Deliverable 5:

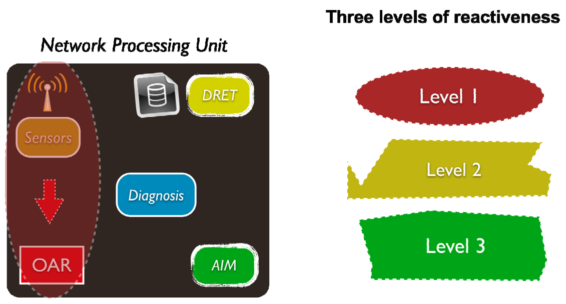
TECHNICAL REPORT DESCRIBING HOW TO WRITE STATISTICAL ANALYSIS AND INSTRUMENTATION THREADS THAT EVENTUALLY TRIGGER ACTIONS

Editor : LIRMM

Authors:
G. Sassatelli
P. Benoit
G. Marchesan Almeida

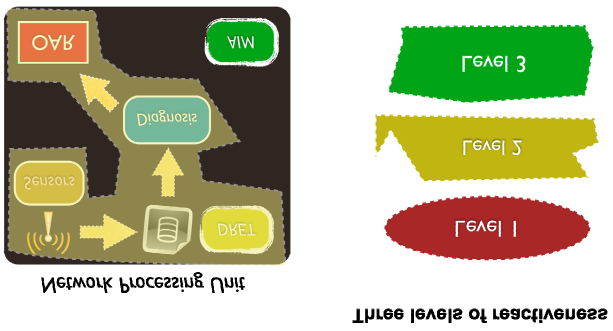
Among the remapping strategies that will later be presented in WP3; some, or a sequence of those applied over time may have hardly predictable effects on application performance. In order to keep track of mid- or long-term consequences of those remapping decisions, a statistical analysis of the DRET and AIM databases will be performed. These information may later help refining the decision-taking policy when, for instance, a previous task migration order led to a worst global solution.

0 represents the three different levels of reactiveness of the system. Starting from the monitoring system to the online application remapping, the system has three different ways to take decisions at run-time. The first one, called *Level 1* is the more reactive among them. Getting information of the system from the sensors (*i.e temperature*), the system in level 1 is responsible for taking decisions as fast as possible. The phase responsible for online application remapping (OAR) is not scope of this deliverable and will be further discussed in the future. A real scenario where level 1 would be applied could be the following: due some reason, the temperature of the *chip* goes up to a given threshold. In this case, a decision has to be taken as soon as possible in order to avoid the chip gets hot or even to be burned or non-functional. The information received from the sensors is not stored in a DRET (Distributed Raw Event Table) due the fact the time to store this information could be greater then the reaction time, damaging the system.



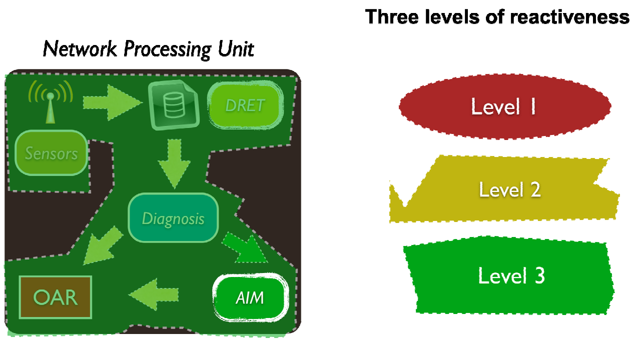
Representation of reactiveness level 1

In the second level of reactiveness, the information received from the sensors (*i.e fifo load, cpu workload*) is stored in a DRET. 0 depicts this scenario. After the information has been stored in a DRET, in the diagnosis phase the information will be analyzed and then a report of the actual state of the system is generated. Based on this information, the system will take some decisions by either changing some feature of the system or performing an online application remapping. In the first case, it could be reducing/increasing the frequency of the processor in order to cope with the goals of the system.



Representation of reactiveness level 2

In the first two examples all the information are processed as local without taking into account the state of neighbors NPUs or others nodes in the platform. In this case, an Application Instant Mapping (AIM) is used for storing information of remote NPUs in order to keep a general view of the platform at a given time. 0 represents the reactiveness level 3.

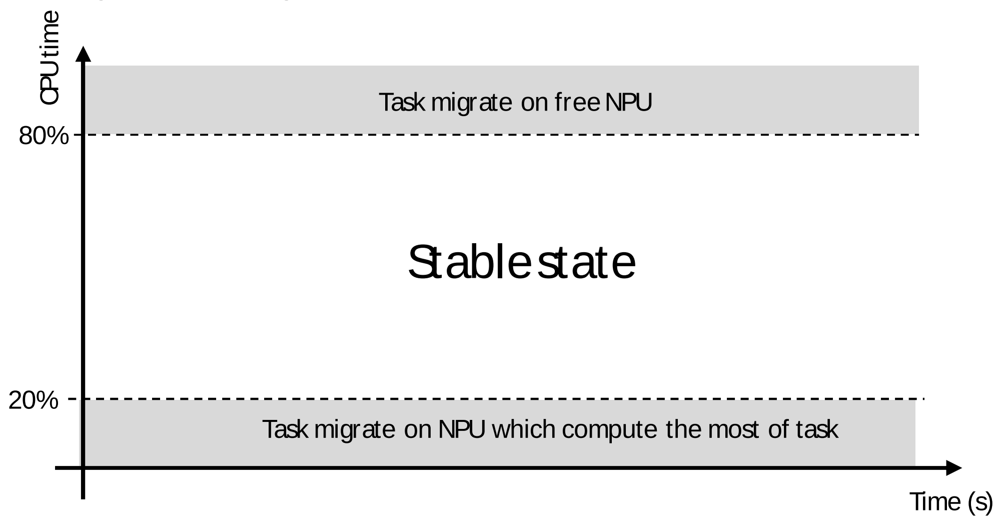


Representation of reactiveness level 3

Monitoring services are implemented to observe the system's behavior. The software threads implemented into the microkernel allow measuring the processor load, the communication load and the locality of tasks. Once monitoring data have been collected into the DRET, it is necessary to perform an analysis in order to take decisions regarding the system policy: this stage of the process is called diagnostic and decision, and will pilot the instrumentation threads that will eventually trigger actions such as task migration or DVFS.
We have implemented two simple mechanisms to improve the system performance (in terms of application throughput) based on the information of processor load and software FIFO load. The decision mechanism can be an order of migration, and/or frequency/voltage scaling (this part will be detailed in the 3rd deliverables of the project).

Statistical analysis and instrumentation thread based on Processor load

The first algorithm is based on processor load monitoring; it consists of triggering an action (*e.g.* task migration) on a given task when the CPU load is lower or greater than given thresholds (0).

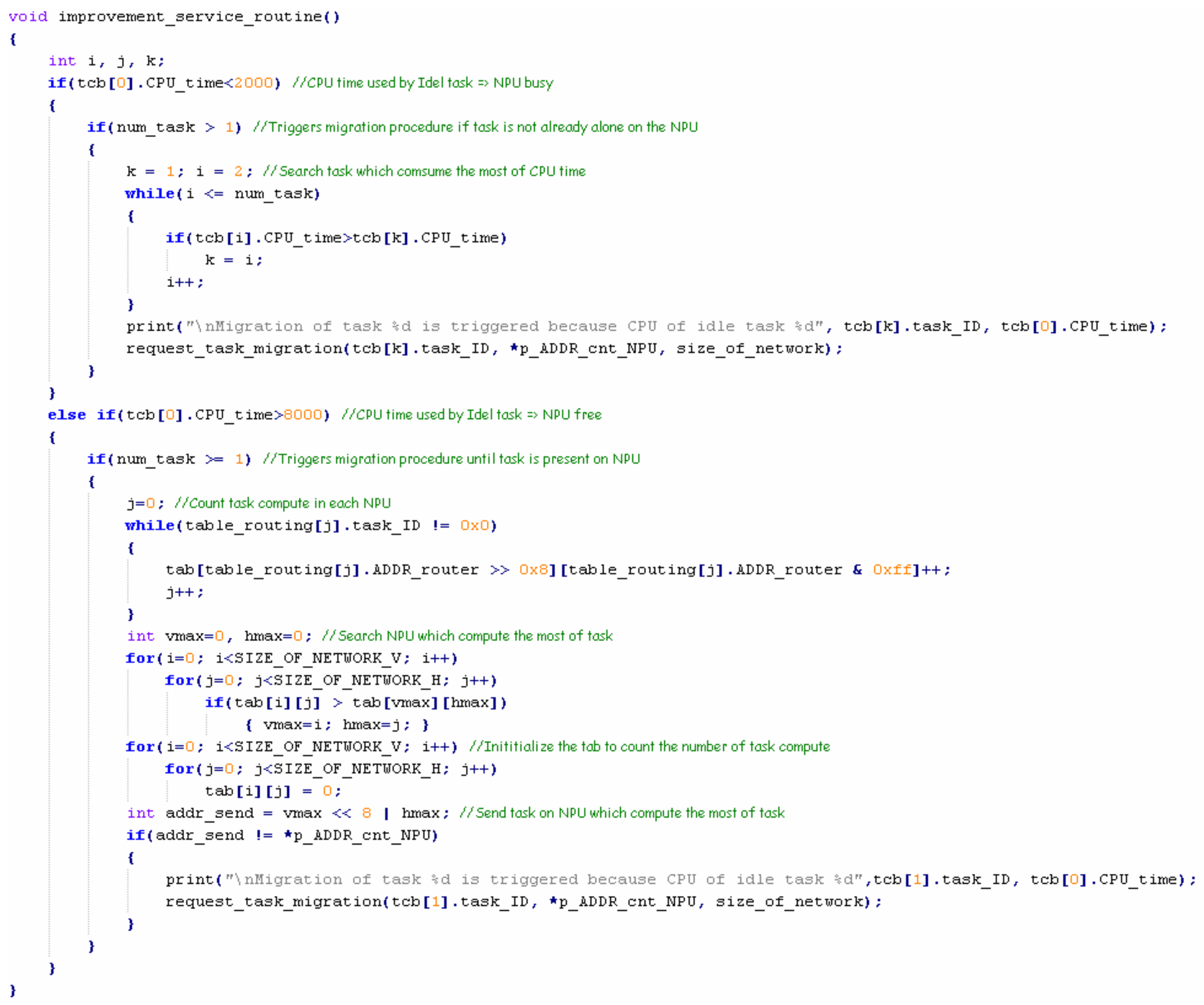


Processor load monitoring over time

The policy implemented here will tend to increase the usage of some processors by freeing marginally used processors, and to provide more CPU time to time-consuming tasks.
Two cases may be considered:

- If a set of tasks executed on a given NPU requires less than 20% of the CPU time, the tasks will be migrated to another NPU in order to reach a stable state (processor load average between 20 to 80%)
- If a set of tasks executed on a given NPU requires more than 80% of the CPU time, the critical task will be migrated to a *free* NPU.

The 0 depicts the thread handling this decision making process.



```c
void improvement_service_routine()
{
    int i, j, k;
    if(tcb[0].CPU_time<2000) //CPU time used by Idel task => NPU busy
    {
        if(num_task > 1) //Triggers migration procedure if task is not already alone on the NPU
        {
            k = 1; i = 2; //Search task which comsume the most of CPU time
            while(i <= num_task)
            {
                if(tcb[i].CPU_time>tcb[k].CPU_time)
                    k = i;
                i++;
            }
            print("\nMigration of task %d is triggered because CPU of idle task %d", tcb[k].task_ID, tcb[0].CPU_time);
            request_task_migration(tcb[k].task_ID, *p_ADDR_cnt_NPU, size_of_network);
        }
    }
    else if(tcb[0].CPU_time>8000) //CPU time used by Idel task => NPU free
    {
        if(num_task >= 1) //Triggers migration procedure until task is present on NPU
        {
            j=0; //Count task compute in each NPU
            while(table_routing[j].task_ID != 0x0)
            {
                tab[table_routing[j].ADDR_router >> 0x8][table_routing[j].ADDR_router & 0xff]++;
                j++;
            }
            int vmax=0, hmax=0; //Search NPU which compute the most of task
            for(i=0; i<SIZE_OF_NETWORK_V; i++)
                for(j=0; j<SIZE_OF_NETWORK_H; j++)
                    if(tab[i][j] > tab[vmax][hmax])
                        { vmax=i; hmax=j; }
            for(i=0; i<SIZE_OF_NETWORK_V; i++) //Inititialize the tab to count the number of task compute
                for(j=0; j<SIZE_OF_NETWORK_H; j++)
                    tab[i][j] = 0;
            int addr_send = vmax << 8 | hmax; //Send task on NPU which compute the most of task
            if(addr_send != *p_ADDR_cnt_NPU)
            {
                print("\nMigration of task %d is triggered because CPU of idle task %d",tcb[1].task_ID, tcb[0].CPU_time);
                request_task_migration(tcb[1].task_ID, *p_ADDR_cnt_NPU, size_of_network);
            }
        }
    }
}
```

Instrumentation thread responsible of triggering actions based on CPU load

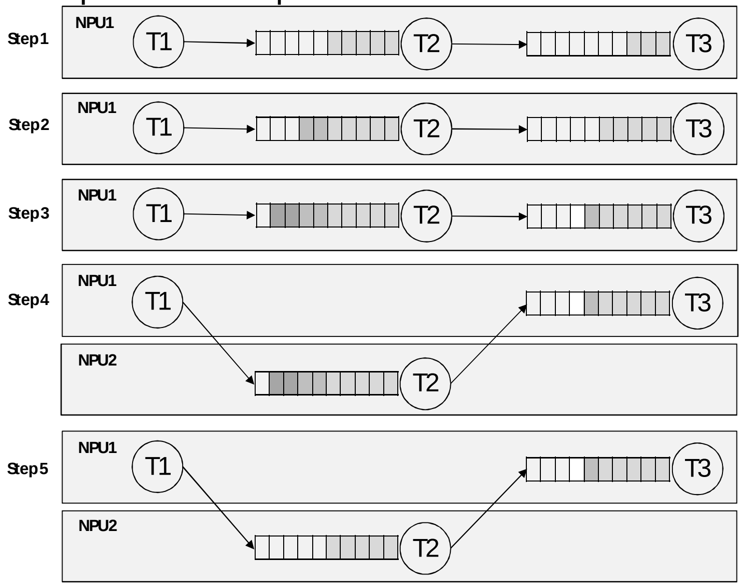
Statistical analysis of CPU load monitoring will allow the system to make some decisions according the better load balancing in the platform. By identifying the most critical tasks (most CPU time consuming), the system can trigger some tasks migrations in order to better distribute the load among all the NPUs.

Parameters of CPU load monitoring depend basically on the observation window. For a given time period, all the tasks in the NPU are monitored and the information regarding the CPU load for each task is stored into the DRET. The chose of the observation window plays an important rule in the life of the system. If the frequency of the monitoring system is high, the monitoring system can cause an overhead of the system. Otherwise, if it is very low, it can be dangerous when some urgent decision has to be ensured based on the

	ADAM	5		2010-03-18	LIRMM		1.1	5/8
	project	Deliverable		Date	Editor		Version	n°page

values recovered from the monitoring system. A good trade-off value is an important choice for the reactiveness/performance of the system.

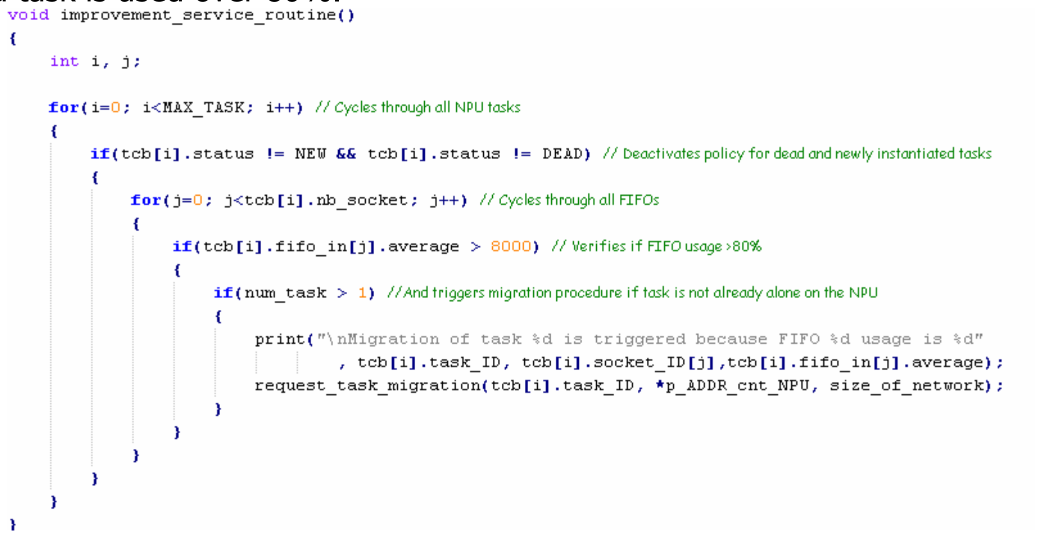
The second algorithm is based on FIFO monitoring; the idea is to trigger an action on a given task whenever the FIFO usage is greater than a given threshold. Basically, if one SW FIFO is used above 80%, it means that computation and communication is not well balanced for a given task. Assuming a fixed incoming communication throughput, the computational speed should be increased in order to improve the application throughput. It would be possible for instance to adjust the operating frequency of the NPU to improve computational to communication load balancing. Considering remapping decisions, one possibility is to give more CPU time to the considered task, if it is sharing a processor with other tasks. As a consequence, if the SW FIFO of a given task exceeds the 80% threshold, the task will be moved to a "free" processor. The 0 depicts an example of such a situation.



Software FIFO load monitoring and remapping decisions

Initially, in the Step 1 the input FIFO usage of T2 is 50% and T3 is 30%. Step 2 shows T1 is sending packet faster than T2 is capable to process. So, the FIFO usage increases rapidly and reaches 90% of usage, as shown in Step 3. Following the algorithm based on FIFO usage, whenever the FIFO usage is greater than a given threshold (80%), the task must be migrated. Therefore T2 is migrated to the first "free" NPU, in this case NPU2 (Step 4), and the whole FIFO content is copied to the new NPU. As a result, the FIFO usage decreases (Step 5) resulting in a well-balanced communication / computational trade-off.
The 0 shows an implementation of the instrumentation thread responsible of triggering task migration based on communication load. The presented

algorithm simply activates task migration, in case that one of the FIFO queues of a task is used over 80%.



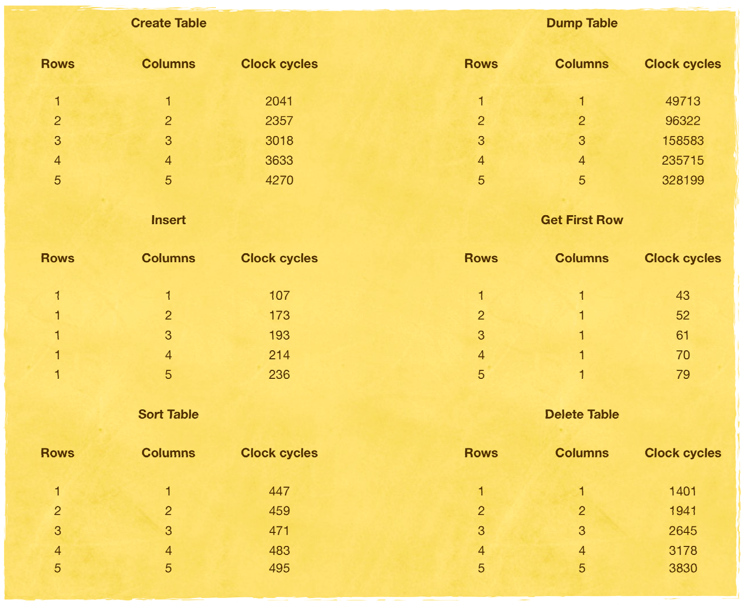
```
void improvement_service_routine()
{
    int i, j;

    for(i=0; i<MAX_TASK; i++) // Cycles through all NPU tasks
    {
        if(tcb[i].status != NEW && tcb[i].status != DEAD) // Deactivates policy for dead and newly instantiated tasks
        {
            for(j=0; j<tcb[i].nb_socket; j++) // Cycles through all FIFOs
            {
                if(tcb[i].fifo_in[j].average > 8000) // Verifies if FIFO usage >80%
                {
                    if(num_task > 1) //And triggers migration procedure if task is not already alone on the NPU
                    {
                        print("\nMigration of task %d is triggered because FIFO %d usage is %d"
                                , tcb[i].task_ID, tcb[i].socket_ID[j],tcb[i].fifo_in[j].average);
                        request_task_migration(tcb[i].task_ID, *p_ADDR_cnt_NPU, size_of_network);
                    }
                }
            }
        }
    }
}
```

Instrumentation thread responsible of triggering actions based on SW FIFO load

Statistical analysis of SW FIFO load monitoring is one more mechanism that will allow the platform to take some decisions in order to have a better load balancing.

DRET functions profiling

In order to calculate the performance of the DRET API, we have profiled some functions available in the DRET API developed by one of the partners of the project (LIP6). 0 shows the performance, in number of clock cycles, for six different functions available in the DRET API (*create table, insert, sort table, dump table, get first row, delete table*).



Create Table			Dump Table		
Rows	Columns	Clock cycles	Rows	Columns	Clock cycles
1	1	2041	1	1	49713
2	2	2357	2	2	96322
3	3	3018	3	3	158583
4	4	3633	4	4	235715
5	5	4270	5	5	328199

Insert			Get First Row		
Rows	Columns	Clock cycles	Rows	Columns	Clock cycles
1	1	107	1	1	43
1	2	173	2	1	52
1	3	193	3	1	61
1	4	214	4	1	70
1	5	236	5	1	79

Sort Table			Delete Table		
Rows	Columns	Clock cycles	Rows	Columns	Clock cycles
1	1	447	1	1	1401
2	2	459	2	2	1941
3	3	471	3	3	2645
4	4	483	4	4	3178
5	5	495	5	5	3830

Profiling DRET API functions

For each function, we present the number of clock cycles required for executing the action varying the number of rows/columns. By this analysis, it is possible to observe that the *dump table* is the most critical function in the API. This is justified by the fact that this function calls many times the *printf* function which has access to the UART and its processing it is very slow. Therefore, this function will be used only for debugging purposes, not affecting the performance of the system.
The same analysis can be done for *create* and *delete table* functions. These functions are used only in the initialization of the system as well in the end of the processing. Then, the performance of the system is affected only twice, at the beginning and at the end of the processing, not being a considerable overhead.
In terms of memory footprint, the DRET API with all the available functions occupies 8848 bytes only. It validates the usage of the DRET API in terms of memory occupation for the context of the ADAM project.
Although the memory overhead of the system imposed by the usage of the DRET, the overall gain can be very beneficial for the system. By using a database for storing monitoring information, it gives to the system the possibility to handle some decisions through the analysis and diagnosis and then allowing the system to better decide the application mapping for tuning it according to the requirements.