$ $f_1 \rightarrow P_2$	1 0 0 0
$f_2, f_3 \rightarrow P_1$ $f_1, f_4 \rightarrow P_2$	1 0 0 1
…	…

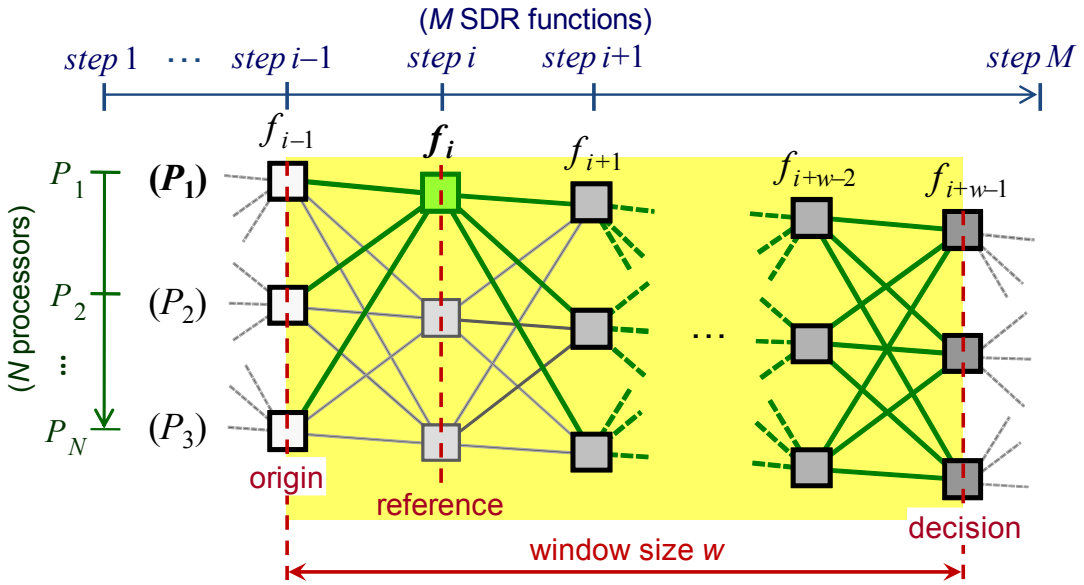

Figure 4 - t_w-mapping diagram and examined paths at *t*-node $\{P_1, f_i\}$.

The t_w-mapping

The t_w-mapping is a general-purpose mapping algorithm. It is a windowed dynamic programming algorithm, where w indicates the window size. The mapping process is organized by the t_w-mapping diagram, which contains a trellis of N times M (row times column) t-nodes. A t-node is identified as $\{P_j, f_i\}$ and absorbs the mapping of SDR function f_i to processor P_j. Any t-node at *step* i (column i in the t_w-mapping diagram) connects to all t-nodes at *step* i+1. The sequence of processors $[P_{k(0)}\ P_{k(1)}\ ...\ P_{k(w)}]_i$ identifies the w-path, a path of length w, that is associated with t-node $\{P_{k(1)}, f_i\}$, where $P_{k(0)}$ is the w-path's origin processor at *step* i–1 and $P_{k(w)}$ the destination processor at *step* i+w–1.

The main feature of the t_w-mapping is that it is cost function independent. That is, any cost function can, in principle, be applied. The cost function guides the mapping process. It is responsible for managing a platform's available computing resources and an application's real-time processing requirements.

The algorithm sequentially pre-assigns, or pre-maps, processes to processors, starting with SDR function f_1 and finishing with SDR function f_M (parts I and II of the algorithm). This is followed by a post processing that determines the final mapping (part III).

t_w-mapping, part I

Part I consists of pre-mapping SDR function f_1 to all N processors and storing the pre-mapping costs at t-nodes $\{P_1, f_1\}$ through $\{P_N, f_1\}$. Costs are computed due to some cost function, which is externally defined.

t_w-mapping, part II

At *step* i of part II ($2 \le i \le M$–w+1) the t_w-mapping examines all N^w w-paths that are associated with $\{P_{k(1)}, f_i\}$. These w-paths originate at a t-node of *step* i–1, pass through $\{P_{k(1)}, f_i\}$, and terminate at a t-node of *step* i+w–1. Figure 4 illustrates this for $P_{k(1)} = P_1$.

In case that $i < M$–w+1, the algorithm highlights the edge between a t-node at *step* i–1 and t-node $\{P_{k(1)}, f_i\}$ that corresponds to the minimum-cost w-path. The minimum-cost w-path is the path that is associated with the minimum accumulated cost due to the corresponding pre-mappings of f_1, f_2, ..., and f_{i+w-1}, where the w-path's origin t-node provides the pre-mapping information of f_1 to f_{i-1}. The algorithm then stores the cost and the remaining resources up to t-node $\{P_{k(1)}, f_i\}$ at $\{P_{k(1)}, f_i\}$. After processing all t-nodes at *step* i, the algorithm considering those at *step* i+1.

If $i = M$–w+1, however, the complete w-path of minimum cost is highlighted. The total cost and the remaining resources are then stored at $\{P_{k(1)}, f_{M-w+1}\}$. After finishing the processing of all N t-nodes at step M–w+1, part III of the algorithm starts.

t_w-mapping, part III

Part III tracks the t_w-mapping diagram backward and forward along the highlighted edges, starting at the minimum-cost t-node at *step* M–w+1. This process finds the complete mapping solution for the given problem and cost function.

The Cost Function

The cost function is responsible for managing the available computing resources of SDR platforms, trying to allocate the required resources to SDR applications. We generally define it as the sum of weighted *cost terms*, where each *cost term* captures the relation between the required and available resource of a specific type. Each of these terms must not be greater than 1. Otherwise, more resources would be allocated than available. Hence, the cost function computes the pre-mapping cost as a function of the remaining and the required resources. This implies dynamic resource updates (Figure 3).

For managing the processing and interprocessor bandwidth resources we define the cost function as

$$cost = q \cdot cost\_comp + (1-q) \cdot cost\_comm. \quad (7)$$

This two-term cost function manages the available processing and bandwidth resources, while trying to meet the corresponding computing resource requirements. Weight q is defined in interval [0, 1]. It defines the relative importance of the *computation cost* with respect to the *communication cost*.

Equation (7) represents the basic cost function. Additional weights are introduced in [3] and are available within the CRM framework.

5.3. Download and use the CRM framework

Go to http://flexnets.upc.edu/trac/wiki/ResourceManagement and download the CRM framework sources. Extract (unzip) the files, which are listed in Figure 7 in the appendix. The framework is completely implemented in C. You can view and modify these files using a text editor.

The *api_test.h* file defines several constants. You can choose among the *umts*, *umts2*, *IEEE*, *Frequenz*, *three_proc*, *four_proc*, *two_dim*, *three_dim*, or *custom* scenario. We briefly introduce these scenarios below.

5.3.1. CRM scenarios

The *umts* scenario corresponds to the first scenario presented in [2] or [1, Section 5.2]. It assumes part of an UMTS downlink receiver processing chain (chip- and bit-rate processing blocks) to be mapped to four SDR platform models of 3 processors each. The UMTS waveform and the four platforms are defined in *umts_models.c*.

The *umts2* scenario defines the bit-rate signal processing chain of a single-user WCDMA transmitter and receiver. The model is obtained from execution time measurements of the corresponding implementation (see *umts2_models.c*).

The *IEEE* scenario reflects the second simulation scenario of [2] or [1, Section 5.3]. The file *IEEE_models.c* specifies the four platform models of [2] and two additional models based on a shared bus communication network. The application models—data flow dependencies and processing and bandwidth requirements—are randomly generated

based on a set of parameters (see *api_test.c* and i*api_test.h*). They correspond to random directed acyclic graphs (DAGs).

Frequenz defines the simulation scenario of [4]. It creates 100,000 random DAGs, representing the computing models of different SDR applications. These DAGs are mapped to 9 SDR platform models defined in *Frequenz_models.c*.

The *three_proc* and *four_proc* constants define two platform models of 3 and 4 processors each (see *three_processor_models.c* and *four_processor_models.c*). The application models correspond to random DAGs.

The scenarios *two_dim* and *three_dim* define regular two- and three-dimensional arrays of processors, a mesh network (*two_dim_models.c*) and a hypercube (*three_dim_models.c*). The application models are, again, randomly generated.

The *custom* scenario defines custom SDR platform and application models. A two-processor platform model and a simple application model are defined in *custom_models.c*. You will edit this file later on in this session.

5.3.2. The UMTS scenario

Choose the UMTS scenario by setting the *umts* constant in *api_test.h*. Figure 8 illustrates the platform and application models of this scenario. Make sure to select only one scenario in *api_test.h*. Now compile and link the source codes. If you are using Linux, open a terminal, switch to the folder where you extracted the files and type

```
gcc -O3 -o mapping *.c
```

You can now execute the just compiled program with seven optional parameters:

```
./mapping [alg] [w] [q] [c_load] [b_load] [order] [k]
```

The first parameter (*alg*) specifies the algorithm: g_w-mapping (0), t_w-mapping (1), or opt-mapping (2). The t_w-mapping is chosen in *api_test.h* as the default algorithm. The second parameter (w) defines the window size of the g_w- or t_w-mapping (Section 5.2.3). The default value is 1. The third parameter modifies the cost function parameter *q*, which is initially set to 0.5. The fourth and fifth parameters specify the desired *c_load* and *b_load*, which we discuss later. The *order* indicates the mapping order. If 1 or 2 is passed as the input parameter, the waveform modeling matrices are processed in such a way that application modules are mapped in the order of decreasing processing (1) or bandwidth (2) requirements. The final parameter (*k*) indicates if an automatic cost function parameter weighting is applied; see [3] for details.

Execute *mapping* without any parameters. Figure 5 shows the output that you should observe. The first lines provide some information about the test suite and the employed mapping algorithm, window size, cost function parameter, and so forth. In this case, we have executed the t_1-mapping with q = 0.5. The mapping results are shown for each of the four platforms. Since only one application model is mapped to 4 platform models 0.0000 % indicates a feasible mapping solution, whereas a 100.0000 % would mean an infeasible mapping.

Figure 5 - Output.

```
This test suite applies the tw-, gw-mapping or optimal mapping
algorithm on different mapping scenarios.
Scenario and mapping options can be modified in "api_text.h".
Execution parameters are the algorithm window size 'w' and the cost
function parameter 'q'.
w = 1, 2, 3, ..., min{Wmax, M-1} - default: w = 1. Wmax = 5 (mapper.h)
and the number of application modules M = 24 (api_test.h).
q = 0, ...,1 - default: q = 0.5.
weighting = 0 (no additional cost function weighting), 1 (static
weights), 2 (dynamic weights).
Additional parameters are the processing and bandwidth loads (c_load,
b_load), which are used for scaling the computing resources (0: no
scaling).

**************************
*   t1-mapping    q = 0.50   *
**************************

 * c-load = 0.00
 * b-load = 0.00
 * Mapping in original (logical) order of processing blocks (no_ord)

Platform 1:       0.0000 % infeasible mappings
Platform 2:       0.0000 % infeasible mappings
Platform 3:       0.0000 % infeasible mappings
Platform 4:       0.0000 % infeasible mappings

Total processing demand (in the mean):    23247.16 MOPS
Total bandwidth demand (in the mean):     7030.22 Mbps
No. of skipped graphs:                    0

====================================
```

Now uncomment line 430 in *api_test.c*:

```
print_mapping(mapping, preproc.ord);
```

This function prints the processor allocation (mapping) and the cost of that solution. Save, compile, and execute the program again. Observe the different solutions for the same SDR application, but four different platforms.

Table 2 - Processing load parameters.

Parameter	Description
c_T	Application's total processing requirement: sum of the *M* processing requirements of SDR functions f_1 to f_M
C_T	Platform's total processing capacity: sum of the *N* processing resources of processors P_1 to P_N
sf_C	Scaling factor: scales the platform's processing resources C_1 to C_N

The UMTS scenario is limited to a single application and four platform models. To simulate different computing system constraints, we introduce the *processing* and *bandwidth loads*—*c_load* and *b_load*—which relate the application's total processing and bandwidth requirements to the platform's total processing and bandwidth resources:

$$c\_load = \frac{c_T}{sf_C \cdot C_T}\ , \tag{8}$$

$$b\_load = \frac{b_T}{sf_B \cdot B_T}\ . \tag{9}$$

The processing load parameters are defined in Table 2. The scaling factors sf_C and sf_B scale the platform's initial processing and bandwidth resources. They are computed from (8) and (9) and the desired *c_load* and *b_load*. Try executing the same scenario with different processing and bandwidth loads—(*c_load*, *b_load*) = (0.5, 2.25) or (0.95, 1.25), for example:

```
mapping 1 1 0.5 0.5  2.25
mapping 1 1 0.5 0.95 1.25
```

Try increasing the window size to *w* = 4, for example, and repeat the last case:

```
mapping 1 4 0.5 0.95 1.25
```

Observe that the mapping problem is now feasibly solved for platform 4. That is, the proposed mapping solution meets the SDR application's real-time computing requirements with the available resources. You can observe the remaining resources using the function `print_resources()`. Therefore, simply uncomment line 433 in *api_test.c*:

```
print_resources(mapping);
```

Do not forget to create a new executable file as any change in the codes requires recompilation and relinking to become effective.

5.4. Create custom computing system models

The *custom_models.c* file has been defined for exemplifying the introduction of new SDR application and platform models. Open the file in a text editor. The platform model corresponds to that of Figure 6a. The application model is only partially defined. Edit the corresponding sections—highlighted in Figure 9 in the appendix—to define the application model of Figure 6b. Note that in the C programming language, array elements are indexed from 0. `m_unsort[0]` then corresponds to the processing requirement of SDR function f_1, `m_unsort[1]` to that of f_2, and so forth. `b_unsort[0][3]`, for example, stands for the minimum bandwidth requirement for the data flow from f_1 to f_4.

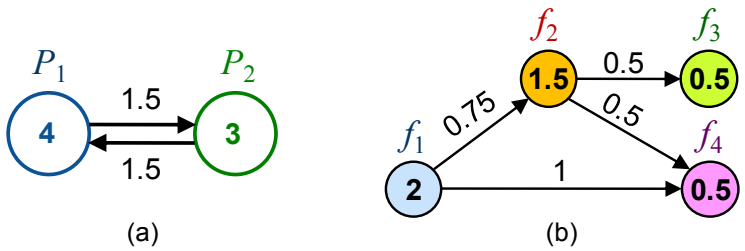

Figure 6 - Custom computing system models.

Deselect the *umts* scenario and select the *custom* scenario. That is, open *api_test.h* and set the *umts* constant to 0 and the *custom* constant to 1. Compile and execute the program:

```
gcc -O3 -o mapping *.c
./mapping
```

Does the mapping correspond to that of Fig. 4.13 of [1, page 58]? (The mapping cost should be $q \cdot 2.75 = 1.375$ here.)

Choose another algorithm or window size and run the simulation again.

6. Exercises

1. How many different mapping solutions exist for an SDR application of 21 functions and an SDR platform of 2 processors? And for 21 processes and 4 processors? Assuming 3 processors and 21 processes, how does the framework represent the following mapping:
   - $f_1, f_2, \ldots, f_8 \rightarrow P_1$
   - $f_9, f_{11}, f_{13}, f_{15}, f_{17}, f_{19}, f_{21} \rightarrow P_2$
   - $f_{10}, f_{12}, f_{14}, f_{16}, f_{18}, f_{20} \rightarrow P_3$

2. Compute the g_w-, t_w- and *opt*-mappings of the UMTS task graph to the four SDR platforms of the *umts* scenario. Use the following simulation parameters:
   - w = 1, 2, 3, 4, 5
   - q = 0.5
   - (*c-load*, *b-load*) = (0.8, 1), (0.5, 2)

   Create two tables, one for each (*c-load*, *b-load*) tuple, containing the mapping costs. In the last column relate the mapping costs of the g_w- and t_w-mappings to the *opt*-mapping. Does the opt-mapping solution depend on the window size?
3. Consider the following platform and application models:
   - $\boldsymbol{C}$ = (4, 4),
   - $\boldsymbol{B} = \begin{pmatrix} \infty & 1.3 \\ 1.6 & \infty \end{pmatrix}$,
   - $\boldsymbol{c}$ = (2, 1.7, 0.75, 0.5)
   - $\boldsymbol{b} = \begin{pmatrix} 0 & 0.75 & 0 & 1 \\ 0 & 0 & 0.6 & 0.4 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \end{pmatrix}$

   The g_2-mapping result is: 0 1 1 0
   The t_2-mapping result is: 1 0 0 1

   Can you tell what the processing load is on processor basis—*c_load*(P_1) and *c_load*(P_2)—for these two mapping results? What is the processing load on platform basis (*c_load*)? Does it depend on the mapping?

This finishes ALOE session 7. Please send your feedback to flexnets.pmt@upc.edu.

References

[1] V. Marojevic, "Computing Resource Management in Software-Defined and Cognitive Radios", Ph.D thesis, Universitat Politècnica de Catalunya, 2009. Available online: http://flexnets.upc.edu/trac/wiki/Publications

[2] V. Marojevic, X. Revés, A. Gelonch, "A computing resource management framework for software-defined radios," *IEEE Trans. Comput.*, vol. 57, no. 10, pp. 1399-1412, Oct. 2008.

[3] Vuk Marojevic, Ismael Gomez, Antoni Gelonch, "The FlexCRM Project", Universitat Politècnica de Catalunya, July 2011. Available online: http://flexnets.upc.edu/trac/wiki/ResourceManagement

[4] V. Marojevic, X. Revés, A. Gelonch, "Dynamic resource allocation in software defined radio – the interrelation between platform architecture and application mapping," *Proc. 4th Karlsruhe Workshop on Software Radios (WSR'06)*, Karlsruhe, Germany, March 22/23, 2006, pp. 39–44.

Appendix

Figure 7 - Computing resource management files.

```
/*******/
/* API */
/*******/
mapper.h



/****************************************/
/* COMPUTING RESOURCE MANAGEMENT FRAMEWORK */
/****************************************/
mapper_functions.h      /* contains the function prototypes */

mapper.c                /* uses the API and calls the corresponding
                        mapping functions */

sort_c.c                /* preprocessing: c-ordering */

sort_b.c                /* preprocessing: b-ordering */

tw_mapping.c            /* tw-mapping algorithm */

gw_mapping.c            /* gw-mapping algorithm */

opt_mapping.c           /* opt-mapping algorithm: computes all N^M
                        different mapping solutions and returns the
                        mapping of minimum cost */

cost_compU.c            /* computation cost (cost function term) */

cost_commU.c            /* communication cost (cost function term) */

generar_k.c             /* computes the k1 and k2 cost function
                        parameters */


/**************/
/* TEST SUITE */
/**************/
api_test.h              /* configures the test suite: defines the
                        mapping scenario and algorithm options*/

api_test.c              /* main function: mapping problem generation
                        and mapping framework invocation. */

printinfo.c             /* prints some basic information about the test
                        suit at the beginning of each execution */


/*******************************/
/* MAPPING PROBLEMS (SCENARIOS) */
/*******************************/
gendag4.c               /* application model with random precedence
                        constraints and data flow requirements */
```

```
umts_models.c            /* UMTS receiver chip- and bit-rate processing
                         model and 4 platforms of 3 fully-interconnected
                         processors */

umts2_models.c           /* UMTS transmitter and receiver bit-rate
                         processing models and 4 platforms of 3 fully-
                         interconnected processors */

IEEE_models.c            /* 6 SDR platforms of 3 fully-interconnected
                         processors (IEEE Trans. Comput., Oct 08) */

Frequenz_models.c        /* 9 SDR platforms of 3 fully-interconnected
                         processors (Frequenz, Sept/Oct 06) */

three_processor_models.c      /* 2 SDR platforms of 3 fully-
                         interconnected processors, one platform is
                         based on 3 half-duplex links and the other
                         features one shared bus */

four_processor_models.c       /* 2 SDR platforms of 4 fully-
                         interconnected processors with different
                         interprocessor communication networks: 6 half-
                         duplex links and one shared bus */

two_dim_models.c         /* An array of Nx = N2 * N2 processors
                         connected through a 2D communication network */

three_dim_models.c       /* An array of Nx = N3 * N3 * N3 processors
                         connected through a 3D communication network
                         (Hypercube) */

custom_models.c          /* custom platform and application models */


/*********************************/
/* RESOURCE MODELs TRANSFORMATION */
/*********************************/
resource_models_transformation.c    /* transforms communication model
                         Bx[][] to the internally used interconnection
                         or topology matrix I[][] and bandwidth resource
                         vector B[] */


/***************************/
/* PRESENTATION OF RESULTS */
/***************************/
printout.c               /* prints the % of infeasible mappings */

print_mapping.c          /* prints the mapping solution and cost */

print_resources.c        /* prints the remaining computing resources */

print_edges.c            /* prints the mapping decisions (highlighted
                         edges of the tw-mapping process */
```

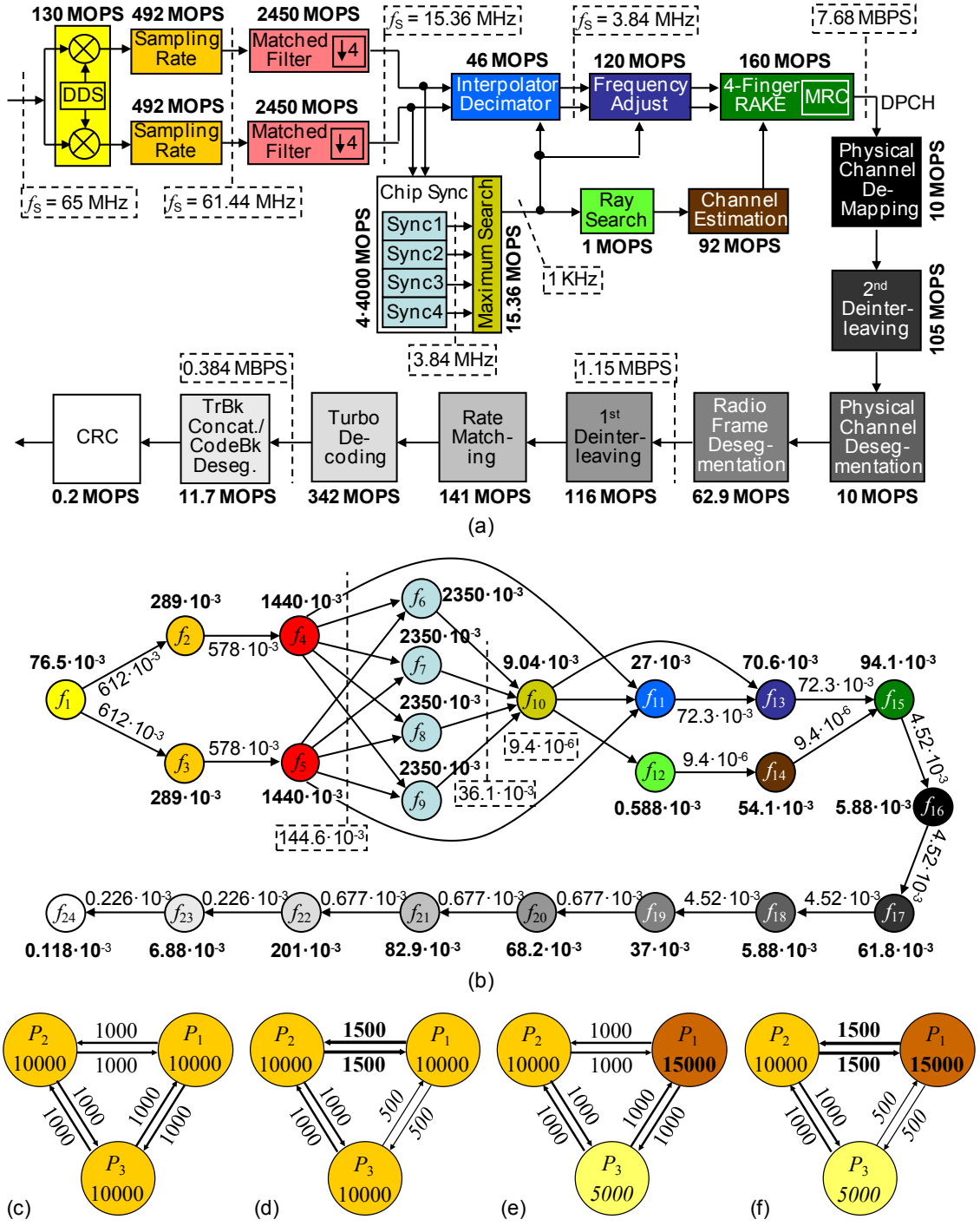

Figure 8 - UMTS scenario: UMTS task graph (a), application model (MOPTS and MBPTS) for a time slot duration of $0.588 \cdot 10^{-3}$ s (b), and platform models (MOPS and Mbps) (c-f).

Figure 9 - *custom-models.c*

```
// Resource Models

#include <stdio.h>
#include <stdlib.h>
#include <math.h>

#include "api_test.h"

extern float Px[Nx][R];             // processing powers of N
processors per resource model
extern float Bx[Nx][Nx][R-RB];            // Bandwidth matrices
extern float B_bus;

extern int arch_type[R];            // architecture

extern float b_unsort[Mx][Mx];
extern float m_unsort[Mx];                // unsorted (original) c-
vector

extern float Ctotal[R];
extern float Btotal[R];

app_totals custom_models(float c_load, float b_load)
{
      int i, j, r;                  // loop indices
      float c_total = 0;            // total processing demand
      float b_total = 0;            // total processing demand

      app_totals waveform_totals;   // structure containing two
parameters: c_total and b_total


      /********** SDR application **********/
      /* processing requirements:*/
      m_unsort[0] = 2;
      //m_unsort[1] = ...

      for (i=0; i<Mx; i++)
            c_total +=m_unsort[i];


      /* bandwidth requirements */
      /* the following 3 lines initialize the data flow matrix b with
zeros (no data flow dependencies) */
      for (i=0; i<Mx; i++)
            for (j=0; j<Mx; j++)
                  b_unsort[i][j] = 0;

      /* add here the non-zero data flow requirements */
      b_unsort[0][1] = (float)0.75;
      //b_unsort[0][3] = ...

      for (i=0;i<Mx;i++)
            for(j=0;j<Mx;j++)
                  b_total += b_unsort[i][j];


      /********* SDR platform ***********/
```

```
	/* processing capacities */
	Px[0][0] = 4;
	Px[1][0] = 3;

	Ctotal[0] = 0;
	for (i=0; i<Nx; i++)
		Ctotal[0] += Px[i][0];

	/* bandwidth capacities */
	/* the following 3 lines define the processor-internal data flow
capacities */
	for (r=0; r<R; r++)
	  for (i=0; i<Nx; i++)
			Bx[i][i][r] = (float)pow(10,8); // 'infinite' intra-
processor bandwidths

	arch_type[0] = fd; // fd/hd/bus: full-duplex/half-duplex/bus
interprocessor communication topology;

	// Full-duplex or half-duplex communication network
	Bx[0][1][0] = (float)1.5;
	Bx[1][0][0] = (float)1.5;

	// shared bus architecture
	B_bus = (float)1.5;

	Btotal[0] = 0;
	if (arch_type[0] == bus)
		Btotal[0] = B_bus;
	else
	{
		for (i=0; i<Nx; i++)
		{
			for (j=0; j<Nx; j++)
			{
				if (j!=i)
					Btotal[0] += Bx[i][j][0];
			}
		}
		if (arch_type[0] == hd)
			Btotal[0] = Btotal[0]/2;	// bidirectional link
is shared, rather than having two individual directional links (fd)
	}

	waveform_totals.c = c_total;
	waveform_totals.b = b_total;

	return waveform_totals;
}
```