FlexRAN Reference Solution Framework

Programmer's Guide

March 2021

Revision 4.0

Intel Confidential

Document Number: 576898-4.0

You may not use or facilitate the use of this document in connection with any infringement or other legal analysis concerning Intel products described herein. You agree to grant Intel a non-exclusive, royalty-free license to any patent claim thereafter drafted which includes subject matter disclosed herein.

No license (express or implied, by estoppel or otherwise) to any intellectual property rights is granted by this document.

All information provided here is subject to change without notice. Contact your Intel representative to obtain the latest Intel product specifications and roadmaps.

Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations, and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

The products described may contain design defects or errors known as errata, which may cause the product to deviate from published specifications. Currently characterized errata are available on request.

Optimization Notice: Intel's compilers may or may not optimize to the same degree for non-Intel microprocessors for optimizations that are not unique to Intel microprocessors. These optimizations include SSE2, SSE3, and SSSE3 instruction sets and other optimizations. Intel does not guarantee the availability, functionality, or effectiveness of any optimization on microprocessors not manufactured by Intel. Microprocessor-dependent optimizations in this product are intended for use with Intel microprocessors. Certain optimizations not specific to Intel microarchitecture are reserved for Intel microprocessors. Please refer to the applicable product User and Reference Guides for more information regarding the specific instruction sets covered by this notice.

Notice Revision #20110804

Tests document performance of components on a particular test, in specific systems. Differences in hardware, software, or configuration will affect actual performance. Consult other sources of information to evaluate performance as you consider your purchase. For more complete information about performance and benchmark results, visit www.intel.com/benchmarks.

Copies of documents which have an order number and are referenced in this document may be obtained by calling 1-800-548-4725 or by visiting: http://www.intel.com/design/literature.htm

Intel technologies' features and benefits depend on system configuration and may require enabled hardware, software or service activation. Learn more at http://www.intel.com/ or from the OEM or retailer.

No computer system can be absolutely secure.

Intel, Xeon, and the Intel logo are trademarks of Intel Corporation in the U.S. and/or other countries.

*Other names and brands may be claimed as the property of others.

Copyright © 2021, Intel Corporation. All rights reserved.

Contents

1.0 Introduction 7

1.1 Purpose of this Document 7
1.2 Intended Audience 7
1.3 Terminology 7
1.4 References and Resources 8

2.0 BBU Pooling Framework 10

2.1 BBU Pooling Framework Components 10
2.2 System Architecture 10
2.3 Overall Relational Model 11
2.4 Task Property 12
2.4.1 Task Function 13
2.4.2 Task Priority 13
2.4.3 Task Deadline 13
2.4.4 Task Dependency 13
2.4.5 Task Chain 14
2.4.6 Task Affinity 15
2.5 Task Queue 15
2.6 Time Period 16
2.7 Task Scheduler 16
2.7.1 Work Flow 16
2.7.2 Priority Sort 17
2.7.3 Get a Job 18
2.7.4 Synchronization Control 18
2.8 Task Generation 19
2.8.1 Application Trigger 20
2.8.2 Auto-generation 20
2.8.3 Task Pre-fetch 20
2.8.4 Task Split 20
2.9 Task Execution 20
2.10 Hardware Tasks 20
2.10.1 Working Mode 21
2.11 Core Management 22
2.12 Status Report 23
2.13 Performance 23
2.13.1 Test System 23
2.13.2 Test Scenario 24
2.13.3 Performance 25

3.0 Enhanced BBU Pool Framework 30

3.1 Introduction 30
3.2 System Architecture 30
3.3 Event 31
3.4 Priority Queue 31
3.5 Thread 32
3.5.1 Consumer Thread(s) 32
3.5.2 Producer Thread(s) 34

3.5.3 Maintenance Thread 35
3.6 Application Usage Introduction 35
3.6.1 Function Calls 37
3.6.2 Task/Event Parameters 38
3.6.3 Consumer Thread Core Utilization Statistics 38
3.7 Test and Performance 39
3.7.1 Test System 39
3.7.2 Test Cases 40
3.7.3 Test Performance 41

4.0 How to Build and Run Framework 43

4.1 Overall Folders 43
4.2 BBU Pooling 44
4.2.1 System Requirements 44
4.2.2 Compile and Run 44
4.2.3 Use Example Code 45
4.3 Enhanced BBU Pool 46
4.3.1 System Requirements 46
4.3.2 Compile and Run 46

Figures

Figure 1. BBU Pooling Framework Components and Hierarchy 10
Figure 2. BBU Pooling Task Architecture 11
Figure 3. BBU Pool Overall Relational Model 12
Figure 4. Task Properties 12
Figure 5. Task Dependency - Type 1 13
Figure 6. Task dependency - Type 2 14
Figure 7. Task dependency - Type 3 14
Figure 8. Task Chain 15
Figure 9. Task Queue 16
Figure 10. Scheduler Work Flow 17
Figure 11. Non-synchronization Mode 18
Figure 12. Synchronization Mode 19
Figure 13. Generation Work Flow 19
Figure 14. HW Pooling Mode 21
Figure 15. HW Trigger Mode 22
Figure 16. Core Management 22
Figure 17. Processing Flow 23
Figure 18. Test System 24
Figure 19. Case 0 performance 26
Figure 20. Case 1 Performance 27
Figure 21. Case 2 performance 28
Figure 22. Case 3 performance 29
Figure 23. Enhanced BBU Pool Architecture 30
Figure 24. Consumer Thread Logic 33
Figure 25. Consumer Thread Event Fetch Logic 34
Figure 26. Producer Thread Event Send Logic 35
Figure 27. Event Chain Example 36
Figure 28. Event Distribution Example 37

Figure 29. eBBUPool Example Event Processing Pipeline 39
Figure 30. eBBUPool Performance Test Print 41
Figure 31. BBU Pooling Framework Folder Structure 43
Figure 32. Enhanced BBU Pool Folder Structure 43

Tables

Table 1. Terminology....7
Table 2. Reference Documents and Resources 8
Table 3. Performance with TTI Length 25
Table 4. Functional Case Definition of eBBUPool....40
Table 5. Performance Case Definition of eBBUPool....40
Table 6. eBBUPool Performance Test Server Config....41
Table 7. eBBUPool Performance Profiling Result....41

Revision History

Revision	Description	Date
4.0	FlexRAN Software Release v21.03: • Updated Chapter 2.0, BBU Pooling Framework • Added Chapter 3.0, Enhanced BBU Pool Framework • Updated Section 4.3	March 2021
3.0	FlexRAN Software Release v19.10: • Updated Section 4.2.1.3	October 2019
2.0	Software Release v18.09 • Revised Table 2, Reference Documents and Resources • Section 3.10 updated • Section 3.11 updated • Added Figure 17, Processing Flow • Added Figure 18, Test System • Section 3.12 Performance updated Tests and added Table 3, Performance with TTI Length and Figures 19 through 22. • Removed Sections 4 and 5, BPU Pooling • Section 6.1, added Figure 30 • Section 6.2 updated BBU Pooling Tasks • Removed Sections 6.3 and 6.4, BPU Pooling	October 2018
1.0	Initial Release for FlexRAN v1.5.0. This document combines the former 574867 Intel® FlexRAN Framework User Guide and 572010 Intel® FlexRAN Framework Software Architecture Specification.	April 2018

§

1.0 Introduction

The *FlexRAN Reference Solution Framework Programmer's Guide* describes the software architecture and running methodology of two provided frameworks respectively: Base Band Unit (BBU) Pooling and Enhanced BBU Pool (eBBUPool). The document also provides instructions on specific test scenarios to determine performance.

1.1 Purpose of this Document

Both frameworks provide the functionality to schedule and dispatch tasks/events in multi-processor system:

- BBU Pooling integrates some RAN application specific functionalities and uses semi-static task chain management to achieve extremely high performance for traditional RAN cases
- eBBUPool acts as a generic software event scheduler that supports various application and flexible event chain management. It can run multi-type applications in same core pool and provides good scalability in core count.

This document helps audience understand both frameworks' mechanism and how to use them. To better understand the contents, the following documents should be read in the suggested order:

- *FlexRAN Reference Solution Software Release Notes, v21.03*

  Release notes provide specific release information including supported features, limitations, fixed issues, and known issues.

- *FlexRAN Reference Solution Framework Programmer's Guide* (this document)
- *FlexRAN Reference Solution Framework API Reference*:

  This API reference provides detailed API information, data structures, and other programming constructs (generated from Doxygen) for both frameworks.

1.2 Intended Audience

The intended audiences for this document include architects, requirements managers, software and system testers, project managers, and software developers and engineers.

1.3 Terminology

Table 1. Terminology

Term	Description
API	Application Programming Interface
BBU	Base Band Unit
CU	Radio Access Network Control Unit

Term	Description
CRC	Cyclic Redundancy Check
DPDK	Data Plane Development Kit
DU	Radio Access Network Data Unit
FPGA	Field-Programmable Gate Array
Framework	Middleware software based on IA oriented parallel task execution of the software module, optimized for multi-core IA architecture.
FlexRAN	Flexible radio access network. Intel IA based LTE,5G RAN implementation, including IA optimized RAN application, protocol stack on IA and HW accelerator.
eBBUPool	Enhanced BBU Pool
Event Device	A hardware or software-based event scheduler
HARQ	Hybrid Automatic Repeat Request
IA	Intel® Architecture
LTE	Long Term Evolution
NUMA	Non-uniform Memory Access
PMD	Polling Mode Driver
PUSCH	Physical Uplink Shared Channel
QoS	Quality of Service
TTI	Transmission Time Interval

1.4 References and Resources

Table 2. Reference Documents and Resources

Document or Reference	Document No./Location
FlexRAN Reference Solution Software Release Notes, v21.03	575822
FlexRAN Reference Solution Software Release Notes, v18.09	
FlexRAN Reference Solution Software Release Notes, v1.6.0	
FlexRAN Reference Solution Software Release Notes, v1.5.0	
FlexRAN Reference Solution Framework API Reference (Doxygen)	572007
DPDK API Documents	http://doc.dpdk.org/api/

§

2.0 BBU Pooling Framework

The BBU pooling framework is a user-space middleware service that takes over Intel FlexRAN application tasks and schedules task executions on multi-core IA processors. It provides processing resources for tasks while minimizing scheduling overhead, cache miss, and processing latency

The main functionality of the BBU Pooling task framework is task dispatching and task management. The application divides processing into software tasks that run on IA CPUs and hardware tasks that run on accelerators and Field-Programmable Gate Arrays (FPGAs). The framework also provides core management and statistical reporting services to help an application achieve intelligent resource management.

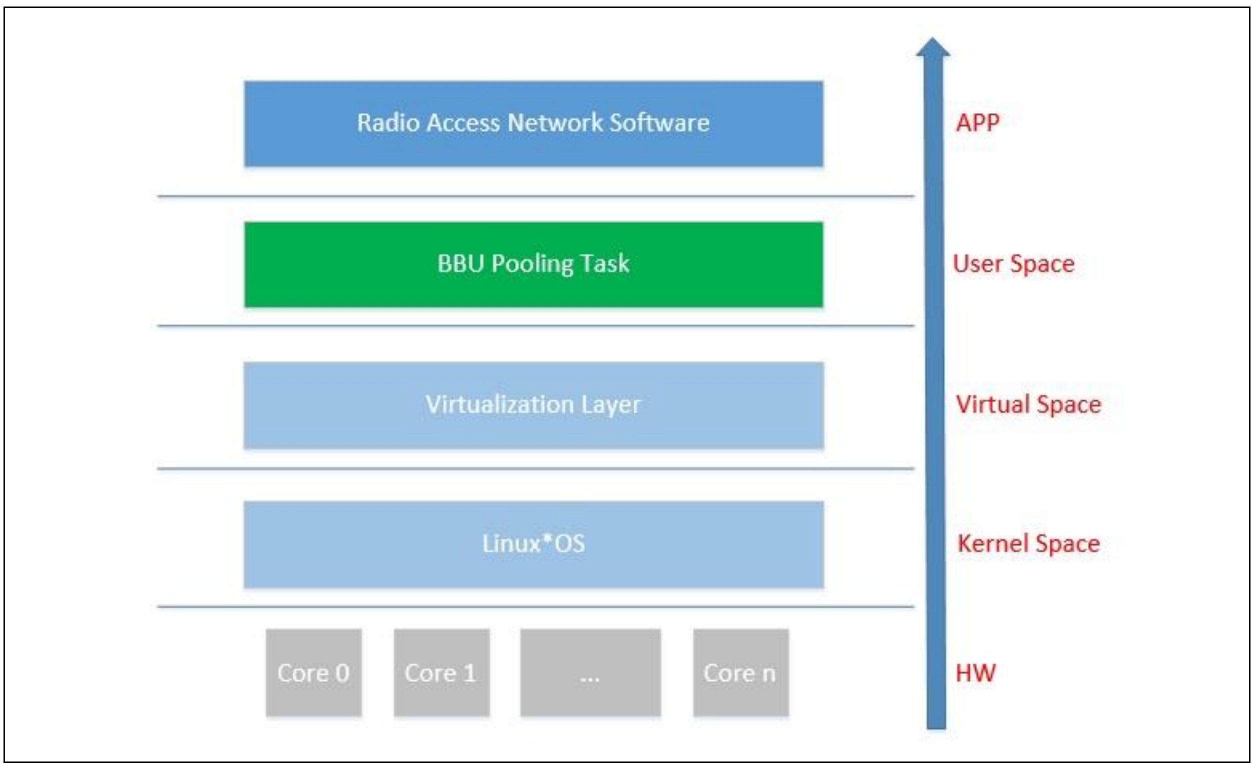
2.1 BBU Pooling Framework Components

The BBU Pooling framework includes:

BBU Pooling task: Addresses radio access network data unit (DU) intensive computing, signal processing characters, and provides a parallelism method and a resource scaling method based on flow graph.

The hierarchical structure diagram is illustrated in Figure 1.

Figure 1. BBU Pooling Framework Components and Hierarchy



2.2 System Architecture

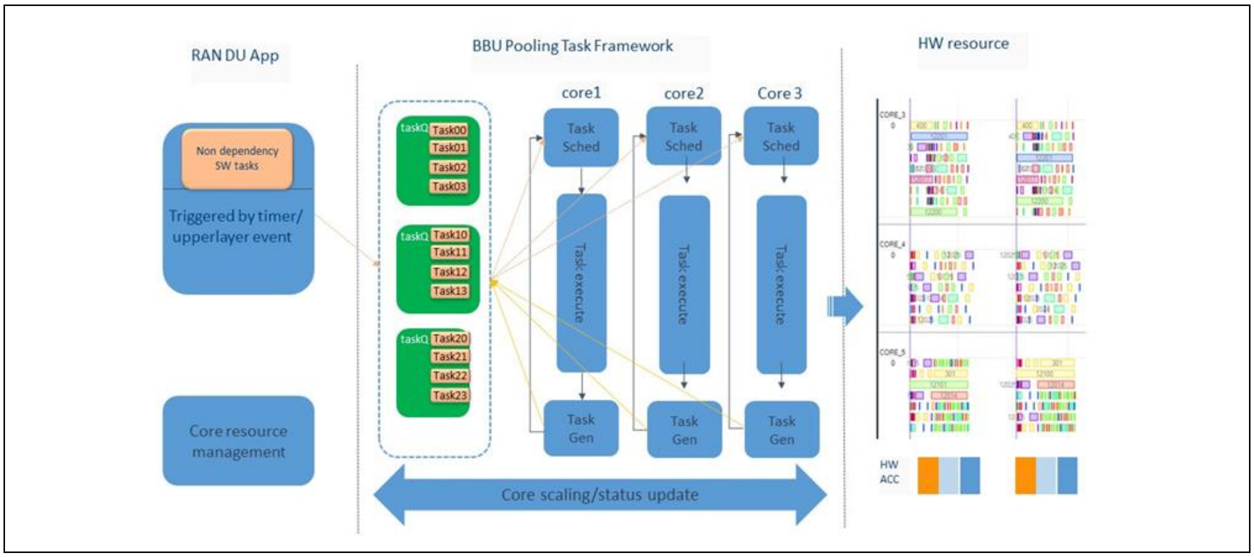
The BBU Pooling task framework is a distributed framework, in which each core does an equal share of task dispatching and task execution, automatically balancing the workload.

Figure 2 shows that each core runs three major tasks:

- **Schedule tasks**: query queue and dispatch tasks according to priorities.
- **Execute tasks**: execute tasks that were put on the core.
- **Generate tasks**: add a new task or dependent task to the queue.

Details of these functions are described in the following chapters.

Figure 2. BBU Pooling Task Architecture

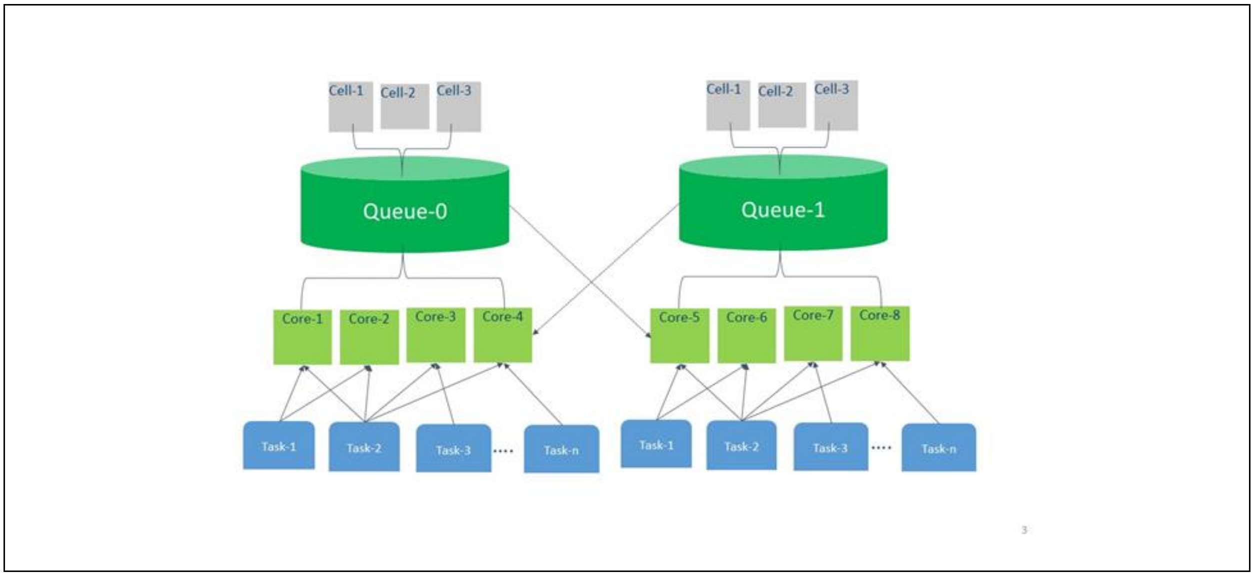


2.3 Overall Relational Model

One BBU pool can manage multiple `taskq`, as shown in Figure 2. Each taskq can bind multiple cores, and each cell must bind with one taskq. This hierarchical relational model was chosen to be compatible with Intel® architecture. For example, if hyper-threading is active, different logical cores belonging to the same physical core can be grouped into one queue to achieve better performance.

Cores belonging to the same socket can be grouped into one queue to avoid cache miss. It is also possible to use individual queues to separate the cell and core group and different quality of service (QoS) levels. The queue and core number and binding relationship are configured at the initialization stage.

Figure 3. BBU Pool Overall Relational Model



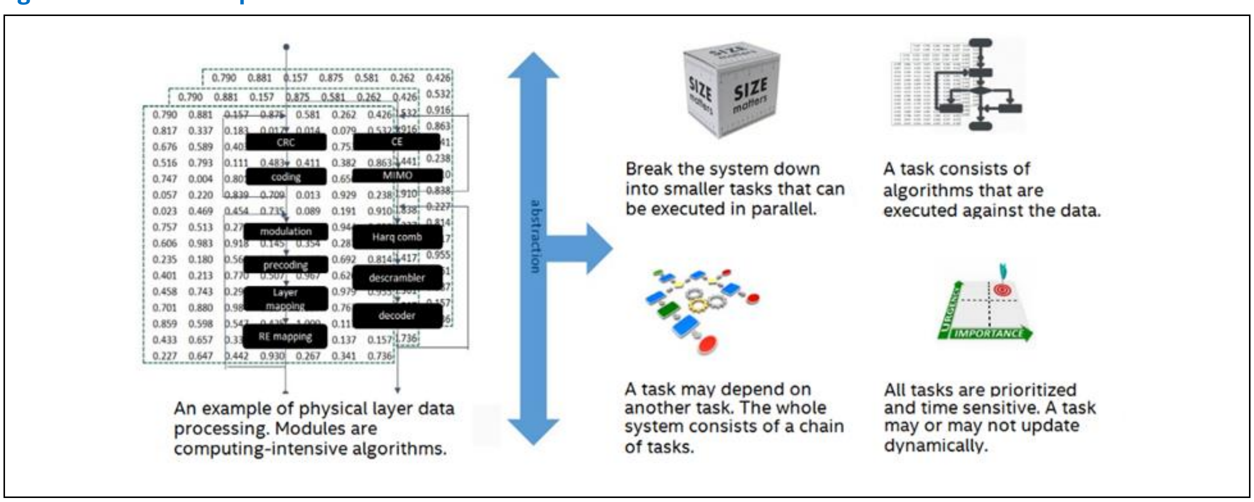
2.4 Task Property

The application breaks down the processing into tasks. Figure 4 shows an example of the physical layer signal processing. All the computing nodes such as Cyclic Redundancy Check (CRC), coding, and modulation are abstracted as tasks and happen at specified times in a dedicated period.

Each task has the following properties:

- It has a reasonable size and can be executed in parallel.
- It consists of the algorithms that are executed against the data.
- It might depend on another task; the whole system consists of a chain of tasks.
- It has a priority and is time-sensitive.

Figure 4. Task Properties



2.4.1 Task Function

Each task has a call-back function and a parameter set function. The call-back function should be registered at the initial stage.

2.4.2 Task Priority

Two-dimension priorities are used in the BBU Pooling Task Framework:

- **Priority time offset**: sets the priority by the time the task should be completed related to the current time (early deadline first).
- **Task priority**: uses the priority of the task itself (highest priority task first).

2.4.3 Task Deadline

Each task must have a task deadline configured for it. The framework uses this value to determine if any task misses the deadline. Any task where the execution time exceeds the expected processing time will be bypassed.

2.4.4 Task Dependency

The following three kinds of task dependency can be supported:

Type 1 (refer to Figure 5): At least one task of the same or different types can generate a single next task, which is dependent on all the previous tasks and is generated atomically after all of the previous tasks finish.

Type 2 (refer to Figure 6): One task can generate multiple tasks of different types.

Type 3 (refer to Figure 7): One task can generate multiple tasks of the same type through task splitting.

Figure 5. Task Dependency - Type 1

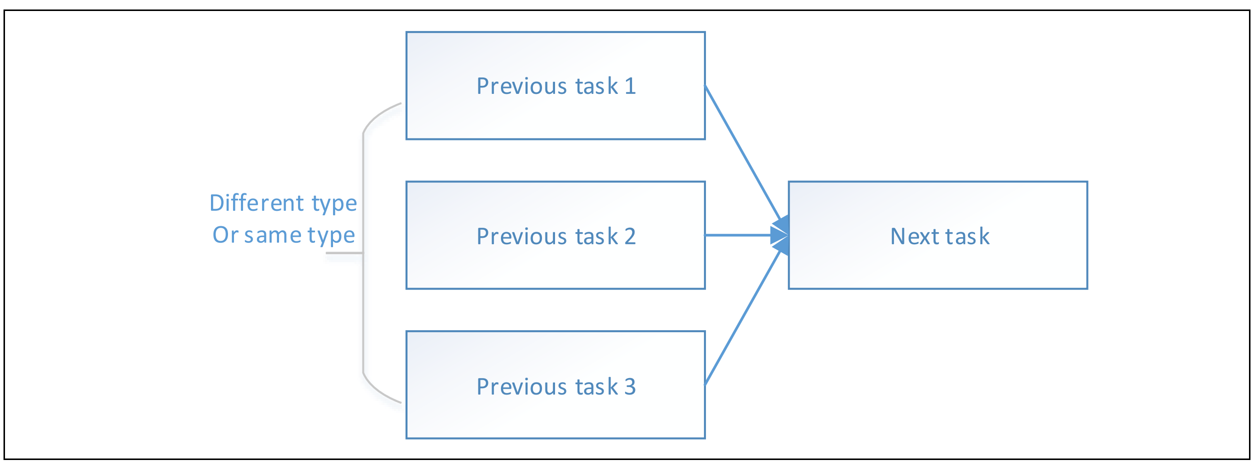


Figure 6. Task dependency - Type 2

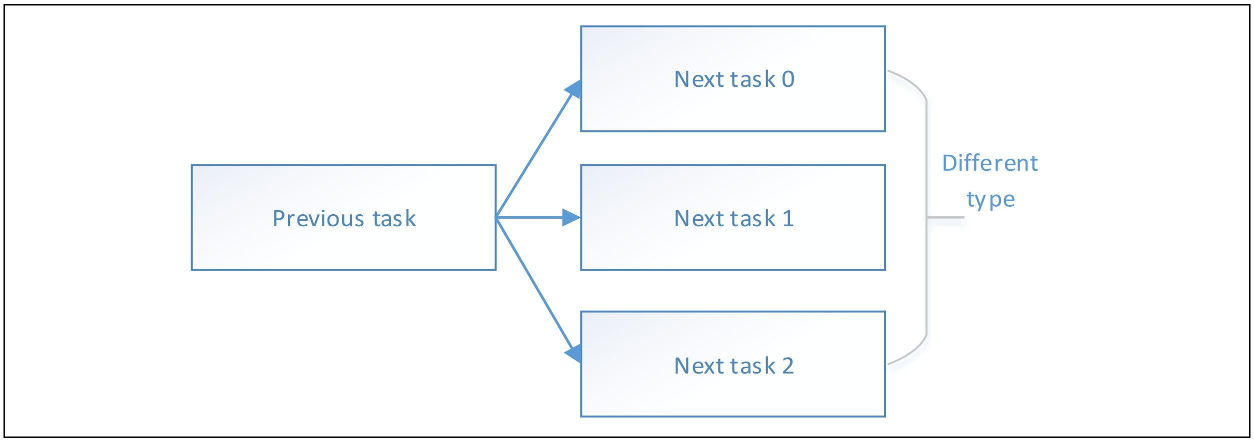
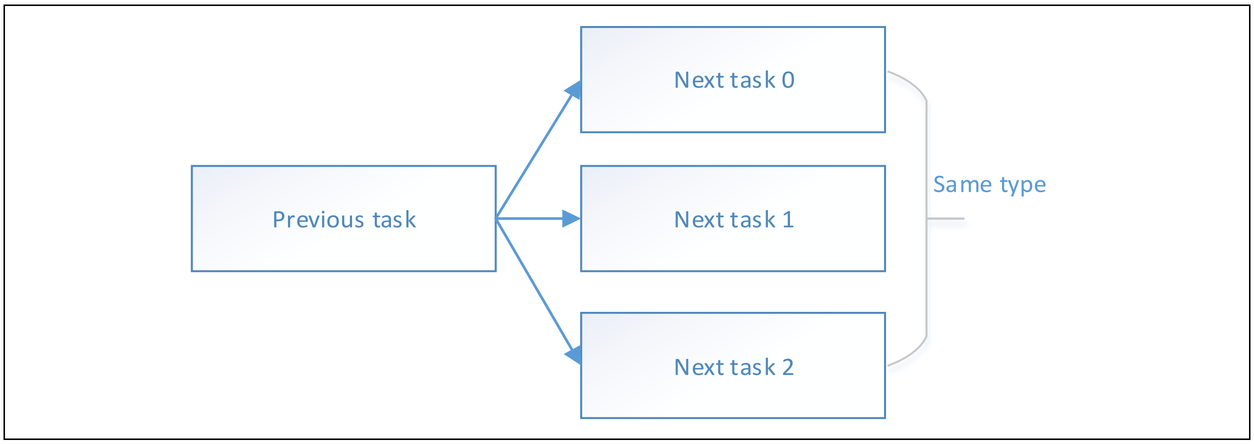


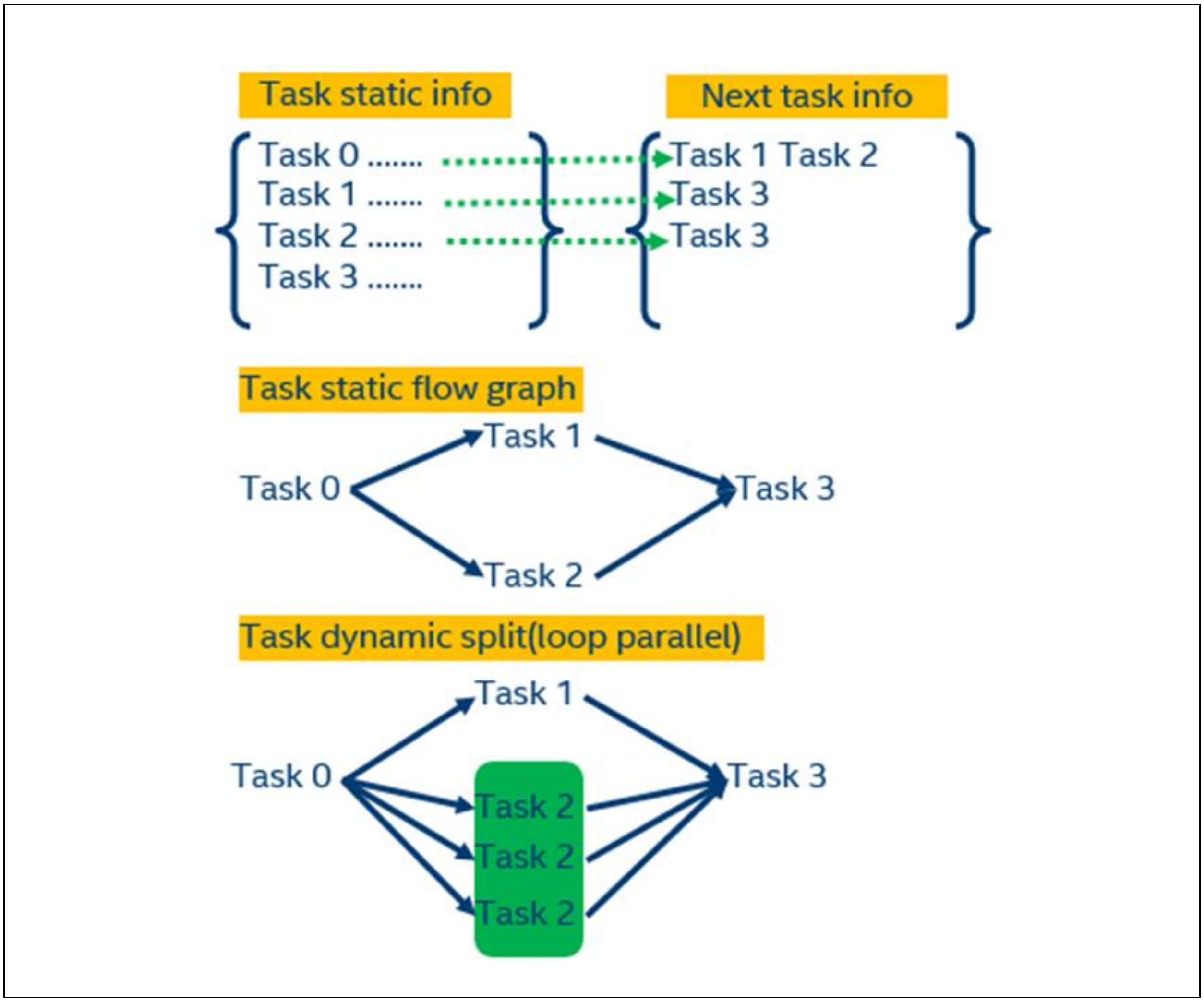
Figure 7. Task dependency - Type 3



2.4.5 Task Chain

According to the above task dependencies, the static task chain can be maintained in the Framework. For example, in Figure 8, there are four tasks. `Task1` and `Task2` will be generated after `Task0`, while `Task3` will be generated after both `Task1` and `Task2` are finished. In some scenarios, one task may need to be split into multiple tasks and run in parallel. In Figure 8, `Task2` was split into three tasks.

Figure 8. Task Chain



2.4.6 Task Affinity

One task may need to bind to one core to meet a specific requirement but, in general, the BBU Pooling Task Framework supports binding one task to one core or multiple cores.

2.5 Task Queue

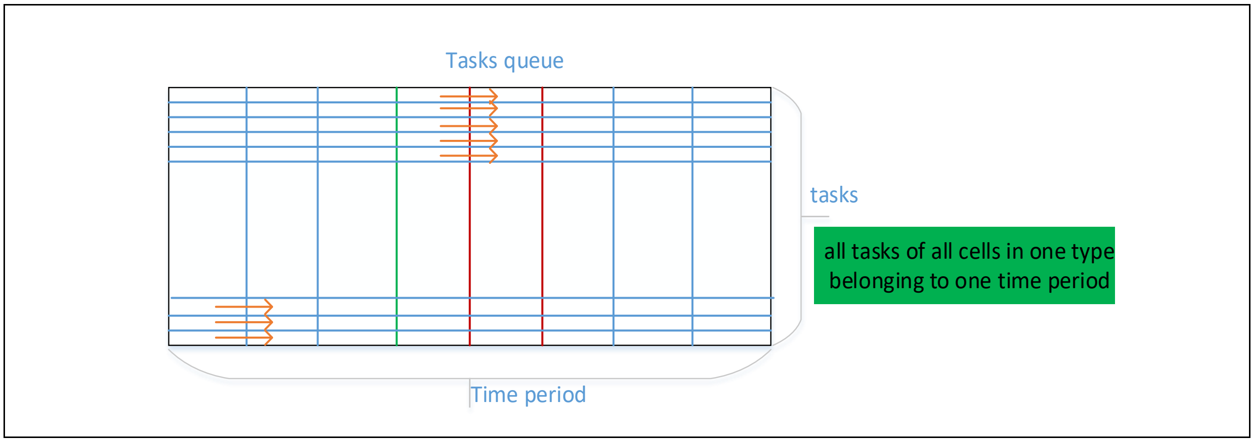
Tasks are put into task queues after they are ready to execute. Each task queue is divided into several task types.

Each task type is represented by a two-dimensional queue array:

- The first dimension is the time period (the current maximum is 40 periods)
- The second dimension is (maximum cell number) x (maximum task split number).

Figure 9 shows the queue array of task types.

Figure 9. Task Queue



2.6 Time Period

The application must provide a Framework with an updated time boundary and index so that BBU Pooling can perform a task frame deadline check and time-synchronized control for tasks.

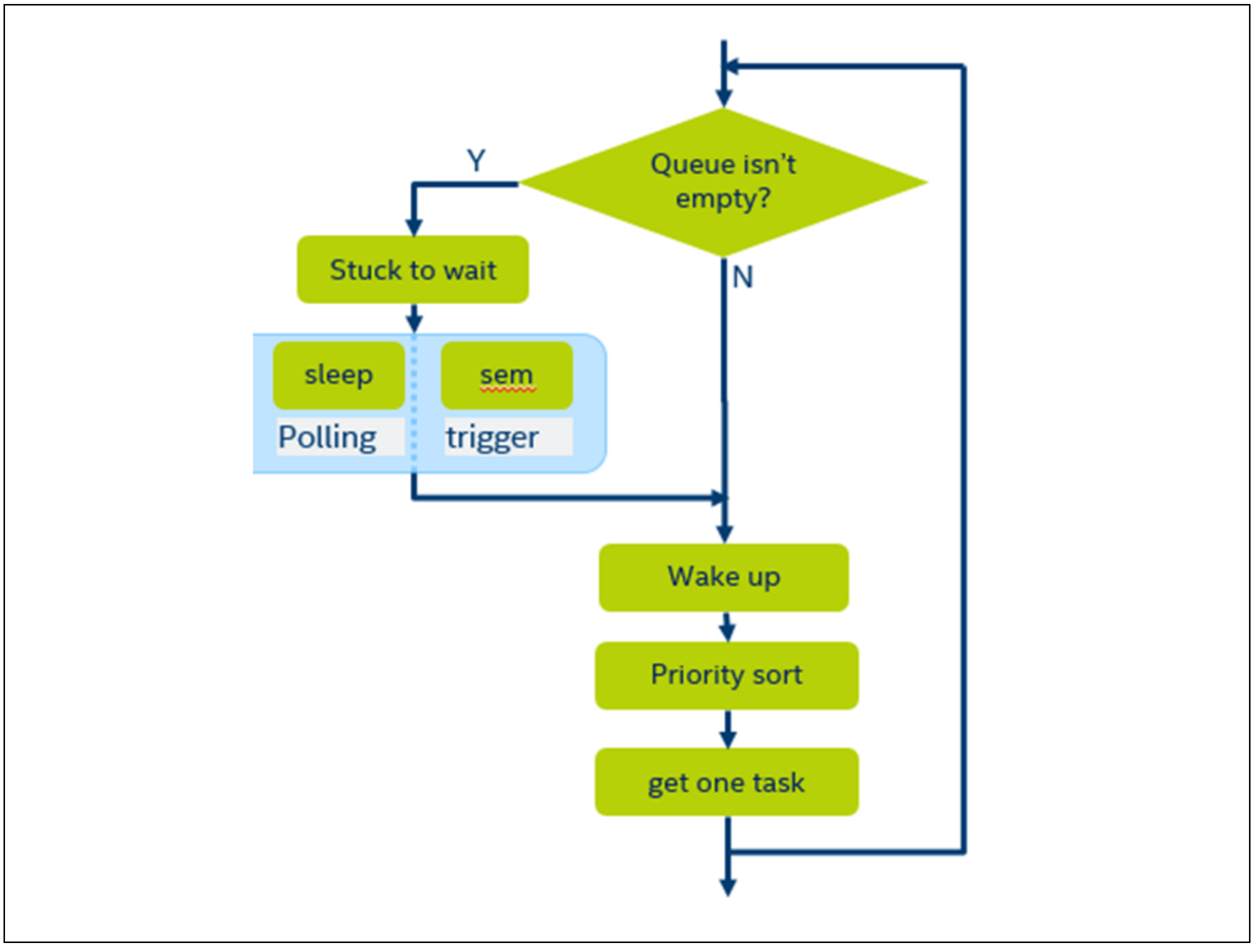
2.7 Task Scheduler

The task scheduler is in charge of task dispatching that can be implemented as a simple function call that can be used in distributed scheduler mode. All details in this chapter describe its use in distributed mode. (However, it is also possible to implement the scheduler as a thread to be used in centralized scheduler mode to implement complex scheduling strategy.)

2.7.1 Work Flow

Figure 10 illustrates the task scheduler workflow.

Figure 10. Scheduler Work Flow



2.7.2 Priority Sort

The priority for processing a task is calculated as:

First Priority = Time Period Index + Priority Time Offset

Because wireless systems are time-sensitive, the earlier time periods should be processed before later time periods. The time period number is the most important factor in setting a task's priority. Smaller values for priority time offset indicate higher-priority tasks.

Second Priority = Task Priority

After the tasks are chosen according to the first priority (urgent tasks), the second priority will determine the final task priority among the urgent tasks.

The tasks that are ready in the queue at a given time can be scheduled out. In some cases, the highest-priority task may have to wait because all cores were running when it arrived in the queue.

2.7.3 Get a Job

When the system gets a job from the highest priority task list, it must set the spinlock or atomic lock before changing the job status to avoid multi-core conflict. However, because the control context is smaller the effect of conflicts on system performance is not significant.

2.7.4 Synchronization Control

In wireless systems, previous subframes should be completely processed before the next subframe begins. In unbalanced situations when a subframe with a relatively heavy workload is followed by a subframe with a lighter workload, task chains from the later subframe may be received before the previous subframe is completely processed. Synchronization mode guarantees that task chains from later subframes are postponed and not processed until corresponding task chains from earlier subframes are entirely processed.

The following examples explain the difference between synchronization and non-synchronization modes. Both examples use uplink Physical Uplink Shared Channel (PUSCH) channel processing in a workload non-balance scenario where:

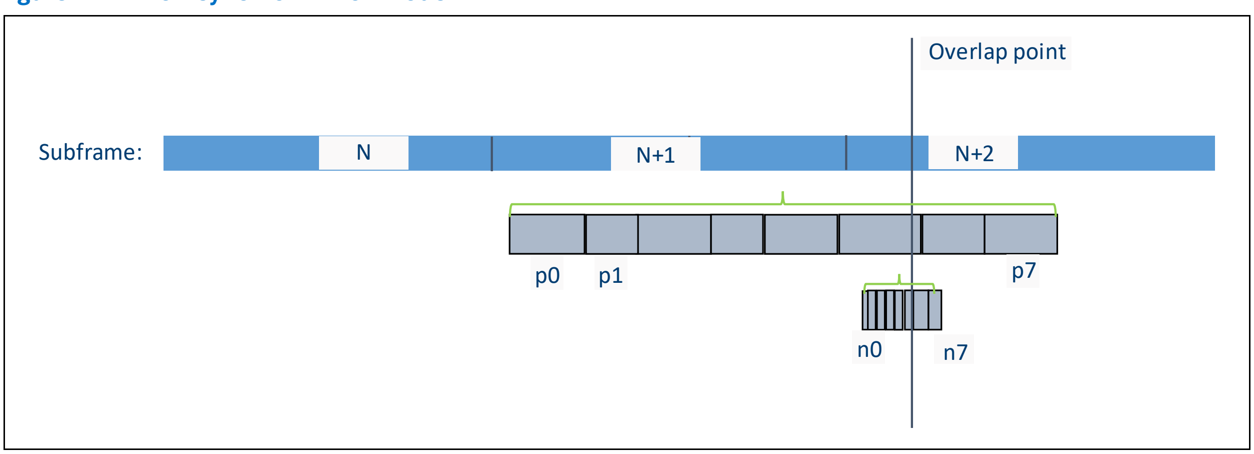
- N is a subframe with a heavy workload containing task chains p0-p7.
- N+1 is a subframe with a light workload containing task chains n0-n7.
- Both task chain p(0...7) in subframe N and corresponding task chain n(0..7) in subframe N+1 have the same task type.
- Task chains belonging to N are processed more slowly than those contained in N+1.
- Sometimes, the task processing times for successive subframes will overlap.

Non- synchronization mode

In Figure 11, task chains have arrived in the order n6, p6, n7, p7. “First-in, first-out” execution using a single queue would run tasks n6 and n7 before p6 and p7. However, p6 and p7 belong to a previous subframe and will expire sooner than n6 and n7 because of the hybrid automatic repeat request (HARQ) time limitation.

In non-synchronization mode, buffers have to be set aside for tasks n6 and n7 so that tasks p6 and p7 can be processed first or parallel with n6 and n7.

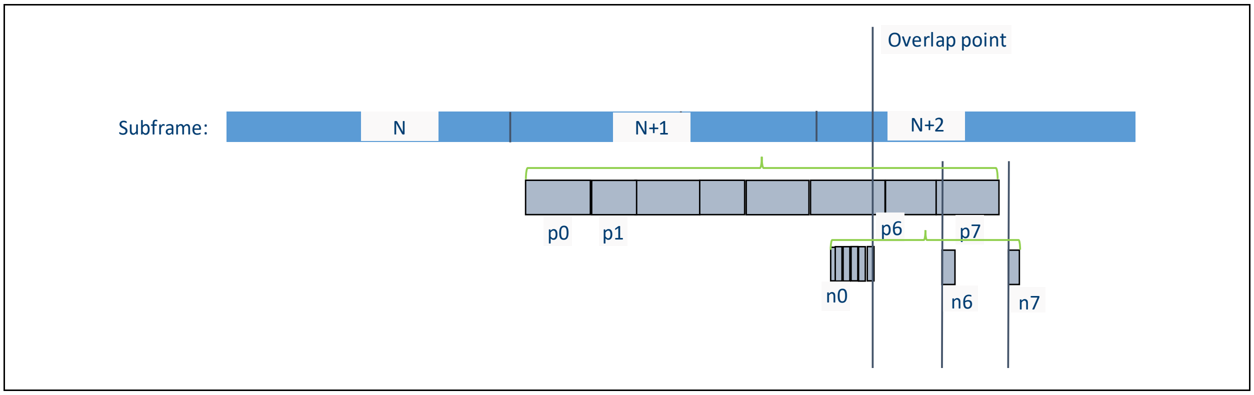
Figure 11. Non-synchronization Mode



Synchronization mode

In Figure 12, task n6 is postponed until processing for all of p6 is complete, and n7 is likewise postponed until p7 processing is complete. Latency is increased, but there is no need for extending buffers for each task.

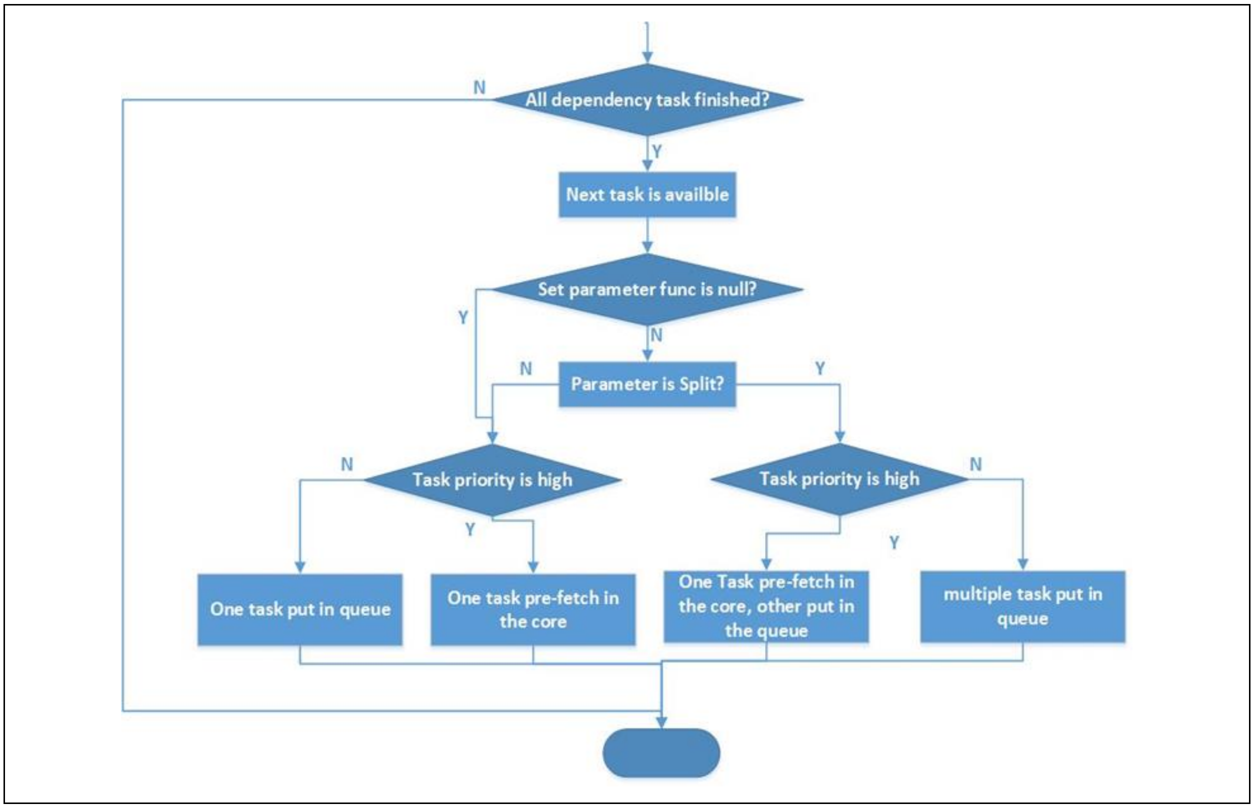
Figure 12. Synchronization Mode



2.8 Task Generation

Tasks are generated according to dependency and put into the proper queue. There are two ways to get tasks into the queue and manage them through the BBU pooling task framework.

Figure 13. Generation Work Flow



2.8.1 Application Trigger

If a task is the first task of a task chain, or if it is not dependent on other tasks, it can be triggered only through the FlexRAN BBU Pooling API by the application. The parameters of this task should be set before generation and passed into BBU pooling.

2.8.2 Auto-generation

If a task is dependent on other tasks, and the dependency is registered, then this task will be generated, scheduled, and run automatically by Framework. The parameters of this task must be previously set through the pre-registered parameter set call-back function.

2.8.3 Task Pre-fetch

To reduce the cache miss and scheduler overhead, one task will be pre-fetched in the core and run immediately if the task's priority is high.

2.8.4 Task Split

When the workload of one task is substantial, the application can split the task at the task generation stage. The pre-handler function must be configured in advance so the system can pass the split valid flag, split number and parameters for each split function before the BBU pooling framework schedules each part and runs them in parallel. After all parts of a split task are executed, the next task will be generated as usual.

2.9 Task Execution

When the scheduler assigns the highest priority task to a core, the task will be run on the core after the system estimates processing time and checks for a bypass flag.

If the task bypass flag is set, the handler function of the task will not be run. The next task will be generated in sequence to follow the standard scheduler policy, but the bypass flag will be inherited by the next task.

2.10 Hardware Tasks

The BBU Pooling task framework supports hardware tasks that are performed on acceleration hardware and/or FPGA. It attempts to manage tasks in the same way as software tasks and reduce the potential additional pooling core for the FPGA/accelerator.

The working flow is as below:

- Software task completes.
- Task Generator enqueuers hardware Task for scheduler.
- Scheduler dequeuers hardware Task and executes.
- Hardware task is enqueued to FPGA/Accelerator.
- Scheduler dequeues/Poll hardware task from Accelerator HW queue.

- Task Generator enqueues software task for the scheduler.

2.10.1 Working Mode

Two modes are supported:

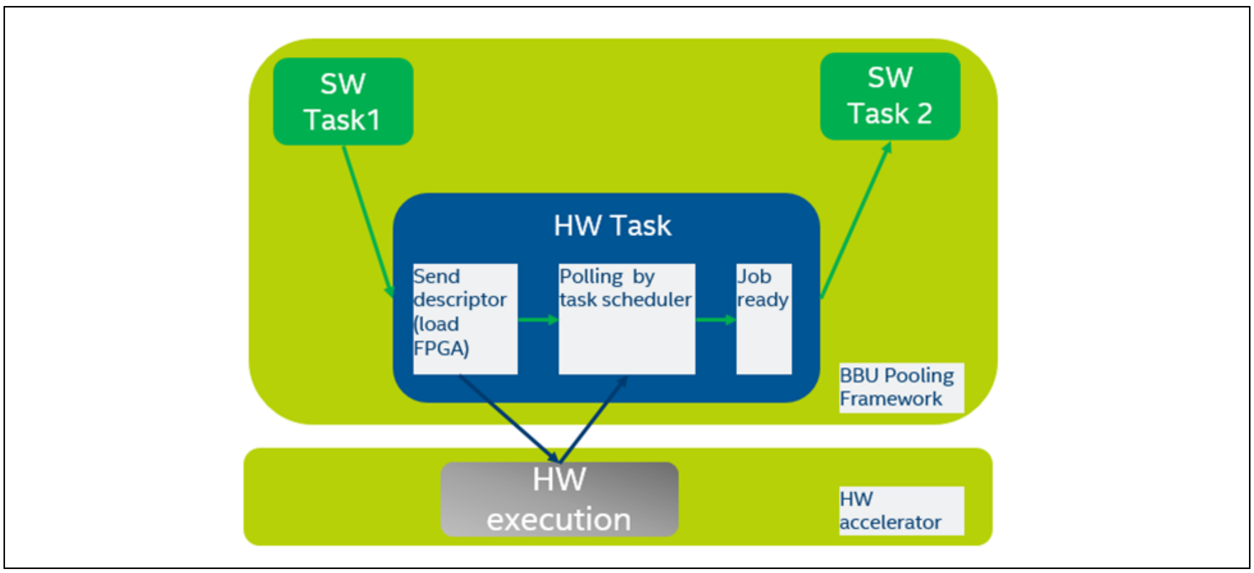
- Pooling mode
- Trigger mode.

In pooling mode:

- If a task is sent to acceleration hardware or the FPGA, it is registered as a hardware task.
- The relationship and priority with other tasks are maintained by Framework.
- Sending a task description (loading FPGA) and a polling function should be registered into Framework.
- The task scheduler does polling when the core is available (distributed polling).

Refer to Figure 14.

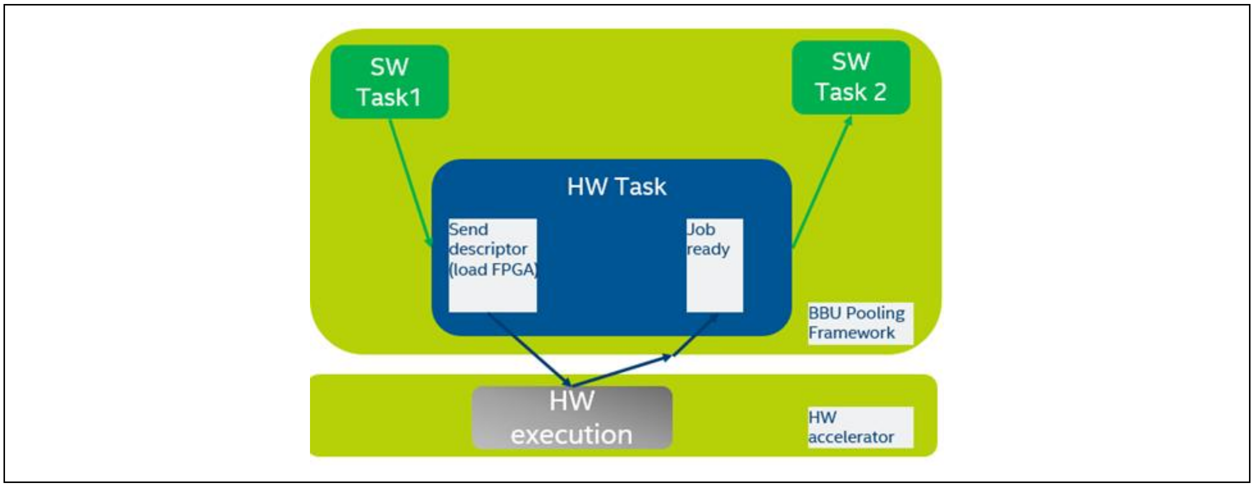
Figure 14. HW Pooling Mode



In trigger mode:

Trigger mode is the same as pooling mode, except the application does polling or uses interrupt mode. It informs Framework by calling the call-back function provided by Framework after hardware execution is ready. Refer to Figure 15.

Figure 15. HW Trigger Mode



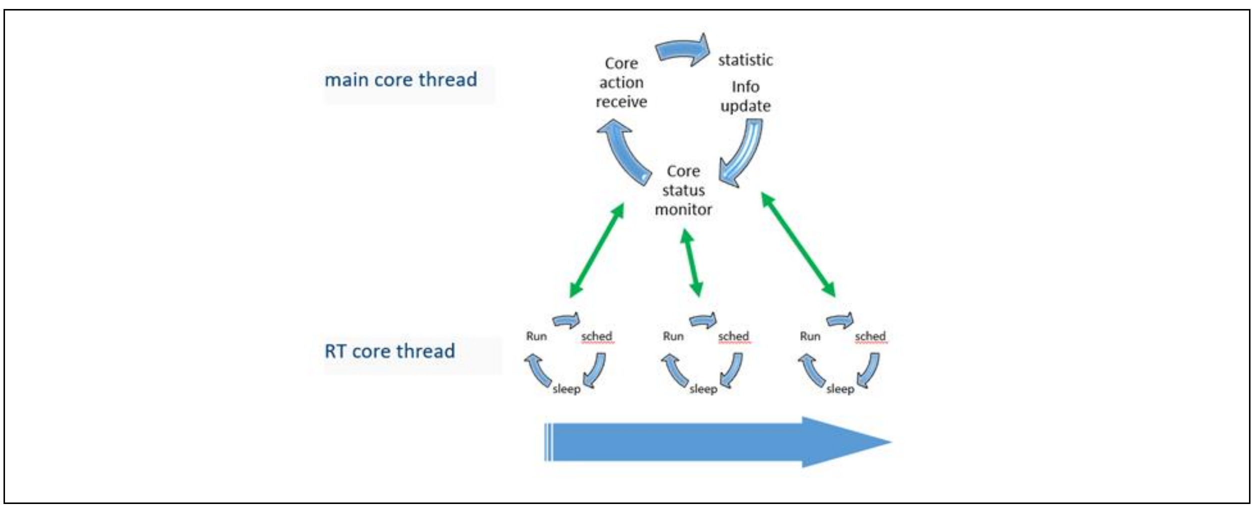
2.11 Core Management

The main control thread was introduced into BBU Pooling Framework to support core action commands. The thread is in charge of non-real-time action, and it can be assigned to the OS core.

Core management commands are:

- Core Suspend (pending): Puts a core to sleep.
- Core Add: Adds a core to bbupool.
- Core Remove: Removes a core from bbupool. Fails when task affinity assigns a specific task to only this core and no other.
- Core Resume: Resume a suspended core.

Figure 16. Core Management



2.12 Status Report

The BBU Pooling task framework provides the following status reports for the application:

- Core CPU workload
- Core mask of task execution
- Maximum, minimum, and average task execution time
- Average task latency time.

2.13 Performance

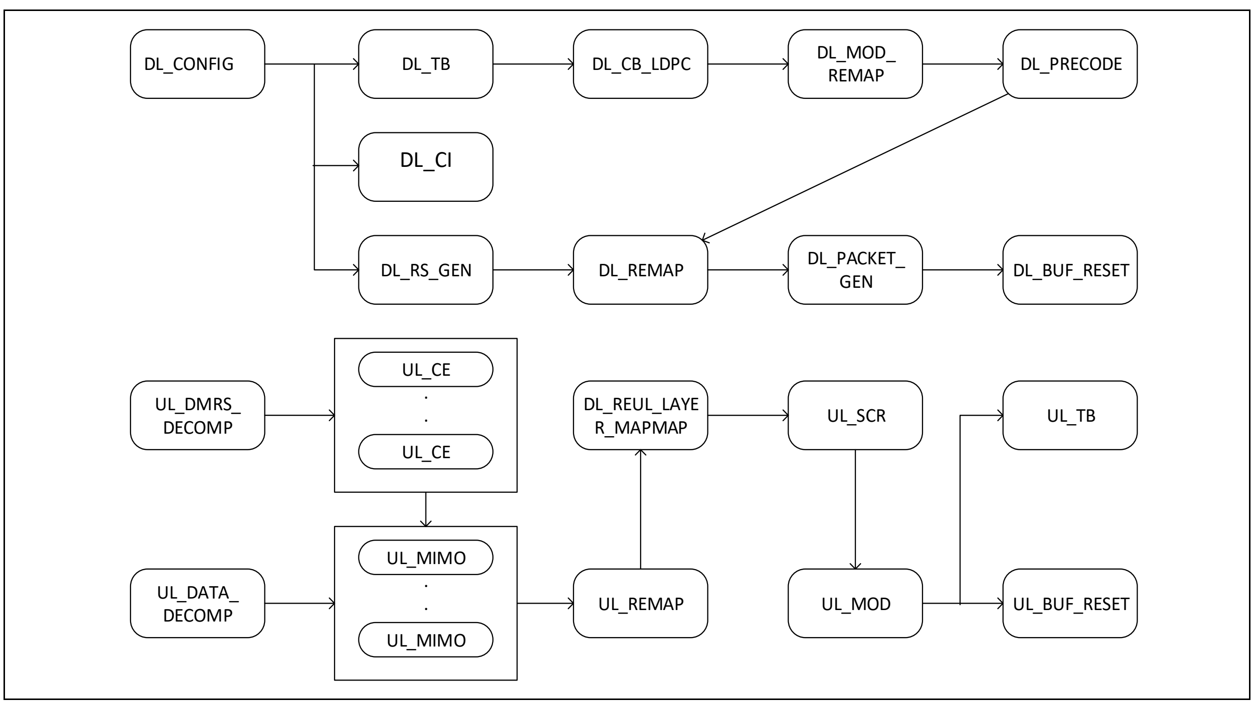
This chapter describes the processing flow and unit system test of the BBU Pooling Task Framework and compares the performance in different system configurations and unit test scenarios. Differences in hardware, software, application or configuration will affect actual performance.

2.13.1 Test System

Figure 17 displays the processing flow used for the performance test. All the modules in this figure were simulations (not functions defined in 3 GPP).

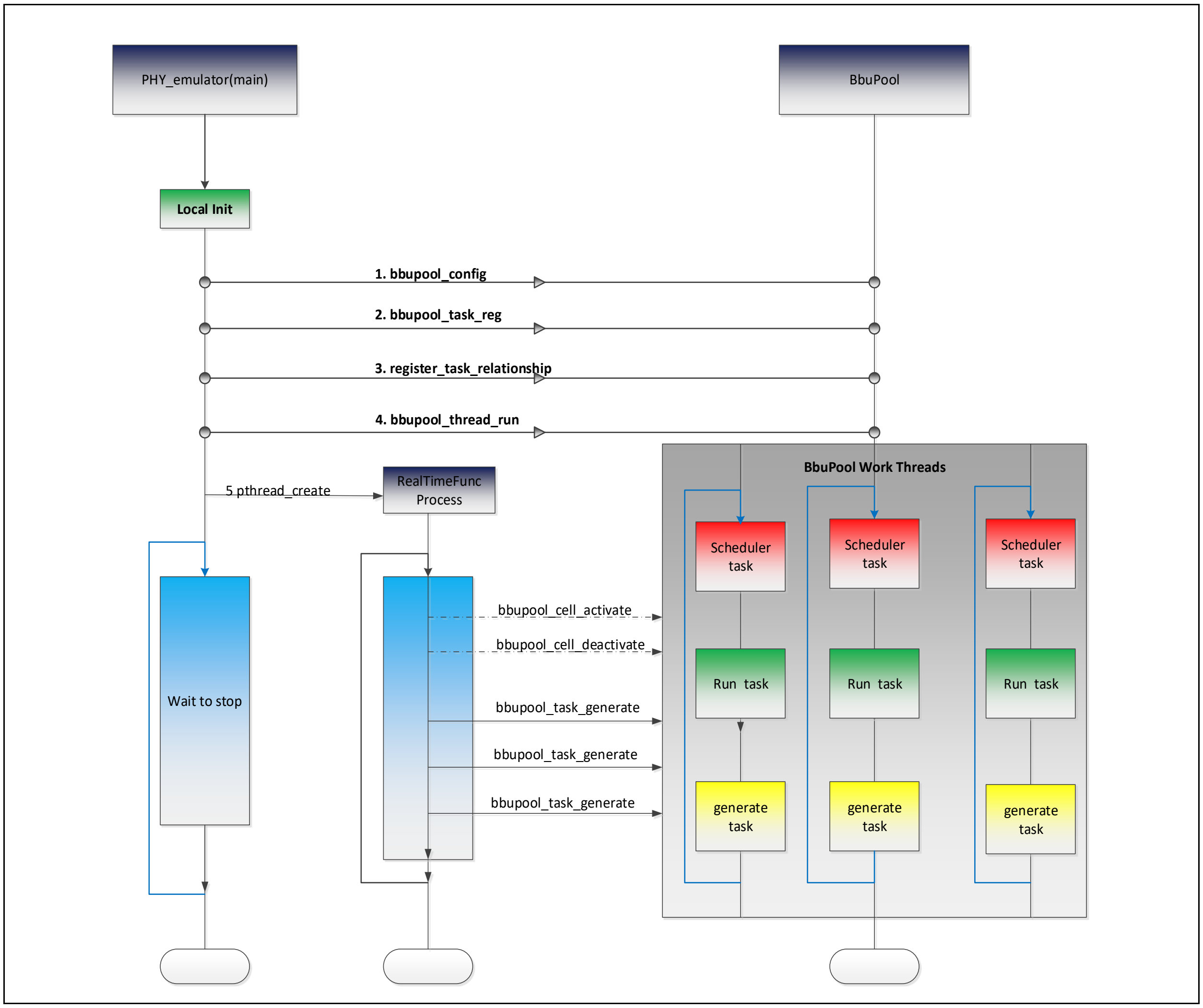
The test tried to simulate the possible dependency flow and measure performance under a changing workload.

Figure 17. Processing Flow



As Figure 18 describes, the test system calls BBU pooling libraries to initialize BBU pooling, create cells, delete cells, and trigger tasks.

Figure 18. Test System



2.13.2 Test Scenario

The performance test measured the impact of changing:

- Number of queues
- Number of cells
- The workload for each task
- Task affinity
- Task split.

The following cases were selected to present the performance with TTI length 125µs, which means all the above tasks in Figure 17 run out in 125µs.

Table 3. Performance with TTI Length

No	Cases	Cell Number	Core Number	Task Queue Number	Task Affinity
0	h125_case1_1cell_1core_1queue_noAff	1	1	1	N
1	h125_case8_2cell_6core_2queue_AffEveryone	2	6	2	Y
2	h125_case7_6cell_6core_2queue_noAff	6	6	2	N
3	l125_case7_6cell_6core_2queue_noAff	6	6	2	N

NOTE: The CPU of the server used in the test is an Intel® Xeon® Platinum 8160 CPU @ 2.10GHz with hyperthreading off.

2.13.3 Performance

BBU Pooling performance can be described through four factors:

- `Max(Kcycles)`: the max scheduler cost for a task to complete its function
- `Min(Kcycles)`: the min scheduler cost for a task to complete its function
- `Ave(Kcycles)`: the average scheduler cost for a task to complete its function
- `aveDelay(k)`: the average time between a task creation generation and running.

The columns in the screen images in this section describe:

- Column 1 - the task name
- Columns 2-4 – statistics for the execution cost for the task
- Columns 5 –the average time between task generation and running.
- Column 6 – the number of times each task has been run
- Column 7 – statistics for task bypass counter
- Column 8 – the queue for the case running
- Column 9 – list of cores running each task

There is overall overhead information of generation, scheduler, and core usage, overhead unit is “k” cycles, and core usage is a percentage.

Case 0 performance[1]

One cell is running on a `BBUPOOL` with only one core.

Figure 19 shows the simplest case of a `BBUPOOL` configuration. From the following screenshot below, we can see the scheduler overhead is 0.32 k cycles, generation overhead is 0.39 k cycles, which are both much lower than `case1` because there are only one cell and one core.

Figure 19. Case 0 performance

```
cellIdx #0 Timing report
========================
  Task Name        Max(k)    Min(k)    Ave(k)    AveD(k)    Counter   Bypass      queue    core
                                                                     (1st,2nd)
--------------   --------  --------  --------  --------   --------  --------    -------  -------
     DL_CONFIG      4.94      3.29      3.89      1.45      cff6     (0,0)        0       (1)
UL_DMRS_DECOMP      6.26      4.79      5.21     25.09      cff6     (0,0)        0       (1)
UL_DATA_DECOMP     19.82     18.19     18.80     34.68      cff6     (0,0)        0       (1)
         DL_TB      4.19      3.28      3.32      0.93      cff6     (0,0)        0       (1)
        DL_DCI      4.78      4.58      4.64     50.10      cff6     (0,0)        0       (1)
     DL_RS_GEN     22.86     17.91     18.08     55.26      cff6     (0,0)        0       (1)
         UL_CE      4.06      3.27      3.30     71.79     33fd8     (0,0)        0       (1)
       UL_MIMO     21.57      3.27      3.29      6.30     33fd8     (0,0)        0       (1)
    DL_CB_LDPC     10.42      4.62      4.70      0.32      cff6     (0,0)        0       (1)
      DL_REMAP     18.21     17.94     18.07      0.33      cff6     (0,0)        0       (1)
      UL_REMAP      3.37      3.28      3.31      0.32      cff6     (0,0)        0       (1)
  DL_MOD_REMAP      4.72      4.57      4.62      0.31      cff6     (0,0)        0       (1)
 DL_PACKET_GEN     23.08     17.91     18.05      0.32      cff6     (0,0)        0       (1)
        UL_MOD      6.05      3.28      3.32      0.31      cff6     (0,0)        0       (1)
        UL_SCR      4.79      4.62      4.69      0.31      cff6     (0,0)        0       (1)
  UL_LAYER_MAP     18.23     17.95     18.11      0.31      cff6     (0,0)        0       (1)
    DL_PRECODE      3.38      3.29      3.32      0.31      cff6     (0,0)        0       (1)
  DL_BUF_RESET      4.77      4.62      4.66      0.32      cff6     (0,0)        0       (1)
  UL_BUF_RESET     18.25     17.96     18.11      5.89      cff6     (0,0)        0       (1)
         UL_TB      5.51      4.60      4.66      0.55      cff6     (0,0)        0       (1)

bbupool schedule task average overhead:0.319000 k cycles,count = 151efc
bbupool generate task average overhead:0.394000 k cycles,count = 103f38

Bbupool: Core ID        Core Usage
        ----------     -----------
         core1            88
```

[1]*Software and workloads used in performance tests may have been optimized for performance only on Intel microprocessors.*

Performance tests, such as SYSmark and MobileMark, are measured using specific computer systems, components, software, operations and functions. Any change to any of those factors may cause the results to vary. You should consult other information and performance tests to assist you in fully evaluating your contemplated purchases, including the performance of that product when combined with other products. For more complete information visit www.intel.com/benchmarks.

Test and System Configurations: The performance test is a unit test based on the CPU Intel® Xeon® Platinum 8160 CPU @ 2.10GHz. Differences in hardware, software, application or configuration will affect actual performance.
Performance results are based on testing as of September 29, 2018 and may not reflect all publicly available security updates. See configuration disclosure for details. No component or product can be absolutely secure.

Case 1 performance

Two cells are running on a `BBUPOOL` with 6 cores.

This is a sufficient resource configuration. Every task is bound to 2 cores, only show the running core in Figure 20.

From the following screenshot, notice the scheduler overhead is 0.64 k cycles, generation overhead is 0.66 k cycles, which are both much higher than `case0`, resulting from the fact of task scheduling lock when multi-cores are running in parallel.

Figure 20. Case 1 Performance

```
cellIdx #0 Timing report
========================
  Task Name        Max(k)    Min(k)    Ave(k)    AveD(k)    Counter   Bypass      queue    core
                                                                     (1st,2nd)
--------------   --------  --------  --------  --------   --------  ---------   -------  -------
     DL_CONFIG      22.39      9.61     13.94      1.61      aff6     (0,0)        0      (1)
UL_DMRS_DECOMP      23.17     13.35     17.29      1.68      aff6     (0,0)        0      (3)
UL_DATA_DECOMP      56.41     51.18     52.41      1.60      aff6     (0,0)        0      (5)
         DL_TB      14.47      9.14     10.14      1.78      aff6     (0,0)        0      (1)
        DL_DCI      14.98     12.99     13.25     41.17      aff6     (0,0)        0      (5)
     DL_RS_GEN      55.43     50.46     51.00     14.88      aff6     (0,0)        0      (3)
         UL_CE      14.21      9.14      9.23     55.90     2bfd8     (0,0)        0      (3)
       UL_MIMO      15.39      9.15      9.87     18.18     2bfd8     (0,0)        0      (5)
     DL_CB_LDPC     13.90     12.84     13.09      0.59      aff6     (0,0)        0      (1)
      DL_REMAP      51.52     50.42     50.80      1.10      aff6     (0,0)        0      (1)
      UL_REMAP      10.49      9.15      9.43      1.17      aff6     (0,0)        0      (3)
  DL_MOD_REMAP      13.85     13.00     13.25     29.66      aff6     (0,0)        0      (5)
 DL_PACKET_GEN      50.95     50.24     50.51      0.78      aff6     (0,0)        0      (1)
        UL_MOD      15.40      9.35     13.78      1.48      aff6     (0,0)        0      (5)
        UL_SCR      19.48     12.87     13.00      1.01      aff6     (0,0)        0      (3)
  UL_LAYER_MAP      51.30     50.35     50.66      0.83      aff6     (0,0)        0      (3)
    DL_PRECODE      13.36      9.14      9.16      0.83      aff6     (0,0)        0      (5)
  DL_BUF_RESET      13.86     12.84     12.96      0.74      aff6     (0,0)        0      (1)
  UL_BUF_RESET      70.02     50.50     50.98      1.35      aff6     (0,0)        0      (1)
         UL_TB      16.05     12.89     13.99      1.30      aff6     (0,0)        0      (3)

bbupool schedule task average overhead:0.642000 k cycles,count = 11defc
bbupool generate task average overhead:0.662000 k cycles,count = dbf38

Bbupool: Core ID          Core Usage
        ----------        -----------
         core1              79
         core2              78
         core3              73
         core4              68
         core5              67
         core6              67
```

Case 2 Performance

Six cells are running on a `BBUPOOL` with 6 cores and heavy load.

This is the usual configuration of the test scenario. Figure 21 screenshot contains only one cell data. This cell is bond to a queue with cores 1, 3, and 5 (each queue has three cores in the test settings).

Figure 21 screenshot, the CPU usage is much higher than `case3` because the `case2's` load is heavy.

Figure 21. Case 2 performance

```
cellIdx #0 Timing report
========================
  Task Name        Max(k)     Min(k)     Ave(k)     AveD(k)     Counter    Bypass      queue      core
                                                                           (1st,2nd)
--------------    --------   --------   ---------  --------    --------   --------    -------    -------
     DL_CONFIG       14.04       4.42       5.57      13.13       8ff6     (0,0)          0      (1,3,5)
UL_DMRS_DECOMP       17.61       6.07       8.91      11.65       8ff6     (0,0)          0      (1,3,5)
UL_DATA_DECOMP       35.72      23.23      25.30      65.15       8ff6     (0,0)          0      (1,3,5)
         DL_TB       16.91       4.08       5.34       3.78       8ff6     (0,0)          0      (1,3,5)
        DL_DCI       16.32       5.73       6.88      82.35       8ff6     (0,0)          0      (1,3,5)
     DL_RS_GEN       33.69      22.65      24.67      86.27       8ff6     (0,0)          0      (1,3,5)
         UL_CE       19.49       4.09       6.30     128.37      23fd8     (0,0)          0      (1,3,5)
       UL_MIMO       22.42       4.08       7.76      13.78      23fd8     (0,0)          0      (1,3,5)
    DL_CB_LDPC       16.33       5.74       6.12       0.56       8ff6     (0,0)          0      (1,3,5)
      DL_REMAP       34.20      22.50      24.13       0.60       8ff6     (0,0)          0      (1,3,5)
      UL_REMAP       13.20       4.09       4.45       0.62       8ff6     (0,0)          0      (1,3,5)
  DL_MOD_REMAP       16.46       5.72       6.21       0.57       8ff6     (0,0)          0      (1,3,5)
 DL_PACKET_GEN       35.17      22.45      23.64       0.59       8ff6     (0,0)          0      (1,3,5)
        UL_MOD       14.27       4.09       4.32       0.61       8ff6     (0,0)          0      (1,3,5)
        UL_SCR       15.68       5.75       6.17       0.64       8ff6     (0,0)          0      (1,3,5)
  UL_LAYER_MAP       44.40      22.44      23.34       0.60       8ff6     (0,0)          0      (1,3,5)
    DL_PRECODE       13.85       4.11       4.67       0.56       8ff6     (0,0)          0      (1,3,5)
  DL_BUF_RESET       15.48       5.73       6.65       0.64       8ff6     (0,0)          0      (1,3,5)
  UL_BUF_RESET       33.14      22.51      23.40       7.87       8ff6     (0,0)          0      (1,3,5)
         UL_TB       18.43       5.72       6.53       1.30       8ff6     (0,0)          0      (1,3,5)

bbupool schedule task average overhead:0.492000 k cycles,count = e9efc
bbupool generate task average overhead:0.818000 k cycles,count = b3f38

Bbupool: Core ID        Core Usage
         ----------     -----------
         core1             72
         core2             73
         core3             71
         core4             73
         core5             71
         core6             73
```

Case 3 Performance

Six cells are running on a `BBUPOOL` with 6 cores and light load.

This is the usual configuration of the test scenario. Figure 22 contains only one cell data. This cell are bond to a queue with cores 1, 3, and 5 (each queue has three cores in the test settings).

From the following screenshot, notice the CPU usage is much lower than `case2` because the `case3's` load is light.

Figure 22. Case 3 performance

```
cellIdx #0 Timing report
========================
  Task Name        Max(k)    Min(k)    Ave(k)    AveD(k)    Counter   Bypass      queue     core
                                                                     (1st,2nd)
--------------    --------  --------  --------  --------   --------  ---------   -------   -------
     DL_CONFIG       7.08      1.09      1.60     12.60      bff6    (0,0)         0       (1,3,5)
UL_DMRS_DECOMP       8.87      1.69      2.20     11.29      bff6    (0,0)         0       (1,3,5)
UL_DATA_DECOMP      13.36      6.30      7.49     34.32      bff6    (0,0)         0       (1,3,5)
         DL_TB       8.11      1.05      1.39      5.00      bff6    (0,0)         0       (1,3,5)
        DL_DCI       8.25      1.48      2.23     37.82      bff6    (0,0)         0       (1,3,5)
     DL_RS_GEN      14.65      5.76      6.47     38.47      bff6    (0,0)         0       (1,3,5)
         UL_CE       9.34      1.05      1.47     46.58     2ffd8    (0,0)         0       (1,3,5)
       UL_MIMO       7.10      1.05      1.36      5.78     2ffd8    (0,0)         0       (1,3,5)
     DL_CB_LDPC      7.33      1.48      1.67      0.46      bff6    (0,0)         0       (1,3,5)
      DL_REMAP      11.47      5.73      6.11      0.66      bff6    (0,0)         0       (1,3,5)
      UL_REMAP       6.66      1.05      1.14      0.71      bff6    (0,0)         0       (1,3,5)
  DL_MOD_REMAP       7.38      1.46      1.71      0.53      bff6    (0,0)         0       (1,3,5)
 DL_PACKET_GEN      11.02      5.70      6.04      0.65      bff6    (0,0)         0       (1,3,5)
        UL_MOD       6.83      1.06      1.15      0.69      bff6    (0,0)         0       (1,3,5)
        UL_SCR       6.73      1.48      1.62      0.70      bff6    (0,0)         0       (1,3,5)
  UL_LAYER_MAP      11.57      5.73      6.04      0.68      bff6    (0,0)         0       (1,3,5)
    DL_PRECODE       7.29      1.07      1.19      0.52      bff6    (0,0)         0       (1,3,5)
  DL_BUF_RESET       7.00      1.46      1.61      0.66      bff6    (0,0)         0       (1,3,5)
  UL_BUF_RESET      11.75      5.72      6.02      3.90      bff6    (0,0)         0       (1,3,5)
         UL_TB       6.88      1.46      1.59      1.28      bff6    (0,0)         0       (1,3,5)

bbupool schedule task average overhead:0.545000 k cycles,count = 137efc
bbupool generate task average overhead:0.929000 k cycles,count = eff38

Bbupool: Core ID         Core Usage
         ----------      -----------
         core1             26
         core2             25
         core3             26
         core4             25
         core5             25
         core6             25
```

§

3.0 Enhanced BBU Pool Framework

3.1 Introduction

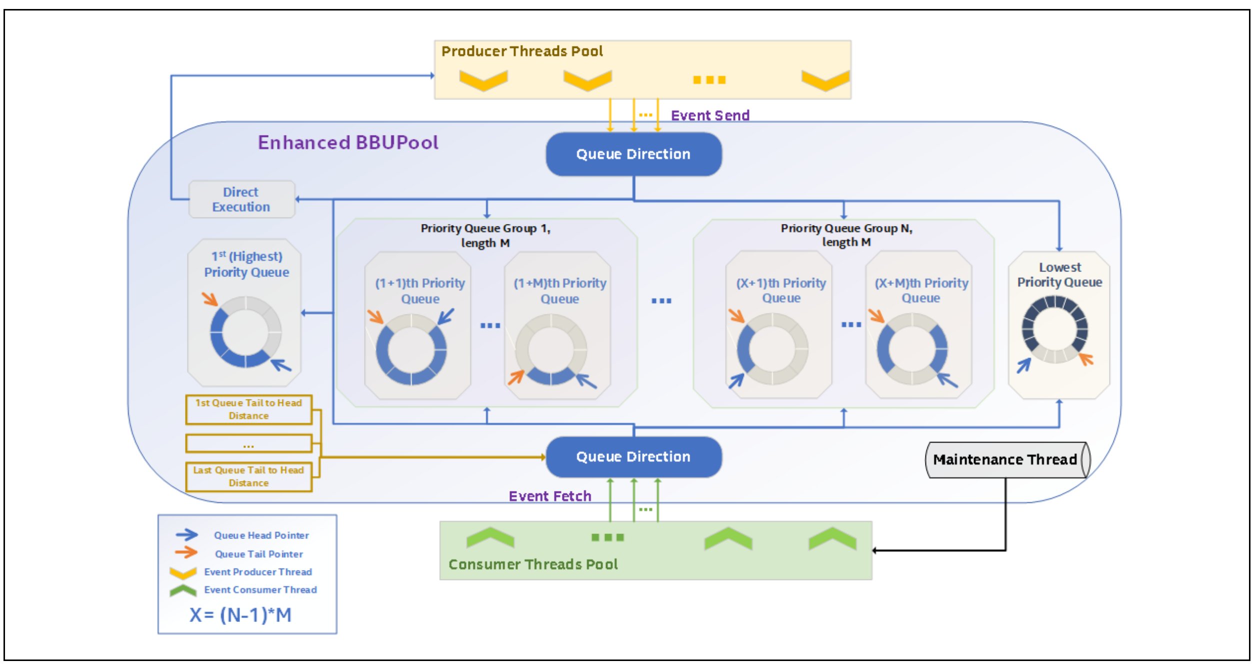
Enhanced BBU Pool (eBBUPool) is a built-in library that addresses the performance, flexibility, and scalability of multi-thread real-time traffic. Much of the real-time traffic running on IA is FlexRAN workload that has following characteristics:

1. Running on Linux operation system
2. Has restricted processing timing budget
3. Has variable workload size
4. Needs flexible parallelization on multiple cores
5. Requires priority control
6. Has core affinity

By considering all these requirements for scalability (for example, running on dual-socket system and virtualization), a generic event scheduler is designed to focus automatic event dispatching and execution. Most of the workload related controls are kept and maintained by application part, and the whole scheduler only runs on light overhead.

3.2 System Architecture

Figure 23. Enhanced BBU Pool Architecture



The whole scheduler is a built-in library, which means it runs as a part to user application. Several entities construct a complete scheduler: Event, Priority Queue, Consumer Thread, Producer Thread, and Maintenance Thread.

The combination of above entities forms a framework level event scheduler.
Various processing pipelines can be constructed by using this framework. There is no limitation on workload type or timing, and the event priority can be adjusted in run-time. The application example refers to Section 3.6 Application Usage Example.

3.3 Event

An event usually carries the data to process by CPU, and it can be data-less as well. All workload and its corresponding processing can be organized and/or chunked to event(s). An event has below attributes:

1. One execution function. Whichever core gets this event will execute this function.
2. Execution function parameters. Both input and output.
3. Core affinity. Decide which core(s) are eligible to get this event.
4. Event priority. Higher priority event is expected to execute earlier than lower priority events which are in waiting list. Apart from the normal priority-based scheduling, a special priority is designed to let the event enqueue thread directly process the event, not go through the scheduler.
5. Event alive time. Sometimes, an event needs to be abandoned if the alive time expires before execution.
6. Event identification. Help application to perform management and measurement.
7. Event status. Indication of the valid status of this event. Will not execute an invalid event. The event is the minimal element that managed by the scheduler.

3.4 Priority Queue

The events scheduling and dispatching are achieved by enabling a priority queue pool. In the pool, there are multiple priority queues. Each of the queue is an individual ring, each element of the ring is an event descriptor. The length of the ring is configurable.

An event descriptor has below attributes:

1. Element Status. This atomic indicator tracks the writing, cleaning, and waiting status of the element in the ring buffer, to ensure multiple-thread operation safe.
2. Event Pointer: Points to an event if this element has been written, otherwise keeps NULL.

A priority queue has below attributes:

1. Priority. The priority of a queue decides the searching order within the pool. A lower number means higher priority. Priority 0 means highest priority. This number is also used as queue indication.
2. Queue Group Context. If more than 2 priority queues are created, the queue groups can be organized. One group uses one context number to indicate. Each group has same number of priority queues. The priority 0 queue and priority lowest queue are not included in the queue groups. The priorities of the queues can be adjusted dynamically by the context number. This indicator helps application do customized priority scheduling.
3. Queue Depth. The ring length of each priority queue
4. Ring Head Pointer. Every event dequeuing will go to the tail of the target priority queue. It is handled by atomic operations to ensure thread safe.
5. Ring Tail Pointer. Every event enqueueing will go to the tail of the targeted priority queue. It is handled by atomic operations to ensure thread safe.
6. Tail to Head Distance. An indicator to show is this ring queue full or not. If this number equals to 0, means the ring buffer is full.

3.5 Thread

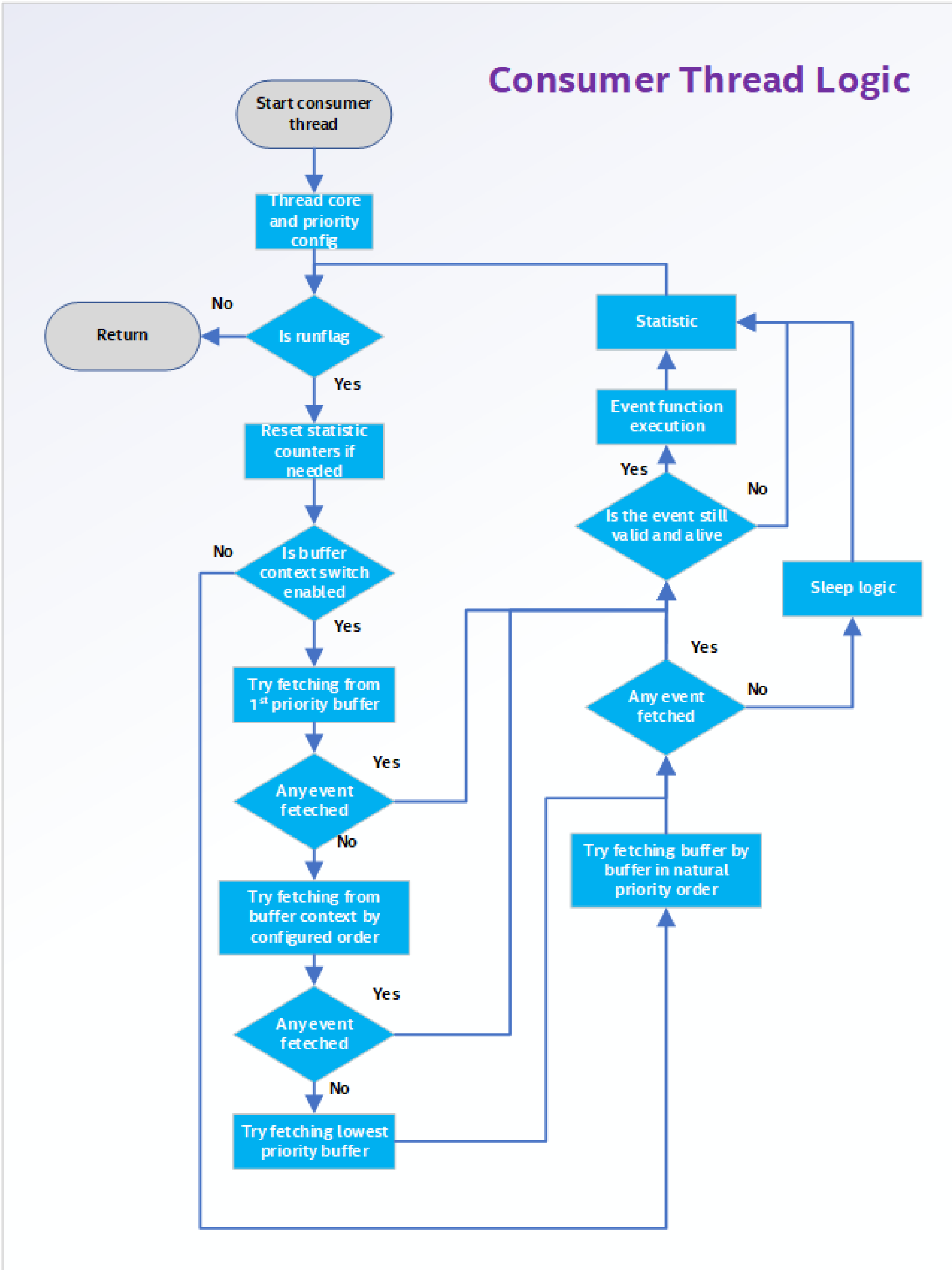
3.5.1 Consumer Thread(s)

They are the real-time thread(s) which are created to process the events in the priority queues. Each of the logic core will create one consumer thread and set thread scheduling policy to FIFO. The consumer threads can be added or removed in run time. Only consumer threads can dequeue events. The consumer threads are running in polling mode thus the event dispatch is acting as event fetch operation, no dedicated thread to do event dispatch, it is a distributed event scheduler. In each of the fetching operation consumer threads need to set two parameters:

1. Own Core Mask. Only when the core mask matches the event core affinity, the consumer thread is eligibly fetch that event. This core mask is set statistically when create one consumer thread.
2. Priority Queue Search Window. Consumer thread not always search from highest priority queue to lowest one. It can be controlled to realize more flexible and efficient event searching. It can work together with the queue group context switch. Once the search start context number is switched, the original queue priority order is changed.

The logic of consumer thread is:

Figure 24. Consumer Thread Logic

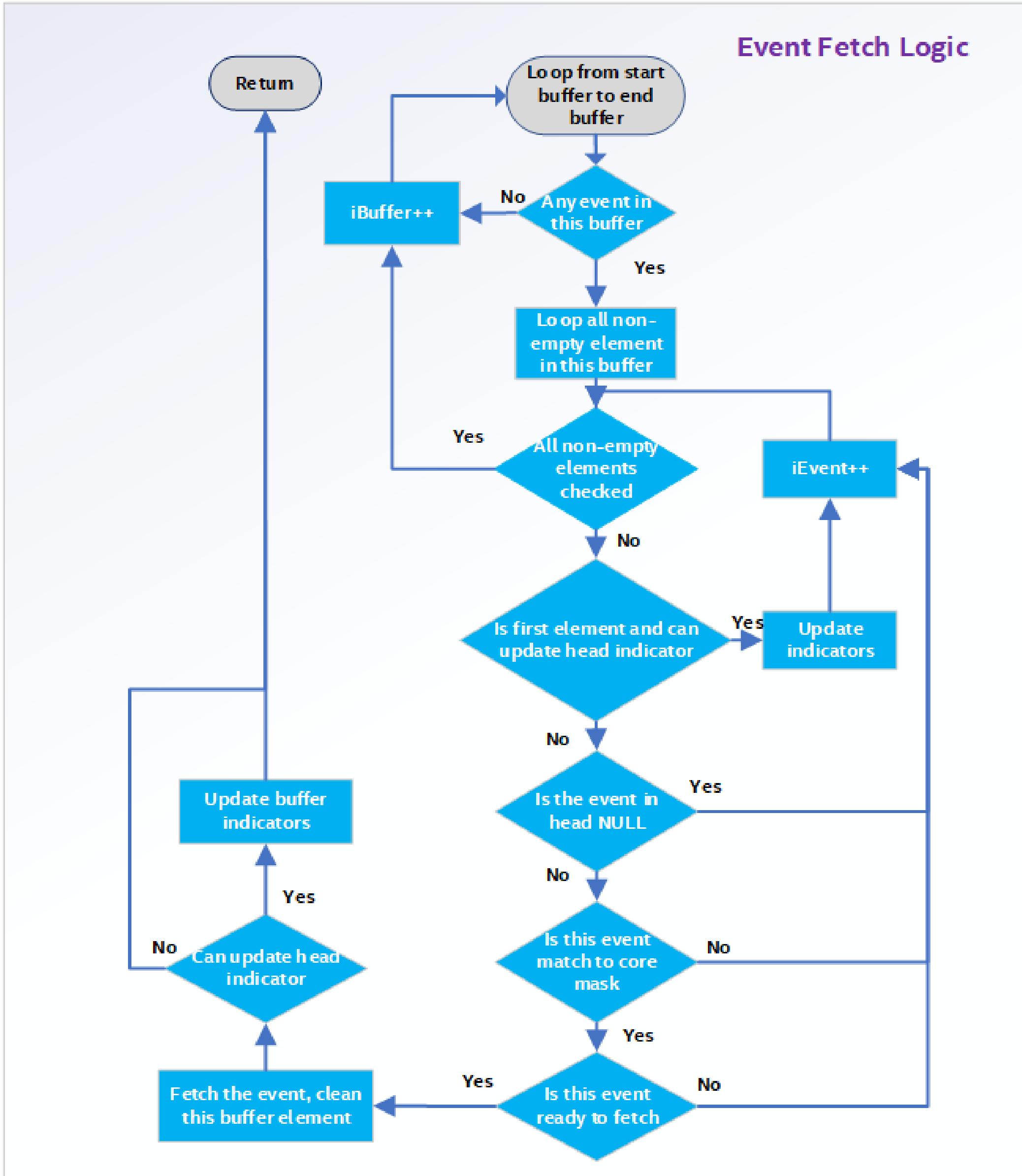


The sleep logic is:

- If the sleep is not enabled, bypass this step.
- If the sleep is enabled, check the fetch counter. If continuous 10 times failure in event fetch, will go to usleep(10). Will recount the failure time until a successful fetch.

The event fetch logic is:

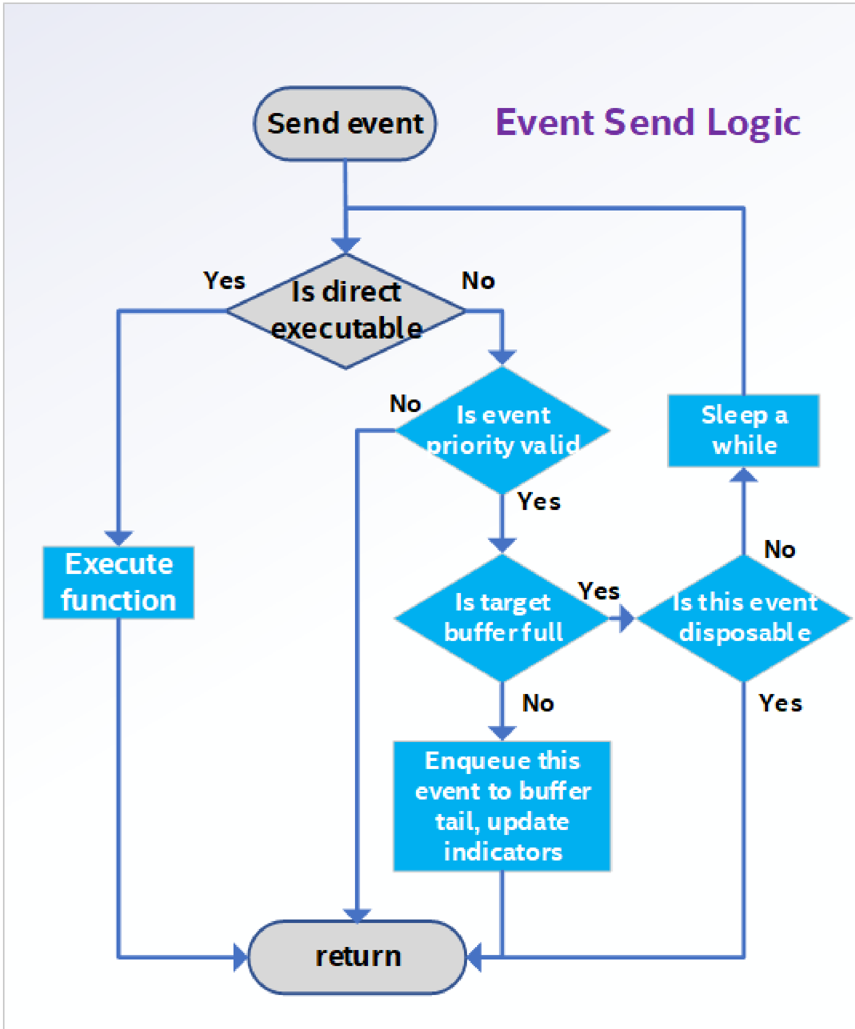
Figure 25. Consumer Thread Event Fetch Logic



3.5.2 Producer Thread(s)

They can be consumer threads or other threads. Once it gets the scheduler handler, it can enqueue events.

Figure 26. Producer Thread Event Send Logic



3.5.3 Maintenance Thread

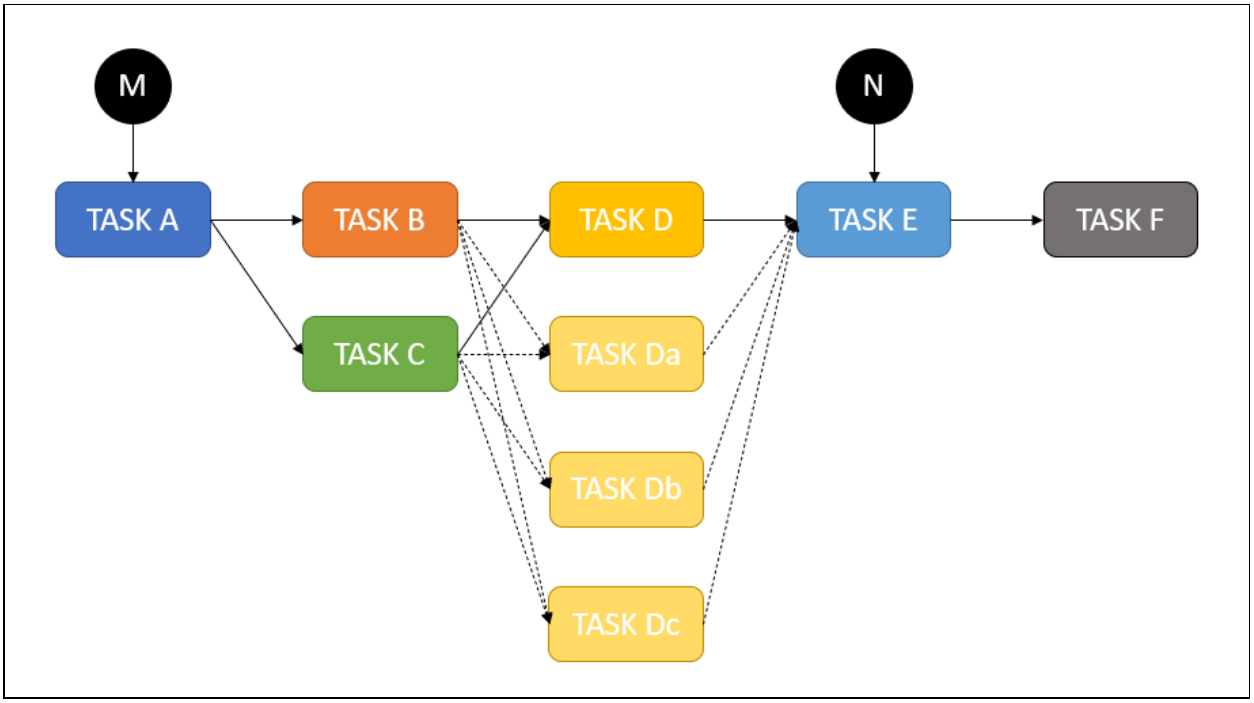
It is the non-real-time thread to control the consumer threads and print the statistics. It checks the consumer threads configuration periodically and adds/removes consumer threads dynamically. It monitors the running status flag of the whole scheduler and shutdowns it if the flag is set. It also prints the consumer core utilization periodically.

3.6 Application Usage Introduction

An application may need to handle internal and external event. The internal events are generated by application and are most likely some software processing jobs; the external events are generally some triggers such as timing or indication. To better distinguish them and make them comparable with BBUPool, in below example we will call internal event as task, and external event as event.

Let us consider a set of tasks A-F which have task and event dependencies as shown in figure below:

Figure 27. Event Chain Example

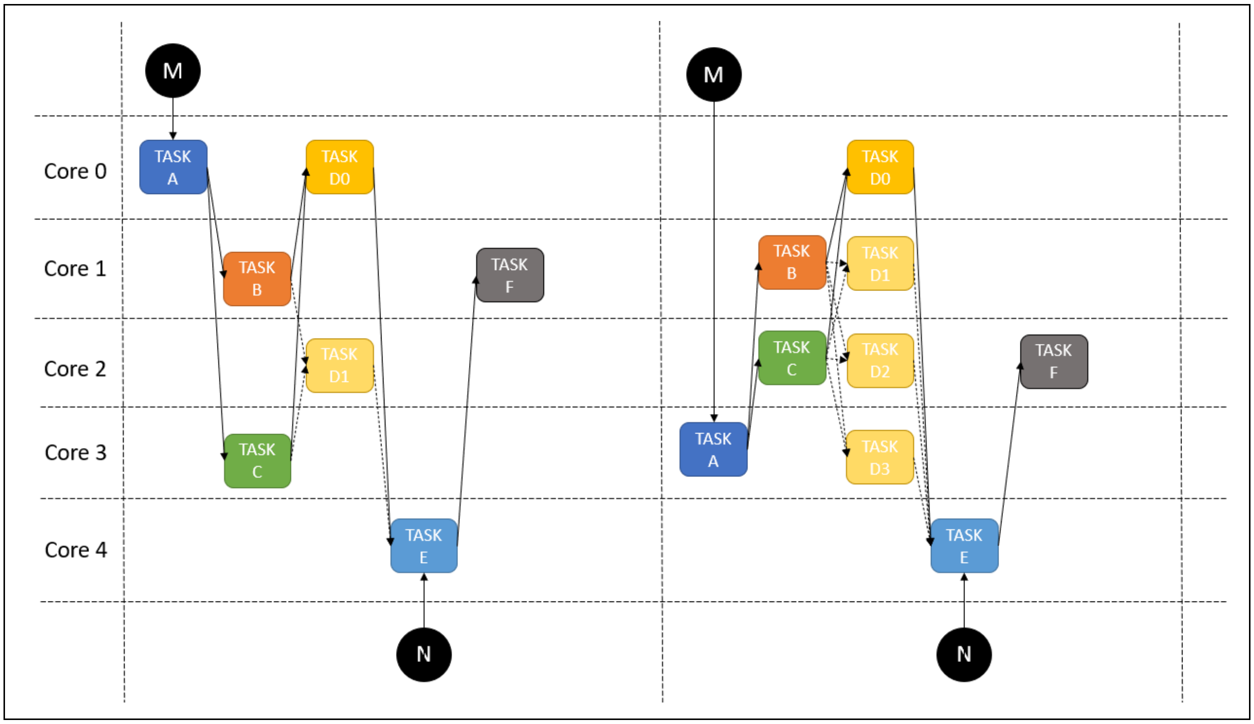


- Task D may dynamically be divided into 4 tasks that can run in parallel based on the TTI's definition. So, for some TTIs, only 1 task is needed. Some TTIs may need 2, and some others maybe 3 or 4. Based on this:
  - Task B and C will have to unlock more than 1 D task
  - And one or more D tasks will block Task E
- There are 2 Events M and N that block task A and E, respectively.
  - When M occurs, it enqueues task A
  - When N occurs, it has to unblock / enqueue E based on whether one or more D tasks are completed.
- Let us also consider in this example that:
  - Tasks A-D and F can run on cores 0,1,2,3
  - Task E can run only on core 4.
  - So a core mapping of 2 TTIs may look like as shown below:
    - The First TTI has 2 D-tasks and the second TTI has 4 D-tasks
- The events M and N can be any event in the pipeline like:
  - Timer event
  - Packet Arrival event

– Offload function on PCIE device completed event

- Based on these events, a task can be unblocked in the chain.

Figure 28. Event Distribution Example



As explained in the section 3.1 Introduction, the eBBUPool is a light-weight software that only deals with scheduling tasks. Dependencies, core masks, task priority etc, needs to be dealt within the application and fed to the scheduler as part of **ebbu_pool_send_event** call. The following sections take the above example and show how one could build a pipeline with functions and make use of the eBBUPool software for scheduling tasks.

3.6.1 Function Calls

The order of functions to be called are as follows:

- **To Initialize**

1. **ebbu_pool_create_handler**: Initialize and create handler
2. **ebbu_pool_create_queues**: Create priority queues
3. **ebbu_pool_queue_ctx_set_threshold**: Setup priority queue context counter threshold
4. **ebbu_pool_queue_ctx_set**: Setup initial priority queue context number
5. **ebbu_pool_create_report *(optional)***: Initialize report
6. **ebbu_pool_consumer_set_thread_params**: Set core parameters
7. **ebbu_pool_consumer_set_thread_mask**: Set core mask (for bringing up threads)

- **During Run time**

1. **ebbu_pool_consumer_stats_update_time (optional)**: To notify scheduler of TTI boundaries (Used by scheduler for TTI usage stats)
2. **ebbu_pool_send_event**: Enqueue tasks
3. **ebbu_pool_queue_ctx_add**: Update the priority queue context counter

- **During Clean up**

1. **ebbu_pool_release_threads**: Release all the threads
2. **ebbu_pool_status_report *(optional)*:** Print scheduler statistics
3. **ebbu_pool_release_report *(optional)*:** Release statistics context
4. **ebbu_pool_release_queues**: Release priority queues
5. **ebbu_pool_release_handler**: Release handler and clean up scheduler

The detail usages can refer to eBBUPool sample code or FlexRAN L1 pipeline code.

3.6.2 Task/Event Parameters

Here is a list of all parameters that need to be filled for each task before enqueueing. Please refer to **EventSendStruct.**

- **Call Back Function** The function to execute when task is to be scheduled
- **Call Back Function Parameters** Any parameters/arguments to be passed to the function
- **Event Id** Unique Task ID (used for some internal benchmarking)
- **Sent Time** Time stamp (using rdtsc) from application when it is going to enqueue the task
- **Alive Time** How long to keep this task in queue before disposing it. Set this number to 10000000 to bypass
- **Event Status** Set this to EBBUPOOL_EVENT_VALID
- **Event Num** Each send operation can enqueue multiple tasks together. In this case programmer also needs to fill above values for each task that is enqueued.
- **Dispose Flag** Set this to EBBUPOOL_NON_DISPOSABLE in case you do not want scheduler to drop tasks
- **Core Affinity** This is a __mm256 variable with 256 bits showing which cores can be used to run this task
- **Task Priority** Priority of the task used to put a queue
- **Task Priority Context** In case priority contexts are enabled, which context to put this task to

Based on these values, the scheduler places the task in an appropriate queue which consumer threads pick up and start executing.

3.6.3 Consumer Thread Core Utilization Statistics

The scheduler internally maintains core usage statistics by logging.

- **Used time**: Useful application work is running
- **Scheduler time**: Scheduler related work is running
- **Idle time**: When sleep flag is enabled, this is time which is returned to OS so some other threads could be scheduled at this time

There are APIs provided to get the **Used Time** for each core as a % of overall time. Please refer to **ebbu_pool_consumer_get_usage_stats**.

The consumer threads also maintain TTI to TTI core usage statistics. The min, avg, and max values in each TTI are stored. This is especially useful for TDD kind of FlexRAN scenarios where in some TTIs cores are occupied 100% of the time, and some others are close to 0%. So, the average kind of skews the utilization and clearly peak loading of the cores are hidden from application programmer. For this to work, the application needs to feed in the TTI update time to the scheduler using API **ebbu_pool_consumer_stats_update_time** and these stats will be returned back using calls of **ebbu_pool_consumer_get_usage_stats**.

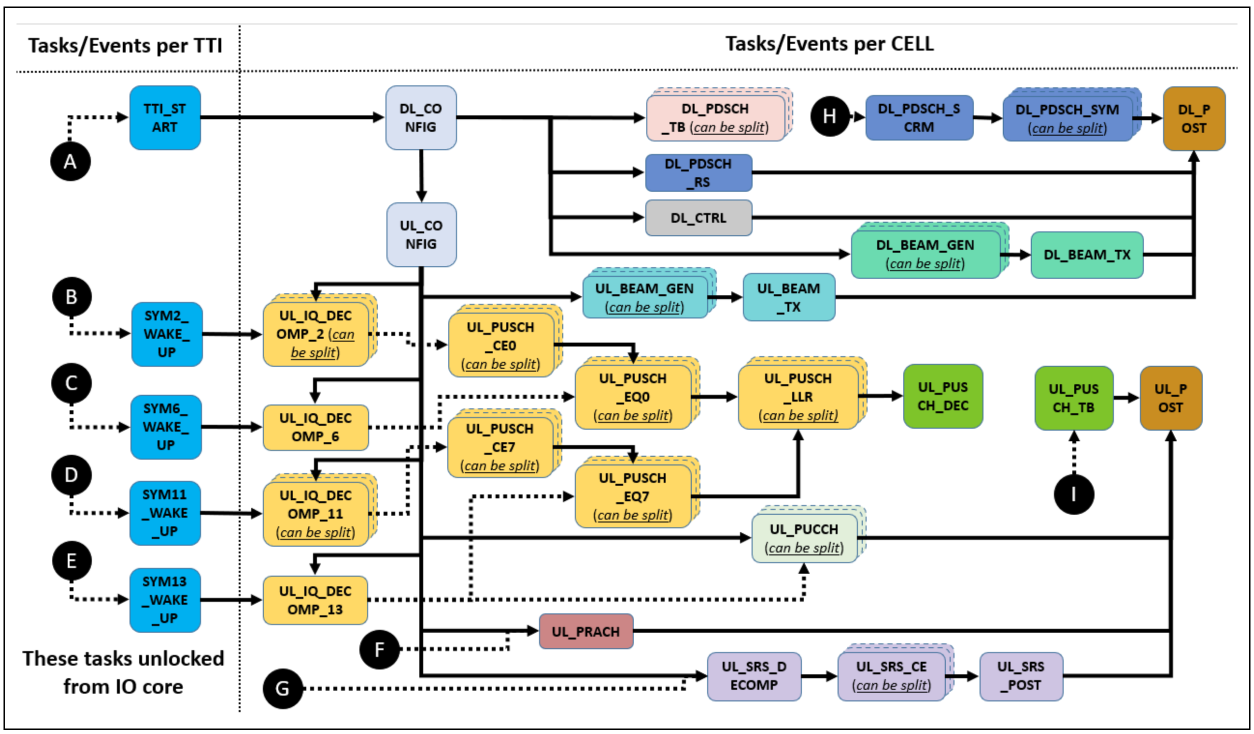
3.7 Test and Performance

3.7.1 Test System

The following figure shows how the test application of eBBUPool construct the processing pipeline.

- There are tasks that are created per TTI:
  - These are typically created from IO core or polling cores. The idea is that this core is very **latency sensitive** and it is not a good idea to unlock multiple tasks from this single core.
  - Based on some timing events, a single task is unlocked and run on the consumer cores. These are given highest priority so they can run first over other tasks scheduled.
  - The consumer cores then create tasks for all cells that have been configured.
- There are tasks that are created per Cell:
  - These are launched from the Consumer cores themselves and are duplicated per cell.

Figure 29. eBBUPool Example Event Processing Pipeline



There are events A-I that launch tasks within the scheduler framework:

- **A**: TTI boundary timing expiry event
- **B**: Sym 2 received from Radio event. This can unlock IQ Decompression Task and PUSCH Channel estimation for first half slot
- **C**: Sym 6 received from Radio event. This can unlock IQ Decompression Task and PUSCH Equalizer for first half slot
- **D**: Sym 11 received from Radio event. This can unlock IQ Decompression Task and PUSCH Channel estimation for second half slot

- **E**: Sym 14 received from Radio event. This can unlock IQ Decompression Task and PUSCH Equalizer for second half slot.
  - This also unlcoks PUCCH processing
  - Even SRS processing can be started for non-massive MIMO scenarios as we dont get separate packets for SRS in standard MIMO cases
- **F**: PRACH IQ samples received from Radio event. This unlocks PRACH task
- **G**: SRS IQ samples received from Radio event. This unlocks SRS decompression task
  - This event happens only for Massive MIMO scenarios. For standard MIMO cases, this event is not there
- **H**: This event is when DL FEC polling returns from BBDEV
- **I**: This event is when UL FEC polling returns from BBDEV

An important distinction with eBBUPool scheduler and BBUPool scheduler is that it is now possible to have tasks being blocked by other tasks and also by events. This brings some more flexibility in building pipeline.

3.7.2 Test Cases

The tests consist of functional test and performance test.

In functional test we need to ensure all functions provided by eBBUPool working well. The functions have:

1. Multiple event chain: Support single event chain and multiple event chains in same scheduler
2. Various event chain: Support various event chains in same scheduler
3. Time-varied event chain: Support different event chains which can be changed in run-time
4. Priority queue management: Support dynamic priority queue context management

Table 4. Functional Case Definition of eBBUPool

No	Cases	Multiple Event Chain	Various Event Chain	Time-varied Event Chain	Priority Queue Management
0	ebbu_pool_cfg_mixed_event_num.xml	Y	Y	N	N
1	ebbu_pool_cfg_mixed_frame_format.xml	Y	Y	N	N
2	ebbu_pool_cfg_mixed_tti.xml	N	N	Y	N
3	ebbu_pool_cfg_multi_cell.xml	Y	N	N	N
4	ebbu_pool_cfg_multi_queue_context.xml	N	N	N	Y
5	ebbu_pool_cfg_mixed_all.xml	Y	Y	Y	Y

The performance test focus on the multi-core scaling performance with certain number of events:

Table 5. Performance Case Definition of eBBUPool

No	Cases	Events Num per TTI	Logical Core Num
0	ebbu_pool_cfg_extreme_parellel_2core.xml	1000	2

1	ebbu_pool_cfg_extreme_parellel_6core.xml	1000	6
2	ebbu_pool_cfg_extreme_parellel_10core.xml	1000	10
3	ebbu_pool_cfg_extreme_parellel_16core.xml	1000	16
4	ebbu_pool_cfg_extreme_parellel_20core.xml	1000	20
5	ebbu_pool_cfg_extreme_parellel_26core.xml	1000	26

Some other features are tested in FlexRAN pipeline cases, such as:

- Event to core affinity
- Event split
- Consumer thread management

3.7.3 Test Performance

The server config is as below:

Table 6. eBBUPool Performance Test Server Config

Server Config	
CPU Model	Xeon(R) Gold 6248
Logical Cores	HT on, 0~39, single socket
Linux Kernel	3.10.0-1127.19.1.rt56.1116.el7.x86_64

When running the test, there are screen prints to show the running core utilization statistics. The internal core utilization describes the overhead of the scheduler, which is as less as better.

Figure 30. eBBUPool Performance Test Print

```
[10] consumer thread(s) running:
Core Index:   2    3    4    5    6   22   23   24   25   26   Avg
Total Util %:33   33   33   33   33   33   33   33   33   33   32.78
Exter Util %:32   32   32   32   32   32   32   32   32   32   31.74
Inter Util %: 1    1    1    1    1    1    1    1    1    1   1.04
```

For the performance test, the statistics below were gathered for each profiling case:

Table 7. eBBUPool Performance Profiling Result

Case	Logical core number	Average internal core utilization per core (%)
ebbu_pool_cfg_extreme_parellel_2core.xml	2	0.7
ebbu_pool_cfg_extreme_parellel_6core.xml	6	0.9

ebbu_pool_cfg_extreme_parellel_10core.xml	10	1.1
ebbu_pool_cfg_extreme_parellel_16core.xml	16	1.5
ebbu_pool_cfg_extreme_parellel_20core.xml	20	4.2
ebbu_pool_cfg_extreme_parellel_26core.xml	26	8.2

Besides the print, the mlog and buffer status dump can be generated after each case running. More automatic performance analysis development is on-going.

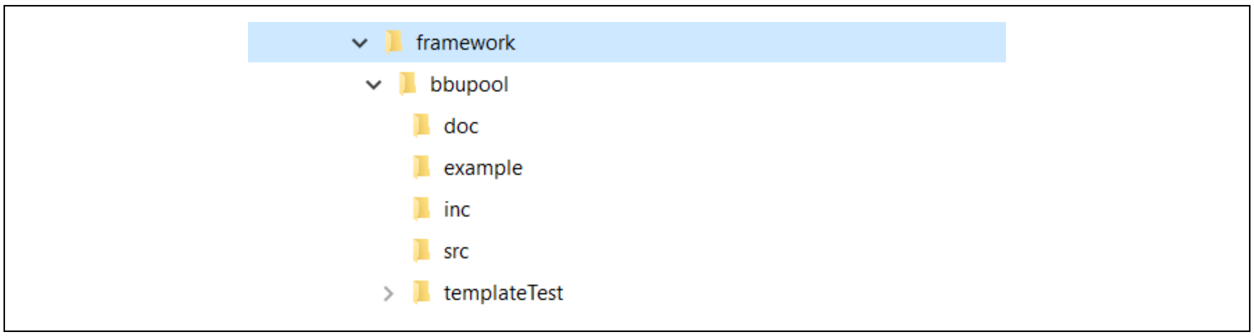
4.0 How to Build and Run Framework

This chapter describes the BBU pooling framework folder structure, system requirements, how to build and run BBU pooling task.

4.1 Overall Folders

Under the root folder of Framework, there is a folder: `bbupool`, as shown in Figure 31.

Figure 31. BBU Pooling Framework Folder Structure

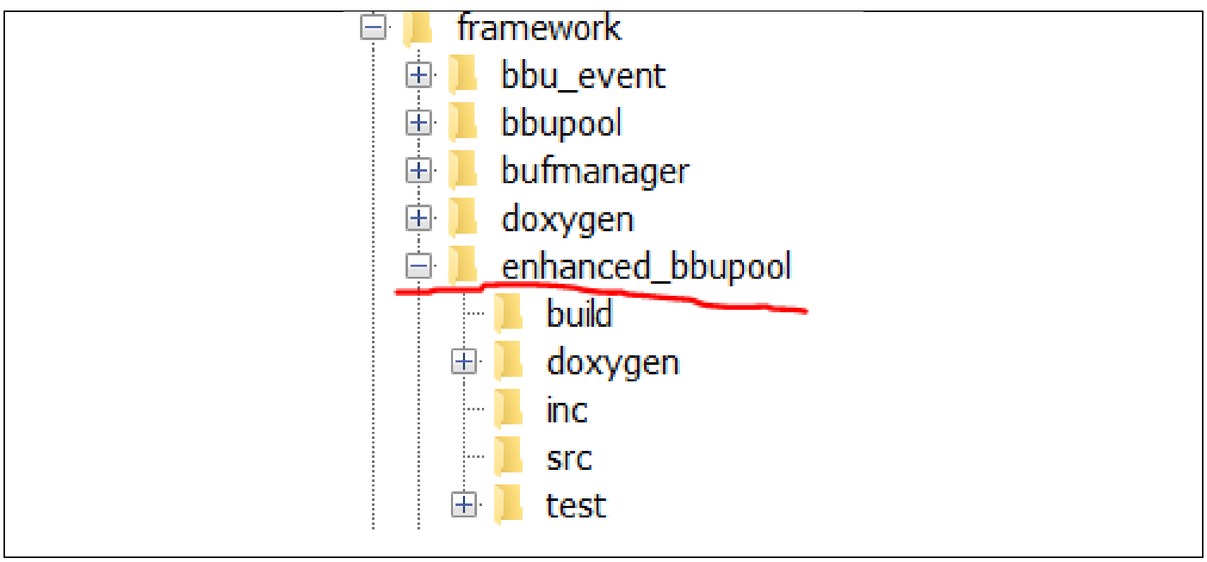


In these folders:

- inc contains header files
- src contains source code
- Template test of `bbu_pool` contains test code, `bbu_pool's` test folder is not used any longer

Under the root folder of Framework, there is a folder: `enhanced_bbupool`, as shown in the following figure:

Figure 32. Enhanced BBU Pool Folder Structure



In these folders:

- inc contains header files
- src contains source code
- build contains build scripts
- doxygen contains the doxygen document
- test contains the test scripts and cases

4.2 BBU Pooling

This chapter describes the system requirements of the BBU pooling and how to compile and run it. The system requirements include hardware, OS, and compiler requirements.

4.2.1 System Requirements

This chapter lists which hardware, OS, and compiler have been verified.

4.2.1.1 Hardware Requirements

There are no specific CPU requirements to run the FlexRAN Framework, but it has been verified on the following processors:

- Intel® Xeon® Processor D-1500 Product Family
- Intel® Xeon® Processor E5-26xx Product Family
- Intel® Xeon® Processor D-Family
- Intel® Xeon® Processor Scalable Family.
- Intel® Xeon® Platinum 8160

4.2.1.2 OS Requirements

The BBU Pooling framework has been verified on Wind River* OVP6, Wind River* Tis3/4/5, and CentOS 7.

4.2.1.3 Compiler Requirements

Intel® C++ Compiler (ICC) v19.0.3.206 was verified and recommended.

4.2.2 Compile and Run

This chapter contains compile directory and command, also describes how to run the test code in detail.

4.2.2.1 Compile Directory

The BBU Pooling library is located in the directory:

```
framework\bbupool
```

BBU Pooling sample code is located in the directory:

```
framework\bbupool\templateTest
```

4.2.2.2 Compile Command

Used to create the BBU Pooling library `libbbupool.a` in the folder `framework/bbupool`.

`Framework/bbupool: make clean;make`

As far as the above compile command, the macro ST is 0 by default, and the statistics output is disabled. If you want to enable the statistics output, please use the following compile command:

`Framework/bbupool: make clean;make ST=1`

4.2.2.3 Run Test Code

Build a BBU Pooling sample code.

Nightly build: `Framework/bbupool/templateTest: sh build.sh`

Normal build: `Framework/bbupool/templateTest: make clean;make all`

The typical build is a long-time test, and the test runs forever unless you terminate it by manual commands, such as "Ctrl+Z".

Nightly build is a short time test. Its duration is decided by `nightlybuildlen` in the case configuration file. When the duration is due, the case terminates automatically.

There are four types of case, and they are "l1000"," l125"," h1000" and "h125". For the prefix "l" or "h" means the "light task load" or "heavy task load". For the "1000" or "125" means the 1000µs or 125µs as 1 TTI.

To run one case, enter:

```
Framework/bbupool/templateTest:
./bbupool_templateTest case/file.xml
```

For example, run the case `l125_case7_6cell_6core_2queue_noAff.xml`:

```
./bbupool_templateTest case/l125_case7_6cell_6core_2queue_noAff.xml
```

In nightly build mode, we also can run all cases, enter:

```
Framework/bbupool/templateTest: sh run.sh
After one case is finished, its result file is created to store the statistics output.
For example, result_l125_case7_6cell_6core_2queue_noAff.txt.
```

Every case has its own result file. At the end of the result file, there is a pass or fail indicator. If it is "Verdict: PASSED", the test passed; otherwise it failed.

4.2.3 Use Example Code

Build and run the example code from the `framework/bbupool/example` folder.

To build example code, enter:

```
Framework/bbupool/example: make clean;make
```

To run example code, enter:

```
Framework/bbupool/example: ./bbupool_example
```

4.3 Enhanced BBU Pool

This chapter describes the system requirements of the eBBUPool and how to compile and run it. The system requirements include hardware, OS, and compiler requirements.

4.3.1 System Requirements

This chapter lists which hardware, OS, and compiler have been verified.

4.3.1.1 Hardware Requirements

There are no specific CPU requirements to run the FlexRAN Framework, but it has been verified on the following processors:

- Intel® Xeon® Processor D-Family
- Intel® Xeon® Processor Scalable Family.

4.3.1.2 OS Requirements

The eBBUPool framework has been verified on CentOS 7.

4.3.1.3 Compiler Requirements

Intel® C++ Compiler (ICC) v19.0.3.206 was verified and recommended.

4.3.2 Compile and Run

This chapter contains compile directory and command, also describes how to run the test code in detail.

4.3.2.1 Compile Directory

The eBBUPool library is located in the directory:

```
framework/enhanced_bbupool/build
```

eBBUPool sample code is located in the directory:

```
framework/enhanced_bbupool/test/
```

4.3.2.2 Compile Command

Used to create the eBBUPool `libebbupool.a` in the folder `framework/enhanced_bbupool/build`

```
framework/enhanced_bbupool/build: sh build.sh
```

4.3.2.3 Run Test Code

Build a eBBUPool sample code.

```
framework/enhanced_bbupool/test/: sh build.sh
```

To run one case, can use the script:

```
framework/enhanced_bbupool/test/: sh run.sh [basic|functional|profiling|all]
```

```
Or directly run the test application:
```

```
framework/enhanced_bbupool/test/: ./ebbupool_test [num of tti] cfgfile [cfg file name]
```

§